Webová aplikace pro sdílení souborů

Marek Olšák

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Marek Olšák**
Osobní číslo:	**A20744**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Webová aplikace pro sdílení souborů**
Téma práce anglicky:	**Web Application for File Sharing**

Zásady pro vypracování

1. Vypracujte literární rešerši na zadané téma.
2. V rámci literární rešerše analyzujte podobná existující řešení.
3. Navrhněte webovou aplikaci pro sdílení souborů, přičemž se zaměřte na řešení nezávislé na službách třetích stran a vyhovující open-source licenčním podmínkám.
4. Vytvořte aplikaci dle návrhu.
5. Věnujte pozornost zabezpečení aplikace.
6. Otestujte funkčnost webové aplikace.
7. Implementaci a testování vhodně popište.

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

Seznam doporučené literatury:

1. LAURENT, Arnaud. *The design of everyday APIs*. Shelter Island (New York): Manning Publications, 2019. ISBN 978--1617295102.
2. LOCK, Andrew. *ASP.NET Core in Action*. Annotated Edition. Second Edition. Shelter Island (New York): Manning Publications, 2021. ISBN 978-1617298301.
3. SAKIMURA, Nat; BRADLEY, John; DE MEDEIROS, Breno; MORTIMORE, Chuck a B. JONES, Michael. *OpenID Connect Core 1.0 incorporating errata set 1*. 2014. OpenID Foundation, 2014. Dostupné z: https://openid.net/specs/openid-connect-core-1_0.html. [cit. 2023-11-07].
4. SPASOJEVIC, Marinko a PECANAC, Vladimir. *ULTIMATE ASP.NET CORE WEB API: From Zero to Six-Figure Backend Developer*. Online. Second Edition. CodeMaze, 2021. Dostupné z: https://code-maze.com/ultimate-aspnetcore-webapi-second-edition/. [cit. 2023-11-07].
5. BAIER, Dominick [@JetBrains]. *Securing SPAs and Blazor Applications using the BFF (Backend for Frontend) Pattern*. Online. 2022. Dostupné z: YouTube, https://www.youtube.com/watch?v=DdNssiaIY_Q. [cit. 2023-11-07].

Vedoucí bakalářské práce: **Ing. Tomáš Vogeltanz, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

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

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

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

Ve Zlíně dne 5. ledna 2024

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
13. 5. 2024 Marek Olšák, v. r.

ABSTRAKT

Bakalářská práce se věnuje vývoji prototypu aplikace pro sdílení dat v síti. Prototyp aplikace klade důraz na její zabezpečení, přenositelnost a open-source licencování. Hlavním přínosem je modulární a udržitelná architektura systému, která se zakládá na vícevrstvém modelu, nebo implementace jednostránkové aplikace (SPA) ve frameworku Blazor WASM. Současně systém nabízí robustní ochranu proti neoprávněnému přístupu implementací vlastního řešení poskytovatele identity založeného na standardech OpenID Connect a OAuth 2.0. Open-source licencování zvyšuje dostupnost a podporuje jeho další komunitní rozvoj. Cílem je nejen představit praktické řešení pro sdílení souborů, ale také přispět k odborné diskusi a podpořit další výzkum v oblasti softwarového inženýrství.

Klíčová slova:

WebAssembly, Blazor, REST, API, OAuth 2.0, OpenID, .NET, Postgresql, mikro-služby, sdílení dat

ABSTRACT

The bachelor thesis is devoted to the development of a prototype application for data sharing in the network. The prototype application emphasizes on its security, portability and open-source licensing. The major contribution is a modular and sustainable system architecture based on a multi-layered model or the implementation of a single page application (SPA) in the Blazor WASM framework. At the same time, the system offers robust protection against unauthorized access by implementing a custom identity provider solution based on OpenID Connect and OAuth 2.0 standards. Open-source licensing increases availability and encourages further community development. The intention is not only to present a practical solution for file sharing, but also to contribute to the professional discussion and promote further research in the field of software engineering.

Keywords:

WebAssembly, Blazor, REST, API, OAuth 2.0, OpenID, .NET, Postgresql, micro-services, data sharing

Chtěl bych vyjádřit hluboké poděkování všem, kteří mě během mého studia a přípravy bakalářské práce podporovali. Zvláštní poděkování patří mé přítelkyni, která byla mojí oporou, rodičům za jejich nekonečnou podporu a trpělivost a vedoucímu mé práce, jehož rady a odborný náhled byly klíčové pro dokončení této práce. Vaše zpětná vazba byla pro mě během celého procesu nesmírně důležitá.

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

OBSAH

ÚVOD **10**
I TEORETICKÁ ČÁST **12**
1 TECHNOLOGIE A NÁSTROJE **13**
1.1 .NET 13
1.2 C# 14
1.3 Blazor WASM 15
1.3.1 MudBlazor 16
1.4 Hangfire 17
1.5 OpenIddict 17
1.6 NSwag 19
1.6.1 NSwagStudio 19
1.7 Entity Framework Core 20
2 PARADIGMATA **21**
2.1 Dependency inversion 21
2.2 Dependency injection 21
2.3 CQRS 22
2.4 MVC 22
2.5 REST 23
2.6 BFF 24
2.7 Cibulovitý model 25
3 ANALÝZA APLIKACÍ PRO SDÍLENÍ DAT **26**
3.1 Sledované vlastnosti a hodnotící kritéria 26
3.2 OwnCloud 26
3.2.1 Hodnocení 27
3.3 NextCloud 27
3.3.1 Hodnocení 28
3.4 Srovnání a vyhodnocení rešerše 28
II PRAKTICKÁ ČÁST **29**
4 PŘEDSTAVENÍ APLIKACE A POŽADAVKY **30**
4.1 Funkční požadavky 31
4.2 Nefunkční požadavky 33
5 ARCHITEKTURA NAVRŽENÉHO SYSTÉMU **34**
5.1 Struktura projektu 35
5.2 Struktura databáze 36
6 IMPLEMENTACE POSKYTOVATELE IDENTITY **38**
6.1 Připojení k databázi 38
6.2 Migrace změn 42
6.3 Persistence hesel 42

6.3.1 Mapování funkcí dynamicky linkované knihovny 43
6.3.2 Nastavení služby pro hashovaní hesel 43
6.3.3 Hashovaní hesel 44
6.3.4 Ověření shody hesel 45
6.4 Autorizace a autentizace 46
6.5 Protokoly OAuth 2.0 a OpenID Connect 49
6.5.1 Koncový bod pro autorizaci klienta 51
6.5.2 Koncový bod pro udělení svolení klientovi pro získání informací uživatele 52
6.5.3 Koncový bod pro zamítnutí svolení klientovi pro získání informací uživatele 52
6.5.4 Koncový bod pro výměnu tokenů 52
6.5.5 Mapování informací uživatele na tokeny 54
6.5.6 Koncový bod pro odhlášení uživatele 55
6.5.7 Koncový bod pro získání dodatečných informací o uživateli 55
7 REST API 56
7.1 Introspekce přístupového tokenu 56
7.2 Autorizační filtr 57
7.3 Základní modely dotazů a odpovědí 60
7.4 Autorizační pipeline v MediatR 60
7.5 Swagger a generování dokumentace 61
7.6 Pravidelné úlohy v pozadí 63
7.6.1 Úloha v pozadí pro mazání souborů 65
7.7 Příkaz pro nahrání souboru 67
7.7.1 Model dotazu a odpovědi 67
7.7.2 Validátor příkazu 69
7.7.3 Obsluha příkazu 70
7.7.4 Koncový bod 72
7.7.5 Dokumentace 72
8 IMPLEMENTACE KLIENTA 74
8.1 Proces zavedení klienta ve webovém prohlížeči 74
8.2 Integrace poskytovatele identity 75
8.2.1 Nastavení komunikace mezi klientem a poskytovatelem identity 76
8.2.2 Koncový bod pro přihlášení uživatele 77
8.2.3 Koncový bod pro odhlášení uživatele 78
8.2.4 Aktualizace stavu přihlášení uživatele 78
8.2.5 Ověření stavu přihlášení během směrování 80
8.2.6 Ověření oprávnění uživatele během vykreslování komponenty 80
8.2.7 Komponenta pro odhlášení uživatele 80
8.3 Integrace REST API 81

8.3.1 Mapování koncových bodů 81
8.3.2 Integrační vrstva a generování HTTP klienta 82
8.3.3 Služby pro komunikaci s REST API 85
8.4 Lokalizace 86
8.4.1 Výběrovník preferovaného jazyka 87
8.4.2 Lokalizace textace 88
8.5 Uživatelské rozhraní 88
8.5.1 Téma aplikace, typografie a piktogramy 89
8.5.2 Rozložení pro mobilní a desktopové aplikace 89
8.5.3 Notifikace 91
8.5.4 Dialogy 91
8.6 Přehled složek a souborů 92
8.6.1 Zobrazení složek a souborů 93
8.6.2 Dialog pro vytváření složek a nahrávání souborů 93
8.6.3 Drag and drop komponenta pro nahrání souborů 94
9 TESTOVACÍ SADY A PŘÍPADY 96
9.1 Stub kontextu databáze 96
9.2 Stubs pro dotazy a příkazy v MediatR a další služby 97
9.3 Testovací případ vytvoření složky 98
ZÁVĚR 100
SEZNAM POUŽITÉ LITERATURY 101
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 106
SEZNAM OBRÁZKŮ 109
SEZNAM TABULEK 110
SEZNAM PŘÍLOH 110

ÚVOD

Vývoj bezpečných, udržitelných a dostupných webových aplikací představuje zásadní výzvu pro vývojáře a výzkumné pracovníky. Tato bakalářská práce předkládá prototyp aplikace pro sdílení dat v síti. Řešení klade důraz na její zabezpečení, využití moderních technologií jako WebAssembly a .NET nebo zásady open-source licencování. Díky těmto prioritám je vytvořený systém nejen udržitelný a modulární, ale také robustní v oblasti ochrany uživatelských dat a souborů.

Hlavní technologií použité během vývoje je Blazor WebAssembly, která umožňuje implementovat jednostránkové webové aplikace (SPA). Aplikace se spouští v prohlížeči uživatele bez nutnosti opakovaných dotazů na server. Tento přístup poskytuje vysoký výkon a interaktivitu aplikace, zároveň však klade velké bezpečnostní nároky, zejména v oblasti autentizace a autorizace uživatelů. V reakci na tyto výzvy bylo implementováno vlastní řešení poskytovatele identity založeného na standardech OpenID Connect a OAuth 2.0. Poskytovatel identity používá mechanismy jako je Authorization Code Flow s Proof Key for Code Exchange (PKCE), které přispívají k vysokému stupni ochrany proti neoprávněnému přístupu ve srovnání s tradičním zabezpečením existujících open-source řešení, která obměňují JWT tokeny.

Dalším přínosem práce je architektura založená na mikro službách, která společně s využitím návrhového vzoru Backend for Frontend (BFF) umožňuje persistenci tokenů pro autentizaci i autorizaci uživatele na serveru. Návrhový vzor zároveň zjednodušuje distribuci statických souborů aplikace a přidává další vrstvu zabezpečení aplikace. Existující open-source řešení se zakládají na monolitické architektuře ve srovnání s prototypem aplikace, ve které poskytovatel identity a webové API jsou součástí jedné služby systému. Tento přístup odebírá na možnostech pro nahrazení poskytovatele identity nebo neumožňuje rozložení zátěže mezi jednotlivé služby systému, čímž klesá jeho udržitelnost. Open-source licencování prototypu aplikace zvyšuje jeho dostupnost a vytváří podklady pro další komunitní rozvoj.

Aplikace a její vývoj tedy nejen představují praktické využití WebAssembly nebo Blazor WASM pro implementaci moderní a bezpečné webové aplikace, ale také nabízí srovnání s existujícími open-source řešeními. Analýza existujících řešení umožňuje lépe pochopit výhody a omezení jednotlivých přístupů nebo technologií oproti systémovým požadavkům.

Cílem této práce je nejen představit funkční prototyp webové aplikace pro sdílení dat, ale také prozkoumat i zdokumentovat proces vývoje a rozhodnutí, které vedly k výběru

technologií a paradigmat pro návrh systému. Přístup práce k vývoji webové aplikace přispívá k odborné diskusi a poskytuje užitečné poznatky pro další výzkum a praxi v oblasti softwarového inženýrství.

I. TEORETICKÁ ČÁST

1 TECHNOLOGIE A NÁSTROJE

První kapitola bakalářské práce se věnuje technologiím a nástrojům, které používá prototyp aplikace během jejího vývoje. Klíčové technologie zahrnují framework .NET, který umožňuje vývoj webových aplikací s frameworkem ASP.NET Core nebo ADO.NET pro připojení ke zdroji dat, načítání, zpracování a aktualizaci dat. Součástí frameworku ASP.NET Core je framework Blazor pro vývoj interaktivních jednostránkových aplikací. Komponentní knihovna MudBlazor slouží pro návrh i implementaci uživatelského rozhraní. Požadavky na zabezpečení aplikace jsou řešeny frameworkem OpenIddict, kde úlohy v pozadí plánuje i spravuje framework Hangfire.

1.1 .NET

Platforma .NET je open-source přenositelné prostředí pro vývoj multiplatformních aplikací. Běžně se využívá při vývoji uživatelských rozhraní, webových aplikací i aplikačních rozhraní nebo vývoji počítačových her, zejména prostřednictvím nástroje Unity. [1]

.NET umožňuje překlad zdrojových souborů pro programovací jazyky jako C#, F#, Visual Basic nebo C++ do mezi úrovňového programovacího jazyka (IL). IL je nezávislý na procesoru a za běhu aplikace překládán na strojový kód před jeho spuštěním. Aplikace používají nejběžněji překladač JIT (Just-In-Time) nebo AoT (Ahead-Of-Time) pro rychlé spuštění. [2][3]

Volba technologie .NET byla motivována její zralostí ve vývoji webových aplikací i API, který umožňuje implementaci jednostránkové i vícestránkové aplikace, gRPC, GraphQL a REST API nebo implementaci standardů jako OpenAPI, OAuth 2.0 nebo OpenID Connect.

Ekosystém .NET zahrnuje několik verzí s rozdílnými charakteristikami:

.NET Framework	Framework se vyznačuje omezenou přenositelností, protože vytváří závislosti na knihovnách operačního systému Windows. Vývoj platformy je nyní zaměřen primárně na bezpečnostní aktualizace.
.NET Core	Technologie, která nabízí přenositelnou a modulární alternativu k .NET Framework, která umožňuje nasazení aplikací na různé platformy použitím kompaktního prostředí běhu Mono.
.NET	Nástupce .NET Core, který netrpí na nedostatky předchozích verzí tím, že zavedl jednotné typové rozhraní od verze 5 a výše, která

	umožnila kompatibilitu a přenositelnost knihoven mezi verzemi platformy.
.NET Standard	Standard, který definoval soubor API, které byly kompatibilní mezi různými verzemi .NET, a které umožňují sdílení knihoven. Vývoj tohoto standardu byl ukončen kvůli sjednocení typového rozhraní ve verzi .NET 5 a výše. [4][5]

1.2 C#

Open-source silně typovaný a objektově orientovaný programovací jazyk s automatizovanou správou paměti. Jazyk ve svých počátcích byl inspirován syntaxí programovacího jazyka Java a během jeho překladu dochází k převodu do IL .NET. K přednostem tohoto jazyka se řadí typová reflexe, která za běhu aplikace umožňuje přistoupit k metadatům datového typu objektu. Metadata obsahují informace jako datový typ, implementovaná rozhraní, vlastnosti, atributy a dále. Řešení implementované v tomto jazyce ovšem mohou narazit na omezení, které se pojí s automatizovanou správou paměti. Alokovanou paměť na haldě obsluhuje mechanismus Garbage Collector (GC) a během vyšší zátěže systému jako při načítání dat z databáze mohou vznikat objekty s krátkou životností, které nemusí být dostatečně rychle uvolněny. Akumulace objektů, na které se neodkazují jiné objekty, může vést k výraznému zpomalení aplikace. Popisovaná situace běžně nastává při manipulaci s většími sestavami dat, kde objektově orientované mapovače jako Entity Framework Core pro každou položku vytváří dodatečné objekty, které uchovávají informace o provedených změnách a dále. Alternativní původce tohoto jevu může být řetězení výrazů LINQ, při kterém může docházet k zapouzdření položek a vzniku nových objektů, které reprezentují výsledek předchozí funkce. Reálným příkladem lze uvést fórum StackOverflow, které se potýkalo s přetížením GC. StackOverflow pro snížení zátěže implementovalo vlastní řešení mikro objektově relačního mapovače. Mikro ORM, přezívaný Dapper, ve srovnání s Entity Framework Core nepodporuje převod LINQ výrazů na SQL dotazy, nesleduje změny modelů entit nebo neumožňuje vývoj databáze přístupem Code-First.

Programovací jazyk podporuje asynchronní programování, generiky a dědičnost, přičemž umožňuje dědění pouze z jedné rodičovské třídy. Součástí jsou i další syntaktické prvky moderních programovacích jazyků jako lambda výrazy nebo jmenné prostory deklarované v hlavičce souboru. C# je kompatibilní i s externími nespravovanými knihovnami a službami, které nespadají pod správu paměti GC a vyžadují manuální uvolnění alokované

paměti, například kontext databáze a její připojení k databázi nebo čtení obsahu souboru z úložiště zařízení. [6][7][8][9]

1.3 Blazor WASM

Framework Blazor umožňuje vývoj interaktivních jednostránkových webových aplikací v .NET, HTML a CSS. Motivací jeho vzniku bylo umožnění spuštění .NET aplikací ve webovém prohlížeči s příchodem WebAssembly. Mezi nejrozšířenější modely nasazení řadíme Blazor Server, který je spouštěn na straně serveru a využívá protokol WebSocket k aktualizaci segmentů virtualizovaného DOM, nebo Blazor WASM, který je předmětem této práce.

Model nasazení Blazor WASM je spouštěn ve webovém prohlížeči uživatele. Běh .NET aplikací ve webové prohlížeči umožňuje technologie WebAssembly. WebAssembly je binární instrukční formát pro virtuální počítače založené na zásobníku, který umožňuje běh aplikací vyvinutých v technologiích jako je .NET s téměř nativní rychlostí ve webovém prohlížeči. [10, 11]

.NET aplikace spouštěné ve webovém prohlížeči závisí na prostředí běhu a závislostech, tj. zdrojových souborech. První spuštění doprovází stažení zavaděče a souboru blazor.boot.json, který obsahuje seznam závislostí včetně prostředí běhu. Ke stažení závislosti je nezbytný server s podporou distribuce statických souborů. Prostředí běhu Mono přeložené do WebAssembly překládá zdrojové soubory na vyžádání JIT (Just-In-Time) do .NET IL, a před jeho spuštěním do WebAssembly. Od verze .NET 6 a vyšší je možné zdrojové soubory přeložit do IL při vystavení aplikace překladačem AoT (Ahead-of-Time), který lze povolit přidáním direktivy do projektového souboru. [10, 12]

Jednostránkové webové aplikace (SPA) se odlišují od vícestránkových navigací a aktualizací DOM. Uživatelské rozhraní těchto aplikací se skládá ze znovu použitelných fragmentů DOM, které přezíváme komponenty. Komponenty umožňují aktualizovat částí DOM namísto jeho natažení ze serveru pro každý požadavek, a tak minimalizovat počet vykreslení. Použití komponent předkládá koncept absence stránek v architektuře webových aplikací. Architektura založená na komponentách pohlíží na stránky jako na komponenty, kde platí striktní hierarchie. Nejméně závislá komponenta se běžně nachází na aplikační vrstvě a v případě Blazor aplikací slouží k obsluze a směrování požadavků na dceřiné komponenty, které představují „stránky“ webové aplikace. [10, 13]

Framework se zakládá na modulu Razor, který umožňuje zabudovat .NET výrazy do HTML dokumentů. Protože Blazor vychází z tohoto frameworku, technologie je kompatibilní s knihovnami, stavebními bloky a pomocnou rutinou značek frameworku MVC pro .NET pro směrování požadavků, vytváření jednosměrných i obousměrných vazeb modelů, vlastností a dále.

Blazor WASM se spouští v sandboxu, který přispívá k integritě a bezpečnosti zdrojového kódu. Sandbox umožňuje volání funkcí jazyka JavaScript prostřednictvím API. Framework obsahuje knihovnu služeb v .NET a funkce jazyka JavaScript pro zprostředkování vzájemné interopability. [10, 14]

Výhodami vývoje webové aplikace za využití WebAssembly, HTML, CSS a .NET ve srovnání s tradičním přístupem je rychlost srovnatelná s nativní, podpora offline režimu, flexibilnější distribuce zátěže systému, nižší stopa zaznamenaná v paměti nebo použití technologií, se kterými jsou vývojáři dobře obeznámeni. Nevýhodami je pomalejší zavedení aplikace ve srovnání s JavaScript a TypeScript frameworky jako React nebo Angular, omezení případně potenciálně nákladnější implementace požadavků na vývoj PWA a TWA aplikací. [10]

1.3.1 MudBlazor

Open-source knihovna Blazor komponent, která byla implementována v .NET, HTML a CSS. Současná verze .NET podporuje statické vykreslení komponent, které nezaručuje zachování interaktivity komponent ve webovém prohlížeči uživatele. Pro zachování interaktivity v případě statického vykreslení komponenty je zapotřebí její implementace v jazyce JavaScript namísto .NET jako komponentní knihovna Syncfusion.

Knihovna umožňuje vývoj interaktivních uživatelských rozhrání přístupem mobile-first, který klade důraz na jejich přizpůsobivost zobrazovací ploše přenositelných zařízení jako mobilní telefony nebo tablety. Předností této knihovny vyjma licenčních podmínek je přizpůsobivost komponent prostřednictvím CSS a jejich API, kde komponentní knihovny jako Radzen nebo Blazorise postrádají. Knihovna umožňuje za běhu aktualizovat nebo globálně nastavit výchozí motiv aplikace, typografii nebo pigmentaci. Komponenty podporují lokalizaci i globalizaci, umožňují integraci s webovými API nebo frameworky jako FluentValidation pro validaci formulářů. [15]

1.4 Hangfire

Framework pro plánování a spouštění úloh v pozadí s open-source licenčními podmínkami. Hangfire umožňuje integraci s objektově relačními mapovači jako Entity Framework Core pro persistenci jeho stavu a sběru analytických dat. Součástí knihoven kromě služeb a nástrojů pro životosprávu úloh je i uživatelské rozhrání, které umožňuje jejich monitorování a manuální spuštění, zobrazit záznamy aktivit a selhání spuštění nebo přehled úloh na pozadí s grafy pro sledování jejich aktivity v reálném čase.

Přínosem frameworku je usnadnění vývoje, plánování a spouštění úloh na pozadí, které minimalizuje dopad na výkonnost aplikace. Současně umožňuje separaci aplikační logiky na úlohy, které můžeme odděleně plánovat. Plánování úloh používá formát zápisu cron, který známe například z plánování spuštění skriptů v operačním systému Linux. [16]

1.5 OpenIddict

Aplikace spouštěné webovými prohlížeči nedokážou bezpečně ukládat citlivé informace jako tajemství a veřejné klíče. Pro zabezpečení přenosu citlivých informací mobilních a webových aplikací nebo zabránění škodlivé aplikaci vydávat se za klienta implementuje prototyp aplikace standard OAuth 2.0. Standard OAuth 2.0 popisuje autorizační tok Authorization Code Flow s Proof Key for Code Exchange (PKCE) pro komunikaci mezi službami a nedůvěryhodnými klienty. Proces autentizace uživatele popisuje standard OpenID Connect, kde standardy společně umožňují bezpečně komunikovat a sdílet informace mezi klienty a službami. [17]

Framework OpenIddict implementuje protokoly OAuth 2.0 i OpenID Connect. Součástí frameworku jsou knihovny pro implementaci poskytovatele identity, jeho integraci s klienty nebo databázovými systémy. Server vyžaduje připojení k databázi pro persistenci nastavení klientů, jejich autorizace pro přístup k informacím uživatele nebo životosprávě tokenů. Služby, metody a modely pro obsloužení dotazů směřované na koncové body API poskytovatele identity jsou součástí knihoven. Koncové body specifikované standardem RFC 7636 pro autorizaci klienta, výměnu tokenů, jejich introspekci a dále ovšem vyžadují pozornost vývojáře, jelikož nejsou součástí frameworku ve srovnání s proprietárními a certifikovanými řešeními jako Duende’s Identity Server. [17, 18]

Následující kroky níže popisují případ užití přihlášení uživatele v klientovi za využití poskytovatele identity:

1. Klient přesměruje nepřihlášeného uživatele na poskytovatele identity po návštěvě webové stránky nebo po iniciativě ze strany uživatele jako stisk tlačítka pro přihlášení. Před přesměrováním na poskytovatele identity vygeneruje klient kód pro ověření jeho totožnosti „code_verifier“, ze kterého odvodí výzvu „code_challenge“, kterou pošle společně s dotazem na poskytovatele identity.
2. Poskytovatel identity přijme dotaz pro autorizaci klienta, který ho přesměruje na koncový bod „/connect/authorize“. Koncový bod podporuje dle standardu http metody dotazu GET i POST.
3. Během autorizace klienta se ověří jeho totožnost, v případě opominutí nastavení, volby nepodporovaného autorizačního toku nebo typu svolení, které uděluje uživatel klientovi pro přístup k jeho informacím, a dále zamítne poskytovatel identity žádost.
4. Pokud poskytovatel identity podporuje nastavení klienta, postupuje jeho autorizace k autentizaci uživatele. Nepřihlášený uživatel bude přesměrován na stránky poskytovatele identity s formuláři pro přihlášení nebo registraci.
5. Po ověření totožnosti přesměruje poskytovatel identity uživatele zpět na koncový bod pro autorizaci klienta, kde proběhnou výše popsané kroky pro validaci klienta a stavu přihlášení uživatele. Následuje ověření udělení explicitního svolení klientovi pro přístup k informacím uživatele. Pokud uživatel neudělil explicitní svolení klientovi, bude uživateli zobrazen formulář pro jeho udělení i zamítnutí.
6. V případě udělení svolení klientovi od uživatele uloží poskytovatel identity výzvu „code_challenge“ a generuje jednorázový autorizační kód „code“ s krátkou platností, který pošle při přesměrování zpět na klienta.
7. Po obdržení autorizačního kódu „code“ provolá klient koncový bod „/connect/token” společně s kódem pro ověření jeho totožnosti „code_verifier“. Poskytovatel identity validuje uloženou výzvu „code_challenge“ oproti kódu pro ověření totožnosti klienta „code_verifier“. Poté ověří platnost autorizačního kódu, který klient uplatňuje pro vydání přístupového tokenu, tokenu identity a tokenu pro obnovení přístupového tokenu.
8. Poskytovatel identity po vydání tokenů přesměruje uživatele na adresu klienta pro dokončení přihlášení, které obnáší persistenci tokenů a mapování informací uživatele. Adresa pro přesměrování uživatele zpět na klienta po úspěšném přihlášení nebo jeho odhlášení je součástí nastavení klienta. Podle doporučení standardu by koncové body klienta pro přihlášení i odhlášení neměli být shodné s poskytovatelem identity pro minimalizaci plochy, na kterou útočník může cílit.

Ve srovnání s tradičními metodami autentizace i autorizace jako rotování JWT tokenů přináší tento přístup vyšší úroveň zabezpečení. Mezi další přednosti řadíme snadnou integraci klienta a poskytovatele identity prostřednictvím rozhraní, které popisují standardy. [17]

1.6 NSwag

Framework, který implementuje specifikaci Swagger pro generování dokumentace webového API. Swagger umožnuje testování koncových bodů prostřednictvím integrovaného klienta, který podporuje odlišené přístupy pro autentizaci i autorizaci uživatele. Přínosem jeho implementace je generování udržitelných HTTP klientů ze specifikace koncových bodů, modelů požadavků a odpovědí. Z této verze vychází novější OpenAPI specifikace, která významově klade důraz na open-source licencování nástrojů a frameworků pro generování dokumentace nebo podpůrných nástrojů pro vývoj aplikací. [19, 20]

1.6.1 NSwagStudio

Nástroj, který umožňuje generování HTTP klientů ze Swagger specifikace ve formě JSON souboru pro .NET nebo TypeScript aplikace. NSwagStudio podporuje vkládání dokumentace jako JSON nebo její natažení prostřednictvím odkazu. Výhodami nástroje je jeho parametrizace generátoru klientů, která podporuje: [20]

- Možnost zvolit verzi .NET.
- Možnost zvolit knihovnu pro serializaci dotazů a deserializaci odpovědí.
- Možnost zapouzdření odpovědí pro zpracování výjimek a HTTP stavových kódů.
- Možnost zvolit jmenné konvence použité při generování názvů metod pro provolání koncových bodů.
- Možnost zvolit datové typy pro modely dotazů a odpovědí.
- Možnost vložení aplikační logiky pro provolání koncových bodů webového API.
- Možnost nastavení závislostí generovaného HTTP klienta.

1.7 Entity Framework Core

Objektově relační mapovač pro .NET, který umožňuje připojení ke zdroji dat, načtení, zpracování a aktualizaci dat. Služba pro komunikaci s databází využívá kontext databáze, který vytváří připojení k databázi. Připojení k databázi nespadá pod životosprávu alokované paměti GC, proto bude zapotřebí zdroje manuálně uvolnit nebo vhodně zvolit scope jeho životnosti pro automatizované uvolnění kontejnerem pro vkládání závislostí. Stejně jako mikro-ORM Dapper byl postaven na frameworku ADO.NET, který umožňuje komunikovat s databází na nízké úrovni a integraci poskytovatele pro připojení k databázi jako framework Npgsql pro Postgresql v .NET. Integrace poskytovatele připojení ke zdroji dat umožní převádět výrazy na dotazy databáze (LINQ na SQL), volat jedinečné funkce a nástroje databázového systému nebo mapovat modely na konkrétní typy databáze jako vektory pro full-text vyhledávání. [21, 22]

2 PARADIGMATA

Při vývoji prototypu byly aplikovány návrhové vzory, které podporují vývoj udržitelných, modulárních a bezpečných webových aplikací.

2.1 Dependency inversion

Princip Inversion of Control (IoC) je jedním ze základních principů SOLID pro návrh a vývoj softwarových aplikací, který se zaměřuje na minimalizaci závislostí mezi vysokoúrovňovými a nízko úrovňovými komponentami. Tento princip stanovuje, že moduly nižší úrovně, například služby datové vrstvy v cibulovém modelu, by neměly být závislé na modulech vyšší úrovně jako jsou služby aplikační vrstvy. Místo toho se doporučuje, aby byly implementace komponent založeny na rozhraních, která jsou definována a umístěna v nižších vrstvách, například v nezávislé doménové vrstvě. Takový přístup podporuje udržitelnost softwarového řešení, jelikož umožňuje výměnu poskytovatelů služeb nebo jejich implementací bez negativního dopadu na ostatní části systému. [23, 24]

2.2 Dependency injection

Návrhový vzor vkládání závislostí umožňuje předávání závislostí do modulů prostřednictvím parametrů konstruktoru, metod tříd nebo pomocí metod pro nastavení hodnot privátních polí. Tento přístup deleguje správu instancí závislostí na vnější objekt, což podporuje udržitelnost, modularitu a testovatelnost aplikace. Modularita a testovatelnost jsou dále podporovány možnostmi nahrazení komponent pomocí zjednodušených verzí pro testování, známých jako "stubs" nebo "drivers".

Platforma .NET poskytuje kontejner pro vkládání závislostí, který automatizuje správu a injektáž závislostí do instancí registrovaných služeb. Kontejner podporuje tři hlavní životní cykly pro instance závislostí: [23, 24]

- Singleton – Kontejner vytvoří instanci komponenty, kterou znovu použije při řešení závislostí komponent v rámci aplikace.
- Scoped – Kontejner vytvoří instanci komponenty v rámci kontextu, kterou znovu použije v jejím rámci. Kontextem pro REST API nebo poskytovatele identity jsou řadiče, případně jeho koncové body. Model nasazení Blazor Server jako kontext požívá webovou patici, z tohoto důvodu se pro vložení nové instance komponenty používá životní cyklus transient. Ve srovnání model nasazení Blazor WebAssembly

jako kontext používá instanci aplikace a jelikož její instance obsluhuje komunikaci s jedním uživatelem, tak životní cykly scoped a singleton mají stejnou životnost.

- Transient – Kontejner vytvoří novou instanci komponenty při řešení závislostí pro každý požadavek.

2.3 CQRS

Návrhový vzor Command Query Responsibility Segregation (CQRS) se zaměřuje na rozdělení operací čtení a zápisu v aplikacích, což přispívá k vyšší organizaci a čistotě kódu. Tento princip rozlišuje mezi dotazy, které pouze čtou data a neovlivňují stav systému, a příkazy, které stav systému mění. Takové oddělení operací umožňuje efektivnější správu dat a zvyšuje rozšiřitelnost aplikace.

Dotazy se specializují na efektivní získávání dat a jejich prezentaci uživatelům bez zásahu do stávajícího stavu databáze. Tím se zajišťuje rychlý a efektivní přístup k informacím. Příkazy naopak zahrnují akce, které aktivně modifikují data v databázi, jako je například registrace uživatele nebo aktualizace hesla. Tyto akce jsou často provázeny komplexnější logikou a validací.

Implementace CQRS vede k lepší separaci zodpovědnosti v aplikaci, což vývojářům usnadňuje správu závislostí a řízení toku dat. Díky tomuto přístupu je možné snadněji rozlišit mezi částmi aplikace zodpovědnými za zpracování uživatelských požadavků a těmi, které slouží k ukládání a manipulaci s daty. To významně přispívá k modularizaci a udržitelnosti kódu.

Framework MediatR pro .NET poskytuje praktickou implementaci vzoru CQRS tím, že umožňuje vývojářům definovat dotazy a příkazy jako samostatné objekty. Díky tomu je možné snížit složitost aplikace a zvýšit její testovatelnost a opětovnou použitelnost kódu. MediatR rovněž podporuje generické požadavky a odpovědi, což přispívá k flexibilitě a snadnému rozšíření aplikace. [25]

2.4 MVC

Model-View-Controller (MVC) je architektonický vzor, který rozděluje webové aplikace do tří hlavních komponent: modelu, pohledu a řadiče. Tento vzor je využíván především v platformě .NET pro usnadnění vývoje, testování a údržby webových aplikací. Model zastupuje datovou logiku a strukturu, pohled se zaměřuje na uživatelské rozhraní a vizualizaci dat, zatímco řadič řídí interakci mezi modelem a pohledem, zpracovává uživatelské vstupy a volí odpovídající pohled pro zobrazení.

V MVC architektuře modely obsahují nejen data, ale také logiku potřebnou pro jejich zpracování, což zahrnuje načítání dat z databáze, validaci a další manipulaci. Pohledy jsou implementovány pomocí Razor syntaxe v .NET, což umožňuje snadnou integraci C# kódu do HTML a dynamickou prezentaci dat. Řadiče pak slouží jako prostředník mezi uživatelským rozhraním a datovou vrstvou, a řídí logiku aplikace.

Separací zájmů MVC usnadňuje strukturování a rozšíření aplikace. Modely mohou být testovány nezávisle na pohledech, což vede k lepší testovatelnosti jednotlivých částí. Pro zvýšení opětovné použitelnosti a usnadnění testování se běžně využívá vzor "ViewModel", který mapuje data modelu pro konkrétní potřeby pohledů.

Architektura MVC umožňuje vývoj interaktivních a dynamických webových aplikací s čistou strukturou. Separací aplikační logiky od uživatelského rozhraní přispívá k čitelnějšímu i udržitelnějšímu zdrojovému kódu. Framework ASP.NET Core MVC zjednodušuje implementaci MVC v .NET, poskytuje knihovny a nástroje pro vývoj webových aplikací a aplikačních rozhraní. Součástí tohoto frameworku je řada zabudovaných funkcí jako nástroje pro autentizaci i autorizaci uživatele, ukládání odpovědí na dotazy do vyrovnávací paměti, hostování aplikace nebo obsluhu směrování požadavků. Framework přispívá a usnadňuje vývoj webových aplikačních rozhraní a více stránkových aplikací. [24]

2.5 REST

Paradigma a nejrozšířenější architektonický vzor pro vývoj webových aplikačních rozhraní. Framework ASP.NET Core používá tento vzor jako výchozí a jeho součástí jsou nástroje a služby pro jeho implementaci. Návrhové vzory jako GraphQL nebo gRPC požadují po vývojářích poskytnutí vlastní aplikační logiky oproti REST. [23, 24, 26]

Předností vzoru je podpora verzování a dokumentace pro webové API, která popisuje koncové body, jejich vstupní a výstupní modely nebo stavové hlášky. Dokumentace je dostupná ve formátu JSON, kterou můžeme ve frameworku jako Swagger pro .NET zobrazit v podobě interaktivního uživatelského rozhraní. Parsováním dokumentace můžeme generovat http klienty a zjednodušit integraci služeb s webovým aplikačním rozhraním. [19, 20, 23, 24]

Výsledek implementace tohoto vzoru je bez stavové webového aplikačního rozhraní, které po obdržení požadavku vytvoří nový kontext. Koncové body poskytovatele identity, které znovu používají kontext přihlášeného uživatele v podobě Session Cookie, nedodržují zásady

tohoto vzoru, z tohoto důvodu se bavíme o webovém aplikační rozhrání, a nikoliv REST API. [23, 24, 26]

Prototyp klade důraz na vývoj bez stavového webového aplikačního rozhrání a jeho dokumentaci. Další rysy jako ukládání požadavků a jejich odpovědí do vyrovnávací paměti, content negotiation, rate-limiting a throttling nebudou součástí první iterace vývoje, případně HATEOAS pro generaci a začlenění odkazů na zdroje do odpovědí požadavků není plánován pro nadcházející iterace, protože jeho přínosy významově nepřispívají ke kvalitě řešení, ale naopak zavádí další abstrakce a zvyšují komplexitu řešení. [23, 24]

2.6 BFF

Architektonický vzor pro vývoj klientských webových aplikací, který se běžně využívá pro jejich integraci s dalšími službami. Tento vzor umožňuje oddělení aplikační logiky pro manipulaci a načtení zdrojů nebo načtení, uložení a aktualizaci bezpečnostně kritických informací mimo klientskou aplikaci v případě jejího hostování na stejné doméně jako aplikační rozhrání. Implementací tohoto vzoru do řešení začlení projekt webového API, které může být použito pro distribuci statických souborů klienta a sloužit jako server. [28, 29]

Níže nalezneme výhody použití tohoto návrhového vzoru:

1. Oddělení prezentace a manipulace dat od aplikační logiky. Namísto provolání jednoho a více koncových bodů aplikačního rozhrání služeb můžeme provolat jeden koncový bod API, který bude volat a mapovat jejich odpovědi na model určený k přenosu a prezentaci dat.
2. Uchování citlivých informací mimo agenta uživatele. Vzor umožňuje uložení, načtení a aktualizaci informací jako tajemství a klíče pro komunikaci se službami mimo agenta uživatele.
3. Vertikální a horizontální autorizace. Aplikační rozhrání umožňuje implementaci aplikační logiky pro ověření vlastnictví zdroje před jeho neoprávněnou manipulací nebo přístupem.
4. Hostování klientské aplikace. Aplikace, které jsou spouštěny v agentovi uživatele, běžně webové prohlížeče, potřebují server pro distribuci statických souborů pro jejich zavedení. Použití vzoru umožňuje distribuce těchto souborů na vlastní infrastruktuře.
5. Zabezpečení relace. Hostování klientské aplikace na stejné doméně jako její webové API umožňuje životosprávu vzájemné relace, která bude ve webovém prohlížeči

uchována v Session Cookie. Session Cookie oproti autentizaci a autorizaci uživatele prostřednictvím tokenů nabízí větší odolnost proti běžným útokům. Příkladem uvedu odolnost proti XSS, protože Cookies nelze získat ani manipulovat za využití programovacího jazyka JavaScript.

Nevýhodami tohoto návrhového vzoru mohou být:

1. Složitost řešení. Implementací vzoru začlením do řešení projekt webového aplikačního rozhrání, které bude zapotřebí integrovat s klientskou aplikací.
2. Zvětšení plochy vystavené pro útoky. Použití API pro vytvoření a životosprávu relace prostřednictvím Session Cookie umožňuje útočníkovi provést útoky jako Session Hijacking, na které autentizace a autorizace za využití tokenů není zranitelná.

2.7 Cibulovitý model

Architektonický model, který separuje aplikační logiku mezi vrstvy řešení projektu. Platí zde nezávislost vnitřní vrstvy na vnější, kde jádro představuje nezávislou vrstvu aplikace a prezenční vrstva nejvíce závislou. Použití tohoto vzoru umožňuje implementaci a využití dalších paradigmat jako dependency injection nebo dependency inversion. Výhodami modelu je zřetelná separace zájmů, která významně přispívá k čitelnosti a udržitelnosti zdrojového kódu. [27]

3 ANALÝZA APLIKACÍ PRO SDÍLENÍ DAT

K nejrozšířenějším open-source aplikacím, jejichž hlavním účelem je sdílení dat v síti, řadíme OwnCloud a NextCloud. Tyto aplikace používají webové aplikační rozhrání v kombinaci s desktopovými a mobilními klienty, kteří společně s API budou předmětem hodnocení.

3.1 Sledované vlastnosti a hodnotící kritéria

Prototyp aplikace se zakládá na moderních technologií jako WebAssembly a její využití v .NET. Hlavním cílem práce je dodat řešení pro sdílení dat v síti s důrazem na jeho zabezpečení a udržitelnost. Provedení rešerše přináší náhled na přednosti této aplikace a její výhodami ve srovnání s existujícími open-source řešeními, která zároveň umožní získat nové poznatky, které přispějí ke kvalitnějšímu řešení a upřesnění funkčních i nefunkčních požadavků. Předmětem hodnocení jsou následující body:

- Možnost integrace aplikace s externími aplikacemi a službami.
- Zabezpečení aplikace.
- Uživatelské rozhrání a jeho přehlednost.
- Spolehlivost a přenositelnost aplikace.
- Portfolio funkcí aplikace a jejich příspěvek ke zlepšení uživatelského zážitku.

Splnění kritérií hodnotím bodově v rozmezí 1 až 10, kde vyšší hodnocení znamená kvalitněji zpracovaný požadavek. Každému kritériu je přidělena váha, která se zohledňuje při výpočtu celkového hodnocení. Výpočet celkového hodnocení je součet násobků bodového hodnocení kritéria s jeho váhou.

$$celkové\ hodnocení = \sum_{index=1}^{počet\ kritérií} (kritérium_{index}) \times (váha\ kritéria_{index})$$

3.2 OwnCloud

Open-source řešení pro sdílení a synchronizaci dat v síti. Aplikace podporuje týmovou kolaboraci při úpravě dokumentů a jejich sdílení zveřejněním, odkazem nebo omezením výčtem uživatelů. Architektura systému obsahuje API, klientské aplikace a databázový systém. Aplikační rozhrání, dále server, a klientské aplikace nejsou součástí jednoho řešení. Server můžeme hostovat na vlastní infrastruktuře nebo využít zpoplatněné cloudové služby SaaS, kde se mezi podporované platformy nasazení neřadí Windows.

Klientské aplikace jsou dostupné jako mobilní aplikace pro Android i iOS nebo desktopové alternativy pro Linux a Windows. Aplikace umožňují rozšíření funkcionality v podobě modulů, které lze získat z oficiální webové stránky. Na většinu rozšíření se ovšem vztahují komerční licenční podmínky. Rozšíření zahrnují kalendář, emailovou schránku nebo integraci s externími službami jako Microsoft Office 365. Zabezpečení komunikace mezi serverem a klienty je implementováno tradičním způsobem za využití výměny a obnovy JWT tokenů. [30, 31]

3.2.1 Hodnocení

Server implementuje specifikaci Swagger, a tudíž umožňuje generování udržitelných HTTP klientů. Aplikační rozhrání umožňuje spravovat uživatelské účty a data jako soubory nebo složky uživatele. Mezi jeho přednosti řadíme rozsáhlé portfolio koncových bodů, které je dobře zdokumentované, otestované a robustní. [31]

Klientské aplikace nabízí přehledné uživatelské rozhrání, které lze rozšířit o modul kalendáře, emailovou schránku a další pro kolaboraci s ostatními uživateli. Aplikace jsou dostupné na nejrozšířenějších platformách jako Android, iOS, Windows, Linux a macOS. [30]

Ačkoliv systém nabízí veškerou požadovanou funkcionalitu pro sdílení dat v síti doplněnou o možnost kolaborace s ostatními uživateli, tradiční autentizace i autorizace uživatele prostřednictvím tokenů není vhodná pro zabezpečení mobilních nebo webových aplikací. Desktopový klient komunikuje Machine-to-Machine, z tohoto důvodu můžeme bezpečně ukládat citlivé informace jako veřejné klíče, které tradiční zabezpečení tokeny používají. Soukromé klíče se bezpečně ukládají na straně serveru a veřejné klíče na straně klienta. Tento přístup není vhodný pro zabezpečení mobilních a webových aplikací, protože se jedná o veřejné klienty. Veřejní klienti se běžně spouští například ve webovém prohlížeči uživatele namísto na straně serveru, proto citlivé informace jako veřejné klíče není možné bezpečně ukládat. Tradiční ověření tokeny ve srovnání s doporučeným protokolem pro zabezpečení jednostránkových webových a mobilních aplikací OAuth 2.0 Authorization Code Flow s PKCE neověřuje totožnost klienta nebo neřeší problém s ukládáním veřejného klíče. [17, 30, 31]

3.3 NextCloud

Server se zakládá na starší verzi repositáře OwnCloud. Projekt byl založen stejnou osobou jako v případě projektu OwnCloud, kde NextCloud klade větší důraz na komunitu namísto vývoje proprietárních rozšíření. Klientské aplikace podporují stejné majoritní platformy jako

OwnCloud a podporují rozšíření, na které nejsou vázány licenční omezení. Portfolio rozšíření je rozsáhlejší ve srovnání s jeho protějškem za pomocí podpory komunity, kde podpora týmová kolaborace je klíčová. Rozšíření zahrnují kalendáře, emailové schránky, konverzace, videohovory a podobně. Zatímco OwnCloud cílí na větší podniky, NextCloud se zaměřuje i na domácnosti a menší firmy. [32, 33]

3.3.1 Hodnocení

Jelikož systém požívá stejný server jako OwnCloud, sdílí výhody i nevýhody. Uživatelská rozhrání obou řešení jsou téměř identická s méně výraznými odlišnostmi jako piktogramy a typografie. Mezi jeho přednosti oproti konkurenčnímu řešení se řadí rozsáhlost portfolia rozšíření a přívětivější licenční podmínky pro jejich použití. [32]

3.4 Srovnání a vyhodnocení rešerše

Ve srovnání s existujícími open-source řešeními nabízí prototyp minimální požadovanou funkcionalitu pro sdílení dat v síti bez možnosti kolaborace. Tohle může být přínosné i postradatelné podle případů užití. Prototyp aplikace poskytuje webovou aplikaci jako klienta pro interakci se serverem, kterou OwnCloud i NextCloud v současnosti neimplementují, ale alternativně nabízí nativní aplikace pro odlišné platformy. Klíčovým přínosem prototypu aplikace je zabezpečení klientských aplikací, které nabízí vyšší stupeň ochrany. Tento přístup umožňuje ochránit citlivé údaje uživatele před neoprávněným přístupem a poskytuje vysokou úroveň zabezpečení pro veřejné i soukromé klienty. [30, 32]

Níže uvedená tabulka poskytuje souhrn bodového hodnocení obou aplikací, které se může lišit podle individuálních potřeb uživatelů:

Tabulka 1: Vyhodnocení srovnání open-source aplikací pro sdílení dat.

Název aplikace	OwnCloud	NextCloud
Integrace	8	8
Zabezpečení	7	7
Uživatelské rozhrání	7	8
Přenositelnost	8	6
Funkcionalita	8	10
Součet	**7.6**	**7.8**

II. PRAKTICKÁ ČÁST

4 PŘEDSTAVENÍ APLIKACE A POŽADAVKY

Uploadify, prototyp webové aplikace, prezentuje standard WebAssembly a jeho realizaci frameworkem Blazor WebAssembly. Jedná se o službu určenou pro ukládání dokumentů s možností online přístupu. Hlavním cílem tohoto projektu je rozšířit povědomí o technologii Blazor WebAssembly, poukázat na její potenciální využití, výhody a nevýhody ve srovnání s tradičními přístupy k vývoji webových aplikací. Prototyp nabízí uživatelům nahrávat soubory ze zařízení na úložiště buď výběrem nebo jejich přetažením bez omezení na kapacitě úložiště. Uživatelé mohou dále soubory organizovat do složek a sdílet je s ostatními.

Aplikace je navržena tak, aby byla přínosná nejen pro uživatele, kteří hledají efektivní a ekonomické řešení pro sdílení souborů na vlastní infrastruktuře bez nutnosti platit poplatky nebo se podřizovat licenčním omezením, ale také pro ty, kteří hledají alternativní nebo doplňující řešení pro své stávající či budoucí projekty. Je vhodná i pro ty, kteří chtějí rozšířit své dovednosti v oblasti vývoje webových aplikací.

Důraz je kladen na dodržování standardů a osvědčených postupů, což zajišťuje, že prototyp bude atraktivní i pro uživatele hledající snadnou a spolehlivou cestu k integraci klienta s aplikačními rozhraními. Prototyp umožňuje integraci s autentizačním a autorizačním rozhraním poskytovatele identity, který využívá kombinaci protokolů OpenID Connect a OAuth 2.0 Authorization Code Flow s PKCE, které zaručují bezpečnou komunikaci nezávisle na typu zařízení a klienta. Rozhraní pro manipulaci s dokumenty je v souladu se specifikací OpenAPI, která podporuje snadné generování udržitelných HTTP klientů.

Zahrnutý klient v prototypu aplikace adoptuje přístup „mobile-first“ pro vývoj uživatelského rozhraní, který zajišťuje přívětivé a přehledné zobrazení informací na mobilních zařízeních. Tento přístup je přenosný i na jiná přenosná zařízení nebo stolní počítače.

Analýza a sběr požadavků umožňuje vhodně zvolit technologie pro řešení projektu a definovat funkcionality, které budou jeho součástí. Výsledkem je výčet funkčních a nefunkčních požadavků, které kladou kvalitativní i kvantitativní omezení na funkcionality a udávají jejich závažnost.

4.1 Funkční požadavky

Z přehledu níže vyčteme funkcionální požadavky, které byly odvozeny z analýzy existujících řešení a představení aplikace. Výsledek analýzy přispěl k výběru technologií a nástrojů, které byly použity během vývoje prototypu aplikace. Funkční požadavky požadují dodání funkcionality systémem a kategorizují se podle závažnosti.

Tabulka 2: Funkční požadavky prototypu aplikace.

ID	Požadavek	Popis	Závažnost
FR001	Podpora protokolů pro bezpečnou autentizaci a autorizaci	Poskytovatel identity musí umožnit klientovi autentizaci a autorizaci uživatele protokoly OAuth 2.0 Authorization Code Flow s PKCE a OpenID Connect.	Vysoká
FR002	Registrace uživatele	Poskytovatel identity musí umožnit uživateli registrovat se prostřednictvím formuláře, který obsahuje povinné pole: „emailová adresa, heslo a jeho opakování, rodné a křestní jméno, uživatelské jméno a telefonní číslo.“	Vysoká
FR003	Přihlášení uživatele	Poskytovatel identity musí umožnit uživateli přihlásit se uživatelským jménem a heslem.	Vysoká
FR004	Autorizace klienta	Poskytovatel identity musí vyžadovat udělení explicitního svolení klientovi k přístupu k citlivým údajům uživatele.	Vysoká
FR005	Audit přihlášení uživatele	Poskytovatel identity musí zaznamenat poslední datum přihlášení uživatele.	Střední
FR006	Bezpečné ukládání hesel	Poskytovatel identity musí před zaznamenání uživatelského hesla v databázi provést jeho hašování prostorově složitým algoritmem.	Střední
FR007	Výměna informací o uživateli	Poskytovatel identity musí umožnit klientovi vydání tokenu přístupu, token identity a volitelně tokenu pro obnovení tokenu přístupu.	Vysoká
FR008	Ověření platnosti a původu tokenu	Poskytovatel identity musí umožnit introspekci přístupového tokenu klientovi pro jeho ověření.	Vysoká
FR009	Zpřístupnění informací o uživateli	Poskytovatel identity musí umožnit klientovi volitelně získat identitu uživatele prostřednictvím koncového bodu.	Střední
FR010	Externí odhlášení uživatele	Poskytovatel identity musí umožnit uživateli odhlásit se.	Vysoká
FR011	Dokumentace REST API	REST API pro sdílení a manipulaci se zdroji musí implementovat specifikaci OpenAPI pro její zdokumentovaní a usnadnění integrace ostatních služeb.	Vysoká
FR012	Vertikální a horizontální autorizace	REST API pro sdílení a manipulaci se zdroji musí ověřit identitu a oprávnění uživatele před zpracováním požadavku.	Vysoká

FR013	Model odpovědi REST API na dotaz klienta	REST API pro sdílení a manipulaci se zdroji musí do odpovědi zahrnout stavový kód a chybu, která v případě úspěšného požadavku není nastavena, obsahuje uživatelsky přívětivou hlášku nebo kód překladu a kolekci chyb vázané na model, který byl jeho součástí.	Střední
FR014	Implementace soft-delete politiky pro mazání záznamů	REST API pro sdílení a manipulaci se zdroji musí při mazání záznamu implementovat politiku soft-delete. Označené záznamy budou odebrány z databáze po uplynutí lhůty 7 dní od data poslední změny.	Vysoká
FR015	Popis koncových bodů REST API	REST API pro sdílení a manipulaci se zdroji musí v dokumentaci vhodně popsat jednotlivé koncové body a modely, uvést příklady jejich požadavků a odpovědí.	Nízká
FR016	Autorizace uživatele při provolání koncových bodů REST API	REST API pro sdílení a manipulaci se zdroji musí při ověření oprávnění a identity uživatele využít přístupový token, pro který musí provést introspekci prostřednictvím poskytovatele identity.	Vysoká
FR017	Hostování klienta	Klient musí být hostován na vlastní infrastruktuře.	Vysoká
FR018	Ukládání citlivých informací pro komunikaci se službami	Klient nesmí uchovávat citlivé údaje jako tajemství nebo certifikáty v agentovi uživatele.	Vysoká
FR019	Autorizace a autentizace uživatele na straně klienta	Klient musí pro autentizaci i autorizaci využít služeb poskytovatele identity.	Vysoká
FR020	Přesměrování nepřihlášeného uživatele	Klient musí nepřihlášeného uživatele po otevření webové aplikace přesměrovat na poskytovatele identity pro jeho autentizaci a autorizaci.	Vysoká
FR021	Přístup ke zdrojům klientem	Klient musí pro získaní a manipulaci se zdroji využít služeb REST API.	Vysoká
FR022	Autorizace uživatele při směrování a zobrazení obsahu	Klient musí při směrování ověřit stav přihlášení a oprávnění uživatele k zobrazení cílové webové stránky.	Vysoká
FR023	Profil uživatele	Klient musí umožnit uživateli zobrazit jeho profil.	Vysoká
FR024	Zobrazení a manipulace se soubory	Klient musí umožnit uživateli zobrazit, přejmenovat, nahrát, stáhnout a organizovat soubory do složek.	Vysoká
FR025	Sdílení obsahu s ostatními uživateli	Klient musí umožnit uživateli sdílet soubory s ostatními uživateli.	Vysoká
FR026	Zobrazení rolí a oprávnění uživatele	Klient musí umožnit oprávněnému uživateli zobrazit přidělené oprávnění rolí.	Střední

FR027	Spravování rolí a oprávnění uživatele	Klient musí umožnit oprávněnému uživateli odebrat nebo přidělit roli oprávnění.	Střední
FR028	Lokalizace klienta	Klientovi musí být umožněno zvolit lokalizaci aplikace.	Nízká
FR029	Motiv klienta	Klient musí implementovat tmavý režim s vhodným výběrem kontrastních barev.	Nízká
FR030	Limitace velikosti nahraného souboru	Klient musí zabránit uživateli nahrát dokument o velikosti souboru větší než 100 MiB, velikost úložiště uživatele nesmí být omezena.	Střední

4.2 Nefunkční požadavky

Ve srovnání s funkčními požadavky kladou nefunkční požadavky kvalitativní omezení na funkcionalitu systému a jeho řešení.

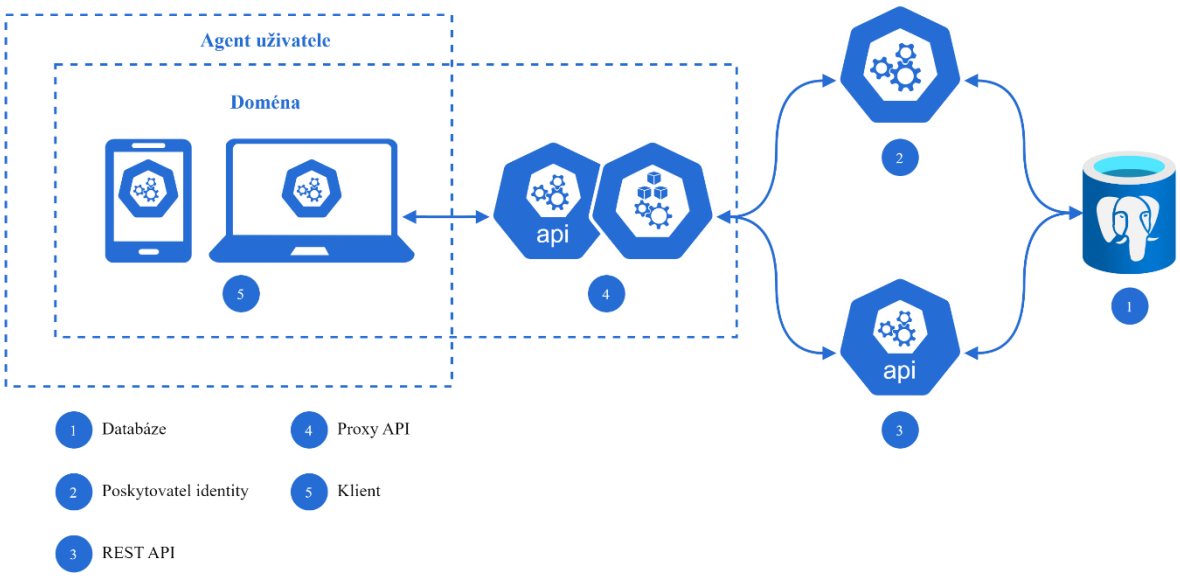
Tabulka 3: Nefunkční požadavky prototypu aplikace.

ID	Požadavek	Popis	Závažnost
NR001	Rychlost zpracování požadavku REST API	Zpracování požadavku REST API nesmí přesáhnout dobu ~1 000 ms.	Vysoká
NR002	Podpora lokalizace	Lokalizované služby musí podporovat jazyky: „americká angličtina a čeština“.	Střední
NR003	Podpora multiplatformního nasazení	Řešení musí být přenositelné a podporovat systémy: „iOS, Windows a Linux“.	Vysoká
NR004	Licenční omezení	Řešení nesmí být licenčně omezené balíčky nebo službami třetích stran.	Vysoká
NR005	Licenční podmínky	Licencování řešení nesmí být omezující pro spotřebitele.	Vysoká
NR006	Soukromí uživatele a telemetrie	Systém nesmí shromažďovat telemetrii a zaznamenávat aktivitu uživatele bez udělení jeho explicitního svolení.	Střední
NR007	Ochrana dat uživatele před přístupem třetích stran	Zaznamenaná aktivita uživatele a telemetrie nesmí být zpřístupněna službám třetích stran.	Střední
NR008	Dostupnost a spolehlivost systému	Dostupnost aplikace nesmí klesnout pod 95% provozní doby.	Vysoká

5 ARCHITEKTURA NAVRŽENÉHO SYSTÉMU

Prototyp aplikace používá architekturu mikro-služeb. Tato architektura separuje aplikační logiku mezi služby. Ve srovnání s monolity se lépe škálují a pro nejvíce zatížené služby použitím orchestrátoru můžeme vytvářet další instance jejich obrazů, mezi které můžeme rozložit zátěž systému. Další výhodou je udržitelnost řešení na úkor jeho komplexity.

Pro připojení ke zdroji dat, načítání, zpracování a aktualizaci dat používá prototyp databázi PostgreSQL. PostgreSQL je populární open-source relační databáze, pro kterou v .NET nalezneme silnou podporu pro integraci s ostatními frameworky a nástroji. Databáze komunikuje s poskytovatelem identity pro správu uživatelských účtů a životosprávu tokenů pro autentizaci a autorizaci uživatele, REST webovým aplikačním rozhraním pro introspekci tokenů, sdílení a manipulaci se zdroji, nebo reverzním proxy serverem a webovým aplikačním rozhraním klienta pro persistenci dat, které jsou nezbytné pro komunikaci mezi poskytovatelem identity a API. Výše popsanou komunikaci mezi službami můžeme vidět na obrázku níže.



Obrázek 1: Architektura mikro-služeb použitá v prototypu aplikace.

Autentizaci i autorizaci uživatelů, správu jejich účtů a životosprávu tokenů provozuje služba poskytovatele identity, se kterou komunikuje REST API a API klienta. Poskytovatel identity vydává a spravuje přístupové tokeny klienta nebo poskytuje informace o uživateli. Aby bylo možné ověřit přístupový token po provolání REST API klientem, volá REST API koncový bod poskytovatele identity pro ověření přístupového tokenu při jeho uplatnění.

Reverzní proxy server a API klienta komunikuje s databází, poskytovatelem identity a REST API. Hlavním účelem tohoto serveru je distribuce statických souborů klienta, skrytí a bezpečné uchování přístupových tokenů mimo prohlížeč uživatele a mapování koncových bodů REST API. Přínosem tohoto přístupu je možnost přidání aplikační logiky pro komunikaci a integraci klienta s ostatními službami. Tato logika se může lišit podle typu aplikace jako mobilní, desktopové nebo v našem případě webová aplikace.

Webovou aplikaci, která komunikuje s reverzním proxy serverem, spouští webový prohlížeč uživatele neboli jeho agent. Po natažení statických souborů aplikace a jejího zavedení operuje aplikace nezávisle na API. Protože webová aplikace a její API jsou spuštěny na stejné doméně, tak prototyp pro přenos citlivých informací jako identita uživatele používá Session Cookies, které přidávají další úroveň abstrakce vrstev zabezpečení.

5.1 Struktura projektu

Každá služba v souboru řešení obsahuje vlastní projekt, který přidáme pomocí šablon, které jsou součástí vývojového sady pro .NET. Prototyp používá vrstvený model, který přispívá ke znovu použitelnosti a udržitelnosti zdrojového kódu, a na kterém závisí projekty služeb. Vrstvy modelu představují v řešení knihovny tříd, pro které můžeme použít šablonu.

Nezávislou vrstva, doména, obsahuje modely databázových entit, konstant, výjimek nebo například kontrakty služeb. Na této vrstvě přímo a nepřímo závisí vyšší vrstvy modelu jako datová a core vrstva. Datová vrstva obsahuje služby pro komunikaci s databází, migrace, skripty a mapování modelů na entity databáze. Ve srovnání core vrstva závisí přímo na doméně i datové vrstvě a její součástí jsou služby a stavební bloky, které používají služby aplikační vrstvy, která představuje aplikační logiku řešení jako obsluha nebo validátory modelů požadavků koncových bodů API, příkazy a dotazy na databázi, které pro odpověď používají modely určené pro přenos, nebo služby pro odvození klíčů a normalizaci hodnot. Příklady služeb core vrstvy je cáchování dat, wrapper služeb jako HTTP klient pro zpracování výjimek nebo dotazy na entity databáze.

Webová aplikace klienta používá oddělený vrstvený model od zbytku systému. Tento model nezahrnuje datovou vrstvu, ale začleňuje integrační vrstvu, která obsahuje služby pro komunikaci s REST API. Separace webové aplikace klienta od zbytku systému je nezbytná kvůli komptabilitě balíčků, které jsou závislé na knihovnách platformy a nemohou být spuštěny v agentovi uživatele. Proxy server závisí na obou vrstvených modelech pro distribuci statických souborů klienta a znovu použití aplikační logiky.

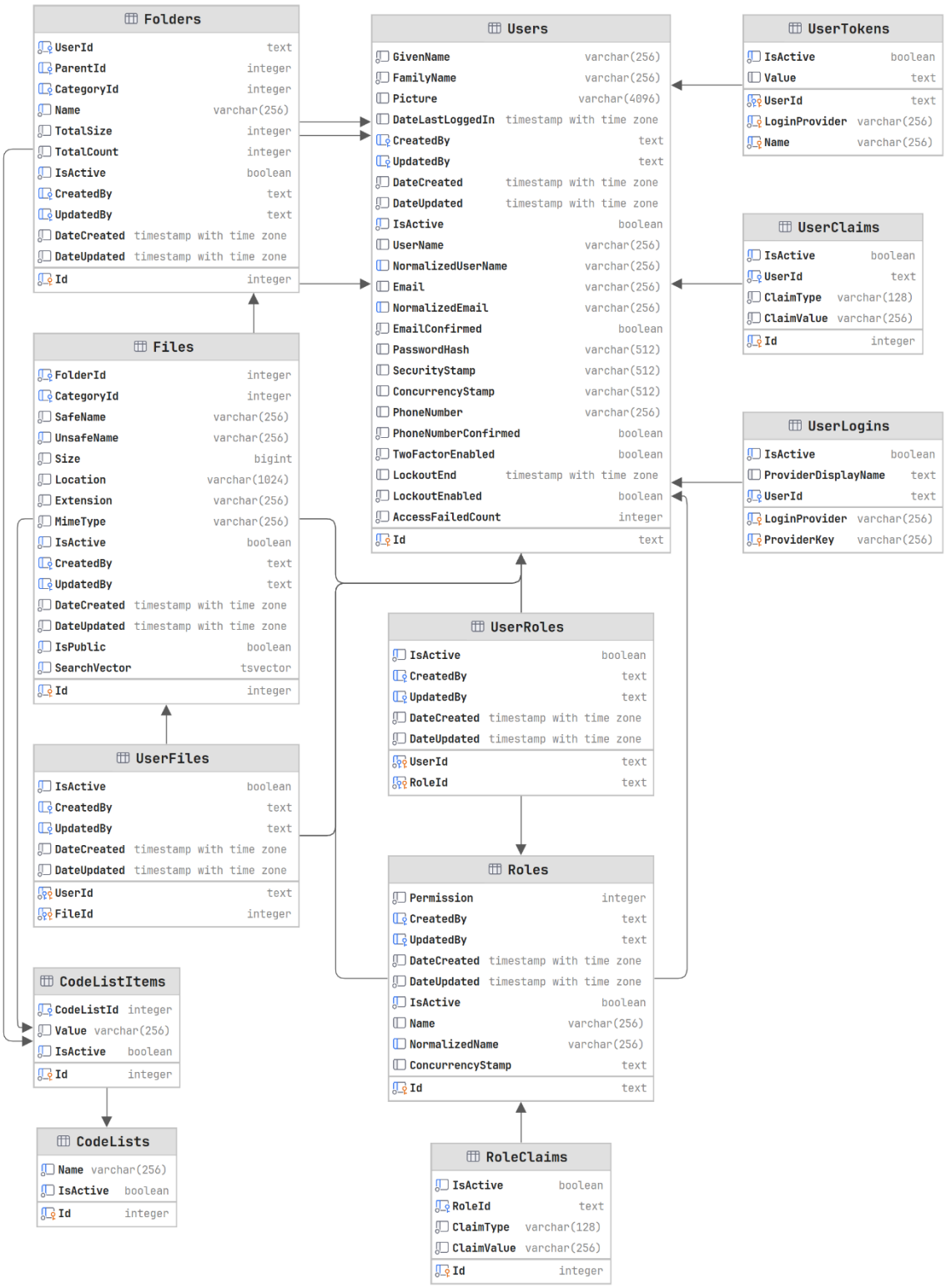
5.2 Struktura databáze

Hlavní entitou modelu databáze je uživatel, na kterého se odkazují tabulky jako složky, soubory nebo přidělené role. Tabulka rolí vytváří vazby na tabulku uživatelů z důvodu auditování změn. Během auditování změn, které nastává při vytváření nového záznamu nebo jeho aktualizaci, se eviduje identifikátor uživatele ve sloupcích „CreatedBy“ nebo „UpdatedBy“ a časová známka ve sloupci „DateCreated“, resp. „DataUpdated“ při aktualizaci záznamu. Kromě tabulky rolí auditování změn podporují další tabulky jako složky, soubory nebo uživatelské role.

Tabulka uživatelů používá jako primární klíč datový typ „GUID“, neboli standardizovaný řetězec znaků, který je výchozím datovým typem ve frameworku pro autentizaci i autorizaci uživatele „Microsoft.AspNetCore.Identity“. Tento datový typ umožňuje ve srovnání s celým číslem ukládat větší množství záznamů na úkor výkonnosti při vyhledávání shody jedinečných klíčů. Dále byla použita indexace vybraných sloupců tabulky, která výrazně urychluje vyhledávání záznamů. Pro tabulku s uživateli se indexují sloupce emailové adresy „NormalizedEmail“, přihlašovacího jména „UserName“ a příznaku „IsEnabled“, který se používá pro politiku mazání dat Soft-Delete.

Jednotlivé tabulky jsou sdružovány do schémat, kde na diagramu níže můžeme vidět tabulky ze schémat „Application“ a „Files“. V diagramu nejsou začleněny schémata vygenerována frameworky Hangfire a OpenIddict, kde OpenIddict vytváří vazby na tabulku uživatelů pro životosprávu tokenů nebo přidává tabulky pro registraci klientů, scopes a dále. Hangfire přidává tabulky, které obsahují registraci jednotlivých úloh a jejich plánování spuštění. V níže uvedeném výčtu je popis existujících schémat prototypu aplikace a jejich účel.

- Application – schéma, které obsahuje informace vázané na uživatele, které slouží pro jeho autentizaci i autorizaci.
- Files – schéma, které obsahuje soubory a složky uživatele.
- OpenIddict – schéma, které obsahuje konfiguraci klientských aplikací a tabulky pro životosprávu a ověření vydaných tokenů.
- Hangfire – schéma, které obsahuje registraci úloh v pozadí, které se spouští v pravidelných intervalech.



Obrázek 2: Diagram databázového systému prototypu aplikace.

6 IMPLEMENTACE POSKYTOVATELE IDENTITY

Pro implementaci poskytovatele identity se využívá MVC webová aplikace, která umožňuje vývoj vícestránkových webových aplikací stejně jako aplikací s rozhraním API. S využitím šablony „ASP.NET Core Web App (Model-View-Controller)“ je možné založit projekt v rámci řešení prototypu aplikace a přidat závislost na aplikační vrstvě.

6.1 PŘIPOJENÍ K DATABÁZI

Pro vytvoření relace mezi poskytovatelem identity a databází Postgresql je rozšíříme nastavení aplikace o parametry jako IP adresa a síťový port, uživatelské jméno a heslo nebo název databáze. Konfiguraci poskytovatele identity doplněnou o nastavení pro připojení k databázi hostované lokálně v kontejnerizovaném prostředí Docker můžeme vidět níže.

```
"Database": {
    "Username": "postgres",
    "Password": "pgAdmin*",
    "Host": "localhost",
    "Database": "Uploadify",
    "Port": 5433,
    "Pooling": true
}
```

Pro stažení obrazu databáze Postgresql a jeho spuštění lze využít příkazový řádek. Příkazový řádek mapuje port 5432 interní sítě Docker na port 5433 sítě, která hostuje virtualizované prostředí Docker a zároveň webové aplikace a aplikační rozhrání prototypu. Uživatelské jméno a heslo nastavíme v konfiguračních souborech pro vývojové účely, pro ostatní prostředí jako testovací nebo interní, produkční a další prostředí přenastavíme hodnoty proměnnými prostředí, které mají přednost před hodnotami konfiguračních souborů. Prototyp aplikace používá verzi PostgreSQL 16.1 databázového systému, v případě migraci na předchozí majoritní verze musíme pozměnit mapování časových známek na datové typy databáze.

```
docker run --name uploadify -e POSTGRES_DB=Uploadify -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=pgAdmin* -p 5433:5432 -d postgres:16.1
```

Abychom zjednodušili implementaci CRUD operací, mapování modelů aplikační logiky na entity databáze a vyvíjeli databázi metodou code-first, využívá prototyp objektově relační mapovač Entity Framework Core. Tento mapovač můžeme přidat do projektu pomocí NuGet balíčku „Microsoft.AspNetCore.Identity.EntityFrameworkCore“, který zahrnuje i integraci s frameworkem pro autentizaci a autorizaci uživatelů, který bude podrobněji popsán v následující kapitole. Samotný objektově relační mapovač nenabízí integraci s databází Postgresql, proto je přidán NuGet balíček „Npgsql.EntityFrameworkCore.PostgreSQL“. Tento

balíček rozšiřuje metody pro mapování modelů na databázové entity a konverzi LINQ metod na SQL dotazy, a také umožňuje implementaci asynchronních volání.

Pro vytváření schémat a tabulek databáze je nutné přidat její kontext, modely a konfiguraci mapování těchto modelů na databázové entity. Kontext databáze dědí ze služby „IdentityDbContext“, která zahrnuje modely pro autentizaci a autorizaci uživatele společně s konfigurací jejich mapování. Modely lze rozšířit o dodatečné vlastnosti jako jméno a příjmení uživatele dědičností.

Objektově relační mapovač pro vytvoření schémat a tabulek vyžaduje přidání kontextu databáze, modelů a jejich konfigurace mapování na entity databáze. Kontext databáze můžeme implementovat zděděním služby „IdentityDbContext“. Součástí této služby jsou modely pro autentizaci i autorizaci uživatele včetně konfigurace mapování. Tyto modely můžeme rozšířit dědičností tříd o další vlastnosti jako jméno a příjmení uživatele.

Níže nalezneme kontext prototypu aplikace, který obsahuje rozšířené modely pro autentizaci i autorizaci uživatele, kolekce modelů pro souborový systém a přetížené metody pro nastavení připojení ke zdroji dat, mapování modelů na entity databáze a ukládání změn prostřednictvím interceptoru. Služba obsahuje více konstruktorů, kde konstruktor dekorovaný atributem bude použit jako výchozí a zbývající bude použit při vytváření migrací.

```
public class DataContext : IdentityDbContext<
    User, Role, string, UserClaim, UserRole, UserLogin, RoleClaim, UserToken>
{
    private readonly ISaveChangesInterceptor _interceptor;

    public virtual DbSet<CodeList> CodeLists { get; set; }
    public virtual DbSet<CodeListItem> CodeListItems { get; set; }
    public virtual DbSet<Folder> Folders { get; set; }
    public virtual DbSet<File> Files { get; set; }

    public DataContext(DbContextOptions<DataContext> options) : base(options)
    {
        _interceptor = null!;
    }

    [ActivatorUtilitiesConstructor]
    public DataContext(
        DbContextOptions<DataContext> options,
        ISaveChangesInterceptor interceptor) : base(options)
    {
        _interceptor = interceptor;
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);

        optionsBuilder.AddInterceptors(_interceptor);
        optionsBuilder.UseOpenIddict();
        optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.TrackAll);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.HasDefaultSchema(Schemas.Application);
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(DataContext).Assembly);

        if (!Database.IsNpgsql())
        {
            modelBuilder.Entity<File>().Ignore(file => file.SearchVector);
        }
    }
}
```

Kontext databáze prototypu obsahuje rozšířené modely pro autentizaci a autorizaci uživatele, kolekci modelů souborového systému a přetížené metody pro nastavení připojení ke zdroji dat, mapování modelů na databázové entity a ukládání změn pomocí interceptoru. Tato služba implementuje dva konstruktory, kde kontejner používá konstruktor dekorovaný atributem jako výchozí pro vyřešení závislostí a druhý pro migraci změn.

Interceptor před uložením změn a aktualizaci entity v databázi nastaví datum vytvoření a poslední změny. Další iterace prototypu aplikace mohou využít interceptor k auditování změn. Aktualizaci data vytvoření a poslední změny můžeme provést rovněž v kontextu

databáze přetížením metod pro ukládání změn, ale použití interceptoru přispívá k čitelnosti a udržitelnosti řešení.

```
public class AuditInterceptor : SaveChangesInterceptor
{
    public override ValueTask<InterceptionResult<int>> SavingChangesAsync(
        DbContextEventData eventData,
        InterceptionResult<int> result,
        CancellationToken cancellationToken = default)
    {
        if (eventData.Context is not DataContext context) return new(result);

        OnBeforeSavedChanges(context);
        return new(result);
    }

    private static void OnBeforeSavedChanges(DataContext context)
    {
        <MODIFY ENTITIES BEFORE PUSHING CHANGES TO DATABASE>
    }
}
```

V souboru pro spuštění aplikace se provádí registrace služeb pomocí rozšiřující metody, jak je ilustrováno níže.

```
public static IServiceCollection AddDatabase(
    this IServiceCollection services,
    bool isDevelopment,
    SystemSettings settings)
{
    services.AddTransient<ISaveChangesInterceptor, AuditInterceptor>();
    services.AddNpgsql<DataContext>(new NpgsqlConnectionStringBuilder
    {
        Username = settings.Database.Username,
        Password = settings.Database.Password,
        Host = settings.Database.Host,
        Database = settings.Database.Database,
        Pooling = settings.Database.Pooling,
        Port = settings.Database.Port
    }.ConnectionString, options =>
    {
        options.EnableRetryOnFailure(5);
        options.MigrationsAssembly("Uploadify.Server.Data");
    }, options =>
    {
        options.EnableDetailedErrors(isDevelopment);
        options.EnableSensitiveDataLogging(isDevelopment);
    });

    return services;
}
```

6.2 Migrace změn

Pro přidání migrace se využívá příkazový řádek a Cmdlet pro generování migrace. Tento příkaz je nutné spustit z kořenového adresáře, kde se nachází soubor s řešením prototypu aplikace „Uploadify.sln“. Vytvořené migrace jsou umístěny v datové vrstvě poskytovatele identity.

```
dotnet ef migrations add AddTablesAndSchemas --project .\src\Uploadify.Server.Data\ -
--startup-project .\src\Uploadify.Server.IdentityServer\ --output-dir
.\Infrastructure\EF\Migrations\
```

Spuštění migrací po zavedení aplikace je automatizováno úlohou v pozadí „Worker.cs“. Migrace, které byly úspěšně provedeny, jsou zaznamenány v databázi v tabulce historie migrací pojmenované „__EFMigrationsHistory“, kterou nalezneme ve schématu „public“.

6.3 Persistence hesel

Během implementace služby pro ověřování identity a oprávnění uživatelů je využíván vestavěný framework „Microsoft.AspNetCore.Identity“. Tento framework poskytuje nástroje pro správu uživatelských účtů, rolí a relací pro přihlášené uživatele, které jsou uchovány ve webovém prohlížeči jako Session Cookie.

Konfigurace modelů a mapování na databázové entity byla provedena v předchozí kapitole. Pro správu identity uživatelů jsou použity standardní služby frameworku s výjimkou služby pro ukládání hesel. Pro tento účel je implementována vlastní služba, která využívá algoritmus Argon2id pro bezpečné šifrování hesel namísto vestavěného algoritmu PBKDF2. [34, 35]

Argon2id je hybridní verzí algoritmů Argon2i a Argon2d, která kombinuje odolnost proti útokům pomocí postranních kanálů algoritmu Argon2i a zvýšenou odolnost algoritmu Argon2d proti útokům založeným na kompromisu mezi pamětí a časem (time-memory trade-off). Vzhledem k absenci certifikované nativní knihovny v .NET pro implementaci tohoto algoritmu je použita externí knihovna „libsodium“ pro kryptografické operace, jejíž funkce jsou integrovány prostřednictvím pomocné třídy. Výhodou této implementace oproti standardnímu řešení je lepší odolnost proti zmíněným útokům a schopnost minimalizovat rizika spojená s výkonnostním přírůstkem výpočetní techniky, zejména více jádrové procesory pro paralelní výpočty jako grafické karty. [36, 37]

6.3.1 Mapování funkcí dynamicky linkované knihovny

Aby prototypu aplikace bylo umožněno používat funkce knihovny pro hashovaní hesel, mapuje statická třída „SodiumLibrary“ API dynamicky linkované knihovny. Před voláním zmapovaných funkcí se inicializuje knihovna prostřednictvím funkce „sodium_init“ spouštěné ve statickém konstruktoru, který se spouští při prvním přístupu k metodám a vlastnostem třídy. Při opakovaném volání nedochází ke znovu inicializování knihovny.

Funkce pro generování náhodné sekvence bajtů a odvození klíče z hesla a soli se mapují stejným způsobem jako funkce pro inicializaci knihovny. Definice funkcí používá při mapování standardní datové typy jako bajt a celé číslo. Typy jako „span“ namísto sekvence bajtů nejsou podporovány a nebudou správně interpretovány při překladu prototypu aplikace. [37]

```
public class SodiumLibrary
{
    private const string Name = "libsodium";

    static SodiumLibrary() => sodium_init();

    [DllImport(Name, CallingConvention = CallingConvention.Cdecl)]
    internal static extern void sodium_init();

    [DllImport(Name, CallingConvention = CallingConvention.Cdecl)]
    internal static extern void randombytes_buf(byte[] buffer, int size);

    [DllImport(Name, CallingConvention = CallingConvention.Cdecl)]
    internal static extern int crypto_pwhash(byte[] buffer, long bufferLen,
        byte[] password, long passwordLen, byte[] salt, long opsLimit,
        int memLimit, int alg);
}
```

6.3.2 Nastavení služby pro hashovaní hesel

Během generování solí a odvození klíče se nastavují parametry jako počet bajtů odvozeného klíče a soli, počet cyklů, které algoritmus provede při jejím výpočtu nebo omezení velikosti použité paměti. Prototyp aplikace používá nastavení aplikace, které se mapuje na modely nastavení služeb. Nastavení služby „ArgonPasswordHasher“ pro hashovaní hesel „ArgonPasswordHasherOptions“ implementuje rozhrání „IOptions“, které umožňuje vkládat závislost na nastavení služeb. Nastavení služby pro hashovaní hesel upřesňuje typ algoritmu Argon2, velikost odvozeného klíče a generované soli, počet cyklů a velikost rezervované paměti použité pro odvození klíče nebo pepper.

```
public class ArgonPasswordHasherOptions
{
    public const string Delimiter = ".";

    public string? Pepper { get; set; }
    public int SaltSize { get; set; } = 16;
    public int KeySize { get; set; } = 32;
    public Argon2Type Algorithm { get; set; } = Argon2Type.Argon2id;
    public long OperationsLimit { get; set; } = 4;
    public int MemoryLimit { get; set; } = 268_435_456;
}
```

6.3.3 Hashovaní hesel

Pro nahrazení výchozí služby pro hashovaní hesel implementuje služba rozhrání „IPasswordHasher“, které obsahuje metody pro hashovaní hesla a jeho ověření. Metoda pro hashovaní hesla odvozuje klíč ze sekvence bajtů hesla, které bylo rozšířeno o pepper na jeho konci. Pepper je součástí nastavení aplikace ve srovnání se solí, která je náhodně generována a předána funkci pro odvození klíče jako parametr. Funkce odvození klíče jako parametry hesla a soli přijímá sekvence bajtů včetně jejich délky, kde pro získání bajtů z řetězce hesla se používá kódování UTF8. Odvozený klíč a jeho sůl jsou před uložením do databáze kódovány na Base64 řetězce, které se dále zřetězí za použití oddělovače, který není součástí znaků kódování Base64 jako tečka. [34, 35]

```
public string HashPassword(TUser user, string password)
{
    var passwordWithPepper = password + Options.Pepper;
    var derivedKeyBytes = new byte[Options.KeySize];
    var randomlyGeneratedSaltBytes = new byte[Options.SaltSize];

    SodiumLibrary.randombytes_buf(randomlyGeneratedSaltBytes,
        randomlyGeneratedSaltBytes.Length);

    var passwordHashingResult = SodiumLibrary.crypto_pwhash(
        derivedKeyBytes,
        derivedKeyBytes.Length,
        Encoding.UTF8.GetBytes(passwordWithPepper),
        passwordWithPepper.Length,
        randomlyGeneratedSaltBytes,
        Options.OperationsLimit,
        Options.MemoryLimit,
        (int)Options.Algorithm);

    return string.Format("{0}{1}{2}",
        Convert.ToBase64String(derivedKeyBytes),
        ArgonPasswordHasherOptions.Delimiter,
        Convert.ToBase64String(randomlyGeneratedSaltBytes));
}
```

6.3.4 Ověření shody hesel

Metoda pro ověření hesla rozdělí hash hesla v Base64 na odvozený klíč a jeho sůl, které dekódujeme na sekvence bajtů stejně jako heslo pro ověření shody. Během odvození klíče z hesla pro ověření shody se používá stejná sůl jako při hashování původního hesla. Po odvození klíče se jeho výsledek srovná s klíčem původního hesla, kde se pro porovnání sekvence bajtů používá kryptografická funkce, která vyhodnocuje různé řetězce stejně dlouhou dobu. Tento přístup ochraňuje systém před útokem postranními kanály.

```
public PasswordVerificationResult VerifyHashedPassword(
    TUser user, string hash, string password)
{
    var passwordWithPepper = password + Options.Pepper;
    var derivedKeyBytes = new byte[Options.KeySize];
    var originalDerivedKeyWithSalt =
        hash.Split(Delimiter, StringSplitOptions.RemoveEmptyEntries);

    var originalDerivedKeyBytes = FromBase64String(originalDerivedKeyWithSalt[0]);
    var originalSaltBytes = FromBase64String(originalDerivedKeyWithSalt[1]);

    var passwordHashingResult = SodiumLibrary.crypto_pwhash(
        derivedKeyBytes,
        derivedKeyBytes.Length,
        Encoding.UTF8.GetBytes(passwordWithPepper),
        passwordWithPepper.Length,
        originalSaltBytes,
        Options.OperationsLimit,
        Options.MemoryLimit,
        (int)Options.Algorithm);

    return CryptographicOperations.FixedTimeEquals(
        derivedKeyBytes, originalDerivedKeyBytes) ? Success : Failed;
}
```

6.4 Autorizace a autentizace

Rozšiřující metoda popsaná níže slouží pro registraci služeb, které spravují uživatele, role, generátory tokenů a normalizátory pro uživatelská jména a emailové adresy, které jsou využívány k indexaci záznamů v databázi. [34]

```
public static IServiceCollection AddIdentity(
    this IServiceCollection services,
    SystemSettings settings)
{
    services.AddIdentity<User, Role>(options =>
    {
        options.ClaimsIdentity.UserNameClaimType = OpenIddictConstants.Claims.Name;
        options.ClaimsIdentity.UserIdClaimType = OpenIddictConstants.Claims.Subject;
        options.ClaimsIdentity.RoleClaimType = OpenIddictConstants.Claims.Role;
        options.ClaimsIdentity.EmailClaimType = OpenIddictConstants.Claims.Email;
        options.ClaimsIdentity.SecurityStampClaimType = Claims.SecurityStamp;

        <OTHER OPTIONS>
    })
    .AddEntityFrameworkStores<DataContext>()
    .AddDefaultTokenProviders();

    services.AddSingleton<IPasswordHasher<User>, ArgonPasswordHasher<User>>();
    services.AddOptions<ArgonPasswordHasherOptions>()
        .Configure(options => options.Pepper = "...")
        .ValidateDataAnnotations()
        .ValidateOnStart();

    return services;
}
```

Prototyp aplikace používá knihovnu MediatR, která implementuje návrhový vzor CQRS. MediatR umožňuje vývojáři řetězení příkazů a dotazů, sekvenční vykonání filtrů před a po jejich spuštění. Knihovna pro příkazy i dotazy nabízí jednotné rozhrání, abychom tyto požadavky vzájemně odlišily, přidáme odvozená a jedinečná rozhrání: [38]

```
interface IQuery<out TResponse> : IRequest<TResponse>
    where TResponse : BaseResponse;

interface IQueryHandler<in TQuery, TResponse> : IRequestHandler<TQuery, TResponse>
    where TQuery : IQuery<TResponse>
    where TResponse : BaseResponse;
```

Obdobným způsobem přidáme abstrakce pro příkazy:

```
interface ICommand<out TResponse> : IRequest<TResponse>
    where TResponse : BaseResponse;

interface ICommandHandler<in TCommand, TResponse> : IRequestHandler<
    TCommand, TResponse>
    where TCommand : ICommand<TResponse>
    where TResponse : BaseResponse;
```

Pipeline obsluhy požadavků zahrnuje dva typy filtrů: první doplňuje údaje z kontextu přihlášeného uživatele do modelů příkazů nebo dotazů jako jméno nebo email uživatele. Druhý filtr validuje modely příkazů, například formát emailové adresy. Filtry implementují rozhraní „IPipelineBehavior“. [38]

Výstupy zpracování dotazů a příkazů jsou standardizovány. Základní model obsahuje stavový kód a chybovou zprávu, která může zahrnovat uživatelsky srozumitelný text nebo kód pro překlad a kolekci validačních chyb. Tato metoda umožňuje jednotně odpovědět na různé požadavky v rámci systému.

```
public class BaseResponse
{
    public Status Status { get; set; }
    public RequestFailure? Failure { get; set; }
}

public class RequestFailure
{
    public string? UserFriendlyMessage { get; set; }
    public Dictionary<string, string[]>? Errors { get; set; }

    [JsonIgnore] public Exception? Exception { get; set; }
}
```

Poskytovatel identity pro validaci příkazů využívá filtr, který před obsluhou spouští validátor jeho modelu. V případě detekce chyb v modelu filtr zabrání dalšímu postupu v pipeline a vrátí odpověď, který obsahuje relevantní stavový kód a validační chyby. Tento postup odděluje validaci a zpracování modelů příkazů od jejich obsluhy.

```
public class ValidationPipelineBehaviour<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
    where TRequest : IBaseRequest<TResponse>, ICommand<TResponse>
    where TResponse : BaseResponse
{
    private readonly IValidator<TRequest>? _validator;

    public ValidationPipelineBehaviour(IValidator<TRequest>? validator)
    {
        _validator = validator;
    }

    public async Task<TResponse> Handle(
        TRequest request,
        RequestHandlerDelegate<TResponse> next,
        CancellationToken cancellationToken)
    {
        if (_validator == null)
        {
            return await next();
        }

        var validationResult = await _validator.ValidateAsync(
            new ValidationContext<TRequest>(request), cancellationToken);

        var errors = validationResult.DistinctErrorsByProperty();
        if (errors.Count <= 0)
        {
            return await next();
        }

        var response = Activator.CreateInstance<TResponse>();

        response.Status = BadRequest;
        response.Failure = new()
        {
            Errors = errors,
            UserFriendlyMessage = Translations.RequestStatuses.BadRequest,
            Exception = new BadRequestException(request.GetType().Name)
        };

        return response;
    }
}
```

Registrace příkazů, dotazů a filtrů se provádí rozšiřujícími metodami knihovny MediatR, přičemž pořadí určuje sekvenci jejich spuštění v pipeline během zpracování dotazů nebo příkazů.

Pro implementaci formuláře pro registraci i přihlášení máme k dispozici potřebné závislosti. Architektura MVC vyžaduje rozdělení aplikace na části: pohledy (webové uživatelské rozhraní), řadiče (implementující aplikační logiku) a modely (pro předávání dat mezi řadiči a pohledy). ASP.NET Core využívá konvence pojmenování a očekává umístění řadičů a pohledů v odpovídajících podadresářích s názvy „Controllers“ a „Views“. Umístění zdrojových souborů mimo tyto adresáře by mohlo vést k chybám při sestavování projektu. [24]

Formuláře jsou součástí řadiče „AccountController“, který poskytuje koncové body s metodami GET pro zobrazení statických stránek a POST pro zpracování odeslaných formulářů. Tyto koncové body mohou přijímat volitelné parametry z URL pro přesměrování uživatele zpět na autorizaci klienta, pokud uživatel nebyl přihlášen nebo mu vypršela platnost relace.

```
public class AccountController : Controller
{
    private readonly SignInManager<User> _manager;
    private readonly IMediator _mediator;
    private readonly ILogger<HomeController> _logger;

    [HttpGet]
    public IActionResult Login(string? returnUrl) {…}

    [HttpPost, ValidateAntiForgeryToken]
    public async Task<IActionResult> HandleLogin(
        string? returnUrl, LoginForm? form, CancellationToken cancellationToken)
    {…}

    [HttpGet]
    public IActionResult Register(string? returnUrl) {…}

    [HttpPost, ValidateAntiForgeryToken]
    public async Task<IActionResult> HandleRegister(string? returnUrl,
        RegisterForm? form, CancellationToken cancellationToken) {…}

    [HttpPost, ValidateAntiForgeryToken]
    public async Task<IActionResult> HandleLogout(string? returnUrl) {…}
}
```

6.5 Protokoly OAuth 2.0 a OpenID Connect

Protože zabezpečení přenosu informací uživatele a zabránění neoprávněnému přístupu k jeho datům je kritický požadavek, implementuje prototyp aplikace ověřené a osvědčené protokoly, které poskytují požadovanou úroveň zabezpečení pro různé typy aplikací jako mobilní, webové nebo desktopové. Vývoj poskytovatele identity, který splňuje požadavky v souladu s protokoly je náročný a složitý proces. Prototyp aplikace klade důraz na licenční podmínky, a proto nesmí závislosti porušovat zásady open-source licencování. Poskytovatele identity třetích stran jako Google Identity API nebo certifikovaný .NET framework

Duende's Identity Server nelze kvůli licenčním omezením využít během implementace prototypu aplikace. Alternativní řešení jako hostování open-source řešení KeyCloak nabízí vysokou úroveň zabezpečení, ale omezuje možnosti rozšíření aplikační logiky pro správu uživatelských účtů nebo přizpůsobení uživatelského rozhrání.

Framework „OpenIddict.AspNetCore" poskytuje stavební bloky pro OAuth 2.0 a OpenID Connect, které zahrnují autorizační toky, funkce a nástroje pro životosprávu tokenů a rozšiřující funkce pro integraci klienta nebo hostování poskytovatele identity. Framework samotný neimplementuje koncové body specifikované v protokolu OAuth 2.0, a proto není certifikovaný jako alternativa Duende's Identity Server.

Pro uchování konfigurace klienta a životosprávu tokenů je nezbytné rozšířit kontext databáze. Modely a jejich mapování na databázové entity spolu s rozšiřujícími metodami pro integraci s objektově-relačním mapovačem jsou součástí balíčku „OpenIddict.EntityFrameworkCore". [39, 40]

Řadiče a jejich koncové body musí dodržovat jmenné konvence a doporučení. Aplikační rozhraní poskytovatele identity implementuje následující koncové body:

- /connect/authorize – koncový bod pro autorizaci klienta, který v případě nepřihlášeného uživatele zobrazuje výzvu k ověření jeho identity nebo v případě neautorizovaného klienta zobrazuje výzvu k udělení explicitního svolení klientovi pro přístup k informacím uživatele.
- /connect/token – koncový bod pro vydání přístupového tokenu, tokenu identity a tokenu pro obnovení přístupu.
- /connect/introspect – koncový bod pro ověření přístupového tokenu.
- /connect/verify – koncový bod pro ověření klienta.
- /connect/logout – koncový bod pro odhlášení uživatele, který zneplatní jeho relaci s poskytovatelem identity.
- /connect/userinfo – koncový bod pro získání identity uživatele, který poskytuje ve srovnání s token identity dodatečné informace jako profilový obrázek, který kvůli velikosti není vhodné začlenit do tokenu identity.

6.5.1 Koncový bod pro autorizaci klienta

Prototyp implementuje autorizační tok Authorization Code Flow s PKCE, který vyžaduje od uživatele udělení explicitního svolení. Koncový bod pro autorizaci klienta ověřuje stav přihlášení uživatele a jeho svolení sdílet informace s klientem. Pokud klient používá jiný typ svolení jako implicitní nebo žádné, poskytovatel identity nesmí požadavku vyhovět. Při přesměrování uživatele pro ověření identity je původní požadavek zachován v zakódované formě v URL dotazu. Název tohoto parametru musí odpovídat názvu použitému při implementaci koncových bodů pro přihlášení a registraci uživatele, který umožňuje po ověření identity obnovit proces autorizace klienta.

Logika tohoto koncového bodu je rozdělena mezi tři koncové body řadiče se stejnou směrovací adresou. První koncový bod ověří identitu uživatele a kontroluje oprávnění poskytnout klientovi informace. [39, 40]

```
[HttpGet("~/connect/authorize"), HttpPost("~/connect/authorize")]
[IgnoreAntiforgeryToken]
public async Task<IActionResult> Authorize()
{
    if (<WHEN INVALID REQUEST THEN RETURN FORBIDDEN RESPONSE OR
         WHEN UNAUTHENTICATED USER THEN RETURN LOGIN REDIRECT>)
    {
        if (request.HasPrompt(OpenIddictConstants.Prompts.None)) return Forbid(…);
        return Challenge(…);
    }

    if (<WHEN UNAUTHENTICATED USER OR USER NOT FOUND
         THEN SIGN OUT CURRENT USER AND RETURN LOGIN REDIRECT>) return Challenge(…);

    switch (<RETRIEVE CLIENT CONSENT TYPE>)
    {
        case <WHEN EXTERNAL CONSENT TYPE AND UNAUTHORIZED CLIENT
              THEN RETURN FORBIDDEN RESPONSE>:
            return Forbid(…);

        case <WHEN IMPLICIT CONSENT TYPE THEN RETURN SIGN-IN RESPONSE>:
        case <WHEN EXTERNAL CONSENT TYPE AND AUTHORIZED CLIENT THEN RETURN SIGN-IN
              RESPONSE>:

        case <WHEN EXPLICIT CONSENT TYPE AND AUTHORIZED CLIENT WITHOUT CONSENT PROMPT
              THEN RETURN SIGN-IN RESPONSE>:
            return SignIn(…);

        case <WHEN EXPLICIT OR SYSTEMATIC CONSENT TYPE WITHOUT AUTHORIZE PROMPTS
              THEN RETURN FORBIDDEN RESPONSE>:
            return Forbid(…);

        default: <DISPLAY AUTHORIZE PROMPT AND RETURN>;
    }
}
```

6.5.2 Koncový bod pro udělení svolení klientovi pro získání informací uživatele

Pokud je zobrazena výzva pro autorizaci klienta po odeslání formuláře, pak bude dotaz přesměrován na koncové body dekorované atributem „FormValueRequired“, který podle hodnoty formuláře určuje destinaci pro obsluhu požadavku. Autorizace klienta generuje nový kontext přihlášeného uživatele, který závisí na konfiguraci klienta a určuje informace, ke kterým klient může přistoupit, nebo služby, které může klient provolat uplatněním přístupového tokenu. [39, 40]

```
[Authorize, ValidateAntiForgeryToken]
[HttpPost("~/connect/authorize"), FormValueRequired("submit.Accept")]
public async Task<IActionResult> Accept()
{
    <RETRIEVE CURRENT USER, CLIENT CONFIGURATION AND AUTHORIZED CLIENTS>

    if (<WHEN UNAUTHORIZED CLIENT AND EXTERNAL CONSENT TYPE
         THEN RETURN FORBIDDEN RESPONSE>)
    {
        return Forbid(…);
    }

    <MAP CURRENT USER CLAIMS TO THEIR DESTINATIONS>

    return SignIn(…);
}
```

6.5.3 Koncový bod pro zamítnutí svolení klientovi pro získání informací uživatele

V případě zamítnutí žádosti o udělení svolení klientovi pro získání informací uživatele odpovídá poskytovatel identity chybovou zprávu a směruje požadavek uživatele zpět do aplikace klienta. [39, 40]

```
[Authorize, ValidateAntiForgeryToken]
[HttpPost("~/connect/authorize")], FormValueRequired("submit.Deny")]
public IActionResult Deny() => Forbid(…);
```

6.5.4 Koncový bod pro výměnu tokenů

Po přihlášení uživatele a autorizaci klienta uplatní klient jednorázový autorizační kód nebo token pro obnovení pro vydaní přístupového tokenu, tokenu identity, který může klient uplatnit při provolání koncových bodů REST API. Pokud se jedná o nepodporovaný typ požadavku, odpoví poskytovatel identity vhodným stavovým kódem a chybovou zprávou.

Po obdržení požadavku a ověření jeho typu poskytovatel identity extrahuje kontext přihlášeného uživatele. V případě, kdy uživatel nebyl nalezen, protože jeho účet byl deaktivován, nebo jeho účet byl zablokován, odpoví poskytovatel identity chybovou zprávou a HTTP

stavovým kódem 403. Pro ostatní scénáře se zmapují informace o uživateli a podle destinací, které se liší konfigurací scopes klienta, je zahrne poskytovatel identity do přístupového tokenu nebo tokenu identity. [39, 40]

```
[IgnoreAntiforgeryToken]
[HttpPost("~/connect/token")]
[Produces(MediaTypeNames.Application.Json)]
public async Task<IActionResult> Exchange()
{
    <EXTRACT OPENIDDICT SERVER REQUEST FROM CONTEXT>

    if (<WHEN AUTHORIZATION CODE OR REFRESH TOKEN GRANT TYPE
         THEN PROCESS REQUEST>)
    {
        <RETRIEVE CURRENT USER AND MAP HIS CLAIMS TO THEIR DESTINATIONS>
        <WHEN CURRENT USER IS UNAUTHENTICATED OR CANNOT SIGN-IN
         THEN RETURN FORBIDDEN RESPONSE>

        return SignIn(…);
    }

    <RETURN INVALID GRANT TYPE RESPONSE>
}
```

6.5.5 Mapování informací uživatele na tokeny

V úloze na pozadí nastavujeme klientovi současně s typem autorizačního toku, autorizace klienta nebo oprávnění pro provolání koncových bodů poskytovatele identity také scopes. Scopes určují, které informace o uživateli začlení poskytovatel identity do přístupového tokenu nebo tokenu identity. Prototyp poskytne informace klientovi v tokenu identity a přístupovém tokenu až na výjimky jako bezpečnostní známky, která se změní při aktualizaci přihlašovacích údajů, aby poskytovatel identity vynutil po uživateli opětovné ověření jeho totožnosti. Bezpečnostní známka by neměla být součástí tokenů, a proto nenastavíme destinaci tohoto typu informace. Přidání vlastních scopes vyžaduje použití předpon. Konvenci použijeme pouze při jejich nastavení na straně serveru ve službě v pozadí. [41]

```
public static IEnumerable<string> GetDestinations(Claim claim)
{
    switch (claim.Type)
    {
        case OpenIddictConstants.Claims.Subject:
            yield return OpenIddictConstants.Destinations.AccessToken;
            yield return OpenIddictConstants.Destinations.IdentityToken;

            yield break;

        case OpenIddictConstants.Claims.Name:
        case OpenIddictConstants.Claims.GivenName:
        case OpenIddictConstants.Claims.FamilyName:
            yield return OpenIddictConstants.Destinations.AccessToken;

            if (claim.Subject.HasScope(Scopes.Name))
            {
                yield return OpenIddictConstants.Destinations.IdentityToken;
            }

            yield break;

        <SET DESTINATIONS FOR OTHER CLAIM TYPES>

        case Claims.SecurityStamp: yield break;

        default:
            yield return OpenIddictConstants.Destinations.AccessToken;
            yield break;
    }
}
```

6.5.6 Koncový bod pro odhlášení uživatele

Provolání koncového bodu pro odhlášení uživatele zneplatní kontext přihlášeného uživatele pro OpenIddict i externí autentizační schéma. Níže můžeme vidět koncové body pro odhlášení uživatele a zobrazení promptu. [40]

```
[HttpGet("~/connect/logout")]
public IActionResult Logout() => View();

[ActionName(nameof(Logout))]
[HttpPost("~/connect/logout")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LogoutPost()
{
    <SIGN OUT CURRENT USER AND THEN REDIRECT HIM BACK TO CLIENT APPLICATION>
    return SignOut(…);
}
```

6.5.7 Koncový bod pro získání dodatečných informací o uživateli

Informace o uživateli jako profilový obrázek není vhodné začlenit do tokenu identity kvůli jeho velikosti. Klientské aplikace pro získání údajů o uživateli používají token identity nebo alternativně provolají koncový bod pro získání dodatečných informací. Prototyp implementuje a podporuje obě možnosti získání identity uživatele. Koncový bod používá nastavení klienta jako koncový bod pro vydání tokenů pro rozhodnutí, které informace začlenit do odpovědi na požadavek. [41]

```
[Authorize(AuthenticationSchemes =
     OpenIddictServerAspNetCoreDefaults.AuthenticationScheme)]
[HttpGet("~/connect/userinfo"), HttpPost("~/connect/userinfo")]
[Produces(MediaTypeNames.Application.Json)]
public async Task<IActionResult> UserInfo()
{
    <RETRIEVE CURRENT USER>
    if (<WHEN CURRENT USER NOT FOUND THEN RETURN AUTHENTICATION FAILED RESPONSE>)
    {
        return Challenge(…);
    }

    <MAP CURRENT USER CLAIMS>
    return Ok(claims);
}
```

7 REST API

Aplikační rozhraní pro sdílení a manipulaci s daty můžeme začlenit do projektové struktury použitím šablony „ASP.NET Core Web API“. Tento projekt sdílí strukturu cibulovitého modelu s poskytovatelem identity a je závislý na aplikační vrstvě. Stěžejní požadavek tohoto webového aplikačního rozhraní je dokumentace, která popisuje koncové body, jeho parametry, vstupní a výstupní modely, stejně jako stavové kódy. Pro zobrazení dokumentace se využívá framework NSwag, který umožňuje dalším službám generování klientů.

7.1 Introspekce přístupového tokenu

Pro zajištění ochrany před neoprávněným přístupem uživatelů ke zdrojům je vyžadováno, aby klient přikládal přístupový token v hlavičce každého požadavku na všechny koncové body. Framework OpenIddict poskytuje balíček, který umožňuje integraci mezi klientem a poskytovatelem identity a poskytuje služby a filtry pro validaci přístupového tokenu. Pro implementaci validace je nezbytné registrovat služby a filtry, které závisí na nastavení vydavatele tokenu a cílové skupiny (audience). Vydavatel tokenu prototypu aplikace odpovídá poskytovateli identity. Cílová skupina je definována identifikátorem webového API, který byl použit při registraci klientských aplikací ve službě v pozadí „Worker.cs“. Předmětem validace jsou atributy tokenu jako vydavatel, cílová skupina, platnost tokenu a digitální podpis. Nastavení introspekce tokenu je možné filtry a službami, které vyžadují identifikátor REST API a jeho tajemství, které byly použity během jeho registrace. [39]

```
services.AddOpenIddict()
    .AddValidation(options =>
    {
        options.SetIssuer(settings.IdentityProvider.Issuer);
        options.AddAudiences(settings.Api.ID);

        options.UseIntrospection()
            .SetClientId(settings.Api.ID)
            .SetClientSecret(settings.Api.Secret);

        options.UseSystemNetHttp();
        options.UseAspNetCore();
    });
```

Introspekce se spouští během autentizace uživatele, kdy dochází k řízení směrování požadavku. Pro začlenění autentizační vrstvy do procesního řetězce REST API je nezbytné zaregistrovat služby, pro které se nastaví výchozí autentizační schéma OpenIddict. Toto nastavení umožňuje provádění validace a introspekce během ověřování identity uživatele. Autentizace uživatele se realizuje na základě atributů, kterými jsou dekorovány koncové body

nebo řadiče. Pokud koncový bod není označen autorizačním atributem, nedochází k validaci ani introspekci přístupového tokenu a k ověření oprávnění uživatele. Autorizační atribut umožňuje upřesnit autentizační schéma a role uživatele. Pro implementaci autorizace založené na oprávněních, která vyžaduje role přidělené nebo obsažené v přístupovém tokenu, je potřeba vytvořit vlastní řešení pro zpracování autorizace uživatele. [39]

```
application.UseRouting();
application.UseAuthentication();
application.UseAuthorization();

application.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

Během konfiguraci pipeline pro zpracování požadavků na REST API prototyp nastaví filtry pro autentizaci i autorizaci uživatele v pořadí, které můžeme vidět na obrázku výše. [39]

7.2 Autorizační filtr

Framework ASP.NET Core umožňuje vložení a spuštění autorizačního filtru před přesměrováním požadavku na řadič a jeho koncový bod. V prototypu aplikace se pro autorizaci uživatele využívají oprávnění, nikoli role. Tato oprávnění reprezentují jednotlivé operace nad zdroji. Obrázek demonstruje výčet oprávnění označených atributem „Flags“. Atribut umožňuje interpretaci proměnné s tímto výčtem jako bitové pole a provádění bitových operací. Výchozí definice v jazyce .NET pro výčtové hodnoty používá 32-bitové celé číslo, které umožňuje přiřazení až 32 různých oprávnění k jedné roli. [42]

```
[Flags]
public enum Permission
{
    None = 1,
    ViewPermissions = 2,
    EditPermissions = 4,
    ViewRoles = 8,
    EditRoles = 16,
    ViewUsers = 32,
    EditUsers = 64,
    ViewFiles = 128,
    EditFiles = 256,
    All = ~None
}
```

Pro poskytnutí vlastní implementace obsluhy autorizace uživatele je nezbytné přetížit a implementovat rozhraní služeb jako poskytovatel politiky autorizace a obsluha autorizace nebo rozšířit autorizační atribut. Vzhledem k tomu, že autorizace založená na oprávněních je

využívána jak REST API, tak klientem, je vhodné přesunout implementaci do samostatných knihoven, které budou integrovat potřebné závislosti.

Rozšířením autorizačního atributu o bitové pole s příznaky oprávnění, které budou využívány během autorizace uživatele, umožní jejich načtení z kontextu a validaci při obsluze autorizace uživatele. Pokud není upřesněna role ani oprávnění, pak atribut validuje stav přihlášení uživatele. Z tohoto důvodu je nastavena výchozí hodnota oprávnění na „None“. [42]

```
public class PermissionAttribute : AuthorizeAttribute
{
    public PermissionAttribute(Permission permission) => Permission = permission;

    public Permission Permission
    {
        get => !IsNullOrWhiteSpace(Policy)
            ? PolicyNameHelpers.GetPermissionsFrom(Policy)
            : Permission.None;
        set => Policy = value != Permission.None
            ? PolicyNameHelpers.GetPolicyNameFor(value)
            : Empty;
    }
}
```

Pro umožnění validace oprávnění uživatele pomocí autorizačního filtru je nutné poskytnout vlastní implementaci politiky autorizace. Tato implementace rozšiřuje autorizační požadavky, které jsou následně vyhodnoceny pro možnou validaci. V případě upřesnění oprávnění uživatele, dojde ke spuštění obsluhy autorizačního požadavku pro dané oprávnění. Podrobnosti o autorizačním požadavku jsou dostupné níže. [42]

```
public class PermissionRequirement(Permission permission)
    : IAuthorizationRequirement
{
    public Permission Permission { get; } = permission;
}
```

Implementaci poskytovatele politiky je možné realizovat za využití základní třídy „DefaultAuthorizationPolicyProvider“. V této třídě přetížíme metodu pro získání autorizační politiky. Poskytovatel autorizační politiky přidá autorizační požadavek v případě, že jsou upřesněna oprávnění uživatele. [42]

```
public class PermissionPolicyProvider : DefaultAuthorizationPolicyProvider
{
    private readonly AuthorizationOptions _options;

    public override async Task<AuthorizationPolicy?> GetPolicyAsync(
        string policyName)
    {
        AuthorizationPolicy? policy = await base.GetPolicyAsync(policyName);
        if (IsNullOrWhiteSpace(policyName) ||
            !PolicyNameHelpers.IsValidPolicyName(policyName))
        {
            return policy;
        }

        Permission permissions =
            PolicyNameHelpers.GetPermissionsFrom(policyName);
        policy = new AuthorizationPolicyBuilder()
            .AddRequirements(new PermissionRequirement(permissions))
            .Build();

        _options.AddPolicy(policyName, policy);
        return policy;
    }
}
```

Autorizační filtr spouští obsluhu pro jednotlivé autorizační požadavky. Aby bylo možné spustit obsluhu autorizačního požadavku pro oprávnění uživatele, je nutné implementovat službu, která z přístupového tokenu extrahuje oprávnění uživatele a následně provádí jeho autorizaci. Validace oprávnění uživatele využívá bitovou operaci AND, která při shodě vrací nenulovou hodnotu. [42]

```
public class PermissionHandler : AuthorizationHandler<PermissionRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        PermissionRequirement requirement)
    {
        Claim? permissionClaim =
            context.User.FindFirst(Permissions.Claims.Permission);

        if (permissionClaim == null)
        {
            return Task.CompletedTask;
        }

        Permission permissions =
            PolicyNameHelpers.GetPermissionsFrom(permissionClaim.Value);

        if ((permissions & requirement.Permission) != 0)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Na závěr registrujeme služby v kontejneru pro vkládání závislostí, kde nahradíme výchozí implementace rozhrání, které poskytuje rozšiřující metoda pro přidání základních autorizačních služeb.

```
public static IServiceCollection AddPermissions(this IServiceCollection services)
{
    services.AddAuthorizationCore();
    services
        .AddSingleton<IAuthorizationHandler, PermissionHandler>()
        .AddSingleton<IAuthorizationPolicyProvider, PermissionPolicyProvider>();

    return services;
}
```

7.3 Základní modely dotazů a odpovědí

Pokud obsluha dotazu nebyla úspěšná, server odpoví klientovi obecnou zprávou, která obsahuje stavový kód a chybovou zprávu. Chybová zpráva se skládá z uživatelsky přívětivé hlášky a kolekci validačních chyb spojených s modelem dotazu. Tento přístup umožňuje validační, autentizační i autorizační pipeline v MediatR odpovědět na dotaz před postupem na obsluhu příkazu i dotazu v MediatR.

```
public class BaseResponse
{
    public Status Status { get; set; }
    public RequestFailure? Failure { get; set; }
}
```

Pro rozlišení, které příkazy nebo dotazy využívají základní model odpovědi v MediatR pipeline, implementují dotazy a příkazy rozhraní, které můžeme pro datové typy detekovat použitím typové reflexe.

```
public interface IBaseRequest<out TResponse> : IRequest<TResponse>
    where TResponse : BaseResponse;
```

7.4 Autorizační pipeline v MediatR

Autentizační pipeline začleňuje údaje z kontextu přihlášeného uživatele do modelu dotazu nebo příkazu. Tento postup umožňuje provádět validaci vlastnictví zdroje nebo přesunout aplikační logiku mimo samotnou obsluhu požadavku. Začlenění uživatelských údajů do modelu je řízeno pomocí rozhraní, která požadavky implementují. Při spuštění filtru v pipeline je možné pomocí reflexe získat informace o implementovaných rozhraních z metadat datového typu a následně transformovat model požadavku na model, který dané rozhraní

implementuje, s možností přiřazení hodnot, jako jsou emailová adresa nebo přihlašovací jméno uživatele.

```
public interface IRequestWithUserName
{
    public string? UserName { get; set; }
}
```

Ilustraci tohoto procesu lze nalézt níže. Pokud uživatel není přihlášen, proces vykonávání dalších filtrů v pipeline nebo samotnou obsluhu požadavku přerušíme.

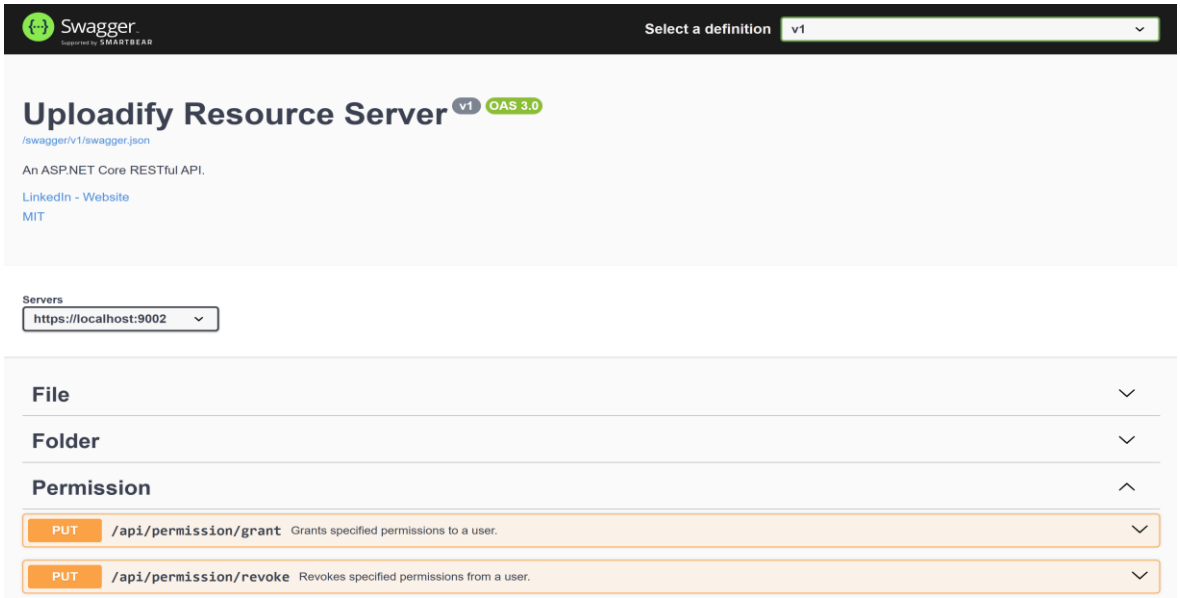
```
public class AuthorizationPipelineBehaviour<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse> where TRequest : notnull
                                             where TResponse : BaseResponse
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public async Task<TResponse> Handle(TRequest request,
        RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)
    {
        ClaimsPrincipal? principal = _httpContextAccessor.HttpContext?.User;
        if (request is IRequestWithEmail requestWithEmail)
        {
            <CAST REQUEST MODEL AND SET CURRENT USER EMAIL>
        }

        if (request is IRequestWithUserName requestWithUserName)
        {
            requestWithUserName.UserName =
                principal?.FindFirst(OpenIddictConstants.Claims.Name)?.Value;

            if (IsNullOrWhiteSpace(requestWithUserName.UserName))
            {
                var response = Activator.CreateInstance<TResponse>();

                response.Status = Unauthorized;
                response.Failure = new()
                {
                    UserFriendlyMessage = Translations.RequestStatuses.Unauthorized,
                    Exception = new UnauthorizedException(Empty, Empty, Empty)
                };

                return response;
            }
        }

        return await next();
    }
}
```

7.5 Swagger a generování dokumentace

Framework pro generování dokumentace koncových bodů, modelů jejich dotazů a odpovědí využívá průzkumníka koncových bodů, XML dokumentaci a typovou reflexi. Součástí balíčků a nástrojů „NSwag.AspnetCore“ a „Microsoft.AspNetCore.OpenAPI“ jsou služby pro její realizaci.

Přidání průzkumníka koncových bodů umožní zmapovat řadiče a jejich koncové body, jak můžeme vidět na obrázku níže. Mapovány jsou informace jako název řadiče a koncového bodu, tělo požadavku a odpovědi, parametry a dotazovací řetězec URL, hlavičky a typ požadavku nebo odpovědi. Průzkumníka koncových bodů registrujeme rozšiřující metodou kontejneru pro vkládání závislostí „AddEndpointsApiExplorer“, která registruje služby pro popis koncových bodů a jejich skupin, nebo služby, které umožňují jejich detailnější dokumentaci dekorováním atributy. Pro nastavení a generování dokumentace registrujeme služby rozšiřující metodou jako na obrázku níže.

```
services.AddEndpointsApiExplorer();
services.AddOpenApiDocument(options =>
{
    options.PostProcess = document => document.Info = new()
    {
        Version = "v1",
        Title = "Uploadify REST API",
        Description = "An ASP.NET Core REST API.",
        Contact = new()
        {
            Name = "LinkedIn",
            Url = "https://www.linkedin.com/in/marek-ol%C5%A1%C3%A1k-1715b724a/"
        },
        License = new()
        {
            Name = "MIT",
            Url = "https://en.wikipedia.org/wiki/MIT_License"
        }
    };
});
```



Obrázek 3: Dokumentace REST API generovaná frameworkem NSwag.

Aby po spuštění aplikace generovala dokumentaci, přidáme do pipeline pro obsluhu dotazů filtry, které zveřejní dokumentaci na výchozí směrovací adrese „/swagger“. Výsledkem generování dokumentace je JSON soubor, který obsahuje zmapované koncové body, modely dotazů a požadavků ve strukturované a standardizované formě. Framework NSwag umožňuje prezentovat tento soubor v podobě interaktivního uživatelského rozhraní, které zpřístupňuje dokumentaci v čitelné podobě a umožňuje testovat aplikační rozhraní prostřednictvím integrovaného klienta.

```
if (application.Environment.IsDevelopment())
{
    application.UseOpenApi();
    application.UseSwaggerUi(options =>
    {
        options.DocumentTitle = "Uploadify Resource Server";
    });

    application.UseReDoc(options => options.Path = "/redoc");
}
```

Ve výchozím nastavení generátor nezačlení XML dokumentaci modelů, koncových bodů a řadičů, a proto bylo rozšířeno nastavení souboru projektového řešení „*.csproj“, ve kterém byla přidána skupina vlastností pro aktivaci této funkcionality.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
        <InvariantGlobalization>true</InvariantGlobalization>
        <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    </PropertyGroup>

    <PROPERTIES AND ITEM GROUPS>

    <PropertyGroup>
        <GenerateDocumentationFile>true</GenerateDocumentationFile>
        <NoWarn>$(NoWarn);1591</NoWarn>
    </PropertyGroup>

</Project>
```

7.6 Pravidelné úlohy v pozadí

Framework a nástroj Hangfire spravuje, monitoruje a vytváří nebo aktualizuje pravidelné úlohy v pozadí prototypu aplikace. Pro životosprávu úloh v pozadí využívá Hangfire připojení ke zdroji dat a ve srovnání s frameworkem OpenIddict nemusíme rozšířit kontext databáze nebo přidat metody pro mapování modelů na entity databáze. Modely, nastavení a

metody pro integraci jsou součástí frameworku, který při registraci závislostí vyžaduje nastavení typu kontextu databáze.

```
public static IServiceCollection AddBackgroundTasks(
    this IServiceCollection services, SystemSettings settings)
{
    services.AddHangfire(options => options
        .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
        .UseSimpleAssemblyNameTypeSerializer()
        .UseRecommendedSerializerSettings()
        .UsePostgreSqlStorage(new NpgsqlConnectionStringBuilder
        {
            Username = settings.Database.Username,
            Password = settings.Database.Password,
            Host = settings.Database.Host,
            Database = settings.Database.Database,
            Pooling = settings.Database.Pooling,
            Port = settings.Database.Port
        }.ConnectionString));

    services.AddTransient<DeleteFilesJob>();

    return services.AddHangfireServer();
}
```

Před spuštěním aplikace se obdobně jako při generaci dokumentace registruje filtr v pipeline obsluhy dotazů, který přidává uživatelské rozhrání pro monitorování, spouštění a správu úloh v pozadí a naslouchá na směrovací adrese „/hangfire“. Následovně se vytváří instance úloh v pozadí, jejichž životnost odpovídá době jejich dokončení. Pro registraci úloh prototyp používá pomocné statické metody pro vytváření a aktualizaci úloh. Metody jako vstup přijímají název úlohy, výraz pro její spuštění a plánování ve formátu cron. Výrazy cron používají stejnou syntaxi jako plánovače spouštění skriptů operačního systému Linux, kde jako vstup předáme formátovaný textový řetězec nebo přednastavenou konstantu pro známé časové milníky.

```
public static IApplicationBuilder UseBackgroundTasks(
    this IApplicationBuilder builder,
    IServiceProvider services,
    SystemSettings settings)
{
    builder.UseHangfireDashboard("/hangfire", new()
    {
        DashboardTitle = "Uploadify task manager",
        Authorization =
        [
            new BasicAuthAuthorizationFilter(new()
            {
                Users = [new BasicAuthAuthorizationUser
                {
                    Login = settings.Hangfire.Login,
                    PasswordClear = settings.Hangfire.Password
                }]
            })
        ]
    });

    using var scope = services.CreateScope();
    var job =
        scope.ServiceProvider.GetRequiredService<DeleteFilesJob>() as IJob;

    RecurringJob.AddOrUpdate(nameof(DeleteFilesJob),
        () => job.Execute(),
        Cron.Hourly());

    return builder;
}
```

7.6.1 Úloha v pozadí pro mazání souborů

Prototyp aplikace používá politiku mazání záznamu v databází soft-delete, kde příkazy aktualizují příznak pro smazání entity včetně datumu poslední změny. Úloha v pozadí kontroluje příznak pro záznamy starší 7 dnů od datumu poslední změny.

```
public class DeleteFilesJob : IJob
{
    public async Task Execute()
    {
        try
        {
            await DeleteFiles();
            await DeleteFolders();
        }
        catch (Exception exception)
        {
            Console.WriteLine(exception);
            _logger.LogError($"…");
        }
    }
}
```

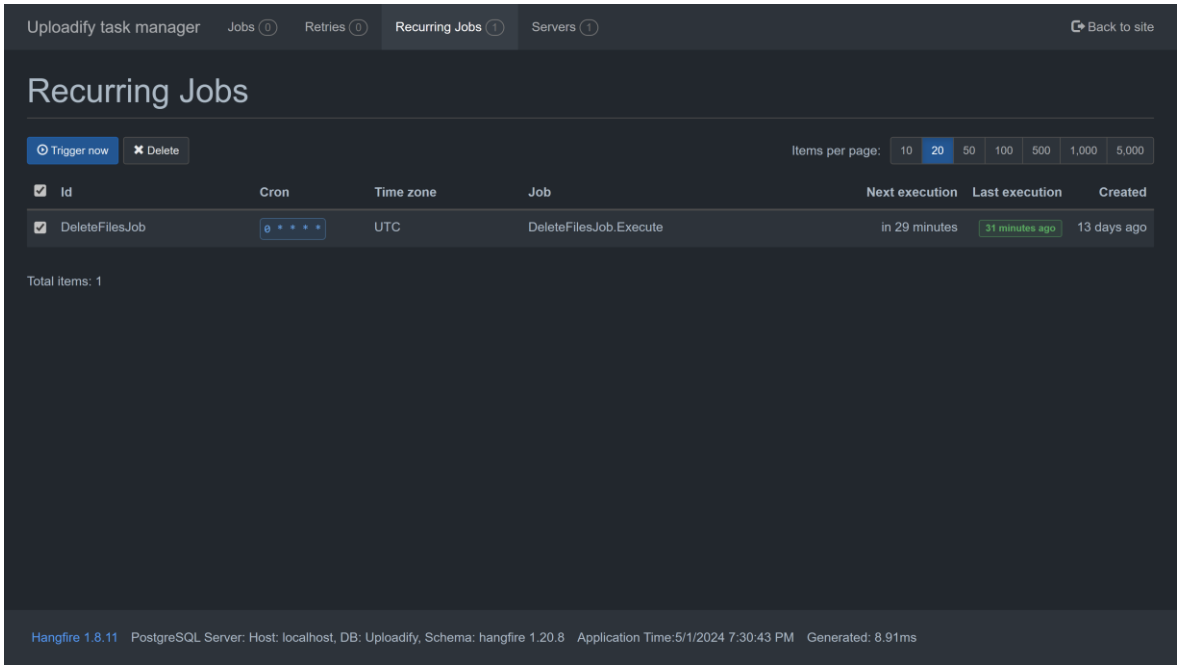
Pro smazání záznamu složek zavoláme jednoduchý dotaz na databázi ve srovnání se soubory, kde odebíráme fyzické soubory souborového systému i záznamy v databázi. Při mazání souborů uvažujeme následující scénáře

- Soubor byl úspěšně odebrán ze souborového systému a záznam smazán v databázi.
- Soubor nebyl úspěšně odebrán ze souborového systému.
- Soubor byl úspěšně odebrán ze souborového systému a záznam nebyl smazán v databázi.

První a hlavní scénář představuje předpokládaný výsledek operace a zbylé scénáře možné komplikace, které mohou nastat během vytížení infrastruktury, která znemožní komunikaci s databází, nebo nedokončení operace. Prototyp aplikace řeší druhý i třetí případ použitím transakcí. Pokud nedojde k odebrání fyzického souboru ze souborového systému, pak budou navráceny provedené změny v databázi během odchycení a zpracování výjimky, tj. smazaný záznam bude obnoven a při dalším spuštění úlohy se provede další pokus o jeho smazání. Nebude-li smazán záznam v databázi, pak úloha nepostupuje na odebrání fyzického souboru.

```
private async Task DeleteFiles()
{
    var files = new List<File>();
    await foreach (var file in DeletedFilesQuery(_context)) files.Add(file);

    foreach (var file in files)
    {
        var executionStrategy = _context.Database.CreateExecutionStrategy();

        await ExecutionStrategyExtensions.ExecuteAsync(
            executionStrategy, async () =>
            {
                await using var transaction =
                    await _context.Database.BeginTransactionAsync(default);

                try
                {
                    if (!System.IO.File.Exists(file.Location)) return;

                    await _context.DeleteEntity(file);
                    await _context.SaveChangesAsync();

                    System.IO.File.Delete(file.Location);

                    await transaction.CommitAsync();
                }
                catch (Exception exception)
                {
                    await transaction.RollbackAsync();
                }
        }).ConfigureAwait(false);
    }
}
```

Na obrázku níže můžeme vidět snímek z přehledu úloh v pozadí panelu pro vývojáře Hangfire. Registrována je pouze úloha pro mazání složek a souborů, pro kterou je nastaveno pravidelné spuštění každou hodinu. Dále můžeme vidět datum posledního a dalšího spuštění. Hangfire pro umožňuje manuálně spouštět vybrané úlohy, čímž přispívá ke zjednodušení testování komponent systému.



Obrázek 4: Plánovač a přehled úloh v Hangfire.

7.7 Příkaz pro nahrání souboru

Po provolání koncového bodu a obdržení dotazu pro nahrání souboru se vytvoří záznam v databázi a překopíruje obsah streamu do fyzického souboru souborového systému. Server odpoví na dotaz pro nahrání souboru stavovým kódem, který indikuje jeho vytvoření, a přehledem informací o souboru.

7.7.1 Model dotazu a odpovědi

Model požadavku obsahuje informace o souboru jako jeho název, velikost a přípona nebo stream obsahu souboru. Umístění souboru a jeho pojmenování v souborovém systému se nepřebírá z dotazu. Název fyzického souboru používá náhodně vygenerovaný GUID bez přípony, kterou není nutné nastavit a její hlavním účelem je zpracování a otevření souboru výchozí aplikací operačním systémem. Součástí modelu požadavku je přihlašovací jméno uživatele, které doplní z kontextu přihlášeného uživatele autorizační pipeline v MediatR. Model požadavku odpovídá modelu příkazu pro nahrání souboru.

```
public class CreateFileCommand : IBaseRequest<CreateFileCommandResponse>,
    ICommand<CreateFileCommandResponse>,
    IRequestWithUserName
{
    [JsonIgnore]
    public string? UserName { get; set; }

    public int? FolderId { get; set; }
    public int? CategoryId { get; set; }
    public IFormFile? File { get; set; }
}
```

Odpověď na dotaz zahrnuje stavový kód, který indikuje stav obsluhy příkazu, chybovou hlášku s uživatelsky přívětivou hláškou nebo kódem pro překlad a validační chyby vázané na model dotazu. V případě úspěšného provolání koncového bodu přiloží obsluha příkazu pro nahrání souboru přehled informací o vytvořeném souboru. Model obsahuje prázdný konstruktor, který používá validační pipeline pro vytvoření nové instance a nastavení chybové zprávy.

```
public class CreateFileCommandResponse : BaseResponse
{
    <CONSTRUCTORS>

    public FileOverview? File { get; set; }
}

public class FileOverview
{
    public int FileID { get; set; }
    public int FolderID { get; set; }
    public string Name { get; set; } = string.Empty;
    public long Size { get; set; }
    public bool IsPublic { get; set; }
    public string Extension { get; set; } = string.Empty;
    public string MimeType { get; set; } = string.Empty;
    public string? CreatedBy { get; set; }
    public string? UpdatedBy { get; set; }
    public DateTime DateCreated { get; set; }
    public DateTime DateUpdated { get; set; }
}
```

7.7.2 Validátor příkazu

Před obsluhou příkazu se spouští validační filtr v MediatR pipeline a jeho validátor, který validuje model příkazu a jeho hodnoty, kde pro neplatný model nastaví validátor chybové hlášení, kódy k překladu vázané na neplatný údaj. Pro implementaci validátorů používá prototyp knihovnu „FluentValidation“ a rozšiřující metody pro seskupení validačních chyb podle vlastnosti modelu.

```
public class CreateFileCommandValidator : AbstractValidator<CreateFileCommand>
{
    public CreateFileCommandValidator()
    {
        RuleFor(command => command.File)
            .NotNull()
            .WithMessage(Translations.Validations.FileRequired);

        When(command => command.File != null, () =>
        {
            RuleFor(command => command.File.Length)
                .GreaterThan(0)
                .WithMessage(Translations.Validations.FileLengthRequired)
                .LessThanOrEqualTo(104857600L)
                .WithMessage(Translations.Validations.FileLengthTooLarge);

            RuleFor(command => command.File.ContentType)
                .NotEmpty()
                .WithMessage(Translations.Validations.FileContentTypeRequired);

            RuleFor(command => command.File.Name)
                .NotEmpty()
                .WithMessage(Translations.Validations.FileNameRequired)
                .MaximumLength(256)
                .WithMessage(Translations.Validations.FileNameMaxLength)
                .Must(filename => !Path.HasExtension(filename))
                .WithMessage(Translations.Validations.FileNameExtensionRequired);
        });
    }
}
```

7.7.3 Obsluha příkazu

Po validaci modelu příkazu následuje jeho obsluha, která před vytvořením záznamu v databází a fyzického souboru souborového systému validuje oprávnění uživatele a jeho vlastnictví adresáře, ve kterém bude soubor založen. Při zpracování příkazu obsluha volá a řetězí další dotazy v MediatR pro získání informací o adresáři nebo uživateli, proto vkládá konstruktor závislosti služeb jako kontext databáze nebo odesílatele příkazů a dotazů v MediatR.

```
public class CreateFileCommandHandler
    : ICommandHandler<CreateFileCommand, CreateFileCommandResponse>
{
    private readonly DataContext _context;
    private readonly ISender _sender;

    public CreateFileCommandHandler(DataContext context, ISender sender)
    {
        _context = context;
        _sender = sender;
    }

    public async Task<CreateFileCommandResponse> Handle(
        CreateFileCommand request,
        CancellationToken cancellationToken)
    {
        var userResponse = await _sender.Send(
            new GetUserQuery(request.UserName));

        if (userResponse is not { Status: Ok, User: not null })
        {
            return new(userResponse);
        }

        <PROCESS COMMAND>
    }
}
```

Pro získání názvu souboru parsuje obsluha hlavičku požadavku, která obsahuje jeho metadata. Následně se vygeneruje název, ze kterého bude odvozena jeho absolutní adresa, a který použije souborový systém při zápisu. Při vytváření záznamu v databázi používá obsluha transakce, které umožní navrátit provedené změny před jejich odevzdáním. Transakce ošetřuje scénář, při kterém nedojde k dokončení procesu kopie obsahu streamu do fyzického souboru souborového systému. Selhání tohoto procesu doprovází vyvolání výjimky, kterou obsluha odchytí a navrátí provedené změny v databázi do původního stavu. Pokud selže provedení transakce, pak obsluha zkontroluje stav fyzického souboru a případně ho odebere ze souborového systému.

```
await ExecutionStrategyExtensions.ExecuteAsync(executionStrategy, async () =>
{
    await using var transaction =
        await _context.Database.BeginTransactionAsync(cancellationToken: default);

    try
    {
        file.FolderId = request.FolderId.Value;
        file.CategoryId = request.CategoryId;
        file.Size = request.File.Length;
        file.Extension = Path.GetExtension(filename);
        file.MimeType = request.File.ContentType;
        file.Location = path;

        await _context.AddAsync(file, cancellationToken: default);
        await _context.SaveChangesAsync(cancellationToken: default);

        await using (var stream = new FileStream(path, FileMode.Create))
        {
            await request.File.CopyToAsync(stream, cancellationToken);
        }

        await transaction.CommitAsync(cancellationToken: default);
    }
    catch (Exception exception)
    {
        await transaction.RollbackAsync(cancellationToken: default);

        if (System.IO.File.Exists(path))
        {
            System.IO.File.Delete(path);
        }

        file = null;
    }
}).ConfigureAwait(false);
```

Na závěr obsluha mapuje informace na model odpovědi určené pro přenos a vrací stavovou hlášku, která indikuje úspěšné nahrání souboru.

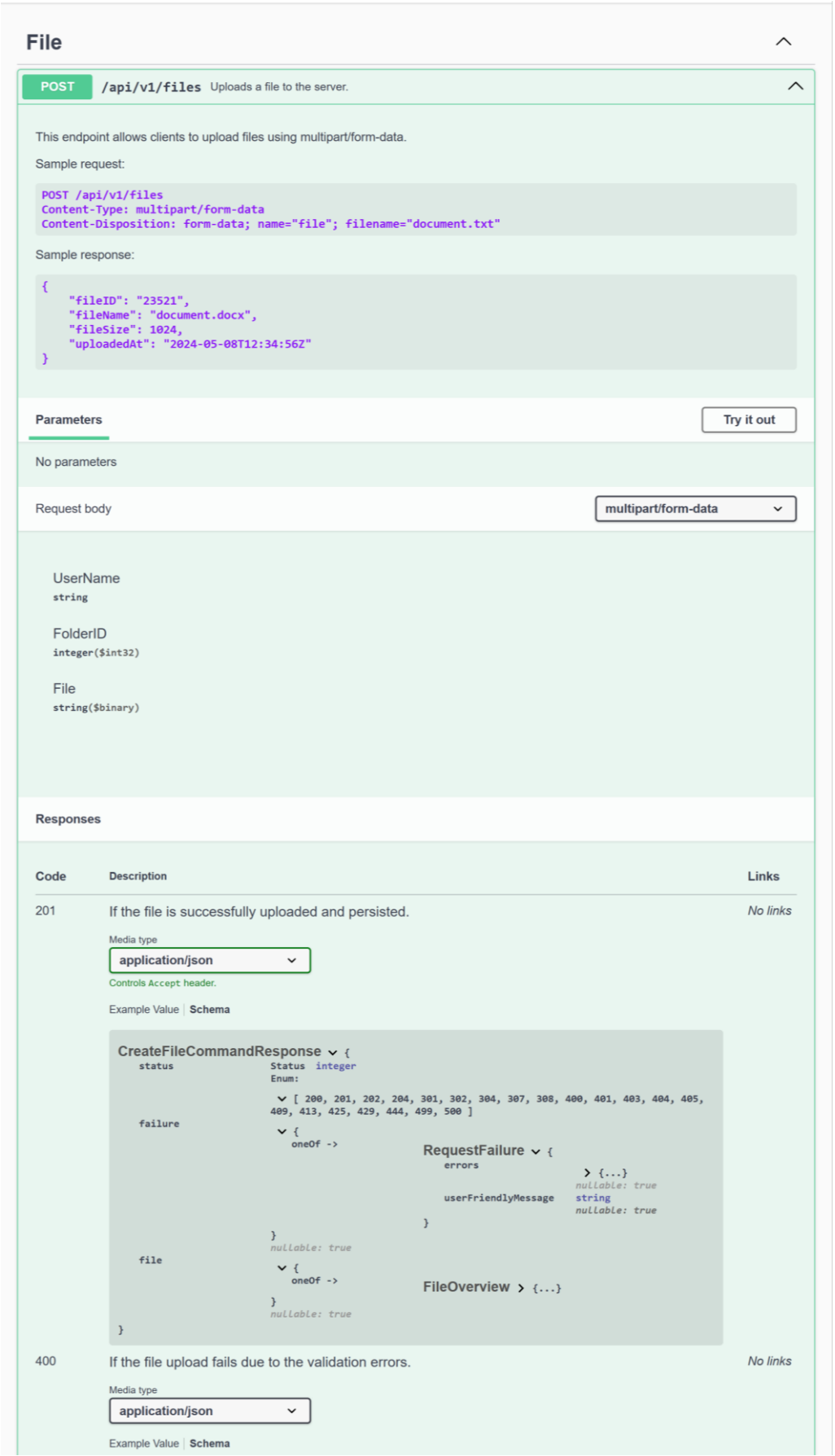
7.7.4 Koncový bod

Koncový bod dekorujeme atributy, které přidají filtry do pipeline zpracování požadavků pro upřesnění a poskytnutí dodatečných informací jako datový typ modelu odpovědi na dotaz, návratové stavové kódy a podporované typy obsahu odpovědi. Prototyp aplikace v první iterace podporuje pouze typ odpovědi JSON. Poskytnutí těchto informací přispívá k pokrytí zdrojového kódu dokumentací REST API.

```
[HttpPost]
[ProducesResponseType(
    typeof(CreateFileCommandResponse), Status201Created, Json)]
[ProducesResponseType(
    typeof(CreateFileCommandResponse), Status400BadRequest, Json)]
[ProducesResponseType(
    typeof(CreateFileCommandResponse), Status401Unauthorized, Json)]
[ProducesResponseType(
    typeof(CreateFileCommandResponse), Status500InternalServerError, Json)]
public async Task<IActionResult> Upload([FromForm] CreateFileCommand command,
    CancellationToken cancellationToken) =>
        ConvertToActionResult(await Mediator.Send(command, cancellationToken));
```

7.7.5 Dokumentace

V dokumentaci nalezneme informace o koncovém bodě, jeho modelu dotazu, odpovědi a hlaviček, jak můžeme vidět na obrázku níže.



Obrázek 5: Dokumentace Swagger koncového bodu pro nahrání souboru.

8 IMPLEMENTACE KLIENTA

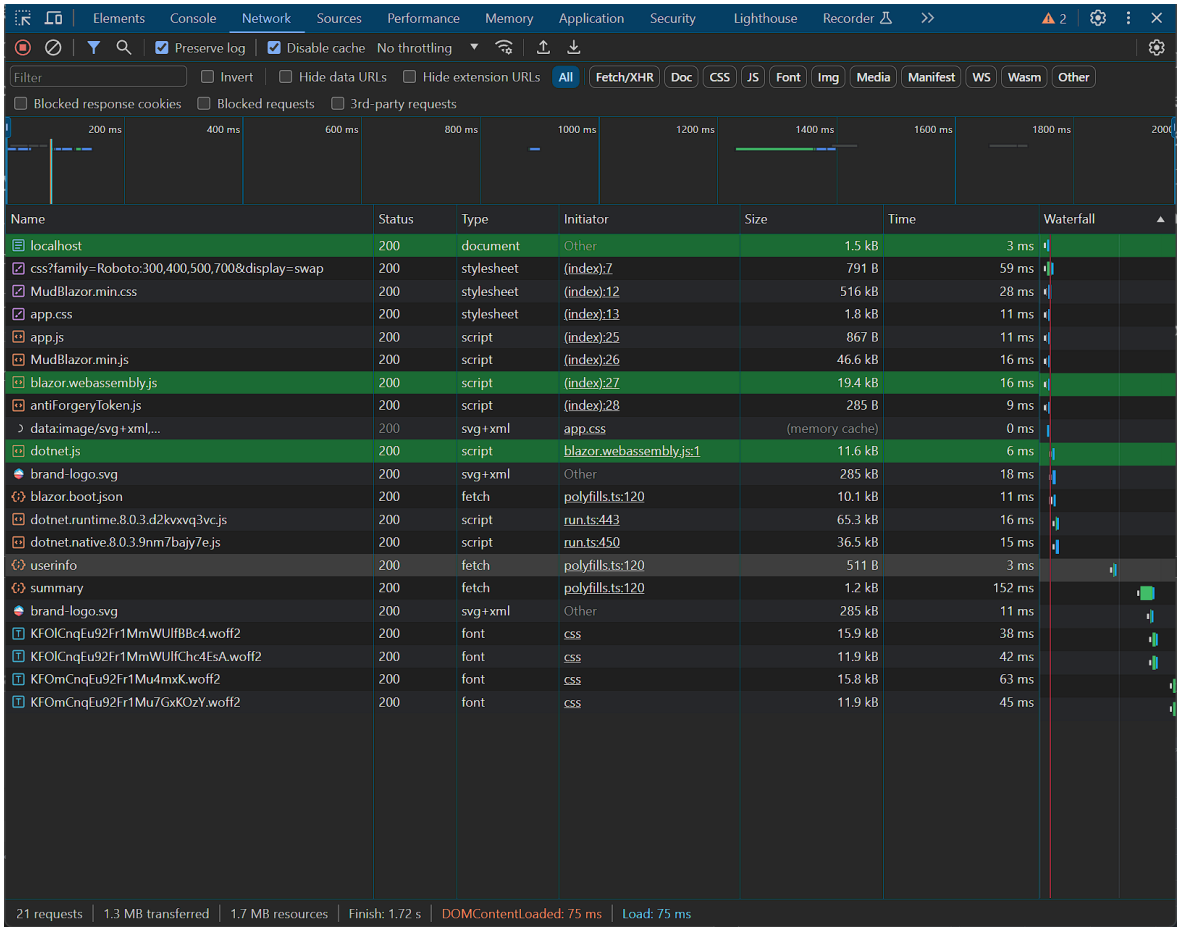
Pro začlenění projektu klienta se využívá šablona hostované verze modelu nasazení Blazor WebAssembly. Tato šablona začlení do struktury řešení webovou aplikaci společně s aplikačním rozhraním, na kterém je webová aplikace závislá z důvodu distribuce jejích statických souborů. Aby se předešlo potenciálním kompatibilním problémům s balíčky ve webové aplikaci, je do prototypu řešení začleněn oddělený cibulovitý model bez datové vrstvy. Místo datové vrstvy je implementována nezávislá integrační vrstva, která obsahuje služby pro integraci s poskytovatelem identity a pro sdílení a manipulaci se zdroji.

Hlavní účel webové aplikace je prezentace dat a zobrazení uživatelsky přívětivého rozhraní pro jejich manipulaci. Aplikační rozhraní zajišťuje bezpečné uchování citlivých informací jako tajemství a klíče pro komunikaci s dalšími službami nebo uchování stavu přihlášeného uživatele.

8.1 Proces zavedení klienta ve webovém prohlížeči

Na snímku nástrojů pro vývojáře níže můžeme vidět natažení a spuštění klienta. Po stažení JS a CSS závislostí HTML dokumentu můžeme vidět natažení zavaděče pro Blazor WASM „blazor.webassembly.js“. Zavaděč stáhne ze serveru manifest „blazor.boot.json“, který obsahuje závislosti klienta jako zdrojové soubory nebo prostředí běhu Mono přeložené do WebAssembly. JavaScript knihovny „dotnet.js“, „dotnet.runtime.js“ a „dotnet.native.js“ umožňují prostředí běhu manipulovat s DOM nebo volat JavaScript funkce z .NET a opačně prostřednictvím API sandboxu.

Z obrázku níže lze dále odvodit rychlost zavedení aplikace a její spuštění. Protože klient vyžaduje při prvotním spuštění stažení závislostí a následně zavedení aplikace, ve srovnání s frameworky jako React nebo Angular Blazor WebAssembly nedoručí klient uživateli požadovaný obsah okamžitě. Rychlost a stopa zanechaná v paměti prohlížeče ovšem kompenzují nedostatky a proces zavedení webové aplikace je značně rychlejší po prvotním zavedení klienta.



Obrázek 6: Snímek záložky Síť v nástrojích pro vývojáře ve webovém prohlížeči po načtení statických souborů klienta a natažení přehledu souborů.

8.2 Integrace poskytovatele identity

Aby bylo klientovi umožněno získat přístup k identitě uživatele, registrujeme klienta v seznamu aplikací poskytovatele identity. Zároveň jsou do webové aplikace a aplikačního rozhraní přidány služby, které vyžadují ověření stavu přihlášení uživatele, jeho prodloužení a uchování při volání aplikačního rozhraní pro manipulaci a sdílení zdrojů nebo při přístupu k webové stránce.

Při registraci klienta je nastaven jedinečný identifikátor společně s tajemstvím, které je konfigurováno i na straně webového aplikačního rozhraní klienta. Poskytovatel identity vyžaduje od uživatele udělení explicitního svolení k poskytnutí přístupu k jeho citlivým informacím. Tyto informace lze specifikovat v seznamu oprávnění nastavení klienta a dále upravit na úrovni aplikační logiky poskytovatele identity. Klient pro autorizaci využívá schéma Authorization Code Flow s PKCE, které je konfigurováno doplněním seznamu oprávnění a požadavků pro autorizaci. Klíčovou součástí nastavení je definice koncových bodů, které

poskytovatel identity zpřístupní klientovi, a konfigurace návratové adresy pro přesměrování odpovědí zpět do klientské aplikace.

```
await applicationManager.CreateAsync(new()
{
    ClientId = settings.Client.ID,
    ClientSecret = settings.Client.Secret,
    DisplayName = settings.Client.DisplayName,
    ClientType = OpenIddictConstants.ClientTypes.Confidential,
    ConsentType = OpenIddictConstants.ConsentTypes.Explicit,
    PostLogoutRedirectUris =
    {
        new Uri(settings.Client.PostLogoutRedirectUri)
    },
    RedirectUris =
    {
        new Uri(settings.Client.RedirectUri)
    },
    Permissions =
    {
        OpenIddictConstants.Permissions.Endpoints.Authorization,
        OpenIddictConstants.Permissions.Endpoints.Logout,
        OpenIddictConstants.Permissions.Endpoints.Token,
        OpenIddictConstants.Permissions.GrantTypes.AuthorizationCode,
        OpenIddictConstants.Permissions.ResponseTypes.Code,
        OpenIddictConstants.Permissions.Prefixes.Scope + Scopes.Name,
        OpenIddictConstants.Permissions.Prefixes.Scope + Scopes.Email,
        OpenIddictConstants.Permissions.Prefixes.Scope + Scopes.Phone,
        OpenIddictConstants.Permissions.Prefixes.Scope + Scopes.Roles,
        OpenIddictConstants.Permissions.Prefixes.Scope + Scopes.Api
    },
    Requirements =
    {
        OpenIddictConstants.Requirements.Features.ProofKeyForCodeExchange
    }
});
```

8.2.1 Nastavení komunikace mezi klientem a poskytovatelem identity

Pro získání přístupového tokenu, tokenu identity nebo tokenu pro obnovení přístupového tokenu se využívá balíček OpenIddict pro klientské aplikace, který umožňuje nastavit komunikaci mezi poskytovatelem identity a klientem, validovat tokeny a spravovat jejich životnost.

Prototyp pro šifrování komunikace využívá certifikáty určené pro vývojové účely, které je před spuštěním a nasazením aplikace nutné vyměnit za produkční certifikáty. Konfigurace klienta upřesňuje autorizační schéma, identifikátor a tajemství klienta, a také výčet informací o uživateli, které budou poskytovatelem identity zahrnuty do přístupového tokenu nebo tokenu identity. Je vyžadováno, aby hodnoty a konstanty použité v nastavení klienta odpovídaly parametrům poskytovatele identity. Aby bylo možné klientovi zpracovat přesměrování

zpět od poskytovatele identity, je nezbytné nastavit koncové body řadiče pro obsluhu těchto požadavků.

```
services.AddOpenIddict()
    .AddCore(options =>
    {
        options.UseEntityFrameworkCore()
            .UseDbContext<DataContext>();

        options.UseQuartz();
    })
    .AddClient(options =>
    {
        options.AllowAuthorizationCodeFlow();

        options.AddDevelopmentEncryptionCertificate()
            .AddDevelopmentSigningCertificate();

        options.UseAspNetCore()
            .EnableStatusCodePagesIntegration()
            .EnableRedirectionEndpointPassthrough()
            .EnablePostLogoutRedirectionEndpointPassthrough();

        options.UseSystemNetHttp()
            .SetProductInformation(typeof(Program).Assembly);

        options.AddRegistration(new()
        {
            Issuer = new(settings.IdentityProvider.Issuer, UriKind.Absolute),

            ClientId = settings.Client.ID,
            ClientSecret = settings.Client.Secret,
            Scopes =
            {
                Scopes.Name,
                Scopes.Email,
                Scopes.Phone,
                Scopes.Roles,
                Scopes.Api
            },

            RedirectUri = new("callback/login/local", UriKind.Relative),
            PostLogoutRedirectUri = new("callback/logout/local", UriKind.Relative)
        });
    });
```

8.2.2 Koncový bod pro přihlášení uživatele

Po úspěšném přihlášení nebo odhlášení je uživatel přesměrován poskytovatelem identity zpět na klienta. V případě přihlášení odpověď obsahuje přístupový token, token pro obnovení a token identity, které jsou uloženy v kontextu přihlášeného uživatele. Tento kontext, vytvořený po přijetí odpovědi, využívá OpenIddict autentizační schéma pro komunikaci webového aplikačního rozhraní s poskytovatelem identity. Koncový bod pro zpracování požadavku na přihlášení uživatele, na který poskytovatel identity zašle požadavek, by podle doporučení neměl být umístěn na stejné adrese jako koncový bod pro přihlášení u

poskytovatele identity. Proto je v adrese zahrnuta přípona pro jejich rozlišení. Po přijetí požadavku tento koncový bod nejen vytvoří kontext přihlášeného uživatele, který využívá externí autentizační schéma pro komunikaci mezi webovou aplikací a aplikačním rozhraním, ale také zmapuje informace o uživateli, včetně tokenů, do Session Cookie.

```
[HttpGet("~/callback/login/{provider}"), HttpPost("~/callback/login/{provider}")]
[IgnoreAntiforgeryToken]
public async Task<ActionResult> LogInCallback()
{
    <RETRIEVE CURRENT USER>
    <MAP CURRENT USER CLAIMS TO CREATE SESSION COOKIE AND STORE TOKENS>
    <RETURN SIGN-IN RESPONSE>
}
```

8.2.3 Koncový bod pro odhlášení uživatele

Po odhlášení uživatele zneplatní BFF kontext přihlášeného uživatele pro externí i OpenID Connect schéma. BFF následovně přesměruje uživatele na domovskou stránku klienta, kde webová aplikace ověří stav přihlášení uživatele a přesměruje ho na poskytovatele identity pro novou autentizaci.

```
[HttpPost("~/logout"), ValidateAntiForgeryToken]
public async Task<ActionResult> LogOut(string returnUrl)
{
    <RETRIVE AND SIGN OUT CURRENT USER>
}
```

8.2.4 Aktualizace stavu přihlášení uživatele

Autentizaci pro Blazor obsluhuje služba „AuthenticationStateProvider“ namísto služeb jako „HttpContext“, která vzniká po přijetí požadavku na koncový bod řadiče, a „HttpContextAccessor“, která umožňuje přistoupit ke kontextu mimo řadič a koncový bod, v případě MVC aplikací. Komponenty jako „CascadingAuthenticationState“ nebo „AuthorizeView“ a atributy pro autorizaci využívají této služby pro přístup ke stavu přihlášení uživatele. Abychom přizpůsobili logiku pro aktualizaci a nastavení stavu přihlášení uživatele, prototyp aplikace registruje vlastní službu, která dědí a přetěžuje metody této služby.

Služba pro aktualizaci a přístupu ke stavu přihlášení uživatele přetěžuje metodu pro získání stavu autentizace. Tato metoda se pokusí získat stav z cache a v případě jeho nenalezení provolává koncový bod aplikačního rozhraní pro natažení informací o uživateli, které použije pro vytvoření nového stavu. Součástí prototypu aplikace není kontrola stavu přihlášení uživatele s periodou, kde ve výchozím nastavení probíhá autentizace při směrování v aplikaci a volání služby komponentami nebo atributy pro ověření oprávnění uživatele.

```
public class HostAuthenticationStateProvider : AuthenticationStateProvider
{
    public override async Task<AuthenticationState>
        GetAuthenticationStateAsync() => new(await GetUser(useCache: true));

    public void SignIn(string customReturnUrl = null)
    {
        <REDIRECTS CLIENT TO IDENTITY PROVIDER TO AUTHENTICATE CURRENT USER>
    }

    private async ValueTask<ClaimsPrincipal> GetUser(bool useCache = false)
    {
        <RETRIEVES CACHED AUTHENTICATION STATE OR
         WHEN CACHE EXPIRED THEN RETRIEVES CURRENT USER INFORMATION AND
         CREATES NEW AUTHENTICATION STATE>
    }


    private async Task<ClaimsPrincipal> FetchUser()
    {
        <RETRIEVES CURRENT USER INFORMATION USING BFF USERINFO ENDPOINT>
    }
}
```

Metoda pro přihlášení uživatele se volá při směrování v aplikaci, která po obdržení dotazu upozorní na jeho stav přihlášení a přesměruje uživatele na poskytovatele identity pro jeho ověření totožnosti. Tento proces probíhá v autorizační vrstvě pipeline, která využívá obsluhu pro autorizaci uživatele „AuthorizedHandler“.

```
public class AuthorizedHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        var authState =
            await _authenticationStateProvider.GetAuthenticationStateAsync();

        HttpResponseMessage responseMessage;
        if (!authState.User.Identity.IsAuthenticated)
        {
            responseMessage = new(HttpStatusCode.Unauthorized);
        }
        else
        {
            responseMessage = await base.SendAsync(request, cancellationToken);
        }

        if (responseMessage.StatusCode == HttpStatusCode.Unauthorized)
        {
            _authenticationStateProvider.SignIn();
        }

        return responseMessage;
    }
}
```

8.2.5 Ověření stavu přihlášení během směrování

Ve výchozím nastavení kontroluje klient stav přihlášení uživatele během směrování nebo přístupu k identitě uživatele prostřednictvím služby „AuthenticationStateProvider“. Pokud uživateli vyprší platnost tokenů nebo relace, pak se při směrování namísto obsahu vykreslí komponenta, která během inicializace přesměruje klienta na poskytovatele identity.

```
@code {
    protected override void OnInitialized()
    {
        NavigationManager.NavigateTo(ApiRoutes.Login, true);
    }
}
```

8.2.6 Ověření oprávnění uživatele během vykreslování komponenty

Pro rozhodnutí o zobrazení obsahu uživateli nebo omezení jeho přístupu se implementuje komponenta, která zapouzdřuje dceřiné komponenty. Tyto komponenty se vykreslují pouze v případě, že uživateli jsou přidělena požadovaná oprávnění. Komponenta je implementována analogicky k atributu pro autorizaci uživatele, který vychází z jeho oprávnění. Pro její implementaci se přidává komponentní knihovna, na které závisí webová aplikace klienta. Tato komponenta je založena na stávající komponentě pro autorizaci uživatele, která je založena na rolích, a využívá politiku autorizačních požadavků implementovaných v předchozí kapitole. Aby bylo umožněno znovu použití této logiky ve webové aplikaci klienta, jsou služby registrovány obdobně jako pro REST API. Vzhledem k tomu, že komponenta pro autorizaci uživatele využívá politiky a její požadavky, je pro implementaci autorizace založené na oprávněních dostačující nastavit hodnotu oprávnění.

```
public class PermissionView : AuthorizeView
{
    [Parameter]
    public Permission Permission
    {
        get => IsNullOrWhiteSpace(Policy)
            ? Permission.None
            : PolicyNameHelpers.GetPermissionsFrom(Policy);
        set => Policy = PolicyNameHelpers.GetPolicyNameFor(value);
    }
}
```

8.2.7 Komponenta pro odhlášení uživatele

Komponenta pro odhlášení uživatele je součástí menu, které je zapouzdřeno komponentou pro ověření stavu přihlášení uživatele. Pro zachování vizuální stránky prototypu aplikace je položka menu začleněna ve formuláři pro odhlášení uživatele jako dceřiná komponenta

tlačítka. Zapouzdření položky menu tlačítkem překryje event, který se volá po jeho kliknutí, a umožní odeslání formuláře na aplikační rozhraní, kde koncový bod pro odhlášení uživatele zneplatní relaci a odhlásí uživatele na straně klienta i poskytovatele identity.

```
<PermissionView Context="authState">
    <Authorized>
        <MudMenu …>
            <ActivatorContent>…</ActivatorContent>
            <ChildContent>
                <MudMenuItem …>…</MudMenuItem>
                <MudMenuItem …>…</MudMenuItem>
                <MudMenuItem …>…</MudMenuItem>
                <form action="@ApiRoutes.Logout" method="post">
                    <AntiForgeryTokenInput/>
                    <button class="mud-width-full" type="submit">
                        <MudMenuItem AutoClose="false"
                                     Icon="@Icons.Material.Rounded.Logout"
                                     IconColor="@Color.Primary">
                            <MudText>
                                @Localizer[Translations….LogOutButton]
                            </MudText>
                        </MudMenuItem>
                    </button>
                </form>
            </ChildContent>
        </MudMenu>
    </Authorized>
</PermissionView>
```

8.3 Integrace REST API

Pro volání koncových bodů jsou ve webové aplikaci klienta a v jejím aplikačním rozhraní přidány služby jako HTTP klient nebo reverzní proxy. Prototyp využívá další služby pro zapouzdření volání HTTP klienta, které zahrnují zpracování výjimek, politiky pro opakované dotazy a aplikací služby pro abstrakci manipulace i přístupu k datům. Tyto služby usnadňují efektivní správu a zabezpečení datových toků mezi klientem a serverem, zvyšují robustnost aplikace a optimalizují interakci s externími zdroji.

8.3.1 Mapování koncových bodů

Pro zjednodušení řešení volání koncových bodů REST API, které se běžně realizuje přidáním koncových bodů ve webovém aplikačním rozhraní klienta, je využita reverzní proxy. Tato proxy zmapuje koncové body REST API a během obsluhy směrování požadavků nejprve zkontroluje shody s koncovými body API klienta a poté s REST API. Pokud shoda není nalezena nebo pokud dojde k opomenutí přidání adresy koncového bodu REST API, uživatel bude přesměrován zpět na původní stránku, odkud požadavek vznikl. [43]

```
services.AddReverseProxy()
    .LoadFromMemory(new[]
    {
        new RouteConfig
        {
            RouteId = "route1",
            ClusterId = "cluster1",
            Match = new()
            {
                Path = "api/{**catch-all}"
            }
        }
    }, new[]
    {
        new ClusterConfig
        {
            ClusterId = "cluster1",
            Destinations = new Dictionary<string, DestinationConfig>
            {
                {
                    "destination1",
                    new DestinationConfig
                    {
                        Address = settings.ReverseProxy.Uri
                    }
                }
            },
            HttpClient = new()
            {
                MaxConnectionsPerServer = 10,
                SslProtocols = SslProtocols.Tls12 | SslProtocols.Tls13
            }
        }
    })
    .AddTransforms(options => options.AddRequestTransform(async context =>
    {
        var token = await context.HttpContext.GetTokenAsync(
            scheme: CookieAuthenticationDefaults.AuthenticationScheme,
            tokenName:….Tokens.BackchannelAccessToken);

        context.ProxyRequest.Headers.Authorization =
            new(OpenIddictConstants.Schemes.Bearer, token);
    }));
```

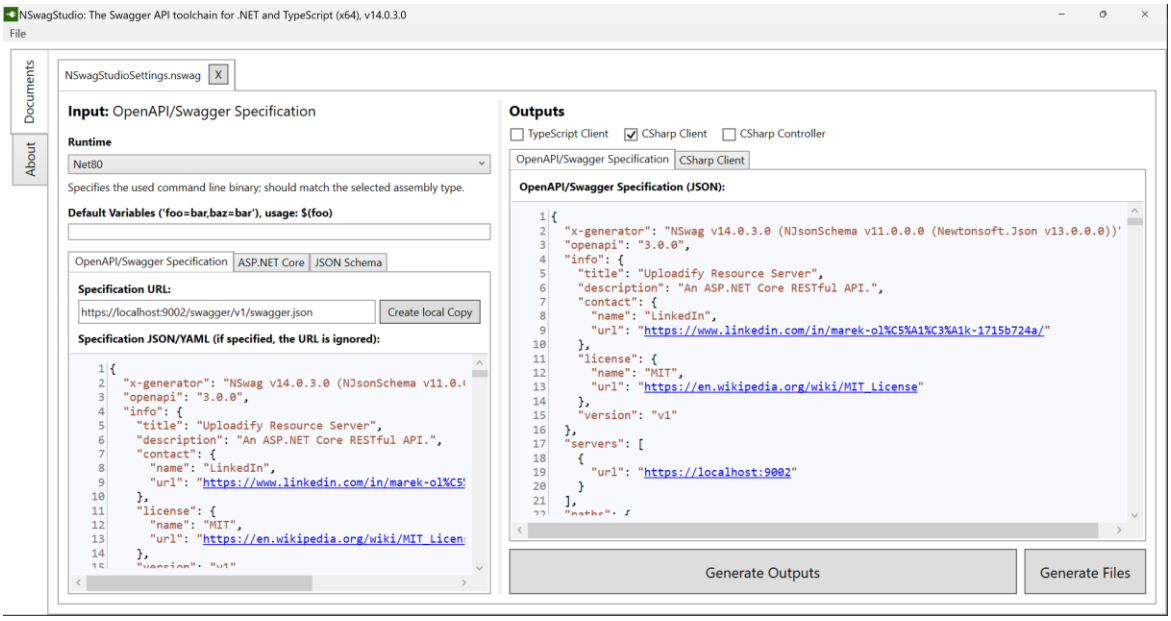
8.3.2 Integrační vrstva a generování HTTP klienta

Vrstva obsahuje konfigurační soubory pro generování HTTP klienta z dokumentace REST API. Výsledkem generování klienta je služba „UploadifyClient“, která zahrnuje jeho metody pro provolání koncových bodů a jejich modely dotazů i odpovědí. Samostatně služba neposkytuje infrastrukturu pro vytváření a obsluhu dotazů REST API, proto při její registraci vkládáme závislost na přednastaveného HTTP klienta pro provolání BFF. HTTP klient pro provolání BFF používá politiku provolání, která umožní opakovat neúspěšné dotazy s

tolerancí a časovou penalizací mezi jednotlivými dotazy. Registraci HTTP klienta a jeho politiky zpracování neúspěšných dotazů můžeme vidět níže.

```
services.AddHttpClient("authorizedClient", client =>
{
    client.BaseAddress = new(builder.HostEnvironment.BaseAddress);
    client.DefaultRequestHeaders.Accept.Add(
        new(MediaTypeNames.Application.Json));
}).AddHttpMessageHandler<AuthorizedHandler>()
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy())
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));

static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions.HandleTransientHttpError()
        .OrResult(response =>
            response.StatusCode == HttpStatusCode.ServiceUnavailable)
        .OrResult(response =>
            response.StatusCode == HttpStatusCode.RequestTimeout)
        .WaitAndRetryAsync(6, retry =>
            TimeSpan.FromSeconds(Math.Pow(2, retry)));
}
```

Klient pro provolání REST API generuje nástroj NSwagStudio, který z dokumentace vytvoří modely dotazů a odpovědí nebo metody provolání jeho koncových bodů. Metody pro provolání REST API nastaví hlavičky, směrovací adresu a obsah zprávy, která se odešle prostřednictvím HTTP klienta na server. Vygenerovaného klienta nástrojem NSwagStudio a dokumentaci jako úryvek souboru ve formátu JSON můžeme vidět na obrázcích níže.



Obrázek 7: Generování HTTP klienta pro provolání REST API nástrojem NSwagStudio.

```
{
  "paths": {
    "/api/file": {
      "post": {
        "summary": "Uploads a file to the server.",
        "description": "This endpoint is responsible for …",
        "requestBody": {
          "content": {
            "multipart/form-data": {
              "schema": {
                "type": "object",
                "properties": {
                  "UserName": {
                    "type": "string",
                    "nullable": true
                  },
                  "FolderId": {
                    "type": "integer",
                    "format": "int32",
                    "nullable": true
                  },
                  "CategoryId": {
                    "type": "integer",
                    "format": "int32",
                    "nullable": true
                  },
                  "File": {
                    "type": "string",
                    "format": "binary",
                    "nullable": true
                  }
                }
              }
            }
          }
        },
        "responses": {
          "201": {
            "description": "If the file is successfully uploaded and persisted.",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/CreateFileCommandResponse"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

8.3.3 Služby pro komunikaci s REST API

Služba „ApiCallWrapper“ zapouzdřuje volání metod vygenerovaného klienta použitím lambda výrazů, které slouží k provolání koncových bodů REST API a využívají vygenerovaného HTTP klienta nástrojem NSwagStudio. Služba obsahuje závislosti na službách pro logování a nastavení hlavičky k ověření XSRF tokenu, který vzniká po zavedení aplikace do prohlížeče uživatele. Registraci služby a vkládání závislostí prostřednictvím konstruktoru můžeme vidět na obrázku níže.

```
services.AddOptions<ApiCallWrapperOptions>().Configure(options =>
    options.IsDevelopment = builder.HostEnvironment.IsDevelopment());

services.AddSingleton(serviceProvider =>
{
    var javascriptRuntime = serviceProvider.GetRequiredService<IJSRuntime>();
    var token = ((IJSInProcessRuntime)javascriptRuntime)
        .Invoke<string>("getAntiForgeryToken");

    var httpClientFactory = serviceProvider.GetRequiredService<IHttpClientFactory>();
    var httpClient = httpClientFactory.CreateClient("authorizedClient");

    httpClient.DefaultRequestHeaders.Add("X-XSRF-TOKEN", token);

    return new ApiCallWrapper(
        httpClient,
        serviceProvider.GetRequiredService<IOptions<ApiCallWrapperOptions>>(),
        serviceProvider.GetRequiredService<ILogger<ApiCallWrapper>>());
});
```

Zapouzdření volání koncových bodů REST API umožňuje zpracovat výjimky a neúspěšné dotazy na aplikační úrovni. V případě úspěšného provolání koncového bodu vrací služba odpověď, která obsahuje příznak jako status dotazu, data a chybové hlášení. Chybové hlášení pro úspěšné dokončení dotazu nemá nastavenou hodnotu a v opačném případě data nejsou vráceny ze serveru. Pro zapouzdření dotazování se serveru vkládají služby výrazy jako parametr metody, které slouží jako selector metody generovaného klienta nástrojem NSwagStudio. Metoda zpracovává typy výjimek, které jsou součástí nastavení generátoru.

```
public async Task<TResponse?> Call<TResponse>(
    Func<UploadifyClient, Task<ApiCallResponse<TResponse>>> client)
    where TResponse : BaseResponse
{
    try
    {
        return (await client.Invoke(new(HttpClient))).Result;
    }
    catch (ApiCallException<TResponse> exception)
    {
        LogInformation(nameof(ApiCallWrapper), exception);
        return exception.Result;
    }
    catch (ApiCallException exception)
    {
        LogInformation(nameof(ApiCallWrapper), exception);

        var response = Activator.CreateInstance<TResponse>();

        response.Status = (Status)exception.StatusCode;
        return response;
    }
    catch (Exception exception)
    {
        LogError(nameof(ApiCallWrapper), exception);

        var response = Activator.CreateInstance<TResponse>();

        response.Status = Status.InternalServerError;
        return response;
    }
}
```

8.4 Lokalizace

Pro nastavení lokalizace v aplikaci se přepisuje výchozí hodnota kultury před spuštěním klienta, kterou ukládáme a aktualizujeme v lokálním úložišti pro její persistenci po ukončení klienta. Abychom uložili kulturu v lokálním úložišti, používá se interpolace mezi jazyky JavaScript a C#. JavaScript funkce pro nastavení kultury je součástí staticky souborů jako ikony a styly v kořenovém adresáři.

```
window.culture = {
    get: () => window.localStorage['culture'],
    set: (value) => window.localStorage['culture'] = value
};
```

Po sestavení aplikace a při jejím zavádění získáme hodnotu z lokálního úložiště zavoláním JavaScript funkce. Pokud hodnota nebyla nastavena, použijeme výchozí hodnotu. Výchozí lokalizace prototypu aplikace je angličtina, dalším podporovaným jazykem je čeština. Nastavení výchozí kultury pro aplikaci umožní lokalizaci textací a globalizaci, která se používá při formátování datumů a časů nebo měny.

```
CultureInfo culture;
var javascriptRuntime = host.Services.GetRequiredService<IJSRuntime>();
var cultureString = await javascriptRuntime.InvokeAsync<string>("culture.get");

if (IsNullOrWhiteSpace(cultureString))
{
    culture = new(Locales.Default);
    await javascriptRuntime.InvokeVoidAsync("culture.set", Locales.Default);
}
else
{
    culture = new(cultureString);
}

CultureInfo.DefaultThreadCurrentCulture = culture;
CultureInfo.DefaultThreadCurrentUICulture = culture;
```

8.4.1 Výběrovník preferovaného jazyka

Komponenta pro nastavení lokalizace textace v aplikaci po jejím výběru přepisuje v lokálním úložišti hodnotu, která se použije při registraci a nastavení pipeline během zavedení aplikace. Po změně této hodnoty v lokálním úložišti se znovu načte aplikace použitím služby pro směrování „NavigationManager“ pro natažení překladů textací, které distribuuje BFF.

```
<MudMenu …>
    <ActivatorContent …>…</ActivatorContent>
    <ChildContent>
        <MudMenuItem
            Disabled="@CultureInfo.CurrentCulture.Name.Equals(Locales.Czech)"
            OnClick="@(() => OnCultureChange(Locales.Czech))">
            <MudText>…</MudText>
        </MudMenuItem>
        <MudMenuItem
            Disabled="@CultureInfo.CurrentCulture.Name.Equals(Locales.English)"
            OnClick="@(() => OnCultureChange(Locales.English))">
            <MudText>…</MudText>
        </MudMenuItem>
    </ChildContent>
</MudMenu>

@code {
    private void OnCultureChange(string culture)
    {
        var cultureInfo = new CultureInfo(culture);
        if (CultureInfo.CurrentCulture.Equals(cultureInfo) ||
            JavaScriptRuntime is not IJSInProcessRuntime javascriptProcessRuntime)
        {
            return;
        }

        javascriptProcessRuntime.InvokeVoid("culture.set", culture);
        Navigation.NavigateTo(Navigation.Uri, true);
    }
}
```

8.4.2 Lokalizace textace

Pro překlad textací klienta se požívají klíče k překladu, které lokalizační služba používá pro získání lokalizované textace ze slovníku. MVC aplikace umožňují přidání slovníku pro každou stránku, řadič nebo celou aplikaci ve srovnání s Blazor WebAssembly, který používá architekturu založenou na komponentách, kde slovník přidáváme oproti nejvýše umístěné rodičovské komponentě v tomto uspořádání. Nejvýše umístěnou komponentou je komponenta pro obsluhu směrování v aplikaci, která je součástí souboru „App.razor“. Abychom usnadnili vkládání závislosti lokalizační služby, registrujeme závislost včetně i bez upřesnění jejího typu.

```
services.AddLocalization();
services.AddTransient<IStringLocalizer>(sp =>
    sp.GetRequiredService<IStringLocalizer<TranslationDictionary>>());
```

8.5 Uživatelské rozhraní

Přístup vývoje uživatelského rozhraní mobile-first klade důraz na přehlednost rozložení prvků rozhraní na mobilních zařízeních. Komponenta pro rozložení těchto prvků reaguje na změny velikosti plochy zobrazení zařízení pomocí breakpoints a současně informuje o její změně dceřiné komponenty prostřednictvím služby.

Prototyp aplikace používá odlišné rozložení komponent podle stavu přihlášení uživatele. Nepřihlášený uživatel je aplikací přesměrován na poskytovatele identity a po jejím ověření přesměrován nazpět. Během autentizace uživatele se zobrazuje ukazatel průběhu kvůli předání informace uživateli o tomto procesu.

```
<PermissionView>
    <Authorized>
        <AuthenticatedLayout>
            <Children>
                @Body
            </Children>
        </AuthenticatedLayout>
    </Authorized>
    <NotAuthorized>
        <UnathenticatedLayout>
            <Children>
                @Body
            </Children>
        </UnathenticatedLayout>
    </NotAuthorized>
</PermissionView>
```

8.5.1 Téma aplikace, typografie a piktogramy

Přepis velikosti fontů textací, jejich barev a dále se provádí nastavením parametru pro téma aplikace v komponentě „MudThemeProvider“, která je součástí balíčku komponent MudBlazor. Při registraci služeb přidáme závislosti pro komponenty knihovny, notifikace, dialogy a styly.

```
services.AddMudServices()
    .AddMudBlazorDialog()
    .AddMudBlazorSnackbar(options =>
    {
        options.MaxDisplayedSnackbars = 3;
        options.NewestOnTop = true;
        options.ShowCloseIcon = false;
        options.SnackbarVariant = Variant.Text;
        options.PositionClass = Defaults.Classes.Position.TopCenter;
    });
```

Nastavení stylů se provádí prostřednictvím komponenty na aplikační úrovni v komponentě pro hlavní rozložení uživatelského rozhraní aplikace.

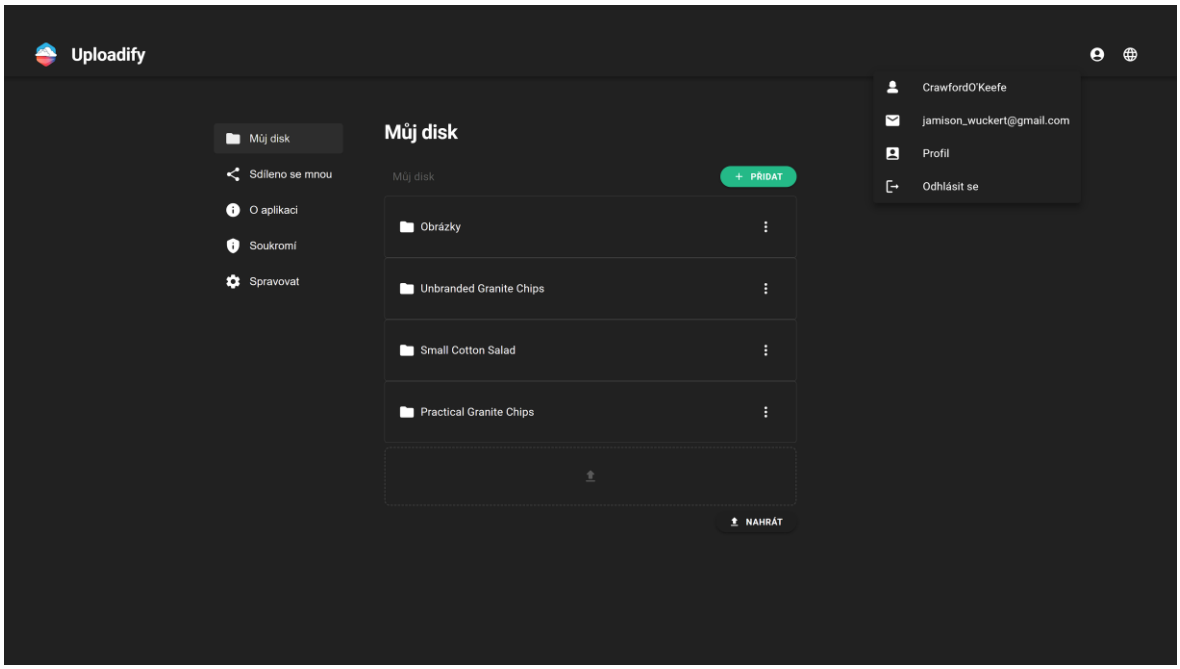
```
<MudThemeProvider Theme="@Themes.DefaultTheme" IsDarkMode/>
<MudDialogProvider FullWidth="true"
                   Position="@DialogPosition.Center"/>

<MudSnackbarProvider/>
```

8.5.2 Rozložení pro mobilní a desktopové aplikace

Uživatelské rozhraní pro mobilní zařízení používá vrchní i spodní navigační lištu, zatímco desktopové zařízení používají namísto spodní lišty postranní panel s centrovaným obsahem. Maximální šířka obsahu pro obě zařízení odpovídá šířky zobrazovací plochy středně velkých zařízení. Při změně velikosti zobrazovací plochy otočením zařízení nebo zmenší okna prohlížeče se vyvolá komponenta „MudBreakpointProvider“ událost, na kterou reaguje rozhraní přizpůsobením obsahu pro desktopové nebo mobilní aplikace. Příznak, který určuje, zdali se jedná o desktopové nebo mobilní rozložením, ukládáme ve službě „MobileViewManager“, kterou registrujeme jako singleton a vkládáme do dceřiných komponent. Alternativní přístup je použití kaskádových parametrů namísto služby.

Pro rozložení komponent požíváme šablonové komponenty, které usnadňují umístění obsahu do hlavičky, těla a zápatí stránky. Pro umístění komponenty do jeho částí použijeme fragmenty vykreslení „RenderFragment“, které můžeme nastavit pomocí vnořených značek jako „MudMainContent“ v komponentě pro rozložení stránky „MudLayout“.

```
<MudBreakpointProvider OnBreakpointChanged="OnBreakpointChanged"/>

<MudLayout>
    <MudAppBar Class="@(MobileViewManager.IsDesktop ? "pt-10 px-5" : "pt-10")"
               Fixed="false">
        <AppLogo/>
        <MudSpacer/>
        <Avatar/>
        <LocaleSelector/>
    </MudAppBar>
    <MudMainContent>
        @if (MobileViewManager.IsDesktop)
        {
            <div class="d-flex flex-grow-1 gap-15" style="padding-bottom:96px;">
                <div class="flex-1">
                    <div class="d-flex justify-end mud-width-full">
                        <DesktopNavigation/>
                    </div>
                </div>
                <div style="min-width:600px;max-width:600px;">
                    @Children
                </div>
                <div class="flex-1"></div>
            </div>
        }
        else
        {
            <MobileNavigation/>
            <MudContainer MaxWidth="@MaxWidth.Small" Style="padding-bottom:96px;">
                @Children
            </MudContainer>
        }
    </MudMainContent>
</MudLayout>
```



Obrázek 8: Snímek webové stránky klienta pro spravování souborů.

8.5.3 Notifikace

Webová aplikace zobrazuje upozornění prostřednictvím toastů. Toast byly nastaveny globálně v přechozí kapitole. Pro zobrazení toastu používá klient službu „ISnackbar“, která umožňuje zvolit typ závažnosti nebo změnit nastavení jeho zobrazení. Ukázka níže obsluhuje událost, která se volá po kliknutí na tlačítko stažení souboru. Tato událost provolá koncový bod pro stažení souboru a podle jeho status zobrazí toast, který informuje uživatele o úspěšném stažení souboru nebo jeho selhání.

8.5.4 Dialogy

Klient pro zobrazení formulářů, vytváření složek, přesunutí souborů do složek a dále používá dialogy namísto stránek. Dialogy ve srovnání se stránkami zachovají stav stránky, ve kterém dialog zobrazujeme. Pro zjednodušení a sjednocení rozložení obsahu dialogu přidáme základní komponenty, která přijímá jako parametry titulek a callback funkci, která se spouští při změně viditelnosti dialogu.

```
<MudDialog ClassContent="px-10 pb-10"
           IsVisible="@IsVisible"
           IsVisibleChanged="@HandleIsVisibleChanged"
           OnBackdropClick="@HandleBackdropClick">
    <TitleContent>
        <MudStack AlignItems="@AlignItems.Center" Row Spacing="2">
            <MudIconButton OnClick="@(() => HandleIsVisibleChanged(false))" … />
            <MudText Color="@Color.Primary" Typo="@Typo.h3">@Title</MudText>
        </MudStack>
    </TitleContent>
    <DialogContent>
        <div>
            @Content
        </div>
    </DialogContent>
</MudDialog>
```

Abychom umožnili nastavení obsahu, komponenta používá vykreslovací fragmenty, které předáváme komponentě jako parametry. Pro přidání a vynucení předání hodnoty parametru komponenty dekorujeme vlastnost atributy „Parameter“, který umožní nastavit a předat hodnotu dceřiné komponentě, a atribut „EditorRequired“, který vyžaduje nastavení hodnoty a podléhá statické kontrole.

Pro informování rodičovské komponentě o změně stavu dceřiné komponenty se používají callback funkce, které se volají při obsluze události jako změně viditelnosti dialogu. Callback funkce jako hodnotu přijímá referenci na metodu rodičovské komponenty nebo lambda výraz, kde v tomto případě se jedná o akci, která přijímá parametry, ale nevrací hodnotu.

```
@code {
    [Parameter] [EditorRequired] public required string Title { get; set; }
    [Parameter] [EditorRequired] public required RenderFragment Content { get; set; }
    [Parameter] [EditorRequired] public required bool IsVisible { get; set; }
    [Parameter] [EditorRequired] public required EventCallback OnClose { get; set; }

    private Task HandleBackdropClick(MouseEventArgs args)
    {
        return OnClose.InvokeAsync();
    }

    private Task HandleIsVisibleChanged(bool isVisible)
    {
        IsVisible = isVisible;

        if (isVisible)
        {
            return Task.CompletedTask;
        }

        return OnClose.InvokeAsync();
    }
}
```

8.6 Přehled složek a souborů

Stránka „DashboardPage“ vyžaduje přihlášeného uživatele, a proto při směrování používá atribut oprávnění „PermissionAttribute“ bez nastavení jeho hodnoty. Současně prostřednictvím atributu Route se nastaví směrovací adresa stránky, na které bude aplikace poslouchat. Pro předčasné ukončení dotazů na REST API implementuje stránka rozhrání „IDisposable“, které přidává metodu, která se volá před uvolněním alokované paměti GC, a inicializuje token, který se při uvolnění použije pro vyvolání výjimky v REST API klientovi. Tento token předáme komponentě pro zobrazení přehledu souborů a složek uživatele prostřednictvím jeho parametru.

```
@attribute [Permission]
@attribute [Route(PageRoutes.Homepage)]

@implements IDisposable

<Dashboard CancellationToken="@CancellationTokenSource.Token"/>

@code {
    public readonly CancellationTokenSource CancellationTokenSource = new();

    void IDisposable.Dispose()
    {
        CancellationTokenSource.Cancel();
        CancellationTokenSource.Dispose();
    }
}
```

8.6.1 Zobrazení složek a souborů

Komponenta pro zobrazení přehledu souborů a složek se skládá z dialogů a listu karet, který prezentuje jednotlivé položky souborového systému. Při inicializaci komponenty se volá koncový bod pro natažení přehledu kořenového adresáře uživatele, který obsahuje seznam složek, souborů a aktuální cestu pro navigaci mezi složkami. Obsah stránky je zapouzdřen v komponentě pro načítání dat, která zobrazí skeleton a po natažení dat obsah stránky.

```
<LoadingWrapper Title="@Localizer[Translations.Pages.Dashboard.Title]"
                ShowTitle
                IsLoading="@IsLoading">
    <Skeleton>
        <PrimarySkeleton FullWidth Height="500px"/>
    </Skeleton>
    <Children>
        …
    </Children>
</LoadingWrapper>
```

Soubory a složky se zobrazují jako karty, které jako parametry přijímají callback funkce pro výsledek jejich operací jako přesunutí nebo nahrání nového souboru. Karta se skládá z ikony a názvu souboru i složky nebo možností pro jejich manipulaci jako menu.

```
<MudCard Elevation="1" Outlined Class="pa-5">
    <div class="d-flex flex-grow-1 align-center justify-space-between"
         @ondblclick="@HandleDoubleClick">
        <div class="d-flex align-center gap-2">
            <MudIcon Title="@(…)" Icon="@(…)" Color="@Color.Primary"/>
            <MudText>@Item.Name</MudText>
        </div>
        <DashboardCardOptions Item="@Item"
                              OnOpen="@OnOpenOptions"
                              OnOpenDetailDialog="@OnOpenDetailDialog"
                              OnOpenFolder="@OnOpenFolder" …/>
    </div>
</MudCard>
```

8.6.2 Dialog pro vytváření složek a nahrávání souborů

Dialog se skládá ze dvou záložek. První záložka obsahuje vstup pro zadání názvu složky a tlačítka pro odeslání požadavku na server. Tlačítko používá časovou prodlevu mezi jednotlivými dotazy, která od doby jejich odeslání do obdržení odpovědi blokuje interakce uživatele s tlačítkem. Po dokončení požadavku se volá callback funkce rodičovské komponenty, která se dialogu nastaví jedním z jejích parametrů vstupu. Callback funkce aktualizuje přehled souborů a složek současně otevřeného adresáře.

```
<Dialog IsVisible="@IsVisible"
        Title="@Localizer[Translations….Create.Title]"
        OnClose="@(() => IsVisible = false)">
    <Content>
        <MudTabs>
            <MudTabPanel Icon="@Icons.Material.Rounded.Folder"
                         Text="@Localizer[Translations….Create.FoldersTab]">
                <ChildContent>…</ChildContent>
            </MudTabPanel>
            <MudTabPanel Icon="@Icons.Material.Rounded.Image"
                         Text="@Localizer[Translations….Create.FilesTab]">
                <ChildContent>…</ChildContent>
            </MudTabPanel>
        </MudTabs>
    </Content>
</Dialog>
```

Komponenta formuláře „MudForm“ nefunguje jako element „<form>“ značkovacího jazyka HTML. Formulář umožňuje integraci frameworků pro validaci vstupů, usnadňuje zobrazení validačních chyb nebo vstupů. Záložky dialogu používají formuláře pro zobrazení validačních chyb, které jsou obsaženy v odpovědi na požadavek, nebo rozložení komponent pro vstupy a tlačítka.

```
<MudForm Errors="@(ResourceResponse?.ErrorMessages ?? Array.Empty<string>())">
    <MudStack Spacing="5" Class="pt-5">
        <MudTextField @bind-Value="@Name"
                      DebounceInterval="100"
                      Error="@(ResourceResponse is { ErrorMessages.Length: > 0 })"
                      ErrorText="@ResourceResponse?.ErrorMessages.FirstOrDefault()"
                      Variant="@Variant.Outlined"
                      Label="@Localizer[Translations….Rename.NameLabel]"
                      OnDebounceIntervalElapsed="@HandleDebounceIntervalElapsed"/>

        <PrimaryButton IsDisabled="@(IsLoading || IsDisabled)"
                       IsLoading="@IsLoading"
                       FullWidth
                       Color="@Color.Success"
                       OnClick="@HandleSubmit"
                       Size="@Size.Large"
                       Text="@Localizer[Translations….Create.SubmitButton]"/>
    </MudStack>
</MudForm>
```

8.6.3 Drag and drop komponenta pro nahrání souborů

Druhá záložka obsahuje komponentu pro nahrání více souborů výběrem nebo jejich přetažením, která se pro desktopovou verzi webové aplikace nachází i na konci listu karet souborů a složek. Komponenta podporuje nahrání více souborů v jeden okamžik o velikosti nejvýše 100 MB. Limitaci lze odůvodnit přenosem obsahu souboru, který nejprve nahraje dokument do paměti webové aplikace a poté odešle společně s metadaty na server. Alternativní přístupy k řešení problému jako implementace JavaScript funkcí pro nahrání souborů po

částech (tzv. blobs) umožní tento limit překročit, avšak funkce nejsou součástí řešení v rámci první iterace.

```
<MudForm Errors="@(ResourceResponse?.ErrorMessages ?? Array.Empty<string>())">
    <div class="pt-5">
        <FileDropzone OnSuccess="@OnSuccess"
                      Destination="@ParentFolder"
                      Color="@Color.Success"/>
    </div>
</MudForm>
```

Po nahrání souborů na server a jejich zpracování se zobrazí upozornění, které informuje uživatele o úspěšnosti operace. Na konci obsluhy se volá callback funkce pro aktualizaci přehledu souborů a složek.

```
<MudFileUpload AppendMultipleFiles
               Disabled="@(IsDisabled || IsLoading)"
               MaximumFileCount="@MaximumFileCount"
               @ondragend="@ClearDropzoneClass"
               @ondragenter="@SetDropzoneClass"
               @ondragleave="@ClearDropzoneClass"
               OnFilesChanged="@(args => HandleFilesChanged(args))"
               T="IReadOnlyList<IBrowserFile>" …>
    <ButtonTemplate>
        <MudPaper …>
            <MudStack …>
                <MudIconButton Disabled Icon="@Icons.Material.Rounded.FileUpload"/>
            </MudStack>
        </MudPaper>
        <div class="relative d-flex justify-end z-30 mt-2">
            <MudButton for="@context.Id"
                       HtmlTag="label"
                       Disabled="@(IsDisabled || IsLoading)" …>
                @Localizer[Translations….FileDropZone.UploadButton]"
            </MudButton>
        </div>
    </ButtonTemplate>
</MudFileUpload>
```

9 TESTOVACÍ SADY A PŘÍPADY

Jednotkové testy společně s ostatními typy jako integrační nebo systémové vykazují jakost systému a jejich hlavním přínosem je předejití nežádoucího chování systému zanesením chyby. Prototyp aplikace se zaměřuje na vytvoření testovacích sad a případů pro dotazy a příkazy v MediatR, které představují aplikační logiku.

9.1 Stub kontextu databáze

Testovací případy ověřují jeden případ užití dané komponenty. Přidání stubs umožňuje nahrazení závislostí testované komponenty, kde stubs zapouzdřují existující komponenty systému. Zapouzdření komponenty umožňuje nastavit vstupní hodnoty jejího rozhrání a očekávané výstupy, které použijeme jako vstupy testované komponenty. Nahrazení komponent za stubs odebírá riziko nežádoucího chování dalších komponent z testovacího případu, a tak umožnit implementaci jednotkových testů pro jednotlivé testovací případy.

Klíčová služba aplikační logiky je kontext databáze, který používají dotazy a příkazy pro získání a manipulaci s daty uživatele. Pro usnadnění implementace jednotkových testů se používá pomocná třída pro tvorbu databázového kontextu a jeho zdrojů jako kolekce uživatelů nebo souborů. Stubs databázového kontextu a dalších služeb implementuje knihovna „Moq“, která umožňuje nastavení vstupních i výstupních parametrů rozhrání komponent.

```
public static class MockDataContextFactory
{
    public static Mock<DbSet<TEntity>> CreateMockDbSet<TEntity>(
        IEnumerable<TEntity> source) where TEntity : class {…}


    public static Mock<DataContext> SetupDataContext<TEntity>(
        Expression<Func<DataContext, DbSet<TEntity>>> selector,
        Mock<DbSet<TEntity>> entities) where TEntity : class
    {
        var context = new Mock<DataContext>(new DbContextOptions<DataContext>());
        context.Setup(selector).Returns(entities.Object);
        return context;
    }

    public static Mock<DataContext> SetupDbSet<TEntity>(
        this Mock<DataContext> context,
        Expression<Func<DataContext, DbSet<TEntity>>> selector,
        Mock<DbSet<TEntity>> entities) where TEntity : class
    {
        context.Setup(selector).Returns(entities.Object);
        return context;
    }
}
```

Pro ověření správnosti dotazů nebo příkazů testovací případy emulují chování objektově relačního mapovače a připojení k databázi. Jelikož příkazy mění stavy záznamů v databázi, prototyp vytváří novou instanci databáze pro každý testovací případ v paměti zařízení, na kterém spouštíme testovací sady. Protože součástí knihovny Moq není podpora mapování LINQ výrazů na dotazy databáze, začleníme rozšiřující funkce do prototypu řešení.

```
public static Mock<DbSet<TEntity>> CreateMockDbSet<TEntity>(
    IEnumerable<TEntity> source) where TEntity : class
{
    var queryable = source.AsQueryable();
    var entities = new Mock<DbSet<TEntity>>();

    entities.As<IAsyncEnumerable<TEntity>>()
        .Setup(m => m.GetAsyncEnumerator(new CancellationToken()))
        .Returns(new AsyncEnumerator<TEntity>(queryable.GetEnumerator()));

    entities.As<IQueryable<TEntity>>()
        .Setup(m => m.Provider)
        .Returns(new AsyncQueryProvider<TEntity>(queryable.Provider));

    entities.As<IQueryable<TEntity>>()
        .Setup(m => m.Expression)
        .Returns(queryable.Expression);

    entities.As<IQueryable<TEntity>>()
        .Setup(m => m.ElementType)
        .Returns(queryable.ElementType);

    entities.As<IQueryable<TEntity>>()
        .Setup(m => m.GetEnumerator())
        .Returns(queryable.GetEnumerator());

    return entities;
}
```

9.2 Stubs pro dotazy a příkazy v MediatR a další služby

Stubs dalších komponent vyjma ORM můžeme implementovat nastavením vstupních a výstupních parametrů rozhrání pomocí knihovny „Moq“. Před jejich implementací se vždy přidávají testovací data do zdrojů kontextu databáze. Ukázkový kód níže požívá nejnovější syntaxi pro inicializaci kolekcí hranatými závorkami, která je součástí verze 12 jazyka C#.

```
var mockDataContext = MockDataContextFactory.SetupDataContext(
    context => context.Folders,
    MockDataContextFactory.CreateMockDbSet([_parentFolder]));

var mockSender = new Mock<ISender>();

mockSender.Setup(sender => sender.Send(
        It.IsAny<GetUserQuery>(),
        It.IsAny<CancellationToken>()))
    .ReturnsAsync(new GetUserQueryResponse(_user));
```

9.3 Testovací případ vytvoření složky

Testovací sada pro příkaz vytváření složek v MediatR obsahuje jednotkové testy, které pokrývají případy užití níže:

- Pro platné vstupy komponenta vrací status pro úspěšné vytvoření složky a v odpovědi její model entity.
- Pro neplatný vstup komponenta vrací status pro neplatný model a v odpovědi jeho validační chyby, uživatelsky přívětivou hlášku a výjimku.
- Pro neautorizovaný přístup k datům uživatele vrací status neoprávněného přístupu a v odpovědi uživatelsky přívětivou hlášku i výjimku.
- Pro neexistujícího uživatele komponenta vrací status nenalezené entity a v odpovědi uživatelsky přívětivou hlášku i výjimku.
- Pro neexistující nadřazenou složkou komponenta vrací status nenalezené entity a v odpovědi uživatelsky přívětivou hlášku i výjimku.

Ukázkový příklad níže pokrývá testovací případ úspěšného vytvoření složky. Jednotkový test před spuštěním testu vytváří stubs služeb jako databázový kontext a dotaz pro natažení účtu přihlášeného uživatele nebo nadřazené složky. Pro kontext databáze se inicializují zdroje složek a uživatelů, který obsahuje záznam jednoho uživatele.

Před spuštěním testované komponenty vytvoříme příkaz a službu pro jeho obsluhu, ve které předáme závislosti nahrazené za příslušné stubs. Po spuštění komponenty uložíme výsledek a ověříme jeho správnost. V případě vyvolání výjimky testovací případ selže, z tohoto důvodu se neošetřují výjimky během obsluhy příkazu. Výsledek musí obsahovat odpovídající stavový kód, vytvořenou složku a chybové hlášení, které není inicializováno. Pokud nebude splněna jedna z vyjmenovaných podmínek pro testování, testovací případ selže.

```
[Fact]
public async Task Handle_WhenValidCommand_ThenReturnCreatedResponse()
{
    // Arrange
    var mockDataContext = MockDataContextFactory.SetupDataContext(
        context => context.Folders,
        MockDataContextFactory.CreateMockDbSet([_parentFolder]));

    mockDataContext.Setup(context =>
        context.SaveChangesAsync(It.IsAny<CancellationToken>())).ReturnsAsync(1);

    var mockSender = new Mock<ISender>();
    mockSender.Setup(sender => sender.Send(
            It.IsAny<GetUserQuery>(),
            It.IsAny<CancellationToken>()))
        .ReturnsAsync(new GetUserQueryResponse(_user));

    mockSender.Setup(sender => sender.Send(
            It.IsAny<GetFolderQuery>(),
            It.IsAny<CancellationToken>()))
        .ReturnsAsync(new GetFolderQueryResponse(_parentFolder));

    var command = new CreateFolderCommand
    {
        UserName = _user.UserName,
        Name = "NewFolderName",
        ParentId = _parentFolder.Id
    };

    var handler =
        new CreateFolderCommandHandler(mockDataContext.Object, mockSender.Object);

    // Act
    var response = await handler.Handle(command, CancellationToken.None);

    // Assert
    response.Should().NotBeNull();
    response.Status.Should().Be(Status.Created);
    response.Folder.Should().NotBeNull();
    response.Folder.Name.Should().Be("NewFolderName");
    response.Failure.Should().BeNull();
}
```

ZÁVĚR

Bakalářská práce se zaměřila na dokumentaci návrhu a vývoje prototypu webové aplikace pro sdílení dat. Průzkum stávajících řešení umožnil definovat funkční a nefunkční požadavky na systém, které byly klíčové pro jeho návrh a výběr technologií. Prototyp byl vytvořen na základě architektury mikro-služeb, která představovala odchylku od tradiční monolitické architektury a usnadnila horizontální škálování systému v závislosti na jeho vytížení. Použití vícevrstvého modelu pro realizaci struktury řešení přispělo k lepší udržitelnosti systému tím, že oddělilo aplikační logiku od prezentace dat, jejich čtení a zápis. Pro vývoj databázového systému byl zvolen přístup code-first, který umožnil reprezentaci struktury databáze pomocí modelů doménové vrstvy a její verzování prostřednictvím generování migrací.

Klíčovým požadavkem na systém bylo zabezpečení ukládání a přenosu informací. Pro zabezpečení ukládání hesel v databázi byl využit algoritmus Argon2id, který odolává nárůstu paralelního výpočetního výkonu, kterým disponují příkladem grafické karty. Pro zabezpečení přenosu citlivých informací, autorizaci požadavků a sdílení identity uživatele byla implementována služba poskytovatele identity s podporou protokolů OAuth 2.0 pro autorizaci a OpenID Connect pro sdílení identity. Tato implementace umožnila bezpečnou komunikaci s mobilními a nedůvěryhodnými webovými klienty, která vyžaduje vyšší úroveň zabezpečení přenosu informací, než je běžné u tradičních přístupů s rotací tokenů.

Další službou bylo REST API, které umožnilo klientům manipulovat a přistupovat k datům uživatele a generovalo dokumentaci pomocí Swagger pro snadnou integraci klientů. Využití vertikální a horizontální autorizace během zpracování dotazů na REST API a introspekce přístupového tokenu po jeho obdržení zvýšilo bezpečnost dat uživatele. K udržitelnosti řešení přispěl návrhový vzor CQRS, který rozdělil aplikační logiku na dotazy (čtení dat) a příkazy (zápis dat).

Součástí prototypu aplikace bylo navržení a implementace klienta, který byl spuštěn ve webovém prohlížeči uživatele s využitím technologie WebAssembly. Tato technologie umožnila použití .NET při vývoji komponent pro front-end i back-end části systému. Přístup mobile-first, použitý během vývoje klienta, přispěl k přenositelnosti webové aplikace na mobilní a desktopové zařízení. K udržitelnosti a zabezpečení klienta dále přispěla implementace API, které využívalo návrhový vzor BFF. API zároveň sloužilo jako reverzní proxy pro přesměrování požadavků z klienta na REST API, které přispělo k udržitelnosti řešení.

SEZNAM POUŽITÉ LITERATURY

[1] MICROSOFT, c2024. .NET | Build. Test. Deploy. [online]. [cit. 2024-04-23]. Dostupné z: https://dotnet.microsoft.com/en-us/

[2] MICROSOFT, c2024. Jazyková nezávislost a komponenty nezávislé na jazyce – .NET | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/language-independence

[3] MICROSOFT, c2024. Proces spravovaného spouštění | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/managed-execution-process

[4] MICROSOFT, c2024. Implementace .NET | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/fundamentals/implementations

[5] MICROSOFT, c2024. .NET Standard | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/net-standard?tabs=net-standard-1-0

[6] MICROSOFT, c2024. Prohlídka jazyka C# | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/csharp/tour-of-csharp/

[7] MICROSOFT, c2024. Základy uvolňování paměti | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/garbage-collection/fundamentals

[8] ARCOVERDE, Roberta a Ryan DONOVAN, 2021. Best practices can slow your application down. STACK EXCHANGE INC. Blog [online]. [cit. 2024-04-23]. Dostupné z: https://stackoverflow.blog/2021/12/22/best-practices-can-slow-your-application-down/

[9] CRAVER, Nick, 2016. Stack Overflow: The Architecture - 2016 Edition. Nick Craver – Software Imagineering [online]. [cit. 2024-04-24]. Dostupné z: https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/

[10] ENGSTRÖM, Jimmy, 2021. Web Development with Blazor: A hands-on guide for .NET developers to build interactive UIs with C#. Birmingham (United Kingdom): Packt. ISBN 978-1800208728.

[11] THOMPSON, Seth et al, 2024. WebAssembly. WebAssembly [online]. [cit. 2024-05-05]. Dostupné z: https://webassembly.org/

[12] MICROSOFT, c2024. ASP.NET nástrojů pro sestavení Core Blazor WebAssembly a kompilace AOT (Head-of-Time) | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/webassembly-build-tools-and-aot?view=aspnetcore-8.0

[13] MICROSOFT, c2024. Modely hostování ASP.NET Core Blazor | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/hosting-models?view=aspnetcore-8.0

[14] MICROSOFT, c2024. Interoperabilita JavaScriptu s ASP.NET Core Blazor (zprostředkovatel komunikace JS) | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/javascript-interoperability/?view=aspnetcore-8.0

[15] MUDBLAZOR, c2020-2024. MudBlazor – Blazor Component Library [online]. [cit. 2024-05-05]. Dostupné z: https://mudblazor.com/

[16] HANGFIRE OÜ, c2013-2024. Hangfire – Background jobs and workers for .NET and .NET Core [online]. [cit. 2024-05-05]. Dostupné z: https://www.hangfire.io/

[17] SAKIMURA, Nat et al., 2023. Final: OpenID Connect Core 1.0 incorporating errata set 2 [online]. 3. dopl. OpenID Foundation [cit. 2024-05-05]. Dostupné z: https://openid.net/specs/openid-connect-core-1_0.html

[18] CHALET, Kévin et al, 2024. OpenIddict [online]. [cit. 2024-05-05]. Dostupné z: https://documentation.openiddict.com/index.html

[19] SMARTBEAR SOFTWARE, c2024. API Documentation & Design Tools for Teams | Swagger [online]. [cit. 2024-05-05]. Dostupné z: https://swagger.io/

[20] SUTER, Rico et al, c2024. GitHub – RicoSuter/NSwag: The Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript. GITHUB, INC. GitHub: Let's build

from here · GitHub [online]. [cit. 2024-05-05]. Dostupné z: https://github.com/RicoSuter/NSwag

[21] MICROSOFT, c2024. Přehled technologie Entity Framework Core (EF Core) | Microsoft Learn. MICROSOFT. Microsoft Learn. Spousta možností. [online]. [cit. 2024-04-23]. Dostupné z: https://learn.microsoft.com/cs-cz/ef/core/

[22] THE NPGSQL DEVELOPMENT TEAM, c2023. Npgsql – .NET Access to PostgreSQL | Npgsql Documentation [online]. [cit. 2024-05-05]. Dostupné z: https://www.npgsql.org/

[23] SPASOJEVIC, Marinko a Vladimir PECANAC, 2021. ULTIMATE ASP.NET CORE WEB API: From Zero to Six-Figure Backend Developer [online]. Second Edition. CodeMaze [cit. 2024-05-05]. Dostupné z: https://code-maze.com/ultimate-aspnetcore-webapi-second-edition/

[24] LOCK, Andrew, 2021. ASP.NET Core in Action. Annotated Edition. Shelter Island (New York): Manning Publications, 832 s. Second Edition. ISBN 978-1617298301.

[25] AMAZON WEB SERVICES, INC., c2024. CQRS pattern – AWS Prescriptive Guidance. Cloud Computing Services – Amazon Web Services (AWS) [online]. [cit. 2024-05-06]. Dostupné z: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/cqrs-pattern.html

[26] LAURENT, Arnaud, 2019. The design of everyday APIs. Shelter Island (New York): Manning Publications, 392 s. ISBN 978-1617295102.

[27] MARTIN, Robert C., 2017. Clean architecture: a craftsman's guide to software structure and design. London (England): Prentice Hall. ISBN 978-0134494166.

[28] BOWDEN, Damien, 2022. Secure a Blazor WASM ASP.NET Core hosted APP using BFF and OpenIddict | Software Engineering. Software Engineering | Web development [online]. [cit. 2024-05-06]. Dostupné z: https://damienbod.com/2022/01/03/secure-a-blazor-wasm-asp-net-core-hosted-app-using-bff-and-openiddict/

[29] BAIER, Dominick, 2022. Securing SPAs and Blazor Applications using the BFF (Backend for Frontend) Pattern [online]. [cit. 2024-05-06]. Dostupné z: https://www.youtube.com/watch?v=DdNssiaIY_Q

[30] OWNCLOUD, c2024. OwnCloud – share files and folders, easy and secure [online]. [cit. 2024-05-06]. Dostupné z: https://owncloud.com/

[31] THE OWNCLOUD DEVELOPERS, c2011-2024. Introduction to ownCloud Server :: Documentation for ownCloud (A Kiteworks Company) [online]. [cit. 2024-05-06]. Dostupné z: https://doc.owncloud.com/server/10.14/

[32] NEXTCLOUD, c2021 - 2024. Nextcloud – Open source content collaboration platform [online]. [cit. 2024-05-06]. Dostupné z: https://nextcloud.com/

[33] JOBKE, Morris et al, 2024. Nextcloud developer documentation — Nextcloud latest Developer Manual latest documentation [online]. [cit. 2024-05-06]. Dostupné z: https://docs.nextcloud.com/server/latest/developer_manual/index.html

[34] SAKELLARIOS, Christos, 2018. ASP.NET Core Identity Series – Getting Started. Chsakell's Blog – WEB APPLICATION DEVELOPMENT TUTORIALS WITH OPEN-SOURCE PROJECTS [online]. [cit. 2024-05-08]. Dostupné z: https://chsakell.com/2018/04/28/asp-net-core-identity-series-getting-started/

[35] LOCK, Andrew, 2017. Exploring the ASP.NET Core Identity PasswordHasher [online]. ANDREW LOCK | .NET ESCAPADES. [cit. 2024-05-08]. Dostupné z: https://andrewlock.net/exploring-the-asp-net-core-identity-passwordhasher/

[36] PREZIUSO, Michele, 2019. Password Hashing: Scrypt, Bcrypt and ARGON2 | by Michele Preziuso | Medium. Medium [online]. [cit. 2024-05-08]. Dostupné z: https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2-e25aaf41598e

[37] HONGJUN, Wu et al, 2023. Password hashing | libsodium. Introduction | libsodium [online]. [cit. 2024-05-08]. Dostupné z: https://doc.libsodium.org/password_hashing

[38] PERUZZI SERVICES LIMITED, 2023. Advanced features of the MediatR package - Pipeline Behaviors | LinkedIn. LINKEDIN CORPORATION. Informační kanál | LinkedIn [online]. [cit. 2024-05-08]. Dostupné z: https://www.linkedin.com/pulse/advanced-features-mediatr-package-pipeline-behaviors/

[39] BUI, Andrii, 2023. OAuth Authorization Code using OpenIddict and .NET. BUI, Andrii. Andreyka26 tech [online]. [cit. 2024-05-08]. Dostupné z: https://andreyka26.com/oauth-authorization-code-using-openiddict-and-dot-net

[40] BUI, Andrii, 2023. OpenId Connect Authorization Code using OpenIddict and .NET. BUI, Andrii. Andreyka26 tech [online]. [cit. 2024-05-08]. Dostupné z: https://andreyka26.com/openid-connect-authorization-code-using-openiddict-and-dot-net

[41] OKTA, INC., 2024. OpenID Connect Scopes. OKTA, INC. Auth0 Docs [online]. [cit. 2024-05-08]. Dostupné z: https://auth0.com/docs/get-started/apis/scopes/openid-connect-scopes

[42] Developing Flexible Authorisation Capabilities in ASP.NET Core - Jason Taylor - NDC London 2023 [online], 2023. [cit. 2024-05-08]. Dostupné z: https://www.youtube.com/watch?v=vkhtdgfHZYc

[43] MICROSOFT, 2024. Getting Started with YARP. YARP Documentation [online]. [cit. 2024-05-08]. Dostupné z: https://microsoft.github.io/reverse-proxy/articles/getting-started.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AoT	Ahead of Time – překladač prostředí běhu .NET, který převádí IL do strojově srozumitelného kódu před spuštěním aplikace.
API	Application Programming Interface – programovatelné aplikační rozhraní pro komunikaci s dalšími komponentami systému nebo službami.
BFF	Backend For Frontend – návrhový vzor pro vývoj webových a mobilních aplikací, ve kterém vystupuje klient (webové i službám aplikace) a webové API jako abstrakce vrstvy pro komunikaci s ostatními službami.
CQRS	Command Query Responsibility Segregation – návrhový vzor, který separuje logiku pro čtení a zápis dat v databázi.
CRUD	Akronym pro Create, Read, Update a Delete, tj. základní operace pro správu dat v databázi.
CSS	Cascading Style Sheets – jazyk pro popis vzhledu a formátování HTML dokumentů.
DOM	Document Object Model – programovací rozhrání pro webové dokumenty.
GC	Garbage Collection – mechanismus pro automatizovanou správu paměti alokované na haldě.
GraphQL	Jazyk pro dotazování webových API pro manipulaci se zdroji. Umožnuje klientovi dotazovat se na konkrétní informace jako emailovou adresu uživatele namísto detailu uživatele.
gRPC	Framework pro vzdálené volání procedur (RPC), který umožňuje efektivní komunikaci mezi serverem a klientem. Ve srovnání s REST například neprovádí serializaci a deserializaci dat, tudíž je výkonnější a efektivnější.
HATEOAS	Hypermedia As The Engine Of Application State – koncept REST architektury, který umožňuje klientům navigovat mezi různými částmi webové aplikace prostřednictvím hypermédií.
HTML	Hypertext Markup Language – standardizovaný značkovací jazyk pro strukturování obsahu webového dokumentu.

HTTP	Hypertext Transfer Protocol – protokol pro přenos hypertextových dokumentů jako webové stránky v síti.
IL	Intermediate Language – mezi úrovňový programovací jazyk, který překládá prostředí běhu .NET do strojově srozumitelného kódu.
IoC	Inversion of Control – paradigma, které přesouvá převrací tradiční řízení programu, kde jeho části jsou řízeny centralizovaným nebo externím mechanismem. Paradigma přispívá k modularitě a udržitelnosti systému.
IP	Internet Protocol – sada pravidel pro adresování a směrování dat mezi zařízeními v síti.
JIT	Just in Time – překladač prostředí běhu .NET, který převádí IL do strojově srozumitelného kódu na vyžádání za běhu aplikace.
JWT	JSON Web Token – popsán ve standardu RFC 7519. Metoda pro zabezpečení přenosu informací mezi systémy.
LINQ	Language Integrated Query – knihovna rozšíření .NET, která umožňuje manipulovat se zdroji dat.
MVC	Model View Controller – návrhový vzor pro vývoj softwarových aplikací, který rozděluje aplikaci do tří komponent: modelu, který reprezentuje data, pohledu, které prezentuje data uživateli, a řadiče, který obsluhuje dotazy uživatele.
ORM	Object Relational Mapping – abstraktní vrstva pro mapování modelů aplikace na databázové entity.
PKCE	Proff Key for Code Exchange – vyslovováno „pixy“. Doplněk autorizačního toku Authorization Code Flow protokolu OAuth 2.0, který přispívá k ochraně proti CSRF a Authorization code injection útoku.
REST	Representational State Transfer – architektura navržená pro distribuované systémy, zejména webové služby.
SOLID	Akronym pro pět zásad návrhu softwaru: Single Responsibility, Open/Closed principle, Lisko Substitution, Interface Segregation a Dependency Inversion. Zásady přispívají k vývoji modulárních systémů.
SPA	Jednostránková webová aplikace.

SQL	Structured Query Language – standardizovaný programovací jazyk pro manipulaci s relačními databázemi.
URL	Uniform Resource Locator – standardizovaná adresa pro lokalizaci zdrojů v síti.
XSRF	Cross Site Request Forgery – kybernetický útok, který zneužívá relaci přihlášeného uživatele k dotazování se serveru, které bez vědomí a souhlasu uživatele provádí změny v systému.
XSS	Cross Site Scripting – kybernetický útok, který vkládá škodlivý skript napsaný v jazyce JS do DOM webové aplikace, který po spuštění bez vědomí a souhlasu uživatele může vést k odcizení informací uložených například v Cookies. Nebezpečný skript může být vložen i jako událost kliknutí na tlačítko nebo umístění kurzoru nad paragraf.

SEZNAM OBRÁZKŮ

Obrázek 1: Architektura mikro-služeb použitá v prototypu aplikace. 34
Obrázek 2: Diagram databázového systému prototypu aplikace. 37
Obrázek 3: Dokumentace REST API generovaná frameworkem NSwag. 62
Obrázek 4: Plánovač a přehled úloh v Hangfire. 67
Obrázek 5: Dokumentace Swagger koncového bodu pro nahrání souboru. 73
Obrázek 6: Snímek záložky Síť v nástrojích pro vývojáře ve webovém prohlížeči po načtení statických souborů klienta a natažení přehledu souborů. 75
Obrázek 7: Generování HTTP klienta pro provolání REST API nástrojem NSwagStudio. 83
Obrázek 8: Snímek webové stránky klienta pro spravování souborů. 90

SEZNAM TABULEK

Tabulka 1: Vyhodnocení srovnání open-source aplikací pro sdílení dat. 28
Tabulka 2: Funkční požadavky prototypu aplikace. 31
Tabulka 3: Nefunkční požadavky prototypu aplikace. 33

SEZNAM PŘÍLOH

Příloha P I: Uploadify – zdrojový kód na CD