TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Komunikační platforma s možností organizace událostí

Bakalářská práce

Studijní program: B0613A140005 Informační technologie
Studijní obor: Aplikovaná informatika

Autor práce: **Bruno Pfohl**
Vedoucí práce: Ing. Jan Hybš
Ústav nových technologií a aplikované informatiky

Liberec 2022

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Komunikační platforma s možností organizace událostí

Jméno a příjmení:	**Bruno Pfohl**
Osobní číslo:	M19000036
Studijní program:	B0613A140005 Informační technologie
Specializace:	Aplikovaná informatika
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	**2021/2022**

Zásady pro vypracování:

1. Proveďte rešerši webových technologií a technologií pro komunikaci v reálném čase.
2. Navrhněte a implementujte modulární webovou aplikaci sloužící jako sociální komunikační platforma pro středně velké podniky.
3. Navrhněte a realizujte responzivní a intuitivní uživatelského rozhraní webové aplikace.
4. Implementace real-time komunikaci pro registrované uživatele.
5. Uživatelům umožněte správu kalendářních událostí. Implementujte funkcionalitu, která bude predikovat a následně automaticky vytvářet nové kalendářní události na základě preferencí uživatelů.
6. Proveďte kritickou analýzu bezpečnosti aplikace a ukládání dat.

Rozsah grafických prací: dle potřeby dokumentace
Rozsah pracovní zprávy: 30-40 stran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

[1] Halvorsen, H.-P. (n.d.). *Web Programming ASP.NET Core*. [online] Available at: https://www.halvorsen.blog/documents/programming/csharp/textbook/aspnet/Web%20Programming%20-%20ASP.NET%20Core.pdf
[2] wadepickett (2021). *ASP.NET documentation*. [online] Microsoft.com. Available at: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0
[3] Aguilar, J. (n.d.). *Professional SignalR Programming in Microsoft ASP.NET*. [online] Available at: https://ptgmedia.pearsoncmg.com/images/9780735683884/samplepages/9780735683884.pdf
[4] Frederick, C. and Adrian-Tudor Pãnescu (2019). *Data Storage*. [online] ResearchGate. Available at: https://www.researchgate.net/publication/337691364_Data_Storage#pfb

Vedoucí práce: Ing. Jan Hybš
Ústav nových technologií a aplikované informatiky

Datum zadání práce: 12. října 2021
Předpokládaný termín odevzdání: 16. května 2022

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2021

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

12. 5. 2022 Bruno Pfohl

Komunikační platforma s možností organizace událostí

Abstrakt

Tato bakalářská práce se zabývá softwarovým návrhem a implementací komunikační platformy s možností organizace kalendářních událostí. Po registraci mohou uživatelé sdílet zprávy s ostatními uživateli, případně s nimi pořádat skupinové kalendářní události. Každý uživatel si může stanovit své preference a volné časové bloky. V případě, kdy se preference více uživatelů shodují, aplikace sama vytvoří kalendářní událost, do které uživatele zařadí.

Webová aplikace je realizována jako SPA, což znamená, že uživatel po prvním načtení stránky zůstává stále na stejné stránce a veškeré změny obsahu na stránce obstarává Javascript kód. Uživatelské rozhraní je naprogramováno pomocí technologií TypeScript, React.js a Next.js. Data jsou ukládána do databáze MS SQL a přístup k nim zprostředkovává klientovi ASP .NET 5 WEB API, které využívá framework GraphQL-NET.

Aplikace je funkční, responzivní a zabezpečená. Všechna data od uživatele prochází validací, což zamezuje tomu, aby data jiného uživatele byla odcizena nebo upravena. Výsledkem je spolehlivá komunikační platforma, na kterou mohou přistupovat uživatelé z různých zařízení.

Klíčová slova: chat, sociální síť, .NET, React.js, GraphQL

Práce

Abstract

This report describes the design and implementation of a communication platform, which also serves as a calendar event planner. Users can create accounts, which then they can use to send messages to one another or to create a group calendar event. Each user can set his preferences as to when they have time to attend a specific type of calendar event. If there are multiple users with corresponding preferences, an event is automatically created.

Application is a single page, which means that the content is initially rendered on the server, then passed to the user's machine, which takes care of its updates by running a JavaScript code. The user interface is built with TypeScript, React.js, and Next.js. Application data is stored in MS SQL relational database. The client application communicates with ASP .NET 5 WEB API with GraphQL-NET framework installed, to manipulate the database.

The application is functional, responsive, and secure. All data passed from the user is validated, so a data breach can't happen. Therefore, the result of this work is a reliable communication platform, which can be accessed from many kinds of devices.

Keywords: chat, social network, .NET, React.js, GraphQL

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Janu Hybšovi, který byl velmi ochotný se mnou práci pravidelně konzultovat. Dále bych chtěl poděkovat Lucii Ječmeňové, která mi pomáhala s testováním aplikace a opravou chyb v tomto dokumentu.

Obsah

Seznam zkratek 9

1 Rešeršní část **11**
1.1 Předchozí práce 11
1.1.1 Shrnutí funkcionality aplikace 11
1.1.2 Databáze 11
1.1.3 Backend API 12
1.1.4 Front end aplikace 13
1.2 Stylizace React komponent 14
1.2.1 MaterialUI framework 14
1.2.2 Tailwind CSS 15
1.2.3 Porovnání frameworků MaterialUI a Tailwind CSS 15
1.3 Generování TypeScript typů pro klientskou aplikaci 15
1.3.1 GraphQL Code Generator 16
1.3.2 Shrnutí 16
1.4 Optimalizace vykreslování React.js aplikace 17
1.4.1 Využití konstruktu Memo 17
1.4.2 Hook useCallback 18
1.4.3 Hook useMemo 18
1.4.4 React.Lazy - dodatečné načítání komponent 18
1.5 Domain Driven Design (DDD) 19
1.5.1 Doménový model 19
1.5.2 Společný jazyk 19
1.5.3 Ohraničený kontext (Bounded Context) 19
1.5.4 Entity a eliminace anemického modelu 20
1.5.5 Hodnotové objekty (Value Objects) 20
1.5.6 Agregát (aggregate route) 21
1.5.7 Služby 22

2 Návrh doménového modelu **23**
2.1 Uživatelé a komunikační kanály 23
2.2 Zasílání zpráv v komunikačním kanálu 24
2.3 Kalendářní události 24
2.4 Přání o vytvoření události 25

3 Implementace API **26**
3.1 Projekt Chattoo.Domain 27
3.1.1 Entity 27
3.1.2 Hodnotové objekty 28
3.1.3 Repozitáře 28
3.2 Projekt Chattoo.Application 29
3.3 Projekt Chattoo.Infrastructure 30
3.4 Projekt Chattoo.GraphQL 31
3.4.1 Autentizace/autorizace uživatele 31
3.4.2 Sestavení API 32
3.5 Automatické vytváření událostí 32
3.5.1 Algoritmus pro nalezení shody 32

4 Implementace uživatelského rozhraní **34**
4.1 Framework Next.js 34
4.2 Komunikace s webovým API 35
4.3 Autorizace uživatele 36
4.4 Struktura klientské aplikace a rozložení komponent 37
4.4.1 Optimalizace vykreslování 39

5 Analýza bezpečnosti aplikace **40**
5.1 Validace na straně klienta 41
5.2 Validace před zpracováním dotazu 41
5.3 Validace v doménové vrstvě 42

6 Prezentace aplikace **43**

Použitá literatura **47**

Seznam zkratek

API	Application Programming Interface - rozhraní pro programování aplikací
ASP	Active Server Pages - technologie od Microsoft pro programování webových aplikací
CQRS	Command and Query Responsibility Segregation - návrhový vzor
REST API	Representational State Transfer API - typ API
JWT	JSON Web Token - standard pro elektronický podpis dat
SQL	Structured Query Language - standardizovaný strukturovaný dotazovací jazyk
HTML	Hyper Text Markup Language - značkovací jazyk pro tvorbu webových stránek
DOM	Document Object Model - objektový model dokumentu
DDD	Domain Driven Design - definované postupy pro vývoj softwaru
SPA	Single Page Application - webová stránka vykreslovaná na klientském zařízení
CSS	Cascading Style Sheets - kaskádové styly

Úvod

Práce si klade za cíl vytvoření webové aplikace, která bude sloužit jako sociální síť pro komunikaci v reálném čase a organizaci kalendářních událostí. Registrovaní uživatelé aplikace mohou nastavit, kdy mají volno na zapojení se do události s určitými parametry. Pokud aplikace nalezne shodu mezi preferencemi uživatelů, automaticky vytvoří událost s odpovídajícími parametry a uživatele do ní přidá. Motivace za tímto projektem je vytvořit jednoduchou sociální síť, která maximalizuje propojení mezi uživateli.

Rešeršní část se zabývá stanovením technologií a obecných principů, na kterých má být aplikace postavena. Tato práce částečně vychází z předešlého bakalářského projektu [1], proto se část kapitoly zabývá i shrnutím stavu původní práce a určením postupu, jak na práci navázat. Dále se rešerše zabývá technologiemi React (front-end framework), Next.js, MaterialUI (balíček front-end komponent) a GraphQL (API). Část kapitoly se věnuje i softwarovému návrhu DDD (Domain Driven Design), ze kterého byly pro tuto práci využity některé základní principy.

Praktická část stručně shrnuje postup vytvoření aplikace od návrhu až po kompletní implementaci. Začíná kapitolou s návrhem doménového modelu aplikace, ze kterého vychází implementace webového API. Na tuto kapitolu pak navazují další, které dále popisují postup implementace GraphQL API. Pojednávají převážně o implementaci entit, organizaci kódu a sestavení webového API.

V kapitole o automatickém vytváření událostí je stručně popsán algoritmus pro nalezení shody mezi preferencemi uživatelů. Součástí je i analýza algoritmu, konkrétně časové složitosti a možných omezení. Algoritmus je využit pro úlohu, která při každé změně preferencí uživatelů zkontroluje možnou shodu, případně vytvoří kalendářní událost.

Implementace klientské aplikace je popsána v kapitole 3. Aplikace je vytvořena pomocí technologií React.js, MaterialUI a ApolloClient. Kapitola popisuje základní strukturu komponent a komunikaci mezi uživatelským rozhraním a webovým API, včetně autorizace.

Veškerá vstupní data od uživatele jsou validována v klientské aplikaci i na serveru. Výsledkem je, že uživatel nemá možnost zobrazit nebo upravit data jiného uživatele. Touto problematikou se detailně zabývá kapitola 5.

1 Rešeršní část

V této kapitole je popsán průběh rešerše, která si klade za cíl určit technologie a postupy pro vývoj webové aplikace. Vzhledem k tomu, že tato práce navazuje na předchozí práci [1], součástí rešerše je i shrnutí stavu předchozí práce a určení postupu, jak na ni navázat.

1.1 Předchozí práce

Tato práce navazuje na předchozí bakalářský projekt „Komunikační platforma s možností organizace událostí" [1]. Jedná se o multiplatformní webovou aplikaci, do které se mohou uživatelé zaregistrovat a v rámci komunikačních kanálů spolu sdílet zprávy v reálném čase.

1.1.1 Shrnutí funkcionality aplikace

Aplikace umožňuje registraci uživatelů, kteří mezi sebou mohou sdílet zprávy v rámci komunikačních kanálů. Jedná se o klasickou formu chatu, kdy nové zprávy uživatelů jsou okamžitě rozeslány/zobrazeny všem klientům.

Jedná se o SPA (single page application) aplikaci, kdy uživatel načte aplikaci v okně prohlížeče a okno nemusí aktualizovat. O veškerou změnu obsahu se stará Javascript kód, konkrétně knihovna React.js (viz. kapitola 1.1.4), která komunikuje s ASP .NET GraphQL API (viz. kapitola 1.1.3).

1.1.2 Databáze

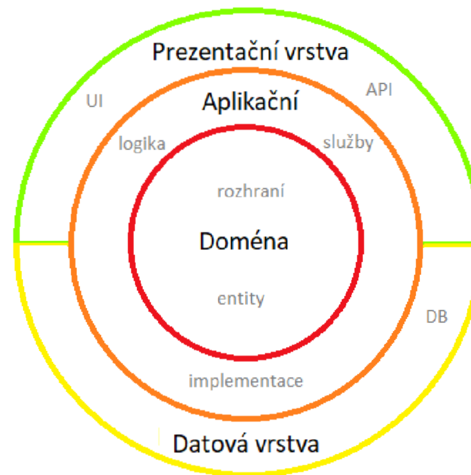
Veškerá data aplikace jsou uložena v MSSQL relační databázi. Databáze je generována pomocí code-first přístupu, tedy v aplikaci jsou definované modely, na základě kterých je automaticky generováno schéma databáze.

Stěžejní tabulky v databázi:

- User - uživatel
- CommunicationChannel - komunikační kanál (skupina uživatelů)
- CommunicationChannelMessage - zpráva z komunikačního kanálu

1.1.3 Backend API

Monolitický projekt, který vychází z šablony CleanArchitecture od vývojáře Jasona Taylora [2]. Architektura vychází z tzv. Onion Architecture, kde veškeré závislosti vedou směrem do jádra aplikace. Struktura projektu je zachycena na následujícím obrázku 1.1.



Obrázek 1.1: Clean architecture

Aplikace je rozdělena na 4 projekty:

- Application
  - definice příkazů a dotazů, které lze vyvolat z API projektu
- Domain
  - entity
  - logika aplikace
- Infrastructure
  - přístup do databáze
  - konfigurace databáze
- GraphQL
  - webové API

Projekt GraphQL nemá přístup k databázovému kontextu a nelze z něj tedy přímo přistupovat do databáze, místo toho se v něm vyvolávají příkazy a dotazy z projektu Application. Toto rozdělení na příkazy a dotazy se nazývá CQRS a pro jeho implementaci aplikace využívá knihovnu MediatR.

Aktuální stav projektu Application není zcela ideální, jednotlivé příkazy a dotazy obsahují značnou část logiky aplikace, která je často duplicitně definovaná napříč projektem. V aktuálním stavu je tedy velmi pravděpodobné, že rozšiřování aplikace by vedlo k nestabilnímu a nepřehlednému kódu.

1.1.4 Front end aplikace

Klientská aplikace je napsaná pomocí knihovny React.js a jazyku Typescript. React.js je webový framework, který vydala společnost Facebook v roce 2013. Kód frameworku je open-source a pracuje na něm velká komunita [3].

Jazyk TypeScript zajišťuje, že veškeré objekty v klientské aplikaci jsou jasně typované, což značně vylepšuje čitelnost a rozšiřitelnost kódu. Definice typů objektů jsou generovány ze schématu GraphQL serveru pomocí nástroje Apollo Codegen.

Pro zasílání dotazů na API využívá aplikace knihovnu Apollo Client, která umožňuje sestavit a odeslat dotazy v GraphQL formátu. Knihovna také v aplikaci slouží pro tzv. state management (správu dat). V případě, kdy klientská aplikace obdrží data od serveru, knihovna zajistí jejich dočasné uložení. Zároveň poskytuje další pokročilé nástroje, např. automatickou obnovu dat.

Kód aplikace je rozdělen do jednotlivých komponent, které obsahují HTML prvky a logiku spojenou s jejich vykreslováním. Vzhled komponent je definován pomocí knihovny Styled Components.

Kód klientské aplikace je do budoucna bohužel velmi těžko rozšiřitelný. Styly komponent definované pomocí knihovny StyledComponents jsou velmi často duplicitně definované napříč různými komponentami. Dalším problémem jsou redundantní definice typů objektů, které byly vygenerovány nástrojem Apollo Codegen. Pro každou možnou odpověď od API nástroj vygeneroval vlastní TypeScript typ bez ohledu na to, že odpovědi mají mnohdy totožnou strukturu.

1.2 Stylizace React komponent

Vzhled React.js komponent lze definovat mnoha způsoby, z nichž každý je vhodný pro jiný typ projektu. Nejzákladnějším způsobem je psát CSS styl přímo do komponenty, tedy nastavit jí vlastnost style. Tento přístup ale často vede ke špatně čitelnému kódu.

Dalším způsobem je vytvořit soubor s příponou CSS, ve kterém budou styly definovány, přičemž tento soubor lze i rozdělit na více souborů pro lepší přehlednost [4]. Každý soubor slouží jako tzv. CSS modul, který je umístěn do složky k určité komponentě, přičemž styl se uplatňuje pouze na danou komponentu. Díky tomuto rozdělení je struktura projektu lépe definována a kód je ve výsledku přehlednější [4].

Velmi populárním způsobem je využití knihoven, které umožňují stylovat komponenty přímo pomocí JavaScript/TypeScript syntaxe [4]. Tento způsob je využit i v dosavadní aplikaci, která využívá knihovnu Styled Components. Výhodou tohoto přístupu je vysoká flexibilita, protože styly komponent lze velmi jednoduše dynamicky měnit pomocí JavaScript kódu.

Pro rychlý vývoj uživatelských rozhraní jsou vhodné knihovny/UI frameworky jakými jsou např. MaterialUI, ReactBootstrap či Tailwind CSS [4]. Vývoj rozhraní pomocí těchto frameworků probíhá oproti standardnímu psaní CSS rychleji, jelikož vývojář nemusí pro vytvoření prototypu aplikace napsat jediný řádek kódu CSS. Místo toho využije předem hotové komponenty. Navíc, výsledný kód je dobře čitelný, protože ho díky využití knihovny vzniká podstatně méně. Většina UI frameworků počítá s individuálními potřebami různých vývojářů a umožňuje vzhled a chování komponent jednoduše upravit.

1.2.1 MaterialUI framework

MaterialUI je open-source framework poskytující komponenty pro React.js aplikace [5]. Vychází z návrhu Material Design od firmy Google, což je sada pravidel, jak by mělo být koncipováno uživatelské rozhraní. Framework je pravidelně aktualizován a jednou z jeho hlavních výhod je skvělá dokumentace. Na oficiální stránce projektu tvůrci sepsali kompletní specifikace frameworku a vlastnosti jednotlivých komponent. Dokumentace obsahuje širokou škálu návodů a jednoduchých ukázkových projektů, kde se vývojář dozví, jak se komponenty využívají v praxi.

Framework nedávno v roce 2021 prošel zásadními změnami [6]. Vyšla nová verze v5, která přinesla řadu oprav a nových funkcí. Framework přešel na nové jádro pro stylizaci komponent. Z původního JSS se přešlo na jádro Emotion. Syntaxe zápisu komponent pro Emotion je téměř stejná jako pro Styled Components [7].

Licence pro použití frameworku stanovuje, že pro nekomerční účely lze framework použít zdarma [8]. Naopak pro komerční účely tvůrci projektu vytvořili placený produkt pro který nabízí i garanci aktualizací a zákaznickou podporu.

1.2.2 Tailwind CSS

Jedná se o CSS framework, který částečně eliminuje potřebu psát CSS kód a zrychluje vývoj [9]. Framework definuje řadu znovupoužitelných CSS tříd, tzv. utilit, které stačí přiřadit elementu v HTML syntaxi.

Využitím kombinace předem definovaných tříd lze vytvořit téměř jakýkoliv styl. Vzhledem k počtu definovaných utilit tříd je velikost CSS souborů značná, jedná se přibližně 3 MB, což může způsobit pomalé načítání stránky [10].

Na potencionální problém s načítáním stránky však autoři mysleli a vytvořili nástroj pro minifikaci výsledného CSS souboru. Po provedení minifikace se do výstupního souboru zapíší pouze styly, které aplikace skutečně využívá. Výsledná velikost souboru se pro většinu aplikací z původních 3 MB sníží na přibližně 10 KB [10].

1.2.3 Porovnání frameworků MaterialUI a Tailwind CSS

Oba frameworky řeší stejný problém, každý jiný způsobem. MaterialUI jde cestou znovupoužitelných komponent, TailwindCSS zase volí cestu znovupoužitelných CSS stylů. Hlavním rozdílem mezi frameworky je, že MaterialUI poskytuje již hotové komponenty, které nedefinují pouze vzhled, ale i chování (např. vyskakovací okna).

Dá se předpokládat, že vývoj uživatelského rozhraní pomocí znovupoužitelných MaterialUI komponent bude rychlejší. Za zmínku ale stojí, že i autoři Tailwind CSS poskytují již hotové komponenty v rámci balíčku TailwindUI. Tento balíček komponent však není zdarma a nejzákladnější balíček přichází s cenovkou přibližně 4000 Kč [11].

Pro účely této práce tedy bude na tvorbu uživatelského rozhraní využita knihovna MaterialUI, protože urychlí vývoj a výsledný kód bude kratší, tím pádem i čitelnější.

1.3 Generování TypeScript typů pro klientskou aplikaci

Klientská aplikace původního projektu je napsaná pomocí jazyku TypeScript, který je silně typovaný. Kód aplikace obsahuje mnoho definic objektů, se kterými aplikace pracuje napříč různými komponentami. Nejčastěji využívané objekty jsou právě ty, které vznikají na základě zpracování odpovědí od API.

Právě proto je důležité, aby typy těchto objektů pro klientskou aplikaci mohly být generovány přímo ze schématu API. V opačném případě by to znamenalo, že při každé změně formátu odpovědí API by musel vývojář upravit ručně typy jak v API, tak i v klientské aplikaci.

V původním projektu jsou typy generované pomocí nástroje Apollo Codegen, který však pro každou možnou odpověď od API vygeneruje vlastní typ. Ve výsledku kód aplikace obsahuje mnoho redundantně definovaných typů.

1.3.1 GraphQL Code Generator

Generátor TypeScript typů, který představuje alternativu k nástroji Apollo Codegen. Pomocí tohoto nástroje lze vygenerovat cílové TypeScript typy ze vstupního souboru, který obsahuje schéma API. Generátor lze i nasměrovat na cílové GraphQL API, ze kterého si schéma sám stáhne [12].

Generátor se instaluje následující příkazy [13]:

```
npm install graphql
npm install @graphql-codegen/cli
npm run graphql-codegen init
npm run install
npm install @graphql-codegen/typescript-react-query
npm install @graphql-codegen/typescript
npm install @graphql-codegen/typescript-operations
```

Následně je nutné vytvořit v kořenové složce React.js projektu soubor codegen.yml, který obsahuje URL API, výstupní soubor pro vygenerované typy a další upřesňující nastavení.

V souboru package.json je vhodné upravit položku “scripts” následujícím způsobem:

```
{
  "scripts": {
    "generate": "graphql-codegen",
  }
}
```

Finálním krokem je spuštění generátoru příkazem „npm run generate“. V případě bezchybného průběhu generátor projde všechny definice GraphQL dotazů v klientské aplikaci a vygeneruje pro ně TypeScript typy, přičemž 1 typ slouží někdy i pro více dotazů. Pro každý dotaz/příkaz vygeneruje tzv. React.js hook, což je metoda pro zavolání dotazu.

1.3.2 Shrnutí

Generátor GraphQL Code Generator oproti Apollo Codegen přináší obrovské výhody, protože negeneruje duplicitní definice Typescript typů pro každý dotaz na API. Navíc, jeho využití zrychlí vývoj, protože již nebude nutné v aplikaci ručně definovat metody pro vyvolání dotazů.

1.4 Optimalizace vykreslování React.js aplikace

Knihovna React.js optimalizuje vykreslování HTML pomocí tzv. virtuálního DOMu, což je objekt reprezentující strukturu komponent [14]. Vývojář nemusí věnovat čas optimalizaci vykreslování na nižší úrovni, musí však ale myslet na chování knihovny a pravidla, dle kterých se překreslování uživatelského rozhraní řídí.

Překreslení komponenty nastane při [14]:

- změně jejího stavu
- změně předaných parametrů
- překreslení rodiče

V případě, kdy je stav komponenty uložen v její rodičovské komponentě a dojde ke změně tohoto stavu, překreslí se obě komponenty. Řešením je stav ukládat přímo k dané komponentě, což eliminuje zbytečné překreslení [14] navíc.

Vykreslování komponent lze analyzovat pomocí rozšíření React Developer Tools pro prohlížeč Chrome. Rozšíření poskytuje nástroj Profiler, který zachycuje posloupnost vykreslení jednotlivých komponent společně s důvodem jejich překreslení.

1.4.1 Využití konstruktu Memo

Konstrukt Memo slouží k uložení vykreslené komponenty do mezipaměti. Překreslení komponenty vynucené překreslením rodičovské komponenty probíhá podstatně rychleji, neboť komponenta se načte z mezipaměti a nedochází ke zbytečným výpočtům [14].

Při využití konstruktu Memo je nutné vzít v potaz dopad na naplnění paměti. Vývojář musí myslet na to, zda-li je výhodnější komponentu ukládat do mezipaměti nebo provést výpočet pro její překreslení.

Příklad Memo komponenty [14]:

```
const ComponentMemo = React.memo(({title}) => (
        <h4>{title}</h4>
);
```

Při změně parametrů (props) Memo komponenty se hodnota v mezipaměti zahodí a znovu přepočte. Tento princip funguje pouze pro primitivní datové typy. Pro referenční datové typy (pole, funkce, objekt) dochází k překreslení komponenty vždy. Tento problém řeší metody useCallback a useMemo.

1.4.2 Hook useCallback

Funkce z knihovny React.js, která slouží pro uložení funkce do mezipaměti. Zamezuje zbytečnému překreslení komponenty, která přijímá parametr referenčního datového typu. UseCallback umožňuje předat pole závislostí, kde změna těchto hodnot způsobí přepočet funkce.

Zápis useCallback [14]:

```
React.useCallback(() => setCount(count + 1), [count]);
```

1.4.3 Hook useMemo

Tato metoda funguje na stejném principu jako useCallback. Místo funkce ale ukládá do mezipaměti přímo vypočtenou hodnotu. Je vhodný k zapamatování instančních typů jako je například objekt nebo pole [14]. Využívá se v případě, kdy je potřeba zamezit zbytečným výpočtům či překreslením podřazených komponent.

Zápis useMemo [14]:

```
const result = React.useMemo(() => {
  return calculate(data);
}, [data]);
```

1.4.4 React.Lazy - dodatečné načítání komponent

Při psaní React.js aplikace je nutné efektivně rozdělovat kód do jednotlivých komponent, aby uživatel byl nucen načíst jen komponenty nezbytné pro vykreslení uživatelského rozhraní. V některých však komponenta musí být složena z velkého počtu jiných komponent, což může zpomalit její načtení. Pro tento problém je určen konstrukt React.Lazy, který zajistí, že se obsah komponenty načte, pouze pokud má dojít k jejímu vykreslení [14].

React.Lazy import [14]

```
const Component = React.lazy(() =>
    import("./components/Component")
);
```

Využití komponenty v kódu [14]:

```
<React.Suspense fallback={<p>Loading …</p>}>
  <Route path="/" exact>
    <Component />
  </Route>
</React.Suspense>
```

1.5 Domain Driven Design (DDD)

Pro Backend projekt je třeba využít návrhu, který zajistí rozšiřitelnost a jasně definovanou logiku. Správný návrh znemožní vývojáři uvést objekty do nevalidního stavu aniž by změnil jádro aplikace. Jednotlivé logické části aplikace musí být definovány vždy na jediném místě, aby byla zajištěna rozšiřitelnost aplikace.

Pro řešení tohoto problému existuje softwarový návrh Domain Driven Design (ve zkratce DDD), který vychází ze stejnojmenné knihy od autora jménem Eric Evans [15]. Autor v knize popisuje zásady, které pomáhají snížit komplexnost aplikace, zvýšit její rozšiřitelnost a poskytují jejím vývojářům lepší způsob vzájemné komunikace [15].

1.5.1 Doménový model

Reprezentuje funkcionalitu aplikace, vazby mezi objekty a logická pravidla pro určitou část aplikace (doménu) [15]. Z modelu musí být jasné, jak aplikace řeší danou problematiku. Každá část modelu musí mít jasně daný název, který následně členové týmu používají během vzájemné komunikace [15].

V případě, kdy nastane změna v přístupu k problematice domény, musí být model aktualizován a s novým přístupem se musí seznámit všichni zúčastnění. Implementace projektu musí vždy odpovídat aktuálnímu stavu doménového modelu, aby nedocházelo k porušení pravidel [15].

1.5.2 Společný jazyk

Vychází z doménového modelu a předpokladu, že doménový model chápe každý člen z týmu. Jasně definuje slovní zásobu a význam jednotlivých slov, čímž je zajištěna efektivní komunikace bez zbytečných nedorozumění mezi členy týmu [15]. Při implementaci aplikace se musí vývojář řídit společným jazykem, např. pojmenovávat jednotlivé třídy dle názvů definovaných v doménovém modelu.

1.5.3 Ohraničený kontext (Bounded Context)

Představuje hranice mezi částmi domény, kde se mění význam objektů. Problematika domény je obvykle rozdělena do dílčích částí, tzv. subdomén, přičemž každá subdoména může vycházet z odlišného modelu a tedy i využívat jiný společný jazyk [15].

Příkladem může být subdoména obchodního oddělení a technické podpory, kde obě subdomény řeší problematiku zákazníka, avšak pro každou z nich má zákazník jiný význam. Obchodní oddělení řeší míru konverze a profitu, podpora naopak řeší, zda-li byly veškeré dotazy zákazníkovi zodpovězeny, pro své fungování nepotřebuje vědět, kolik zákazník firmě přinesl peněz.

1.5.4 Entity a eliminace anemického modelu

Název vychází z ang. slova "Identity". Jedná se o objekt, který je rozeznatelný dle svého unikátního identifikátoru [16]. V aplikaci musí být jasně definovaná logika spojená se změnou stavu entit. Nesmí dojít k porušení pravidel z doménového modelu.

Opakem tohoto přístupu je anemický model, který umožňuje stav entity měnit bez validace. Pro implementaci anemického modelu typicky vznikají pomocné třídy pro práci s entitami, které implementují validaci externě [17]. Pokud vývojář pro manipulaci s entitou nevyužije tyto pomocné třídy, může uvést entitu do nevalidního stavu, což porušuje zásady DDD.

Obrázek 1.2 zachycuje podobu správně implementované entity. Konkrétně se jedná o pronajímatelnou položku, která může být pronajata pouze jednomu uživateli.

```
public class RentableObject : Entity
{
    public string RenterId { get; private set; }

    public void Rent(string renterId)
    {
        if(RenterId is not null)
            throw new AlreadyRentedException(Id);

        RenterId = renterId;
    }

    public void CloseRent()
    {
        RenterId = null;
    }
}
```

Obrázek 1.2: Implementace entity

1.5.5 Hodnotové objekty (Value Objects)

Objekty, které jsou rozeznatelné dle jejich stavu. Hodnotové objekty jsou si navzájem rovny, pokud všechny jejich vlastnosti mají stejnou hodnotu [18]. Příkladem je adresa, kdy 2 adresy jsou totožné, pokud představují stejné místo.

Dle pravidel DDD je stav hodnotových objektů neměnný a jediným způsobem, jak upravit takový objekt, je vytvořit novou instanci se změněnou podobou původních dat. Tím je zajištěno, že změna stavu objektu nezpůsobí nežádoucí chování.

Na následujícím příkladu je princip hodnotových objektů předveden pomocí využití objektu typu DateTime, jehož stav nelze měnit a jeho metody vždy vrací novou instanci objektu.

Využití hodnotového objektu DateTime:

```
// Aktuální čas
var now = DateTime.Now;

// Vyvolání AddDays zanechá stejnou hodnotu now.
var tomorrow = now.AddDays(1);
```

Pokud s vytvořením hodnotového objektu souvisí validace vstupních parametrů, měla by být umístěna do tzv. factory metody. V takovém případě má třída mít neveřejný konstruktor a statickou veřejnou metodu pro vytvoření její instance. Na obrázku 1.3 je příklad hodnotového objektu představující časový interval, kde konec intervalu musí následovat počátek.

```
public class DateInterval
{
    private DateInterval(DateTime begin, DateTime end)
    {
        Begin = begin;
        End = end;
    }

    public DateTime Begin { get; private set; }

    public DateTime End { get; private set; }

    public static DateInterval Create(DateTime begin, DateTime end)
    {
        if(begin > end)
            throw new ArgumentOutOfRangeException("End must follow after the beggining.");

        return new DateInterval(begin, end);
    }
}
```

Obrázek 1.3: Hodnotový objekt s validací

1.5.6 Agregát (aggregate route)

Návrhový vzor popisující kolekci souvisejících objektů, kde jeden z nich je tzv. kořen agregátu (aggregate root). S objekty agregátu lze manipulovat pouze skrze tento kořen. Pro agregát platí striktní pravidla definovaná doménovým modelem a všechny objekty, které zastřešuje, musí být součástí stejného ohraničeného kontextu [19].

Kořen agregátu zajišťuje konzistentní stav všech objektů, které sdružuje. Pro upravení objektu z agregátu je nutné načíst celý kořen agregátu a využít jeho metody pro změnu stavu. Uložení změn (např. do databáze) musí proběhnout v rámci jediné transakce, přičemž součástí této transakce nesmí být změna jiného agregátu [19].

Agregát by měl sdružovat minimální možný počet objektů. Čím více objektů sdružuje, tím déle trvá jeho načtení i uložení, což může způsobovat problémy s konkurencí, kdy více vláken mění stav stejného agregátu, který následně uloží, aniž by brala v potaz změny provedené ostatními vlákny [20].

V některých případech však problémům s konkurencí nelze předcházet. Je tedy nutné tyto problémy detekovat a vyřešit. Příkladem řešení jsou např. zámky (pessimistic concurrency), které pozdrží operaci vlákna, dokud ostatní vlákna nedokončí operaci. Tento přístup má negativní vliv na výkon aplikace, protože předpokládá nejhorší možný stav, kdy dochází k problému s konkurencí vždy [20].

Alternativou je optimistické řízení (optimistic concurrency), kdy kořen agregátu nese informaci o jeho verzi a uložení do databáze proběhne pouze pokud je dohledán záznam se stejnou verzí [20]. V opačném případě je uživateli prezentována výjimka.

1.5.7 Služby

Pokud v programu existuje proces, který nelze dle pravidel DDD vložit přímo do entity nebo hodnotového objektu, je zakomponován do služby. V závislosti na povaze operace se služby dělí na doménové nebo aplikační [15].

Doménové služby poskytují operace, které obstarávají logiku vyplývající z doménového modelu. Jedná se např. o operace, které pracují s dvěma agregáty. Dle pravidel DDD nesmí agregát měnit stav jiného agregátu, proto takové chování musí být vloženo do služby [15].

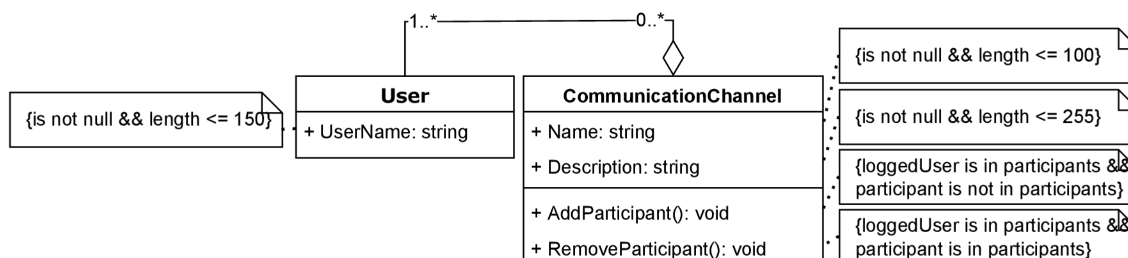
Aplikační služby obsahují logiku aplikace, která je nutná pro implementaci, ale není stanovena doménovým modelem [15]. Vnitřně pracují s entitami a doménovými službami. Entity však nesmí přijímat ani vracet [15]. Implementace aplikačních služeb je typicky spjata s danou platformou, např. mobilní aplikace řeší určitou problematiku, kterou webová aplikace řešit nemusí.

2 Návrh doménového modelu

Tato kapitola se zabývá návrhem doménového modelu aplikace. Z návrhu v této kapitole vyplývají názvy objektů, jejich atributy, pravidla a vzájemné provázání.

2.1 Uživatelé a komunikační kanály

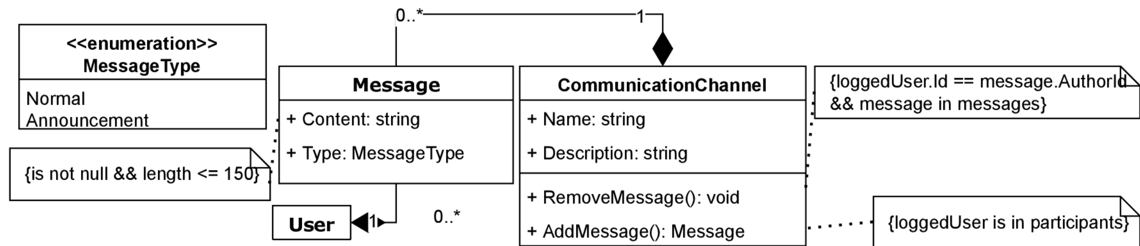
V aplikaci musí existovat uživatelé, kteří se budou moct přidávat do různých komunikačních kanálů. Komunikační kanál může být založen kterýmkoliv uživatelem. Upraven může být pouze autorem, tedy uživatelem, co jej založil. Výjimkou je správa účastníků, uživatelů zařazených do komunikačního kanálu, ty může spravovat každý z účastníků.



Obrázek 2.1: Uživatelé a komunikační kanály

2.2 Zasílání zpráv v komunikačním kanálu

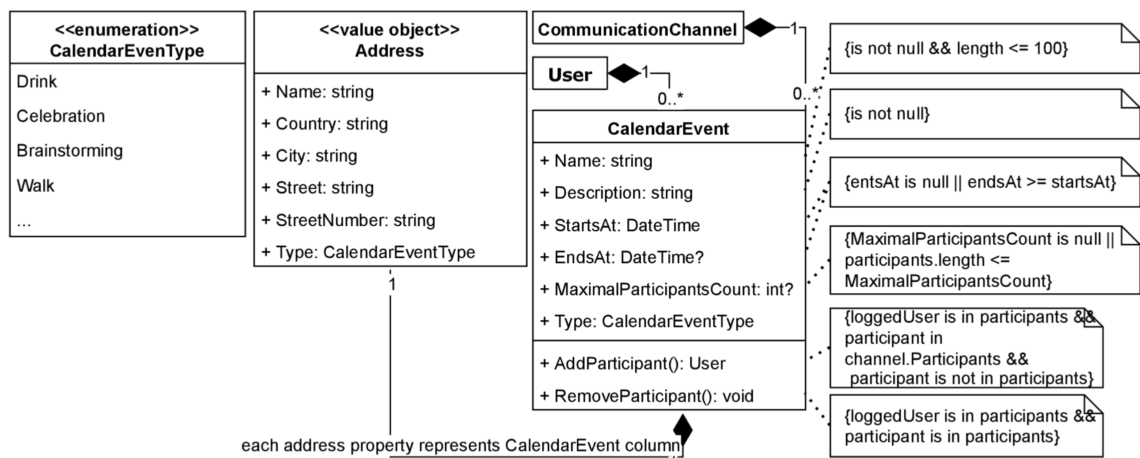
V rámci komunikačního kanálu mohou účastníci zasílat zprávy. Zprávu může číst kterýkoliv z účastníků, avšak pouze autor zprávy ji může upravit nebo smazat.



Obrázek 2.2: Zprávy v komunikačním kanálu

2.3 Kalendářní události

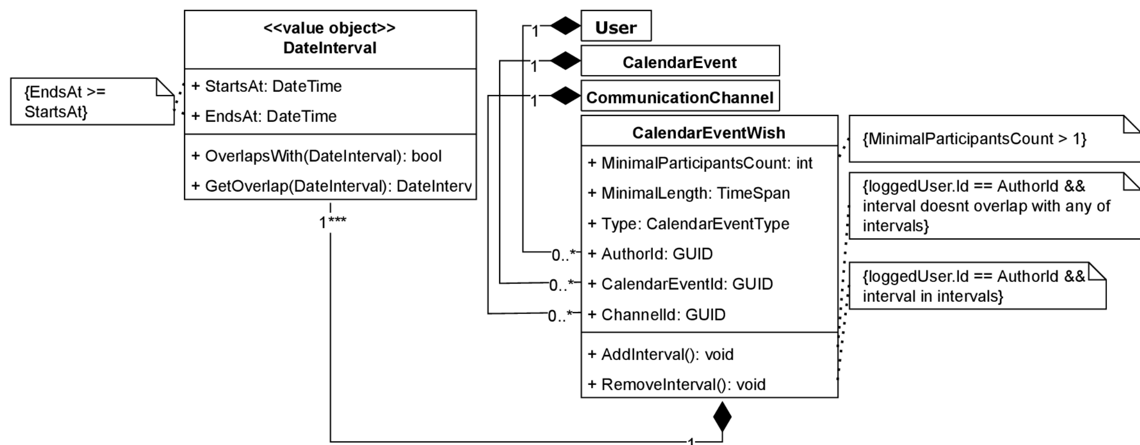
Každý uživatel má možnost vytvářet kalendářní události. K vytvoření události musí uživatel mimo jiné specifikovat počátek a typ události. Událost může být uspořádána na určité adrese. Každá událost musí mít určen komunikační kanál, jehož účastníci se mohou do události zapojit. V případě, kdy uživatel vyplní i maximální počet účastníků, událost zamezí uživatelům se zapojit, pokud by to znamenalo překročení kapacity. Událost smí zrušit pouze její autor.



Obrázek 2.3: Kalendářní událost

2.4 Přání o vytvoření události

Události mohou vznikat také automaticky na základě přání. Každý uživatel může vytvořit přání o vytvoření události, kde určí skupinu uživatelů se kterými chce událost pořádat. Dále musí vyplnit volné časové bloky, typ události, minimální délku a minimální počet účastníků. Jednotlivé časové bloky se nesmí navzájem protínat. Přání smí být zobrazeno a upraveno pouze jeho autorem. Slouží jako vstup pro algoritmus, který uživatelům automaticky vytváří události.



Obrázek 2.4: Přání o vytvoření události

3 Implementace API

K vytvoření API byly použity technologie od firmy Microsoft, dodatečné balíčky či open source knihovny. Výsledné API je typu GraphQL, tedy poskytuje klientovi jediný cílový bod, na který může posílat dotazy. Aplikace je dle cibulové architektury rozdělena do 4 projektů. Projekt částečně vychází z šablony CleanArchitecture od vývojáře jménem Jason Taylor [2].

Seznam použitých technologií:

- ASP .NET 5 (webový framework)
- MS SQL (relační databáze)
- Entity Framework (práce s databází)
- MediatR (CQRS - rozdělení příkazů a dotazů)
- graphql-dotnet (framework pro sestavení API endpointu a zpracování dotazů)

Rozdělení aplikace na 4 projekty:

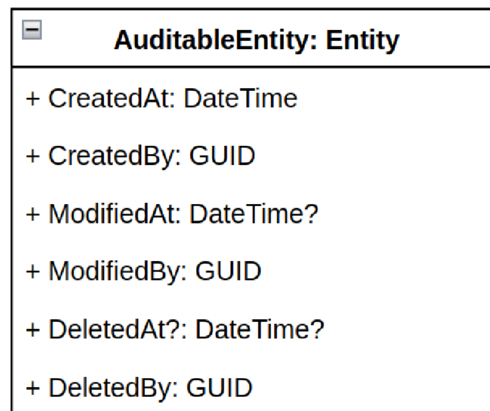
- Chattoo.Domain
  - entity, pravidla, omezení, rozhraní repozitářů
- Chattoo.Application
  - příkazy, validace, rozhraní aplikačních služeb
- Chattoo.Infrastructure
  - implementace repozitářů a aplikačních služeb
  - napojení na databázi včetně konfigurace
- Chattoo.GraphQL
  - zpracování dotazů od klienta, konfigurace aplikace

3.1 Projekt Chattoo.Domain

Vychází striktně z doménového modelu. Obsahuje entity, hodnotové objekty, doménové služby a rozhraní služeb, které jsou implementovány v projektu Chattoo.Infrastructure.

3.1.1 Entity

Všechny entity v projektu dědí z bázové třídy Entity, která definuje identifikátor entity. Některé entity také dědí z bázové třídy AuditableEntity, která obsahuje vlastnosti určující, kdy a kým byla entita upravena.



Obrázek 3.1: Třída AuditableEntity

Entity jsou navzájem provázány skrze navigační vlastnosti. Díky tomu lze skrze entitu upravit stav jiné entity. Výjimkou jsou ovšem kořenové entity agregátů, na které žádná jiná entita nedosahuje skrze navigační vlastnost. Důvodem jsou zásady pro implementaci dle Domain Driven Design. Veškeré kořenové entity v projektu jsou lehce rozeznatelné, protože implementují rozhraní IAggregateRoot.

Seznam kořenových entit agregátů

- CommunicationChannel - komunikační kanál
- CalendarEvent - kalendářní událost
- CalendarEventWish - přání o vytvoření kalendářní události
- User - uživatel

Vlastnosti entit jsou zpravidla soukromé, tj. není možné je upravit mimo třídu dané entity. Úprava je možná pouze pomocí veřejných metod, které entity poskytují. Tím je docíleno dodržení pravidel a omezení, které jsou definovány doménovým modelem.

Např. entita CalendarEventWish obsahuje soukromou kolekci časových intervalů. Tato kolekce je následně zveřejněna mimo kontext této třídy jako IReadOnlyCollection. Do kolekce lze přidávat pouze skrze metodu AddInterval, která kontroluje, zda-li předaný interval nezasahuje do jednoho z již definovaných intervalů.

3.1.2 Hodnotové objekty

Veškeré hodnotové objekty v projektu dědí z abstraktní bázové třídy ValueObject, která obsahuje přetížení metod GetHashCode, Equals a EqualOperator, pomocí kterých je docíleno ekvivalence 2 různých hodnotových objektů, pokud uchovávají stejná data. Každá třída dědící z ValueObject definuje přetížení metody GetEqualityComponents, která vrací enumerátor vlastností pro vyhodnocení ekvivalence.

V aplikaci jsou definovány 2 hodnotové objekty:

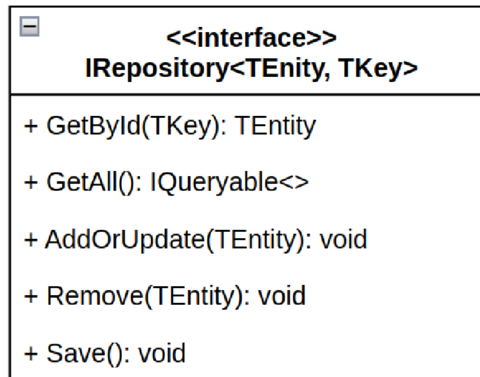
- Address (adresa)
- DateInterval (časový blok od/do)

3.1.3 Repozitáře

Slouží pro načítání/ukládání kořenových entit, přičemž pro každou z nich je v aplikaci definován vlastní repozitář. Uložení změn kořenových entit probíhá v rámci jedné transakce, během které se uloží i změny entit, které jsou kořenovou entitou spravovány. Rozhraní repozitářů jsou definována v projektu Chattoo.Domain a jsou implementována v Chattoo.Infrastructure.

Každý repozitář nabízí jiné funkce. Např. repozitář pro správu komunikačních kanálů může mít metodu GetByUserId pro získání všech komunikačních kanálů určitého uživatele. Tato funkce ale nemá využití v repozitáři pro správu zpráv, kde nikdy není potřeba načíst všechny zprávy uživatele napříč. všemi komunikačními kanály.

Každý repozitář musí ovšem umět stejné základní operace pro čtení a zápis [21]. Proto veškeré repozitáře implementují rozhraní IRepository definující předpis základních operací. Rozhraní přijímá generické parametry, které určují, s jakou entitou repozitář pracuje.



Obrázek 3.2: Společné rozhraní repozitářů

3.2 Projekt Chattoo.Application

Struktura projektu vychází z přístupu CQRS (Command and Query Responsibility Segregation). Dle principů CQRS jsou veškeré operace v aplikační vrstvě rozděleny na čtení a zápis [22].

V projektu je nainstalovaný balíček MediatR, knihovna implementující principy softwarového návrhu Mediator. Tento návrh zajišťuje zpřehlednění interakcí mezi objekty v aplikační vrstvě, protože centralizuje interakci mezi objekty do jednotlivých příkazů [23].

Aplikační vrstva sděluje výsledky operací vnějším vrstvám pouze ve formě jednoduchých DTO objektů, které nemají definované chování. Pokud je potřeba z vnější vrstvy vyvolat nějakou operaci, prostě se zavolá příkaz.

Každá interakce (příkaz) je rozdělena do 3 částí:

- kontrakt
  - DTO se vstupními parametry
- zpracování
  - tj. Handler
  - vykoná příkaz a vrátí výsledek
- validace
  - probíhá před zpracováním příkazu

Ukázka volání příkazu:

```
var command = new AddUserToCalendarEventCommand()
{
      UserId = ctx.GetString("userId"),
      EventId = ctx.GetString("eventId")
};

await mediator.Send(command);
```

Obrázek 3.3: Ukázka příkazu

3.3 Projekt Chattoo.Infrastructure

Slouží pro verzování a napojení na databázi. Pro práci s MSSQL databází využívá balíček Entity Framework Core, což je ORM framework, který zajišťuje, že úprava stavu entity se promítne i do databáze.

V projektu se nachází konfigurace databáze, tedy jednotlivých tabulek a relací mezi nimi, přičemž pro každou entitu je implementováno rozhraní repozitáře z projektu Chattoo.Domain. Konfigurace entit jsou definovány v jednotlivých souborech pomocí Entity Framework Core Fluent API, viz. obrázek 3.4.

```
public class EntityConfiguration : IEntityTypeConfiguration<Entity>
{
    public void Configure(EntityTypeBuilder<Entity> builder)
    {
        builder.HasKey(e => e.Id);

        builder.Property(e => e.Id)
            .ValueGeneratedOnAdd();

        builder.Property(e => e.MessageId)
            .IsRequired();

        builder.Property(e => e.Name)
            .HasMaxLength(200)
            .IsRequired();

        builder.Property(e => e.Content)
            .IsRequired();
    }
}
```

Obrázek 3.4: Konfigurace entity/tabulky

Verzování je uskutečněno pomocí databázových migrací, které jsou generovány příkazem „dotnet ef add-migrations“. Po spuštění příkazu je vygenerován soubor zachycující změny struktury databáze oproti předchozí verzi. Změny lze následně uplatnit pomocí příkazu „dotnet ef update-database“.

3.4 Projekt Chattoo.GraphQL

Spustitelná webová aplikace poskytující API pro klientskou aplikaci. Kromě funkce API zprostředkovává autentizaci uživatele a automatické vytváření událostí na základě preferencí uživatelů.

3.4.1 Autentizace/autorizace uživatele

Pro využití API musí být uživatel autorizován. Autorizace probíhá na základě JWT tokenu, který uživatel získá od autentizačního serveru. Pro autentizaci je využita technologie Identity Server, která je do projektu nainstalována jako Nuget balíček.

Proces autorizace:

1. Klientská aplikace nemá k dispozici autorizační token
2. Uživatel je přesměrován na přihlašovací formulář Identity Server
3. Identity Server ověří správnost zadaných údajů
4. Při úspěšné autentizaci Identity Server předá klientské aplikaci autorizační token
5. Autorizační token je zasílán společně s každým požadavkem na API

Hlavní výhodou využití Identity Server je jednoduchost a rozšiřitelnost. Pomocí jednoduché konfigurace je možné nastavit přihlašování např. přes Facebook nebo Google účet. Navíc, tento autentizační server lze spustit jako samostatnou aplikaci na jiném fyzickém serveru.

3.4.2 Sestavení API

API je typu GraphQL, což znamená, že naslouchá požadavkům skrze jediný cílový bod. Umožňuje také klientovi specifikovat formát odpovědi, např. která pole mají být zastoupena v odpovědi, případně jak mají být transformována.

Pro sestavení API je využit balíček graphql-dotnet, který umožňuje nejen zpracování dotazů, ale i aktivní zasílání informací o změnách např. skrze websocket. Veškeré funkce a chování API jsou definované pomocí tzv. schéma, které je v aplikaci Chattoo.GraphQL zapsáno v souboru GraphQLSchema.cs. Schéma je pro přehlednost rozděleno do více souborů.

Schéma také definuje formát objektů, se kterými API pracuje. Tyto objekty mohou být využity jako vstupní i výstupní. Na obrázku 3.5 je zachycena zjednodušená podoba objektu CommunicationChannelGraphType představující komunikační kanál. Implementace zároveň určuje mapování mezi API objektem a aplikačním DTO.

```
public sealed class CommunicationChannelGraphType : AuditableObjectGraphType<CommunicationChannelDto>
{
    public CommunicationChannelGraphType()
    {
        Name = "CommunicationChannel";

        Field(o => o.Name);
        Field(o => o.Description);
    }
}
```

Obrázek 3.5: API objekt - komunikační kanál

3.5 Automatické vytváření událostí

Každý uživatel může vytvořit přání o vytvoření události. Tato přání určují, kdy a s kým chce uživatel organizovat kalendářní událost s určitými parametry. Server pravidelně kontroluje, jestli nedošlo ke shodě preferencí uživatelů. Pokud nalezne shodu mezi preferencemi uživatelů, splní jim jejich přání, tedy vytvoří jim společnou kalendářní událost s adekvátními parametry. Toto chování zajišťuje aplikační služba, která je volána při každém vytvoření nového přání.

3.5.1 Algoritmus pro nalezení shody

Algoritmus přijímá jako vstup nově vytvořené přání a vyhledává v databázi veškerá přání, která by se s ním mohla shodovat. Následně kontroluje pro všechny smysluplné kombinace nalezených přání, zda-li nedošlo k průniku jejich časových intervalů. Při nalezení průniku algoritmus vrátí aktuální nalezenou kombinaci přání a jejich největší společný časový interval.

Složitost algoritmu v nejhorším případě je přibližně $O(C(N, k) * k * J^2)$, kde:

- N = počet přání, která se mohou shodovat s nově přidaným přáním
- k = průměrný minimálních počet účastníků pro každou kombinaci
- J = průměrný počet časových bloků u jednoho přání

Př. 1: N = 100; k = 2; J = 5
$C(100, 2) * 2 * 5 * 5 = 4950 * 2 * 25 = 247500$

Př. 2: N = 100; k = 50; J = 5
$C(100, 50) * 50 * 5 * 5 = 1.009 * 10^29 * 50 * 25$

Z předchozích příkladů vyplývá, že počet iterací je velmi náchylný na změnu k (minimální počet účastníků). Je tedy nutné omezit max. hodnotu k např. na 5, tím bude jednak zajištěn nižší počet iterací a přání budou splněna dříve, tedy průměrná hodnota N se také sníží. Algoritmus je detailně popsán blokovým diagramem v příloze 6.

4 Implementace uživatelského rozhraní

Klientská aplikace je implementovaná pomocí technologií React.js, TypeScript a Next.js. Jednotlivé komponenty jsou sestaveny pomocí již hotových komponent z knihovny MaterialUI.

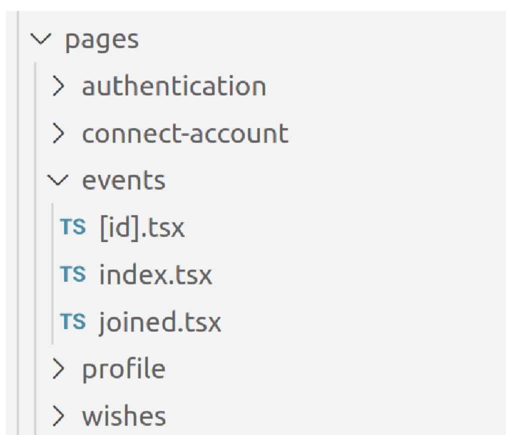
4.1 Framework Next.js

Jedná se o framework, který poskytuje nástroje pro zrychlení a optimalizaci webových stránek napsaných pomocí technologie React.js. Tento nástroj řeší řadu problémů, z nichž nejpodstatnější je počáteční vykreslení webové aplikace [24].

Tradiční React.js aplikace jsou vykreslovány až na straně klienta. Uživatel obdrží od serveru pouze kostru HTML a JavaScript kód, který následně zajistí vykreslení obsahu. Tento přístup však není kompatibilní s vyhledávači jako je např. Google, neboť počáteční obsah stránky neobsahuje dostatek informací, aby stránka mohla být indexována vyhledávačem [24].

Framework Next.js zajišťuje, že k prvotnímu vykreslení dojde přímo na serveru, který následně uživatelovi předá již vykreslenou stránku. O následující překreslování aplikace se již stará klientský počítač [24].

Také obstarává směrování. V projektu je složka pages, která obsahuje veškeré podstránky aplikace. Uvnitř této složky jsou další podsložky, ve kterých jsou umístěny React komponenty představující jednotlivé podstránky. Tj., struktura složky pages reflektuje URL adresy, které uživatel může prohlížet.



Obrázek 4.1: Struktura složky pages

4.2 Komunikace s webovým API

Uživatelské rozhraní vykresluje svůj obsah na základě dat, které získává z webového API. Pro komunikaci s tímto API využívá knihovnu apollo-client, která umožňuje zasílat dotazy ve formátu GraphQL. Klient může v dotazu specifikovat v jakém formátu si přeje obdržet odpověď. Knihovna také podporuje tzv. subscribtions, kdy klientská aplikace žádá API o pravidelné zasílání změn.

GraphQL dotazy se dělí na 3 kategorie [25]:

- query (získání dat)
- mutation (úprava dat)
- subscription (pravidelné informování klienta)

Např. při zobrazení komunikačního kanálu v uživatelském rozhraní se aplikace přihlásí k odběru nových zpráv. Toho je docíleno zavoláním metody vygenerované generátorem graphql-generator. Zprávy jsou zasílány klientské aplikaci skrze websocket. V případě, kdy aplikace obdrží novou zprávu, aktualizuje mezipaměť knihovny apollo-client a prezentuje zprávu uživateli. Zápis graphql dotazu je zachycen na obrázku 4.2, jeho využití je na obrázku 4.3.

```
export const GET_CHANNEL = gql`
    Execute Query
    query GetChannel($id: ID!) {
        communicationChannels {
            get(id: $id) {
                id,
                name,
                description,
                createdAt,
                modifiedAt,
                createdBy,
                deletedBy,
                modifiedBy,
            }
        }
    }
`;
```

Obrázek 4.2: GraphQL query - získání komunikačního kanálu

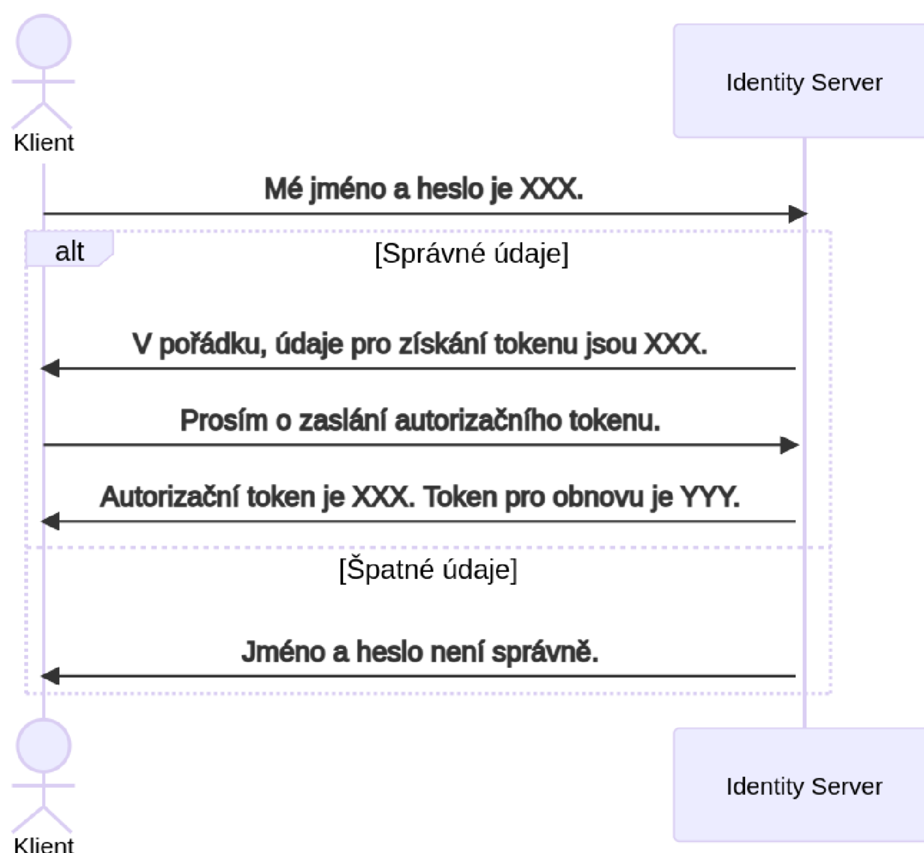
```
const { data: channelQueryData, error: channelQueryError } = useGetChannelQuery({
    variables: {
        id: event?.communicationChannelId
    }
});
```

Obrázek 4.3: Vyvolání GraphQL dotazu - získání komunikačního kanálu

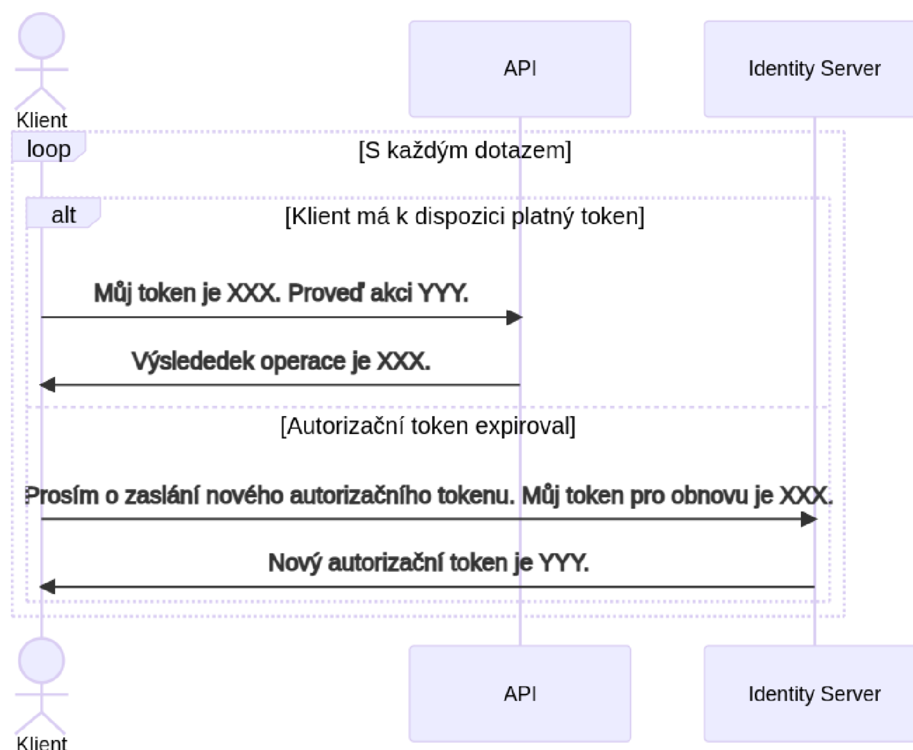
4.3 Autorizace uživatele

Pro plný přístup k funkcím aplikace se musí uživatel nejdříve přihlásit. Učiní tak kliknutím na tlačítko přihlásit na úvodní stránce, které ho následně přesměruje na přihlašovací formulář vygenerovaný technologií Identity Server.

Po úspěšném přihlášení je uživatel přesměrován zpět do uživatelského rozhraní. Součástí přesměrování je i předání informací k získání autorizačního tokenu. Aplikace následně získá autorizační token od serveru, k tomu je využit balíček oidc-client. Tento balíček zároveň obstarává další funkce nutné pro správu autorizačního tokenu, např. jeho obnovení či načtení informací, které jsou v tokenu uloženy. Tímto způsobem lze z tokenu získat např. datum a čas expirace nebo údaje o uživateli.



Obrázek 4.4: Autentizace uživatele - získání autorizačního tokenu



Obrázek 4.5: Komunikace s API a autorizace tokenem

4.4 Struktura klientské aplikace a rozložení komponent

Uživatelské rozhraní je rozděleno do jednotlivých funkcionálních komponent psaných pomocí jazyku TypeScript. Komponenty se navzájem ovlivňují a využívají funkce ostatních komponent, k čemuž ovšem není použita dědičnost či objektově orientované programování.

Stav komponent je většinou uložen do mezipaměti, kterou poskytuje knihovna Apollo Client. V některých případech je ale stav uložen pomocí metody useState z knihovny React.js. Obě varianty zajišťují reaktivitu, kdy změna stavu způsobí překreslení. Výhodou tohoto přístupu je, že v aplikaci nemusí být nainstalovaný dodatečný balíček pro správu stavu, jako např. knihovna Recoil či Redux.

Rozdělení aplikace do komponent (pouze stěžejní komponenty):

- ConnectAccount - přihlášení/registrace uživatele
- Header - odhlášení, přesměrování na jednotlivé podstránky
- CommunicationChannelList - seznam komunikačních kanálů
- CommunicationChannel - detail komunikačního kanálů, psaní zpráv

- CommunicationChannelSettings - nastavení komunikačního kanálu
- CommunicationChannelCreate - vytvoření komunikačního kanálu
- EventDashboard - seznam kalendářních událostí
- EventDetailParticipants - správa uživatelů v kalendářní události
- EventCreateForm - vytvoření kalendářní události
- WishesDashboard - seznam přání
- CreateWishForm - vytvoření přání

Na obrázku 4.6 je zachycena podoba komponenty pro odeslání zprávy. Tato komponenta obsahuje stav textového pole, do kterého uživatel zadává zprávu. Po kliknutí na tlačítko se vyvolá odeslání zprávy ostatním uživatelům.

```
const MessageBox: FC<MessageBoxProps> = ({callback}) => {
    const [text, setText] = useState<string>("");

    const handleInputOnChange = useCallback((ev: any) => {
        setText(ev.target.value);
    }, [setText]);

    const handleOnSubmit = useCallback((ev: any) => {
        ev.preventDefault();
        callback(text || "👍");
        setText("");
    }, [callback, text]);

    return (
        <Box>
            <form onSubmit={handleOnSubmit}>
                <Stack direction="row">
                    <CustomInput placeholder="Zadejte zprávu..." value={text}
                        onChange={handleInputOnChange} full={true} />
                    <IconButton color="primary" onClick={handleOnSubmit}>
                        {text && text.length > 0 ? <SendIcon /> : <ThumbUp />}
                    </IconButton>
                </Stack>
            </form>
        </Box>
    );
}
```

Obrázek 4.6: Příklad komponenty - zaslání zprávy

4.4.1 Optimalizace vykreslování

Pro optimalizaci vykreslení jsou v aplikaci využity příkazy useMemo a useCallback z knihovny React.js. Např. v okně chatu jsou jednotlivé zprávy vykresleny různě dle toho, kým a kdy byly zaslány, přičemž tento výpočet má smysl pouze v případě, kdy uživatel obdrží novou zprávu. Proto je kolekce vykreslených zpráv uložena do mezipaměti pomocí metody useMemo, viz. obrázek 4.7.

```
/** Pole zpráv pro zobrazení (memoized pro zamezení zbytečného výpočítávání) */
const messagesFiltered = useMemo(() => messages &&
    [...messages].reverse().map((msg, i, arr) =>
        renderMessage(msg, arr[i - 1], arr[i + 1])), [messages]);
```

Obrázek 4.7: Optimalizace vykreslování zpráv v chatu

5 Analýza bezpečnosti aplikace

Při implementaci aplikace byl kladen důraz na zabezpečení dat. Veškeré dotazy a příkazy procházejí validací.

Validace dat od uživatele probíhá následujícím způsobem:

1. validace příkazu v klientské aplikaci
   (a) pouze pro formuláře
   (b) pomocí knihovny yup
2. validace knihovnou graphql-dotnet
   (a) dotaz musí odpovídat schématu API
3. validace v aplikační vrstvě
   (a) pomocí zápisu FluentValidation
   (b) kontrola formátu dat
   (c) někdy i kontrola logických omezení dle doménového modelu
4. validace v doménových službách a entitách
   (a) validace dodržení doménových pravidel a omezení

Veškerá funkcionalita aplikace byla otestována skrze uživatelské rozhraní. Uživatel nemá možnost zobrazit nebo upravit data jiného uživatele.

5.1 Validace na straně klienta

V klientské aplikaci dochází k validaci formulářů vyplněných uživatelem. K validaci je využita knihovna yup, která umožňuje vytvořit schéma jednoznačně určující správný formát vstupních dat. Tento způsob validace je využit např. ve formuláři pro vytvoření komunikačního kanálu, kde uživatel musí zadat text o určité délce.

```
export const createCommunicationChannelFormValidationSchema = yup.object().shape({
    name: yup.string().required("Zadejte text.").max(100, "Max. délka názvu je 100."),
    description: yup.string().required("Zadejte text.").max(255, "Max. délka popisku je 255.")
});
```

Obrázek 5.1: Knihovna yup - validační schéma

5.2 Validace před zpracováním dotazu

V momentě, kdy webové API obdrží dotaz od klienta, dojde k validaci vstupních parametrů knihovnou graphql-dotnet. Během této kontroly knihovna vyhodnotí, zda-li dotaz odpovídá schématu. V případě, kdy je formát v pořádku, pokusí se o provedení žádané akce, tedy zpracuje dotaz a vyvolá obslužnou metodu z aplikační vrstvy.

Následně je dotaz validován znovu pomocí knihovny Fluent Validation, která kontroluje nejen správný formát jednotlivých polí, ale i dodržení pravidel definovaných v doménovém modelu. Cílem validace v aplikační vrstvě je zamezit zpracování dotazu, který by doménová vrstva nezpracovala kvůli nedodržení pravidel.

```
public CreateCommunicationChannelCommandValidator(ChannelManager channelManager)
{
    RuleFor(v => v.Name)
        .MaximumLength(100)
            .WithMessage("Název komunikačního kanálu nesmí být delší než 100 znaků.")
        .NotEmpty().WithMessage("Název komunikačního kanálu je nutné vyplnit.");

    RuleFor(v => v.Description)
        .MaximumLength(255)
            .WithMessage("Popis komunikačního kanálu nesmí být delší než 100 znaků.")
        .NotEmpty()
            .WithMessage("Popis komunikačního kanálu je nutné vyplnit.");
}
```

Obrázek 5.2: Aplikační vrstva - Fluent Validation

5.3 Validace v doménové vrstvě

V případě, kdy požadavek zaslaný na API projde validací v aplikační vrstvě, dojde k jeho zpracování. Pro toto zpracování aplikační vrstva využívá metod definovaných v doménové vrstvě. Každá z těchto metod validuje, zda-li pro danou kombinaci vstupních parametrů může operaci provést, pokud ne, vyhodí výjimku. Na obrázku 5.3 je příklad metody pro přidání účastníka do kalendářní události, která kontroluje, že uživatel nebyl již přidán.

```
public void AddParticipant(string userId)
{
    if (HasParticipant(userId))
        throw new DuplicateUserInCalendarEventException(Id, userId);

    var participant = UserToCalendarEvent.Create(userId, Id);

    _participants.Add(participant);
}
```

Obrázek 5.3: Doménová vrstva - validace

6 Prezentace aplikace

Aplikace je responzivní. Lze ji tedy využívat nejen osobním počítači, ale i na tabletu či telefonu. Výslednou podobu uživatelského rozhraní pro různá zařízení naleznete v příloze 6.

Závěr

Práce stručně shrnuje postup vývoje webové aplikace, rešerši, technologie, návrh a implementaci. Při vypracovávání práce byl kladen důraz především na to, aby zvolené postupy nebyly slepou cestou do budoucna a bylo možné aplikaci dále rozšiřovat bez zásadních změn její stávající podoby.

Výsledkem práce je funkční SPA webová aplikace. Uživatelé se mohou registrovat a zapojit se do skupin, kde mezi sebou mohou sdílet textové zprávy v reálném čase. V rámci zmíněných skupin mohou uživatelé organizovat kalendářní události různých typů, přičemž podle těchto typů lze následně události filtrovat. Každý uživatel si může nastavit volné časové bloky a určit si tak, kdy má čas na určitý typ události, přičemž musí určit i skupinu uživatelů, kteří se s ním mohou události účastnit. Úloha na serveru pravidelně prochází preference uživatelů a pokud mezi nimi nalezne shodu, automaticky vytvoří kalendářní událost, do které uživatele zařadí. Aplikaci lze využívat na počítači i na telefonu.

Front-end stránka aplikace byla naprogramována pomocí React.js v kombinaci s knihovnou MaterialUI. Knihovna MaterialUI se velmi osvědčila, značně urychlila vývoj klientské části aplikace a umožnila vytvořit jednotný design aplikace.

Webové API bylo postaveno na platformě ASP .NET 5 a využívá mnoho knihoven, z nichž stěžejními jsou EntityFramework pro persistenci dat a graphql-dotnet pro obsluhu API dotazů. Za zmínku také stojí, že struktura Backend API je inspirována ověřenými metodikami Clean Architecture a Domain Driven Design, díky čemuž výsledný kód je mnohem přehlednější. Využití knihovny EntityFramework umožnilo definovat strukturu databáze přímo pomocí entit psaných v jazyce C, díky čemuž nebylo nutné věnovat moc času konfiguraci databáze.

Definitivně největší chybou v návrhu aplikace byla volba technologie GraphQL. Přesto, že technologie je populární a má své opodstatnění na trhu, její využití pro tuto práci se nevyplatilo. Implementace API byla časově podstatně náročnější než by byla např. implementace REST API. Výsledný React.js klient navíc nevyužívá výhod GraphQL API natolik, aby tím bylo využití GraphQL opodstatněné.

Aplikace splňuje veškeré požadavky dané zadáním a výstup této práce je tedy pro mě uspokojivý. Jedná se o jednoduchou aplikaci, která má však ještě potenciál pro rozšíření do budoucna. V případě navazujícího vývoje by mohlo být zajímavé např. implementovat list přátel, uživatelské role, přezdívky uživatelů a možnost odeslat přílohu jako součást zprávy v chatu. Některé zmíněné funkce jsou již částečně připraveny.

Použitá literatura

[1] PFOHL, Bruno. *Návrh a realizace modulární sociální platformy* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://drive.google.com/file/d/1TY04mrrvQpSsAFI2UeMpwxcos62rAoG-/view?usp=sharing.

[2] TAYLOR, Jason. *Clean Architecture with .NET Core: Getting Started* [online]. 2020 [cit. 2022-03-02]. Dostupné z: https://jasontaylor.dev/clean-architecture-getting-started/.

[3] META PLATFORMS, Inc. *React - a JavaScript library for building user interfaces* [online]. 2022 [cit. 2022-03-02]. Dostupné z: https://reactjs.org/.

[4] GUPTA, Mayank. *4 Ways to add Styles to React Component* [online]. 2020 [cit. 2022-03-02]. Dostupné z: https://medium.com/technofunnel/4-ways-to-add-styles-to-react-component-37c2a2034e3e.

[5] AMBERJ. *5 Delightful Things about Material-UI* [online]. 2019 [cit. 2022-03-07]. Dostupné z: https://dev.to/amberjones/5-delightful-things-about-material-ui-5402.

[6] *Migration from v4 to v5* [online]. 2022 [cit. 2022-03-07]. Dostupné z: https://mui.com/material-ui/guides/migration-v4/#main-content.

[7] KHOSRAVI, Kasra. *Styled-components vs. Emotion for handling CSS* [online]. 2021 [cit. 2022-03-07]. Dostupné z: https://blog.logrocket.com/styled-components-vs-emotion-for-handling-css/.

[8] *Pricing - Start using MUI for free!* [Online]. 2022 [cit. 2022-03-13]. Dostupné z: https://mui.com/pricing/.

[9] JAHODA, Bohumil. *Tailwind CSS* [online]. 2021 [cit. 2022-03-13]. Dostupné z: https://jecas.cz/tailwind-css.

[10] *Optimizing for Production* [online]. 2022 [cit. 2022-03-13]. Dostupné z: https://tailwindcss.com/docs/optimizing-for-production.

[11] *Buy once, use it forever* [online]. 2022 [cit. 2022-03-19]. Dostupné z: https://tailwindui.com/pricing.

[12] *Guide: React and GraphQL* [online]. 2022 [cit. 2022-03-19]. Dostupné z: https://www.graphql-code-generator.com/docs/guides/react.

[13] *Installing Codegen* [online]. 2022 [cit. 2022-03-19]. Dostupné z: https://www.graphql-code-generator.com/docs/getting-started/installation.

[14] MOJEED, Ibadehin. *Optimizing performance in a React application* [online]. 2021 [cit. 2022-03-25]. Dostupné z: https://blog.logrocket.com/optimizing-performance-react-application/.

[15] EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[16] FOWLER, Martin. *EvansClassification* [online]. 2005 [cit. 2022-03-25]. Dostupné z: https://martinfowler.com/bliki/EvansClassification.html.

[17] FOWLER, Martin. *AnemicDomainModel* [online]. 2003 [cit. 2022-04-01]. Dostupné z: https://martinfowler.com/bliki/AnemicDomainModel.html.

[18] FOWLER, Martin. *ValueObject* [online]. 2016 [cit. 2022-04-01]. Dostupné z: https://martinfowler.com/bliki/ValueObject.html.

[19] FOWLER, Martin. $DDD_{Aggregate}$ [online]. 2016 [cit. 2022-04-01]. Dostupné z: https://martinfowler.com/bliki/DDD_Aggregate.html.

[20] GRZYBEK, Kamil. *Handling concurrency – Aggregate Pattern and EF Core* [online]. 2020 [cit. 2022-04-11]. Dostupné z: http://www.kamilgrzybek.com/design/handling-concurrency-aggregate-pattern-and-ef-core/.

[21] DYKSTRA, Tom. *Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)* [online]. 2021 [cit. 2022-04-11]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application.

[22] WILLIAMS, Jonathan. *CQRS using C and MediatR* [online]. 2020 [cit. 2022-04-11]. Dostupné z: https://www.youtube.com/watch?v=mdzEKGlH0_Q.

[23] BANANAS, Derek. *Mediator Design Pattern* [online]. 2012 [cit. 2022-04-11]. Dostupné z: https://www.youtube.com/watch?v=8DxIpdKd41A.

[24] FIRESHIP. *Next.js in 100 Seconds // Plus Full Beginner's Tutorial* [online]. 2021 [cit. 2022-05-08]. Dostupné z: https://www.youtube.com/watch?v=Sklc_fQBmcs.

[25] RINALDI, Brian. *How to Write GraphQL Queries* [online]. 2020 [cit. 2022-04-14]. Dostupné z: https://stepzen.com/blog/how-to-write-graphql-queries.

[26] HALVORSEN, H.-P. (n.d.) *Web Programming ASP.NET Core*. 2021. Dostupné také z: https://www.halvorsen.blog/documents/programming/csharp/textbook/aspnet/Web%20Programming%20-%20ASP.NET%20Core.pdf.

[27] WADEPICKETT. *ASP.NET documentation*. 2022. Dostupné také z: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0.

[28] AGUILAR, J. (n.d.) *Professional SignalR Programming in Microsoft ASP.NET*. 2014. Dostupné také z: https://ptgmedia.pearsoncmg.com/images/9780735683884/samplepages/9780735683884.pdf.

[29] BLUMZON, Christopher a Adrian-Tudor PĂNESCU. Data Storage. In: 2019, sv. 257. ISBN 978-3-030-33655-4. Dostupné z DOI: 10.1007/164_2019_288.

[30] HUGHES, John. *7 Best React UI Framework and Component Libraries* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://themeisle.com/blog/best-react-ui-framework/.

Seznam obrázků

1.1 Clean architecture 12
1.2 Implementace entity 20
1.3 Hodnotový objekt s validací 21

2.1 Uživatelé a komunikační kanály 23
2.2 Zprávy v komunikačním kanálu 24
2.3 Kalendářní událost 24
2.4 Přání o vytvoření události 25

3.1 Třída AuditableEntity 27
3.2 Společné rozhraní repozitářů 29
3.3 Ukázka příkazu 30
3.4 Konfigurace entity/tabulky 30
3.5 API objekt - komunikační kanál 32

4.1 Struktura složky pages 34
4.2 GraphQL query - získání komunikačního kanálu 35
4.3 Vyvolání GraphQL dotazu - získání komunikačního kanálu 36
4.4 Autentizace uživatele - získání autorizačního tokenu 36
4.5 Komunikace s API a autorizace tokenem 37
4.6 Příklad komponenty - zaslání zprávy 38
4.7 Optimalizace vykreslování zpráv v chatu 39

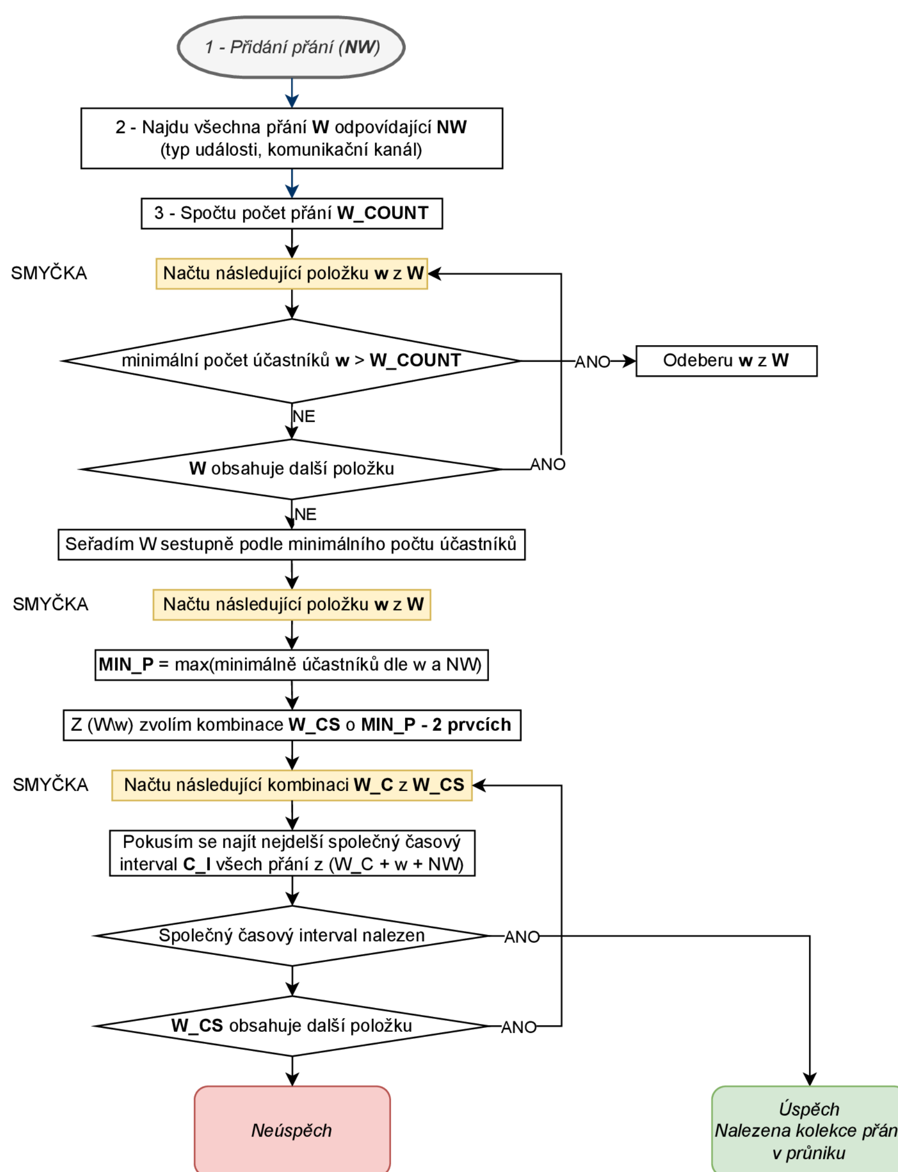
5.1 Knihovna yup - validační schéma 41
5.2 Aplikační vrstva - Fluent Validation 41
5.3 Doménová vrstva - validace 42

Seznam příloh

Příloha - algoritmus pro nalezení shody mezi přáními **50**

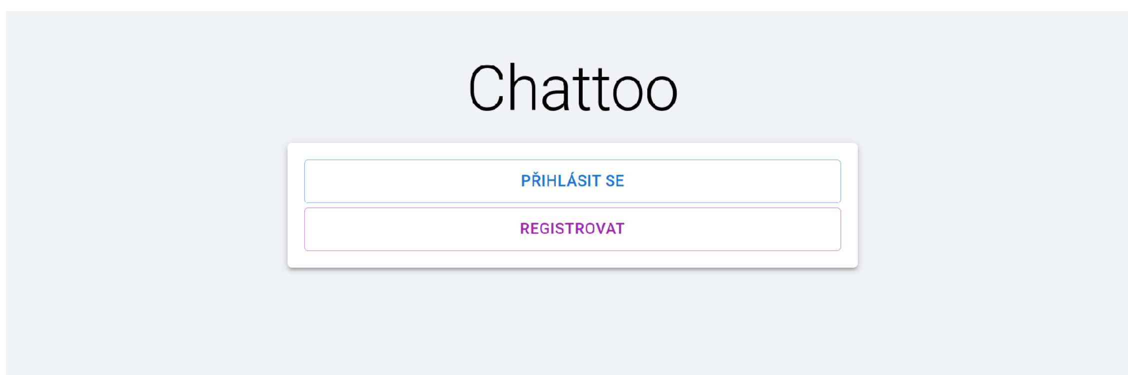
Příloha - prezentace aplikace **51**

Příloha - algoritmus pro nalezení shody mezi přáními

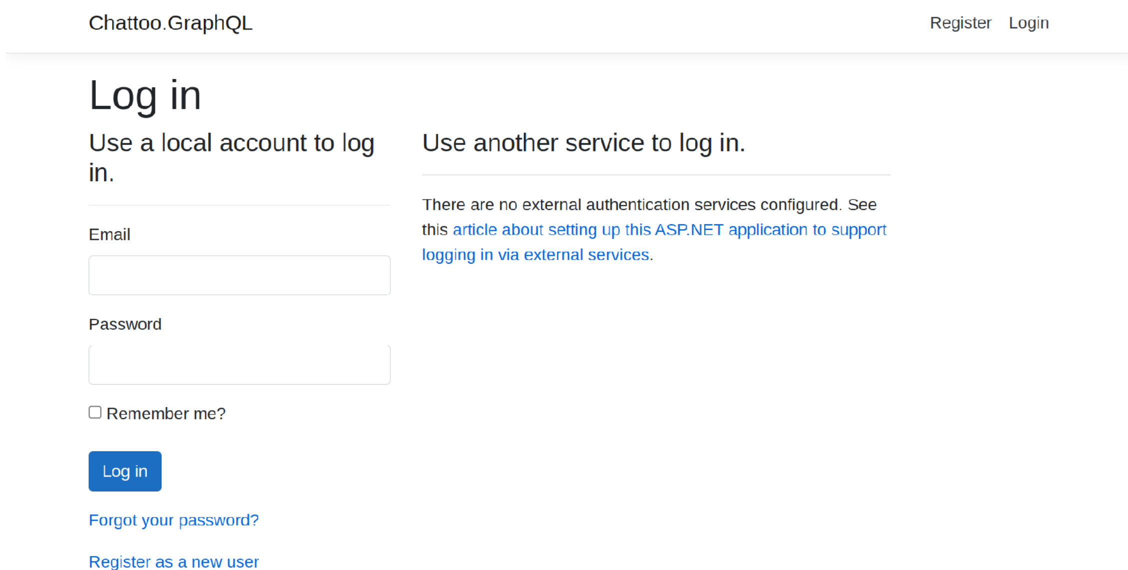


Obrázek 1: algoritmus pro nalezení shody mezi přáními

Příloha - prezentace aplikace



Obrázek 2: Prezentace aplikace - přihlášení/registrace



Obrázek 3: Prezentace aplikace - přihlašovací formulář

Chattoo.GraphQL Register Login

Register

Create a new account.

Email

Password

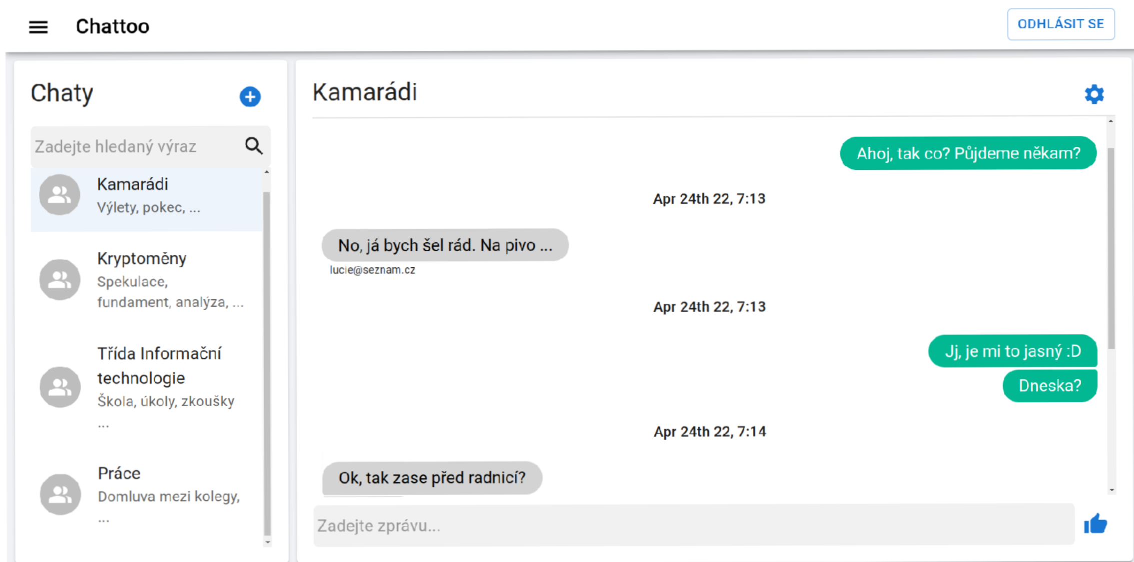
Confirm password

Register

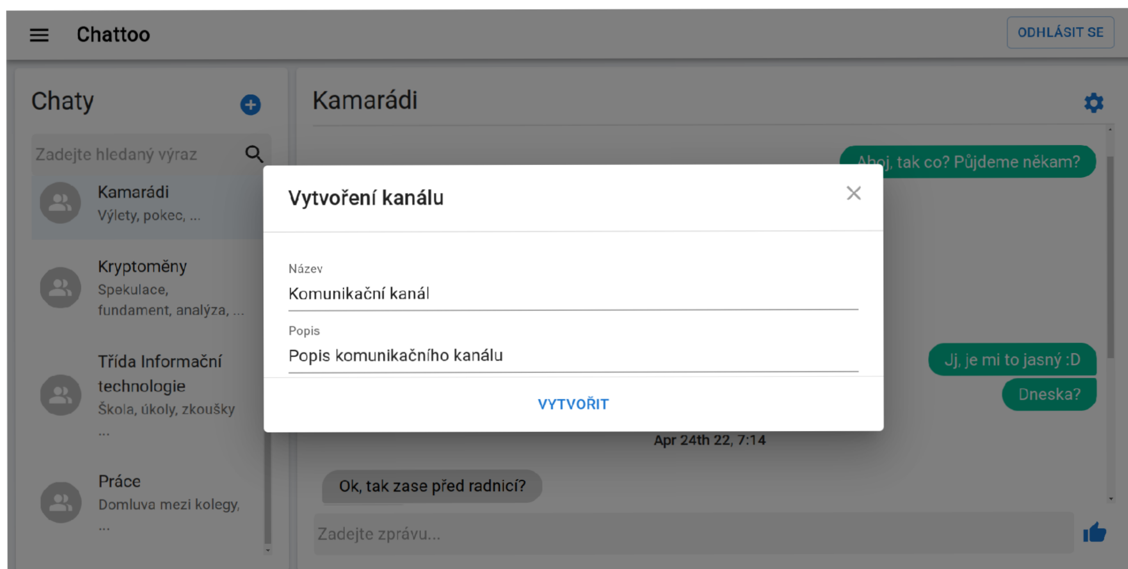
Use another service to register.

There are no external authentication services configured. See this article about setting up this ASP.NET application to support logging in via external services.

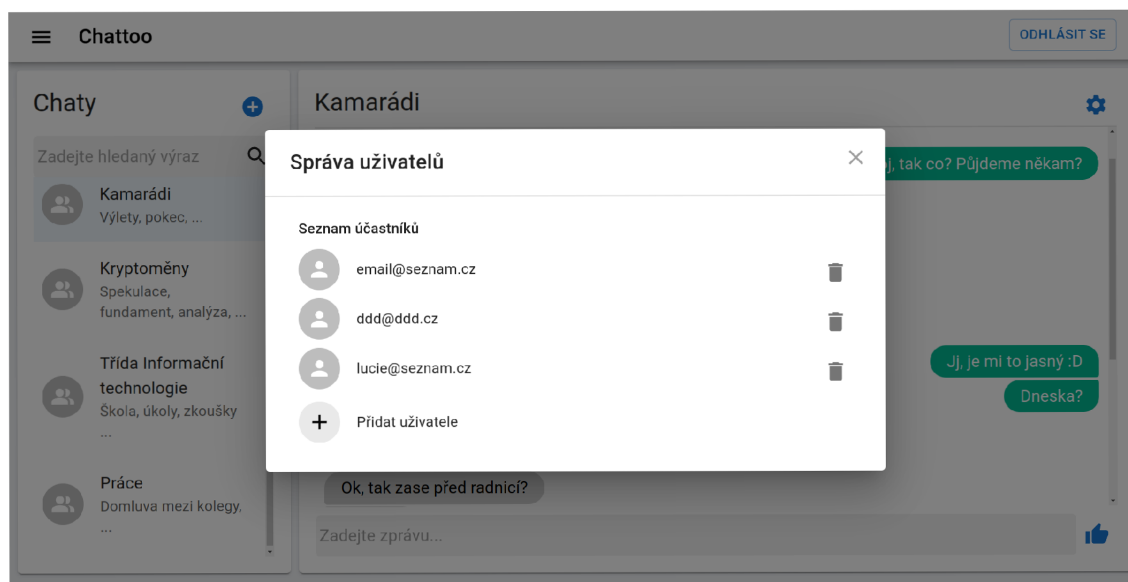
Obrázek 4: Prezentace aplikace - registrační formulář



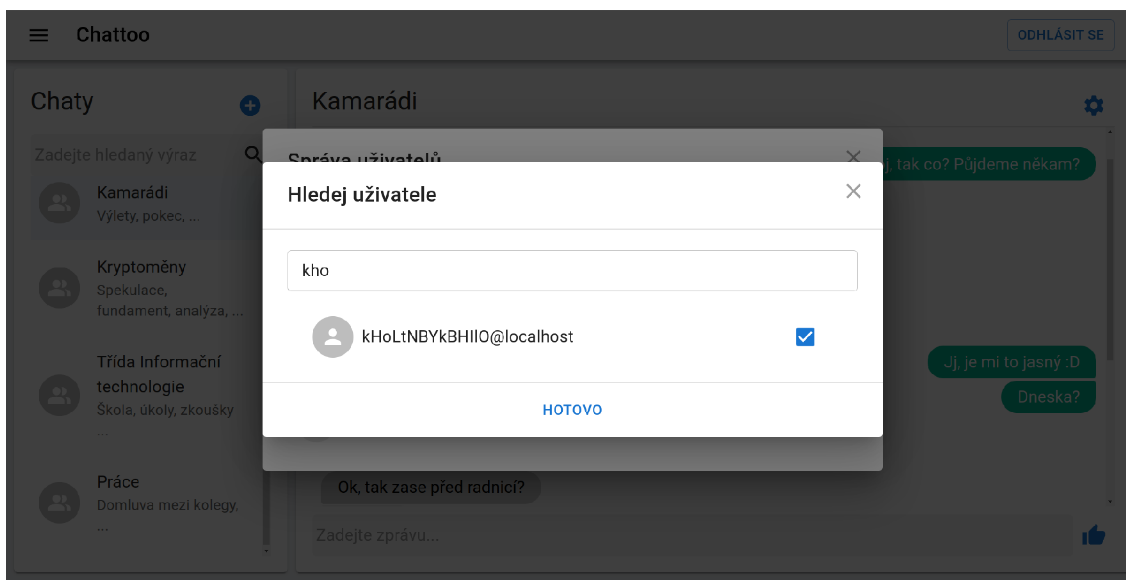
Obrázek 5: Prezentace aplikace - chat



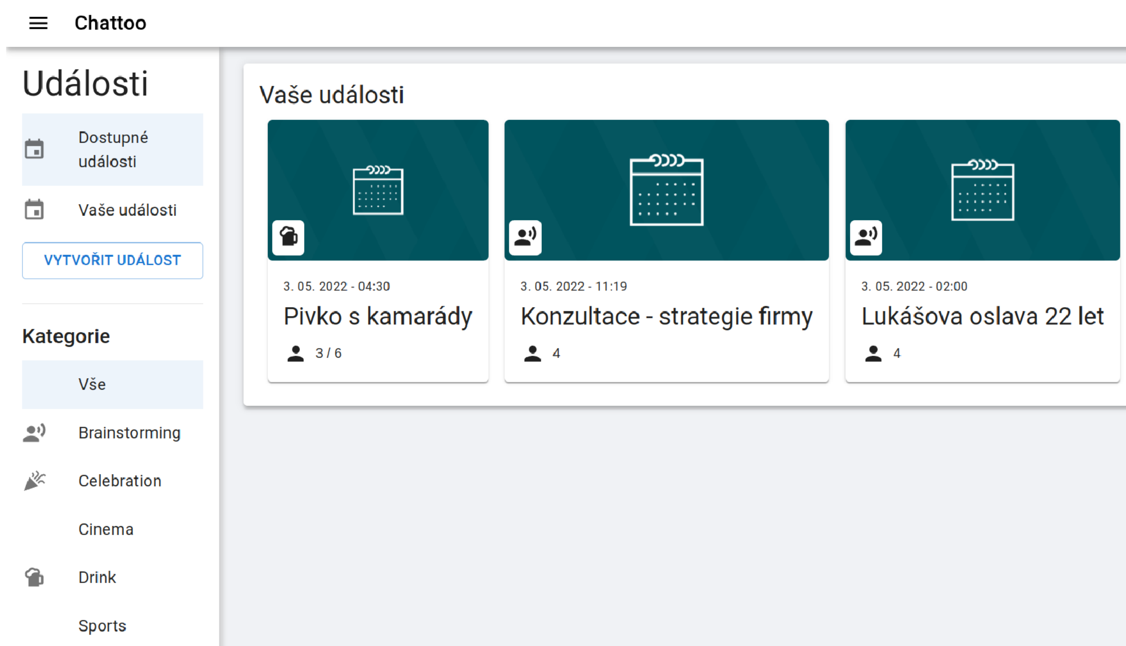
Obrázek 6: Prezentace aplikace - vytvoření komunikačního kanálu



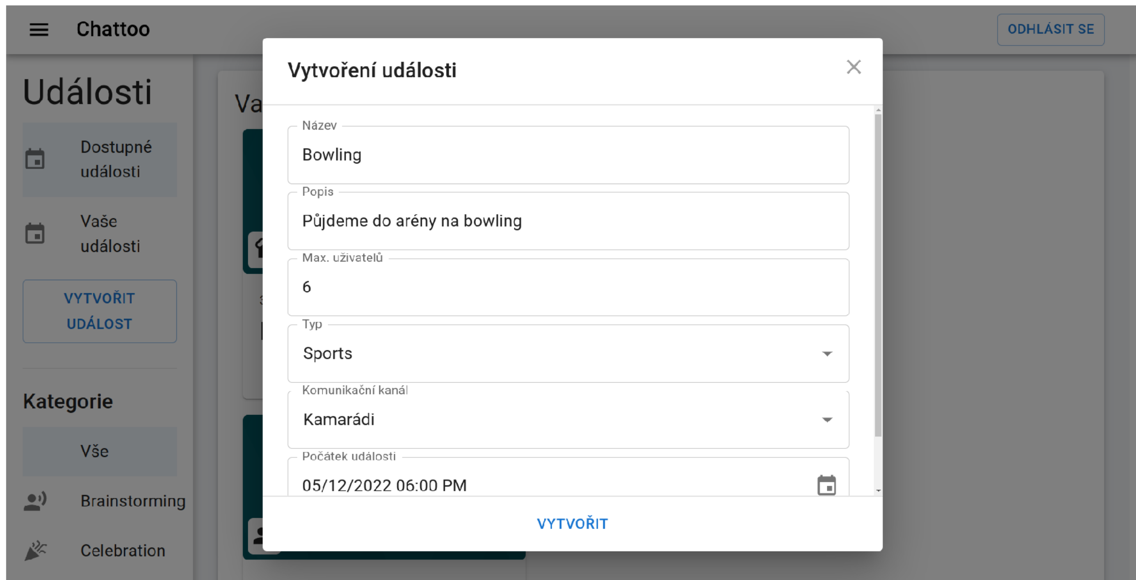
Obrázek 7: Prezentace aplikace - správa účastníků komunikačního kanálu



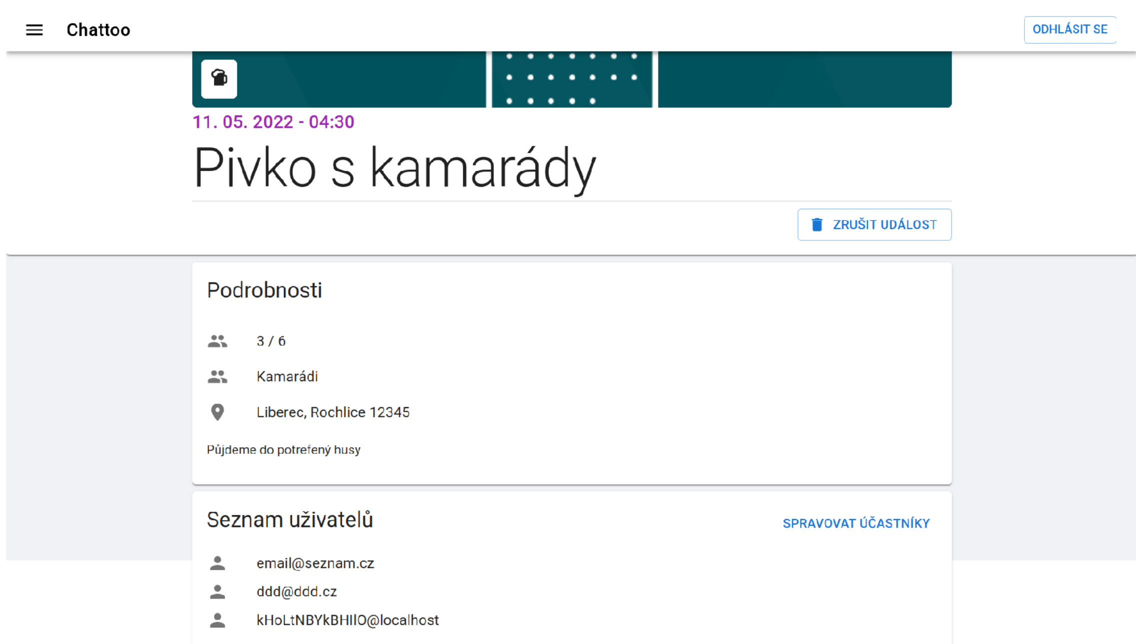
Obrázek 8: Prezentace aplikace - přidání uživatele do komunikačního kanálu



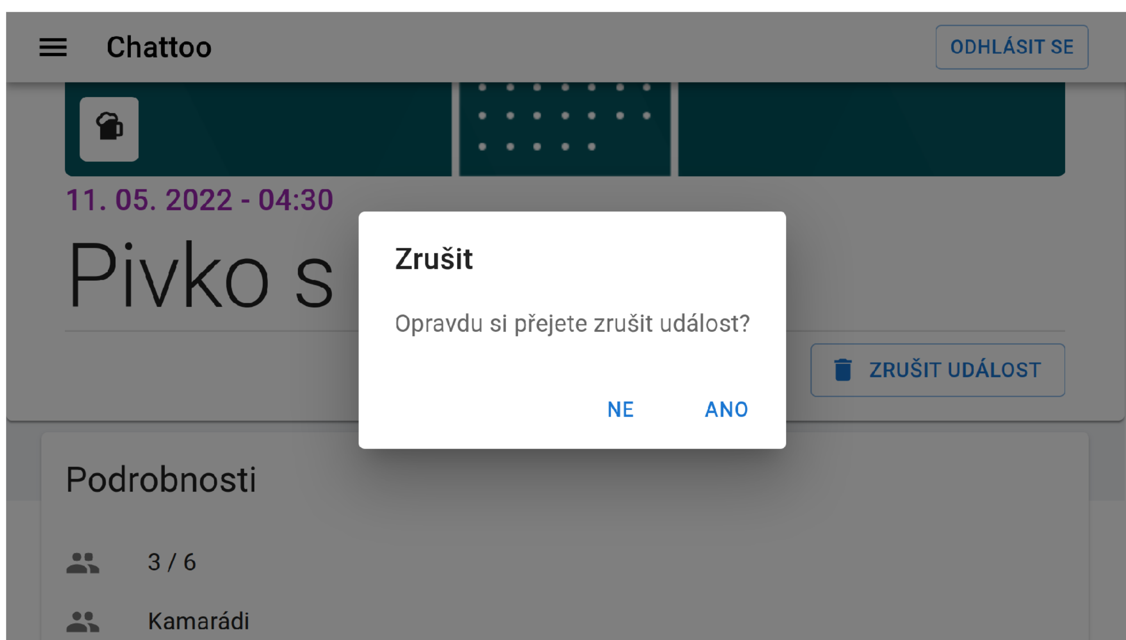
Obrázek 9: Prezentace aplikace - panel s kalendářními událostmi



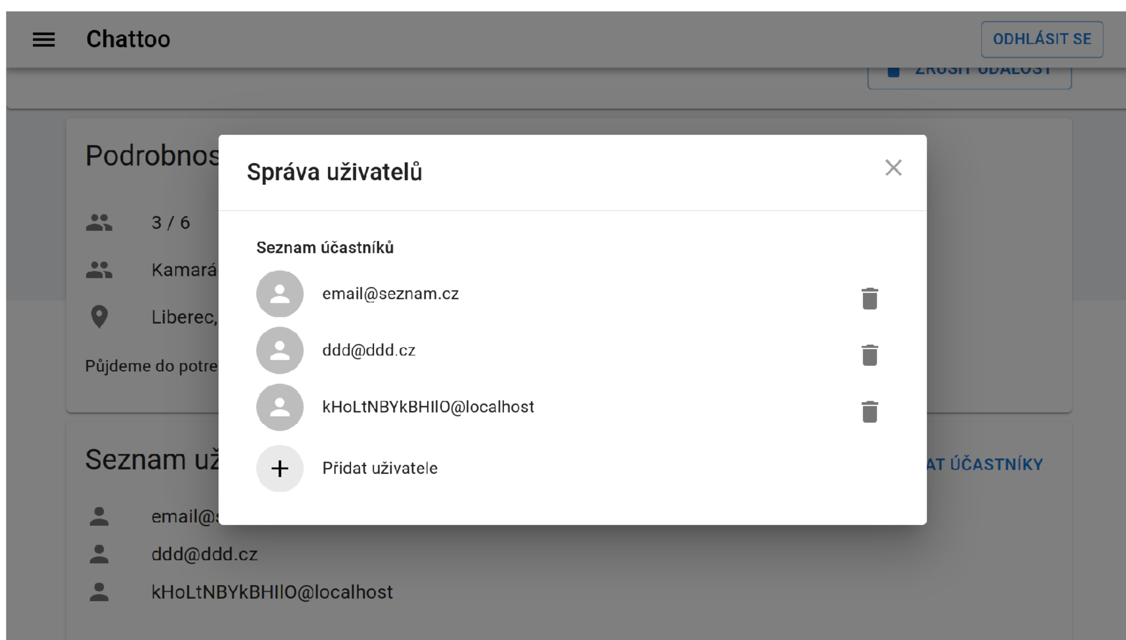
Obrázek 10: Prezentace aplikace - vytvoření kalendářní události



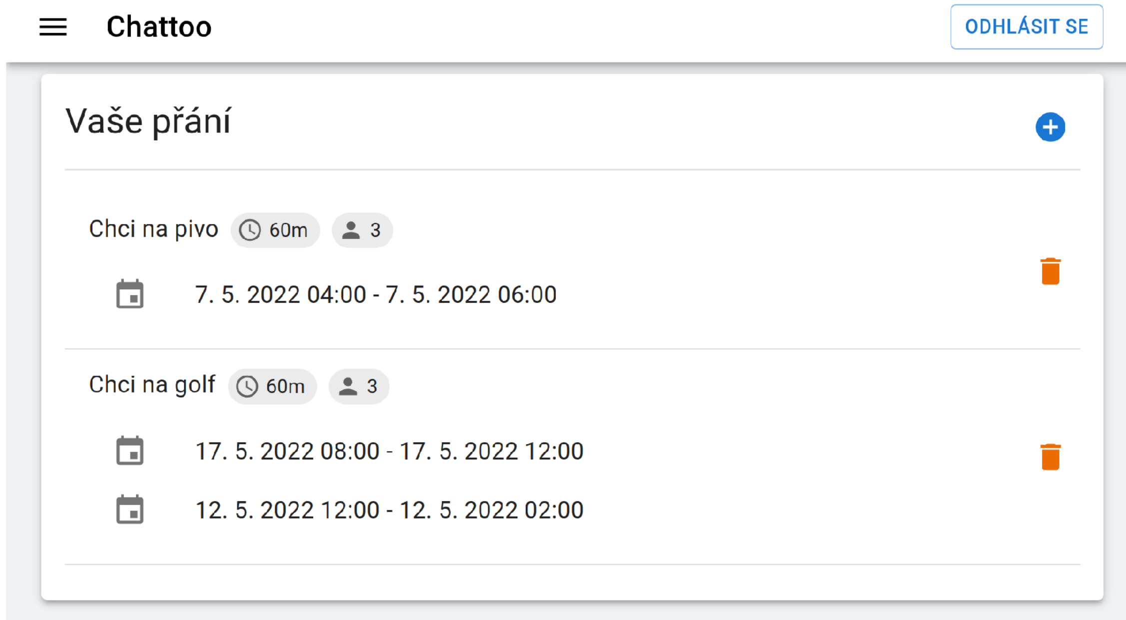
Obrázek 11: Prezentace aplikace - detail kalendářní události



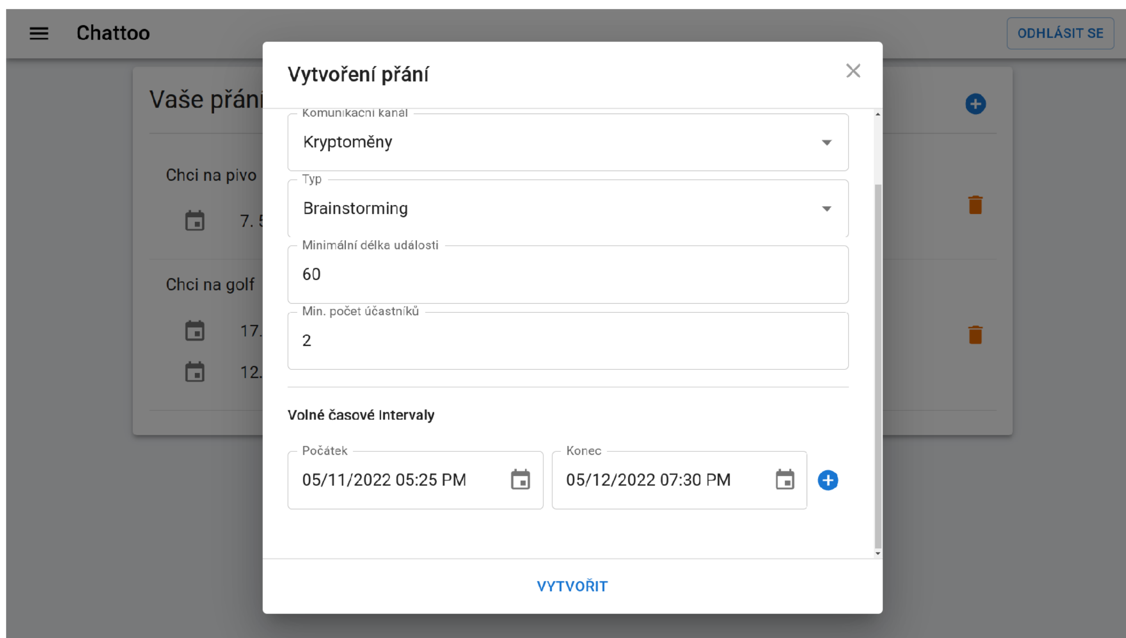
Obrázek 12: Prezentace aplikace - smazání kalendářní události



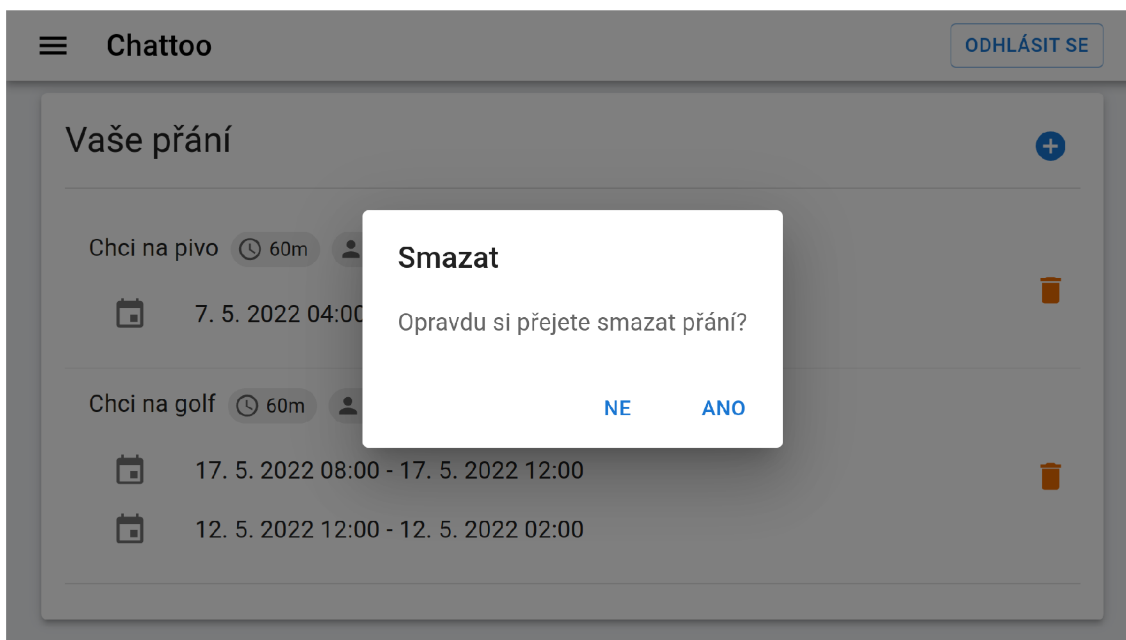
Obrázek 13: Prezentace aplikace - správa účastníků události



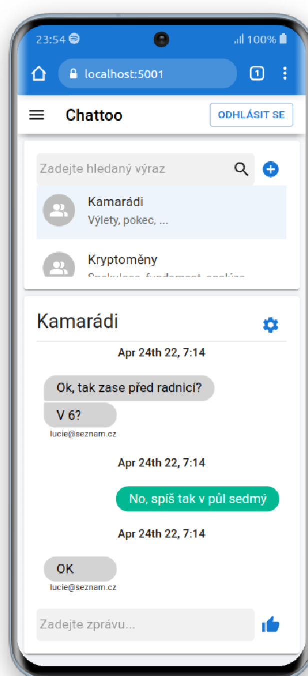
Obrázek 14: Prezentace aplikace - seznam přání uživatele



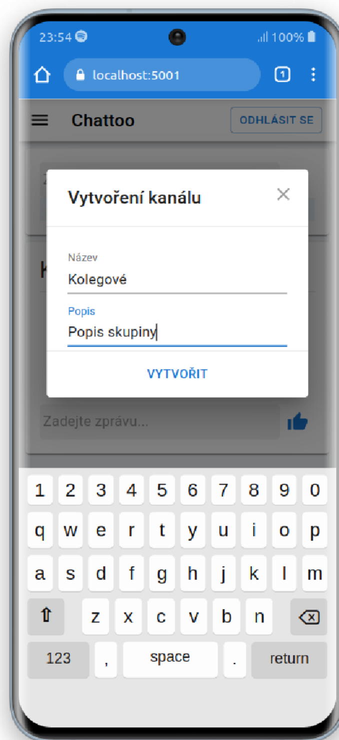
Obrázek 15: Prezentace aplikace - vytvoření přání



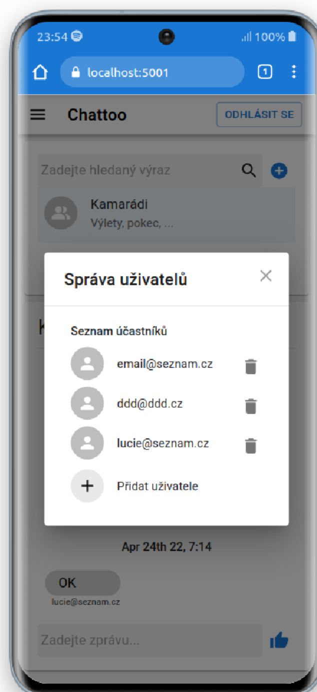
Obrázek 16: Prezentace aplikace - smazání přání



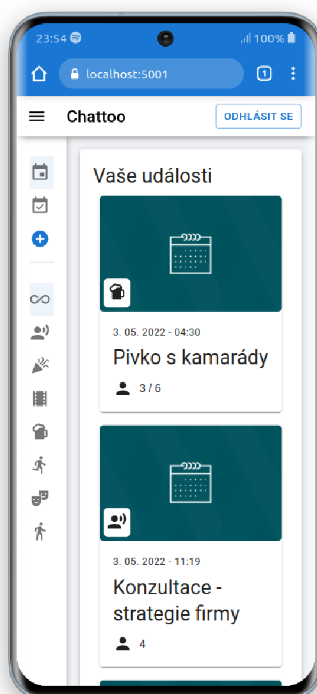
Obrázek 17: Prezentace aplikace - chat - telefon



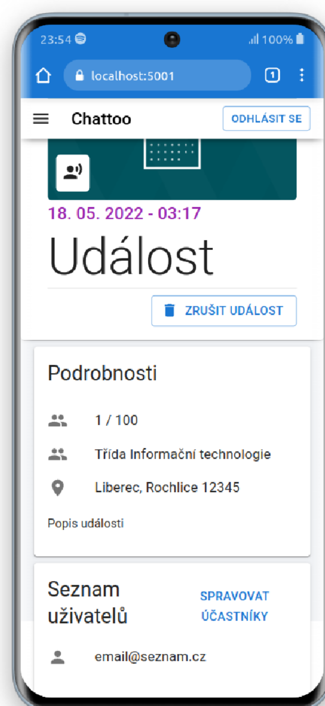
Obrázek 18: Prezentace aplikace - přidání komunikačního kanálu - telefon



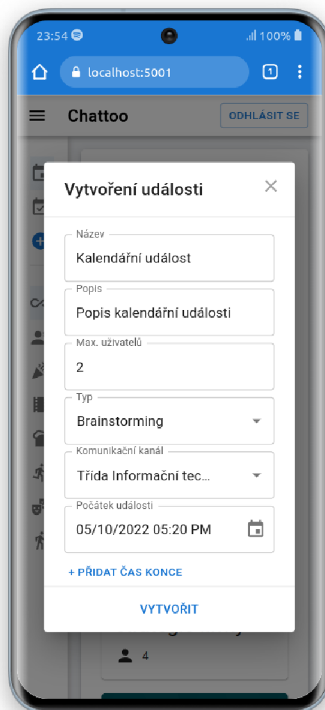
Obrázek 19: Prezentace aplikace - správa účastníků - telefon



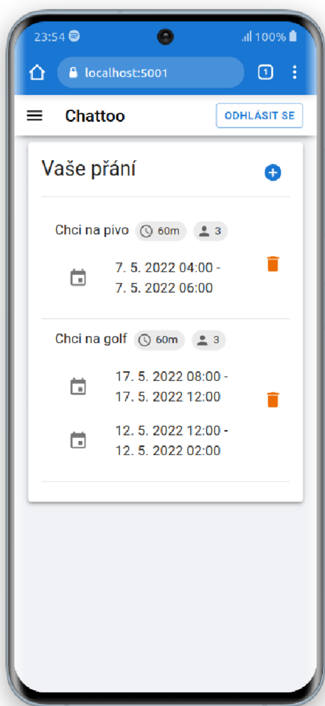
Obrázek 20: Prezentace aplikace - kalendářní události - telefon



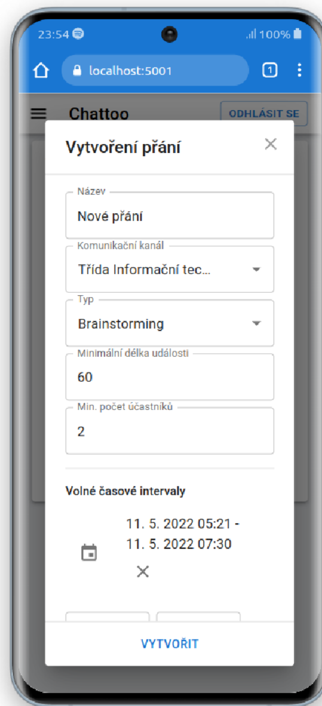
Obrázek 21: Prezentace aplikace - detail kalendářní události - telefon



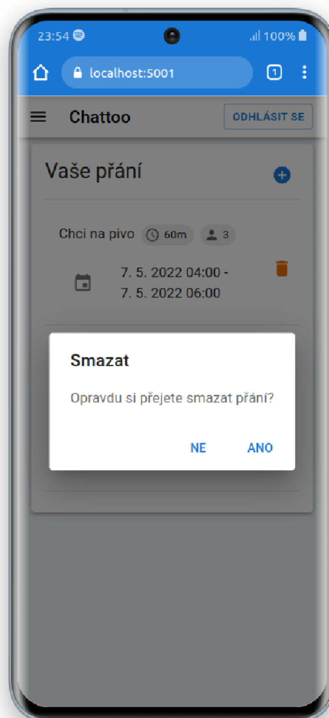
Obrázek 22: Prezentace aplikace - vytvoření kalendářní události - telefon



Obrázek 23: Prezentace aplikace - přání - telefon



Obrázek 24: Prezentace aplikace - vytvoření přání - telefon



Obrázek 25: Prezentace aplikace - smazání přání - telefon