Webový správce hesel s možností rozdělení do skupin a rolí

Jindřich Caletka

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:	**Jindřich Caletka**
Osobní číslo:	**A21527**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Webový správce hesel s možností rozdělení do skupin a rolí**
Téma práce anglicky:	**A Web-Based Password Manager with the Ability to Divide into Groups and Roles**

Zásady pro vypracování

1. Nastudujte a rozveďte problematiku v kontextu tématu práce.
2. Navrhněte webovou aplikaci pro správu hesel, která bude umožňovat skupinové sdílení.
3. Návrh realizujte za využití tokenů.
4. Zvolte vhodné prostředky a technologie pro implementaci.
5. Implementujte vlastní řešení a otestujte.
6. Výsledky vhodně prezentujte a popište.

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

Seznam doporučené literatury:

1. SULLIVAN, Bryan a Vincent LIU. *Web Application Security, A Beginner's Guide*. McGraw Hill, 2011. ISBN 978--0071776165.
2. BURNETT, Mark. *Perfect Password: Selection, Protection, Authentication*. Syngress, 2005. ISBN 978-1597490412.
3. PINE, David. *Learning Blazor: Build Single-Page Apps with WebAssembly and C#*. O'Reilly Media, 2022. ISBN 978--1098113247.
4. SMITH, Jon. *Entity Framework Core in Action, Second Edition*. 2nd edition. Manning, 2021. ISBN 978-1617298363.
5. PEYROTT, Sebastián. *The JWT Handbook* [online]. Auth0, 2018 [cit. 2023-11-11]. Dostupné z: https://auth0.com/resources/ebooks/jwt-handbook

Vedoucí bakalářské práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **26. července 2024**
Termín odevzdání bakalářské práce: **23. srpna 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 29. července 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 …….

podpis studenta

ABSTRAKT

Vývoj správce hesel jako webové aplikace je tím, čím se tato bakalářská práce zabývá. Principy autentizace a ověřování identity byly zkoumány v teoretické části, včetně zkoumání důležitosti silných hesel, dvoufaktorové autentizace a bezpečnostních aktualizací. Byla popsána odolná, avšak zapamatovatelná hesla spolu s technikami jejich vytváření, jako je například hashování a solení.

Byla vysvětlena potřeba správců hesel na základě limitací tradičního přístupu k heslům. Bylo definováno, co je správce hesel, jeho vývoj, bezpečnostní aspekty a různé typy. Závěr byl zaměřen na budoucnost správy hesel s využití AI, blockchainu, kvantových výpočtů, důležitost kybernetické bezpečnosti a nové technologie jako Passkeys.

V praktické části bylo popisováno plánování vývoje aplikace, včetně výběru technologií. Byly probírány i přípravné kroky. Dále byly zdůrazněny implementační detaily jako propojení s databází, vývoj klientské a serverové části v ASP.NET Core, rovněž správa repozitářů a služeb. Nakonec byla zmíněna konfigurace serverových částí aplikace.

Klíčová slova: Blazor, Web API, ASP.NET Core, Správce hesel, Webová aplikace, JWT

ABSTRACT

The development of a password manager as a web application is the focus of this bachelor thesis. The principles of authentication and identity verification were examined in the theoretical part, including an exploration of the importance of strong passwords, two-factor authentication, and security updates. Resilient yet memorable passwords were described along with techniques for their creation, such as hashing and salting.

The need for password managers was explained based on the limitations of the traditional approach to passwords. The role of a password manager, its development, security aspects, and various types were defined. The conclusion was focused on the future of password management utilizing AI, blockchain, quantum computing, the importance of cybersecurity, and new technologies like Passkeys.

In the practical part, the planning of the application's development was described, including the selection of technologies. Preparatory steps were also discussed. Furthermore, implementation details were emphasized, such as database integration, development of the client and server sides in ASP.NET Core, as well as repository and service management. Finally, the configuration of the server components of the application was mentioned.

Keywords: Blazor, Web API, ASP.NET Core, Password manager, Web application, JWT

Chtěl bych poděkovat panu Ing. Petru Žáčkovi, PhD. za podporu a znalosti, které mi daly jasnou představu, jak by měla moje bakalářské práce vypadat.

OBSAH

ÚVOD

V době, kdy digitální identita získává na stále větším významu, nám otázky bezpečnosti účtů a hesel připomínají, jak je důležité chránit naše digitální stopy. Existuje spousta aplikací, do kterých se denně registrují a přihlašují miliony lidí. Někteří pak mají několik účtů se stejnými přihlašovacími údaji, nebo používají externího poskytovatele identity, např. přihlášení pomocí Google účtu. Jistě, je to snadný způsob a člověk si nemusí pamatovat heslo, ale o to jednoduší je to pak pro člověka, který chce zneužít vašeho účtu.

Z toho důvodu se obecně doporučuje používat unikátní a složitá hesla. Složitá v tom smyslu, že jsou dlouhá, skládající se z malých a velkých písmen, čísel a speciálních znaků. Pamatovat si taková hesla může být složité a určitě nikoho nepotěší, když zadá své složité heslo a po kliknutí na přihlásit dostane hlášku, že zadané heslo není správné.

Jako řešení se nabízí správce hesel, který uživateli umožní vygenerovat náhodná a dlouhá hesla, které si do aplikace uloží a může si ho jednoduše zkopírovat nebo mu aplikace přímo navrhne, že sama vyplní přihlašovací formulář. Právě tohle je tématem téhle bakalářské práce s cílem takový správce hesel vytvořit. Další předností bude rozdělení do skupin, kde administrátoři budou moci přidávat uživatele a poskytovat jim sdílené přihlašovací údaje.

I. TEORETICKÁ ČÁST

1 AUTENTIZACE

Autentizace je pojem, který označuje proces dokazování pravosti nějaké skutečnosti nebo dokumentu. V informatice je tento termín spojován s prokazováním identity uživatele. Obvykle uživatel prokazuje svou totožnost poskytnutím svých přihlašovacích údajů, tj. dohodnuté informace sdílené mezi uživatelem a systémem. [1]

1.1 Přihlašovací údaje

Kombinace uživatelského jména a hesla je nejoblíbenějším mechanismem ověřování a je také známá jako ověřování heslem. [1]

Známým příkladem je přístup k uživatelskému účtu na webových stránkách nebo u poskytovatele služeb, jako je Facebook nebo Gmail. Před přístupem k účtu je nutné prokázat, že uživatel vlastní správné přihlašovací údaje. Služby obvykle zobrazí obrazovku, která požaduje zadání uživatelského jména spolu s heslem. Poté porovnají údaje vložené uživatelem s hodnotami dříve uloženými v interním úložišti. [1]

Po zadání platné kombinace těchto údajů, poskytovatel služby umožní pokračovat a umožní přístup k uživatelskému účtu. [1]

Zatímco uživatelské jméno může být veřejné, jako například e-mailová adresa, heslo musí být důvěrné. Kvůli své důvěrnosti musí být hesla chráněna před krádežemi ze strany kyberzločinců. Ačkoli jsou totiž uživatelská jména a hesla na internetu široce používána, jsou notoricky známá jako slabý bezpečnostní mechanismus, který hackeři pravidelně zneužívají. [1]

1.2 Faktory ověřování

Určitá kategorie pověření, jako je uživatelské jméno a heslo, se obvykle označuje jako autentizační faktor. I když je ověřování pomocí hesla nejznámějším typem ověřování, existují i další ověřovací faktory. Existují tři typy autentizačních faktorů, které se obvykle klasifikují následovně:

- Něco, co je vám známo, například heslo.
- Něco, co je vámi vlastněno, například chytrý telefon.
- Něco, co je ověřováno na vás, například biometrické ověřování. [1]

1.3 Dvoufaktorová autentizace a Biometrie

Dvoufázové ověřování (2FA), též známé jako dvoufaktorové ověřování, je bezpečnostní proces, během něhož jsou při přihlašování k online účtu vyžadovány dvě formy identifikace. Místo jednoduchého přihlášení pomocí uživatelského jména a hesla je přístup k účtu v rámci 2FA podroben opětovnému ověření totožnosti. [2]

1.3.1 Autentikační metody pro 2FA

Při využívání dvoufaktorového ověřování jsou dostupné různé metody ověření identity. Zde je uveden seznam nejčastěji preferovaných možností:

- **Hardwarové tokeny:**

  Hardwarové tokeny mohou být poskytovány ve formě klíčenky, která generuje kódy každých několik sekund až minut. Jedná se o jednu z nejstarších forem dvoufaktorového ověřování. [3]

- **Push notifications:**

  Metoda dvoufaktorového ověřování pomocí push nevyžaduje žádné heslo. Tento typ 2FA odesílá signály na telefon, který vás buď vyzve k schválení/odmítnutí přístupu na webovou stránku nebo do aplikace za účelem ověření vaší totožnosti. [3]

- **Ověření pomocí SMS:**

  Tedy ověření pomocí textových zpráv, lze využít jako formu dvoufaktorového ověřování, když je zpráva odesílána na důvěryhodné telefonní číslo. Uživatel je následně vyzván k interakci s textem nebo k použití jednorázového kódu pro ověření své totožnosti na webu nebo v aplikaci. [3]

- **Hlasové ověřování:**

  Ověřování hlasem funguje podobně jako push oznámení, s tím rozdílem, že identita je potvrzena prostřednictvím automatizace. Hlas vás vyzve k stisknutí klávesy nebo k poskytnutí svého jména a identifikaci. [3]

- **Biometrické ověřování:**

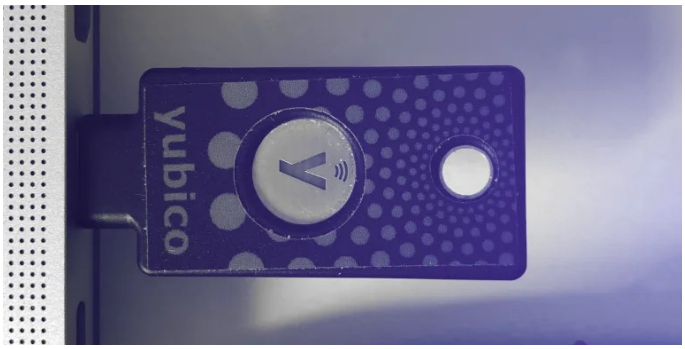
  Biometrické 2FA se stále častěji využívá, kdy jste vy sami bezpečnostním tokenem. Pro prokázání, že jste to vy, kdo se přihlašuje, lze využít otisky prstů, sken obličeje, duhovky či sítnice. [2]

  V mnoha aplikacích je například umožněno využít Touch ID v iPhonu nebo Fingerprint Unlock v systému Android pro přístup k do aplikace po zadání přihlašovacích údajů.

Biometrické ověření často slouží jako třetí způsob ověření totožnosti pro vícefaktorové ověřování (MFA). [2]

Vzhledem k obtížné možnosti padělání biometrických údajů jsou stále častěji využívány i jako přístupové metody, které zcela eliminují používání hesel, známé též jako ověřování bez hesla. [2]

Biometrické ověřování však není bez rizika. Pokud někdo pronikne do biometrického systému 2FA a dojde k úniku nebo replikaci otisku prstu nebo skenu obličeje, nelze je změnit. [2]



Obrázek 1: Hardware Token YubiKey od Yubico

1.4 Aktualizace softwaru a Security patches

Pokud se vyhýbáte aktualizacím softwaru nebo odkládáte jejich provedení, můžete být více vystaveni hrozbám kybernetické bezpečnosti. Věnování času aktualizaci softwaru a opravě bezpečnostních chyb se z dlouhodobého hlediska vyplatí, protože chrání vaše soukromé informace. [4]

1.4.1 Aktualizace softwaru

Zlepšení uživatelské zkušenosti je dosaženo prostřednictvím aktualizací softwaru. Připomínky k aktualizaci softwaru mohou být nepříjemné, avšak mají svůj důvod. Aktualizace softwaru označují soubor změn, které opravují nebo vylepšují software. Tato změna může zlepšit výkon a použitelnost nebo opravit kritické bezpečnostní chyby. V závislosti na softwaru mohou aktualizace probíhat automaticky. Aktualizace lze provádět na veškerém softwaru, ale aktualizace operačního systému jsou nejdůležitější, neboť na něm závisí další software, jako jsou aplikace nebo ovladače. [4]

1.4.2 Security patches

„Software updates include security patches that fix software vulnerabilities and keep malware out of your system.“ [4]

Bezpečnostní nedostatky jsou využívány kybernetickými zločinci. Slabiny nebo díry v softwarových programech, které umožňují proniknutí škodlivého softwaru do zařízení, jsou označovány jako softwarové zranitelnosti. Tyto zranitelnosti následně umožňují hackerům získat přístup k datům, ovládnout počítač nebo zpomalit systém. Bezpečnostní záplaty, které opravují slabá místa softwaru a brání průniku škodlivého softwaru do systému, jsou obsaženy v aktualizacích softwaru. Aktualizace softwaru a operačních systémů by měly být považovány za jednu z nejdůležitějších součástí rutinní kybernetické hygieny, což jsou pravidelné kroky, které mohou být podniknuty ke zlepšení online bezpečnosti. [4]

2 AUTORIZACE

Autorizace nastává poté, co byla identita uživatele úspěšně ověřena (po autentizaci). Je to proces, který určuje, jaká práva a přístupy má ověřený uživatel v rámci systému. Jinými slovy, autorizace definuje, co uživatel smí nebo nesmí dělat.

Existují různé typy autorizace jako Role-Based Access Control, Policy-Based Access Control, nebo JWT.

2.1 Role-Based Access Control

RBAC je jednou z nejběžnějších metod autorizace ve webových aplikacích. Uživatelům jsou přiděleny role, a každá role má určitá práva a oprávnění. Např. administrátoři mohou mít přístup k administrátorskému panelu, zatímco běžní uživatelé pouze ke svým profilům.

2.2 Policy-Based Access Control

PBAC využívá předdefinované politiky pro rozhodování o přístupu. Tyto politiky jsou často definovány pomocí specifických pravidel, která mohou zahrnovat různé podmínky a výjimky. Např. Azure AD.

2.3 JSON Web Token

JWT se často používají k uchovávání informací o uživateli a jeho oprávněních v rámci tokenu, který se předává mezi klientem a serverem. Po úspěšné autentizaci je uživateli vydán JWT, který obsahuje informace o jeho oprávněních. Při každém požadavku aplikace ověřuje a dekóduje JWT, aby určila, zda má uživatel právo provést požadovanou akci.

3 HESLA

V online světě je nutností používat hesla. Jsou považovány za jeden z nejdůležitějších prostředků pro zabezpečení digitálního života. Jsou používána k zabránění komukoli jinému v přístupu k cizím bankovním účtům, e-mailům, účtům na sociálních sítích a všemu ostatnímu, čemu se lidé věnují online. [5]

Proto je důležité používat silná hesla, která nejsou uhodnutelná nikým jiným. Důležité je také, aby hesla nebyla používána opakovaně na různých webových stránkách a službách, protože pokud jsou údaje získány protivníkem z jedné stránky (například v důsledku úniku dat), mohou být tyto údaje použity k přístupu ke všem dalším službám, které sdílejí stejné přihlašovací údaje. [5]

Pro skutečně bezpečné heslo je vyžadováno použití kombinace velkých a malých písmen, číslic a symbolů a mělo by mít alespoň osm znaků (čím delší, tím lepší). Bohužel, si lidský mozek nezapamatuje dlouhé a složité řetězce typu *g1y2<gU+9fnq;smg0:;+*, natož na zapamatování několika takových řetězců pro každou webovou službu. [5]

3.1 Složité, ale zapamatovatelné heslo

Existují různé způsoby, jak si vymyslet složité a zároveň zapamatovatelné heslo.

3.1.1 Použití mnemotechnického prostředku

Hesla, jako je h9!fdjhGH68%J@, jsou bohužel bezpečná, avšak jejich zapamatování není snadné pro člověka. Jedním způsobem, jak to vyřešit, je vytvořit frázi nebo větu, kterou lze snadno zapamatovat. Například: "Skákal pes přes oves".

Vaši frázi lze následně přeměnit na heslo tím, že se využije první písmeno každého slova a přidají se čísla a symboly. Z uvedeného příkladu by heslo mohlo být "SkakalP3s PresOves*".

Posledním krokem při vytváření úspěšné mnemotechnické pomůcky je spojení hesla s mentálním obrazem, který vám pomůže si jej zapamatovat. Například vzpomínka na lidovou píseň pro děti, kterou jistě většina lidí zná, může vyvolat vzpomínku na heslo.

3.1.2 Použití náhodné heslové fráze

Další možností je použití řetězce náhodných, avšak zapamatovatelných slov. Například může být vytvořen řetězec "Blue Tiger Pizza Rainbow" (mezi slovy ponechané mezery,

které zvyšují složitost). Tento druh náhodné heslové fráze lze snadno vytvořit pomocí skvělého, nenáročného nástroje, kterým je Diceware, nebo můžete využít generátor ve správci hesel. [6]

3.2 Odolnost hesel

Navíc, silná hesla používaná k ověřování mohou být odolná vůči útokům hrubou silou na jednu stranu, ale na druhou stranu jsou nepoužitelná proti útokům typu phishing a keylogger software nebo password stuffing. Tyto typy útoků nejsou zaměřeny na odhadnutí hesla uživatele, ale přímo na jeho krádež. [1]

3.3 Prolamování hesel

Kdysi specializovaná dovednost je nyní dostupné téměř všem. Kdokoli pomocí široce dostupných nástrojů, jako jsou John the Ripper, Medusa nebo Cain & Abel, může zkoušet prolomit heslo. [7]

Prolamování hesel je proces, při kterém se zašifrované nebo zahashované heslo - formy, v níž jsou hesla v systémech obvykle uložena - přemění zpět do původní podoby otevřeného textu, kterou vytvořil uživatel. Tento druh dešifrování hesla se nazývá prolomení pouze tehdy, pokud se provádí mimo proces ověřování. [8]

„*Hashování není šifrování.*“ – pan Ing. Petr Žáček, Ph.D.

Záměr prolamování hesel může být jak benevolentní, tak i zlomyslný. Správci systémů tak mohou využívat prolamování hesel ke kontrole síly hesel, aby eliminovali ta slabá. Mezitím mohou hackeři pomocí nástrojů pro prolamování hesel získat neoprávněný přístup k různým účtům a systémům a vykrást citlivé údaje. [8]

Dobré je taky znát způsoby, jakými systémy hesla ukládají.

3.3.1 Otevřený text, šifrování a hash

Tři základní metody, které mohou být použity systémem k uložení hesla. Pokaždé, co uživatel zadá heslo, musí být systémem určena metoda, jak zjistit, zda je heslo správně zadané. Musí být *„něco“* uloženo. První a nejzřejmější metodou je prosté uložení hesla přesně tak, jak bylo zadáno. Tato metoda otevřeného textu ukládá data bez jakéhokoli zastírání, šifrování nebo kódování. [7]

Další metodou je šifrování každého hesla před jeho uložením např. do databáze. Šifrování kombinuje otevřený text s dalším tajným klíčem, čímž vzniká zkomolený řetězec, který lze získat pouze pomocí stejného klíče. Jinými slovy, šifrování je pouze uložení hesla chráněného heslem. Opět, kdokoli s tímto hlavním heslem by měl přístup k celé databázi, takže je jen o něco málo bezpečnější než prostý text. [7]

Šifrování hesel je obecně pro mnoho účelů nepřijatelné, ale je to rozhodně lepší než otevřený text. Někdy aplikace musí uložit heslo jako slovo a načíst otevřený text pro pozdější použití, a to nelze obejít. Například systém Windows šifruje a ukládá různá hesla, aby mohl spouštět systémové služby a připojovat se k různým zdrojům. Často se s tím setkáte, když se objeví přihlašovací dialogové okno a vaše heslo je již zadáno, reprezentováno řetězcem hvězdiček či teček. [7]

Při ukládání hesel je běžnou praxí použití hashovacích funkcí. Hash je výsledkem algoritmu, který zpracovává vstupní heslo a vytváří z něj specifický řetězec znaků, který ho reprezentuje. Tyto algoritmy jsou jednosměrné, což znamená, že není možné z hash hodnoty zpětně získat původní heslo. Pro ověření hesla systém použije stejný hashovací algoritmus na zadaný vstup a porovná výsledek s údaji uloženými v hashovací databázi. Shodují-li se, systém ví, že obě hesla musela být totožná, aby vytvořila stejný hash. [7]

3.3.2 Hash a Solnička

I když hash nelze přímo převést zpět, hackeři mají technické možnosti provést útoky na hashe, což jim umožňuje poměrně přesně odhadnout původní vstup. S přístupem k hashům webových stránek a trochou času mohou být schopni zjistit heslo uživatele v prostém textu.

Právě v takových situacích může být solení hesel užitečné. Solení hesla zahrnuje přidání dalších znaků do otevřeného textu hesla před zahashováním.

3.4 Limitace tradičního přístupu k heslům a Potřeba správce hesel

Navzdory snaze o využívání silných hesel k posílení bezpečnosti existují stále hrozby, kterým je tento tradiční přístup neodolný. S rostoucím komplexním prostředím kybernetických hrozeb je zapotřebí inovativního přístupu k ochraně hesel. Moderní nástroje, známé jako správci hesel, nabízejí efektivní řešení těchto výzev a umožňují správcům systémů lépe řídit a chránit přístupové údaje.

4 DEFINICE SPRÁVCE HESEL

Správce hesel je počítačový program (aplikace), která je schopna generovat bezpečná hesla a bezpečně je ukládat. Také ukládá hesla, vytvořená uživatelem. Většina moderních správců hesel umožňuje snadné zadávání těchto uložených hesel (a dalších přihlašovacích údajů, například uživatelského jména) při přihlašování na webové stránky a do aplikací.

Protože jsou však klíče uchovávány k digitálnímu životu uživatele, je velmi důležité, aby byl samotný správce hesel bezpečný.

Používají tedy jisté hlavní heslo, aby zabránili neoprávněnému přístupu. To znamená, že uživatelé takové aplikace si musí pamatovat pouze jejich hlavní heslo, místo aby si pamatovali hesla ke všem svým účtům. [1][5]

4.1 Vývoj

Vývoj správců hesel odráží neustálou snahu o zvýšení bezpečnosti od počátků hesel v prostém textu až po dnešní sofistikované šifrovací algoritmy. Sledování historie správců hesel odhaluje příběh inovací, odolnosti a přizpůsobování se neustále se měnícímu prostředí hrozeb. [9]

4.2 Bezpečnost

Správci hesel jsou koncipováni s cílem zajistit bezpečnost vašich hesel; avšak otázka, jak jsou sami o sobě bezpeční, přetrvává. Ve hře je mnoho faktorů. Obecně lze konstatovat, že kvalitní správce hesel je navržen s ohledem na ochranu před kyberzločinci a zajištění soukromí vašich údajů. Úspěch této ochrany však značně závisí na konkrétní službě. [10]

Existuje několik společných prvků, které nejlepší správci hesel sdílejí. Prvním z nich je inteligentní používání šifrování. Kvalitní správce hesel šifruje trezory, kde ukládáte svá hesla, pomocí nejmodernějšího šifrovacího algoritmu, jako je například AES-256. [10]

Pokud však služba skutečně dbá na soukromí uživatelů, využívá také end-to-end šifrování. Při běžném šifrování jsou šifrovací klíče sdíleny jak uživatelem, tak službou, při odesílání dat z vašeho počítače na servery správce hesel. [10]

Naopak end-to-end šifrování vám poskytuje klíč pouze vám. Tím je eliminována možnost sledování ze strany služby při přenosu i ukládání dat. I když je implementace této technologie technicky obtížná, někteří správci hesel ji nenabízejí. [10]

V případě, že služby nevyužívají end-to-end šifrování, mohou mít následky katastrofální důsledky. Největší únik dat v historii cloudového úložiště, k němuž došlo v roce 2012 u služby Dropbox, byl způsoben opakovaným používáním zaměstnance této služby jeho hesel, což umožnilo hackerům proniknout do systému. Hackeři poté dešifrovali databázi a ukradli hesla 70 milionů uživatelů. V případě využívání end-to-end šifrování by k takovému incidentu nemohlo dojít. [10]

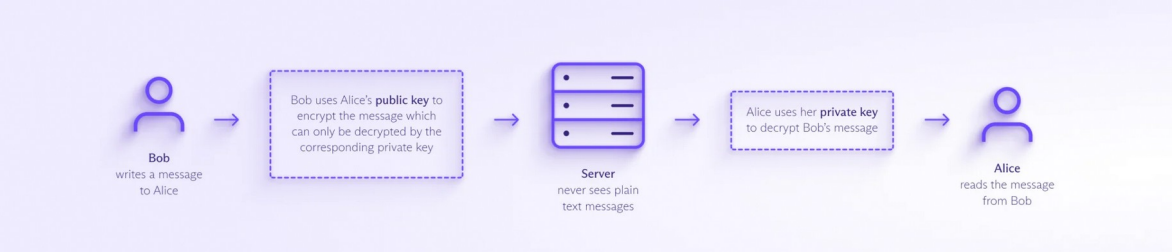
Používání bezpečnostních opatření, jako je end-to-end šifrování, minimalizuje riziko lidské chyby a s ní spojenou pravděpodobnost narušení. [10]

4.2.1 End-to-end šifrování

E2EE šifrování je procesem transformace dat, která jsou normálně čitelná pro lidi (například e-mail s otevřeným textem), do nečitelného šifrovaného textu, jenž může být rozluštěn pouze legitimními stranami prostřednictvím správného kryptografického klíče. [11]

Termín "šifrování end-to-end" popisuje metodu šifrování, při níž jsou data zakódována v každém kroku své cesty z jednoho zařízení na druhé. "End-to-end" označuje počáteční a konečný bod trasy dat. Například při odesílání e-mailu je počátečním bodem vaše zařízení a cílovým bodem je zařízení příjemce. [11]

Takovým způsobem E2EE reprezentuje bezpečný způsob komunikace, který brání všem třetím stranám v přístupu k obsahu vašich zpráv, a to jak během přenosu z jednoho zařízení na druhé, tak i "v klidu" na serveru. Když někomu pošlete e-mail pomocí E2EE, obsah vaší zprávy zůstává nedostupný pro každého kromě příjemce, který má jediný klíč umožňující dešifrování obsahu. [11]



Obrázek 2: Proces E2EE

4.3 Typy

Optimální výběr správce hesel pro vaše potřeby závisí na několika faktorech. Je preferováno využívání vestavěného správce hesel v zařízení nebo prohlížeči před absencí

takového. Je třeba však mít na paměti, že jsou omezené a nabízejí podstatně menší hodnotu ve srovnání s bezplatným cloudovým správcem hesel. [12]

4.3.1 Cloudový správci hesel

Patří mezi oblíbené možnosti pro jednotlivce i firmy. Hesla a citlivé údaje jsou šifrovány a chráněny na vlastních serverech. Hlavní výhodou je snadný přístup odkudkoli pomocí jakéhokoli počítačového zařízení. Služba je centrálně hostována a udržována poskytovatelem služby. [12]

4.3.2 On-premise (lokální) správci hesel

I když jsou dostupné mnohé cloudové aplikace pro správu hesel v mobilních zařízeních, k dispozici jsou i nativní správci hesel v systémech iOS a Android, jako jsou Apple Keychain a Google Password Manager, přičemž umožňují bezpečné ukládání hesel do mobilního zařízení a automatické vyplňování hesel na webových stránkách a v aplikacích. [12]

4.3.3 Open-source správci hesel

Transparentnost a možnost přizpůsobení činí open-source správce hesel oblíbenými. Jsou poskytovány zdarma, a jejich veřejně dostupný zdrojový kód umožňuje komunitě vývojářů provádět revize bezpečnosti. [13]

4.3.4 Komerční

Na druhé straně, prémiové funkce, jako je podpora zákazníků, týmové sdílení a další pokročilé možnosti, jsou nabízeny komerčními správci hesel. Tyto správce hesel jsou obvykle zpoplatněny, i když některé mohou poskytovat omezené verze zdarma. [13]

4.4 Budoucnost

Budoucnost správců hesel se předpokládá v inovacích, které přinesou novou úroveň zabezpečení a zvládnou výzvy budoucnosti, od biometrického ověřování po decentralizovaná řešení identity. [13]

Hesla prošla vývojem od svých skromných počátků v podobě základních alfanumerických kódů až po složité řetězce vyžadující kombinaci znaků, čísel a symbolů. Tento vývoj odráží zvýšené povědomí o hrozbách kybernetické bezpečnosti. [14]

Svět správy hesel se rychle vyvíjí pod vlivem nových technologií a měnících se potřeb uživatelů. Podívejme se na současný stav vývoje v oblasti správy hesel a zkoumejme, jak tyto trendy formují náš přístup k digitálnímu zabezpečení. [14]

1. Častější používání správců hesel

   S rostoucí složitostí požadavků na hesla roste oblíbenost správců hesel. Kromě pamatování hesel umožňují generování silných a jedinečných hesel pro každý účet. Další významnou výhodou je jejich schopnost automatického vyplňování přihlašovacích údajů, což snižuje riziko phishingu tím, že zajišťuje zadávání přihlašovacích údajů pouze na legitimních stránkách.

2. Vzestup vícefaktorového ověřování (MFA)

   Dalším trendem, který nabírá na síle, je vícefaktorové ověřování. MFA přidává další vrstvu zabezpečení tím, že pro získání přístupu k účtu vyžaduje dva nebo více ověřovacích faktorů. Např. heslo, PIN, otisk prstu nebo rozpoznání obličeje.

3. Biometrické ověřování

   Tahle oblast se rychle stává klíčovým hráčem v oblasti správy hesel. Tato metoda využívá k ověření identity jedinečné biologické charakteristiky, jako jsou otisky prstů, rozpoznávání obličeje, skenování duhovky nebo rozpoznávání hlasu.

   - Rozpoznávání otisků prstů a obličeje

     Tyto technologie nabízejí pohodlný a bezpečný způsob odemykání zařízení a ověřování totožnosti uživatelů. Novější modely jsou schopny 3D mapování a pokročilých rozpoznávacích algoritmů, které zabraňují falšování a falešným poplachům.

   - Skenování duhovky a rozpoznávání hlasu

     Tyto technologie, méně běžné, ale v prostředí s vysokým stupněm zabezpečení se stále více prosazují. Skeny duhovky poskytují vysokou úroveň přesnosti díky jedinečným vzorům v duhovce každého člověka. Rozpoznávání hlasu, i když pohodlné, se stále potýká s problémy při přesném zachycení hlasových vzorů, zejména v hlučném prostředí. [14]

Biometrické ověřování představuje významný posun oproti tradičním systémům hesel a nabízí kombinaci vyšší bezpečnosti a uživatelského pohodlí. Vyvolává však také důležité

obavy o ochranu soukromí, protože biometrické údaje jsou vysoce citlivé a v případě kompromitace je nelze změnit jako heslo. [14]

S tím, jak se pouštíme do oblasti nových technologií, je jasné, že inovativní řešení budou určovat budoucnost správy hesel. Převratné pokroky, které jsou připraveny v příštích letech, budou nově definovat zabezpečení hesel. [14]

4.4.1 Umělá inteligence ve správě hesel

Revoluci ve správě hesel přináší umělá inteligence prostřednictvím prediktivní analýzy a automatického generování silných hesel. Algoritmy umělé inteligence jsou schopny analyzovat chování uživatelů, předvídat potenciální bezpečnostní hrozby a navrhovat změny k zvýšení bezpečnosti. Správci hesel s umělou inteligencí navíc mohou generovat složitá hesla, která jsou pro hackery téměř neuhádnutelná, ale pro uživatele je správa jednodušší. [14]

4.4.2 Technologie blockchain pro bezpečnou správu hesel

Získávající pozornost technologie blockchain přináší decentralizaci a zvyšuje bezpečnost systémů pro správu hesel. Ukládáním hesel v decentralizované síti se výrazně snižuje riziko narušení centralizované databáze. Vlastnosti blockchainu, jako je neměnnost a transparentnost, poskytují robustní rámec pro bezpečné procesy ověřování. [14]

4.4.3 Kvantová výpočetní technika a zabezpečení hesel

Nástup kvantové výpočetní techniky představuje výzvy i příležitosti pro zabezpečení hesel. Kvantové počítače by potenciálně mohly prolomit současné metody šifrování, ale také otevírají cestu k vývoji šifrovacích algoritmů odolných vůči kvantům. Tato nově vznikající oblast je připravena radikálně změnit koncepci a implementaci zabezpečení hesel. [14]

4.4.4 Kybernetická bezpečnost a vzdělávání uživatelů

V oblasti správy hesel je nejdůležitější vzdělávání uživatelů. Informovanost o osvědčených postupech kybernetické bezpečnosti může výrazně snížit riziko narušení bezpečnosti. To zahrnuje pochopení významu používání silných a jedinečných hesel, rozpoznávání pokusů o phishing a bezpečné používání správců hesel a metod ověřování.

Jednotliví uživatelé i organizace musí přijmout osvědčené postupy pro správu hesel. Uživatelé by měli používat renomované správce hesel, zapínat funkci MFA a být ostražití

v oblasti online bezpečnosti. Organizace by měly prosazovat přísné zásady používání hesel, provádět pravidelné bezpečnostní audity a poskytovat zaměstnancům školení o rizicích kybernetické bezpečnosti a preventivních opatřeních. [14]

4.4.5 Předpovědi do budoucna a názory odborníků

Názory a předpovědi odborníků nabízejí cenné pohledy na směřování správy hesel do budoucna. Předpokládá se větší integrace umělé inteligence a biometrie do autentizačních procesů a potenciální posun k decentralizovanějším systémům, jako je blockchain. Dalším velkým skokem ve správě hesel by mohlo být široké přijetí ověřování bez hesla. [14]

Správci hesel také mohou nabídnout užitek z poskytování informací o kreditních kartách, adresách a dalších osobních údajích uživatelů. Tím umožňují pohodlnější a bezpečnější nakupování online. [15]

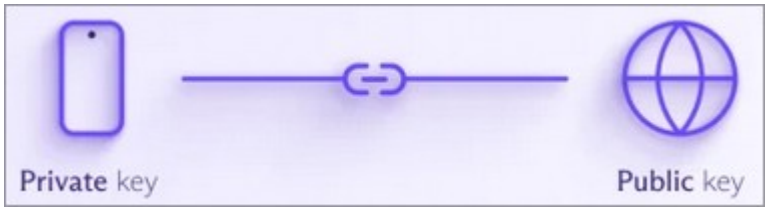
"*The future of password management is likely to be a combination of passwordless authentication and password managers. Passwordless authentication is undoubtedly the future, as it offers significant advantages over traditional password-based authentication. However, it's unlikely to completely replace password managers, as they still offer value in managing other sensitive data and syncing passwords across multiple devices,*" - Kevin Garce, Vedoucí marketingu ve společnosti iwoolfelt [15]

4.4.6 Passkeys

Passkeys jsou formou identifikace, která může být použita k získání přístupu k účtu. Nahrazují potřebu hesla a dvoufaktorové ověřování (2FA) tím, že k identifikaci uživatele využívají zařízení. Díky tomu, že není potřeba zadávat heslo, které by mohlo být ukradeno, passkeys chrání před phishingem. Jsou také méně náchylné k útokům hrubou silou, protože fungují na základě vytváření kryptografických klíčů. [16]

Obvykle získáte přístup k účtu zadáním pověření, které jste uvedli při jeho vytváření: uživatelské jméno (často e-mailová adresa) a heslo. Ne však v případě passkeys. Když si vytvoříte účet u služby, která podporuje přístupové klíče, vygeneruje se sada šifrovacích klíčů správcem hesel. Při příštím pokusu o přístup k webu rozpozná klíče, které máte, a přihlásí vás bez nutnosti zadávat heslo. [16]

Využívají princip asymetrické kryptografie neboli kryptografie s veřejným klíčem. Jde o to, že při vytvoření passkeys se vygenerují dva matematicky propojené číselné klíče: jeden veřejný a druhý soukromý. [16]



Obrázek 3: Soukromý a veřejný klíč

Veřejný klíč má služba, ke které se přihlašujete, zatímco vy jako uživatel, máte soukromý klíč, který je uložen ve správci hesel. Při přihlašování ke službě odešle veřejný klíč do vašeho zařízení výzvu, na kterou může správně odpovědět pouze váš soukromý klíč, čímž vás identifikuje jako vlastníka účtu. [16]

Systém je velmi bezpečně navržen a je prakticky odolný proti útokům hrubou silou. Prolomení čísel, která se používají v kryptografii s veřejným klíčem, by vyžadovalo kombinaci kvantových počítačů a velkou spoustu času. [16]

5 VHODNÉ TECHNOLOGICKÉ PROSTŘEDKY

Pro úspěšnou implementaci projektu je nutné vybrat technologické prostředky, se kterými bude možné dosáhnout odpovídajícím požadavkům a cílům. S těmito prostředky je možné efektivně a spolehlivě realizovat navržené funkcionality a konečné aplikace.

5.1 Technogie a Framework pro Webovou Aplikaci

Framework pro vývoj webových aplikací je souborem knihoven, nástrojů a standardizovaných postupů, které poskytují strukturu pro vytváření aplikací. [17]

Proces vývoje je zjednodušen tím, že nabízí opakovaně použitelné komponenty, předpřipravený kód a sadu pokynů, které umožňují vývojářům rychle a efektivně vytvářet webové stránky. Frameworky odstraňují potřebu začínat od nuly, což šetří čas a úsilí a podporuje osvědčené postupy kódování. [17]

Frameworky podporují tvorbu spolehlivých, bezpečných a škálovatelných aplikací s využitím osvědčených postupů v oboru. Vnáší do procesu vývoje řád a strukturu, což umožňuje efektivní spolupráci. Kromě toho frameworky podporují přijetí standardizovaných kódovacích postupů, čímž zvyšují čitelnost kódu, udržovatelnost a celkovou kvalitu výsledného produktu. [17]

5.1.1 ASP.NET Core 8

ASP.NET Core je multiplatformní open-source architektura postavená na vývojářské platformě .NET 8 vytvořený specificky pro vývoj webových aplikací. [18]

ASP.NET Core nabízí následující výhody:

- AntiForgeryToken, což je funkce, která se používá k prevenci útoků typu Cross-Site Request Forgery (XSRF/CSRF). Tyto útoky jsou možné, protože webové prohlížeče automaticky odesílají některé typy autentizačních tokenů s každým požadavkem na webovou stránku.
- Razor Pages usnadňují a zefektivňují kódování scénářů zaměřených na stránky oproti používání controllers a views.
- Blazor umožňuje pracovat s jazykem C# v prohlížeči spolu s JavaScriptem. Je tedy možné sdílet logiku aplikace na straně serveru a na straně klienta, která je napsaná pomocí .NET.

- Podpora poskytování služeb vzdáleného volání procedur (RPC) pomocí gRPC.
- Konfigurační mechanismus založený na prostředí, který je připravený pro cloud.
- Možnost nasazení na následujících zařízeních:
    - Nginx
    - Apache
    - Docker [18]

5.1.2 Blazor WASM

Blazor je front-endový webový framework postavený na ASP.NET Core, který podporuje vykreslování na straně klienta pomocí WebAssembly:

- Pomocí jazyka C# mohou být vytvářena bohatá interaktivní uživatelská rozhraní.
- Logika aplikace je spouštěna přímo v prohlížeči, což umožňuje rychlou a responzivní interakci.
- Uživatelská rozhraní jsou vykreslována jako HTML a CSS, což zajišťuje širokou podporu prohlížečů. [19]

Architektura založená na komponentách s vykreslováním na straně klienta a úplnou interaktivitou je poskytována službou Blazor WebAssembly, kde mohou být vytvářeny plně funkční aplikace bez nutnosti serverového vykreslování. [19]

Uživatelské rozhraní je rychle doručováno do prohlížeče díky statickému vykreslování obsahu HTML, který je součástí aplikace. Stránka je načítána svižně, protože vykreslování uživatelského rozhraní je prováděno přímo v prohlížeči bez potřeby stahování objemné sady JavaScriptu. Uživatelské prostředí může být dále vylepšováno různými progresivními metodami, jako je zlepšená navigace a asynchronní načítání obsahu. [19]

Interaktivita při vykreslování na straně klienta (CSR) je podporována službou Blazor WebAssembly, která využívá modul runtime .NET vytvořený pomocí WebAssembly. Při spuštění Blazoru na WebAssembly je vašemu kódu .NET umožněn přístup k plné funkčnosti prohlížeče a spolupráce s JavaScriptem. Kód .NET je spouštěn v sandboxu zabezpečení prohlížeče, což poskytuje ochranu proti škodlivým akcím na klientském počítači. Blazor aplikace mohou být zcela provozovány na WebAssembly v prohlížeči bez zapojení serveru. U samostatné Blazor WebAssembly aplikace jsou zdroje nasazovány jako statické soubory na webovém serveru nebo službě, které mohou obsluhovat klientům

statický obsah. Po stažení mohou být samostatné Blazor WebAssembly aplikace ukládány do mezipaměti a spouštěny offline jako progresivní webové aplikace (PWA). [19]

5.1.3 ASP.NET Core Web API

V projektu Web API jsou podporovány jak tradiční API založené na controllerech, tak i minimal API. API založené na controllerech umožňuje, aby byl použit strukturovaný přístup k tvorbě API pomocí MVC (Model-View-Controller) architektury, zatímco minimal API nabízí jednodušší a rychlejší způsob zahájení vývoje API, což je ideální pro menší projekty nebo mikroslužby. [20]

Různé strategie pro zlepšení výkonu jsou nabízeny frameworkem, jako je caching, stránkování, optimalizace databázových dotazů a asynchronní programování. Těmito technikami je zajištěno, že API dokáže efektivně zpracovávat velké množství požadavků a poskytovat rychlé odezvy. [20]

Dále jsou nabízeny robustní mechanismy autentizace a autorizace. Podpora pro JWT umožňuje, aby byla prováděna bezpečná autentizace uživatelů a aby byly chráněny API endpointy pomocí autorizačních atributů. Tím je zajištěno, že k citlivým datům a funkcím mají přístup pouze oprávnění uživatelé. [20]

Pro snadný přístup k databázím a provádění CRUD operací (Create, Read, Update, Delete) může být využit Entity Framework Core. Tento ORM (Object-Relational Mapping) nástroj umožňuje, aby bylo s databázemi pracováno pomocí objektově orientovaného přístupu, což zjednodušuje správu dat a zvyšuje produktivitu. [20]

5.1.4 Entity Framework ORM

Relační část je systémem pro řízení relačních databází (tedy databází). Existují i jiné druhy databází, ale oblíbenější je relační (tabulky, sloupce, primární klíč, cizí klíč atd. jsou známé, např. Oracle, MySQL, MSSQL). [21]

A konečně mapovací sekce je částí, kde se uskutečňuje propojení mezi objekty a tabulkami. [21]

Entity Framework Core je odlehčenou, rozšiřitelnou a multiplatformní verzí oblíbené technologie pro přístup k datům Entity Framework. [21]

EF Core může fungovat jako objektově relační mapovač (ORM), což:

- Dovoluje vývojářům pracovat s databází pomocí objektů.

- Odstraňuje potřebu většiny kódu pro přístup k datům, který je obvykle nutno sepsat.

EF Core podporuje mnoho databázových enginů. [21]

5.2 Vývojářské prostředí

Integrované vývojové prostředí (IDE) je softwarová sada, ve které jsou všechny vývojové nástroje spojeny do jediného grafického uživatelského rozhraní. Díky tomu může být proces vývoje efektivněji a rychleji realizován. Příklady oblíbených integrovaných vývojových prostředí jsou Microsoft Visual Studio nebo produkty od JetBrains. [22]

5.2.1 Typy integrovaných vývojových prostředí

Je důležité, aby byly zváženy různé aspekty každého projektu, protože může být vyžadováno jiné IDE. Aspekty jako programovací jazyk, snadnost použití, spolehlivost a typ musí být zváženy. Existuje mnoho typů IDE, které nabízejí různé funkce, a tyto funkce umožňují efektivní vytváření kvalitních aplikací. [22]

5.2.2 JetBrains Rider

Pro vývojáře v jazycích C# a .NET je určeno multiplatformní IDE Rider, přičemž podporovány jsou i další jazyky. Díky své inteligentní architektuře je Rider oproti mnoha jiným IDE výrazně rychlejší. Za účelem zefektivnění procesu vývoje je poskytována rozsáhlá sada funkcí. Od inteligentního dokončování kódu a detekce chyb za běhu až po robustní nástroje pro refaktorizaci, je vše k dispozici k rozšíření možností kódování.

Uživatelské rozhraní aplikace Rider je přehledné a přizpůsobitelné. Postranní panel nástrojů, okno editoru, struktura projektu a další, git branche, spouštěcí profily, commit do cloudové služby, jako je například Azure, projektové okno při startu a další součásti jsou navrženy tak, aby zlepšily a usnadnily práci. [23]

5.2.3 Docker

Docker je platformou pro vývoj, distribuci a spouštění aplikací. Umožňuje oddělit aplikace od infrastruktury, aby bylo možné rychle dodávat software. Infrastruktura je spravována stejným způsobem jako aplikace s využitím nástroje Docker. Využitím metodik nástroje pro distribuci, testování a nasazování kódu může být výrazně zkrácena prodleva mezi napsáním kódu a jeho spuštěním v produkci. [24]

Možností, kterou Docker poskytuje, je zabalit a spustit aplikaci ve volně izolovaném prostředí zvaném kontejner. Izolace a zabezpečení umožňují spustit na jednom hostiteli mnoho kontejnerů současně. Kontejnery jsou odlehčené a obsahují vše potřebné ke spuštění aplikace, takže se nemusí spoléhat na to, co je nainstalováno na hostiteli. Kontejnery lze snadno sdílet a mít jistotu, že každý, s kým jsou sdíleny, dostane stejný kontejner, který funguje stejně. [24]

Dále poskytuje nástroje a platformu pro správu životního cyklu kontejnerů:

- Vývoj aplikace a jejích podpůrných komponent pomocí kontejnerů.
- Kontejner se stává jednotkou pro distribuci a testování aplikace.
- Až bude připravenost dosažena, aplikace může být nasazena do produkčního prostředí jako kontejner nebo orchestrovaná služba. Tento postup je stejný, bez ohledu na to, zda je produkčním prostředím lokální datové centrum, poskytovatel cloudu nebo jejich hybrid. [24]

Docker se používá pro rychlé a konzistentní dodávání aplikací. Životní cyklus vývoje je efektivnější díky Dockeru, který umožňuje vývojářům pracovat ve standardizovaných prostředích pomocí místních kontejnerů. Ty jsou skvělé pro kontinuální integraci a kontinuální dodávání (CI/CD). [24]

- Vývojáři píší kód lokálně a sdílejí svou práci s kolegy pomocí kontejnerů Docker.
- Pomocí Dockeru přesouvají své aplikace do testovacího prostředí a spouštějí automatizované a manuální testy.
- Když vývojáři najdou chyby, mohou je opravit ve vývojovém prostředí a znovu je nasadit do testovacího prostředí k testování a ověření.
- Po dokončení testování je předání opravy zákazníkovi stejně jednoduché jako odeslání aktualizovaného kontejneru do produkčního prostředí. [24]

Kontejnery mohou být spuštěny na lokálním notebooku vývojáře, na fyzických nebo virtuálních strojích v datovém centru, u poskytovatelů cloudových služeb nebo v kombinaci různých prostředí. [24]

Přenositelnost a odlehčená povaha kontejnerů usnadňuje také dynamická správa pracovních zátěží, škálování nebo rušení aplikací a služeb podle potřeb podniku, a to téměř v reálném čase. [24]

Docker využívá architekturu klient-server. Klientem Docker je komunikováno s daemonem Docker, který se stará o sestavování, spouštění a distribuci kontejnerů. Klient a daemon mohou běžet na stejném systému, nebo klient může být připojen k vzdálenému daemonu. Mezi nimi je komunikováno pomocí rozhraní REST API, přes sockety UNIX nebo síťové rozhraní. Dalším klientem je Docker Compose, který umožňuje práci s aplikacemi složenými ze sady kontejnerů. [24]

Kontejnery jsou spustitelnými instancemi image. Mohou být vytvářeny, spouštěny, zastavovány, přesouvány nebo odstraňovány pomocí rozhraní Docker API nebo CLI. Kontejner může být připojen k jedné nebo více sítím, může být k němu připojeno úložiště nebo může být dokonce vytvořen nový image na základě jeho aktuálního stavu. [24]

Ve výchozím nastavení je kontejner relativně dobře izolován od ostatních kontejnerů a svého hostitelského počítače. Úroveň izolace sítě, úložiště nebo jiných základních subsystémů kontejneru od ostatních kontejnerů nebo od hostitelského počítače může být ovlivněna. [24]

Kontejner je definován svou bitovou kopií a všemi konfiguračními možnostmi, které mu jsou poskytnuty při jeho vytváření nebo spuštění. Při odstranění kontejneru jsou ztraceny veškeré změny jeho stavu, které nejsou uloženy v trvalém úložišti. [24]

Image je šablonou pouze pro čtení s pokyny pro vytvoření kontejneru. Často je založena na jiném image s určitými dalšími úpravami. Například může být vytvořen image, který je založen na image ubuntu, ale nainstaluje webový server Apache a vaši aplikaci, spolu s konfiguračními údaji potřebnými k jejímu spuštění. [24]

Vlastní images mohou být vytvářeny nebo mohou být používány ty, které byly vytvořeny jinými a jsou zveřejněny v registru. Chcete-li vytvořit vlastní image, musí být vytvořen soubor Dockerfile s jednoduchou syntaxí pro definování kroků potřebných k vytvoření obrazu a jeho spuštění. Každým pokynem v souboru Dockerfile je vytvářena vrstva image. Když je soubor Dockerfile změněn a image je obnoven, jsou obnoveny pouze ty vrstvy, které se změnily. Tím se image stávají lehkými, malými a rychlými ve srovnání s jinými virtualizačními technologiemi. [24]

5.2.4 Docker Compose

Docker Compose je nástrojem pro definování a spouštění aplikací s více kontejnery. [25]

Compose je nástrojem, který zjednodušuje kontrola celého vašeho aplikačního stacku a usnadňuje správa služeb, sítí a svazků v jednom srozumitelném konfiguračním souboru

YAML. Poté jedním příkazem jsou vytvořeny a spuštěny všechny služby z vašeho konfiguračního souboru. [25]

Compose je používán ve všech prostředích; produkci, stagingu, vývoji, testování, a také pracovních postupech CI. Má také příkazy pro správu celého životního cyklu vaší aplikace:

- Služby jsou spouštěny, zastavovány a přebudovávány
- Stav běžících služeb je zobrazován
- Výstup protokolu běžících služeb je streamován
- Jednorázový příkaz je spouštěn na službě [25]

5.2.5 MSSQL Server a Databáze

Microsoft SQL Server je relačním systémem pro správu databází (RDBMS). K instanci nebo databázi SQL Serveru je připojováno aplikacemi a nástroji a je s nimi komunikováno pomocí jazyka Transact-SQL (T-SQL). [26]

Server SQL Server může být nainstalován do systému Windows nebo Linux, nasazen v linuxovém kontejneru nebo nasazen na virtuálním stroji Azure či jiné platformě virtuálního stroje. [26]

SQL Server se skládá ze dvou hlavních součástí:

- Databázový engine
- SQLOS [27]

Základní komponentou SQL Serveru je Database Engine, který se skládá z relačního enginu, který zpracovává dotazy, a úložného enginu, který spravuje databázové soubory, stránky, indexy atd. Navíc vytváří databázové objekty, jako jsou uložené procedury, pohledy a spouštěče. [27]

Relační engine obsahuje součásti, které určují optimální způsob provedení dotazu. Je také známý jako procesor dotazů. Relační engine si na základě vstupního dotazu vyžádá data z úložného enginu a zpracuje výsledky. Mezi jeho úkoly patří zpracování dotazů, správa paměti, správa vláken a úloh, správa vyrovnávací paměti a distribuované zpracování dotazů. [27]

Úložný engine je zodpovědný za ukládání a načítání dat z úložných systémů, jako jsou disky a SAN. [27]

Pod relačním enginem a úložným enginem se nachází operační systém SQL Server Operating System neboli SQLOS. Poskytuje různé služby operačního systému, jako je

správa paměti a vstupně-výstupních operací, stejně jako zpracování výjimek a synchronizační služby. [27]

SQL server je možné rychle nainstalovat na Windows nebo Linux, ale pohodlnější možnost je rozběhnout SQL server v Docker kontejneru. Lépe řečeno, není potřeba žádná instalace, pokud již máte nainstalovaný Docker. Stačí si stáhnout image z Docker hubu, nebo využít desktop aplikace Docker desktop, kde je možné prohledávat Docker hub podle názvu image, bez potřeby použít prohlížeč. Po stažení image stačí zadat 3 env proměnné (typ serveru, heslo a přijmout EULA) do spouštění dockeru a je to, výchozí uživatel je pak sa, a typ serveru je výchozí Developer, tudíž tahle env proměnná je volitelná.

SQL je možné spustit i pomocí příkazu v terminálu, není tedy potřeba interaktivní proces uživatele. Např.:

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=yourStrong(!)Password" -p 1433:1433 -d mcr.microsoft.com/mssql/server:2022-latest
```
[28]

5.2.6 Výhody SQL Serveru v Docker kontejneru před lokální instalací

- Cloud-ready

  Je možné snadno spustit řešení na VPS, nebo GCP, AKS, nebo AWS. Vaše kontejnery se dají spustit na jakémkoli místě.

- Lepší ceny

  Testování řešení na Linux runtime je možné, což vám ušetří peníze. To je proto, že virtuální servery na Windows jsou více nákladné než virtuální servery na Linux.

- Testování proti různým serverům/verzi

  Je možné testovat řešení s různými verzemi SQL serveru, stačí změnit variantu image nebo typ serveru v prostředí proměnné.

- Izolace

  Lze snadno vytvořit samostatné mosty sítě s SQL serverem, kontrolovat přístup. Můžete spustit několik instancí na jednom počítači najednou, snadno, pouze oddělováním sítí pomocí prostředků Docker.

- Resetování

Testování vyžaduje, aby všechny změny mohly být resetovány a všechny testy mohly být spuštěny od začátku (od stejného počátečního bodu). S kontejnery a svazky toho dosáhnete jedním příkazem.

- Transparentní konfigurace

  Poskytujete Dockerfile a compose.yaml, kde jsou všechny kroky explicitně napsány jasně. Nepotřebujete poskytnout další readme's o tom, jak nastavit váš server.

- Cross-platform

  Konfigurace Dockeru běží na jakémkoli operačním systému bez změn. Možná, že designéři používají MacOS a také chtějí spustit řešení lokálně? Je to snadné s Dockerem. [28]

5.3 Model

Manipulace s daty je prováděna pomocí modelu v EF Core. Model je konstruován třídami entit a kontextovým objektem, který je reprezentací spojení s databází. Dotazování a ukládání dat je umožněno kontextovým objektem. [29]

EF nabízí následující metody k vývoji modelů:

- Model je vytvořen z existující databáze.
- Kód modelu je ručně sepsán tak, aby odpovídal databázi.
- Databáze je vytvořena z modelu pomocí migrací EF po jeho vytvoření. Evoluce databáze je umožněna migracemi při změnách modelu.

Dotazování je proces, během kterého jsou instance tříd entit načítány z databáze pomocí dotazů LINQ (Language Integrated Query). [29]

5.4 Přístup k databázi a update UI

Na nastavení DbContext se EF Core opírá pro přístup k databázi a je využíván jako jednotka práce. Rozhraní pro aplikace ASP.NET Core jsou nabízeny EF Core AddDbContext, které ve výchozím nastavení registrují kontext jako službu jako scoped. V aplikacích na straně Blazor serveru může být registrace vymezených služeb považována za problematickou, protože instance jsou sdíleny mezi komponentami v okruhu uživatele. DbContext není považován za bezpečný pro vlákna a není určen pro souběžné použití. [30]

Stávající životnosti jsou považovány za nevhodné z těchto důvodů:

- Singletonem je sdílen stav napříč všemi uživateli aplikace a vede k nevhodnému souběžnému použití.
- Scoped (výchozí) představuje podobný problém mezi komponentami pro stejného uživatele.
- Transientem je vytvořena nová instance na požadavek, ale vzhledem k tomu, že komponenty mohou být dlouhodobé, výsledkem je delší kontext, než bylo zamýšleno. [30]

Následující doporučení jsou navržena tak, aby poskytovala konzistentní přístup k používání EF Core v aplikacích na straně Blazor serveru. Ve výchozím nastavení je doporučeno použití jednoho kontextu pro každou operaci. Kontextem je navrženo rychlé vytváření instancí s nízkou režií:

```
using var context = new MyContext();
return await context.MyEntities.ToListAsync();
```

Pokud chcete zabránit více souběžným operacím, je doporučeno použití příznaku:

```
if (Loading)
{
    return;
}

try
{
    Loading = true;
}
finally
{
    Loading = false;
} [30]
```

Operace jsou umístěny za řádek `Loading = true;` v bloku `try`. Logika načítání nevyžaduje uzamčení záznamů databáze, protože zabezpečení vláken není považováno za problém. Logika načítání je používána k zakázání ovládacích prvků uživatelského rozhraní, aby uživatelé neúmyslně nevybírali tlačítka nebo aktualizovali pole při načítání dat. Pokud existuje nějaká šance, že více vláken může přistupovat ke stejnému bloku kódu, je doporučeno vložit továrnu a vytvořit novou instanci na operaci. V opačném případě je vkládání a používání kontextu obvykle považováno za dostatečné. U délších operací EF Core, které využívají sledování změn nebo řízení souběžnosti, je doporučeno rozsah kontextu na dobu životnosti komponenty. [30]

Co se týče aktualizace uživatelského rozhraní, je důležité zdůraznit, že aktualizace UI by měly být prováděny s ohledem na aktuální stav načítání dat. To znamená, že ovládací

prvky uživatelského rozhraní by měly být zakázány nebo povoleny v závislosti na tom, zda jsou data právě načítána. Toto je klíčové pro zabránění neúmyslného výběru tlačítek nebo aktualizace polí uživatelem během načítání dat.

Pokud jde o CSS, může být také užitečné využít dynamické styly nebo třídy, které reagují na stav načítání dat. Například, můžete mít CSS třídu, která zakáže určité ovládací prvky nebo je udělá méně viditelnými během načítání dat. Tímto způsobem můžete zajistit, že uživatelské rozhraní je vždy v souladu se stavem načítání dat a poskytuje uživatelům jasné vizuální zpětné vazby.

5.4.1 Tailwind CSS

Tailwind CSS je utility-first CSS framework, který je navržen tak, aby umožnil vývojářům rychle a efektivně vytvářet uživatelské rozhraní. Místo toho, aby byly psány vlastní styly pro každou komponentu, je poskytnuta sada malých, znovupoužitelných tříd, které mohou být kombinovány přímo v HTML kódu.

```
<button class="w-10 h-9 flex lg:hidden justify-center
        items-center rounded-md transition-all
        hover:bg-white hover:bg-opacity-10 duration-150">
```

Obrázek 4: Tailwind classes

Fungování Tailwind CSS spočívá v tom, že jsou skenovány všechny soubory, které musí být nastaveny v konfiguračním souboru tailwind.config.js v sekci `content`. Následně jsou generovány odpovídající styly a ty jsou zapsány do statického CSS souboru. To znamená, že je možné vytvářet komplexní designy bez nutnosti psát velké množství vlastního CSS kódu.

Pro instalaci Tailwind CSS je nejjednodušší a nejrychlejší způsob použití nástroje Tailwind CLI s použitím Node.js a npm. Po spuštění příkazů `npm install -D tailwindcss` a `npx tailwindcss init` je Tailwind nainstalován do projektu a je vytvořen konfigurační soubor.

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

Obrázek 5: Tailwind konfigurační soubor

Tailwind lze také využít v Blazor webové aplikaci, ale v tomto případě se styly nesestaví automaticky. Je nutné to udělat pomocí příkazu, který sleduje změny v souborech a podle toho sestaví všechny třídy do jednoho CSS souboru. To znamená, že při nasazení aplikace není nutné mít Node.js na straně serveru.

II. PRAKTICKÁ ČÁST

6 PLÁNOVÁNÍ

Před začátkem vývoje webové aplikace pro správu hesel bylo provedeno důkladné plánování celého projektu. Plánování zahrnovalo několik kroků, které pomohly strukturovat problém, stanovit požadavky, zvolit vhodné technologie a navrhnout architekturu celého řešení.

Nejprve bylo nutné rozdělit hlavní problém na menší podproblémy a identifikovat klíčové funkční celky aplikace. Tento krok umožnil lépe pochopit rozsah projektu a usnadnil řešení dílčích částí.

6.1 Rozdělení podproblémů

Jako hlavní funkční celek byla identifikována správa uživatelů. Ta zahrnuje registraci nových uživatelů, při které bude potřeba zachytit údaje jako e-mail a heslo. Dále pak samozřejmě přihlašování a odhlašování uživatelů.

Nedílnou součástí správy uživatelů je autentizace a autorizace. Zde bude klíčová implementace ověřování identity uživatelů pomocí e-mailu a hesla. Autorizace by neměla dovolit přístup do aplikace kromě registračního a přihlašovacího formuláře.

Jako další hlavní funkční celek byla stanovena samotná správa hesel. Ta bude zahrnovat vytváření a mazání přihlašovacích údajů, generování bezpečných hesel a ukládání hesel v zašifrované podobě v databázi.

Velmi důležitou částí aplikace bude sdílení hesel mezi uživateli. Zde půjde o vytváření skupin pro sdílení, správu členů skupin, určení vlastníka každé skupiny a následně samotné sdílení vybraných hesel mezi členy. Členské role budou moci pouze číst nebo sdílet vlastní přihlašovací údaje.

Nedílnou součástí je samozřejmě uživatelské rozhraní. To musí být responsivní pro různá zařízení, musí poskytovat přehledný výpis a tlačítka, které budou otevírat vyskakovací okna. Veškeré akce by měly být odděleny do modálních nebo vyskakovacích oken, celkově by UI mělo být intuitivní pro ovládání a snadnou navigaci.

Tímto rozdělením na menší podproblémy byl získán přehled o jejich vzájemných vazbách a závislostech. Bude možné systematicky postupovat implementací po částech, kdy některé budou stavět na jiných.

6.2 Výběr technologií

Aplikace bude typu API. Část aplikace, která vykresluje UI, získává a posílá data, provozována na straně klienta bude v projektu Blazor WASM a druhá část aplikace, jež zpracovává HTTP požadavky a provádí databázové operace bude projekt Web API.

Pro práci s daty a ukládáním hesel do databáze bude využíván objektově-relační mapovací nástroj Entity Framework Core. EF Core podporuje širokou škálu databázových enginů, jedním z nich je zvolený Microsoft SQL Server jako primární úložiště dat. [31][32]

JWT je také součástí plánu pro vývoj aplikace. Jedná se o standard, kterým lze provádět autentizaci a zabezpečit API endpointy.

Takže stručný popis technologií by mohl vypadat takto:

- **ASP.NET Core 8** - Jako primární webový framework pro vývoj celé aplikace.
- **Blazor WASM** - Pro vývoj uživatelského rozhraní pomocí C#.
- **Web API** - Pro vytvoření endpointů a provádění funkcí na straně serveru.
- **Entity Framework Core** - Jako ORM pro přístup k databázi a správu dat.
- **JWT** - Pro správu autentizace a autorizace uživatelů.
- **Microsoft SQL Server** - Jako databázový engine pro ukládání dat. [31][32][33]

7 ARCHITEKTURA APLIKACE

Design patterns a techniky používané při návrhu a vytváření aplikace jsou popsány architekturou aplikace. Plán a osvědčené postupy, které je třeba dodržovat při vytváření aplikace, poskytuje architektura, aby byla nakonec vytvořena dobře strukturovaná aplikace. [34]

Při vytváření aplikace mohou být nápomocny design patterns softwaru, které jsou popsány vzory a návrhy řešení problému. [34]

Služby front-endu i back-endu budou součástí architektury aplikace. Uživatelským prostředím aplikace se zabývá vývoj front-endu, zatímco přístup k datům, službám a dalším existujícím systémům, díky nimž aplikace funguje, je zaměřením vývoje back-endu. [34]

Pro vývoj webové aplikace pro správu hesel byl zvolen architektonický vzor Clean Architecture.

7.1 Clean architecture

Tento přístup klade důraz na oddělení součástí aplikace do logických vrstev podle jejich zodpovědností a minimalizaci vzájemných závislostí mezi vrstvami. U Clean Architecture je směřováním jednotlivých vrstev dovnitř zajišťováno oddělení zájmů. [35]

7.1.1 API vrstva (API)

- Obsahuje controllery a middleware pro zpracování a HTTP požadavků
- Tato vrstva má referenci na vrstvu Infrastructure

7.1.2 Aplikační vrstva (Application)

- Implementuje business logiku aplikace jako servisky (services)
- Obsahuje DTO a interface pro servisky i repositáře
- Má referenci pouze na vrstvu Domain

7.1.3 Doménová vrstva (Domain)

- Modeluje klíčové entity a koncepty aplikace (ApplicationUser, Vault apod.)
- Nemá žádné externí závislosti

7.1.4 Infrastrukturní vrstva (Infrastructure)

- Kód pro přístup k externím zdrojům (databáze, externí služby atd.)
- Implementace repozitářů pro práci s daty
- Má referenci pouze na vrstvu Application

7.1.5 Projekt Client

- Je to samostatný projekt
- Obsahuje UI komponenty, stránky, obrázky a kaskádové styly
- Nemá referenci na žádnou z předchozích vrstev

Clean Architecture je založena na striktním dodržování referenční integrity mezi vrstvami. Na vnějších vrstvách jsou vnitřní vrstvy, jako jsou doménová a aplikační vrstva, udržovány zcela nezávislé. [36]

Pouze prostřednictvím veřejných rozhraní, nikoli přímými referencemi, je umožněna komunikace mezi vrstvami. Services z aplikační vrstvy jsou volány prezentační vrstvou, které interně využívají repozitáře z infrastrukturní vrstvy k přístupu k datům. [37]

Vzor Dependency Injection je v aplikaci použit pro řízení životního cyklu objektů, vkládání závislostí a propojování různých komponent. ASP.NET Core obsahuje vestavěný kontejner IoC, který je konfigurován v programové části aplikace. Ale v rámci dodržení principů Clean Architecture jsou definovány třídy, které budou zaregistrovány v kontejneru, a také je specifikována jejich životnost (Singleton, Scoped, Transient). K tomu je ve vrstvách Infrastructure a Application, a jiných, když je potřeba, vytvořena třída DependencyInjection, která vrací IServiceCollection.

```
public static class DI
{
    public static IServiceCollection AddApplication(this IServiceCollection services)
    {
        services.AddScoped<IIdentityService, IdentityService>();
        services.AddScoped<IAesService, AesService>();
        services.AddScoped<IKeyService, KeyService>();

        return services;
    }
}
```

Obrázek 6: Registrace services v aplikační vrstvě

Tato třída obsahuje metodu AddApplication, která rozšiřuje IServiceCollection a registruje services v kontejneru. Metoda AddApplication umožňuje registraci servisek a jejich

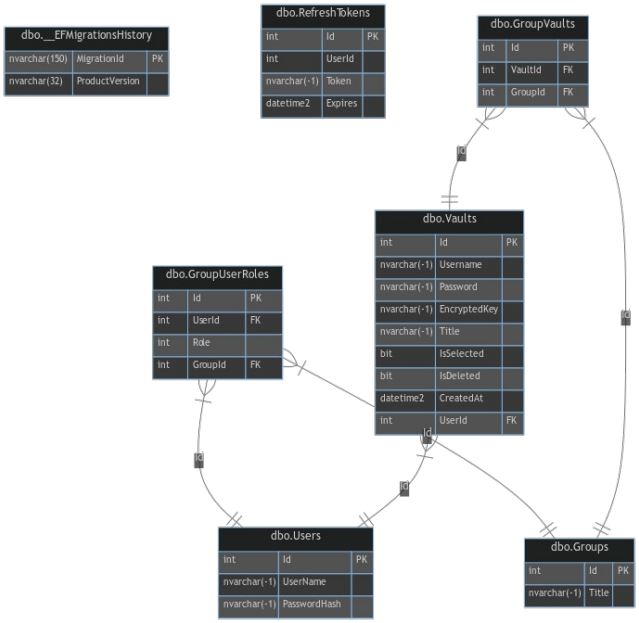
implementace do kontejneru. Tyto služby jsou poté dostupné pro vkládání závislostí v rámci celé aplikace, když se zaregistrují v programové části aplikace:

Obrázek 7: Registrace service collection z různých vrstev

Díky použití Dependency Injection mohou být v konstruktorech tříd deklarovány závislosti na jiných službách, které budou automaticky vloženy kontejnerem při vytváření instance. Toto usnadňuje testování, neboť v jednotkových testech mohou být závislé objekty nahrazeny falešnými implementacemi (Mock).

8 NÁVRH DATABÁZE

V C# lze udělat návrh databáze pomocí modelů, třídy s atributy představující tabulku a sloupce. Veškeré třídy a typy výčtu enum jsou v projektu Domain.



Obrázek 8: Návrh databáze

Schéma databáze obsahuje sedm tabulek, které jsou vzájemně propojeny a tvoří komplexní systém pro správu uživatelů, skupin a zabezpečení.
Vztahy mezi tabulkami jsou následující: Tabulky Users jsou propojeny s tabulkami GroupUserRoles a Vaults v poměru 1:N, tabulky Groups jsou propojeny s tabulkami GroupUserRoles a GroupVaults v poměru 1:N, a tabulky Vaults jsou propojeny s tabulkami GroupVaults v poměru 1:N.

8.1 Entity a datové třídy

Objekt reálného světa, který je relevantní pro danou aplikační doménu a jehož data jsou žádoucí ukládat do databáze trvale, je představen jako entita. V objektově orientovaném návrhu aplikace jsou datové třídy přesně reprezentovány jako entity v kódu aplikace.

1. User

V aplikaci je entita User reprezentována pro uživatele. Tvoří ji atributy Id, což je primární klíč, jedinečný identifikátor uživatele, UserName je uživatelské jméno v podobě emailové adresy a PasswordHash, což je hashovaná verze hesla uživatele pro bezpečné uložení.

2. Vault

```
public class User
{
    [Key] public int Id { get; set; }
    [Required] [EmailAddress] public string UserName { get; set; }
    [Required] public string PasswordHash { get; set; }
}
```

Obrázek 9: Třída User

Koncept trezoru slouží pro ukládání citlivých informací v trezorech. Má devět sloupců včetně Id jako primárního klíče, Username a Password pro přihlašovací údaje, EncryptedKey pro dodatečné zabezpečení, Title pro název trezoru, IsSelected a IsDeleted jako příznaky, CreatedAt pro datum a čas vytvoření a UserId jako cizí klíč odkazující na tabulku Users.

```
public class Vault
{
    public int Id { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
    public string EncryptedKey { get; set; }
    public string Title { get; set; }
    public bool IsSelected { get; set; }
    public bool IsDeleted { get; set; }
    public DateTime CreatedAt { get; set; }

    public int UserId { get; set; }
    public User User { get; set; }

    public List<GroupVault> GroupVaults { get; set; } = new List<GroupVault>();
}
```

Obrázek 10: Třída Vault

3. Group

Entita Group je reprezentována jako koncept skupiny uživatelů v rámci aplikace. Skupina je identifikována svým unikátním identifikačním číslem Id. Dále je obsažen atribut Title, která určuje název dané skupiny.

```
public class Group
{
    public int Id { get; set; }
    public string Title { get; set; }

    public List<GroupUserRole>? Members { get; set; } = new List<GroupUserRole>();
    public List<GroupVault>? GroupVaults { get; set; } = new List<GroupVault>();
}
```

Obrázek 11: Třída Group

4. GroupUserRole

Třída GroupUserRole je spojovací tabulkou mezi skupinami a trezory. Obsahuje tři sloupce: Id jako primární klíč záznamu, VaultId jako cizí klíč odkazující na tabulku Vaults a GroupId jako cizí klíč odkazující na tabulku Groups.

```
public class GroupUserRole
{
    public int Id { get; set; }

    public int UserId { get; set; }
    public User? User { get; set; }
    public GroupRoles Role { get; set; }

    public int GroupId { get; set; }
    public Group Group { get; set; }
}

public enum GroupRoles
{
    Owner,
    Member
}
```

Obrázek 12: Třída GroupUserRole a enum Roles

5. GroupVaults

Třída je používána jako spojovací tabulka mezi skupinami a trezory. Obsahuje tři sloupce: Id jako primární klíč záznamu, VaultId jako cizí klíč odkazující na tabulku Vaults a GroupId jako cizí klíč odkazující na tabulku Groups.

```
public class GroupVault
{
    public int Id { get; set; }

    public int VaultId { get; set; }
    public Vault Vault { get; set; }

    public int GroupId { get; set; }
    public Group Group { get; set; }
}
```

Obrázek 13: Třída GroupVault

6. RefreshTokens

```
public class RefreshToken
{
    [Key] public int Id { get; set; }

    public int UserId { get; set; }

    public string Token { get; set; }

    public DateTime Expires { get; set; }
}
```

Obrázek 14: Třída RefreshToken

Třída je používána pro správu obnovovacích tokenů. Skládá se ze čtyř sloupců: Id jako primární klíč tokenu, UserId pro identifikaci uživatele spojeného s tokenem, Token pro samotný obnovovací token a Expires pro datum a čas vypršení platnosti tokenu.

8.2 Vytvoření Databáze

Vytváření a správa databáze v Entity Frameworku je značně zjednodušena. Databáze není nutné vytvářet ručně, o to se postará samotný EF Core. V projektu Domain jsou nadefinovány modely entit, reprezentující tabulky v databázi. Tyto modely jsou seskupeny v databázovém kontextu. Pro vytvoření databázových tabulek a dalších objektů podle definovaných modelů se používají migrace databáze, vytvořené nástrojem dotnet ef. Existují jistá pravidla, jak se tento příkaz spouští, ale nejjednodušší je se v terminálu přepnout do projektu Infrastructure a spustit dotnet ef odsud. Tím pádem konkrétní příkaz pro vygenerování migračního skriptu je:

```
dotnet ef migrations add "název migrace"
```

Po vygenerování migračního skriptu je třeba migraci aplikovat na cílovou databázi příkazem:

```
dotnet ef database update .
```

9 NÁVRH UI

Aplikací bude nabízena široká škála funkcí, proto je nezbytné, aby bylo uživatelské rozhraní interaktivní, umožňující plynulou práci s různými prvky.

9.1 Interaktivní design

Toho bude dosaženo využitím vyskakovacích oken (modal) a vysouvacích panelů, které se budou otevírat po kliknutí na příslušná tlačítka reprezentována popisky nebo ikonami, které budou jasně popsány nebo vizuálně odlišeny podle své funkce, což zajistí intuitivní ovládání celé aplikace. Tímhle způsobem bude dosažena celková jednoduchost a přehlednost, což vede k dobré uživatelské zkušenosti (UX) a zároveň bude klást důraz na design, UI. Jednotlivé prvky budou rozmístěny a navrhovány tak, aby bylo rozhraní vzdušné, ale zároveň poskytovalo uživateli snadný přístup ke všem potřebným funkcím a informacím.

9.2 Stánky a komponenty

Aplikace bude rozdělena do čtyř hlavních stránek a dvou přihlašovacích, přičemž na každé z nich budou specifické komponenty umožňující provádění různých akcí a interakcí s aplikací. Tyto komponenty budou navrženy s ohledem na uživatelskou přívětivost a srozumitelnost.

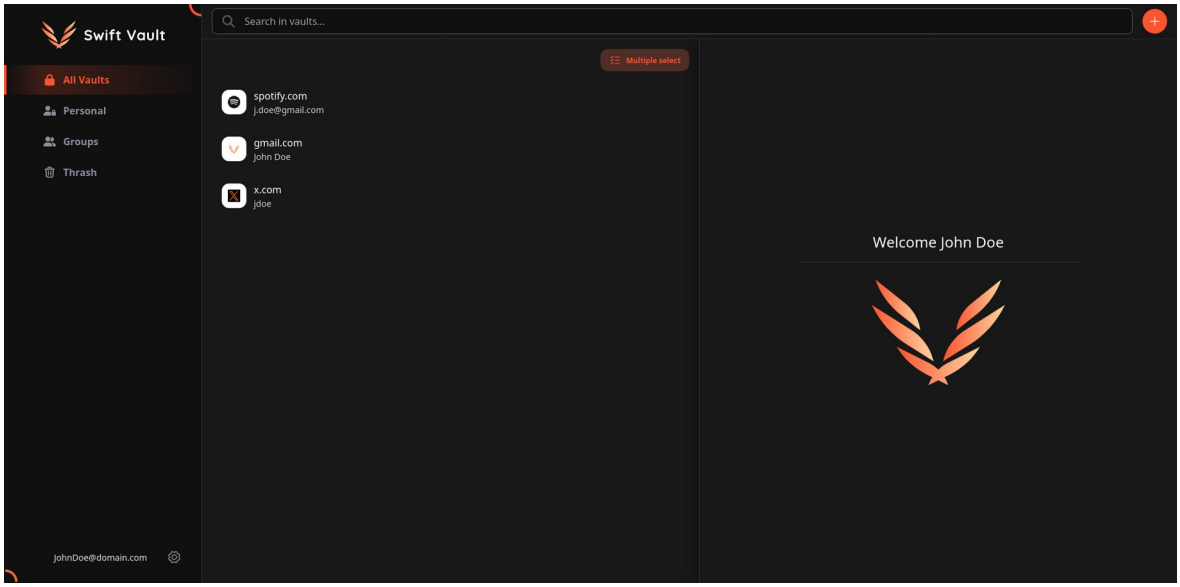
9.3 Stylování

Stylování aplikace nebude realizováno prostřednictvím klasických CSS stylů, ale bude využit moderní CSS framework Tailwind CSS. Ten usnadňuje vývoj díky svému utility-first přístupu, kdy jsou styly přiřazovány přímo k jednotlivým HTML elementům pomocí předpřipravených utilit namísto vytváření složitých kaskádových stylů.

9.4 Barevný motiv

Za pozornost taky stojí výběr vhodné barevné palety, která bude ladit s celkovým designem aplikace a zároveň bude příjemná pro oči uživatelů. Barvy budou účelně využity pro odlišení různých prvků rozhraní, jako jsou tlačítka nebo textová pole, čímž se opět zvýší uživatelská přívětivost a zážitek. Moderní webové aplikace často umožňují přizpůsobení motivu, světlého nebo tmavého, podle systémových nastavení uživatele. V případě této aplikace však bude použito pouze jedno barevné schéma - tmavý motiv.

Důvodem je lepší čitelnost a nižší úroveň oslnění při delším používání obrazovky počítačů nebo mobilních zařízení. Tmavé motivy obecně méně zatěžují zrak uživatele a snižují únavy očí. Pro případ, že by uživatel používal aplikaci za přímého slunečního svitu, kdy by standardní jas nemusel stačit pro dostatečnou čitelnost obsahu, tak bude možné zvýšit jas displeje k plné čitelnosti.



Obrázek 15: Design úvodní stránky

10 CLIENT-SIDE

V této fázi vývoje je kladen důraz především na tvorbu uživatelského rozhraní (UI) a všech vizuálních prvků aplikace, jako jsou komponenty, formuláře, tlačítka a navigační menu. Kromě vizuálního návrhu je nezbytné, aby byla implementována i interaktivita aplikace, tedy například reakce na uživatelské vstupy, validace formulářů a dynamické načítání obsahu.

10.1 Komunikace se serverem

Blazor WASM využívá k této komunikaci HTTP požadavky prostřednictvím upraveného `HttpClient`.

Klientská část odesílá HTTP požadavky na server prostřednictvím třídy, která je v Blazor aplikacích k dispozici jako standardní služba, běžně injektována do komponent pomocí DI a použije se direkrivou `@inject`. Je nakonfigurován tak, aby komunikoval se serverem, a umožňuje odesílat různé typy požadavků, jako jsou GET, POST, PUT, PATCH a DELETE. Tyto požadavky jsou nezbytné pro interakci s RESTful API na straně serveru, kde každý požadavek obvykle odpovídá konkrétní akci.

`HttpClient` podporuje asynchronní zpracování požadavků pomocí klíčových slov `async` a `await` se hlavní vlákno aplikace neblokuje během čekání na odpověď ze serveru.

10.1.1 BaseAddress

Aby bylo zajištěno, že všechny HTTP požadavky jsou směrovány na správný server, je důležité správně nastavit základní adresu `BaseAddress`. Tato adresa odpovídá URL, na které běží ASP.NET Core Web API, a je automaticky připojována k relativním cestám v požadavcích. Tím je zajištěno, že aplikace vždy komunikuje s předem určeným serverem, což je kritické zejména v produkčním prostředí, kde může být API hostováno na specifické doméně nebo subdoméně. Adresa se nastavuje v souboru Program.cs.

```
builder.Services.AddScoped(sp =>
{
    var handler = sp.GetRequiredService<AuthenticatedHttpClientHandler>();
    handler.InnerHandler = new HttpClientHandler();
    var httpClient = new HttpClient(handler)
    {
        BaseAddress = new Uri("http://localhost:5158")
    };
    return httpClient;
});
```

Obrázek 16: BaseAddress a upravený HttpClient

10.1.2 Autorizace

V aplikacích, které vyžadují autentizaci uživatelů, je nezbytné přidávat odpovídající hlavičky do HTTP požadavků, které odesílají citlivá data nebo přistupují k chráněným prostředkům. Typickým příkladem je hlavička `Authorization`, která obsahuje například JWT token, jenž serveru umožňuje ověřit identitu uživatele. `HttpClient` poskytuje jednoduchý mechanismus, jak tyto hlavičky přidávat do každého požadavku, čímž se zabezpečuje, že pouze autentizovaní uživatelé mají přístup k určitému obsahu nebo funkcionalitě API.

Pro získávání dat ze serveru se v Blazor WASM používá metoda `HttpClient.GetAsync`. Tato metoda odesílá HTTP GET požadavek na specifikovaný endpoint API a vrací odpověď, kterou lze následně deserializovat do C# objektů, obvykle pomocí knihovny jako je `System.Text.Json`. Tento proces je klíčový pro načítání dynamického obsahu do aplikace, například seznamu položek z databáze nebo detailů uživatele.

```
@code {
    List<VaultViewModel> _vaults = new ();

    protected override async Task OnInitializedAsync()
    {
        try
        {
            var response = await HttpClient.GetAsync("Vaults");
            response.EnsureSuccessStatusCode();

            var jsonResponse = await response.Content.ReadAsStringAsync();
            _vaults = JsonSerializer.Deserialize<List<VaultViewModel>>(jsonResponse, new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error fetching vaults: {ex.Message}");
            _vaults = new List<VaultViewModel>();
        }
    }
}
```

Obrázek 17: HTTP GET požadavek

Metody `PostAsync`, `PutAsync` a `DeleteAsync` umožňují Blazor WASM aplikaci provádět operace na serveru, které mění stav aplikace nebo data. Každá z těchto metod odesílá požadavek spolu s daty, která jsou serializována do formátu JSON, což je univerzální formát pro výměnu dat mezi klientem a serverem.

```
private async Task LoginUser()
{
    _err = string.Empty;

    var response = await HttpClient.PostAsJsonAsync("Identity/login", ViewModel);

    if (response.IsSuccessStatusCode)
    {
        var token = await response.Content.ReadAsStringAsync();
        await JsRuntime.InvokeVoidAsync("localStorage.setItem", "jwtToken", token);

        ((JwtAuthenticationStateProvider)AuthStateProvider).MarkUserAsAuthenticated(token);

        var authState = await AuthStateProvider.GetAuthenticationStateAsync();
        var username = authState.User.FindFirst(ClaimTypes.Name)?.Value;
        userState.Username = username ?? "Guest";

        NavigationManager.NavigateTo("/");
    }
    else if (response.StatusCode == HttpStatusCode.Unauthorized)
    {
        _err = await response.Content.ReadAsStringAsync();
    }
    else
    {
        // _err = "An unexpected error occurred.";
        _errr = response.Content.ReadAsStringAsync();
    }
}
```

Obrázek 18: HTTP POST požadavek

10.2 Pages

Razor syntaxe je využívána Blazorem pro tvorbu stránek, layoutů a komponent. Klasické HTML a CSS umožňují vytvořit strukturu a design uživatelského rozhraní, ale Razor syntaxe obohacuje UI o reaktivní stav pomocí direktiv a vkládání kódu v jazyce C#. Reaktivní chování je dosahováno v direktivě `@code`, která umožňuje psát C# kód přímo v Razor souboru. Tímto způsobem jsou definovány atributy, metody a zachycovány události, které ovlivňují chování a stav komponenty. Pro sdílení logiky mezi komponentami je poskytována Blazorem vestavěná podpora pro Dependency Injection. To umožňuje vytvořit třídu reprezentující určitou funkcionalitu a využít ji v libovolné Blazor komponentě pomocí direktivy `@inject`. Každá stránka v Blazoru obsahuje direktivu `@page`, která specifikuje vzor URL pro danou stránku. Tento vzor určuje, pro jaké cesty bude stránka renderována.

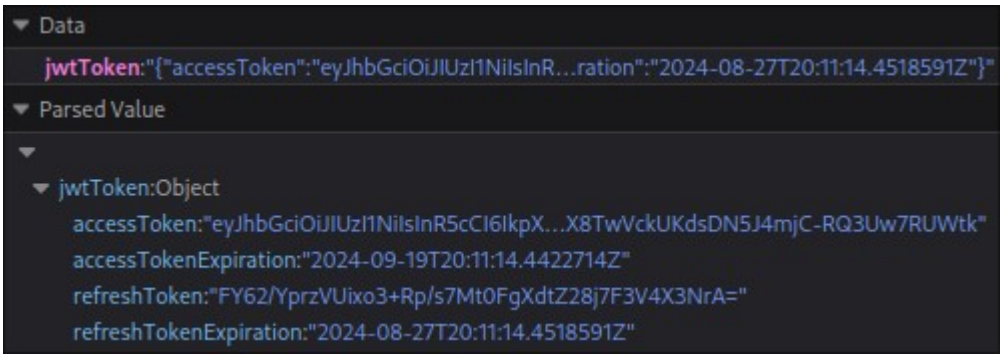
10.2.1 Login Page

Stránka `"/Identity/Login"` je navržena tak, aby byla responsivní a vizuálně atraktivní.

Formulář pro přihlášení je implementován pomocí komponenty `<EditForm>`, která je napojena na model `LoginViewModel`. Tento model zahrnuje dvě hlavní vlastnosti: `UserName` a `Password`.

Stránka zobrazuje chybové zprávy, pokud se přihlášení nezdaří, a poskytuje odkaz pro registraci nového uživatele, pokud ještě účet nemá.

Po kliknutí na tlačítko formuláře se provádí POST požadavek na endpoint `"Identity/login"`, kde jsou odeslány přihlašovací údaje uživatele. Pokud server vrátí úspěšnou odpověď, JWT je uložen do localStorage a uživatel je označen jako autentizovaný. Po úspěšném přihlášení je uživatel přesměrován na úvodní stránku. Pokud dojde k chybě, zobrazí se příslušná chybová zpráva.



Obrázek 19: JWT v Local Storage

10.2.2 Registration Page

Stránka `"/Identity/Registration"` obsahuje formulář pro registraci, který je rovněž implementován pomocí `<EditForm>`, který je napojen na model `RegisterViewModel`.

Registrační údaje se odesílají na endpoint `"Identity/register"` prostřednictvím POST požadavku. Po úspěšném zpracování serverem je uživatel přesměrován na přihlašovací stránku. Pokud registrace selže, zobrazí se chybová zpráva.

Hesla jsou zpracovávána a odesílána jako hash, což zajišťuje bezpečnost citlivých údajů.

10.2.3 Index Page

Hlavní stránka uživatelského rozhraní, která slouží k zobrazení a správě uložených údajů uživatele. Tato stránka je navržena tak, aby uživatelům poskytla intuitivní a interaktivní prostředí pro práci s jejich daty, přičemž umožňuje jak prohlížení jednotlivých položek, tak provádění hromadných operací.

Stránka je definována jako Razor komponenta s cestou `"/"`, což znamená, že se jedná o úvodní obrazovku aplikace. Klíčové součásti této stránky zahrnují načítání dat z externího zdroje, zobrazení seznamu položek a poskytnutí interaktivních prvků pro správu těchto položek.

Při načtení stránky je spuštěna asynchronní metoda `OnInitializedAsync`, která zajišťuje načtení dat ze serveru pomocí služby `HttpClient`. Konkrétně jsou data stahována z endpointu `"Vaults"`, následně deserializována z formátu JSON a uložena do seznamu typu `List<VaultViewModel>`. Tento seznam poté slouží jako datový zdroj pro zobrazení jednotlivých položek na stránce. Po výběru konkrétní položky se na pravé straně obrazovky zobrazí její detailní informace prostřednictvím komponenty `VaultDetails`. Tato komponenta nabízí možnost prohlížet podrobné údaje o vybrané položce a provádět akce, jako je její smazání.

V případě chyby při načítání dat z API se zachytí výjimka a uživatel je o této chybě informován. Toto zajišťuje robustnost aplikace a minimalizuje možné problémy při interakci se serverem.

10.2.4 Personal Page

Tato stránka je obdobná té hlavní, s několika malými rozdíly. Je definovaná pod cestou `"/personal"`, je určena pro zobrazení pouze osobních údajů uživatele.

Stránka při načtení volá metodu `OnInitializedAsync`, aby získala data z API. Data jsou získávána prostřednictvím `HttpClient` z endpointu `"Vaults/personal"`, kde jsou poté deserializována z formátu JSON a uložena do seznamu `_vaults` typu `List<VaultViewModel>`. Tento seznam slouží jako datový zdroj pro zobrazení uložených přihlašovacích údajů uživatele.

10.2.5 Groups Pages

Stránka definovaná pod cestou `"/groups"`, je určena pro zobrazení uživatelských skupin. Umožňuje uživateli zobrazovat a vybírat různé skupiny, ke kterým patří, a detailně prozkoumávat informace o vybrané skupině prostřednictvím samostatného panelu s detaily.

Stránka umožňuje dynamické zobrazování dat o uživatelských skupinách, která jsou načítána z API po inicializaci stránky. Data jsou ukládána do seznamu `_groups` typu `List<GroupViewModel>`, který je následně použit k zobrazení jednotlivých skupin uživatele.

Hlavní obsah stránky je tvořen seznamem skupin, které jsou reprezentovány jako tlačítka. Pokud uživatel není členem žádné skupiny, zobrazí se místo seznamu zpráva „*You are not in a group*“ ve středu obrazovky.

Při kliknutí na tlačítko určité skupiny se spustí metoda `OpenSelectedGroup`, která:

1. Nastaví vybranou skupinu jako aktivní (`_selectedGroup`).
2. Přepne viditelnost panelu s detaily skupiny (`_open`).

Tento panel se zobrazí jako překryvné okno, které se objeví na pravé straně obrazovky. V případě menších obrazovek nebo v režimu zobrazení pro mobilní zařízení se panel zobrazí přes celou obrazovku. Panel obsahuje komponentu `GroupDetail`, která zobrazuje podrobné informace o vybrané skupině.

Data jsou načítána asynchronně v metodě `OnInitializedAsync`, která je volána při prvotní inicializaci stránky. Tato metoda využívá `HttpClient` k získání seznamu skupin z API endpointu `"Groups"`. Pokud se data úspěšně načtou, jsou deserializována a uložena do seznamu `_groups`. V případě chyby při načítání je uživatel informován prostřednictvím konzole a seznam skupin je prázdný, což zajistí, že aplikace bude i nadále fungovat správně bez pádů. Kliknutím na tlačítko skupiny se zobrazuje panel s detaily, který lze snadno zavřít kliknutím na tlačítko `Close` v panelu.

10.2.6 Trash Page

Stránka `"/trash"` poskytuje intuitivní rozhraní pro správu položek přesunutých do koše. Umožňuje uživatelům efektivně vybírat a spravovat smazané položky, ať už jde o obnovení nebo trvalé odstranění. Díky asynchronnímu načítání dat, interaktivním prvkům a modernímu designu stránka přispívá k celkově příjemnému a uživatelsky přívětivému prostředí aplikace. Položky v koši se načítají asynchronně při inicializaci stránky prostřednictvím API volání na endpoint `"Vaults/trash"`. Data jsou deserializována a uložena do seznamu `_vaults`.

10.3 Komponenty

Namísto komplexních stránek a layoutů je aplikace sestavována z menších, znovu použitelných komponent. Tento přístup je upřednostňován kvůli řadě výhod, které přináší:

- Opakovatelnost: Komponenty jsou využívány na více místech v rámci aplikace, což vede k efektivnějšímu kódování a udržování konzistentního vzhledu a chování.
- Oddělená kódová báze: Kód komponent je oddělen od kódu stránek a layoutů, což umožňuje snadnější údržbu a lepší přehlednost celého projektu.

- Zlepšená čitelnost: Menší komponenty jsou snadněji čitelné a pochopitelné, což usnadňuje vývoj a budoucí rozšiřování aplikace.

10.3.1 Komunikace mezi komponenty

Existují tři hlavní způsoby, jak mohou komponenty vzájemně komunikovat:

1. Parent to Child: Tento způsob je nejčastěji používán, když je žádoucí, aby byl dynamický obsah vykreslován v dceřiné komponentě. Na hlavní stránce se zobrazuje seznam přihlašovacích údajů, trezorů, pokud chceme upravit informace o některém záznamu, kliknutím se nastaví proměnná. Dceřiná komponenta je pak volána a jsou jí předány potřebné parametry. Podle obdržených parametrů jsou pak vykreslována data.

```
[Parameter] public Vault? Vault { get; set; }
```

Obrázek 20: Parametr komponenty

```
<EditVault Vault="SelectedVault" />
```

Obrázek 21: Volání komponenty s parametrem

2. Child to Parent: Tento přístup může být využit v situaci, kdy je akce prováděna v dceřiné komponentě, která ovlivňuje uživatelské rozhraní rodičovské komponenty. K tomu je potřebný EventCallback jako parametr dceřiné komponenty. V aplikaci je tento přístup využíván, když je vybrán nějaký trezor a komponenta EditVault je otevřena. V ní je tlačítko používáno pro přesunutí aktuálního trezoru do koše.

```
[Parameter] public EventCallback Action { get; set; }
```

Obrázek 22: EventCallback parametr

Když se tento event vyvolá, stránka na něj musí reagovat a provést změny v seznamu trezorů a aktualizovat UI.

```
async Task MoveToTrash()
{
    var result = await VaultService.MoveToTrashAsync(SelectedVault!.Id);
    await Action.InvokeAsync(result);
}
```

Obrázek 23: Vyvolání EventCallbacku

```
<VaultDetails Action="() => HandleItemDeleted(_selectedItem!.Id)" />

void HandleItemDeleted(Guid itemId)
{
    _selectedItem = null;
    _vaults.RemoveAll(item => item.Id == itemId);
    StateHasChanged();
}
```

Obrázek 24: Event handler

3. State Container / Service: Tento způsob je používán, když mezi sebou potřebují komunikovat sourozenecké komponenty (dva nebo více komponent na stejné úrovni).

   V komponentě Header jsou volány další tři různé komponenty, které fungují na stejném principu. Jsou otevírány a zavírány po kliknutí na tlačítko. Je považováno za zbytečné vytvářet tři proměnné a tři funkce, které by udržovaly a měnily jejich stav. V takovém případě je vhodné využít centralizovaného správce stavu.

```
public class AddComponentState<T>
{
    public bool IsComponentOpen { get; set; }

    public void SetComponentOpen(bool value)
    {
        IsComponentOpen = value;
        NotifyStateChanged();
    }

    public event Action? OnChange;

    private void NotifyStateChanged() => OnChange?.Invoke();
}
```

Obrázek 25: State Container

Jednotlivé komponenty mohou měnit své stavy aktualizací sdíleného stavu ve správci. Všechny komponenty, které musí reagovat na změny stavu, pak využívají události `OnChange`. Takový State Container je service, proto je nutné ho zaregistrovat v programové části:

```
builder.Services.AddScoped(typeof(AddComponentState<>));
```

Ve všech komponentách je pak používána služba jako DI a je ještě přidán řádek `@implements IDisposable`. Tím je umožněno využití metody `Dispose()`.

```
@inject AddComponentState<AddGroup> GroupComponentState
@inject AddComponentState<AddPassword> PasswordComponentState
@inject AddComponentState<AddLogin> LoginComponentState
@implements IDisposable
```

Obrázek 26: DI AddComponentState service

Vhodný způsob komunikace mezi komponentami je klíčově důležitý pro zajištění požadovaného chování uživatelského rozhraní a umožňuje efektivní sdílení dat a koordinaci akcí napříč celou aplikací.

10.3.2 Update UI

V Blazoru je využíván koncept reaktivního programování, kde se daná část UI automaticky aktualizuje pokaždé, když dojde ke změně stavu nebo dat, na kterých je závislá. Pro aktualizaci stavu komponenty a tím i příslušné části UI může být v Blazoru využita metoda StateHasChanged() pro ruční oznámení změny stavu. Tahle metoda je využita při změně stavu komponent využívající AddComponentState service. I to je důvod, proč je nutné zavolat Dispose.

Životní cyklus komponent Blazor poskytuje také řadu životních metod komponent, jako jsou OnInitialized, OnParametersSet a OnAfterRender, které můžete využít pro načítání dat, nastavování počátečního stavu a provádění operací po vykreslení komponenty. Tyto metody umožňují efektivně spravovat aktualizace UI v různých fázích životního cyklu komponenty.
Pro aktualizaci stavu komponenty je nutné v těle metod OnInitialized a Dispose z IDosposable zavolat událost z třídy AddComponentState. Kdykoli se změní stav, událost StateHasChanged je vyvolána a komponenta je buď vykreslena nebo zmizí z obrazovky. Je důležité nezapomenout odhlásit metodu StateHasChanged komponenty z události OnChange, mohlo by to způsobit únik paměti. [43]

10.3.3 Formuláře a data binding

EditForm je zabudovanou součástí v Blazoru. Není to pouze samotná formulářová komponenta, ale komplexní řešení pro propojení dat, ověřování vstupů a odesílání formuláře. V případě použití Blazoru v režimu WebAssembly je dosaženo plné interaktivity, protože aplikace běží přímo v prohlížeči.

Jedním z atributů je FormName, který slouží k jednoduchému pojmenování formuláře. Dalším důležitým atributem je Model. Aby EditForm věděl, na jaká data se má vázat, je třeba vytvořit třídu ViewModel pro konkrétní formulář. ViewModel obsahuje pouze ty

atributy, které se očekávají ve formuláři. Tím se liší od klasických modelových tříd entit, kde může být atributů více, protože mohou být například vyplňovány automaticky.

V code bloku Blazor komponenty je poté vytvořena a inicializována nová instance ViewModelu. V registračním formuláři v aplikaci je od uživatele vyžadováno zadání emailové adresy, hesla a jeho potvrzení. Tyto informace jsou nutné nadefinovat ve ViewModel třídě:

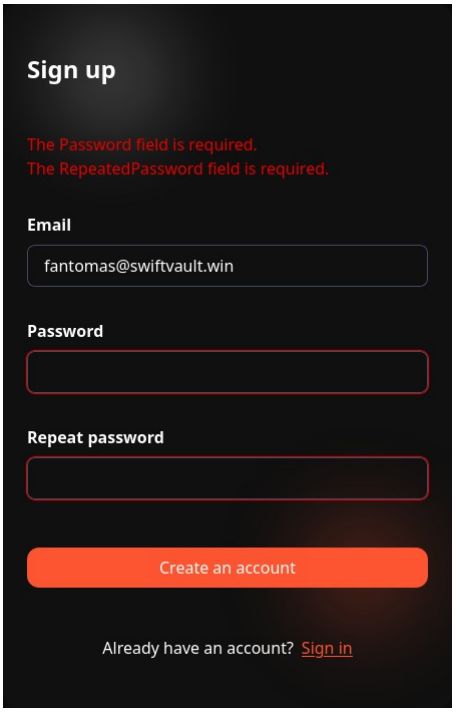
```
public class RegistrationViewModel
{
    [Required] [EmailAddress] public string Email { get; set; } = null!;

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
        MinimumLength = 6)]
    [DataType(DataType.Password)]
    public string Password { get; set; } = null!;

    [Required]
    [Compare(nameof(Password), ErrorMessage = "Passwords don't match!")]
    public string RepeatedPassword { get; set; } = null!;
}
```

Obrázek 27: Registrační View Model

Podle toho pak bude vypadat i registrační formulář. Pro validaci dat a informování uživatele o špatně vyplněném formuláři se do EditFormu volají komponenty `<DataAnnotationsValidator />` a `<ValidationSummary/>`.

```
<EditForm Model="ViewModel" method="post" OnValidSubmit="RegisterUser" FormName="register">
    <DataAnnotationsValidator/>
    <ValidationSummary/>
    ...
</EditForm>

@code {
    [SupplyParameterFromForm] private RegistrationViewModel ViewModel { get; set; } = new();
}
```

Obrázek 28: EditForm s atributy a inicializace ViewModelu



Obrázek 29: Registrační formulář

10.3.4 AddGroup komponenta

Tato komponenta slouží k přidání nové skupiny uživatelem. Uživatel zde může vyplnit název skupiny a přidat e-mailové adresy členů, kteří budou mít přístup ke skupině.

Obsahuje formulář, kde uživatel zadává název skupiny (GroupName). Umožňuje přidávání a odstraňování e-mailových adres členů skupiny. Formulář je odeslán prostřednictvím HTTP POST požadavku na server, kde jsou data zpracována.

10.3.5 AddLogin komponenta

Tato komponenta umožňuje uživatelům přidávat nové přihlašovací údaje.

Uživatel může zadat název, uživatelské jméno a heslo pro nové přihlášení. Je zde také možnost zobrazení další komponenty pro vygenerování silného hesla. Po validaci je formulář odeslán na server, kde jsou přihlašovací údaje zpracovány.

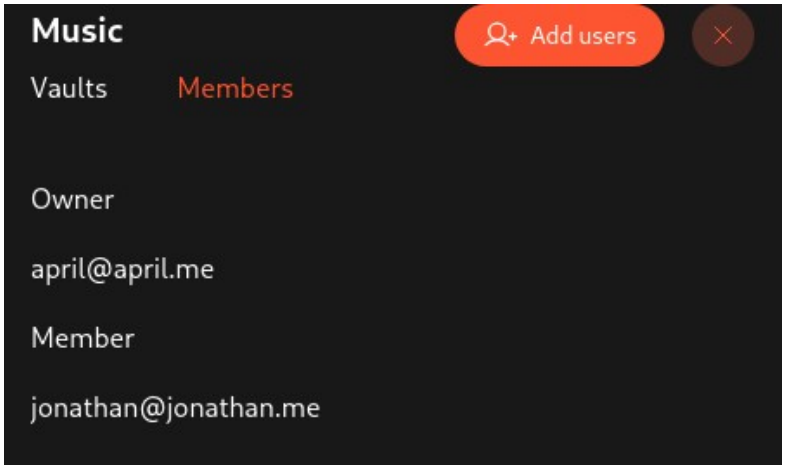
10.3.6 GroupDetails komponenta

```
@code {
    [SupplyParameterFromForm] private AddLoginViewModel ViewModel { get; set; } = new ();

    async Task CreateLogin()
    {
        var response = await HttpClient.PostAsJsonAsync("Vaults/new", ViewModel);
        if (response.IsSuccessStatusCode)
        {
            Dispose();
        }
    }

    protected override void OnInitialized()
    {
        LoginComponentState.OnChange += StateHasChanged;
        PasswordComponentState.OnChange += StateHasChanged;
    }

    public void Dispose()
    {
        LoginComponentState.OnChange -= StateHasChanged;
        PasswordComponentState.OnChange -= StateHasChanged;
    }

}
```

Obrázek 30: Code block v AddLogin komponentě

Detail komponenta zobrazuje podrobnosti o skupině, včetně seznamu členů a přihlašovacích údajů přidružených ke skupině. Uživatelé mohou přepínat mezi záložkami pro zobrazení přihlašovacích údajů ("Vaults") a členů skupiny ("Members"). Taktéž umožňuje přidání nových uživatelů do skupiny prostřednictvím modálního okna.

Zobrazuje název skupiny a seznam přihlašovacích údajů nebo členů skupiny v závislosti na aktivní záložce. V modálním okně je formulář pro přidání nových členů do skupiny, kde uživatelé mohou zadat e-mailové adresy nových členů. Po odeslání formuláře pro přidání uživatelů jsou data odeslána na server, kde jsou noví členové přidáni do skupiny, pokud zadaná emailová adresa existuje.



Obrázek 31: GroupDetail komponenta

10.3.7 ShareVault komponenta

Share komponenta samozřejmě umožňuje sdílení určitého přihlašovacího údaje (vault) mezi různými skupinami. Zobrazuje seznam skupin, do kterých lze přihlašovací údaje sdílet, a umožňuje uživateli vybrat, se kterými skupinami chce přihlašovací údaje sdílet.

Nejdřívě načítá seznam všech skupin z serveru a zobrazuje je jako zaškrtávací políčka. Uživatel může vybrat, které skupiny budou mít přístup k vybranému přihlašovacímu údaji. Po odeslání formuláře se vybrané skupiny aktualizují na serveru. Komponenta se zobrazuje v modálním okně, které je možné otevřít a zavřít kliknutím na tlačítka.

10.3.8 VaultDetails komponenta

Tato komponenta slouží k zobrazení detailů konkrétního přihlašovacího údaje (vault). Zobrazuje informace jako uživatelské jméno, heslo a další relevantní data. Umožňuje uživatelům kopírovat informace do schránky a spravovat přihlašovací údaje (např. přesunutí do koše).

10.4 Integrace s Tailwind CSS

Existuje několik možností, jak přidat Tailwind do projektu. Jednou z tradičních a zároveň lepších metod je použití NodeJS a npm. Spuštěním příkazu v terminálu `npx tailwindcss init` se vytvoří konfigurační soubor tailwind.config.js, který bude na začátku prázdný, a proto Tailwind třídy nebudou funkční. V této chvíli závisí na struktuře složek projektu. Ve šabloně Blazor WASM jsou layouty, stránky i komponenty ukládány do kořenového adresáře. Tuto cestu je nutno nastavit v konfiguračním souboru, v sekci content.

```
module.exports = {
    content: [
        './Components/*.razor',
        './Layout/*.razor',
        './Pages/**/*.razor',
        './wwwroot/index.html'
    ],
    plugins: [],
}
```

Obrázek 32: Tailwind config content

Dále je potřeba vytvořit soubor kaskádových stylů. Běžným způsobem je vytvořit složku Styles a soubor stylů v ní. Do tohoto souboru se pak vloží 3 řádky kódu:

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

```
@tailwind utilities;
```

Tyto direktivy umožňují vytvářet vlastní třídy a aplikovat na ně Tailwind třídy. Tímto způsobem lze zjednodušit vývoj, jelikož takové třídy jsou opakovatelné a budou mít stejný vzhled na každém elementu, kde se vlastní třída aplikuje.

Tailwind třídy musí být sestaveny do souboru umístěného ve složce wwwroot. Pro automatické sledování změn v Razor Pages je nutné spustit příkaz `npx tailwindcss -i ./Styles/app.css -o ./wwwroot/app.css –watch`, kde `-i` je vstupní soubor, tedy vlastní soubor se styly, a `-o` je výstupní soubor, tedy soubor, kam se Tailwind třídy sestaví. Tento soubor stylů je pak nutné připojit do hlavičky HTML v souboru `./wwwroot/index.html`.

10.5 Local Storage a Autorizace

Local Storage je webové API, které umožňuje ukládání dat na straně klienta. V Blazor WASM je možné Local Storage použít k ukládání uživatelských preferencí, session dat a dalších typů dat, která potřebují přetrvávat mezi relacemi prohlížeče.

Veškeré stránky jsou chráněny atributem `Authorize`, který povolí přístup pouze autentizovaným uživatelům. O tom, zda-li bude uživatel moci přistoupit na stránky, rozhoduje třída `JwtAuthenticationStateProvider`.
V rámci aplikace jsou implementovány tři klíčové třídy pro práci s JWT. Tyto třídy jsou zodpovědné za správu autentizace a autorizace uživatelů.

1. `JwtAuthenticationStateProvider`

Třída dědí od třídy `AuthenticationStateProvider` a je zodpovědná za správu stavu autentizace uživatele na základě JWT. Tato třída integruje JavaScript Interop (JSInterop) a `HttpClient` pro manipulaci s autentizačními tokeny. Obsahuje metody `GetAuthenticationStateAsync` a `MarkUserAsAuthenticated`. Tyto metody načítají JWT z Local Storage pomocí JSInterop. Pokud je token přítomen, parsuje jeho nároky (claims) a vytváří identitu uživatele, která je použita k vytvoření `ClaimsPrincipal`. Pokud token není k dispozici, vrátí anonymní `ClaimsPrincipal`. Následně vytváří `ClaimsPrincipal` z tokenu a aktualizuje stav autentizace. Uloží token do Local Storage, nastaví hlavičku autorizace pro `HttpClient` a aktualizuje stav uživatele v `userState`.

```
public class JwtAuthenticationStateProvider(IJSRuntime jsRuntime, HttpClient httpClient, UserState userState)
    : AuthenticationStateProvider
{
    public override async Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var token = await jsRuntime.InvokeAsync<string>("localStorage.getItem", "jwtToken");

        if (string.IsNullOrEmpty(token))
        {
            return new AuthenticationState(new ClaimsPrincipal(new ClaimsIdentity()));
        }

        var claims = JwtParser.ParseClaimsFromJwt(token);
        var identity = new ClaimsIdentity(claims, "jwt");
        var user = new ClaimsPrincipal(identity);

        return new AuthenticationState(user);
    }

    public void MarkUserAsAuthenticated(string token)
    {
        var authenticatedUser = new ClaimsPrincipal(new ClaimsIdentity(JwtParser.ParseClaimsFromJwt(token), "jwtToken"));
        var authState = Task.FromResult(new AuthenticationState(authenticatedUser));
        NotifyAuthenticationStateChanged(authState);

        // Set the Authorization header for HttpClient
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        // Optionally, store token in localStorage or secure storage
        jsRuntime.InvokeVoidAsync("localStorage.setItem", "jwtToken", token);

        // Optionally, store user state
        userState.Username = authenticatedUser.Identity?.Name; // Store username or relevant user state
    }

    public async Task LogoutAsync()
    {
        await jsRuntime.InvokeVoidAsync("localStorage.removeItem", "jwtToken");
        httpClient.DefaultRequestHeaders.Authorization = null;

        userState.Username = null; // Clear user state

        var anonymousUser = new ClaimsPrincipal(new ClaimsIdentity());
        var authState = Task.FromResult(new AuthenticationState(anonymousUser));
        NotifyAuthenticationStateChanged(authState);
    }
}
```

Obrázek 33: AuthenticationStateProvider třída

2. `AuthenticatedHttpClientHandler`

Třída dědí od `DelegatingHandler` a slouží k přidání JWT do hlaviček HTTP požadavků. Načítá token z Local Storage a přidává ho do hlavičky `Authorization` požadavku jako Bearer token. Poté předává upravený požadavek dál v řetězci zpracování.

3. `JwtParser`

Třída poskytuje statické metody pro analýzu a parsování JWT. Rozděluje JWT na části, dekóduje payload (tělo tokenu) z Base64 formátu, deserializuje JSON do slovníku klíč-hodnota a konvertuje tento slovník na kolekci `Claim` objektů.

Tyto komponenty spolupracují na zajištění správné autentizace a autorizace uživatelů v aplikaci, přičemž poskytují bezpečný a efektivní způsob správy uživatelských relací a přístupových práv.

Zabezpečení klientské strany aplikace nedovolí uživateli přistoupit na stránky, ale i tak by si mohl získat JSON z endpointů, proto je nutné autorizovat i metody endpointy na straně serveru.

11 SERVER-SIDE

V této fázi vývoje je hlavní pozornost věnována tvorbě serverové části aplikace, která zajišťuje logiku zpracování dat, bezpečnost a komunikaci mezi klientem a serverem. Server-side implementace zahrnuje vývoj API, které poskytuje data pro uživatelské rozhraní, a zpracovává požadavky přicházející z klientské strany. Klíčovou součástí je práce s databází. Dále je důležité zajistit bezpečnost aplikace prostřednictvím autentizace a autorizace uživatelů.

11.1 Propojení s MSSQL Databází a Entity Framework Core

Před samotným používáním databáze je nutné nastavit Connection String a související konfigurace databázového kontextu. DbContext je klíčovou součástí Entity Framework Core, která je zapouzdřována datovým kontextem aplikace a poskytuje rozhraní pro práci s databází.

V projektu Infrastructure je vhodné vytvořit složku s názvem Database, ve které budou dva soubory jako třídy.

1. ApplicationDbContext

Třída je odvozena z třídy `DbContext`, která je součástí frameworku Entity Framework Core. Hraje klíčovou roli v komunikaci mezi aplikací a databází. Je to hlavní třída, která umožňuje aplikaci interagovat s databázovými tabulkami pomocí objektově orientovaného přístupu. Tato třída definuje kolekce objektů typu `DbSet`, z nichž každý reprezentuje konkrétní tabulku v databázi. Pomocí těchto `DbSet` kolekcí může aplikace provádět základní operace v příslušných tabulkách. Vytvořená třída databázového kontextu vyžaduje parametr DbContextOptions, který je předáván konstruktoru třídy. V jazyce C# 12 bylo představeno zjednodušení syntaxe pro konstruktory, tzv. „primary constructors“. Díky tomu lze deklarovat konstruktor na úrovni definice třídy a parametry jsou dostupné kdekoliv v těle třídy.

```
public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : DbContext(options)
{
    public DbSet<User> Users { get; set; }
    public DbSet<RefreshToken> RefreshTokens { get; set; }
    public DbSet<Group> Groups { get; set; }
    public DbSet<Vault> Vaults { get; set; }
    public DbSet<GroupUserRole> GroupUserRoles { get; set; }
    public DbSet<GroupVault> GroupVaults { get; set; }
}
```

Obrázek 34: Databázový kontext

2. ApplicationDbContextFactory

Třída implementuje rozhraní `IDesignTimeDbContextFactory<ApplicationDbContext>`, které je součástí Entity Framework Core. Toto rozhraní poskytuje způsob, jak vytvořit instanci databázového kontextu během návrhu (design-time). Tento přístup je zvláště užitečný při práci s nástroji, jako jsou migrace databáze, které potřebují přístup k databázovému kontextu mimo hlavní běh aplikace.

Hlavní funkcí třídy je konfigurace a vytváření instancí `ApplicationDbContext` v době návrhu. To znamená, že tato třída je zodpovědná za správné nastavení připojení k databázi a za poskytování této konfigurace nástrojům, které ji potřebují během procesu vývoje, jako jsou migrace nebo generování schémat databáze. Tím, že umožňuje vytvářet kontext v době návrhu, podporuje flexibilitu a umožňuje snadné nasazení změn v databázi.

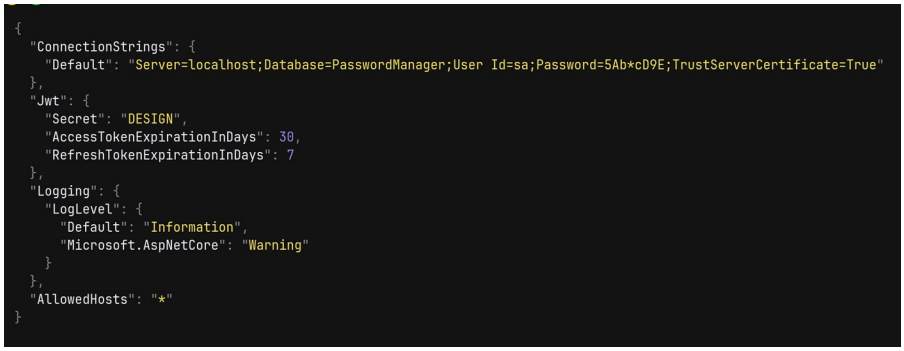
Metoda CreateDbContext je hlavní metodou třídy, která vytváří a vrací instanci `ApplicationDbContext`. Po načtení konfigurace metoda vytvoří objekt `DbContextOptionsBuilder`, který slouží k nastavení parametrů, jako je typ databáze a připojovací řetězec. V aplikaci je využit SQL Server, což se specifikuje pomocí metody `UseSqlServer`. Nakonec metoda vytvoří novou instanci třídy `ApplicationDbContext` s nakonfigurovanými možnostmi a vrátí ji.

Soubor appsettings.json obsahuje různé nastavení včetně připojovacích řetězců, parametrů autentizace a logování. Jelikož se soubor při vytvoření projektu nachází v projektu API, je nutné udělat kopii do projektu Infrastructure vedle souboru s DbContext třídou.

```
public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
{
        public ApplicationDbContext CreateDbContext(string[] args)
        {
            var basePath = Path.Combine(Directory.GetCurrentDirectory(), "Database/Context");
            Directory.SetCurrentDirectory(basePath);

            var configuration = new ConfigurationBuilder()
                .SetBasePath(basePath)
                .AddJsonFile("appsettings.json")
                .Build();

            var connectionString = configuration.GetConnectionString("Default") ??
                                   throw new NullReferenceException("No connection string");

            var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
            optionsBuilder.UseSqlServer(connectionString);

            return new ApplicationDbContext(optionsBuilder.Options);
        }
}
```

Obrázek 35: Factory databázový kontext

Nakonec je nutné databázový kontext zaregistrovat jako service, aby bylo možné jej použít způsobem Dependency Injection v dalších services a repositories aplikace.



Obrázek 36: Connection String pro připojení k databázovému serveru

11.2 Projekt Infrastructure

Backend je postaven na repositories a services, které jsou zaregistrovány jako metody, připravené k použití jako DI. Každá implementace bude mít svůj interface, který bude injectován v konstruktorech tříd. V tělě pak budou zpřístupněny všechny veřejné metody v interface.

Dalším důležitým bodem je, že repositories budou využívat DbContextFactory. To je kvůli životnímu cyklu DbContextu, který by ovlivňoval komponent v režimu interaktivního renderování. DbContext by stále existoval, tudíž by nenačítal nová data, kdybychom se třeba přepli na novou stránku. Ale právě tento problém řeší DbContextFactory. Začne nebo ukončí životní cyklus pokaždé, když se uživatel přepne na jinou stránku.

11.2.1 Repositories

Repositories jsou v podstatě služby, které se využívají pro manipulaci dat a komunikaci s databází. Mohou sem být zařazeny základní funkce modelu CRUD.

Namísto ručních SQL dotazů budou použity metody EF Core, které pro práci s daty využívají LINQ, technologii založenou na dotazování. Ruční psaní rozhraní není nutné. V Rideru je možné rozhraní vygenerovat kliknutím pravým tlačítkem na třídu a v sekci Refactor vybrat možnost Extract Interface. Rozhraní lze vygenerovat do souboru s implementací, avšak lepším řešením je uložit jej do dalšího souboru v jiném adresáři, a sice v projektu Application ve složce Interfaces.

1. VaultRepository

```
public class VaultRepository(ApplicationDbContext dbContextFactory, IKeyService keyService, IAesService aesService)
    : IVaultRepository
{
    protected readonly ApplicationDbContext Context = dbContextFactory;

    public async Task<List<VaultSummaryDto>> GetVaultSummariesAsync(int userId)
    {
        return await Context.Vaults
            .Where(c => c.UserId == userId && c.IsDeleted == false)
            .Select(v => new VaultSummaryDto { Id = v.Id, Title = v.Title })
            .ToListAsync();
    }

    public async Task<Vault?> GetByIdAsync(int id)
    {
        var vault = await Context.Vaults.FirstOrDefaultAsync(u => u.Id == id);
        if (vault is null) return new Vault();
        var encryptedKey = vault.EncryptedKey;
        var decryptedKey = await keyService.DecryptKey(encryptedKey);
        var decryptedPassword = await aesService.DecryptAsync(vault.Password, decryptedKey);
        vault.Password = decryptedPassword;
        return vault;
    }

    public async Task<List<Vault>> GetDeletedVaultsAsync(int userId)
    {
        return await Context.Vaults.Where(c => c.UserId == userId && c.IsDeleted).ToListAsync();
    }

    public async Task<List<Vault>> GetPersonalVaultsAsync(int userId)
    {
        return await Context.Vaults.Where(c => c.UserId == userId).ToListAsync();
    }

    public async Task<bool> CreateLoginAsync(AddVaultDto model, int userId)
    {
        var key = keyService.GenerateMasterKey();
        var encryptedKey = await keyService.EncryptKey(key);
        var encryptedPassword = await aesService.EncryptAsync(model.Password, key);

        var vault = new Vault
        {
            Title = model.Title,
            Username = model.UserName,
            Password = encryptedPassword,
            EncryptedKey = encryptedKey,
            UserId = userId,
            CreatedAt = DateTime.Now
        };

        Context.Vaults.Add(vault);
        return await Context.SaveChangesAsync() > 0;
    }

    public async Task<bool> MoveToTrashAsync(int vaultId)
    {
        var vault = await Context.Vaults.FindAsync(vaultId);
        if (vault == null) return false;

        vault.IsDeleted = true;
        Context.Entry(vault).State = EntityState.Modified;
        return await Context.SaveChangesAsync() > 0;
    }
}
```

Obrázek 37: VaultRepository třída

Třída `VaultRepository` se zaměřuje na správu a manipulaci s objekty typu `Vault`, které představují entity uložené v databázi, jež uchovávají citlivé informace, jako jsou hesla.

Implementuje rozhraní `IVaultRepository`, které definuje kontrakt pro práci s trezory (vaulty). Využívá metod Entity Framework Core pro komunikaci s databází prostřednictvím třídy `ApplicationDbContext`.

Metody třídy:

- `GetVaultSummariesAsync(int userId)`: Tato metoda získává seznam shrnutí trezorů (vaultů) pro konkrétního uživatele. Z databáze jsou načteny pouze nezbytné informace, jako je `Id` a `Title`, což umožňuje efektivnější přístup k datům.
- `GetByIdAsync(int id)`: Tato metoda načítá trezor (vault) na základě jeho `Id`. Pokud je trezor nalezen, jeho heslo je dešifrováno pomocí klíče, který je rovněž dešifrován. To je důležité pro zajištění bezpečného přístupu k citlivým informacím uloženým v trezoru.
- `GetDeletedVaultsAsync(int userId)`: Tato metoda vrací seznam trezorů, které byly označeny jako smazané pro konkrétního uživatele. Umožňuje to například správu nebo obnovu smazaných trezorů.
- `GetPersonalVaultsAsync(int userId)`: Metoda vrací všechny trezory, které jsou asociovány s konkrétním uživatelem, což umožňuje uživateli spravovat své osobní trezory.
- `CreateLoginAsync(AddVaultDto model, int userId)`: Tato metoda vytvoří nový trezor na základě dodaných dat, zašifruje heslo pomocí šifrovací služby `IAesService` a uloží jej do databáze. Tato operace zahrnuje generování šifrovacího klíče a jeho uložení v zašifrované podobě. Heslo pak v databázi bude nečitelné.

Username	Password
april@april.me	eE7PNaswfGMnO6Sk1lRWN43Ar9xBA0E/wG6Ecy6xv38=
april@april.me	1RWzj5Iy1QgDL9up1fMIzlraNisyWm66qnLDNrzZvB4=

Obrázek 38: Hesla v databázi

- `MoveToTrashAsync(int vaultId)`: Metoda označí trezor jako smazaný, aniž by jej skutečně odstranila z databáze. To umožňuje funkce jako "obnovení z koše".

2. GroupRepository

```
public class GroupRepository(ApplicationDbContext dbContextFactory,
    IUserRepository userRepository, IKeyService keyService, IAesService aesService) : IGroupRepository
{
    protected readonly ApplicationDbContext Context = dbContextFactory;

    public async Task<List<GroupDto>> GetGroupsAsync(int userId)
    {
        return await Context.Groups
            .Where(g => g.Members.Any(m => m.UserId == userId))
            .Select(g => new GroupDto
            {
                Id = g.Id,
                Title = g.Title
            })
            .ToListAsync();
    }

    public async Task CreateGroupAndAssignRolesAsync(string groupName, int userId, List<string> shareWith)
    {
        var group = new Group
        {
            Title = groupName
        };
        Context.Groups.Add(group);
        await Context.SaveChangesAsync();

        var ownerRole = new GroupUserRole
        {
            UserId = userId,
            Role = GroupRoles.Owner,
            Group = group
        };
        Context.GroupUserRoles.Add(ownerRole);

        if (shareWith.Count != 0)
        {
            foreach (var username in shareWith)
            {
                var member = await Context.Users.SingleOrDefaultAsync(u => u.UserName == username);
                var memberRole = new GroupUserRole
                {
                    UserId = member!.Id,
                    Role = GroupRoles.Member,
                    Group = group
                };
                Context.GroupUserRoles.Add(memberRole);
            }
        }

        await Context.SaveChangesAsync();
    }

    public async Task<bool> ShareToGroupAsync(int vaultToShare, List<int> groupsToShareIn)
    {
        foreach (var groupId in groupsToShareIn)
        {
            var group = await Context.Groups.FindAsync(groupId);
            if (group == null) continue;
            if (group.GroupVaults.All(gv => gv.VaultId != vaultToShare))
            {
                group.GroupVaults.Add(new GroupVault { VaultId = vaultToShare });
            }
        }

        return await Context.SaveChangesAsync() > 0;
    }

    public async Task<List<Vault>> GetVaultsAsync(int groupId)
    {
        var vaults = await Context.GroupVaults.Where(gv => gv.GroupId == groupId)
            .Include(gv => gv.Vault)
            .Select(gv => gv.Vault)
            .ToListAsync();

        if (vaults.Count == 0) return [];

        foreach (var vault in vaults)
        {
            var encryptedKey = vault.EncryptedKey;
            var decryptedKey = await keyService.DecryptKey(encryptedKey);
            var decryptedPassword = await aesService.DecryptAsync(vault.Password, decryptedKey);
            vault.Password = decryptedPassword;
        }

        return vaults;
    }

    public async Task<List<GroupUserRole>> GetMembersAsync(int groupId)
    {
        var members = await Context.GroupUserRoles.Where(gv => gv.GroupId == groupId)
            .Include(gur => gur.User)
            .ToListAsync();

        return members;
    }

    public async Task<string> AddMember(int groupId, AddUserToGroupDto model)
    {
        var memberToAdd = await userRepository.GetByUsernameAsync(model.Email);
        if (memberToAdd == null)
        {
            return "User not found.";
        }

        var existingMembership = await Context.GroupUserRoles
            .SingleOrDefaultAsync(gur => gur.UserId == memberToAdd.Id && gur.GroupId == groupId);

        if (existingMembership != null)
        {
            return "User is already a member.";
        }

        var newGroupUserRole = new GroupUserRole
        {
            UserId = memberToAdd.Id,
            GroupId = groupId,
            Role = GroupRoles.Member,
            User = memberToAdd
        };

        Context.GroupUserRoles.Add(newGroupUserRole);

        await Context.SaveChangesAsync();

        return "User has been added to the group.";
    }
```

Obrázek 39: GroupRepository třída

Metody třídy:

- `GetGroupsAsync(int userId)`: Tato metoda vrací seznam skupin, do kterých je konkrétní uživatel zařazen. Výsledkem je seznam objektů `GroupDto`, které obsahují základní informace o skupinách, jako je `Id` a `Title`.
- `CreateGroupAndAssignRolesAsync(string groupName, int userId, List<string> shareWith)`: Metoda vytvoří novou skupinu s názvem `groupName` a přiřadí k ní roli vlastníka (Owner) pro aktuálního uživatele. Pokud jsou uvedeny další uživatelská jména (ve formě `shareWith`), tato metoda přidá další uživatele do skupiny jako členy (Members).
- `ShareToGroupAsync(int vaultToShare, List<int> groupsToShareIn)`: Tato metoda sdílí konkrétní trezor (`vaultToShare`) se skupinami, jejichž ID jsou uvedena v seznamu `groupsToShareIn`. Zajišťuje, že trezor je přidán do skupin, do kterých ještě nebyl přiřazen.
- `GetVaultsAsync(int groupId)`: Metoda vrací seznam trezorů, které jsou přiřazeny k dané skupině. Trezory jsou dešifrovány před tím, než jsou vráceny, což zahrnuje dešifrování klíče a hesla.
- `GetMembersAsync(int groupId)`: Tato metoda vrací seznam členů konkrétní skupiny. Výsledkem je seznam `GroupUserRole`, který obsahuje informace o uživatelích a jejich rolích v dané skupině.
- `AddMember(int groupId, AddUserToGroupDto model)`: Metoda přidává nového člena do skupiny. Pokud uživatel neexistuje, nebo je již členem skupiny, vrátí odpovídající zprávu. Pokud uživatel existuje a není členem, přidá ho do skupiny a uloží změny.

3. UserRepository

Třída `UserRepository` je součástí vrstvy infrastruktury aplikace a implementuje rozhraní `IUserRepository`. Její hlavní úlohou je poskytovat operace pro práci s uživatelskými účty a refresh tokeny v systému.

```
public class UserRepository(ApplicationDbContext dbContextFactory) : IUserRepository
{
    protected readonly ApplicationDbContext Context = dbContextFactory;

    public async Task<RefreshToken> AddAsync(RefreshToken token)
    {
        await Context.RefreshTokens.AddAsync(token);
        await Context.SaveChangesAsync();
        return token;
    }

    public async Task<RefreshToken?> GetByTokenAsync(string token)
    {
        return await Context.RefreshTokens.FirstOrDefaultAsync(rt => rt.Token == token);
    }

    public async Task DeleteByUserIdAsync(int userId)
    {
        var tokensToDelete = await Context.RefreshTokens.Where(rt => rt.UserId == userId).ToListAsync();
        Context.RefreshTokens.RemoveRange(tokensToDelete);
        await Context.SaveChangesAsync();
    }

    public async Task DeleteAsync(RefreshToken refreshToken)
    {
        Context.RefreshTokens.Remove(refreshToken);
        await Context.SaveChangesAsync();
    }

    public async Task AddUser(User user)
    {
        Context.Users.Add(user);
        await Context.SaveChangesAsync();
    }

    public async Task<bool> UserExistsAsync(string username)
    {
        return await Context.Users.AnyAsync(u => u.UserName == username);
    }

    public async Task<User?> GetByIdAsync(int id)
    {
        return await Context.Users
            .FirstOrDefaultAsync(u => u.Id == id);
    }

    public async Task<User?> GetByUsernameAsync(string email)
    {
        return await Context.Users
            .FirstOrDefaultAsync(u => u.UserName == email);
    }
}
```

Obrázek 40: UserRepository třída

Metody třídy:

- `AddAsync(RefreshToken token)`: Tato metoda přidává nový refresh token do databáze. Po přidání tokenu do kontextu databáze metoda uloží změny a vrátí přidaný token.
- `GetByTokenAsync(string token)`: Metoda slouží k vyhledání refresh tokenu podle jeho hodnoty. Vrátí první nalezený token odpovídající zadané hodnotě, nebo `null`, pokud žádný odpovídající token neexistuje.
- `DeleteByUserIdAsync(int userId)`: Tato metoda maže všechny refresh tokeny přiřazené k danému uživatelskému ID. Nejprve načte všechny tokeny, které odpovídají zadanému ID, a poté je odstraní z databáze.

- `DeleteAsync(RefreshToken refreshToken)`: Metoda odstraní specifický refresh token z databáze. Po odstranění tokenu metoda uloží změny v databázi.
- `AddUser(User user)`: Tato metoda přidává nového uživatele do databáze. Po přidání uživatele do kontextu databáze metoda uloží změny a nový uživatel je tak trvale uložen.
- `UserExistsAsync(string username)`: Metoda ověřuje, zda uživatel s daným uživatelským jménem již existuje v databázi. Vrátí `true`, pokud uživatel existuje, jinak vrátí `false`.
- `GetByIdAsync(int id)`: Tato metoda vyhledá uživatele podle jeho ID. Vrátí uživatele odpovídající zadanému ID, nebo `null`, pokud uživatel s tímto ID neexistuje.
- `GetByUsernameAsync(string email)`: Metoda vyhledá uživatele podle jeho uživatelského jména (v tomto případě emailu). Vrátí uživatele odpovídající zadanému jménu, nebo `null`, pokud žádný odpovídající uživatel neexistuje.

Formuláře sloužící pro vytvoření záznamu vyžadují ViewModely nebo DTO. Rozdíl mezi nimi je, že DTO jsou data, přijatá z API, a ViewModel jsou data, které by reprezentovala zobrazení v UI. Tyhle modely jsou často stejné, proto se někdy jmenují Dto nebo ViewModel.

```
public class AddVaultDto
{
    public string Title { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }

    public string? Websites { get; set; }
}
```

Obrázek 41: DTO pro přidání trezoru

11.3 Projekt Application

Application vrstva slouží jako prostředník mezi business logikou a prezentační vrstvou (například API nebo frontend). Projekt poskytuje důležitou business logiku a služby pro aplikaci, včetně generování a správy klíčů, šifrování dat, a autentizace uživatelů prostřednictvím JSON Web Tokenů (JWT).

Tyto služby využívají repository pro přístup k datům a provádějí operace, které nejsou přímo zodpovědné za manipulaci s databází, ale spíše za zpracování a logiku aplikace.

Projekt zahrnuje také třídy DTO, které slouží jako struktury pro přenos dat mezi vrstvami aplikace. DTOs jsou často využívány k přenášení dat mezi API a frontendem nebo mezi různými částmi aplikace, a to v optimalizované podobě, která neobsahuje všechny detaily databázových entit.

11.3.1 Services

Služby jsou definovány prostřednictvím rozhraní, která určují, jaké metody a vlastnosti budou implementovány v konkrétních službách. Tento přístup zajišťuje, že implementace služeb mohou být snadno měněny nebo nahrazovány bez ovlivnění ostatních částí systému, které tyto služby využívají.

Služby definované v projektu Application jsou injektovány do kontrolerů v API projektu nebo jiných tříd, které potřebují přístup k business logice. To umožňuje kontrolerům delegovat složité operace na služby, které jsou odpovědné za zpracování těchto operací.

1. KeyService

```
public class KeyService(IAesService aesService) : IKeyService
{
    private readonly byte[] MasterKey = Convert.FromBase64String("U2FsdGVkX1/TxeFjRZ+3C8/KeJPinIGx4ZuEEAqOQ3Q=");

    public string GenerateMasterKey()
    {
        using (var aes = Aes.Create())
        {
            aes.KeySize = 128;
            aes.GenerateKey();
            return Convert.ToBase64String(aes.Key);
        }
    }

    public async Task<string> EncryptKey(string userKey)
    {
        return await aesService.EncryptAsync(userKey, Convert.ToBase64String(MasterKey));
    }

    public async Task<string>  DecryptKey(string encryptedUserKey)
    {
        return await aesService.DecryptAsync(encryptedUserKey, Convert.ToBase64String(MasterKey));
    }
}
```

Obrázek 42: KeyService třída

Třída `KeyService` implementuje rozhraní `IKeyService` a je zodpovědná za generování, šifrování a dešifrování klíčů. Má pevně definovaný klíč (ve formě base64), který slouží k šifrování a dešifrování dalších klíčů.

Metody třídy:

- `GenerateMasterKey()`: Vytváří nový master key (hlavní klíč) použitím algoritmu AES. Generovaný klíč je ve formátu base64, což umožňuje jeho jednoduché uložení a přenos.

- `EncryptKey(string userKey)`: Šifruje zadaný uživatelský klíč pomocí master key a `aesService`. Šifrování se provádí asynchronně, což zajišťuje efektivní výkon v aplikacích s vysokým počtem operací.
- `DecryptKey(string encryptedUserKey)`: Dešifruje zadaný šifrovaný uživatelský klíč použitím master key a `aesService`. Tato metoda vrací dešifrovaný klíč a také pracuje asynchronně.

2. AesService

```
public class AesService : IAesService
{
    private const int IvSizeInBytes = 16; // Fixed IV size for AES encryption

    public async Task<string> EncryptAsync(string plainText, string key)
    {
        var keyBytes = Convert.FromBase64String(key);
        using (var aes = Aes.Create())
        {
            aes.Key = keyBytes;
            aes.GenerateIV();

            MemoryStream msEncrypt;
            using (var encryptor = aes.CreateEncryptor(aes.Key, aes.IV))
            using (msEncrypt = new MemoryStream())
            using (var csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                await swEncrypt.WriteAsync(plainText);
            }

            var iv = aes.IV;
            var encryptedBytes = msEncrypt.ToArray();

            var result = new byte[IvSizeInBytes + encryptedBytes.Length];
            Buffer.BlockCopy(iv, 0, result, 0, IvSizeInBytes);
            Buffer.BlockCopy(encryptedBytes, 0, result, IvSizeInBytes, encryptedBytes.Length);

            return Convert.ToBase64String(result);
        }
    }

    public async Task<string> DecryptAsync(string cipherText, string key)
    {
        var keyBytes = Convert.FromBase64String(key);
        var fullCipher = Convert.FromBase64String(cipherText);

        using (var aes = Aes.Create())
        {
            if (aes.LegalKeySizes.Any(s => s.MaxSize >= keyBytes.Length * 8 && s.MinSize <= keyBytes.Length * 8))
            {
                aes.Key = keyBytes;

                var iv = new byte[IvSizeInBytes];
                var cipher = new byte[fullCipher.Length - IvSizeInBytes];

                Buffer.BlockCopy(fullCipher, 0, iv, 0, IvSizeInBytes);
                Buffer.BlockCopy(fullCipher, IvSizeInBytes, cipher, 0, cipher.Length);

                aes.IV = iv;

                using (var decryptor = aes.CreateDecryptor(aes.Key, aes.IV))
                using (var msDecrypt = new MemoryStream(cipher))
                using (var csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))
                using (var srDecrypt = new StreamReader(csDecrypt))
                {
                    return await srDecrypt.ReadToEndAsync();
                }
            }
            else
            {
                throw new ArgumentException(
                    $"The provided key is not a valid size for the AES algorithm. Supported key sizes are:
                    {string.Join(", ", aes.LegalKeySizes.Select(s => $"{s.MinSize / 8}-{s.MaxSize / 8} bits"))}.");
            }
        }
    }
}
```

Obrázek 43: AesService třída

Třída `AesService` implementuje rozhraní `IAesService` a poskytuje funkce pro šifrování a dešifrování dat pomocí algoritmu AES (Advanced Encryption Standard).

Metody třídy:

- `EncryptAsync(string plainText, string key)`: Šifruje zadaný text (`plainText`) použitím zadaného klíče (`key`).
- Vytváří nový inicializační vektor (IV) pro šifrování, který je uložen spolu se šifrovanými daty.
- Používá `CryptoStream` pro zapisování šifrovaných dat do paměťového proudu a pak vrací šifrovaný text ve formátu base64.
- `DecryptAsync(string cipherText, string key)`: Dešifruje zadaný šifrovaný text (`cipherText`) použitím zadaného klíče (`key`).
- Extrahuje IV z šifrovaných dat a používá ho pro dešifrování dat.
- Využívá `CryptoStream` pro čtení dešifrovaných dat z paměťového proudu a vrací dešifrovaný text.

3. IdentityService

```
public class IdentityService(IConfiguration configuration, IUserRepository userRepository)
    : IIdentityService
{
    public async Task<bool> RegisterAsync(IdentityDto newUser)
    {
        if (await userRepository.UserExistsAsync(newUser.UserName))
        {
            return false;
        }

        var user = new User
        {
            UserName = newUser.UserName,
            PasswordHash = HashPassword(newUser.Password, newUser.UserName)
        };

        await userRepository.AddUser(user);
        return true;
    }

    public async Task<JwtTokenDto?> LoginUserAsync(IdentityDto login)
    {
        var user = await userRepository.GetByUsernameAsync(login.UserName);

        if (user == null || !VerifyPassword(login.Password, login.UserName, user.PasswordHash))
        {
            return null;
        }

        return await GenerateJwtToken(user.Id, user.UserName);
    }

    public async Task<JwtTokenDto> GenerateJwtToken(int userId, string userName)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(configuration.GetSection("Jwt:Secret").Value!);
        var accessTokenExpiration = int.Parse(configuration.GetSection("Jwt:AccessTokenExpirationInDays").Value!);
        var refreshTokenExpiration = int.Parse(configuration.GetSection("Jwt:RefreshTokenExpirationInDays").Value!);

        // Access Token
        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new Claim[]
            {
                new(ClaimTypes.NameIdentifier, userId.ToString()),
                new(ClaimTypes.Name, userName)
            }),
            Expires = DateTime.UtcNow.AddDays(accessTokenExpiration),
            SigningCredentials =
                new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
        };
        var accessToken = tokenHandler.CreateToken(tokenDescriptor);
        var refreshToken = GenerateRefreshToken();
        var refreshExpirationDate = DateTime.UtcNow.AddDays(refreshTokenExpiration);

        // Store the refresh token in the database
        await userRepository.AddAsync(new RefreshToken
            { UserId = userId, Token = refreshToken, Expires = refreshExpirationDate });

        return new JwtTokenDto
        {
            AccessToken = tokenHandler.WriteToken(accessToken),
            AccessTokenExpiration = tokenDescriptor.Expires.Value,
            RefreshToken = refreshToken,
            RefreshTokenExpiration = refreshExpirationDate
        };
    }

    private string HashPassword(string password, string email)
    {
        string salt = $"{email}%salting*";
        return Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: Encoding.ASCII.GetBytes(salt),
            prf: KeyDerivationPrf.HMACSHA256,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
    }

    private bool VerifyPassword(string password, string email, string hashedPassword)
    {
        return HashPassword(password, email) == hashedPassword;
    }

    private string GenerateRefreshToken()
    {
        var randomNumber = new byte[32];
        using var rng = System.Security.Cryptography.RandomNumberGenerator.Create();
        rng.GetBytes(randomNumber);
        return Convert.ToBase64String(randomNumber);
    }

    public Task<ClaimsPrincipal?> ValidateTokenAsync(string token)
    {
        try
        {
            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes(configuration.GetSection("Jwt:Secret").Value!);
            var claimsPrincipal = tokenHandler.ValidateToken(token, new TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey(key),
                ValidateIssuer = false,
                ValidateAudience = false,
                ClockSkew = TimeSpan.Zero,
                ValidateLifetime = true
            }, out _);

            return Task.FromResult(claimsPrincipal)!;
        }
        catch
        {
            return Task.FromResult<ClaimsPrincipal>(null!)!;
        }
    }
}
```

Obrázek 44: IdentityService třída

Třída `IdentityService` implementuje rozhraní `IIdentityService` a poskytuje služby týkající se autentizace a správy uživatelů, včetně registrace uživatelů, generování a validace JWT.

Metody třídy:

- `RegisterAsync(IdentityDto newUser)`: Registruje nového uživatele, pokud uživatel s daným uživatelským jménem ještě neexistuje. Vytváří nového uživatele s hashovaným heslem (pomocí funkce `HashPassword`) a ukládá ho do databáze.
- `LoginUserAsync(IdentityDto login)`: Přihlašuje uživatele na základě zadaných přihlašovacích údajů. Ověřuje, zda uživatel existuje a zda heslo odpovídá uloženému hashi. Pokud jsou údaje správné, generuje JWT token a vrací ho.
- `GenerateJwtToken(int userId, string userName)`: Generuje JWT token pro autentizaci uživatele. Vytváří access token a refresh token. Access token obsahuje identifikátory a data o uživatelském jménu, a je podepsán symetrickým klíčem. Ukládá refresh token do databáze a vrací DTO obsahující oba tokeny.
- `ValidateTokenAsync(string token)`: Validuje zadaný JWT token. Používá klíč a parametry pro ověření platnosti tokenu, včetně životnosti a podpisu. Vrací `ClaimsPrincipal`, který obsahuje informace o uživateli a platnosti tokenu, nebo `null` v případě neúspěšné validace.

11.4 Projekt API

Nejdůležitějším projektem je samotný API. Je klíčovým komponentem moderní webové aplikace, který poskytuje rozhraní pro komunikaci mezi klientskou aplikací a backendovými službami. V tomto kontextu se API zaměřuje na poskytování metod pro interakci s datovými modely a logikou aplikace prostřednictvím HTTP požadavků.

Slouží jako most mezi uživatelským rozhraním a logikou aplikace, a umožňuje provádění operací, jako jsou:

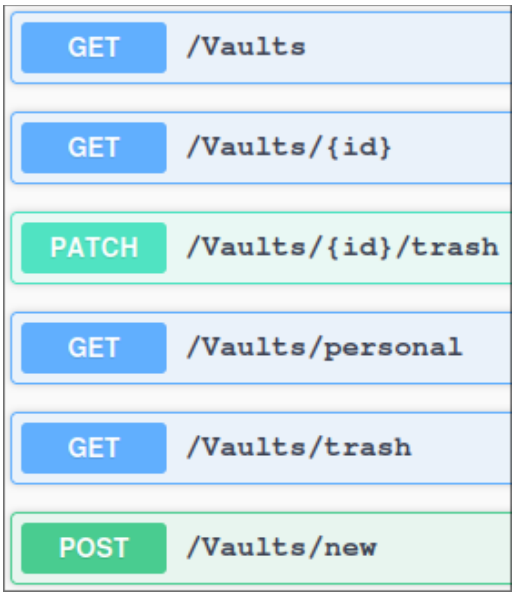
- Vytváření, čtení, aktualizace a mazání dat (CRUD): poskytuje endpointy, které umožňují klientům vytvářet nové záznamy, načítat existující data, aktualizovat údaje a mazat záznamy z databáze.
- Autentizace a autorizace: spravuje bezpečnostní aspekty aplikace, včetně ověřování uživatelů a zajišťování, že mají přístup pouze k oprávněným zdrojům.

Projekt API obsahuje následující klíčové komponenty:

- Kontrolery (Controllers)
- Middleware
- Konfigurace

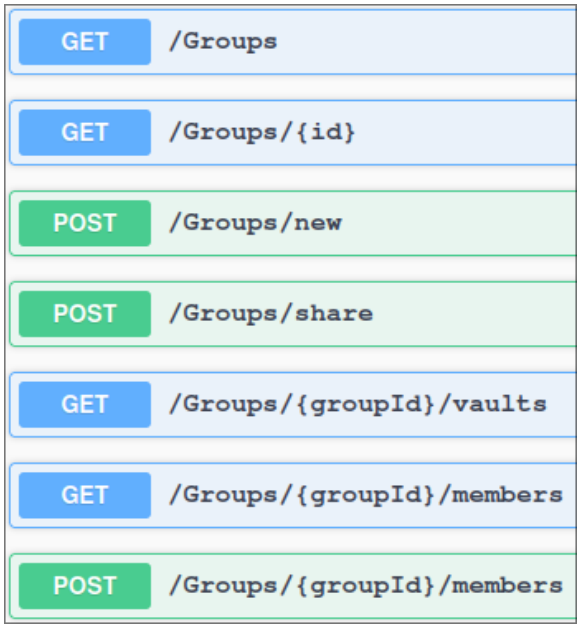
11.4.1 Dostupné endpointy pro uživatele

1. Endpointy pro vaulty:



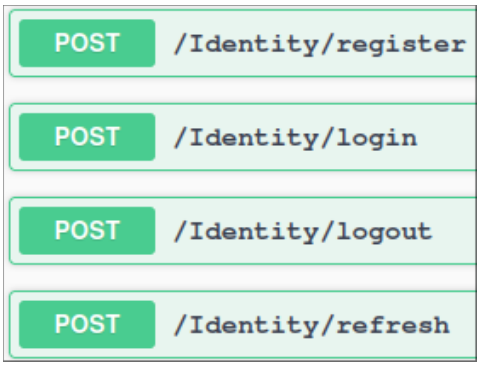
Obrázek 45: Vaults endpointy

2. Endpointy pro groups:



Obrázek 46: Groups endpointy

3. EndPointy pro autentizaci:



Obrázek 47: Identity endpointy

11.4.2 Kontrolery

Kontrolery jsou třídy, které přijímají HTTP požadavky, zpracovávají je a vrací odpovědi. Každý kontroler je zodpovědný za určitou část aplikace nebo za určité operace, například uživatelskou správu nebo správu dat.

V téměř každé controller třídě bude nutné zjistit Id uživatele, aby se neduplikoval kód, tak byla vytvořena třída `BaseController`, která implementuje takovou metodu. Pro použití pak stačí dědit z této třídy.

Pro kontrolery existují atributy, jež ovlivňují chování kontroleru a jeho metod. Používané atributy jsou [Route], který určuje cestu (URL) na kterou bude daný kontroler nebo metoda odpovídat, [ApiController], jež označuje třídu jako Web API kontroler. Automaticky

zajišťuje některé funkce, jako například automatickou validaci modelů vstupních dat nebo návrat 400 (Bad Request) při neplatných datech. Tento výstup je pak možné vidět v konzoli prohlížeče. A ještě taky [Authorize] pro omezený přístup, ten je ale v aplikaci nahrazen vlastní middleware, která ověřuje JWT v HTTP požadavcích.

1. BaseController

```
public class BaseController : ControllerBase
{
    [NonAction]
    protected int GetCurrentUserId()
    {
        var userIdClaim = User.FindFirstValue(ClaimTypes.NameIdentifier);
        if (userIdClaim == null || !int.TryParse(userIdClaim, out var userId))
        {
            throw new UnauthorizedAccessException("User ID not found in claims.");
        }

        return userId;
    }
}
```

Obrázek 48: BaseController třída

Třída `BaseController` je základní třída pro kontrolery v aplikaci, která dědí z `ControllerBase` a poskytuje základní funkcionalitu, kterou mohou využívat další kontrolery. V tomto případě obsahuje následující klíčovou metodu:

- Metoda `GetCurrentUserId()` je určena k získání ID aktuálně přihlášeného uživatele z jeho autentizačních nároků (claims). Metoda využívá objekt `User`, který je poskytován frameworkem ASP.NET Core a obsahuje informace o aktuálním uživatelském kontextu. Získává hodnotu nároku `ClaimTypes.NameIdentifier`, který by měl obsahovat ID uživatele. Pokud nárok není přítomen nebo není možné převést jeho hodnotu na celé číslo (`int`), metoda vyvolá výjimku `UnauthorizedAccessException`, což indikuje, že uživatelská identifikace není dostupná nebo je neplatná.

2. VaultsController

```
[Route("[controller]")]
[ApiController]
[AuthorizeMiddleware]
public class VaultsController(IVaultRepository vaultRepository) : BaseController
{
    [HttpGet]
    public async Task<ActionResult<List<VaultSummaryDto>>> GetVaultSummaries()
    {
        var userId = GetCurrentUserId();
        var vaults = await vaultRepository.GetVaultSummariesAsync(userId);
        return Ok(vaults);
    }

    [HttpGet("{id:int}")]
    public async Task<ActionResult<Vault>> GetVaultById(int id)
    {
        try
        {
            var vault = await vaultRepository.GetByIdAsync(id);
            return Ok(vault);
        }
        catch (KeyNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
    }

    [HttpPatch("{id:int}/trash")]
    public async Task<ActionResult<string>> MoveToTrashAsync(int id)
    {
        try
        {
            var success = await vaultRepository.MoveToTrashAsync(id);
            if (success)
            {
                return Ok(new { Message = "Vault moved to trash successfully." });
            }

            return NotFound(new { Message = "Vault not found." });
        }
        catch (Exception ex)
        {
            return StatusCode(500,
                new { Message = "An error occurred while moving the vault to trash.", Details = ex.Message });
        }
    }

    [HttpGet("personal")]
    public async Task<ActionResult<List<Vault>>> GetPersonalVaults()
    {
        var userId = GetCurrentUserId();
        var vaults = await vaultRepository.GetPersonalVaultsAsync(userId);
        return Ok(vaults);
    }

    [HttpGet("trash")]
    public async Task<ActionResult<List<Vault>>> GetDeletedVaults()
    {
        var userId = GetCurrentUserId();
        var vaults = await vaultRepository.GetDeletedVaultsAsync(userId);
        return Ok(vaults);
    }

    [HttpPost("new")]
    public async Task<IActionResult> CreateVault([FromBody] AddVaultDto addVault)
    {
        Console.WriteLine(addVault.UserName);
        var userId = GetCurrentUserId();
        var success = await vaultRepository.CreateLoginAsync(addVault, userId);
        return Ok(success);
    }
}
```

Obrázek 49: VaultsController třída

Tato třída je kontrolerem, který poskytuje API endpointy pro operace související se správou "Vaults" (trezorů). `VaultsController` je odvozen od základní třídy `BaseController`, která poskytuje společné funkcionality pro všechny kontrolery v aplikaci. Tato třída se řídí atributy, které určují její chování v rámci ASP.NET Core frameworku.

Obsahuje atributy jako:

- [Route("[controller]")], jež určuje základní cestu a `[controller]` je placeholder, který je nahrazen názvem kontroleru – v tomto případě `Vaults`. Výsledkem je, že všechny metody tohoto kontroleru budou mít cestu začínající `/Vaults`.
- [ApiController]
- [AuthorizeMiddleware]

Kontroler využívá DI pro získání instance `IVaultRepository`, což je rozhraní, které definuje operace nad úložištěm trezorů. Toto rozhraní umožňuje kontroleru přistupovat k datům trezorů a manipulovat s nimi bez nutnosti přímé interakce s databází.

Metody třídy:

- `GetVaultSummaries()`: Tato metoda získává seznam shrnutí (souhrnných informací) všech trezorů, které patří aktuálně přihlášenému uživateli. Shrnutí obsahuje ID a název trezoru. Cesta k metodě je endpoint `/Vaults`.
- `GetVaultById(int id)`: Získává detailní informace o konkrétním trezoru podle jeho ID. Pokud trezor neexistuje, vrací chybu 404 (Not Found). Endpoint je: `/Vaults/{id}`.
- `MoveToTrashAsync(int id)`: Přesouvá trezor do koše, což ho označí jako "smazaný" (ale fyzicky se z databáze neodstraní). Pokud se operace nezdaří, vrací chybu 500 (Internal Server Error) nebo 404, pokud trezor neexistuje. Používá HTTP metodu PATCH, která změní pouze konkrétní sloupec oproti PUT, který by přepsal celý záznam. Zpřístupňuje endpoint `/Vaults/{id}/trash`.
- `GetPersonalVaults()`: Získává seznam všech trezorů aktuálně přihlášeného uživatele, které nejsou označeny jako smazané na endpointu `/Vaults/personal`.
- `GetDeletedVaults()`: Získává seznam všech trezorů aktuálně přihlášeného uživatele, které jsou označeny jako smazané na endpointu `/Vaults/trash`.
- `CreateVault([FromBody] AddVaultDto addVault)`: Vytváří nový trezor na základě údajů, které jsou odeslány v těle požadavku (JSON objekt typu `AddVaultDto`). Nový trezor je přidán do databáze. Používá HTTP metodu POST s endpointem `/Vaults/new`.

3. GroupsController

```
[Route("[controller]")]
[ApiController]
[AuthorizeMiddleware]
public class GroupsController(IGroupRepository groupRepository) : BaseController
{
    [HttpGet]
    public async Task<ActionResult<List<Group>>> GetAllGroups()
    {
        var userId = GetCurrentUserId();
        var vaults = await groupRepository.GetGroupsAsync(userId);
        return Ok(vaults);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Group>> GetGroupById(int id)
    {
        var group = await groupRepository.GetByIdAsync(id);
        if (group == null)
        {
            return NotFound();
        }

        return Ok(group);
    }

    [HttpPost("new")]
    public async Task<ActionResult<List<Group>>> CreateGroup([FromBody] AddGroupDto group)
    {
        var userId = GetCurrentUserId();
        group.ShareWith ??= [];

        await groupRepository.CreateGroupAndAssignRolesAsync(group.GroupName, userId, group.ShareWith);
        return Ok("Group created");
    }

    [HttpPost("share")]
    public async Task<ActionResult> ShareToGroup([FromBody] ShareWithGroupDto shareVaultDto)
    {
        var success = await groupRepository.ShareToGroupAsync(shareVaultDto.VaultId, shareVaultDto.GroupsId);
        if (success)
        {
            return Ok("Vault shared to groups successfully.");
        }

        return BadRequest("Failed to share vault to groups.");
    }

    [HttpGet("{groupId}/vaults")]
    public async Task<ActionResult<List<Vault>>> GetVaultsByGroupId(int groupId)
    {
        var vaults = await groupRepository.GetVaultsAsync(groupId);
        return Ok(vaults);
    }

    [HttpGet("{groupId}/members")]
    public async Task<ActionResult<List<GroupUserRole>>> GetMembersByGroupId(int groupId)
    {
        var members = await groupRepository.GetMembersAsync(groupId);
        return Ok(members);
    }

    [HttpPost("{groupId}/members")]
    public async Task<ActionResult<string>> AddMemberToGroup(int groupId, AddUserToGroupDto model)
    {
        var result = await groupRepository.AddMember(groupId, model);
        return result switch
        {
            "User not found." => NotFound(result),
            "User is already a member." => BadRequest(result),
            _ => Ok(result)
        };
    }
}
```

Obrázek 50: GroupsController třída

Třída `GroupsController` je kontrolerem, který poskytuje API endpointy pro operace související se správou skupin (`Groups`). Tento kontroler je odvozen od základní třídy `BaseController`, která poskytuje společné funkcionality pro všechny kontrolery v aplikaci, jako je získávání ID aktuálně přihlášeného uživatele. `GroupsController` je

zaměřen na operace, které umožňují správu skupin, jejich členů a sdílení trezorů mezi skupinami.
Tato třída je řízena několika atributy, které určují její chování v rámci ASP.NET Core frameworku:

- [Route("[controller]")]: Tento atribut definuje základní cestu pro všechny metody tohoto kontroleru. `"[controller]"` je nahrazen názvem kontroleru, tedy "Groups". Výsledkem je, že všechny metody tohoto kontroleru budou mít cestu začínající `/Groups`.
- [ApiController]
- [AuthorizeMiddleware]

Kontroler využívá dependency injection (DI) pro získání instance `IGroupRepository`, což je rozhraní, které definuje operace nad úložištěm skupin. Toto rozhraní umožňuje kontroleru přistupovat k datům skupin a manipulovat s nimi bez nutnosti přímé interakce s databází, což podporuje princip oddělení logiky od datové vrstvy.

Metody třídy:

- `GetAllGroups()`: Tato metoda získává seznam všech skupin, do kterých je aktuálně přihlášený uživatel zapojen. Endpoint pro tuto metodu je `/Groups`.
- `GetGroupById(int id)`: Tato metoda vrací detailní informace o konkrétní skupině podle jejího ID. Pokud skupina neexistuje, vrací chybu 404 (Not Found). Endpoint je `/Groups/{id}`.
- `CreateGroup([FromBody] AddGroupDto group)`: Tato metoda vytváří novou skupinu na základě údajů, které jsou odeslány v těle požadavku jako JSON objekt typu `AddGroupDto`. Nová skupina je přidána do systému, přičemž aktuální uživatel se stává jejím vlastníkem. Používá HTTP metodu POST s endpointem `/Groups/new`.
- `ShareToGroup([FromBody] ShareWithGroupDto shareVaultDto)`: Tato metoda umožňuje sdílet konkrétní trezor do jedné nebo více skupin. Pokud je operace úspěšná, vrací se potvrzení, jinak vrací chybu. Endpoint je `/Groups/share`.
- `GetVaultsByGroupId(int groupId)`: Tato metoda vrací seznam trezorů, které jsou sdíleny v rámci konkrétní skupiny. Endpoint pro tuto metodu je `/Groups/{groupId}/vaults`.

- `GetMembersByGroupId(int groupId)`: Tato metoda vrací seznam členů skupiny spolu s jejich rolemi v rámci dané skupiny. Endpoint je `/Groups/{groupId}/members`.
- `AddMemberToGroup(int groupId, AddUserToGroupDto model)`: Tato metoda přidává nového člena do skupiny. Pokud člen již existuje, nebo uživatel není nalezen, vrací se příslušná chyba. Endpoint je `/Groups/{groupId}/members`.

4. IdentityController

```
[Route("[controller]")]
[ApiController]
public class IdentityController(IUserRepository userRepository, IIdentityService identityService) : ControllerBase
{
    [HttpPost("register")]
    public async Task<IActionResult> Register([FromBody] IdentityDto newUser)
    {
        var result = await identityService.RegisterAsync(newUser);

        if (result)
        {
            return Ok("User registered successfully.");
        }
        return BadRequest("Username or email is already taken.");
    }

    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody] IdentityDto login)
    {
        var token = await identityService.LoginUserAsync(login);

        if (token is not null)
        {
            return Ok(token);
        }

        return Unauthorized("Invalid username or password.");
    }

    [HttpPost("logout")]
    public async Task<IActionResult> LogoutAsync([FromBody] string token)
    {
        var storedRefreshToken = await userRepository.GetByTokenAsync(token);
        if (storedRefreshToken == null || storedRefreshToken.Expires < DateTime.UtcNow)
        {
            return Unauthorized("Invalid refresh token.");
        }

        await userRepository.DeleteAsync(storedRefreshToken);
        Console.WriteLine($"Token: {storedRefreshToken.Token} Deleted successfully.");
        return Ok("Token deleted");
    }

    [HttpPost("refresh")]
    public async Task<IActionResult> RefreshToken([FromBody] RefreshTokenDto refreshTokenDto)
    {
        var storedRefreshToken = await userRepository.GetByTokenAsync(refreshTokenDto.Token);
        if (storedRefreshToken == null || storedRefreshToken.Expires < DateTime.UtcNow)
        {
            return Unauthorized("Invalid refresh token.");
        }

        await userRepository.DeleteAsync(storedRefreshToken);
        Console.WriteLine($"Token: {storedRefreshToken.Token} Deleted successfully.");

        var userId = storedRefreshToken.UserId;
        var user = await userRepository.GetByIdAsync(userId);
        if (user == null)
        {
            return Unauthorized("User associated with refresh token not found.");
        }

        var newTokens = identityService.GenerateJwtToken(userId, user.UserName);

        return Ok(newTokens);
    }
}
```

Obrázek 51: IdentityController třída

Třída `IdentityController` slouží jako kontroler pro operace související s autentizací a správou uživatelské identity v aplikaci. Tato třída poskytuje API endpointy pro registraci, přihlášení, odhlášení a obnovu tokenů, což jsou klíčové funkce pro správu uživatelských účtů a jejich přístupových práv. Oproti předchozím kontrolerům nevyžaduje autorizaci a nedědí z `BaseController` třídy. Používá metody z rozhraní IuserRepository a IidentityService, které poskytují metody pro přístup k datům o uživatelích, včetně jejich autentizačních tokenů a operace související s autentizací uživatelů, jako je registrace, přihlášení a generování JWT tokenů.

- [Route("[controller]")]: Tento atribut určuje základní cestu pro všechny metody tohoto kontroleru. `"[controller]"` je nahrazen názvem kontroleru, tedy "Identity". To znamená, že všechny metody v této třídě budou dostupné pod cestou `/Identity`.
- [ApiController]: Tento atribut usnadňuje práci s kontrolerem, zejména co se týče validace modelů, návratu správných HTTP kódů a dalších funkcí specifických pro API kontrolery.

Metody třídy:

- `Register([FromBody] IdentityDto newUser)`: Tato metoda umožňuje registraci nového uživatele. Přijímá údaje o novém uživateli v těle požadavku ve formě `IdentityDto`. Pomocí `identityService` provádí registraci uživatele. Pokud registrace proběhne úspěšně, metoda vrací HTTP odpověď 200 (OK) s potvrzením. Pokud je uživatelské jméno nebo email již obsazený, vrací chybu 400 (Bad Request).
- `Login([FromBody] IdentityDto login)`: Tato metoda umožňuje přihlášení uživatele na základě zadaného uživatelského jména a hesla. V případě úspěšného přihlášení vrací JWT token, který umožňuje ověřený přístup k chráněným částem aplikace. Pokud přihlášení selže (například kvůli nesprávným přihlašovacím údajům), vrací se chyba 401 (Unauthorized).
- `LogoutAsync([FromBody] string token)`: Tato metoda zajišťuje odhlášení uživatele tím, že invaliduje a smaže refresh token, který byl použit pro udržování přihlášeného stavu. Pokud je token neplatný nebo již expiroval, vrací se chyba 401 (Unauthorized).
- RefreshToken([FromBody] RefreshTokenDto refreshTokenDto): Tato metoda zajišťuje obnovu přístupových tokenů na základě platného refresh tokenu. Pokud je token

platný a neexpirovaný, metoda vygeneruje nové přístupové tokeny (JWT) a starý token smaže. Pokud je token neplatný nebo expirovaný, vrací se chyba 401 (Unauthorized).

11.4.3 Middleware

Middleware jsou komponenty, které se nacházejí mezi příjmem požadavku a odesláním odpovědi. Mohou vykonávat úkoly, jako je autentizace, logování, zpracování výjimek nebo úprava požadavků a odpovědí.

Třídy `AuthorizationMiddleware` a `AuthorizeMiddlewareAttribute` tvoří dohromady mechanismus, který zajišťuje, že uživatelé musí být autentizováni před tím, než získají přístup k chráněným částem aplikace. Middleware provádí ověření tokenu a nastavuje kontext uživatele, zatímco atribut `AuthorizeMiddlewareAttribute` umožňuje snadnou aplikaci tohoto middleware na konkrétní akce nebo kontrolery.

1. **AuthorizationMiddleware**

Třída `AuthorizationMiddleware` je middleware komponenta v ASP.NET Core aplikaci, která se stará o zpracování a ověření autentizačních tokenů v HTTP požadavcích. Middleware je navržen tak, aby fungoval před samotným zpracováním požadavku v kontrolerech a zajistil, že jen autentizovaní uživatelé mají přístup k chráněným zdrojům.
Middleware zpracovává každý příchozí HTTP požadavek a provádí následující kroky:

1. Kontrola hlavičky Authorization:
   Middleware nejprve kontroluje, zda je v HTTP požadavku přítomna hlavička `Authorization`. Pokud hlavička chybí a anonymní přístup není povolen, požadavek je odmítnut se stavem 401 Unauthorized.

2. Deserializace tokenu:
   Pokud je hlavička přítomna, middleware získá token z hlavičky, deserializuje jej (pomocí JSON) a extrahuje hodnotu `accessToken`.

3. Ověření tokenu:
   Token je poté předán službě `IIdentityService`, která se stará o ověření platnosti tokenu. Pokud je token neplatný, požadavek je opět odmítnut se stavem 401 Unauthorized.

4. Nastavení uživatele v kontextu:
   Pokud je token platný, middleware nastaví `ClaimsPrincipal` (identitu uživatele)

na `HttpContext.User`. Dále zkontroluje, zda uživatel existuje v systému pomocí `IUserRepository`.

5. Pokračování v požadavku:
   Pokud vše projde v pořádku, požadavek je předán dále k dalšímu middleware nebo kontroleru.

Výhodami je centralizované ověřování autentizačních tokenů a zároveň možnost povolit anonymní přístup k některým akcím nebo částem aplikace.

2. AuthorizeMiddlewareAttribute

Třída `AuthorizeMiddlewareAttribute` je atribut, který lze aplikovat na kontrolery nebo jejich metody. Tento atribut integruje middleware `AuthorizationMiddleware` přímo do životního cyklu zpracování požadavků v ASP.NET Core.
`AuthorizeMiddlewareAttribute` implementuje rozhraní `IAsyncActionFilter`, které umožňuje vykonání vlastní logiky před a po akci kontroleru. Tento atribut funguje následovně:

1. Inicializace middleware:
   Atribut inicializuje middleware `AuthorizationMiddleware` a předá mu aktuální kontext HTTP, služby `IIdentityService` a `IUserRepository`, a informaci o tom, zda je povolen anonymní přístup (`allowAnonymous`).

2. Vložení middleware do řetězce zpracování akce:
   Middleware je následně spuštěn v rámci zpracování požadavku na akci kontroleru.

Výhodami je snadná integrace autentizace a autorizace přímo na úrovni kontroleru nebo jeho metod. Dále taky podpora konfigurace, kdy lze povolit nebo zakázat anonymní přístup pro konkrétní akce.

11.5 Konfigurace

API vyžaduje konfiguraci pro správu nastavení, jako jsou připojovací řetězce a JWT secrets v souboru appsettings.json, nastavení autentizace a další parametry v souboru Program.cs. V tomto souboru se často načítají a nastavují různé konfigurace, které mohou být specifické pro různá prostředí (vývoj, testování, produkce). Tyto konfigurace jsou obvykle načítány z konfiguračních souborů, jako je právě appsettings.json, nebo z proměnných prostředí.

```
{
  "ConnectionStrings": {
    "Default": "Server=localhost;Database=PasswordManager;User Id=sa;Password=5Ab*cD9E;TrustServerCertificate=True"
  },
  "Jwt": {
    "Secret": "SwiftVault@Rules@The@SZZ@Of@2024",
    "AccessTokenExpirationInDays": 30,
    "RefreshTokenExpirationInDays": 7
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Obrázek 52: Appsettings.json

11.5.1 Konfigurace služeb (Dependency Injection)

V souboru Program.cs se konfiguruje Dependency Injection (DI) kontejner. To zahrnuje registraci služeb, middleware a dalších závislostí, které bude aplikace používat. Příkladem může být registrace databázového kontextu, autentizačních služeb, loggeru, konfigurací apod. Jelikož existují soubory s DI v Application a Infrastructure projektech, stačí zaregistrovat statické metody, které obsahují.

```
builder.Services
    .AddApplication()
    .AddInfrastructure();

var connectionString = builder.Configuration.GetConnectionString("Default") ??
                       throw new NullReferenceException("No connection string");

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Obrázek 53: Registrace služeb a databázového kontextu

11.5.2 Nastavení CORS

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowBlazorClient",
        policy =>
        {
            policy.WithOrigins("http://localhost:5057") // Blazor app URL
                .AllowAnyHeader()
                .AllowAnyMethod()
                .AllowCredentials();
        });
});
```

Obrázek 54: Nastavení CORS

Tato část kódu definuje CORS politiku s názvem "AllowBlazorClient". Tato politika povoluje přístup aplikace Blazor běžící na `http://localhost:5057` k API, včetně všech

hlaviček, metod a také povoluje přenos přihlašovacích údajů (credentials), jako jsou cookies nebo autentizační hlavičky.

11.5.3 Konfigurace middleware

Taková konfigurace zahrnuje nastavení routingu, autentizace, autorizace, zpracování chyb, statických souborů, MVC, apod.

```
app.UseHttpsRedirection()
    .UseRouting()
    .UseAuthentication()
    .UseAuthorization()
    .UseCors("AllowBlazorClient")
    .UseEndpoints(endpoints => { endpoints.MapControllers(); })
```

Obrázek 55: Konfigurace middleware

- `UseHttpsRedirection()`: Přesměruje HTTP požadavky na HTTPS.
- `UseRouting()`: Aktivuje routování, což umožňuje mapování požadavků na příslušné kontrolery a akce.
- `UseAuthentication()`: Zapíná autentizaci uživatelů.
- `UseAuthorization()`: Zapíná autorizaci, což umožňuje kontrolu oprávnění.
- `UseCors("AllowBlazorClient")`: Aplikuje dříve definovanou CORS politiku.
- `UseEndpoints(endpoints => { endpoints.MapControllers(); })`: Mapuje HTTP požadavky na kontrolery v aplikaci.

11.5.4 Globální zpracování chyb

Tento middleware zachycuje neobsloužené výjimky a zajišťuje, že klient dostane zpět 500 Internal Server Error s podrobnou chybovou zprávou ve formátu JSON. Kromě toho loguje chybu a její stack trace do konzole.

```
app.UseExceptionHandler(appError =>
    {
        appError.Run(async context =>
        {
            context.Response.StatusCode = StatusCodes.Status500InternalServerError;
            context.Response.ContentType = "application/json";

            var contextFeature = context.Features.Get<IExceptionHandlerFeature>();
            if (contextFeature != null)
            {
                var error = new { message = contextFeature.Error.Message };
                await context.Response.WriteAsync(JsonSerializer.Serialize(error));

                // Logging
                Console.WriteLine($"Error: {contextFeature.Error}");
                Console.WriteLine($"Stack Trace: {contextFeature.Error.StackTrace}");
            }
        });
    });
```

Obrázek 56: Globální zpracování chyb

ZÁVĚR

Do téhle práce jsem šel s tím, že jsem neměl tušení, jak pro mě budou některé části opravdu složité, ale hlavně časově náročné. Navzdory tomu je výsledkem této bakalářské práce funkční webová aplikace. Poskytuje jistou míru zabezpečení a nejrůznější akce. JSON Web Tokeny skvěle zabezpečují aplikaci před nedovoleným přístupem. Uživatel se může registrovat i přihlásit a myslím si, že vzhledově stránky nemohl dopadnout lépe. Kombinace oranžové s odstíny tmavě šedé až černé barvy skvěle ladí.

Při vývoji nastaly mnohé problémy a překážky, avšak jejich vyřešení bylo úspěšné díky ChatGPT a této aplikaci podobným repositářům na GitHubu. A taky samozřejmě čas, kterého bylo stráveno až přespříliš, nad některými implementacemi.

Myslím, že práce má rozhodně potenciál a je možné do budoucna rozšířit její funkčnost nejen o správu hesel.

SEZNAM POUŽITÉ LITERATURY

[1] OKTA. What is Authentication? Online. Auth0: Secure access for everyone. But not just anyone. C2024. Dostupné z: https://auth0.com/intro-to-iam/what-is-authentication. [cit. 2024-02-28].

[2] BONE, Harry. What is two-factor authentication (2FA)? Online. The Proton Blog. 2023, roč. 2023, č. 2, s. 1. Dostupné z: https://proton.me/blog/what-is-two-factor-authentication-2fa. [cit. 2024-05-12].

[3] What is two-factor authentication? Online. In: Microsoft Security. C2024. Dostupné z: https://www.microsoft.com/en-us/security/business/security-101/what-is-two-factor-authentication-2fa. [cit. 2024-05-12].

[4] The Importance of Computer Software Updates & Security Patches. Online. Microsoft 365 Life Hacks. 2022, roč. 2022, č. 1, s. 1. Dostupné z: https://www.microsoft.com/en-us/microsoft-365-life-hacks/privacy-and-safety/computer-software-update. [cit. 2024-05-12].

[5] CRAWFORD, Douglas. *What is a password manager and why do I need one?* Online. The Proton Blog. 2023, 18.8.2023. Dostupné z: https://proton.me/blog/what-is-a-password-manager. [cit. 2024-02-27].

[6] CRAWFORD, Douglas. How to create strong passwords you'll actually remember. Online. *The Proton Blog*. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://proton.me/blog/create-remember-strong-passwords. [cit. 2024-05-12].

[7] BURNETT, Mark. *Perfect Password: Selection, Protection, Authentication*. Syngress, 2005. ISBN 1597490415.

[8] GRIGUTYTĖ, Monika. What is password cracking and what techniques do hackers use. Online. *NordVPN*. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://nordvpn.com/blog/password-cracking/. [cit. 2024-05-12].

[9] BHATT, Paritosh. The Art and Science Behind Password Managers. Online. Medium. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://medium.com/@paritoshblogs/the-art-and-science-behind-password-managers-fbf5fb9c7f99. [cit. 2024-05-12].

[10] O'SULLIVAN, Fergus. Are password managers safe? Online. *The Proton Blog*. 2024, roč. 2024, č. 2, s. 1. Dostupné z: https://proton.me/blog/are-password-managers-safe. [cit. 2024-05-12].

[11] MARTINOLI, Marco. What is end-to-end encryption and how does it work? Online. *The Proton Blog*. 2023, roč. 2023, č. 2, s. 1. Dostupné z: https://proton.me/blog/what-is-end-to-end-encryption. [cit. 2024-05-12].

[12] *Introduction to different types of password managers*. Online. In: Zoho Vault. C2024. Dostupné z: https://www.zoho.com/vault/educational-content/different-types-of-password-managers.html. [cit. 2024-05-12].

[13] BHATT, Paritosh. The Art and Science Behind Password Managers. Online. Medium. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://medium.com/@paritoshblogs/the-art-and-science-behind-password-managers-fbf5fb9c7f99. [cit. 2024-05-12].

[14] *The Future of Password Management: Trends to Watch*. Online. In: Privacy Hawk. C2024. Dostupné z: https://privacyhawk.com/resources/the-future-of-password-management-trends-to-watch/. [cit. 2024-05-12].

[15] BISHOP, Edward. The Future of Password Management: Is it time to say goodbye to Password Managers? Online. *Market Business News*. 2023, roč. 2023, č. 1, s. 1. Dostupné z: https://marketbusinessnews.com/the-future-of-password-management-is-it-time-to-say-goodbye-to-password-managers/327495/. [cit. 2024-05-12].

[16] O'SULLIVAN, Fergus. What is a passkey? Online. *The Proton Blog*. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://proton.me/blog/what-is-a-passkey. [cit. 2024-05-12].

[17] WASIKE, Bravin. 30 Best Web Development Frameworks for 2023: A Comprehensive Guide. Online. *Dev Community*. 2023, roč. 2023, č. 2, s. 1. Dostupné z: https://dev.to/bravinsimiyu/30-best-web-development-frameworks-for-2023-a-comprehensive-guide-512i. [cit. 2024-05-12].

[18] Přehled ASP.NET Core. Online. *Microsoft Learn*. 2023, roč. 2023, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0. [cit. 2024-05-12].

[19] ASP.NET Core Blazor. Online. Microsoft Learn. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-8.0. [cit. 2024-05-12].

[20] *Create web APIs with ASP.NET Core*. Online. Microsoft Learn. 2024. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-8.0. [cit. 2024-08-20].

[21] Entity Framework Core. Online. *Microsoft Learn*. 2021, roč. 2021, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/en-us/ef/core/. [cit. 2024-05-12].

[22] M., Sopha. What Is a Development Environment? How Does It Differ From an Integrated Development Environment (IDE)? Online. *Hostinger Tutorials*. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://www.hostinger.com/tutorials/development-environment. [cit. 2024-05-12].

[23] OBREGON, Alexander. Embrace the World of Efficient Coding with JetBrains Rider. Online. *Medium*. 2023, roč. 2023, č. 1, s. 1. Dostupné z: https://medium.com/@AlexanderObregon/embrace-the-world-of-efficient-coding-with-jetbrains-rider-f67524000b5b. [cit. 2024-05-12].

[24] *Docker overview*. Online. Docker Docs. C2013-2024. Dostupné z: https://docs.docker.com/get-started/overview/. [cit. 2024-05-12].

[25] *Docker Compose overview*. Online. Docker Docs. C2013-2024. Dostupné z: https://docs.docker.com/compose/. [cit. 2024-05-12].

[26] What is SQL Server? Online. *Microsoft Learn*. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16. [cit. 2024-05-12].

[27] *What is SQL Server*. Online. In: SQL Server Tutorial. C2023. Dostupné z: https://www.sqlservertutorial.net/getting-started/what-is-sql-server/. [cit. 2024-05-12].

[28] grapes. What are the advantages of running SQL Server in a Docker container? Online. In: Stack Overflow. 2019. Dostupné z: https://stackoverflow.com/questions/54841211/what-are-the-advantages-of-running-sql-server-in-a-docker-container. [cit. 2024-05-12].

[29] Entity Framework Core. Online. *Microsoft Learn*. 2023, roč. 2023, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/cs-cz/ef/core/. [cit. 2024-05-12].

[30] ASP.NET Core Blazor s Entity Framework Core (EF Core). Online. Microsoft Learn. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/blazor-ef-core?view=aspnetcore-8.0. [cit. 2024-05-12].

[31] ASP.NET Core Blazor. Online. Microsoft Learn. 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-8.0. [cit. 2024-05-12].

[32] Identity model customization in ASP.NET Core. Online. Microsoft Learn. 2022, roč. 2022, č. 1, s. 1. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-8.0. [cit. 2024-05-12].

[33] PINE, David. Learning Blazor. O'Reilly Media, 2022. ISBN 978-1098113247.

[34] What is an application architecture? Online. *Red Hat*. 2020, roč. 2020, č. 1, s. 1. Dostupné z: https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture. [cit. 2024-05-12].

[35] GOD, Patrick. Clean Architecture w/ Blazor 🔥, Blazor in the Industry 🚀, .NET Web Academy Enrollment Closing Soon 🎓. Online. Patrick God Posts. March 2024, roč. 2024, č. 1, s. 1. Dostupné z: https://dotnet8.patrickgod.com/posts/clean-architecture-w-blazor-blazor-in-the-industry-net-web-academy-enrollment-closing-soon. [cit. 2024-05-12].

[36] BERGMAN, Per-Erik. Understanding Clean Architecture. Online. Medium. 2023, roč. 2023, č. 1, s. 1. Dostupné z: https://medium.com/@pererikbergman/understanding-clean-architecture-5f7a39a965fb. [cit. 2024-05-12].

[37] VASCONCELOS, Rubem. Clean Architecture: The concept behind the code. Online. Dev Community. 2022, roč. 2022, č. 1, s. 1. Dostupné z: https://dev.to/rubemfsv/clean-architecture-the-concept-behind-the-code-52do. [cit. 2024-05-12].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

2FA	Dvoufaktorová autentizace
MFA	Multifaktorová autentizace
PIN	Personal Identification Number
SMS	Short Message Service
AES	Advanced Encryption Standard
E2EEE	End-to-End Encryption
XSRF/CSRF	Cross-Site Request Forgery
RPC (gRPC)	Remote Procedure Call
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
CSR	Client-Side Rendering
PWA	Progressive Web Application
DI	Dependency Injection
MSSQL	Microsoft SQL Server
SQL	Structured Query Language
ORM	Object-Relational Mapping
EF	Entity Framework
IDE	Integrated Development Environment
CI/CD	Continuous Integration/Continuous Deployment
REST	REpresentational State Transfer
API	Application Programming Interface
UNIX	Uniplexed Information and Computing Service
CLI	Command Line Interface
RDBMS	Relational Database Management System
SQLOS	SQL Server Operating System
SAN	Storage Area Network
EULA	End User License Agreement
env	Environment
VPS	Virtual Private Server
GCP	Google Cloud Platform
AKS	Azure Kubernetes Service
AWS	Amazon Web Services
LINQ	Language Integrated Query
npm	Node Package Manager
SPA	Single Page Application
SSR	Server-Side Rendering
IoC	Inversion of Control

SDK	Software Development Kit
GUID	Globally Unique Identifier
UI	User Interface
UX	User Experience
CRUD	Create, Read, Update, Delete
HTTPS	HyperText Transfer Protocol Secure
JWT	JSON Web Token
CORS	Cross-Origin Resource Sharing

SEZNAM OBRÁZKŮ