FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Knihovna generující silně otypované API
Student:	Ludvík Prokopec
Vedoucí:	Ing. Ladislav Louka
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Technologie tRPC je často využívaný nástroj pro tvorbu silně typovaného API. Garance otypovaného API dává klientovi předpoklad o datových typech, které může odeslat a které obdrží. Na rozdíl od alternativních technologií jako GraphQL je tRPC jednodušší a nevyžaduje generování kódu, avšak funguje pouze pro velmi úzké nastavení projektu. Je navržená pro strukturu jednotného repozitáře a pouze pro proprietární API. Tato práce se snaží překročit tyto hranice pomocí návrhu a implementace vlastní knihovny, která nabízí silně otypované API a introspekci schématu k tvorbě otevřeného silně otypovaného API, zatímco zachovává jednoduchost tvorby projektu kterou podporuje tRPC.

1) Analyzujte existující metody řešení při použití tRPC a jiných technologií API
2) Navrhněte řešení podporující export schématu v jednom, případně více standardních formátech
3) Implementujte toto řešení
4) Sepište dokumentaci a testy
5) Publikujte vaši knihovnu na službu npm, a zveřejněte repozitář na GitHub připravený pro open source vývoj
6) Zveřejněte vzorový projekt využívající implementovanou knihovnu

---

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 27. října 2023 v Praze.

Bakalářská práce

KNIHOVNA GENERUJÍCÍ SILNĚ OTYPOVANÉ API

Ludvík Prokopec

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Ladislav Louka
16. května 2024

Obsah

Seznam výpisů kódu

Seznam obrázků

Seznam tabulek

Chtěl bych poděkovat svému vedoucímu bakalářské práce Ing. Ladislavu Loukovi za odborné vedení, čas, trpělivost, zkušenosti a cenné rady k vypracování jak teoretické, tak i praktické části práce. V neposlední řadě patří poděkování mé rodině, bez které bych tuto práci nemohl dokončit.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. v souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce se zaměřuje na řešení problému tvorby silně otypovaného API bez nutnosti generování kódu v programovacím jazyce TypeScript. Garance otypovaného API dává klientovi přehled o datových typech, které může odeslat a které obdrží. Na rozdíl od technologie GraphQL nevyžaduje manuální spuštění příkazů, čímž se proces vývoje stává plynulejší. Technologie tRPC je další alternativou pro vytváření otypovaného API bez nutnosti generování kódu, ta však funguje pouze pro velmi úzké nastavení projektu. Je navržená pro strukturu jednotného repozitáře a silné typování je k dispozici pouze se znalostí zdrojového kódu aplikace. Tato práce se snaží překročit tyto hranice pomocí návrhu a implementace vlastní knihovny, která nabízí silně otypované API a introspekci schématu k tvorbě otevřeného silně otypovaného API, kterou podporuje GraphQL, zatímco zachovává jednoduchost tvorby projektu, kterou podporuje tRPC.

Klíčová slova API, silně otypované API, schéma, introspekce, TypeScript, GraphQL, tRPC

Abstract

This bachelor thesis focuses on solving the problem of creating a strongly typed API without generating code in the TypeScript programming language. The guarantee of an otyped API gives the client an overview of the data types it can send and receive. Unlike GraphQL technology, it does not require manual execution of commands, making the development process smoother. The tRPC technology is another alternative for creating an otyped API without code generation, but it only works for very narrow project settings. It is designed for a monorepository structure and strong typing is only available with knowledge of the application source code. This work attempts to transcend these boundaries by designing and implementing a custom library that provides a strongly typed API with schema introspection to create an open strongly typed API, which is supported by GraphQL, while maintaining the simplicity of project creation that is supported by tRPC.

Keywords API, strongly typed API, schema, introspection, TypeScript, GraphQL, tRPC

Seznam zkratek

ACL	Access Control List
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
Blob	Binary large object
CI	Continuous Integration
CLI	Command Line Interface
CORS	Cross-origin Resource Sharing
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
GraphQL	Graph Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JSX	JavaScript Syntax eXtension
JWT	JSON Web Tokens
MVC	Model-View-Controller
OAS	Open API Specification
ORM	Object–relational Mapping
PTSQ	Public Type-safe Query
REST	Representational state transfer
RPC	Remote procedure call
SDK	Software Development Kit
SEO	Search Engine Optimization
SSR	Serser Side Rendering
TCP	Transmission Control Protocol
TLS	Transport Layer Security
tRPC	TypeScript Remote Procedure Call
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WHATWG	Web Hypertext Application Technology Working Group
XML	Extensible Markup Language

Úvod

Aplikace udržují data o jejich obsahu, ty však nejsou dostupná pro ostatní služby. Propojení dat aplikací třetích stran podporuje integraci služeb, což vede k efektivní a snadné interakci pro uživatele. Webové API umožňuje přístup k těmto datům a současně zajišťuje dodržování omezení, jako je například autorizace uživatelů.

Populární architekturou pro tvorbu webových služeb je REST, která má v ekosystému jazyka JavaScript širokou podporu různých frameworků. Při tvorbě REST API je běžnou praxí přiložit kódovou dokumentaci pro detailní popis rozhraní služby, jako je popis vstupů, výstupů a endpointů. Vývojář jiné aplikace pak díky dokumentaci ví, jak se službou pracovat. Samotná architektura ani nástroje pro tvorbu služby REST totiž obvykle nepodporují silně otypované API, které může dokumentaci nahradit. Typování vyobrazuje strukturu rozhraní na úrovni vývojového prostředí tím, že poskytuje datové typy přímo v rámci programovacích jazyků. To zjednodušuje a vytváří plynulejší a efektivnější práci vývojáře pracujícího se službou, protože není nucen neustále konzultovat webovou dokumentaci.

Další architekturou vhodnou pro vytváření webových služeb je RPC, ta definuje, na rozdíl od architektury REST, velmi volná omezení. Může se tak snadno stát, že některé operace budou mít pro klienta nepředvídatelné následky, což je důvodem, proč mnoho nástrojů pro tvorbu RPC vytváří schéma, které vede na silně otypované API. Architektura jako taková však schéma nevyžaduje.

Zavedení GraphQL znamenalo zásadní posun a změnu tradičního pohledu na datový model. GraphQL poskytuje schéma služby a prostřednictvím introspekce umožňuje distribuci tohoto schématu mezi externími klienty, čímž vytváří silně otypované otevřené API. Protože cílí na většinu programovacích jazyků, nedbá na žádné výhody jednoho z nich, a právě kvůli tomuto omezení je nutné neustále generovat kód.

Programovací jazyk TypeScript nebo JavaScript má v tvorbě webových aplikací unikátní možnost spouštět kód jak na straně serveru, tak i klienta. Tato flexibilita TypeScriptu umožnila vznik projektům jako je tRPC nebo Zodios. Díky možnosti sdílet kód i datové typy mezi serverem a klientem lze dosáhnout silně otypovaného rozhraní bez nutnosti generování kódu. Eliminace potřeby sestavení představuje významný pokrok v oblasti vývoje, který nabízí pohodlí, plynulost a zvyšuje efektivitu. Změny provedené na serverové straně se automaticky promítnou na klientské bez nutnosti manuálního spouštění příkazů.

tRPC je knihovna pro tvorbu silně otypovaného API bez nutnosti generovat kód. Tím, že se zaměřuje pouze na programovací jazyk TypeScript, umožňuje vytvářet rozhraní na straně serveru a sdílet jeho strukturu s klientem. Sdílení je však omezeno potřebou zdrojového kódu serveru, bez něhož ho nelze dosáhnout. Protože tRPC neposkytuje žádnou možnost exportu schématu jako GraphQL, nepodporuje tak vytvoření silně otypovaného otevřeného rozhraní mimo open-source služeb, typování je v takovém případě pouze proprietární.

Tato bakalářská práce se soustředí na vývoj knihovny pro snadnou tvorbu silně otypovaného otevřeného webového API bez nutnosti generování kódu. Hlavním cílem je tak analyzovat ostatní řešení a na základě této analýzy a výzkumu vytvořit službu umožňující snadný vývoj typovaného rozhraní. Silně otypované otevřené API poskytuje jasná pravidla a typové předpoklady pro interní i externí klienty, kteří se službou pracují. Díky vývojovému prostředí, které umožňuje zobrazovat typové nápovědy, se vývoj stává efektivnějším a plynulejším. Tyto nápovědy také částečně eliminují chybovost vývojářů, což vede k bezpečnějšímu, jednoduššímu a rychlejšímu procesu vývoje aplikace. Celkově tedy práce směřuje k usnadnění tvorby komplexních dynamických webových aplikací prostřednictvím pevného základu silně otypovaného API.

Práce se skládá ze čtyř hlavních částí: výzkum a analýza, knihovna PTSQ, implementace a porovnání s ostatními nástroji řešící stejnou nebo podobnou problematiku. Každá kapitola obsahuje několik podkapitol, z nichž mnohé by mohly být rozpracovány do samostatných prací. Některé informace jsou tak zjednodušeny na nezbytnou úroveň.

Kapitola Výzkum a analýza se bude věnovat zejména porovnání technologií a architektur pro tvorbu webových služeb a silně otypovaných API. Jednotlivé nástroje jsou důkladně analyzovány a jsou zmíněny jejich nedostatky, které je třeba eliminovat. Jsou také vyobrazeny jejich hlavní přednosti, kterých je třeba se držet při tvorbě této práce. Porovnání a výzkum již existujících řešení je nejdůležitější pro tvorbu efektivního nástroje. Kapitola analýzou stanovuje požadavky na samotnou knihovnu. Porovnáváním architektur a návrhových vzorů jiných projektů pak souběžně navrhuje rozhraní a možnosti pro samotný vývoj.

Knihovna PTSQ popisuje vlastnosti a použití samotné knihovny, která je výstupem této práce. Kapitola obsahuje detailní popis funkčnosti jednotlivých komponent a zobrazuje jejich použití a kompatibilitu s jinými frameworky. Zachycuje základní myšlenky a mechanismy, které stojí za fungováním knihovny.

Kapitola Implementace seznamuje se základními nástroji pro tvorbu řešení. Zároveň zachycuje proces testování nejen pomocí automatických testů, ale i testování použitelnosti rozhraní knihovny uvnitř komplexních projektů spolu s mnoha dalšími nástroji. Testy tak hovoří nejen o funkčnosti, ale i o kompatibilitě a možnostech nasazení knihovny uvnitř složitých projektů a webových aplikací.

Kapitola Porovnání s ostatními nástroji staví současné řešení vytvořené v rámci této práce mezi již existující nástroje. Tato řešení porovnává a zachycuje výhody i nevýhody knihovny PTSQ. Porovnání je jakýmsi předzávěrem, který poukazuje na konkurenceschopnost této práce mezi populárními projekty hojně využívanými v praxi.

Kapitola 1
Cíle práce

Tato kapitola se zabývá stanovením klíčových cílů bakalářské práce a hlavním důvodem a motivací pro její vypracování. Kromě toho slouží i jako rozšíření úvodu o detailní popis cílů, kterých má práce dosáhnout.

Hlavním cílem této bakalářské práce je vývoj knihovny pro snadné vytvoření silně otypovaného otevřeného webového API bez nutnosti generování kódu a manuálního spouštění příkazů. Důkladná analýza existujících nástrojů pro tvorbu takového typovaného rozhraní je klíčová pro výzkum jejich předností a nevýhod. Knihovna se bude inspirovat hlavními výhodami a bude se snažit minimalizovat nebo eliminovat různé nevýhody. Tyto nástroje neideálně přistupují k některým problémům při tvorbě silně otypovaného API. Motivací tak je sjednotit hlavní výhody jednotlivých projektů řešící tento problém při eliminaci klíčových nedostatků. Práce tak kombinuje nejlepší vlastnosti různých přístupů.

Celkově pak práce směřuje k usnadnění tvorby komplexních dynamických webových aplikací. Cílem tedy je vytvořit knihovnu, která plně zapadá a podporuje ostatní nástroje v ekosystému programovacího jazyka JavaScript. Tento ekosystém nabízí širokou škálu frameworků a knihoven pro tvorbu různých částí aplikace, jako jsou uživatelská rozhraní, databázová ORM nebo služby pro autentizaci a autorizaci uživatelů. Práce je tedy zaměřena na harmonickou spolupráci s těmito nástroji, aby dohromady tvořily komplexní webové aplikace.

Existují různá běhová prostředí pro tvorbu serveru podporující programovací jazyk JavaScript, jako je například Node.js, full-stack frameworky nebo alternativní prostředí, jako je Bun. Knihovnu, která bude vytvořena v rámci této bakalářské práce, by tak mělo být možné nasadit v téměř libovolném prostředí. To usnadní podporu různých stylů a preferencí vývojářů nebo organizací pro tvorbu silně otypovaného otevřeného API pomocí knihovny. Zároveň existuje obrovské množství frontend frameworků, které nabízejí různé přístupy k tvorbě uživatelského rozhraní a nemají ucelený způsob dotazování se serveru. Je tedy třeba zajistit kompatibilitu i s těmito frameworky.

Řešení musí být řádně otestováno, nejen automatickými testy, ale i testováním nasazení knihovny v různých ekosystémech a prostředích. To ověřuje správnost návrhu rozhraní knihovny a kompatibilitu s ostatními nástroji.

Knihovna pak musí být zdokumentovaná pomocí webové dokumentace. Ta je velmi důležitá pro popis funkcionalit a pochopení jinými vývojáři. Zároveň musí být snadno dostupná a publikovaná na službě NPM, aby ji bylo možné snadno nainstalovat pomocí správců závislostí jazyka JavaScript. Dalším požadavkem je zveřejnění zdrojového kódu a repozitář, který bude připraven pro další open-source vývoj.

Vytvoření několika vzorových ukázkových projektů je pak klíčové pro znázornění práce s knihovnou. Zasazení knihovny do několika prostředí a frameworků ukazuje její praktickou použitelnost a kombinovatelnost s ostatními nástroji.

Kapitola 2
Definice pojmů

Definice a vymezení klíčových termínů použitých v následujících částech práce a jejich stručný popis.

API

API představuje rozhraní, které nabízí snadný způsob připojení, integrace a rozšíření softwarových systémů. Software, který používají různé entity, je obvykle izolovaný a nedostupný z jiných systémů. Rozhraní API umožňuje propojení těchto oddělených softwarových celků, čímž umožňuje propojení produktů a služeb s jinými službami [1].

Webové API

Webová služba nebo webové API je systém určen k interakci strojů prostřednictvím sítě a je popsán ve strojově zpracovatelném formátu [2].

URI a URL

URI je řetězec představující identifikátor zdroje na internetu. URL je identifikátor umístění zdroje a jedná se o podmnožinu URI [3].

HTTP

Protokol HTTP je bezstavový internetový textový protokol na aplikační úrovni pro distribuované hypermediální informační systémy.

HTTP funguje stylem požadavek-odpověď. Klient odesílá požadavek, který obsahuje HTTP metodu, URI zdroje a verzi protokolu HTTP. Dodatečně může obsahovat hlavičky a tělo požadavku. Některé hlavičky jsou povinné v různých verzích HTTP, například hlavička Host ve verzi HTTP/1.1. Server následně odpovídá na požadavek odpovědí, která obsahuje verzi HTTP, status kód odpovědi a reason phrase. Reason phrase je standardní zápis daného status kódu ve formě řetězce. Dodatečně může odpověď obsahovat hlavičky a tělo.

HTTP metody udávají účel akce, která má být provedena nad zdrojem, ke kterému se klient snaží přistoupit.

Pro označení úspěšnosti odpovědi serveru používá HTTP status kódy. Výhodou těchto kódů je možnost určit na klientovi úspěšnost dotazu bez nutnosti číst tělo odpovědi [4].

Samotný protokol HTTP neumožňuje šifrování, pro šifrované spojení se využívá protokolu TLS. Šifrované spojení se označuje jako HTTP/TLS nebo HTTPS [5].

Protokol je bezstavový, podporuje však hlavičky, které umožňují udržovat informace o spojení na straně klienta. Pomocí rozšíření hlaviček o cookies si server může uložit informace o komunikaci s klientem na klientovi [6].

CORS

Protokol CORS se skládá ze sady hlaviček, které určují, zda lze odpověď sdílet mezi různými doménami. Prohlížeč z bezpečnostních důvodů nepovoluje mezi-doménové HTTP požadavky, které jsou vytvořeny skripty.

CORS definuje přípravný HTTP požadavek (preflight) s metodou OPTIONS, který slouží k domluvě klienta a serveru na mezi-doménovém odesílání odpovědí. Odpověď přípravného požadavku určuje, zda je zdrojová doména a ostatní nastavení povoleny. Server také může klientovi oznámit, že povoluje příjem „credentials“, což mohou být cookies nebo HTTP autentizace [7, 8].

Middleware

Middleware je software nebo služba nacházející se mezi volající a přijímající entitou. Běžně se nachází mezi operačním systémem a aplikací. Díky middleware je možné vytvořit vysokoúrovňové bloky aplikace a ty použít k úpravě komunikace s nějakou službou.

V rámci webových aplikací na straně serveru se tento termín konkrétněji používá k označení předem vytvořených softwarových komponent, které lze přidat do toku (pipeline) zpracování požadavků a odpovědí. Tato komponenta umožňuje měnit požadavky a odpovědi a zároveň tuto změnu abstrahovat ze samotného finálního zpracování, čímž zlepšuje modularitu systému. Může provádět různé úkoly, jako je validace dat, logování, zabezpečení, řízení přístupu, transformace dat nebo řízení toku dat [9, 10].

Endpoint

Koncový bod v kontextu webových služeb označuje zdroj API, který zpracovává daný požadavek.

JavaScript

JavaScript je multiparadigmatický dynamický programovací jazyk, jehož standard se nazývá ECMAScript. Je hlavním jazykem pro vývoj interaktivního obsahu na webových stránkách a je často kombinován s HTML a CSS na straně klienta. Běží v prohlížeči uživatele a umožňuje manipulaci s obsahem stránky, interakci s uživatelem a komunikaci se serverem bez nutnosti obnovy celé stránky.

JavaScript může běžet i na straně serveru, zejména prostřednictvím platformy Node.js. To umožňuje vývojářům využívat tento jazyk mimo webové prohlížeče. Tato flexibilita nabízí řadu výhod v procesu vývoje jako psaní celé aplikace v jediném programovacím jazyce nebo sdílení kódu mezi serverem a klientem [11, 12].

Node.js

„Node.js je asynchronní běhové prostředí JavaScriptu řízené událostmi, které je určeno k vytváření škálovatelných síťových aplikací.“ [13]

Díky Node.js je možné vytvářet backend webových i jiných aplikací nebo nativní a CLI aplikace pomocí jazyka JavaScript.

Polyfill

Polyfill je část kódu, která slouží k poskytování moderních funkcí pro starší prohlížeče, které je nativně nepodporují. Nejběžnějším příkladem je fetch API, které je definováno níže. Starší prohlížeč podporuje pouze zastaralou funkci XMLHttpRequest, polyfill tak využije této funkce pro vytvoření náhradní funkce fetch, čímž dosáhne stejného rozhraní. Samotný polyfill nemusí vždy nahrazovat JavaScriptový kód, existují i pro CSS [14].

Polyfill nelze vytvořit na všechny funkcionality. Například chování některých objektů, které poskytují a implementují prohlížeče, jako je Battery API[1], nelze napodobit.

Kompilátory jako TypeScript nebo Babel[2] při kompilaci automaticky přidají polyfill funkcionality a kód upraví tak, aby pracoval s dodaným náhradním kódem.

AJAX

AJAX je množina technologií pro vývoj webových aplikací, při které stránka získává obsah ze serveru asynchronními HTTP požadavky a používá nový obsah k aktualizaci příslušných částí stránky, aniž by ji bylo nutné načíst celou znovu. Do těchto technologií lze zahrnout reprezentaci HTML elementů pomocí JavaScript objektů DOM, XMLHttpRequest sloužící k asynchronní výměně dat se serverem nebo HTML a CSS, které zobrazují informace na webu [15, 16].

WHATWG

Web Hypertext Application Technology Working Group je sdružení, které se zaměřuje na vývoj a standardizaci webových technologií a specifikací. Vzniklo jako reakce na rozdílné názory a různé způsoby implementace mezi W3C a vývojáři prohlížečů především v otázkách HTML a DOM.

Zaměřuje se na praktické implementace a zpětnou kompatibilitu webových technologií. Hlavním cílem je vytvoření živého standardu, který reaguje na aktuální potřeby webových vývojářů a přináší rychlé inovace do webových technologií [17, 18].

[1] Battery API umožňuje získat informace o stavu baterie a napájení. Je popsáno na webové stránce https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API.

[2] Babel je dostupný na webové stránce https://babeljs.io/.

Fetch API

Fetch API je moderní rozhraní pro práci s HTTP požadavky a odpověďmi v JavaScriptu. Nahrazuje starší metodu XMLHttpRequest a nabízí mnoho vylepšení, zejména pokud jde o čistší syntaxi, podporu asynchronního kódu pomocí Promise[3] a podporu service workerů[4].

Fetch API podporuje různé typy datových formátů, včetně JSON, textu nebo Blob objektů[5] a je součástí standardu WHATWG [21].

CommonJS

CommonJS je systém pro vytváření modulů v prostředí Node.js, který vznikl před vznikem standardu ES modules. Aplikace na straně serveru nevyužívají HTML a neměly tak možnost rozdělit skript do více souborů (modulů), jako je tomu na straně klienta pomocí HTML tagu `script` [22].

Nynější podpora obou typů modulů především právě v prostředí Node.js způsobuje nekonzistence projektů a potřebu provádět konverzi mezi jednotlivými formáty, což může být zdrojem komplikací při vývoji softwaru. Převod mezi moduly zajišťují nástroje zvané bundlery, mezi které patří například Webpack[6] nebo Rollup[7]. Bundlery podporují nejen převod ale i generování kódu pro oba typy modulů z jednoho zdroje, například z kódu TypeScriptu [23].

ES modules

Zpočátku byly projekty a programy psané v JavaScriptu zpravidla malé. Většinou se používaly k skriptovacím úlohám, které poskytovaly webovým stránkám trochu interaktivity, takže rozsáhlé skripty nebyly potřeba. S postupem času se webové aplikace stávaly více dynamické a komplexnější a bylo potřeba kód JavaScriptu rozdělit a mít možnost použít kód na více místech. Prostředí Node.js podporuje tvorbu modulů ještě před vznikem standardu ES modules. Oficiální specifikace pro definici modulů v JavaScriptu, ES modules, je podporována jak webovými prohlížeči, tak i v novějších verzích Node.js. Na rozdíl od CommonJS podporují asynchronní importování kódu [23].

TypeScript

TypeScript je programovací jazyk, který do jazyka JavaScript přidává statickou kontrolu typů. Je nadmnožinou jazyka JavaScript (ECMAScript 2015), což znamená, že vše, co je k dispozici v jazyce JavaScript, je k dispozici také v jazyce TypeScript. Každý kód v jazyce JavaScript je také syntakticky validním v jazyce TypeScript. Protože prohlížeče, Node.js a ostatní běhová prostředí podporují pouze JavaScript, kód napsaný v TypeScriptu se tak kompiluje do JavaScriptového [24, 25].

[3] Promise je obal hodnoty, která není v době vytvoření nutně známá. Promise razantně zjednodušuje psaní asynchronních operací v JavaScriptu [19].

[4] Service worker funguje jako proxy server, který je mezi prohlížečem a sítí. Přesnější specifikace je dostupná na webové stránce https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[5] Blob objekt obsahuje nezpracovaná a dále nespecifikovaná data, která lze číst textově nebo binárně [20].

[6] Webpack je JavaScriptový bundler a je dostupný na webové stránce https://webpack.js.org/.

[7] Rollup je JavaScriptový bundler a je dostupný na webové stránce https://rollupjs.org/.

React

React je JavaScriptový framework, který vytváří reaktivní uživatelské rozhraní.

HTML kód webové stránky prohlížeč převádí do objektového modelu jazyka JavaScript, který pomocí metod a funkcí zpětně provádí úpravy stránky. Úpravy se často kaskádovitě šíří komplexním modelem stránky, což snižuje její výkon. To je částečně kompenzováno množstvím optimalizací, díky nimž není většina těchto objektů reaktivní. To znamená, že změna proměnných uvnitř jazyka JavaScript se nepropíše přímo do změny a překreslení stránky a rozhraní je nutné aktualizovat explicitně. Tyto nevýhody eliminují reaktivní frameworky jako je React vytvořením reaktivního proxy modelu [26].

S postupem času se webové stránky a aplikace stávaly čím dál komplexnějšími. I když knihovny jako jQuery[8] poskytly určitou pomoc při vytváření těchto složitých webových aplikací, imperativní manipulace s elementy byla často komplikovaná a náchylná k chybám. Deklarativní manipulace s elementy, kterou tyto reaktivní frameworky poskytují, razantně zjednodušila vývoj komplexních dynamických webových aplikací.

React zavádí virtuální DOM, který stejně jako DOM reprezentuje HTML element webové stránky. Je použit především při vytváření virtuálních stromů stránky. Ty jsou při změně stavu vytvořeny znovu a porovnány. Podle rozdílů nového a starého stromu se poté aplikují pomocí rozhraní DOM změny na HTML elementy stránky [27].

React také nabízí způsob definice komponent, tím vytváří znovupoužitelné části uživatelského rozhraní, jejichž chování lze upravovat atributy, podobně jako chování funkce argumenty. K definování stavu aplikace mezi překreslením stránky React využívá hooky. Hook je funkce, která je registrována uvnitř komponenty a která může udržovat stav komponenty nebo reagovat na změny jiných stavů. Klíčová vlastnost hooků je možnost tvorby vlastních komplexních na základech jednoduchých předdefinovaných. Logika komponent tak lze rozdělit do několika funkcí a tím výrazně zjednodušit a zpřehlednit kód aplikace [28].

Svelte

Svelte podobně jako React je nástroj pro tvorbu reaktivního uživatelského rozhraní webové stránky.

Na rozdíl od Reactu je Svelte kompilátor, kompiluje Svelte kód do JavaScriptu, čímž vytváří reaktivní proxy model. Díky kroku sestavení umožňuje jednoduchou a přehlednou tvorbu aplikace a výsledkem je velice optimalizovaný JavaScriptový kód [29].

[8] jQuery je knihovna pro manipulaci s HTML elementy, tvorbu asynchronních požadavků na server, vytváření animací a další. Hlavním přínosem je především kompatibilita kódu s většinou prohlížečů při jednotném rozhraní knihovny. jQuery je dostupné na webové stránce https://jquery.com/.

Kapitola 3
Výzkum a analýza

Tvorbě knihovny předcházely následující poznatky a zjištění. Analýza a zhodnocení již existujících technologií sloužilo k lepšímu porozumění kontextu a potenciálu práce. Výzkum se zabývá moderními nástroji a protokoly pro tvorbu nejen silně otypovaného API, ale obecně webových aplikací a služeb. Zároveň poukazuje na hlavní vzory různých řešení, čímž vytváří návrh samotné knihovny.

3.1 HTTP

Klient se při komunikaci se serverem dotazuje na zdroje pomocí URI a HTTP metod, které definují operaci, jež má být nad zdrojem provedena.

Tyto metody se dají dělit podle toho, zda jsou bezpečné nebo idempotentní. Bezpečné metody jsou ty, které nezpůsobují změny dat na serveru. Mezi ně patří metody GET a HEAD. Bezpečné metody jsou zároveň idempotentní, což znamená, že opakované volání téže metody s identickými parametry má stejný výsledek jako jedno volání. Idempotentní jsou například DELETE nebo PUT. Naopak metody, které nejsou ani bezpečné, ani idempotentní, jsou POST nebo PATCH. Toto rozdělení HTTP metod informuje uživatele, jak lze jednotlivé metody používat. Zároveň definují omezení pro tvorbu serverové části, například metoda GET by neměla být použita pro aktualizaci dat. HTTP metody představují význam operace a jsou klíčové pro efektivní a předvídatelnou komunikaci serveru a klienta, díky metodám klient mimo jiné ví, jaké požadavky lze ukládat do mezipaměti a jaké neupravují data.

Pro označení úspěšnosti odpovědi používá protokol status kódy. Výhodou standardizovaných kódů je předvídatelnost odpovědi, klient tak nemusí očekávat speciální status kódy. Naproti tomu nevýhodou je nemožnost kódy rozšířit o své vlastní, čímž HTTP znemožňuje definovat překlady chybových hlášek serveru na straně klienta, protože definování překladů závisejících pouze na status kódech může být nedostatečné. Tvorba překladů na straně serveru také není příliš vhodná, každý klient by měl mít možnost reprezentovat odpověď serveru po svém. Odesílání zprávy v chybové odpovědi bez dalších informací také není dostačující, ta totiž nenahrazuje identifikátor dané chyby jako vlastní chybový kód. HTTP umožňuje promítnout vlastní kód nebo označení chybové hlášky uvnitř těla odpovědi [4].

Knihovna, jež je výstupem této práce, musí rozdělovat operace pomocí předem určených metod jako samotný protokol HTTP. Díky metodám si jednotlivé akce nesou význam a klient před odesláním požadavků zná vlastnosti operací. Knihovna tak buď využije metod HTTP nebo definuje své vlastní. Samotná implementace pak bude používat protokol HTTP pro komunikaci serveru a klienta, protože tento protokol je v kontextu komunikace s webovými službami nejpoužívanější.

3.2 REST

Architektura REST představuje styl psaní distribuovaných hypermediálních systémů a vychází z návrhů webové architektury a principů síťové architektury webu. Samotná architektura však pokrývá mnohem víc než jen webové služby. REST definuje několik omezení a stylů, jejichž dodržováním podporuje tvorbu distribuovaných a škálovatelných hypermediálních systémů.

Prvním omezením je oddělení zodpovědností klienta a serveru. Tímto opatřením se dosahuje oddělení zájmů klienta a serveru. Rozdělením starostí týkajících se uživatelského rozhraní od těch, které se týkají ukládání dat, se zvyšuje přenositelnost uživatelského rozhraní mezi různými platformami. Zjednodušením komponent na straně serveru se zároveň dosáhne lepšího škálování systému. Největší výhodou tohoto omezení je pak možnost vyvíjet strany klienta a serveru nezávisle.

Bezestavovost komunikace je dalším důležitým omezením. Každý požadavek klienta na server musí obsahovat veškeré informace nutné k zpracování požadavku. Nevýhodou tohoto omezení je snížení výkonu sítě zvýšením opakujících se dat, protože data nelze ukládat na serveru mezi jednotlivými požadavky.

Za účelem zvýšení efektivity sítě, které je sníženo kvůli bezestavovosti komunikace REST definuje práci s mezipamětí. Server v odpovědi určuje, zda je možné tuto odpověď uložit do mezipaměti či nikoliv. Klient se poté nemusí vždy dotazovat serveru s identickými požadavky. Mezipamětí se zvyšuje efektivita sítě a škálovatelnost, protože se omezí nebo úplně eliminují interakce se serverem. Nevýhodou může být nekonzistence dat, která je uložena v mezipaměti a kterou by server vrátil při dotazu na něj.

Poskytnutí jednotného rozhraní zjednodušuje práci s REST systémem a zlepšuje přehlednost interakcí [30]. Při implementaci pomocí protokolu HTTP jsou standardně definovány CRUD operace nad jednotlivými dokumenty pomocí příslušných HTTP metod. Omezení se týká i správného zápisu URI zdrojů API a status kódů vrácených při různých odpovědích [31].

Styl vícevrstvého systému definuje chování komponent, kdy každá má přístup pouze k bezprostřední vrstvě, se kterou komunikuje. Omezením znalostí systému na jednu vrstvu se omezuje složitost systému.

Posledním omezením je kód na vyžádání, díky němuž je umožněno rozšíření funkčnosti klienta pomocí dodání a spuštění skriptů [30].

Výhodou síťové architektury REST je jasně definovaná sémantika operací a jednotné rozhraní. Nejdůležitějším předpokladem pro tvorbu knihovny generující silně otypované API je však stále dodržet oddělení zodpovědností klienta a serveru. I při vytváření typovaného rozhraní je zásadní mít možnost vytvářet kód serveru a klienta nezávisle. Zároveň je třeba umožnit oddělení klienta od serveru, klient se tak stává použitelným pro různé servery vytvořené pomocí knihovny. Je potřeba dodržovat i další omezení, která REST definuje, jako bezestavovost komunikace, ukládání do mezipaměti nebo jednotné rozhraní, komunikace pak probíhá efektivně a předvídatelně. Tato pravidla podporují tvorbu škálovatelných webových aplikací a služeb, kterou tato práce, i přes typované rozhraní, také musí splňovat.

3.3 RPC

Vzdálené volání procedur je architektura distribuovaných systémů umožňující volat proceduru na vzdáleném počítači.

Procedura je funkce nebo metoda, která je v kontextu RPC k dispozici na vzdáleném serveru a může být volána klientem ze vzdáleného počítače. RPC umožňuje klientovi volat metody nebo provádět funkce na vzdáleném serveru tak, jako by byly volány lokálně. Procedura může vyžadovat parametry, které jsou dodány klientem, zároveň může vracet klientovi odpověď.

V případě lokálního volání procedur volající předá argumenty a řízení proceduře. Procedura po vykonání operací předá řízení zpět volajícímu spolu s výsledky a volající pokračuje v provádění

operací.

Model vzdáleného volání procedur je velmi podobný. Volající odešle zprávu (požadavek) serveru a čeká na zprávu s odpovědí. Pokud server nemá požadavek, který by měl zpracovávat, je nečinný a čeká. Po přijetí požadavku začne server vykonávat operace procedury s argumenty, které přišly od klienta. Zatímco REST pevně definuje operace, které se mohou vykonat nad dokumentem, RPC takové omezení nepřináší. Výhodou je tak flexibilita rozhraní, která je však znevýhodněna nedostatečnou předvídatelností nebo významem akce. Definování operace, která se má provést nad daty, tak popisuje pouze název procedury.

RPC lze implementovat různými transportními protokoly jako TCP nebo UDP a různými protokoly aplikační vrstvy jako HTTP. Protokol RPC pouze definuje způsob předávání argumentů a volání procedur, samotná komunikace mezi klientem a serverem není protokolem RPC přímo specifikována.

Existuje několik protokolů, knihoven a frameworků, které usnadňují implementaci RPC, například tRPC, gRPC[1] nebo JSON-RPC[2] [32].

Výhodami RPC je flexibilita rozhraní, narozdíl od REST, který striktně definuje jednotné rozhraní, RPC umožňuje vytvářet službu volněji. Do jisté míry je to však i nevýhodou, server totiž může poskytovat procedury nepředvídatelně. To může mít za následek neefektivní komunikaci klienta a serveru, protože klient nezná sémantiku operace, která musí být popsána pouze správným názvem procedury. Schéma služby, které některé frameworky pro tvorbu RPC podporují, je tak klíčové pro efektivitu komunikace a do jisté míry umožňuje kompenzovat volně definované rozhraní.

3.4 Formáty pro výměnu dat

Formáty pro výměnu dat jsou standardizované formáty, které umožňují systémům sdílet informace a data mezi sebou. Definují způsob, jakým jsou data strukturována, uložena a přenášena, což usnadňuje interoperabilitu mezi různými systémy a aplikacemi.

JSON je textový zápis dat nezávislý na počítačové platformě nebo programovacím jazyku. Byl odvozen ze standardu programovacího jazyka ECMAScript. JSON definuje sadu pravidel pro přenášení strukturovaných dat. Data mohou být objekty, pole, řetězce, čísla anebo literálního typu null, false nebo true [33].

Pro validaci JSON dokumentů je možné využít JSON schéma. Toto schéma je JSON dokument, který umožňuje popisovat datové typy a strukturu dat. Na základě popisu je možné data validovat. JSON schéma je široce používaným nástrojem v oblasti vývoje softwaru a webových služeb, zejména tam, kde je nutné jasně definovat formát a strukturu datových objektů. Mnoho moderních frameworků a nástrojů podporuje validaci JSON dat pomocí JSON schématu [34].

Alternativou formátu JSON je například XML. XML je obecný značkovací jazyk určen především pro přenos dat. Stejně jako u JSONu je zpracování XML podporováno řadou nástrojů a programovacích jazyků a je nezávislé na počítačové platformě. Na rozdíl od JSONu využívá elementy a atributy pro ukládání dat [35].

Výhodou XML je zejména možnost definovat jmenný prostor (namespace), má však komplexní strukturu a mnoho nástrojů v ekosystému JavaScriptu s XML nepracuje. JSON se tak přímo nabízí jako formát pro přenos dat mezi serverem a klientem pro tvorbu této práce, zejména kvůli kompatibilitě a podpoře nástrojů v ekosystému jazyka JavaScript, který bude použit jak na straně klienta, tak i serveru. Při možnostech typování v jazyce TypeScript je navíc JSON vhodný pro možnosti snadno popsat jeho strukturu objektovým datovým typem. Vytvořit popis struktury formátu XML bez nějakých zásadních úprav pomocí datového typu TypeScriptu je nemožné.

[1] gRPC je RPC framework a je dostupný na webové stránce https://grpc.io/.

[2] JSON-RPC je jeden z mnoha protokolů RPC. Je dostupný na webové stránce https://www.jsonrpc.org/.

3.5 TypeScript

Protože JavaScript je dynamicky typovaný jazyk, není možné definovat typy staticky v kódu. To přináší jisté výhody jako je flexibilita jazyka a možnost úplně změnit typ proměnné za běhu programu. Nicméně to také přináší určitá rizika při vývoji rozsáhlejšího projektu. Statické typování poskytuje vývojáři předpoklady datových typů, kterých může proměnná nabývat, což dynamické typování nenabízí. Vývojář je tak nucen psát kód bez typových nápověd a kontrol.

Prohlížeč, Node.js a další běhová prostředí jsou schopná spustit pouze JavaScriptový kód. Kód napsaný v TypeScriptu tak musí být pro jeho spustitelnost přeložen do jazyka JavaScript. Tím je každý platný JavaScriptový kód také platný v TypeScriptu, což usnadňuje postupné zavádění typování do existujícího JavaScriptového projektu.

Jazyk TypeScript přidává definice datových typů, rozhraní, tříd a zapouzdření jejich vlastností a metod nebo funkcí a jejich parametrů i návratových hodnot. Kromě primitivních datových typů, jako jsou `string`, `number`, `boolean`, `bigint`, `null` a `undefined`, umožňuje TypeScript definovat složitější typy pomocí typových aliasů a rozhraní, případně pak spojováním datových typů díky sjednocení nebo průniku. Typové aliasy slouží k pojmenování existujícího typu, zatímco rozhraní definují strukturu objektů. Protože je TypeScript nadstavbou dynamicky typovaného jazyka, nabízí typy jako `any` a `unknown` pro pokrytí všech situací, které mohou nastat v jazyce JavaScript. Těmito typy umožňuje definovat volná data, která nemají více specifikovanou strukturu. Poskytuje speciální operátory jako `typeof` pro zjištění datového typu proměnné nebo `keyof` pro odvození typů klíčů objektu. Umožňuje přistupovat k datovému typu hodnot objektu nebo pole pomocí indexace. Podporuje detekce datových typů uvnitř podmínek, čímž z dané větve kódu (scope) odfiltruje nemožné typy. Poskytuje také již předdefinované typy pro jednoduché transformace nebo odvození datových typů, například `ReturnType`, `Awaited` nebo `Partial`.

Podpora generického programování umožňuje vytvářet obecné šablonové typy nezávislé na rozhraní objektu. Tato vlastnost podporuje tvorbu znovupoužitelného kódu, který je přizpůsobivý různým datovým typům. Při definování generických typů je možné vyžadovat, aby typ rozšiřoval jiný, což umožňuje omezit vstupy datových typů.

Tvorbou pokročilých konstrukcí na úrovni typů, jako jsou podmínky, cykly a odvozování typů, kterými TypeScript podporuje vytváření transformací, lze přesně specifikovat očekávanou hodnotu proměnných a parametrů funkcí, což zvyšuje bezpečnost a srozumitelnost kódu.

Přetěžování funkcí, tedy definování více verzí stejné funkce s různými parametry a implementacemi, je pak užitečné zejména pro manipulaci s různými typy vstupů a výstupů. Díky tomu je možné vytvořit stejnou funkci, která podle typů parametrů vrací jiné datové typy z návratové hodnoty.

Další vlastností jazyka TypeScript je možnost přetypovat hodnotu na `as const`. Všechna data jsou pak určena pouze pro čtení a TypeScriptu to dává možnost znát všechny hodnoty i na úrovni typů (kód 1). Přetypování `as const` se hojně využívá a je to velmi účinný nástroj pro definování typu nejčastěji nějakého komplexního objektu. Pro knihovnu této práce lze toto přetypování využít pro typování schématu API.

```ts
const object1 = {
  name: 'John',
  age: 18,
};

typeof object1; /*{
  name: string;
  age: number;
}*/

const object2 = {
  name: 'John',
  age: 18,
} as const;

typeof object2; /*{
  name: 'John';
  age: 18;
}*/
```

Výpis kódu 1 Přetypování objektu pouze pro čtení

TypeScript podporuje použití definic typů v hlavičkových souborech značených koncovkou `.d.ts`. Tím lze JavaScriptovému kódu přiřadit datové typy a integrovat tak externí knihovny, které v TypeScriptu napsány nejsou. Současně TypeScript umožňuje automatickou generaci těchto deklaračních map pro typování již zkompilovaného JavaScriptového kódu.

Při tvorbě knihovny je klíčové kód zkompilovat do JavaScriptu, uživatel knihovny totiž může aplikace vytvářet v JavaScriptu a knihovna, která by byla poskytnuta v TypeScriptu, by se tak, kvůli absenci kompilátoru na straně uživatele knihovny, nezkompilovala a nebyla spustitelná. Dokonce i když vývojář vytváří aplikace v JavaScriptu, vývojové prostředí pomocí dodaných deklaračních map umožňuje zobrazovat typové nápovědy a kontroly datových typů. Přeložení TypeScript kódu knihovny do JavaScriptu s přiloženými deklaračními mapami je tak nejlepší možnost pro sdílení funkcionality při poskytnutí typových nápověd.

Knihovnu je pak důležité poskytnout jak v podobě zdrojových kódů TypeScriptu, tak i v podobě spustitelného přeloženého kódu s deklaračními mapami. Díky zdrojovým kódům umožňuje vývojové prostředí zobrazovat i vnitřní implementace knihovny, nejen datové typy deklaračních map. Všechny tyto prvky razantně zjednodušují používání knihovny a hledání chyb v aplikaci.

3.6 Tanstack query

Většina webových klientských frameworků neobsahuje ucelený způsob načítání nebo aktualizace dat na webový server. Vývojáři jsou tak nuceni vytvářet vlastní funkce pro dotazování se serveru, což vede na neoptimalizované a neideální rozhraní pro správu dat.

Tanstack query je rodina knihoven pro různé reaktivní frontendové frameworky jako je React, Vue, Angular, Svelte nebo Solid.js, která aplikacím usnadňuje práci s posíláním požadavků na webový server a uchováváním dat. Poskytuje nástroje pro správu HTTP požadavků, dat aplikace, mezipaměti a další pokročilé funkce.

Jednou z hlavních vlastností je schopnost transformovat libovolný HTTP požadavek na objekty, které tyto reaktivní frameworky používají pro udržení stavu aplikace. V Reactu se tak jedná o hook, ve Svelte zase o store a tak dále. Takový objekt pak udržuje informace o úspěšnosti HTTP požadavku, zda se požadavek zpracovává, nebo obsahuje samotná data získaná ze

serveru. Tanstack query tak vytváří jednoduché, efektivní a ucelené dotazování a zpracování dat v aplikacích vytvořených v různých frontendových frameworcích. Způsob získávání dat ze serveru se pak mezi různými frameworky liší velmi minimálně.

Pro významové rozdělení požadavků využívá Tanstack query mutace a dotazy. Mutace je označení libovolného HTTP požadavku, který by měl jakkoliv upravovat data na serveru. Dotazy pak pro požadavky, které data na straně serveru neupravují. Tento přístup může být limitován neznalostí operací serveru a je tedy velmi důležité poskytnou klientům informace o jednotlivých operacích.

Tanstack query pak podporuje ukládání dat dotazů do mezipaměti na straně klienta. Tím zvyšuje rychlost odezvy aplikace, která pak působí pro koncového uživatele plynuleji a svižněji. Je však důležité podotknout, že i při uloženém výsledku v mezipaměti, odesílá Tanstack query vždy požadavek na server. Tato strategie ukládání dat se nazývá „stale-while-revalidate“. Samotné ukládání do mezipaměti pak funguje na principu klíčů, které svými hodnotami určují jednoznačnost dotazu. Každý stejný dotaz by tak měl mít stejné klíče a každý, byť jen trochu odlišný dotaz, by měl mít klíče rozdílné. Pokud mají dva nebo více dotazů stejné klíče, Tanstack query odešle požadavek na server pouze jednou, data pak aktualizuje na všech místech, kde byl dotaz vytvořen. Při dotazování se s parametry by měl klíč dotazu obsahovat veškeré závislosti a vstupy, čímž se odliší dotaz na stejný endpoint s jinými argumenty.

Pro správu a invalidaci dat uložených v mezipaměti pak poskytuje funkce, kterými umožňuje aktualizovat data a udržovat je synchronizovaná s daty na serveru. Na základě klíčů dotazů je možné invalidovat cache jen některých, jednoho nebo všech dotazů aplikace. Invalidace smaže data v mezipaměti a odešle nový požadavek pro aktualizovaná data. Výhoda invalidování dotazů spočívá především v kombinaci se spouštěním mutací, kdy v případě komplexní aplikace napsané v reaktivním frameworku je velmi obtížné přidat data vrácená z úspěšné mutace k ostatním datům, obzvláště pak pokud jsou vykreslena na více místech aplikace nebo pokud se mají například kvůli různým filtrům objevit jen v některých případech. Místo toho se invalidují dotazy, které znovu odešlou požadavek požadující po serveru nová upravená data. Tanstack query pak data aktualizuje kdekoliv uvnitř aplikace bez nutnosti znalosti mutace, která úpravu dat zavinila. Uvnitř složitého uživatelského rozhraní představuje invalidace obrovské usnadnění, vývojář se nemusí soustředit na úpravu lokálního stavu po provedení HTTP požadavku, stav se automaticky aktualizuje na všech místech aplikace.

Nevýhodou tohoto přístupu je overfetching. Data, která jsou nejspíše vrácena i z úspěšné mutace, jsou znovu dotázána přes invalidovaný dotaz. Zde je třeba balancovat mezi výkonem aplikace a jednoduchostí a přehledností kódu.

Knihovna také podporuje optimistické aktualizace. Ty umožňují okamžitě aktualizovat stav aplikace na základě uživatelem vyvolané akce a následně synchronizovat hodnoty s odpovědí ze serveru. Tím se zlepšuje uživatelská zkušenost a aplikace působí rychleji.

Tanstack query nabízí širokou škálu dalších funkcí, jako je stránkování, nekonečné rolování, vytváření závislostí mezi dotazy, přerušení požadavků, a celkově patří mezi nejpopulárnější knihovny pro správu HTTP požadavků [36].

Tanstack query je ideální pro tvorbu adaptérů frontend frameworků klienta knihovny. Nabízí velmi komplexní funkce pro tvorbu optimalizované komunikace se serverem a je modulární, čímž je umožněno využít Tanstack query pro tvorbu dalších knihoven na něm závislých. Klienti knihovny této práce tak mohou využívat všech funkcí Tanstack query při silném typování rozhraní klienta, které pak umožňuje typové nápovědy a kontroly.

3.7 Validační schémata a dvojí validace

V jazyce JavaScript existuje mnoho knihoven pro tvorbu validačních schémat a validací. V této práci jsou důležité především ty, které podporují statické odvození datového typu. To znamená, že lze odvodit, jaký datový typ budou mít data, která jsou oproti danému schématu validní. Toho bude využito při definování vstupu a výstupu endpointu API, čímž by bylo umožněno sdílet tyto typy bez generování kódu se stranou klienta. Na klientovi by pak TypeScript povolil pouze takové vstupy, které jsou validní vůči schématu, výstupní schéma by zas definovalo strukturu dat, která přijde ze serveru jako odpověď. Odvození typů bude zároveň sloužit pro povědomí o hodnotách vstupu nebo výstupu na serveru, kde by vstupní schémata udávala datové typy dat přicházejících od klienta, ty výstupní zase omezení serveru odeslat data v jiné než v pevně definované formě. Všechny tyto výhody vedou na efektivní a bezpečné API nejen v běhovém prostředí, ale i na úrovni typů TypeScriptu a tím vytváří silně otypované rozhraní.

Populárními knihovnami pro vytváření validačních schémat a validací s možností statického odvození typů jsou například Zod[3] nebo Yup[4]. Ani jedna z těchto knihoven neumožňuje převod validačního schématu do formátu JSON, který je klíčový pro možnosti sdílení schématu služby, tento převod by tak musel proběhnout pomocí vlastní implementace. Ve srovnání je Zod obecně lepší než Yup, obzvlášť v komplexních validacích a odvození typu schématu. Obě knihovny mají vcelku velký ekosystém a podporují je různé knihovny pro tvorbu formulářů na straně klienta jako React Hook Form[5] nebo Formik[6]. Tato kompatibilita je velmi důležitá z hlediska možnosti dvojí validace.

Dvojí validace snižuje počet požadavků na server, čímž optimalizuje síťovou komunikaci a umožňuje tak tvořit škálovatelné aplikace. Na frontendu se použije stejné validační schéma, které je použito na serveru, k ověření platnosti formuláře. Pokud formulář není platný, není třeba zatěžovat server odesláním neplatných dat. Teprve pokud je formulář platný, jsou data odeslána na server. Nicméně i server musí kvůli bezpečnosti data znovu validovat. Díky dvojí validaci webová aplikace působí pro uživatele svižněji.

3.8 Kontext požadavku

Kontext požadavku je obecně objekt, který se vytváří při každém dotázání API. Samotný objekt požadavku sice poskytuje značné množství důležitých informací o požadavku, nicméně není standardně rozšířitelný.

Výhodou vlastního kontextu požadavku tak je možnost vytvořit data v závislosti na požadavku a propojit tak zpracování s nějakou externí službou nebo dalšími dodatečnými informacemi. Vytvoření kontextu přináší mnohem přehlednější a širší informace o požadavku než jen objekt samotného HTTP požadavku. Protože middleware umožňuje upravovat tok dat a data samotného požadavku i odpovědi, při úpravě požadavku bez kontextu jsou aplikovány nestandardní změny na jinak standardní objekt. Tento přístup velmi znesnadňuje vývoj aplikace a může být zdrojem problémů.

[3] Zod je dostupný na webové stránce https://zod.dev/.
[4] Yup je dostupný na webové stránce https://github.com/jquense/yup.
[5] React Hook Form je dostupná na webové stránce https://react-hook-form.com/.
[6] Formik je dostupný na webové stránce https://formik.org/.

3.9 Frameworky pro tvorbu REST API

Frameworky pro tvorbu REST API v ekosystému jazyka JavaScript, zvláště pak v prostředí Node.js, nabízí velmi podobné rozhraní a liší se pouze v některých detailech. Právě tyto podobnosti jsou důvodem věnovat jim pouze tuto společnou podkapitolu.

Node.js http modul poskytuje základní rozhraní pro vytvoření HTTP serveru. Neumožňuje však jednoduché vytváření endpointů, modul totiž nabízí tvorbu serveru pouze pomocí jedné funkce. Ta představuje posluchače HTTP serveru, do které vstupují všechny požadavky [37]. To je velmi nepraktické, uvnitř této funkce je totiž nutné požadavky roztřídit podle endpointů nebo HTTP metod (kód 2). Frameworky jazyka JavaScript nebo TypeScript většinou zjednodušují tvorbu služby pomocí routerů. Ten narozdíl od přístupu http modulu umožňuje pro každý endpoint a HTTP metodu vytvořit oddělenou funkci (posluchače), která se spustí pouze při dotázání se na specifický endpoint s definovanou metodou (kód 3). Tento přístup výrazně zjednodušuje tvorbu REST API a umožňuje vytvořit komplexní webové služby.

```
createServer((request, response) => {
  const url = new URL(request.url, `http://${request.headers.host}`);

  switch(url.pathname) {
    case: '/user':
      if(request.method === 'GET') {
        // ...
      }

      if(request.method === 'POST') {
        // ...
      }
    break;
    // ...
  }
});
```

Výpis kódu 2 Node.js http modul pro vytvoření REST API

```
app.get('/user', (request, response) => {
  // ...
});

app.post('/user', (request, response) => {
  // ...
});
```

Výpis kódu 3 Použití Express routeru pro tvorbu REST API

Frameworky pro tvorbu REST API v ekosystému jazyka JavaScript většinou nepodporují silně otypované API. Klient webové aplikace se tak dotazuje na jednotlivé endpointy webové služby bez znalosti povolených vstupů a struktury výstupů. Při tvorbě otevřeného REST API je běžnou praxí dodávat ke službě webovou dokumentaci, popisující vlastnosti endpointů, jejich vstupy a výstupy. Ta se při vývoji musí neustále konzultovat a vývoj se tak stává velmi neplynulý. Protože dokumentace není pro vytvoření REST API povinná, existovat nemusí, a pak jedinou

možností pro zjištění vlastností API je zkusit se ho dotázat. To je však velmi neefektivní pro obě strany, klient nebo vývojář musí strávit čas zkušebními dotazy a server je zbytečně zatížen, dotazy totiž nejspíš nemají žádný význam a jsou jen testovací. Vytvořením schématu služby, které se promítne typováním uvnitř vývojových prostředí, se docílí nejen typové bezpečnosti, ale i plynulosti a efektivity s dotazováním se služby. Samotná dokumentace bez schématu, tedy typových předpokladů, tak není příliš praktická a používání API třetích stran je pro vývojáře značně obtížné.

Většina z nástrojů nebo frameworků pro tvorbu REST API neumožňuje snadné nasazení aplikace v různých prostředích. Známe knihovny, jako Express[7], Koa[8] nebo Fastify[9], sice podporují různá nasazení, ovšem v každém prostředí se proces velmi liší a občas je velmi komplikovaný a problematický. Tato vlastnost je dána zejména tím, že všechny tyto frameworky používají základní Node.js http modul pro vytvoření serveru.

Express a většina dalších frameworků neumožňuje ani vytvoření vlastního kontextu požadavku, Koa je jediný z vyjmenovaných tří, která nativně podporuje tvorbu kontextu, Fastify pak umožňuje vytvoření kontextu požadavku pouze pomocí pluginu. Některé podporují vytvoření middleware, protože však neposkytují kontext požadavku, úpravy uvnitř middleware jsou většinou aplikovány přímo na objekt požadavku, což není příliš vhodné zejména kvůli nestandardní úpravě standardního objektu. Taková úprava pak může přinést spoustu nečekaných problémů a může vést k nepředvídatelnému chování aplikace. Žádný ze zmíněných frameworků nepodporuje typově bezpečnou úpravu kontextu, jakou podporuje tRPC, které je popsáno níže. Vývojář tedy nemá přehled, jakých hodnot může objekt požadavku nabývat nebo musí objekt manuálně přetypovat. Takový přístup je velmi neefektivní a přispívá k tvorbě chyb.

Jedním z nejmodernějších nástrojů pro tvorbu REST API je feTS. Tato knihovna využívá JSON validačních schémat pro validaci dat v běhovém prostředí, ale i k definování schématu silně otypovaného API. Z JSON schémat je totiž možné odvodit datový typ validních dat a tím definovat typové předpoklady pro klienta. Ovšem pro možnost odvození typů je nutné, aby validační schéma bylo definováno pouze pro čtení, tedy jako `as const`. feTS také umožňuje vytvoření silně otypovaného otevřeného API díky exportu schématu rozhraní ve formě Open API specification (OAS). Podporuje snadné nasazení aplikace uvnitř několika prostředí jako je Node.js, Bun, Deno nebo serverless prostředí. Nevýhodou feTS je náročnost tvorby otypovaného API. Je třeba definovat jak úspěšné výstupy endpointů tak i všechny chybové. Zároveň neumožňuje, podobně jako tRPC, vytváření předpřipravených komponent, a právě tato limitace je jedním z hlavních důvodů, proč je tvorba API uvnitř této knihovny zdlouhavá a náročná. Dalším problémem může být komplikovaná podpora middleware ve formě pluginů, ty jsou aplikovatelné pouze na celou aplikaci, nikoliv na jeden endpoint [38]. Middleware jednoho endpointu podporují i knihovny jako Express nebo Fastify.

Při tvorbě knihovny je tak důležité vytvořit podobně snadné rozhraní routerů, které definují endpointy. Zároveň je třeba umožnit vytvářet middleware jak celého serveru, tak i jednotlivých endpointů.

[7]Express je jeden z nejznámějších frameworků pro tvorbu webových služeb v prostředí Node.js. Je dostupný na webové stránce https://expressjs.com/.

[8]Koa je alternativou frameworku Express, poskytuje však možnost tvorby kontextu požadavku. Framework Koa je dostupný na webové stránce https://koajs.com/.

[9]Fastify je framework pro tvorbu webových služeb, který cílí především na optimalizaci a výkon. Je dostupný na webové stránce https://fastify.dev/.

3.10 GraphQL

GraphQL je dotazovací jazyk poskytující flexibilní datový model a syntaxi k popisu požadavků. Je nezávislý na programovacím jazyku a v mnohých má podporu v podobě knihoven a nástrojů, a to jak na straně klienta, tak i serveru [39].

GraphQL se řídí požadavky klienta, tím se liší od klasické architektury REST nebo RPC. Klient žádá API, sám si ale vybere strukturu a data, která mu mají ze serveru přijít. GraphQL pouze zveřejňuje schopnosti, které mohou klienti využít. Klient je zodpovědný za přesnou specifikaci toho, jak bude tyto zveřejněné schopnosti spotřebovávat. Tento způsob definování dat na úrovni klienta, je jednou z největších výhod.

Každá služba GraphQL definuje systém typů specifický pro danou aplikaci a tím vytváří otevřené silně otypované API. Celý systém těchto typů se nazývá GraphQL schéma, které definuje strukturu a popis celého GraphQL API. V rámci typů definuje GraphQL pole (fields), která popisují vlastnosti objektových typů a mají definované vlastní GraphQL typy, které mohou být primitivní nebo objektové. Všechny požadavky na dané rozhraní jsou pak validovány oproti tomuto schématu. Při neúspěšné validaci volání skončí chybou. Tato automatická validace je velkou výhodou pro vývojáře serveru, protože validace příchozích i odchozích dat je abstrahována ze samotného zpracování požadavků. Vlastní skalární typy, které umožňují definovat serializace a deserializace na úrovni schématu, také napomáhají k vytvoření přehledného zpracování žádostí, které se nemusí zabývat úpravou vstupů před jejich použitím ani výstupů před odesláním na klienta. Vlastní skalární typy díky deserializaci vstupu umožňují také jeho validaci, v praxi však může být tento přístup nevhodný kvůli nesynchronizovanému validačnímu procesu na serveru a klientovi. Častěji se využívá validačních schémat, které díky sdílení kódu dovolují dvojí validaci, ta se provede jak na straně serveru při přijetí argumentů, tak i na straně klienta před odesláním dat formuláře. Validační schémata dovolují také mnohem přesnější a komplexnější validace.

```
scalar Date

type User {
  id: ID!
  name: String!
  email: String!
  password: String
  emailVerifiedAt: Date
}
```

Výpis kódu 4 Objektový GraphQL typ s vlastním skalárním typem

Na rozdíl od architektury REST, GraphQL poskytuje pouze 3 základní operace, které se vykonávají nad daty. Objektové typy schématu Query, Mutation a případně Subscription, definují operace a vytvářejí tak vstupní bod API. Protože typů operací je méně, částečně tak postrádají sémantiku, což v důsledku znamená, že pro vytváření entit, aktualizace i mazání obsahu se používají mutace. To je však kompenzováno názvem dotazu, který musí být dobře formovaný, a schématem, které popisuje rozhraní API. Jedním z nejdůležitějších polí GraphQL schématu jsou pak meta pole, která popisují samotné schéma. Meta pole `__typename` popisuje název objektu a většinou se odesílá automaticky s každou odpovědí. Toho se využívá zejména pokud je pole definováno jako spojení několika typů, pomocí typename pak lze rozhodnout jaký z několika typů je skutečně v odpovědi. Tento způsob provázání klienta a serveru pomocí názvu objektu také přináší výhody v oblasti cachování dotazů a vytváří efektivní komunikaci.

Pro export schématu poskytuje GrahpQL introspekci, při které se klient dotazuje speciálním metadotazem na samotné schéma. Introspekce musí obsahovat pouze potřebné informace a minimalizovat tak riziko úniku citlivých dat. Protože odpovědi GraphQL jsou serializovány

do formátu pro přenos dat, nejčastěji do JSONu, běžně se při introspekci používají transformační nástroje na straně klienta pro zpětný převod odpovědi introspekce na formát GraphQL schématu. Jedním z mnoha nástrojů pro tento převod v ekosystému jazyka JavaScript je GraphQL codegen[10], který dokáže nejen transformovat JSON do GraphQL schématu, ale umožňuje z introspekce vygenerovat například typově bezpečného klienta nebo React klienta závislého na dalších knihovnách třetích stran. Generování kódu sice není plynulé a vyžaduje manuální spouštění příkazů, poskytuje však možnost vytvoření otypovaného externího klienta.

Pro chybové odpovědi specifikuje GraphQL pole `errors`, které je součástí celkové odpovědi pouze v případě, že server vrátí nějakou chybu. Každá chybová odpověď musí obsahovat zprávu, která popisuje problém a slouží vývojářům jako vodítko pro pochopení a opravu chyby. Pokud lze chybu přiřadit k určitému poli GraphQL odpovědi, chyba musí obsahovat cestu, která uvádí, kde se toto pole nachází. Samotná chybová odpověď lze rozšířit o vlastní kód chyby, nevýhodou však je, že tento kód není součástí schématu, může tak nabývat nepředvídatelné hodnoty [39].

Pro vytvoření GraphQL serveru existuje v ekosystému jazyka JavaScript spoustu knihoven. Samotný server většinou využívá jediný endpoint například REST API, většina frameworků pro tvorbu GraphQL serveru pak umožňuje snadno nasadit GraphQL API v mnoha prostředích. Nejznámějšími představiteli mohou být GraphQL Yoga[11] nebo Apollo GraphQL[12]. Jednotliví poskytovatelé serveru se příliš neliší a poskytují téměř stejné funkcionality, a to buď přímo pomocí modulu serveru nebo pomocí různých externích knihoven.

Při samotném dotazování pak server pro každý požadavek vytváří GraphQL kontext. Tento kontext představuje kontext požadavku v aplikacích GraphQL. Je předáván do všech funkcí (resolverů), které definují výslednou hodnotu každého jednoho GraphQL pole. Obsah kontextu požadavku může být dynamicky měněn v průběhu zpracování dotazu, což umožňuje měnit chování aplikace v závislosti na konkrétních podmínkách a požadavcích [40, 41].

Protože jedním z hlavních cílů knihovny této práce je tvorba silně otypovaného otevřeného API, největší inspirací tak je introspekce schématu. Knihovna musí také podporovat export schématu, při němž nesmí docházet k úniku citlivých dat serveru jako je připojení k databázi nebo tajné klíče. Díky tomu lze vytvořit otypované externí klienty, kteří službu využívají.

Další vhodnou vlastností, kterou by knihovna měla splňovat, je snadná možnost a implementace dvojí validace. Tím by bylo umožněno vytvářet škálovatelné aplikace.

Velmi důležitá je také možnost definování vlastních serializací nebo deserializací mimo funkci pro zpracování požadavku. Tento přístup velmi zjednodušuje a zpřehledňuje tvorbu serveru, čímž vytváří obecně kratší kód potřebný pro zpracování samotného požadavku. Funkce zpracovávající požadavek se tak nestará o úpravu dat před jejich použitím nebo po něm a tyto úpravy jsou abstrahovány do jiné vrstvy aplikace.

Pro knihovnu tvořící silně otypované API je také důležité inspirovat se možností definování vlastních chybových kódů při chybové odpovědi. Klienti pak mohou specifikovat vlastní překlady určené pro dané chybové hlášky. Pro takové možnosti je však nutné, aby byl kód chyby vždy obsažen ve schématu služby, tímto přístupem se knihovna bude lišit od GraphQL, které vlastní chybové kódy ve schématu nepromítá. Samotný kód chyby ale není dostatečný a musí být umožněno odeslat zprávu chyby, stejně jako umožňuje GraphQL, zejména pro možnosti pochopení chyby vývojáři.

Možnost definování vlastního kontextu požadavku je pak nutná pro vytvoření přehledných informací o požadavku. Tento objekt umožňuje přidat dodatečné informace k samotného objektu reprezentující HTTP požadavek a podporuje úpravy tohoto kontextu v průběhu zpracování dotazů.

[10] GraphQL codegen je dostupný na webové stránce https://the-guild.dev/graphql/codegen.
[11] GraphQL Yoga je dostupná na webové stránce https://the-guild.dev/graphql/yoga-server.
[12] GraphQL server Apollo je dostupný na webové stránce https://www.apollographql.com/.

Generátory GraphQL schématu

Většinou je GraphQL schéma na straně serveru reprezentováno v programovacích jazycích ve formě řetězce [42]. Takový způsob reprezentace však není příliš praktický. Zároveň se schéma odděluje od samotných resolverů, což není velmi přehledné a spíše to samotný vývoj komplikuje. Proto se používají generátory GraphQL schémat, které umožňují vytvářet typově bezpečný kód na straně serveru a na jeho základě generují samotné schéma. Generátory většinou spojují kód schématu i resolverů pro větší přehlednost v aplikaci, je tak možné vytvořit část schématu, která je později připojena k celku. Kvůli nutnosti generování kódu se však vývoj stává méně plynulým.

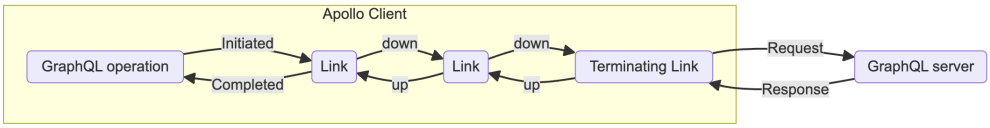
V programovacím jazyce TypeScript jsou dva nejpoužívanější generátory, jedním z nich je Nexus [43], který pro typovou bezpečnost vyžaduje aplikaci na straně serveru neustále generovat, a to i v případě nepotřeby sestavení GraphQL schématu. Novějším generátorem je Pothos [44], který tuto velkou nevýhodu eliminuje a poskytuje typovou bezpečnost na základě typů v jazyce TypeScript bez nutnosti generování kódu, pro vytvoření samotného schématu však také vyžaduje manuální spouštění příkazů. Většina generátorů zároveň poskytuje tvorbu pluginů (middleware), které obalují volání GraphQL resolverů, čímž mohou upravovat jak vstup, tak i výstup.

V knihovně, která je výstupem této práce, je tak třeba podporovat podobnou typově bezpečnou tvorbu serveru jakou nabízejí generátory schématu. Generování kódu ale zpomaluje a znesnadňuje vývoj, je tedy třeba snažit se předejít manuálnímu spouštění příkazů pro vytvoření silně otypovaného API.

GraphQL klient

Pro samotnou komunikaci s GraphQL serverem není speciální klient potřeba [45]. GraphQL klient ale zpravidla poskytuje jednoduché a přehledné rozhraní pro sestavení dotazů a získávání odpovědí ze serveru. Někteří klienti poskytují možnosti optimalizace dotazů, jako je například redukce počtu dotazů pomocí hromadného získávání dat (batching) nebo cachování odpovědí.

Existují knihovny poskytující možnost vytvoření middleware nebo pluginů na straně klienta. Nejpopulárnějším přístupem jsou Apollo linky [46], které jsou implementovány v klientovi Apollo GraphQL (obrázek 3.1). Tyto linky umožňují upravit dotaz ale také odpověď a abstrahovat tuto úpravu ze samotného definování požadavku.



Obrázek 3.1 GraphQL Apollo klient – linky [46]

Posledním linkem je terminating link, který je zodpovědný za odesílání dotazů na server a přijímání odpovědí. Výhodou různých a vlastních terminating linků je možnost definovat chování odesílání dat na server.

GraphQL klient sám o sobě není typově bezpečný a obvykle neposkytuje možnosti pro vytvoření typování. K tomu slouží generátory[13], které mohou z GraphQL schématu získaného z introspekce vygenerovat otypovaný klientský kód.

Z popsaných vlastností GraphQL klienta si lze odvodit, že tvorba otypovaného klienta pouze pomocí informací na klientské straně není možná, server však otypovaného klienta vytvářet může, má totiž přístup ke schématu API. V praktické části práce je tak potřeba věnovat pozornost

[13] Generátory klientů nejsou totéž, co generátory GraphQL schématu. Slouží k vytvoření otypovaného klienta, nikoliv schématu na serveru.

získávání otypovaného klienta nebo schématu ze serveru. Server tak definuje nejen serverový kód, ale i strukturu klienta, a jen tímto provázáním je možné dosáhnout silného typování.

Zároveň by bylo vhodné umožnit tvorbu komponent ve formě linků pro možnosti úpravy požadavků. Toho pak lze využít pro definování vlastních terminating linků, čímž se zlepší modularita klienta.

Dalším předpokladem je pak možnost dotázat se serveru vytvořeného pomocí knihovny bez nutnosti speciálních klientů. Tím se server stává použitelným pro klienty různých implementací, samotné fetch API, XMLHttpRequest nebo třeba pro testovací a navrhovací nástroje jako je Postman[14]. Ovšem využitím vlastní tvorby požadavků přichází klient o možnosti typování.

3.11 tRPC

tRPC je knihovna nebo nástroj, umožňující vytvářet typově bezpečné RPC bez nutnosti generování kódu. Je určený výhradně pro programovací jazyk TypeScript nebo JavaScript. Toto striktní omezení programovacího jazyka na straně klienta i serveru přináší určité výhody při přenášení kódu mezi oběma částmi aplikace. Díky této vlastnosti je možné vytvářet typově bezpečné API bez nutnosti generování kódu.

Jednou z charakteristických vlastností tRPC je absence schématu a možnosti introspekce. Namísto toho tRPC generuje TypeScript typy, které jsou přenášeny ze serverové části kódu na klienta. Tato implementace však přináší několik omezení, jako je nutnost znalosti zdrojového kódu serveru. To znamená, že tRPC není vhodné pro projekty, které mají rozdělenou kódovou základnu v různých repozitářích, v takovém případě by totiž kód serveru musel být distribuován mezi klienty jako NPM balíček. Z toho vyplývá, že použití jednotného repozitáře pro přenos typů mezi serverem a klientem je téměř nutností, dokumentace samotného nástroje ani jinou strukturu projektu nedoporučuje. Stejně tak není možné vytvářet proprietární otevřené silně otypované API, neboť tRPC poskytuje typování pouze při znalosti zdrojového kódu serveru. Knihovna sice umožňuje pomocí externích nástrojů vytvořit otevřené API, externí klienti ale kvůli neznalosti zdrojového kódu přicházejí o typované rozhraní.

tRPC definuje vstupy a výstupy endpointů API pomocí validačních schémat, z nichž jsou odvozeny TypeScript typy validních dat (toto odvozování bylo popsáno v podkapitole 3.7). Proces tvorby serveru pomocí definování validačních schémat je poměrně pracný, tRPC však umožňuje tvořit serverové komponenty, které používají pouze část validačního schématu, jehož zbytek lze dodefinovat až na konkrétní potřeby endpointů. Tím se snižuje opakování kódu a zásadně to zvyšuje efektivitu při psaní aplikace.

Definování struktury API pouze pomocí validačních schémat umožňuje vynechat vytváření schématu samotného nástroje, které vyžaduje GraphQL. Nicméně to znemožňuje tvorbu vlastních skalárních a speciálních typů s vlastní serializací a deserializací. Místo toho tRPC využívá transformátory, které definují serializace různých JavaScript konstruktů, a tak umožňují přenášet komplexní data jako například objekty `Date`, `Set`, `Map` nebo vlastní třídy mezi serverem a klientem. Omezením transformátorů je označování JavaScriptových struktur při serializaci, aby při deserializaci mohl parser strukturu identifikovat a převést zpět do správného objektu. V důsledku to znamená, že klient i server musí používat stejný transformátor.

V rámci tRPC je možné vytvářet operace typu Query, Mutation a Subscription, stejně jako tomu bylo u GraphQL. Dotazy jsou posílány s HTTP metodou GET a mutace s metodou POST. Pro implementaci subscription využívá web socketů.

tRPC, stejně jako GraphQL, vytváří při každém požadavku na server kontext a plně podporuje tvorbu middleware přímo uvnitř knihovny. Výhodou oproti generátorů schémat GraphQL, které také umožňují tvorbu middleware, je typová bezpečnost přenášení kontextu požadavku z jedné middleware do jiné, což zajišťuje menší chybovost a větší efektivitu vývojářů. Pokud

[14]Postman je nástroj pro testování a vytváření webových API a je dostupný na webové stránce https://www.postman.com/.

například uvnitř middleware ověřím, zda je uživatel přihlášený, v další middleware už datový typ uživatele nezobrazuje informaci o tom, že by mohl být nepřihlášen [47].

Největší inspirací pro knihovnu této práce je možnost tvorby silně otypovaného API bez spouštění příkazů. Definování schématu služby pomocí validačních schémat pak velmi zjednodušuje vývoj a umožňuje komplexní validace při možnosti sdílení validačních schémat pro dvojí validace. Dále je velkou inspirací velmi silná typová bezpečnost při tvorbě serverové části, zvlášť pak při definici middleware a úpravě kontextu požadavku.

tRPC klient

tRPC nabízí dva klienty, pro JavaScript a pro React nebo Next.js framework. Oba klienti fungují na základě JavaScript Proxy[15] díky čemuž je možné, aby se tRPC klient choval jako objekt a zároveň postupně tvořil proceduru, na kterou se klient bude dotazovat (kód 5).

```
const response = await client.user.get.query();
```

Výpis kódu 5 tRPC klient – spuštění dotazu

Díky Proxy objektu je možné sledovat při přístupu k objektu `client` klíče a tvořit tak název procedury, například `user.get`. Klient je typově bezpečný díky importování typu kořenového routeru tRPC, který je vytvořen na serverové straně (kód 6). Kód samotného serveru je do klienta možné vložit jako vývojovou závislost, protože klient vyžaduje pouze datový typ kořenového routeru. Z čehož plyne, že samotný klient neobsahuje žádné citlivé informace o serveru.

```
import type { BaseRouter } from './server';

const client = createTRPCProxyClient<BaseRouter>({
  ...
});
```

Výpis kódu 6 vytvoření tRPC klienta

Klient pro Next.js a React využívá knihovnu React query, která je jednou z knihoven z rodiny Tanstack query [47].

3.12 Zodios a TS-REST

Mezi populárními projekty jako GraphQL a tRPC existují i menší alternativy, jako je Zodios [49] a TS-REST [50]. Tyto projekty, stejně jako tRPC, umožňují definici schémat bez nutnosti generování kódu a využívají validačních schémat pro definování datových typů. Na rozdíl od tRPC, které pracuje s dvěma vrstvami – klientem a serverem, Zodios a TS-REST se snaží o trochu odlišný přístup. Operují se třemi vrstvami: klient, schéma a server. Na rozdíl od GraphQL se schéma nedá generovat a je nutné ho vytvořit manuálně. Zároveň žádný z nástrojů neumožňuje introspekci tohoto schématu. Často se tak distribuuje jako NPM balíček, což může být velmi nepraktické. Výhodou však je možnost vytvořit validace a typovaní nad již existujícím REST API, a to bez nutnosti generování kódu.

[15] Proxy je zástupný objekt, který umožňuje předefinovat základní operace daného objektu [48].

3.13 SDK

SDK je sada nástrojů pro tvorbu software určité platformy [51]. V kontextu webových API se může použít pro odstínění nevýhody netypovaného rozhraní. Vytvořením zástupného kódu napsaného v určitém programovacím jazyce lze na klientovy maskovat nevýhody spojené s netypovaným API, protože typování dodává SDK. Nevýhody tohoto přístupu jsou podobné jako ty u Zodios a TS-REST popsané v předchozí podkapitole. Tvorba SDK vyžaduje manuální vývoj rozhraní, které se pak distribuuje, v kontextu jazyka TypeScript, pomocí NPM balíčků. Výhodou tak opět je možnost vytvořit typovaní nad existujícím netypovaným API, nevýhodou však distribuce typovaní a náročnost tvorby. Přístup tRPC nebo GraphQL vytváří v tomto kontextu „SDK“ nad službou automaticky a knihovna této práce se bude snažit o stejně automatický přístup.

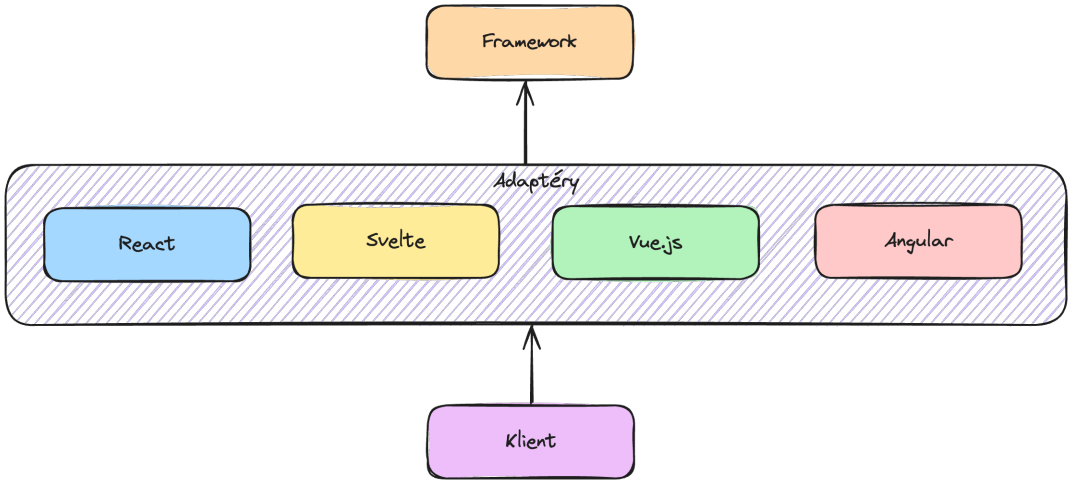
3.14 Návrhové vzory

Návrhové vzory představují architektonický návrh pro běžně vyskytující se problémy návrhu v určité situaci. Obsahují osvědčená řešení a návod k jejich použití. Na rozdíl od algoritmu, který popisuje postup k cíli jasně danými kroky, návrhový vzor popisuje řešení problému na vyšší úrovni. Implementace návrhových vzorů v různých systémech se tedy může lišit. Framework na rozdíl od vzoru představuje konkrétní řešení a může být implementován pomocí několik instancí více návrhových vzorů.

Následující popisované návrhové vzory jsou buď velmi obecné koncepty nebo vychází především z vzorů označovaných jako GoF (Gang of Four), které jsou popsány v knize Design Patterns, jenž byla vytvořena čtyřmi autory: Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides. Návrhové vzory GoF rozdělujeme do tří skupin: vytvářející, strukturální a návrhové vzory chování. Vytvářející vzory popisují mechanismy pro efektivní vytváření objektů. Strukturální vysvětlují, jak sestavovat objekty do větších komponent nebo struktur systému. Návrhové vzory chování se pak týkají přidělování odpovědností mezi objekty.

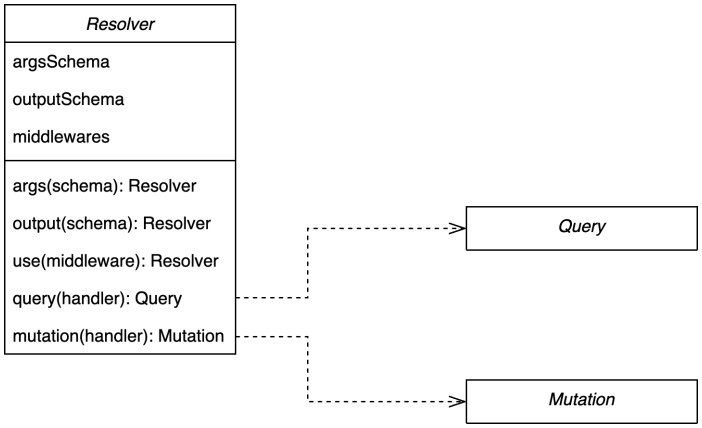
Pro vývoj knihovny, která bude výstupem této práce, jsou některé návrhové vzory velmi důležité a je třeba se jimi zabývat. Následující vzory popisují návrh řešení různých problémů při tvorbě knihovny a vysvětlení uplatnění daného vzoru pro určitou problematiku.

Jedním z klíčových strukturálních návrhových vzorů pro tuto práci je adaptér. Ten umožňuje spolupráci různých objektů s navzájem nekompatibilním rozhraním. Pro tuto práci může být tento vzor uplatněný zejména v podobě adaptérů pro frontendové frameworky. Každý frontendový framework definuje své vlastní rozhraní pro udržení stavu aplikace a nevytvářejí jednotný způsob pro dotazování se serveru. Adaptér tak poskytne rozhraní pro daný framework při použití základního modulu pro komunikaci serveru a klienta. Práce s knihovnou tak bude mnohem snazší a rozhraním bude klient zapadat do daného ekosystému frameworku. Tento vzor velmi souvisí s kompatibilitou a možností nasadit knihovnu v různých prostředích a ekosystémech (obrázek 3.2). Samotný adaptér pak může představovat knihovna Tanstack query, která vytváří ucelený způsob v komunikaci se serverem pro mnoho frontendových reaktivních frameworků.



Obrázek 3.2 Využití adaptérů klienta pro frontendové frameworky

Dalším vzorem vhodným pro implementaci knihovny je builder. Tento vytvářející návrhový vzor umožňuje tvorbu komplexních objektů krok za krokem a pomocí stejného konstrukčního kódu tvoří různé objekty. tRPC implementuje builder pro sestavení jednotlivých procedur, hlavním důvodem je vhodné použití builderu s typováním v TypeScriptu. Stejné důvody využívá pro sestavování validačních schémat knihovna Zod i Yup. Datové typy se totiž nedají přepisovat na rozdíl od proměnné, pro zajištění úplného typování je tedy neustále nutné vytvářet nové neměnné struktury. Neměnnost (immutability) je základním návrhovým vzorem, který popisuje chování objektů a to tak, že po jejich vytvoří je již dále nelze měnit. V samotné knihovně je vhodné využít neměnného builderu pro postupné sestavování endpointů API, čímž bude zajištěna naprostá typová bezpečnost díky možnosti neustálé změny datových typů pomocí tvorby neměnných objektů. Toto řešení navíc podporuje typově bezpečnou úpravu kontextu požadavku v middleware, kterou implementuje tRPC. Komponentu, která se bude starat o formování koncových bodů pojmenujeme „resolver“ (obrázek 3.3).



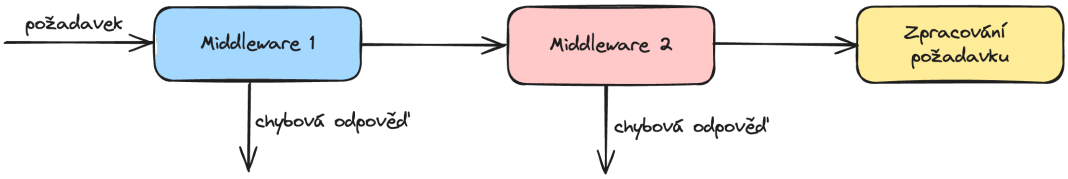
Obrázek 3.3 Návrh neměnného builderu endpointů

Je důležité zdůraznit, že aby vše bylo otypované, je nutné, aby resolver definoval generické typy pro všechny atributy. Jen takovým způsobem je možné, aby při změně části kódu došlo k okamžitému přetypování objektu, což se poté ihned promítne na straně klienta.

Návrhový vzor composite se pak přímo nabízí pro tvorbu routerů. Composite je strukturální

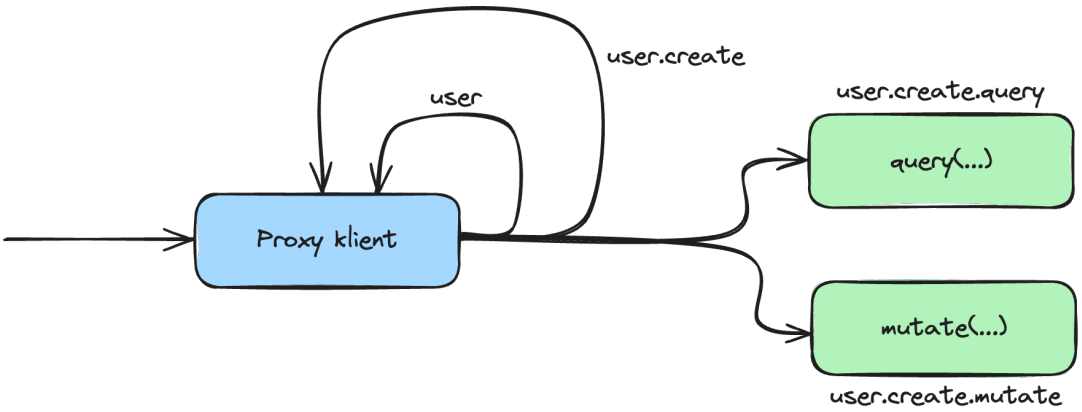
vzor pro skládání objektů do stromové struktury s kterými se následně pracuje jako se samostatným objektem. Tento vzor je vhodný pro vytváření routerů, které umožňují definovat strukturu služby pomocí rodičovských a podřízených uzlů. Samotný uzel pak bude buď opět typu routeru, tedy umožní zanoření další logické části API, nebo bude typu endpoint a bude tak definovat samotný bod otevřený pro dotazování. Tímto vytvářejí hierarchickou strukturu služby a vždy bude existovat jeden kořenový router zastřešující celou aplikaci. Volání jednotlivých endpointů pak může fungovat rekurzivně, vždy začne u kořenového routeru a ten podle prvního fragmentu dotázané cesty přesune odpovědnost na podřízený uzel. Tento proces bude pokračovat, dokud uzlem nebude endpoint, který zpracuje požadavek a data odešle zpět v rekurzi. Nakonec to bude vypadat, že odpovídá pouze kořenový router a stromová struktura API se tak bude chovat jako jedna komponenta systému.

Pro možnosti efektivní autorizace pomocí middleware pak lze využít chain of responsibility. Tento návrhový vzor chování umožňuje předávat požadavky podél řetězce. Po přijetí požadavků se každý uzel řetězce rozhoduje, zda požadavek zpracuje nebo jej předá dalšímu uzlu. Při implementaci middleware jakožto řetězce je možné jednoduše definovat stupňující se omezení oprávnění. Navazující middleware tak již nemusí kontrolovat předešlá omezení (obrázek 3.4).



Obrázek 3.4 Řetězec middleware

Při inspiraci tRPC je možné na frontendu implementovat klienta knihovny pomocí zástupného objektu proxy. Strukturální návrhový vzor proxy vytváří zástupce objektu, který umožňuje provést nějakou akci před nebo po odesláním požadavku na původní objekt. tRPC toho využívá při sestavování dotazovaných cest na straně klienta. Díky znalosti datového typu kořenového routeru pak TypeScript nabízí při tomto sestavování typové nápovědy v rámci vývojového prostředí. Proxy objekt je zde velmi důležitý, umožňuje totiž maskovat chování prázdného objektu a postupně sbírat fragmenty dotazovaného endpointu (obrázek 3.5) [52, 53].



Obrázek 3.5 Proxy klient

3.15 Architektury

Aplikace se v průběhu času může aktualizovat a měnit. To je běžný proces všech webových, mobilních nebo desktopových aplikací. Při vývoji je tak velmi důležité dbát i na možné budoucí změny nebo rozšíření, aby tyto aktualizace byly co možná nejméně komplikované. Volba správné architektury podporuje možnost snadných úprav, čímž lze rychleji nasadit nové funkcionality a udržet si tak uživatele aplikace. Architektury rozdělíme do dvou skupin: dvouvrstvé a třívrstvé.

Dvouvrstvá architektura klient-server zahrnuje databázi a klienta. Celá aplikace se spouští na straně klienta a připojuje se k serveru. V klientské části se tak nachází veškerá obchodní logika a samotné uživatelské rozhraní. Klient má v tomto případě přímí přístup k databázi bez dalšího prostředníka.

Třívrstvá architektura systému klient-server zahrnuje databázi, server pro zpracování požadavků a klienta. V této architektuře klient zobrazuje pouze prezentační logiku, například uživatelské rozhraní. Díky tomu se snižují nároky na zdroje klienta, protože veškerá logika se nachází na serveru pro zpracování požadavků [54].

Jasným požadavkem na samotnou knihovnu je, aby nijak neomezovala použití třívrstvé architektury, která se využívá v moderních webových aplikacích a která podporuje škálování a snadnou údržbu i aktualizace systému.

Třívrstvá architektura může být dále specifikována na architekturu Model-View-Controller, která ještě více dbá na oddělení obchodní logiky od zobrazení. MVC specifikuje tři hlavní komponenty, které jsou díky rozdělení zaměnitelné, systém se tak stává velmi dobře upravitelný a migrovatelný. Díky této architektuře je možné nezávisle aktualizovat pouze jednu ze tří částí aplikace při zachování funkčnosti ostatních. Model reprezentuje, jaká data má aplikace obsahovat. View pak reprezentuje, jak se mají data zobrazit, v případě webové stránky jde nejčastěji o uživatelské rozhraní. Controller pak obsahuje logiku, která aktualizuje model a/nebo zobrazení v závislosti na uživatelském vstupu [55].

Dalším klíčovým požadavkem na knihovny generující silně otypované API tak je, aby umožnila implementaci aplikace pomocí architektury MVC. Díky tomu je umožněno tvořit pomocí knihovny škálovatelné modulární webové aplikace s možností aktualizace pouze jednotlivých částí.

3.16 Kompatibilita

V návrhu knihovny je nutno zvážit i fakt, že v ekosystému jazyka JavaScript existuje spousta frameworků jak na straně klienta, tak i serveru a je potřeba navrhnout řešení tak, aby bylo kompatibilní s většinou těchto technologií.

Mnoho full-stack frameworků, jako je Next.js[16] nebo SvelteKit[17], umožňuje tvorbu jak klientského, tak serverového kódu a obvykle nepoužívají Node.js http modul na straně serveru. Ve většině případů využívají file-system router. To je technika definování endpointů a stránek jak klientské, tak i serverové části pomocí struktury složek a souborů zdrojového kódu. Umožňují vytvořit REST, GraphQL, tRPC, nebo i jakékoliv jiné API. Žádný ze zmiňovaných nástrojů neumožňuje přímo vytvoření silně otypovaného API, usnadňují ale vývoj takového systému především kvůli možnosti jednoduchého sdílení kódu mezi serverem a klientem.

Většina Node.js knihoven pro tvorbu API, jako je Express, Koa nebo Fastify, má velmi podobné rozhraní, které vychází právě ze základního Node.js http modulu.

Kromě toho je třeba zajistit kompatibilitu s dalšími prostředími, jako jsou například Bun, Deno, Cloudflare Workers[18] nebo AWS Lambda[19]. Všechny totiž definují mírně odlišné rozhraní,

[16] Next.js je dostupný na webové stránce https://nextjs.org/.
[17] SvelteKit je dostupný na webové stránce https://kit.svelte.dev/.
[18] Služba Cloudflare Workers je dostupná na webové stránce https://workers.cloudflare.com/.
[19] AWS Lambda je dostupná na webové stránce https://aws.amazon.com/lambda/.

míra odlišnosti však není příliš veliká. Avšak ne všechna prostředí podporují různé standardní struktury a je třeba pro ně vytvořil polyfill.

Knihovna `@whatwg-node/server`[20] vytváří generický server, který lze nasadit téměř v jakémkoliv prostředí. Zároveň transformuje nestandardní objekty HTTP požadavku nebo odpovědi na standardní specifikované standardem WHATWG.

Dále je důležité podporovat kompatibilitu i na straně klienta, což může být realizováno například pomocí knihoven jako Tanstack query nebo jiných nástrojů pro správu HTTP požadavků určených pro různé frontendové nástroje. Je rovněž nutné zvážit, že většina frontendových frameworků podporuje vykreslování na straně serveru a přednačítání dat. Všechny tyto funkce jsou velmi důležité při tvorbě webových aplikací a knihovna nesmí znesnadňovat jejich použití.

3.17 Monorepo

Monorepo je úložiště, které obsahuje více souvisejících projektů. Jednotný repozitář přináší několik výhod jako je zjednodušená správa závislostí, lepší správa změn napříč projekty, snadné refaktorování, zjednodušení organizace jednotlivých projektů, lepší koordinace mezi vývojáři a lepší podpora pro sestavení a testování aplikací. Společné verzování projektů a možnost přidávat funkcionality napříč projekty bez nutnosti přepínání repozitářů je hlavní motivací monorepozitáře.

Mezi hlavní nevýhody patří především složitost konfigurace projektu nejen pro vývoj ale i nasazení jednotlivých aplikací. Další nevýhodou může být delší doba sestavení při provedení malých změn třeba jen v jednom z mnoha projektů monorepa [56].

Většina open-source projektů pro tvorbu webových aplikací jako je tRPC, React, Svelte, Prisma ORM a další využívají strukturu jednotného repozitáře. Monorepozitář přináší výhody v rozšiřitelnosti nástrojů přidáním dalších modulů nebo adapterů. Jednotný repozitář je ideální volbou pro tvorbu projektu této bakalářské práce, protože umožňuje udržovat stejné verze více modulů (klient, server) v jednom úložišti. Vývoj a nasazení by tak mělo být rychlejší a plynulejší.

Mezi nástroje pro správu jednotného repozitáře patří například Turborepo nebo Nx. Většina těchto nástrojů funguje na podobných principech, přičemž hlavním z nich je cachování obsahu repozitáře tak, aby se stejný neupravený kód nemusel sestavovat redundantně. Další nejdůležitější funkcionalitou, kterou tyto nástroje disponují, je možnost vytvářet závislosti mezi jednotlivými projekty uvnitř monorepozitáře. Tím velmi zjednoduší proces vývoje, protože nebude potřeba myslet na to, který projekt je nutné sestavit dříve a který později. Také umožňují hromadně spouštět příkazy z kořenového modulu na všechny ostatní, a to i paralelně. Práce s mezipamětí však vyžaduje pečlivé nastavení, například invalidaci dat v mezipaměti při úpravě globálních závislostí jako jsou environmentální proměnné. Nx pak navíc umožňuje spouštět příkazy distribuovaně a tím rozdělit zátěž výpočtů mezi více počítačů, to však pro vývoj této práce není příliš podstatné.

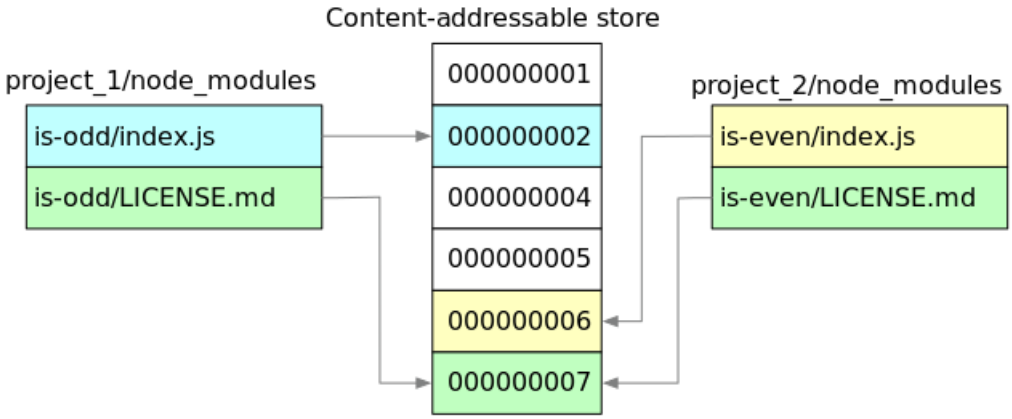
V porovnání jsou oba nástroje velmi podobné, Nx určen spíše pro tvorbu komplexních aplikací při mírně složitější konfiguraci, Turborepo pak pro středně velké projekty s téměř nulovou konfigurací [57, 58].

[20] Knihovna `@whatwg-node/server` je dostupná na webové stránce https://github.com/ardatan/whatwg-node.

3.18 PNPM

PNPM je rychlý a úsporný správce balíčků pro JavaScriptové projekty. Je alternativou k populárním správcům NPM nebo Yarn a je navržen tak, aby optimalizoval instalaci a správu závislostí v projektech. Podporuje také strukturu projektu jakožto jednotného repozitáře.

Na rozdíl od tradičních správců balíčků používá jediné globální úložiště pro všechny balíčky napříč projekty. To znamená, že pokud se více projektů spoléhá na stejnou závislost, je uložena pouze jednou, což eliminuje duplikaci a šetří místo na disku. Nejen, že PNPM neprovádí instalaci stejných závislostí, dokonce neduplikuje ani totožné soubory mezi různými závislostmi (obrázek 3.6).



Obrázek 3.6 PNPM – ukládání závislostí na disk [59]

PNPM využívá pevné a symbolické odkazy k těmto optimalizacím využití disku a zrychlení instalací. Tím, že se soubory modulů stahují a ukládají pouze jednou a jsou využívány odkazy k jejich používání napříč projekty, může PNPM výrazně snížit potřebné množství místa na disku, především ale razantně zrychlit instalaci závislostí. Kromě toho poskytuje funkce pro paralelní instalace, která může urychlit proces prováděním několika instalací najednou.

PNPM workspace je nástroj, který umožňuje spravovat více projektů v jednom pracovním prostoru, například uvnitř jednotného repozitáře. S pomocí jednoho konfiguračního souboru dokáže spravovat závislosti napříč celým monorepozitář [59].

Kapitola 4
Knihovna PTSQ

Podrobný popis konceptů a principů knihovny, která byla vytvořena v rámci bakalářské práce. Kapitola zachycuje základní myšlenky a mechanismy, které stojí za jejím fungováním. Zároveň vyobrazuje tvorbu API a odpovídá na základní otázky při tvorbě webové služby pomocí knihovny, jako je logování, testování nebo autorizace uživatelů.

4.1 PTSQ

PTSQ je knihovna, která usnadňuje vytváření silně otypovaných otevřených API. Díky tomu, že je napsaná v TypeScriptu poskytuje robustní typovou bezpečnost a zároveň umožňuje snadnou práci s datovými typy, včetně jejich přenosu mezi serverem a klientem.

PTSQ je sada modulů a nástrojů pokrývající serverovou i klientskou část pro vytváření silně otypovaného API. Tyto moduly jsou snadno dostupné prostřednictvím služby NPM. Protože v modulech převažuje typový kód TypeScriptu, jsou malé v běhovém prostředí, což zajišťuje velmi efektivní a optimalizované využití a nasazení.

Inspirací pro vývoj knihovny byly již existující projekty, které se zaměřují na řešení stejného problému, jako GraphQL, tRPC, Zodios a TS-REST. PTSQ se snaží sjednotit nejlepší vlastnosti těchto projektů a eliminovat jejich nedostatky. Zatímco tRPC nabízí vysokou typovou bezpečnost bez nutnosti generování kódu, postrádá introspekci schématu, což omezuje jeho použitelnost pro tvorbu silně otypovaných otevřených API. GraphQL poskytuje introspekci schématu, ale pro typově bezpečný kód vyžaduje generování kódu jak na straně klienta, tak i serveru.

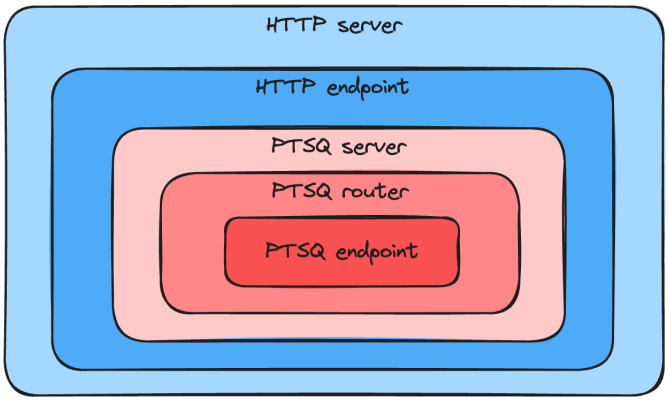
PTSQ kombinuje snadnost použití a jednoduchost vývoje projektu podobnou tRPC s možností vytvoření otypovaného otevřeného API pomocí introspekce schématu jako nabízí GraphQL. Díky tomu je PTSQ výkonným a všestranným nástrojem pro tvorbu silně typovaných otevřených API, který minimalizuje obtíže spojené s vývojem a údržbou komplexních systémů.

Knihovna spoléhá na TypeScript a odvozování typů, proto je nutné pro správné fungování knihovny nastavit překladač v souboru `tsconfig.json`. Nastavení `strictNullChecks` nebo `strict` musí mít hodnotu `true`. To je nezbytné pro správné odvozování typů TypeScriptu z validačních schémat. Nastavení způsobuje přísné definování nullable typů. Při výchozí hodnotě `false` není možné určit, zda je proměnné povoleno přiřazovat hodnoty `null` nebo `undefined`.

Knihovnu je možné nainstalovat pomocí libovolného správce závislostí, ať už je to NPM, Yarn, PNPM nebo Bun. Všechny moduly knihovny začínají prefixem `@ptsq`, který označuje NPM organizaci modulů. Po názvu organizace následuje název samotného modulu. Tento jednotný formát názvů usnadňuje identifikaci a použití jednotlivých modulů v rámci projektu. Díky tomu jsou moduly přehledně organizované a snadno rozpoznatelné, což usnadňuje práci s knihovnou.

4.2 Vytvoření PTSQ instance

Funkce `ptsq` vytvoří builder PTSQ serveru, který reprezentuje instanci knihovny, samotná aplikace pak může vytvořit a využívat několik instancí. PTSQ samotné pak nenabízí žádnou možnost vytvoření HTTP serveru a podobně jako GraphQL nebo tRPC využívá pouze jeden HTTP endpoint (obrázek 4.1). Knihovna poskytuje posluchače (listenery) pro HTTP servery vytvořené externími moduly, jako jsou například Node.js http modul, Express, Fastify nebo Koa.



Obrázek 4.1 PTSQ server nasazený uvnitř HTTP endpointu

Tato modularita umožňuje uživatelům vybrat nejvhodnější framework pro jejich konkrétní potřeby a preferovaný styl psaní kódu. Kromě toho PTSQ nabízí posluchače i pro serverless prostředí, jako jsou AWS Lambda nebo Cloudflare Workers. Knihovnu tak lze využívat v kombinaci s moderními serverless technologiemi a vytvářet škálovatelné a efektivní aplikace bez potřeby správy vlastního serveru. Díky své kompatibilitě s full-stack frameworky pro tvorbu webových aplikací, jako je Next.js nebo SvelteKit, je PTSQ ideální volbou pro projekty, které vyžadují kompletní řešení frontendu a backendu. Současně knihovna poskytuje možnost vytvořit posluchače pro prostředí Bun nebo Deno. Zajištěním flexibility, tedy možností nasazení knihovny v různých prostředích a ekosystémech, podporuje knihovna širokou škálu vývojářských scénářů a potřeb.

Kvůli obrovské kompatibilitě je nutné PTSQ server správně nakonfigurovat, především pomocí nastavení `endpoint` a `fetchAPI`. Tyto parametry ovlivňují chování HTTP endpointu uvnitř použitých frameworků vytvářejících HTTP server. Nastavení `endpoint` určuje cestu, na které bude PTSQ server přijímat požadavky. Výchozí hodnota tohoto nastavení je `/ptsq`, což znamená, že server bude poslouchat na HTTP endpointu `/ptsq`, například `http://localhost:3000/ptsq`.

Pokud je PTSQ server vložený uvnitř nějakého endpointu HTTP API, například `/api`, nastavení `endpoint` to musí respektovat a jeho hodnota tak musí být nastavena na `/api/ptsq`. To je nutné zejména při použití PTSQ s full-stack frameworky jako Next.js nebo SvelteKit, kde je možné aplikaci nasadit ve file-system routerech, které tyto frameworky poskytují. Při tomto nastavení jsou automaticky odstraněna přebytečná lomítka, takže není důležité, zda je hodnota nastavena jako `/api/ptsq` nebo `api/ptsq`.

PTSQ server samotný pracuje pouze s HTTP metodami POST a GET. Při požadavku s ostatními metodami na využívaný HTTP endpoint serveru je automaticky vrácena chybová odpověď 405, `Method not allowed`. Metoda POST odkazuje na samotný server a volání dotazů a mutací. GET vrací introspekci schématu ve formátu JSON. Pro introspekci je automaticky vyhrazen stejný HTTP endpoint jako ten, který používá PTSQ server. S nastavením `endpoint` by tedy požadavek na introspekci mohl vypadat jako `GET http://localhost:3000/api/ptsq`.

Nastavení `fetchAPI` pak určuje objekty odpovídající požadavku, odpovědi a dalším objektům standardu WHATWG jako je `URL`, `fetch`, `FormData` a další. Výchozí hodnoty jsou z polyfillů

knihovny `@whatwg-node/fetch`[1]. Toto nastavení se však standardně nevyužije. Využití má v případě, že by framework, ve kterém je PTSQ nasazeno, poskytoval vlastní polyfill pro některé tyto standardní objekty.

Pro vytvoření samotných komponent PTSQ serveru je nutné v builderu serveru zavolat metodu `PtsqServerBuilder.create`. Ta vytvoří 3 hlavní komponenty, tedy `resolver`, `router` a funkci pro nasazení aplikace `serve` (kód 7).

```
import { ptsq } from '@ptsq/server';

const { resolver, router, serve, ptsqEndpoint } = ptsq({
  endpoint: '/api/ptsq',
}).create();
```

Výpis kódu 7 Vytvoření komponent PTSQ

4.3 Kontext požadavku

Podobně jako GraphQL a tRPC, i PTSQ umožňuje vytvářet vlastní kontext požadavku. Kontext je objekt, který se vytváří při každém požadavku na server PTSQ, ne však při dotázání na introspekci schématu. Uvnitř kontextu je možné uchovávat informace, které jsou přístupné v rámci všech middleware, dotazů a mutací, jako například aktuálně přihlášený uživatel, připojení k databázi nebo aktuální HTTP požadavek.

Samotný kontext se vytváří uvnitř funkce, která může být jak synchronní, tak i asynchronní. Tato flexibilita umožňuje zpracovat nějakou jinou asynchronní operaci před vytvořením samotného kontextu, například vytvoření připojení k databázi nebo dekódování JWT[2] tokenu uživatele (kód 8). Datový typ kontextu je pak automaticky odvozen z návratové hodnoty této funkce a je později dostupný ve všech dotazech, mutacích a middleware. Díky uchovávanému typu kontextu vývojář stále přesně ví, jaké hodnoty kontext požadavku obsahuje a jakých může nabývat, což výrazně minimalizuje chybovost a zvyšuje efektivitu práce při vytváření celé serverové části.

Do funkce vytvářející kontext vstupuje jeden objektový parametr, ve kterém je vždy obsažena vlastnost `request` typu `Request`, což je standardní objekt WHATWG reprezentující HTTP požadavek. Parametr vstupující do tvorby kontextu může být dále rozšířen o další vlastnosti, typicky to mohou být objekty vytvořené frameworkem ve kterém je PTSQ server integrován. Knihovna také podporuje vytvoření vlastních parametrů pro inicializaci kontextu. To navazuje na samotné nasazení aplikace, které je pak popsáno v podkapitole 4.14.

```
ptsq({
  ctx: async ({ request }) => {
    const db = await createConnection();
    const user = await jwt.decode(request);

    return {
      user,
      db
    };
  }
});
```

Výpis kódu 8 Vytvoření kontextu požadavku

[1] Knihovna `@whatwg-node/fetch` je dostupná na webové stránce https://github.com/ardatan/whatwg-node.
[2] JWT je bezpečný prostředek pro přenášení zakódovaných dat ve formátu JSON mezi klientem a serverem [60].

4.4 Pluginy a CORS

Plugin představuje možnost dodatečně upravit požadavek a odpověď PTSQ serveru na velmi nízké úrovni, přičemž nepracuje s vlastním kontextem požadavku, pouze se standardními objekty požadavku a odpovědi. Obaluje přijímání a odesílání zpráv pomocí metod `onRequest` a `onResponse`. Pluginy zajišťuje knihovna `@whatwg-node/server`, která je použita pro vytvoření posluchače HTTP serveru a jediný plugin, který poskytuje přímo knihovna PTSQ slouží pro nastavení CORS. Samotné pluginy nejsou příliš podstatné a vlastní plugin uživatel knihovny vytvoří velmi zřídka.

4.5 Validace a validační schémata

Hlavním konceptem PTSQ je vytvoření celkově bezpečného API nejen na typové úrovni, ale i v běhové. Validační schéma je nástroj pro definování struktury a typů dat, na základě kterého jsou pak reálná data validována. V PTSQ se pro definování validačních schémat využívá knihovny Typebox. Ta umožňuje vytvořit JSON schéma za pomocí builderu s možností statického odvození typu validních dat. Schéma tedy validuje data na úrovni běhové, zároveň však poskytuje typový předpoklad dat na úrovni typů. Díky tomu, že jsou pro validaci využita JSON schémata, lze je zároveň velmi jednoduše a efektivně využít k introspekci schématu pro vytvoření otevřeného silně otypovaného API. Protože introspekce poskytuje schéma služby ve formátu JSON, stačí tak vrátit pouze definici samotného validačního schématu. Jedná se tak o „single source of truth“. Typebox umožňuje vytvářet i složité validační konstrukty jako rekurzivní schémata, sloučení nebo průnik dvou schémat, indexace schémat objektů nebo polí a další. Také poskytuje komplexní validace řetězců pomocí formátu, který musí být před použitím zaregistrovaný. Pro registrace formátů lze použít jmenný prostor `FormatRegistry`, který knihovna poskytuje. Formát je pak funkce, která pro správné vstupy vrací hodnotu `true` (kód 9).

```
FormatRegistry.Set('foo', (value) => value === 'foo');

Type.String({
  format: 'foo'
});
```

Výpis kódu 9 Registrování formátu pro validace řetězců

Všechna JSON schémata jsou pak navíc kompilována kompilátorem, který převede schéma na jednoduchou JavaScript validační rutinu (funkci), která validuje data mnohem rychleji než validace oproti samotnému schématu. Tyto validační funkce nejsou nijak ukládány, a tak je nutné na každý požadavek schéma před validačním procesem zkompilovat. Protože samotný krok kompilace je vcelku náročný, knihovna PTSQ podporuje vytvoření vlastní mezipaměti pro zkompilovaná JSON schémata, čímž eliminuje nevýhodu pomalého překladu.
Modul `@ptsq/cached-json-schema-parser` umožňuje vytvořit in-memory cache[3] pro přeložené validační funkce. Využívá líné kompilace, tedy JSON schéma je přeloženo pouze při prvním spuštění validace nad tímto schématem. Při dalších validacích pomocí daného schématu je využita validační rutina uložena v mezipaměti. Je důležité upozornit, že schémata se odlišují referencí, nikoliv hodnotou, dvě různá schémata se stejnou strukturou jsou tak zkompilována dvakrát.

[3] In-memory cache ukládá data do paměti RAM a tato data využívá jako mezipaměť.

Validátor nebo parser schémat lze nastavit při vytváření PTSQ instance a lze ho upravit tak, aby používal i jiné kompilátory a validátory, například populární Ajv[4] (kód 10). Typebox schémata jsou kompatibilní s jakýmkoliv nástrojem podporující JSON Schema Draft 7 specifikaci[5] [61].

```
ptsq({
  parser: {
    encode: ({ value, schema }) => {...},
    decode: ({ value, schema }) => {...},
  }
});
```

Výpis kódu 10 Nastavení parseru

Parser obsahuje 2 části, decode, který slouží k validaci argumentů, a encode, který slouží k validaci výstupů. Obě části určují, zda je hodnota `value` validní oproti schématu `schema`. Důvod rozdělení do dvou částí je možnost vytváření transformací na úrovni validačních schémat.

Transformace mohou sloužit k serializaci a deserializaci hodnot. Při odeslání požadavku na PTSQ API lze například datum a čas poslat pouze ve formě řetězce, protože JSON formát nepodporuje JavaScriptový objekt `Date` nebo jiný podobný objekt pro reprezentaci datumu a času. Na server tak dojde hodnota pouze ve formátu řetězce, to je však velmi nepraktické pro jakoukoliv manipulaci s časem, zároveň je velmi nevhodné deserializovat nebo serializovat hodnotu přímo uvnitř zpracování požadavků. Transformace umožňují vytvořit funkce pro serializaci i deserializaci přímo v rámci validačních schémat a abstrahovat tak úpravu dat ze samotné funkce pro zpracování. Vytváří tím velmi přehledné rozhraní, které není příliš náchylné k chybám, protože kódu v samotném zpracování požadavků bude obecně méně a bude čitelnější díky absenci nutnosti přípravy dat. Tato vlastnost je inspirována jak vlastními skalárními typy GraphQL, tak tRPC transformátory. Funkci zpracovávající požadavek to umožňuje přijímat JavaScriptové objekty a třídy a zároveň takové typy a konstrukty i vracet. Argumenty jsou automaticky deserializovány, výstupy pak serializovány před odesláním dat na klienta.

```
Type.Transform(Type.Number())
  .Decode(timeStamp => new Date(timeStamp))
  .Encode(date => date.getTime());
```

Výpis kódu 11 Ukázka transformace z číselného časového razítka na objekt `Date` a obráceně

Před deserializací automaticky dochází k validaci dat podle validačního schématu, stejně tak se děje i po serializaci.

Při transformacích je důležité dbát na to, že většina externích validátorů nebo kompilátorů jako Ajv nepodporují automatickou deserializaci či serializaci hodnot. Po samotném validačním procesu je tak důležité při tvorbě vlastního parseru využít Typebox funkcí pro aplikaci transformací.

[4] Ajv je JavaScriptový validátor umožňující JSON schémata kompilovat do validačních rutin. Je dostupný na webové stránce https://ajv.js.org/.

[5] Přehled specifikací JSON schémat je dostupný na webové stránce https://json-schema.org/specification-links.

4.6 Resolver

Jednou ze tří komponent PTSQ serveru je resolver. Ten je zodpovědný za vytváření dotazů a mutací a umožňuje přidávat validační schémata argumentů a výstupů. Resolver funguje jako builder, který formuje konečný PTSQ endpoint.

Metoda `Resolver.args` umožňuje přidat argumenty k resolveru. Při použití metody se vytvoří nový resolver s validačním schématem, původní resolver zůstane nezměněn. Operace přidání argumentů je neměnná především z důvodu nutnosti registrace datových typů. Ty totiž nepodporují žádnou možnost přepisování, takže je vždy nutné vytvořit datový typ nový. Argumenty jsou specifikovány pomocí JSON validačních schémat, vytvořených pomocí Typebox builderu. Díky možnosti odvození TypeScript typů z těchto schémat je zajištěno vytváření typově bezpečných argumentů. Zároveň je však zajištěna bezpečnost i za běhu, a to pomocí ověření a validace na základě JSON schémat. Metodu `Resolver.args` lze volat opakovaně, čímž lze argumenty řetězit. K řetězení může docházet kdekoliv v procesu formování endpointu a zřetězená validační schémata jsou pak spojena pomocí `allOf`. Toto klíčové slovo JSON schématu určuje, že data jsou validní jen tehdy, pokud jsou validní oproti všem schématům z definovaného seznamu.

```
const resolverWithFirstName = resolver
  .args(
    Type.Object({
      firstname: Type.String()
    })
  );
```

Výpis kódu 12 Vytvoření resolveru s argumenty

Výhodou řetězení je možnost kdykoliv v řetězci resolver nahradit jiným, již připraveným, a tak vytvářet znovupoužitelné komponenty (kód 13). Vůbec nevadí, pokud se některé z částí zřetězených schémat překrývají. Data totiž musejí být validní oproti všem zřetězeným schématům najednou, překrýváním tak nedochází k žádným nesrovnalostem. Pokud by během překrývání došlo k neshodě typů stejné vlastnosti objektu, data by nikdy nebyla validní.

```
resolverWithFirstName
  .args(
    Type.Object({
      lastname: Type.String()
    })
  );
```

Výpis kódu 13 Přidání argumentů k předpřipravenému resolveru

Argumenty lze řetězit, dokud se z resolveru nezformuje endpoint. Ten pak získá přístup k objektu `input`, který bude typově bezpečný a jeho typ bude přesně odpovídat datům, validním při vstupním validačním JSON schématu včetně aplikovaných transformací. Tímto způsobem vývojář nikdy neztratí přehled o očekávaném vstupu endpointu, což minimalizuje chybovost a neefektivitu. Pokud by validace vstupu za běhu vyhodnotila data jako nevalidní, server automaticky vrátí HTTP odpověď 400, Bad request s PTSQ kódem chyby `PTSQ_VALIDATION_FAILED`.

PTSQ server pak automaticky validuje tělo každého požadavku oproti základní struktuře, jako je název a typ endpointu. Pokud by se tato validace vyhodnotila jako neplatná, server také automaticky vrátí HTTP odpověď 400, Bad request s PTSQ kódem chyby `PTSQ_BODY_PARSE_FAILED`.

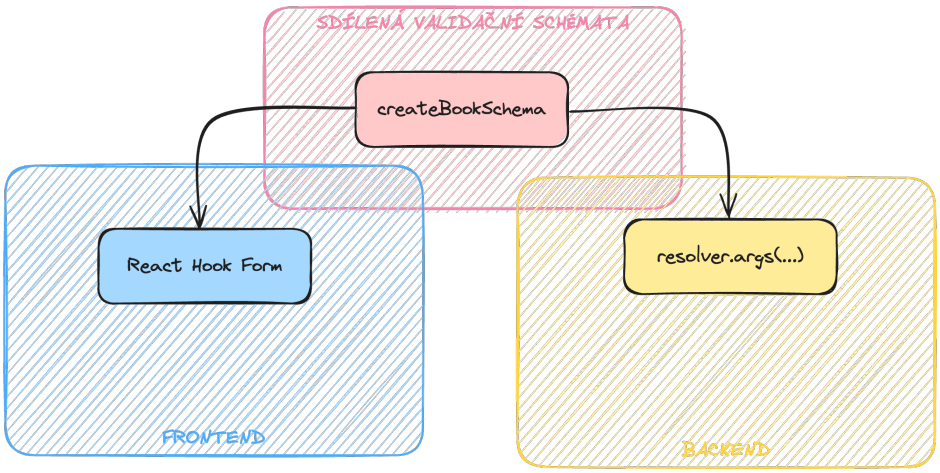
Kromě argumentů má resolver také možnost přidávat validační schémata výstupů (kód 14). Ty lze řetězit naprosto totožným způsobem a stejně jako přidání argumentů i přidání výstupů je neměnnou operací, původní resolver tak zůstane nezměněn. Validační schéma výstupu kontroluje nejen datový typ návratové hodnoty funkce (handleru), která představuje finální zpracování požadavku, ale také provádí validaci dat výstupu v běhovém prostředí. Pokud by výstup nebyl platný, server vrátí HTTP odpověď 500, Internal Server Error s PTSQ kódem chyby `INTERNAL_SERVER_ERROR`.

```
resolver
  .output(
    Type.Object({
      firstname: Type.String()
    })
  );
```

Výpis kódu 14 Vytvoření resolveru s validací výstupu

Při situaci nevalidního vstupu nebo výstupu jsou v chybové odpovědi serveru, která je vysvětlena v podkapitole 4.10, automaticky zaznamenány informace o samotné validaci. Tyto informace jsou uloženy ve vlastnosti chyby `cause` a zdůvodňují proč jsou data oproti danému schématu nevalidní.

Díky tomu, že je jazyk TypeScript při práci s knihovnou využit jak na straně serveru, tak i klienta, otevírají se tak možnosti sdílení validačních schémat mezi frontendem a backendem a tím i možnost tvorby dvojí validace. Protože validační schémata jsou nutná pro definici argumentů i výstupů na straně serveru, použití schématu na frontendu je tak bez vynaložení většího úsilí. Tímto knihovna umožňuje naprosto snadnou tvorbu optimalizace síťové komunikace v podobě dvojí validace, čímž napomáhá tvorbě škálovatelných aplikací (obrázek 4.2). Různé knihovny pro tvorbu dynamických formulářů, jako React Hook Form, podporují validace na základě Typebox validačních schémat.



Obrázek 4.2 Sdílené validační schéma pro dvojí validaci

K resolveru je dále možné přidat popis, což je opět operace, která nemění původní objekt. Popis je dostupný na klientovi při dotazování se na daný PTSQ endpoint. Protože resolver může mít pouze jeden popis, při řetězení je vždy předchozí přepsán novou hodnotou. Popis musí být dostupný i na úrovni typů TypeScriptu, vytvoření popisu s typem řetězce tak nemá žádný význam. Typ řetězce musí být konstantní, typu literál (kód 15).

```
const correctDescription = "Correct description.";
resolver.description(correctDescription);

let wrongDescription = "Wrong description.";
resolver.description(wrongDescription);
```

Výpis kódu 15 Správné a nesprávné použití popisu resolveru

Zde je ještě velmi důležité zdůraznit, že všechny tyto definované operace musejí neustále vytvářet generický datový typ uvnitř třídy `Resolver`. Jen díky tomu je totiž možné z validačních schémat odvozovat datový typ vstupu nebo výstupu. Definování validačních schémat nebo popisu resolveru bez generických typů by je neumožnilo dále využívat na typové úrovni. Tím by vývojář přišel o znalost typů vstupů a výstupů na straně serveru nebo o znalost těchto informací na straně klienta. Práce s generickými typy je v tomto případě velmi zásadní a vytváří tak způsob sdílení datových typů s klientem. Zároveň umožňuje pevné definování struktury a kontroly dat ve vývojovém prostředí.

Pomocí metody `Resolver.use` je možné přidat k resolveru middleware. Ta bude mít přístup k vstupu vytvořeném v resolveru a také ke kontextu požadavku. Při přidání middleware je opět vytvořen nový resolver a původní zůstává nezměněn. Middleware je pak jediné místo, kde je možné dodatečně upravit kontext nebo vstup požadavku.

Popis middleware a jejich hlavní využití a možnosti jsou pak více popsány v podkapitole 4.12. Zde je popsán pouze koncept připojení middleware k resolveru.

```
resolver
  .use(({ ctx, next, input }) => {
     if(!ctx.user) throw new PtsqError({ code: 'UNAUTHORIZED' });

    return next({
      ctx: {
        user: ctx.user
      }
    });
  });
```

Výpis kódu 16 Vytvoření resolveru s middleware

Proměnná `input` v middleware definované v ukázce kódu 16 má hodnotu `undefined`. Resolver, který middleware vytváří, totiž před její tvorbou nedefinuje validační schéma argumentů, není tak známý žádný typ vstupu. Protože nutnost definovat schéma argumentů před definicí middleware je vcelku omezující a resolver tím ztrácí znovupoužitelnost, je možné definovat pouze část vstupu (kód 17).

```
const bookMutationResolver = resolver
  .args(
    Type.Object({
      id: Type.String()
    })
  )
  .use(({ input, ctx, next }) => {
    // ...
  });
```

Výpis kódu 17 Vytvoření resolveru s middleware a argumenty

Parametr `id` však nemusí být jediným vstupem, který je zapotřebí pro provedení celé operace nějakého endpointu, který tento resolver zformuje. Zbytek vstupu tak lze dodefinovat později pomocí již vysvětleného řetězení argumentů. Definovaný resolver `bookMutationResolver` se tak stává znovupoužitelnou komponentou serveru a lze využít pro tvorbu mnoha různých endpointů se společnou logikou definované middleware, kterou si tento resolver nese s sebou (kód 18).

```
const updateBook = bookMutationResolver
  .args(
    Type.Object({
      title: Type.Optional(Type.String()),
      content: Type.Optional(Type.String())
    })
  )
  .output(...)
  .mutation(...);
```

Výpis kódu 18 Rozšíření argumentů po definici middleware

Styl řetězení je v tomto kontextu silně inspirován knihovnou tRPC, z části pak i validačními knihovnami Zod a Yup. Tento přístup definování neměnných builderů je ideální pro vytváření typově bezpečných a znovupoužitelných komponent serveru. Implementací tohoto stylu knihovna podporuje velkou modularitu. Samotný vzor rozdělení logiky do několika komponent je velmi efektivní, což pak v trochu jiném světě dokazují i frontendové reaktivní frameworky.

4.6.1 Vytvoření samostatného resolveru

Doteď popisovaný resolver byl vždy závislý na instanci serveru. Nebylo tak možné vytvořit externí komponentu, například jako knihovnu nebo modul. Samostatný resolver tak představuje možnost tvorby nezávislé komponenty na instanci PTSQ serveru. Pomocí statické metody `Resolver.createRoot` je možné vytvořit externí resolver a nadefinovat typ kontextu požadavku, který tento resolver vyžaduje. Vytvořením nezávislého resolveru knihovna podporuje vytváření samostatných komponent, které umožňují tvorbu různých knihoven založených na PTSQ. Samostatný resolver může, stejně jako klasický, vytvářet konečné endpointy aplikace a tím ho lze připojit k jakékoliv instanci PTSQ. Pro připojení k aplikaci lze také využít řetězení resolverů, které je popsáno níže. Při připojování samostatného resolveru stačí, aby typ kontextu požadavku instance serveru rozšiřoval vyžadovaný typ kontextu samostatného resolveru. Kontext aplikace tedy může obsahovat více vlastností, musí ale pokrývat vlastnosti definované samostatným resolverem. Pokud toto omezení neplatí, TypeScript na úrovni typů nepovolí připojení resolveru k instanci serveru.

```
const standaloneResolver = Resolver.createRoot<{ user?: User }>();
```

Výpis kódu 19 Vytvoření samostatného resolveru

Pro připojení samostatného resolveru definovaného v kódu 19 musí kontext požadavku instance PTSQ serveru obsahovat hodnotu `user` stejného datového typu, může však zároveň obsahovat i další jiné hodnoty.

4.6.2 Řetězení resolverů

Resolvery, jakožto komponenty, je možné mezi sebou řetězit pomocí metody `Resolver.pipe`. Ta vytvoří nový resolver a přitom žádný z původních nijak neupraví. Při řetězení dojde k sjednocení všech vstupních i výstupních validačních schémat a výsledný resolver využije všechny middleware prvního i druhého resolveru. Protože při definovaní middleware je důležité pořadí, v jakém se spouštějí, při sjednocení se middleware druhého resolveru v operaci řetězení přidají za middleware prvního. To zároveň způsobí, že datový typ kontextu požadavku bude velmi ovlivněn poslední middleware druhého resolveru, ta je totiž spuštěna jako poslední. Může se tak stát, že datový typ kontextu, který vrací poslední middleware druhého resolveru, přepíše překrývající se typy poslední middleware prvního resolveru. Popis resolveru je při řetězení vždy nastaven na hodnotu `undefined`, tedy není přenesen ani z jednoho ze spojovaných resolverů.

Díky řetězení je možné vytvářet serverové komponenty v podobě resolverů a ty uvnitř aplikace propojovat. To umožňuje vytvořit několik částí aplikace, které řeší svoji logiku, a při řetězení logiku propojit a vytvořit tak komplexní systém (kód 20). Zároveň díky samostatnému resolveru lze připojit i externí komponenty, které jsou vytvořené bez instance PTSQ serveru, čímž je umožněno připojovat jiné knihovny a moduly.

```
loggingResolver
  .pipe(loggedInResolver)
  .args(...)
  .output(...)
  .pipe(argsDeepTrimResolver)
  .output(...);
```

Výpis kódu 20 Řetězení více resolverů s řetězením validačních schémat

4.7 Vytvoření endpointu

Jak již bylo řečeno, resolver slouží k formování konečného endpointu. Ten může být v závislosti na účelu operace typu `query` nebo `mutation`. Query endpoint by neměl měnit data a měl by být bezpečný a idempotentní. Naopak mutace by měly měnit data a nemusejí být idempotentní. Tyto typy endpointů jsou také využity na klientské straně pro účely cachování a invalidace dotazů a jsou inspirovány typy operací GraphQL i tRPC. Tento přístup se liší od architektury REST, kde existují minimálně čtyři HTTP metody: GET, POST, DELETE a PUT. V PTSQ se pro aktualizaci, vytváření i mazání používají mutace, zatímco dotazy jsou analogií k metodě GET. Typy endpointů mohou mít částečně nedostatečný význam, jak ale ukazují nástroje GraphQL i tRPC, při znalosti schématu a dobře formovaném názvu operace to nezpůsobuje problémy ani nepředvídatelná chování pro klienta. Oba typy endpointů jsou volány pomocí HTTP metody POST, a to především kvůli komplexitě obsahu dat, která se přenášejí v těle požadavku. Využitím pouze jedné HTTP metody knihovna nijak neomezuje žádné funkcionality ani možnosti ukládání dat do mezipaměti na straně klienta, a to především díky vlastním typům endpointů.

Vytvořit samotný endpoint lze z resolveru jen tehdy, pokud má nadefinované jakékoliv schéma výstupu. Při vytvoření endpoint získá veškerá validační schémata, popis i middleware a nelze pokračovat v řetězení argumentů, výstupů a dalších operací, jako tomu bylo u resolveru. Dotaz nebo mutace tuto řetězovou posloupnost vždy ukončují.

Při vytváření endpointu jsou poskytnuty typy z validačních schémat vstupů a zároveň se kontroluje návratový typ funkce dotazu nebo mutace. Pokud by se z této funkce vracela data, která neodpovídají typu výstupu, TypeScript by nedovolil takový kód zkompilovat a vývojové prostředí by zobrazilo chybovou hlášku. To samé platí i pro vstup, u kterého nelze přistoupit k žádné jiné vlastnosti objektu `input` než takových, které jsou definovány schématem argumentů.

```
resolver
  .args(
    Type.Object({
      firstname: Type.String()
    })
  )
  .output(Type.String())
  .query(({ input, ctx }) => `Hello, ${input.firstname}`);
```

Výpis kódu 21 Vytvoření dotazu s argumenty

Pokud by návratová hodnota funkce dotazu v kódu 21 byla jiného datového typu, než je řetězec, TypeScript by kód nezkompiloval a vývojové prostředí by rovnou zobrazilo danou chybu. Vývojář je tak neustále velmi efektivně informován o správnosti návratové hodnoty, což je velmi užitečné zejména při úpravě zdroje samotných dat, například databáze nebo jiné služby. Tato kontrola výstupních dat je obvzvlášť přínosná, pokud zdroj dat poskytuje plně otypovaná data, jako poskytují některá TypeScript ORM[6].

U vstupních hodnot pak nelze přistoupit k žádné jiné vlastnosti objektu `input` než `firstname`, protože žádné jiné vlastnosti nejsou povoleny validačním schématem. Datový typ vstupu je tak synchronizován s reálnou hodnotou, ta totiž také nemůže obsahovat jiné než povolené typy. Nevalidní vstup vůči danému schématu by totiž automaticky vyústil v chybovou odpověď. V běhové úrovni však toto schéma povoluje i další vlastnosti objektu vstupu, protože nenastavuje `additionalProperties` na hodnotu `false`. Při výchozí hodnotě je tak povoleno, aby data měla další, schématem nedefinované, vlastnosti. Uvnitř dotazu i mutace lze přistoupit i k vytvořenému kontextu požadavku, který je opět plně otypovaný.

[6] ORM je mapování mezi objektovým a relačním paradigmatem [62].

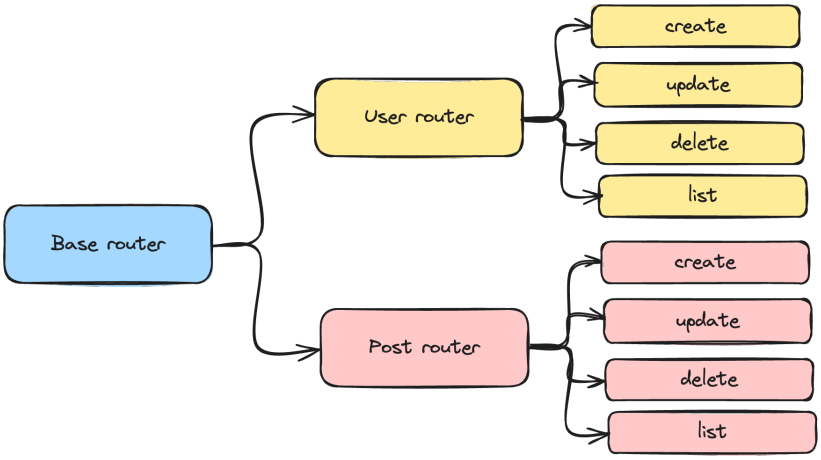
4.8 Handler

Handler je označení pro samotnou funkci která provádí dotazy nebo mutace. Tato funkce je zodpovědná za provádění samotných operací endpointu jako je dotaz na databázi a vrácení požadovaných dat. Handler může být jak synchronní, tak asynchronní. Díky tomu je možné uvnitř této funkce provádět jiné asynchronní operace a počkat na jejich výsledek.

4.9 Router

Definováním resolveru bylo popsáno, jak vytvořit endpoint aplikace. Nyní bychom tyto endpointy potřebovaly nějak logicky sdružovat a seskupovat. K tomu je určen router, který představuje další komponentu PTSQ serveru. Router slouží k definování cest endpointů a je inspirován podobným rozhraním, které vytváří tRPC. Toto rozhraní je velmi výhodné při tvorbě komplexního otypovaného API, protože objekt definující cesty uvnitř routeru automaticky poskytuje i datový typ těchto cest. Na rozdíl od přístupu frameworků pro tvorbu REST API, které definují cesty pomocí řetězců, je tento způsob z hlediska typování výhodnější. Přitom však nijak nekomplikuje samotný vývoj. PTSQ a tRPC routery se pak velmi liší svou implementací, i když rozhraním jsou téměř totožné. Hlavním důvodem odlišnosti je podpora introspekce.

Routery definují rozhraní samotného API a nabízejí strukturovaný způsob organizace a správy endpointů. Vnořování routerů tvoří hierarchickou strukturu, která odráží logickou organizaci API (obrázek 4.3).



Obrázek 4.3 Router a logická struktura API

Routery pak můžeme zanořovat mezi soubory. Instance routerů definované v jednom souboru lze exportovat a v jiném souboru importovat a vložit do zastřešujícího routeru (kód 22). Tímto způsobem je umožněno rozdělit aplikaci na menší části, které po spojení tvoří kompletní API.

```
import { userRouter } from './user';
import { postRouter } from './post';

export const baseRouter = router({
  user: userRouter,
  post: postRouter
});
```

Výpis kódu 22 Vnoření routerů z jiných souborů

Do routeru lze vkládat kromě dalších vnořených routerů také samotné endpointy. Vložení endpointu funguje stejně jako vložení routeru. Zastřešující routery mohou dokonce definovat vnořený router i endpoint na stejné úrovni zanoření (kód 23).

```
import { getUser } from './getUser';
import { createUser } from './createUser';
import { accountRouter } from './account';

export const userRouter = router({
  create: createUser,
  get: getUser,
  account: accountRouter
});
```

Výpis kódu 23 Vložení endpointů i routeru na stejné úrovni zanoření

Klíče uvnitř objektu routeru popisují cestu k danému endpointu. Mutaci `createUser` z ukázkového kódu 23 je tak přiřazena cesta `create`. Při vytvoření rodičovského routeru může být tato cesta prodloužena, například na hodnotu `user.create`. Samotný endpoint žádným způsobem neovlivňuje cestu, která je mu přiřazena. Za vytvoření těchto cest jsou vždy zodpovědné pouze routery a je doporučeno vytvářet a seskupovat endpointy a routery logicky, například podle modelů databáze. Každý router zastřešující databázový model pak může obsahovat všechny funkce CRUD pro kompletní možnost správy modelu a zároveň se při dobré logické organizaci stává jednoduše rozšířitelným o další operace.

Veškeré požadavky na PTSQ server vždy začínají v kořenovém routeru aplikace. Podle požadovaného endpointu klientem, zavolá rodičovský router některého z potomků. Toto rekurzivní volání pokračuje do doby, než narazí na endpoint. Pokud při zpracování požadavku není endpoint nalezen volání automaticky skončí odpovědí 404, Not found s PTSQ kódem chyby `NOT_FOUND`. Pokud je endpoint nalezen s nesprávným typem, je vrácena odpověď 400, Bad request s chybovým kódem `PTSQ_BAD_ROUTE_TYPE`.

V případě, že je endpoint nalezen, jsou spuštěny všechny jeho middleware a následně i samotný handler zpracovávající daný požadavek. Samotné zpracování požadavků a middleware je podrobně popsáno v podkapitole 4.12.

Router samotný vytváří TypeScript typ, který je následně v případě struktury projektu jednotného repozitáře exportován ze serveru a importován na straně klienta pro vytvoření typově bezpečného dotazování. Změny serveru jsou tak okamžitě reflektovány na klientovi, protože při změně vstupů, výstupů nebo cest se automaticky změní i některý z datových typů kořenového

routeru. Okamžité promítnutí těchto změn je tak jednou z největších výhod této knihovny. Díky tomu se celkový vývoj aplikace rapidně zrychluje a je méně chybový.

V případě více repozitářů nebo externího API lze využít introspekci schématu, která je popsána v podkapitole 4.18.

4.9.1 Sjednocení routerů

Routery, kromě vnoření, podporují i sjednocení pomocí statické metody `Router.merge`. Tato operace vytvoří nový router a žádným způsobem nemění obsah původních. Funkce sjednotí cesty obou routerů na jedné úrovni zanoření. Tím se liší od vnoření, která naopak zanořuje router hlouběji v hierarchické struktuře. Pokud se některá z cest uvnitř routerů překrývá, vždy se použije ta, jejíž router vstupuje do operace sjednocení jako druhý. Ve správně definované struktuře cest by však k překrývání docházet nemělo.

4.9.2 Verzování API

Verzování API je klíčovým prvkem správy rozhraní aplikace. Cílem verzování je zajistit stabilitu a kompatibilitu mezi různými verzemi API, zatímco umožňuje postupné aktualizace a úpravy. Poskytuje tak klientům jistotu, že rozhraní, které používají, zůstane konzistentní i při aktualizacích a změnách na straně poskytovatele služby.

Verzování REST API většinou funguje pomocí rozšiřování URI zdrojů o danou verzi, například `https://api.example.com/v1/endpoint`. Stejného výsledku lze ale v PTSQ dosáhnout vnořením routerů (kód 24).

```
import { baseRouterV1 } from './routers/v1';
import { baseRouterV2 } from './routers/v2';

const baseRouter = router({
  v1: baseRouterV1,
  v2: baseRouterV2
});
```

Výpis kódu 24 Verzování API

Klient si poté může zvolit jakou verzi použije. Dokonce lze vytvořit jednoho klienta a rozdělit ho na dva podklienty tak, že každý bude využívat jiné verze API.

4.10 PTSQ chybová odpověď

Při zpracovávání požadavků může dojít k určitým chybám, například z důvodu nedostatečného oprávnění nebo nenalezení záznamu v databázi. Chybová odpověď odesílá klientovi informace o chybném zpracování požadavku z různých důvodů. Lze ji vytvořit vyhozením výjimky `PtsqError` a to kdekoliv uvnitř aplikace na straně serveru (kód 25). Vyhození jakékoliv jiné výjimky se automaticky transformuje na PTSQ chybu a vede na odpověď s HTTP status kódem 500, Internal server error a PTSQ kódem chyby `SERVER_INTERNAL_ERROR`.

Třída `PtsqError` pro vytvoření chybové odpovědi serveru používá předdefinovaný kód chyby, zprávu chyby a vlastnost `cause`, která slouží k detailnímu popisu. Tato vlastnost v případě interních chyb může obsahovat nastavení serveru, připojení k databázi a další soukromé informace, a tak ji není vždy vhodné, například v produkčním prostředí, obsáhnout v konečné odpovědi. V podkapitole 4.12 je popsáno, jak vytvořit formátování chyb před odesláním chybové odpovědi na klienta.

```
throw new PtsqError({
  code: 'UNAUTHORIZED',
  message: 'JWT malformed',
  cause: jwtDecodeResult.error
});
```

Výpis kódu 25 Chybová odpověď

Všechny předdefinované PTSQ chybové kódy mají odpovídající HTTP status kód (tabulka 4.1). Vždy je tak chybová odpověď serveru odeslána s příslušných chybovým status kódem. Informace samotné PTSQ chyby se pak nacházejí v těle odpovědi.

PtsqError code	HTTP status
PTSQ_VALIDATION_FAILED	400
PTSQ_BODY_PARSE_FAILED	400
PTSQ_BAD_ROUTE_TYPE	400
BAD_REQUEST	400
UNAUTHORIZED	401
FORBIDDEN	403
NOT_FOUND	404
METHOD_NOT_SUPPORTED	405
TIMEOUT	408
CONFLICT	409
PRECONDITION_FAILED	412
PAYLOAD_TOO_LARGE	413
UNPROCESSABLE_CONTENT	422
TOO_MANY_REQUESTS	429
CLIENT_CLOSED_REQUEST	499
INTERNAL_SERVER_ERROR	500

Tabulka 4.1 Tabulka PTSQ chybových kódů

4.11 Chyba typové úrovně

Chybová odpověď specifikuje chybu serveru na úrovni běhové. V případě, že samotný server nepracuje s datovými typy správně, je možné tyto chyby odhalit již ve vývojovém prostředí bez nutnosti spustit samotnou aplikaci.

Při definování endpointů, spojování resolverů nebo jiných operacích může dojít k typovým chybám. Sjednocení resolverů nebo routerů nemusí být validní operací v případě, že si jejich kontexty navzájem neodpovídají. Vytvoření endpointů je zase možné pouze v případě, že je definované jakékoliv validační schéma výstupu. Kontrola těchto chyb v běhové úrovni není příliš vhodná. Chyba by totiž byla odhalena až při spuštění aplikace. Možnost definovat chyby na úrovni datových typů jazyka TypeScript umožňuje efektivní odhalení chyb, a přitom není nutné aplikace spustit. Taková chyba se totiž objeví již ve vývojovém prostředí.

Výjimky jazyka JavaScript vracejí speciální datový typ TypeScriptu `never`. Tento typ specifikuje, že operace je v dané větvi kódu vždy neplatná, a tak nemá „žádný“ datový typ. Pro specifikaci chyb na typové úrovni tak lze vždy využít datového typu `never`. Uživatel knihovny tak sice ví, že je daná operace neplatná, neví však proč.

Specifikováním vlastního chybového typu, který umožňuje přidat zprávu chyby, zvýšíme šanci na pochopení dané situace. Díky definované chybě na úrovni typů pak není nutná kontrola

na úrovni běhové, TypeScript totiž nezkompiluje kód, který není v pořádku. Absence běhové kontroly některých vstupů knihovny tak umožňuje vytvořit rychlejší kód.

Díky transformacím a podmínkám na úrovni datových typů je možné vytvořit kontroly přímo ve vývojovém prostředí. Chybu tak lze zobrazit při nesplnění nějaké podmínky (kód 26).

```
type HasSchema<T extends Schema | undefined> =
  T extends Schema ? T : 'Schema cannot be undefined';
```

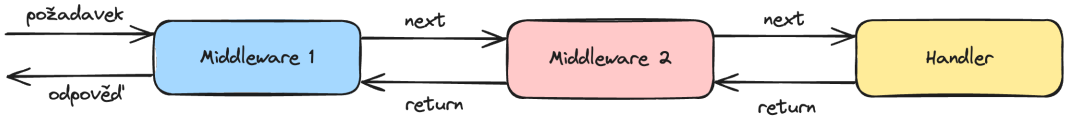
Výpis kódu 26 Chyba na úrovni datových typů

4.12 Middleware a zpracování požadavků

Definováním resolveru a routeru je již možné tvořit celé aplikace. Chybí ale možnost úpravy požadavku či odpovědi, která je však abstrahována ze samotného zpracování.

Middleware je komponenta, která je schopná upravit vstup, kontext požadavku nebo výstup. Jedná se o funkci, která je spuštěna v rekurzivním volání a vývojář má možnost zvolit, zda logika middleware poběží před voláním handleru, po něm nebo v obou případech. Tím je v rámci jedné této komponenty možné upravovat jak vstupní požadavek, tak i výstupní data odpovědi. Samotná funkce definující middleware může být jak synchronní, tak i asynchronní, což umožňuje provádět různé další asynchronní operace, například dotaz na databázi.

Každá middleware má přístup k funkci `next`. Tato funkce spouští další middleware v rekurzivním volání a vrací objekt Promise s dosavadní finální odpovědí. Pro správné fungování rekurzivního volání je povinností, aby každá middleware vracela objekt odpovědi (obrázek 4.4). Jedině tak lze znát jak požadavek, tak odpověď, v rámci jedné funkce.



Obrázek 4.4 Rekurzivní volání middleware

Kromě funkce `next` má middleware přístup k proměnným `input`, `ctx` a `meta`. Proměnná `input` obsahuje dosud zvalidovaná vstupní data. Protože je middleware registrována uvnitř resolveru, který v danou chvíli nemusí mít kvůli řetězení argumentů definovaná všechna validační schémata, může nastat situace, že `input` uvnitř funkce middleware nabízí v rámci datových typů méně vlastností než v samotném handleru.

Před každým voláním funkce middleware dojde k validaci vstupů podle validačního schématu argumentů, které je přiřazeno resolverem. Pokud je validace neúspěšná server automaticky vrátí odpověď 400, Bad request s PTSQ kódem chyby `PTSQ_VALIDATION_FAILED`.

Objekt `ctx` obsahuje dosud vytvořený kontext požadavku. Middleware dovolují upravit hodnotu tohoto kontextu v parametru volání funkce `next`. Pokud při úpravě kontextu dojde ke změně datového typu, knihovna dokáže aktualizovaný typ odvodit z volání funkce `next` a dosadit do dalších middleware, které tak mají přístup nejen k aktualizované hodnotě v běhovém prostředí, ale i na úrovni typů, čímž je minimalizována chybovost vývojáře.

Obecné informace o požadavku jsou uloženy v objektu `meta`. Ten uchovává požadovanou cestu, typ endpointu a všechna nezvalidovaná vstupní data. I tento objekt je možné uvnitř volání funkce `next` upravit, toho využívá především první middleware, která zpracovává a validuje tělo HTTP požadavku a následně odesílá získané informace do dalších middleware v rekurzivním volání.

Jak již bylo uvedeno, volání middleware je rekurzivní (kód 27). Nejprve se volá první middleware a čeká na spuštění funkce `next` po jejímž volání je spuštěna další middleware. Takto se

pokračuje, dokud se nenarazí na handler. Ten se také zpracovává v rekurzivním volání, avšak místo spouštění funkce `next` vrátí výsledek, tedy odpověď na požadavek, čímž ukončuje rekurzi.

```
const response = middlewares[index]._def.middlewareFunction({
  input: middlewareInput,
  meta: meta,
  ctx: ctx,
  next: ((options) => {
    return Middleware.recursiveCall({
      ctx: { ...ctx, ...options?.ctx },
      meta: options?.meta ?? meta,
      index: index + 1,
      middlewares: middlewares,
    });
  }),
});
```

Výpis kódu 27 Rekurzivní volání middleware

Celé rekurzivní volání začíná s indexem 0 a spouští první middleware. Ta po zavolání `next` spustí další posunutím indexu. Díky rekurzivnímu volání je možné znát uvnitř jedné funkce middleware jak vstupní, tak i výstupní data. Znalosti požadavku i odpovědi pak nabízejí možnosti především v oblasti cachování, logování nebo formátování výstupu na základě vstupu.

4.12.1 Middleware serveru

Občas middleware resolveru nestačí. Ta je totiž spuštěna až pokud je úspěšně nalezena cesta k danému endpointu, který resolver vytváří. Pokud žádaná cesta neexistuje nebo nastane jiná chyba ještě předtím, než je vybrán PTSQ endpoint, middleware se nespustí. Kvůli této situaci existuje podpora serverových middleware. Ty jsou definovány na úrovni builderu serveru a spouští se na každý požadavek ještě předtím, než je zavolán kořenový router pro směrování požadavku.

Middleware serveru tak může řešit problém zmíněný v podkapitole 4.10, tedy formátování chybových výstupů v produkčním prostředí, které zamezuje úniku citlivých informací. Ve vývojovém prostředí naopak dovoluje všechny chybové výstupy vypisovat do konzole. Umožňuje také vytvářet logování událostí, ať už jednoduché konzolové nebo s využitím nějaké externí služby.

4.12.2 Samostatná middleware

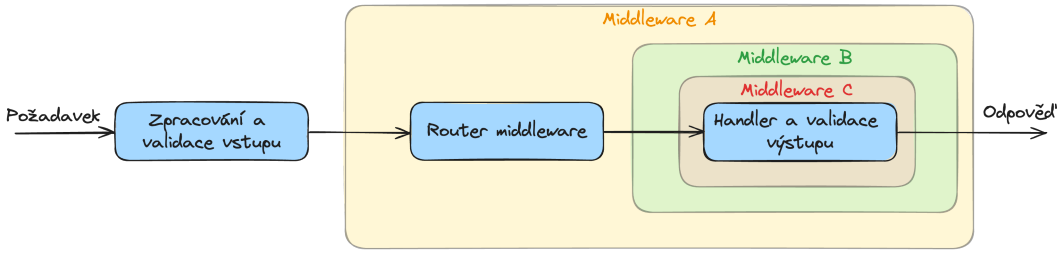
Kromě samostatného resolveru umožňuje knihovna vytvořit i samostatnou middleware, kterou lze také vytvořit bez PTSQ instance serveru. Při definování lze na úrovni typů specifikovat jaké vstupy a kontext požaduje (kód 28), tyto požadavky jsou poté respektovány při připojování middleware k resolveru.

```
const standaloneMiddleware = middleware<{
  ctx: {
    user?: User;
  };
}>().create(({ input, ctx, next }) => {...});
```

Výpis kódu 28 Samostatná middleware

4.12.3 Zpracování požadavků

Na obrázku 4.5 je názorně ukázáno zpracování požadavků PTSQ serverem. Middleware A, B a C jsou middleware definované uživatelem knihovny a mohou mít různá chování. Middleware A je definována jako middleware serveru, zatímco B a C jsou middleware nějakého resolveru. Z diagramu je patrné, že pokud první middleware serveru A čeká na výstup handleru, přijde výsledek do této middleware jako poslední ze všech. Naopak čím blíže v rekurzi je middleware handleru, tím dříve je spuštěna po vrácení odpovědi.



Obrázek 4.5 Middleware a zpracování požadavků

V textové reprezentaci se tak budou uživatelem definované middleware spouštět následovně:

$$A \to B \to C \to Handler \to C \to B \to A$$

Bloky diagramu „Zpracování a validace vstupu“, „Router middleware“ i „Handler a validace výstupu“ jsou middleware definované přímo knihovnou a jsou spouštěny spolu s vlastními middleware v rekurzivním volání.

4.12.4 Řetězení middleware a autorizace

Pomocí resolverů lze uvnitř aplikace middleware řetězit, podobně jako lze řetězit validační schémata argumentů či výstupů. Tím je možné vytvořit komplexní navazující logiku a připravit si několik komponent v podobě resolverů, které lze používat na více místech. Řetězení middleware se hodí při řešení autorizace uživatele a kontrolu oprávnění. Při vytváření jednotlivých resolverů pokrývajících různá oprávnění je možné díky řetězení navázat na již vyřešenou logiku jiného resolveru.

Pro vytvoření endpointů dostupných pouze pro přihlášené uživatele vytvoříme resolver, který pomocí middleware rozhoduje, zda je uživatel přihlášen (kód 29).

```
const loggedInResolver = resolver.use(({ ctx, next }) => {
  if(ctx.user === undefined) throw new PtsqError({ code: 'UNAUTHORIZED' });

  return next({
    ctx: {
      user: ctx.user
    }
  });
});

loggedInResolver.output(...).query(...);
```

Výpis kódu 29 Vytvoření resolveru povolující komunikaci pouze přihlášeným uživatelům

Tím je vytvořen jeden znovupoužitelný blok nebo komponenta, kterou lze uvnitř aplikace využívat na více místech. Při vytvoření endpointů pomocí tohoto resolveru pak získáme jistotu, že do handleru vstupuje pouze přihlášený uživatel. Zároveň ale tuto kontrolu abstrahujeme ze samotného konečného zpracování. Pokud je uživatel nepřihlášen a dotazuje se endpointu vytvořeného tímto resolverem, je vrácena příslušná chybová odpověď. Využívá se zde návrhového vzoru chain of responsibility.

V tomto případě middleware vytváří velmi čitelný kód zpracování požadavku, především díky řešení logiky oprávnění mimo samotný handler. Tím celkově napomáhá k menší chybovosti, kódu v handleru bude obecně méně, čímž bude méně náchylný k chybám.

Od této chvíle při používání resolveru `loggedInResolver` kontext požadavku již nenabízí hodnotu `undefined`, tedy možnost, že by uživatel mohl být nepřihlášen. Podmínka při vyhození výjimky daný datový typ odfiltruje a uvnitř parametru funkce `next` je datový typ uživatele v kontextu přetypován.

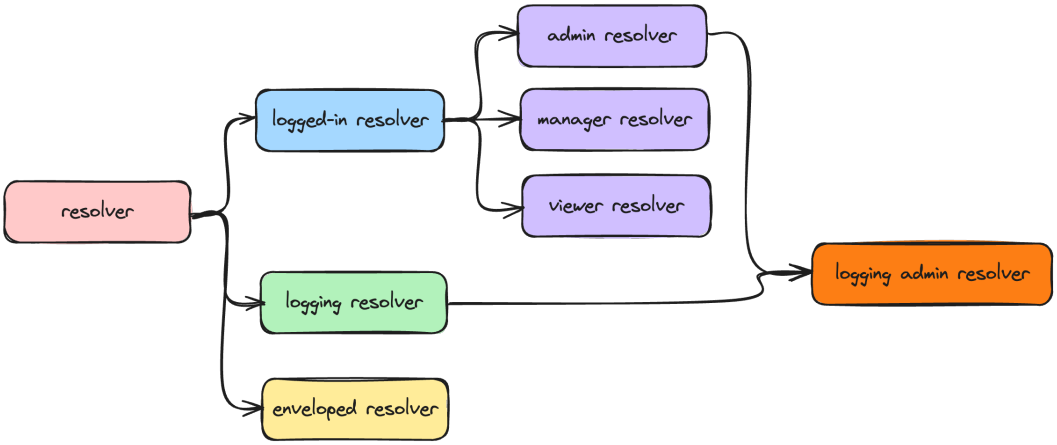
Předpokládejme, že aplikace definuje endpoint povolený jen určité roli uživatele, který musí být tedy automaticky přihlášen. Využitím předchozího resolveru vytvoříme nový, který efektivně použije omezení definovaná v předchozím a dodá své vlastní. Zároveň není potřeba uvnitř navazující middleware nijak manuálně přetypovávat hodnotu `user` pro označení, že už nemůže nabývat hodnoty `undefined`. Kontext požadavku je totiž převzat z předchozího resolveru a ten hodnotu uživatele již velmi elegantně přetypoval (kód 30). Tento způsob navazování na předchozí již vytvořené komponenty podporuje velmi efektivní způsob autorizace. Pro komplexní autorizaci je pak možné využít metody jako ACL[7] nebo voter[8].

```
const adminResolver = loggedInResolver.use(({ ctx, next }) => {
  if(ctx.user.role !== Role.Admin) throw new PtsqError({ code: 'FORBIDDEN' });

  return next({
    ctx: {
      user: ctx.user
    }
  });
});

adminResolver.output(...).query(...);
```

Výpis kódu 30 Vytvoření resolveru povolující komunikaci pouze pro uživatele s rolí administrátor

[7] ACL je komplexní soubor omezení pro definování přístupu uživatele [63].
[8] Voter definuje omezení pro přístup k objektu.

Využíváním předchozích již definovaných resolverů s middleware jednoduše tvoříme typově bezpečné a znovupoužitelné komponenty serveru, které lze používat i rozšířit v různých místech aplikace. Tyto komponenty si díky middleware nesou logiku, kterou řeší, a pomocí řetězení resolverů, které bylo popsáno v podkapitole 4.6.2, pak lze funkcionalitu připojit k jakémukoliv jinému resolveru. To znamená, že i resolver povolující komunikaci pouze pro uživatele s rolí administrátor může zajišťovat logování (obrázek 4.6). Tímto stylem lze definovat velmi komplexní systémy, které jsou stále přehledně organizované díky možnosti logiku rozdělit. Elegantním přetypováním a udržováním datových typů jsou, i přes možnosti rozdělení logiky, stále velmi typově bezpečné.



Obrázek 4.6 Řetězení middleware a resolverů

4.13 Autentizace

PTSQ knihovna sama nezajišťuje autentizaci. Ta musí být provedena externě buď pomocí vlastní implementace nebo knihovny. Pro tvorbu snadné autentizace lze použít knihovnu Next-auth[9], která je určena pro práci s frameworkem Next.js, nebo PassportJS[10], která poskytuje obecnější rozhraní a podporuje jakékoliv frameworky. Po úspěšné autentizaci bývá odeslán například JWT token na stranu klienta pro následnou možnost autorizace v dalších požadavcích. Při dalším dotazování je tak token přihlášeného uživatele většinou uveden v hlavičce Authorization s každým požadavkem nebo je automaticky připojen k požadavkům pomocí nastavených cookies. Na straně serveru pak lze JWT token dekódovat a data o uživateli přidat do kontextu požadavku a tím tuto informaci zpropagovat do všech endpointů a middleware, které s daty mohou dále operovat.

Celkový proces autentizace i následné dekódování tokenů pro autorizaci je velmi snadný. Knihovna navíc udržuje datový typ kontextu požadavku, vývojář tak získává přehled o atributech uživatele uvnitř vývojového prostředí.

[9]Knihovna Next-auth je dostupná na webové stránce https://next-auth.js.org/.
[10]Knihovna PassportJS je dostupná na webové stránce https://www.passportjs.org/.

4.14 Nasazení aplikace

Dosud bylo popsáno, jak aplikaci vytvořit. Bylo rozebráno, jak tvořit endpointy, jak je sdružovat pomocí routerů a jak efektivně využívat middleware. Nyní jen zbývá celou aplikaci nasadit.

Nasazení zároveň odhaluje poslední komponentu PTSQ serveru, kterou je Proxy objekt `serve` zastupující funkci. Různá volání této funkce zajišťují podporu nasazení aplikace v mnoha různých prostředích počínaje Node.js přes full-stack frameworky až po alternativní runtime. Funkce `serve` přijímá jediný argument a tím je kořenový router celé aplikace, který bude následně vždy volán při požadavku na server jako první ze všech routerů. `Serve` pak vrací generickou funkci posluchače HTTP serveru, která naslouchá požadavkům na nasazeném HTTP endpointu. Samotnou funkci posluchače pak zajišťuje knihovna `@whatwg-node/server`.

Protože rozhraní různých frameworků v daných prostředích je velmi podobné, není nutné tvořit speciální adaptéry pro každý jeden framework zvlášť, jako tomu je na straně frontendu. Pro nasazení uvnitř nestandardních prostředí nabízí `serve` kromě samotného volání funkce i metody. Metoda `Serve.fetch` slouží pro nasazení PTSQ serveru uvnitř serverless prostředí, jako je AWS lambda, nebo v alternativním runtime, jako je Bun. Pro nasazení uvnitř frameworku, který využívá upravené objekty standardní pro Node.js aplikace jako Koa nebo Fastify, poskytuje `serve` metodu `Serve.handleNodeRequestAndResponse`.

Kromě nasazení aplikace v různých prostředích umožňuje `serve` definovat i vlastní argumenty pro tvorbu kontextu požadavku. Pro vytvoření vlastních argumentů je nutné, aby funkce vytvářející kontext přijímala tyto parametry na úrovni typů TypeScriptu (kód 31). Funkce `serve` automaticky zahrne do argumentů objekt požadavku `request` datového typu `Request` definovaného standardem WHATWG.

```
const { resolver, router, serve } = ptsq({
  // parametry kontextu musejí být definované na úrovni TypeScriptu
  ctx: (args: { req: FastifyRequest; reply: FastifyReply }) => {...}
}).create();

const response = await serve(baseRouter).handleNodeRequestAndResponse(
  req,
  reply,
  {
    req,
    reply,
  }
);
```

Výpis kódu 31 Nasazení aplikace pomocí Fastify s vlastními argumenty kontextu požadavku

4.15 Testování aplikace

Testování aplikace je jedním z nejdůležitějších procesů v oblasti vývoje softwaru a webových aplikací. PTSQ v rámci testování umožňuje spouštět dotazy a mutace včetně routování a middleware bez nutnosti vytvoření HTTP serveru. Je tak možné jednoduše spouštět testy například v prostředí CI[11].

Bez HTTP serveru však není k dispozici žádný objekt požadavku, a tedy ani možnost vytvořit kontext požadavku aplikace. Knihovna poskytuje testovací caller, který umožňuje nahradit kontext požadavku umělým objektem, dodaným do testu zvenčí. Tím je při tvorbě testů umožněno simulovat právě přihlášeného uživatele nebo další informace související s požadavkem. PTSQ tak podporuje možnost vytvářet testy závislé na různých oprávněních uživatelů (kód 32).

```
const baseRouterCallerBuilder = Router.serverSideCaller(baseRouter);

const adminCaller = baseRouterCallerBuilder.create({
  user: new User({ admin: true })
});

const unauthorizedCaller = baseRouterCallerBuilder.create({
  user: undefined
});

const adminCallerGreetingsResponse = await adminCaller.greetings.query();
```

Výpis kódu 32 Vytvoření testovacího calleru

Pomocí metody `Router.serverSideCaller` se vytvoří builder testovacího calleru nad daným routerem. Je však vždy doporučeno vytvářet caller nad kořenovým routerem aplikace. Tím se samotné testy nejvíce přiblíží reálnému chování. Při tvorbě testů pomocí calleru nejsou nikdy spuštěny middleware serveru, middleware testovaného endpointu ale ano. Je tak důležité rozmyslet, zda je pro testování vhodnější využít calleru nebo vytvořit kompletní HTTP server s nasazenou aplikací. Samotný caller se pak vytvoří metodou `ServerSideCallerBuilder.create` a jako argument se uvede umělý objekt kontextu požadavku. Pro testování je doporučeno volat funkci `Router.serverSideCaller` pouze jednou a pro různé scénáře testů volat opakovaně metodu `ServerSideCaller.create`. Testovací caller je plně otypovaný, to znamená, že nabízí datové typy vstupů, výstupů a endpointů, stejným způsobem jako klient na straně frontendu.

PTSQ tímto podporuje na straně serveru jak jednotkové, tak i integrační testy, které lze díky absenci HTTP serveru jednoduše spustit uvnitř různých prostředí. Usnadněním procesu testování umožňuje knihovna vývojářům snadno vytvářet bezpečné webové služby.

Samotný caller pak nemusí sloužit pouze k testování aplikace, ale lze díky němu spouštět dotazy nebo mutace aplikace uvnitř jiného dotazu. Kvůli spouštění všech middleware volaného endpointu to však není vždy vhodné po optimalizační stránce.

[11] Kontinuální integrace (CI) je postup umožňující často a spolehlivě nasazovat nové funkcionality a produkty [64].

4.16 Logování

Zaznamenávání dotazů a případně i úspěšných nebo chybových odpovědí je jedna z klíčových vlastností serveru. Díky logování je možné odhalit chyby v různých prostředích nasazení nebo měřit různé důležité metriky. Logování je uvnitř PTSQ aplikace umožněno pomocí vytvoření logovací middleware. Ta může být v závislosti na typu zaznamenávání nasazena jako middleware serveru nebo uvnitř resolveru.

4.17 Obálka odpovědi

Webové API občas může definovat obálku odpovědi (envelope). Obálka je běžnou praxí při vytváření otevřených API a přináší nějakou standardní strukturu nebo kostru odpovědi pro celou webovou službu nebo specifickou operaci. Vytvoření obálky uvnitř netypovaných systémů je velmi jednoduché, schéma totiž neexistuje, není tedy nutné nějak typovat obal odpovědi. Při tvorbě obálky uvnitř PTSQ je nejlepším způsobem vytvořit resolver, který má předpřipravenou strukturu odpovědi (kód 33).

```
const envelopedAggregationResolver = resolver
  .output(
    Type.Object({
      data: Type.Unknown(),
      paging: Type.Object({...}),
      count: Type.Number(),
    })
  );
```

Výpis kódu 33 Resolver s obálkou odpovědi

Použitím takto nadefinovaného resolveru je zajištěna nutnost vytvoření struktury obálky odpovědi v handleru. Zároveň není nutné kód opakovat a datový typ dat lze jednoduše rozšířit za pomoci řetězení výstupních validačních schémat. Použitím `Type.Unknown()` místo `Type.Any()` jako základního typu výstupu dat je pak zaručena nutnost definování dodatečného schématu pro data. Žádný výstup handleru totiž nebude kompatibilní s datovým typem `unknown`, pokud není manuálně přetypován. Takto nadefinovaná obálka odpovědi je zároveň validní pro export schématu API.

4.18 Introspekce

Introspekce, podobně jako v GraphQL, slouží k exportování schématu PTSQ API. To otevírá možnosti pro tvorbu otevřeného silně otypovaného API a klienta, který bude využívat znalosti schématu. Dokonce i externí klienti třetích stran se tak budou moci dotazovat PTSQ serveru se znalostí typů vstupů, výstupů i možných endpointů. Samotnou introspekci není třeba nijak nastavovat, neposkytuje žádná citlivá data jako části kódu serveru, hesla, tokeny nebo připojení k databázi. Introspekce pouze reflektuje strukturu API a využívá znalosti validačních JSON schémat. Klient tedy zná strukturu aplikace i validační schémata použitá pro validaci argumentů, respektive výstupů.

Využívání introspekce je nevhodné při tvorbě klienta, který je ve stejném repozitáři jako PTSQ server. V takovém případě je vždy lepší využít možnosti exportovat datový typ kořenového routeru ze serveru a importovat ho na klientovi. Tím vznikne plynulý přenos schématu a klientská část kódu může okamžitě reagovat na změny serveru.

Každý PTSQ router i endpoint pak umožňuje export svého schématu. Při introspekci dochází k dotazování se na schéma jednotlivých komponent serveru. Proces vždy začíná u kořenového routeru nasazeného na HTTP serveru. Router prochází všechny specifikované cesty, zanořuje se do nich a exportuje jejich schémata. Pokud narazí na další router, proces pokračuje zanořením. Pokud narazí na endpoint, proces končí a rekurzivně se vrací zpět do kořenového routeru, který nakonec vrátí kompletní schéma celého API.

Introspekce reaguje na požadavky s HTTP metodou GET a stejným HTTP endpointem na kterém je nasazený PTSQ server. Schéma je odesláno jako odpověď serveru ve formátu JSON, nikoliv však JSON schéma. PTSQ schéma pak obsahuje informace o routerech a endpointech, včetně vstupů, výstupů a typech endpointů.

Formát JSON je sice vhodný pro přenášení dat mezi klientem a serverem, neumožňuje však definovat datové typy nebo struktury, které by bylo možné víc využít v jazyce TypeScript. Pro práci se získaným schématem je tedy nutné ho převést do TypeScriptu. Převod spočívá v přetypování JSON objektu schématu na `as const` (kód 34). Všechny vlastnosti objektu schématu se tak stanou určeny pouze pro čtení a TypeScript získá typové informace o jednotlivých vlastnostech. V budoucnu možná bude přímá podpora vkládání JSON souboru tak, jako kdyby byl určen pouze pro čtení (`as const`), čímž by se celý krok převodu schématu eliminoval. Vývojáři tuto funkcionalitu jazyka TypeScript hojně požadují.

```
import type { IntrospectedRouter } from '@ptsq/server';

export const BaseRouter = {
  nodeType: 'router',
  routes: {
    greetings: {
      type: 'query',
      nodeType: 'route',
      outputSchema: { type: 'string' },
    },
  },
} as const satisfies IntrospectedRouter;
```

Výpis kódu 34 Ukázka převedeného schématu z introspekce

Samotné odvození typů validních hodnot z JSON schémat získaných z introspekce, jako je `outputSchema` v kódu 34, je umožněno pomocí typové knihovny `json-schema-to-ts`[12]. Pro její funkčnost musí být právě tato schémata určena pouze pro čtení, tak aby bylo schéma dostupné i na úrovni datových typů.

[12] Typová knihovna `json-schema-to-ts` je dostupná na webové stránce https://github.com/ThomasAribart/json-schema-to-ts.

Převod schématu z introspekce lze provést buď manuální úpravou výsledné JSON odpovědi, nebo nástrojem `@ptsq/introspection-cli`. Tento skript stáhne schéma ze zadaného PTSQ API a automaticky ho přetypuje v jazyce TypeScript. Výhodou tohoto nástroje je možnost aktualizovat typy před každým sestavením aplikace (kód 35). Generované schéma je tak vždy aktuální a umožňuje reagovat na změny. V případě častých změn je však doporučeno vytvořit různé verze API, jak bylo popsáno v podkapitole 4.9.2.

```
{
  "name": "example",
  "scripts": {
    "prebuild": "introspection-cli --url='https://example.com/ptsq'",
  },
}
```

Výpis kódu 35 Ukázka souboru package.json

Skript umožňuje pomocí přepínače `--out` vybrat, do jakého souboru se schéma externího API uloží. Přepínačem `--lang` je pak možné zvolit formát, ve kterém bude výsledné schéma uloženo, na výběr jsou možnosti „ts“ a „raw“. Možnost „raw“ uloží schéma ve formátu JSON, tak jak přijde ze serveru, tedy bez dodatečných převodů. To může být praktické při tvorbě nějaké vlastní webové dokumentace. Ve výchozím nastavení je vybrána možnost „ts“. Díky přepínači `--name` pak lze vybrat název proměnné schématu.

4.19 Dokumentace

Schéma velmi dobře popisuje strukturu API, především pak uvnitř vývojového prostředí. Je však možné dodatečně vytvořit webovou dokumentaci, která spočívá především ve vytvoření přehledného webového rozhraní popisující API pro použití služby externími klienty. Pro tvorbu dokumentace lze využít samotné schéma služby. Pomocí HTTP metody GET lze dotazem na server získat schéma z introspekce, na jehož základě bude vytvořena samotná webová dokumentace (kód 36).

```
const schema = await fetch('http://localhost:3000/api/ptsq');
```

Výpis kódu 36 Introspekce v kódu

Proměnná `schema` však nemá žádný datový typ, a tak se se schématem pracuje velmi obtížně. Typ schématu lze odvodit pomocí speciálního TypeScript typu `inferPtsqSchema`. Odvození lze provést jak pomocí převedeného schématu z introspekce, tak pomocí datového typu kořenového routeru.

Na základě schématu pak lze vytvořit frontend, který popisuje PTSQ API přehledně na webové stránce. Ve full-stack frameworcích, jako je Next.js nebo SvelteKit, lze stránku dokumentace vykreslit již na serveru pomocí pomocí SSR[13]. Stránka dokumentace tak může být přívětivá i co se týče SEO[14].

[13] SSR je technika vykreslování webové stránky na straně serveru.
[14] SEO představuje optimalizování indexace webové stránky pro vyhledávače [65].

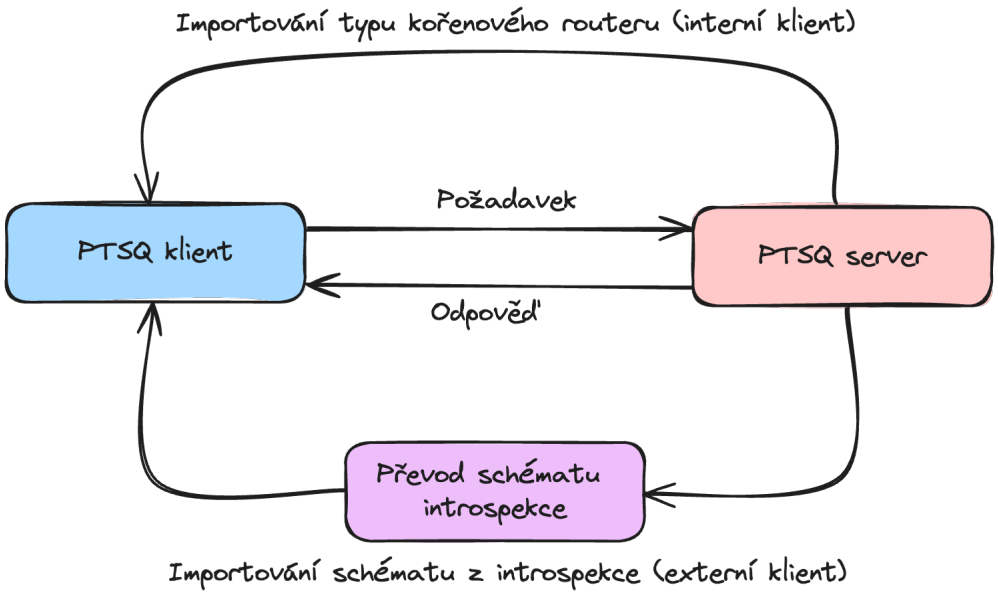
4.20 PTSQ klient

Kromě vytvoření serveru, nabízí knihovna možnost tvorby typovaného klienta. Pro netypované odesílání požadavků na PTSQ server však není modul klienta potřeba. Samotná knihovna poskytuje tři různé moduly klientů: pro JavaScript nebo TypeScript bez frameworků, pro React a pro Svelte. PTSQ ovšem dovoluje vytvořit vlastního klienta pro jakýkoliv jiný frontendový framework. Všechny klientské adaptéry frameworků odkazují na základní klientský modul `@ptsq/client`.

Pro vytvoření klienta je vyžadován typový argument, jehož typem musí být kořenový router ze serveru nebo výsledek introspekce (obrázek 4.7). Nad tímto typem jsou dále prováděny různé transformace na typové úrovni, díky čemuž je možné sestrojit klienta s typovými předpoklady endpointů, vstupů a výstupů.

Samotný klient je Proxy objekt obalující prázdnou funkci. Klient nemá žádnou kontrolu endpointů, vstupů nebo výstupů na běhové úrovni, kontrola existuje pouze na úrovni typové. Stále je tedy možné dotázat se na nesprávný endpoint, použít nesprávnou metodu či zadat nesprávný vstup. Pro takové chování je nutné typovou kontrolu jazyka TypeScript vypnout, protože ten by jinak aplikaci nedovolil zkompilovat.

Vytvoření samotného klienta vyžaduje jediné povinné nastavení `url`, které specifikuje, jakého PTSQ serveru se má klient dotazovat.



Obrázek 4.7 PTSQ klient – konzumace schématu API

Dalším již nepovinným nastavením klienta je vlastnost `fetch`. Ta dovoluje nahradit volání požadavků na HTTP serveru vlastním voláním, což může být užitečné při nastavení vlastních HTTP hlaviček a dalších možností dotazu. Vlastnost `fetch` se dá také využít k vytvoření klienta v prostředí, které nemá k dispozici fetch API. Při nastavování vlastní fetch funkce lze celé volání na HTTP server nahradit polyfillem a je tak možné klienta spouštět i z jiných prostředí, například ze starších verzí Node.js, které funkci fetch nativně nepodporovaly (kód 37).

```
import type { BaseRouter } from './server';

const client = createProxyClient<BaseRouter>({
  url: 'http://localhost:4000/ptsq',
  fetch: async (input, init) => {
    const jwt = await cookie.get('token');

    return fetchPolyfill(input, {
      ...init,
      headers: {
        ...init.headers,
        Authorization: `Bearer ${jwt}`
      }
    });
  }
});
```

Výpis kódu 37 Tvorba otypovaného klienta s vlastní metodou fetch

Pro odeslání požadavku zvolí klient endpoint, kterého se chce dotázat. Pomocí klíčů Proxy objektu klienta vybere endpoint, následně pak zavolá danou metodu pro vybraný typ endpointu. Vše je plně otypované, vývojové prostředí tak samo nabízí možné cesty při výběru požadovaného endpointu. Poslední klíč Proxy objektu pak určuje, zda se jedná o dotaz nebo mutaci. Po zavolání této metody je odeslán HTTP požadavek s příslušnými daty na PTSQ server (kód 38). Kvůli bezpečnosti API se musí typ endpointu validovat i na straně serveru. To se však děje automaticky, uživatel knihovny se o tuto validaci nestará.

```
/* endpoint: user.list */
const response = await client.user.list.query({
  filter: {...}
});

/* endpoint: user.create */
const response = await client.user.create.mutate({
  email: 'john@example.com',
  firstName: 'John',
  lastName: 'Doe'
});
```

Výpis kódu 38 Zavolání dotazu a mutace

Parametrem metody `query` nebo `mutate` jsou pak samotné vstupy daného endpointu, které jsou také plně otypované. Výstup `response` má také typové předpoklady, klient tak vždy ví, jaká data může odeslat a jaká přijdou v odpovědi.

4.20.1 Importování kódu serveru uvnitř klienta

Kód klienta v případě znalosti zdrojového kódu serveru vyžaduje datový typ kořenového routeru. Tento datový typ poskytuje klientovy kompletní typové předpoklady. Při importování routeru pro tvorbu a definici rozhraní klienta je nevhodné odvození datového typu routeru direktivou `typeof` na straně frontendu. Ten by totiž musel importovat téměř veškerý kód serveru, především kvůli provázanosti dalších routerů a endpointů, které kořenový router zastřešuje. Odvození typu je tak doporučeno provést v části serveru a tím umožnit vkládat typ kořenového routeru jako vývojovou závislost. Při importování samotného TypeScript typu nedochází k ovlivnění velikosti kódu klienta v běhovém prostředí (kód 39).

```
import type { BaseRouter } from './server';

const client = createProxyClient<BaseRouter>({
  url: 'http://localhost:4000/ptsq',
});
```

Výpis kódu 39 Správné vytvoření klienta se schématem odvozeného z routeru

```
import { baseRouter } from './server';

const client = createProxyClient<typeof baseRouter>({
  url: 'http://localhost:4000/ptsq',
});
```

Výpis kódu 40 Nesprávné vytvoření klienta s odvozením typu routeru na klientovi

Pokud by hodnota kořenového routeru při nesprávném importování (kód 40) nebyla využita jinak než pro odvození datového typu, TypeScript by import během kompilace také odstranil. Pro bezpečnost a čitelnost kódu je však doporučeno exportovat a importovat pouze datový typ. Předejde se tak možným únikům citlivých dat serveru.

K odvození typu na klientovi ale dochází v případě importování schématu z introspekce. Protože však schéma, jehož struktura a podoba byla popsána v podkapitole 4.18, ovlivňuje kód velmi minimálně, nevadí tak vložení samotného schématu a následné odvození datového typu direktivou `typeof` přímo na straně klienta. Schéma z introspekce navíc neobsahuje žádná citlivá data, podobně jako může obsahovat kořenový router serveru. Nemůže tak dojít k jejich nechtěnému úniku. Výhodou běhového kódu schématu navíc je, že lze přímo použít pro tvorbu webové dokumentace.

4.20.2 Link

Link je, podobně jako middleware, funkce, která umožňuje obalit volání požadavku klienta na HTTP server. Tato architektura je silně inspirována architekturou GraphQL Apollo Link, která byla popsána v podkapitole 3.10. Linky jsou dostupné ve všech adaptérech klientských frameworků. Podobně jako middleware, jsou spouštěny rekurzivně, tedy jeden link může upravovat jak vstup požadavku, tak i výstup. Posledním linkem je terminating link, který je zodpovědný za koncovou komunikaci s HTTP serverem.

Linky mohou sloužit například k přidání HTTP hlaviček k požadavku nebo k logování operací na straně klienta.

Je důležité zmínit, že link nezná typ vstupu ani výstupu, který bude linkem procházet. Není tedy vždy bezpečné manipulovat s těmito daty. Při úpravě především výstupu může dojít k nekonzistenci mezi typovou definicí výstupu endpointu získanou ze schématu a reálnými daty upravenými po přijetí uvnitř linku.

4.20.3 React klient

Standardní React klient knihovny v modulu `@ptsq/react-client` vychází ze základního modulu klienta PTSQ. Pro dotazování serveru a správu dat využívá klient knihovny React query z rodiny knihoven Tanstack query. To umožňuje využívat efektivního cachování, invalidace dotazů, implementace aplikací v ekosystémech React nebo Next.js, podporu dotazů při vykreslování stránky na straně serveru, stránkování a mnoho dalších výhod, které React query přináší. Největší předností však stále zůstává typová bezpečnost. I přes to, že klient nabízí různé pokročilé funkce Tanstack query, je stále silně otypovaný. Tato kombinace typování a komplexních funkcí velmi významně zjednodušuje tvorbu React nebo Next.js frontendu s využitím knihovny PTSQ.

4.20.4 Svelte klient

Kromě React klienta nabízí knihovna i Svelte klienta pro práci s frameworkem Svelte nebo SvelteKit. Ten funguje stejně jako React klient na základech Tanstack query, přesněji Svelte query. Silné typování opět přetrvává a s pomocí Tanstack query vytváří efektivního klienta pro framework Svelte.

4.20.5 Stránkování a nekonečné rolování

Stránkování a nekonečné rolování je technika poptávání obsahu postupnými dotazy. Při stránkování se klient dotazuje serveru pro obsah dané stránky s omezenou velikostí, což výrazně zrychluje počáteční načítání dat, ty se totiž nemusejí získat všechna najednou. Nekonečné rolování se naopak dotazuje serveru v závislosti na scrollování uživatelem na stránce.

Knihovna Tanstack query podporuje snadné stránkování a nekonečné rolování, a protože PTSQ React a Svelte klient tuto knihovnu rozšiřují, nabízejí stejné jednoduché rozhraní pro tyto techniky načítání dat. Základní klient knihovny PTSQ nevyužívá Tanstack query, jednoduché rozhraní pro tvorbu stránkování tedy nepodporuje, umožňuje však tvorbu vlastního systému stránkování.

Tanstack query poskytuje funkci „Infinite query“, která vytváří jednoduché a efektivní stránkování nebo nekonečné rolování. Pro možnost využít Infinite query uvnitř aplikace, je nutné, aby server přijímal argument `pageParam` jakéhokoliv datového typu. Ten musí být definován uvnitř validačních schémat argumentů na straně serveru. Pokud server tento argument nepřijímá, jazyk TypeScript nepovolí vytvoření stránkovacího dotazu. Od serveru se pak očekává, že vrátí odpověď s daty a informacemi o stránkách, což je klíčové pro vytvoření efektivního stránkování.

4.20.6 Suspense query

Suspense query je technika dotazování uvnitř React klienta. Při klasickém dotazování je potřeba pro zobrazení načítání nebo chybových hlášek podmínečně vykreslit načítání na základě dat z hooku (kód 41). Při suspense query je možné podmínečné vykreslení abstrahovat na komponentu vyšší úrovně a nezabývat se tak tím uvnitř komponenty, která data požaduje. Výhodou suspense query může být zpřehlednění kódu komponenty, nevýhodou je však možná nepřehlednost při zpracovávání požadavků. Abstrahováním načítání nebo odchytávání chyb ven z komponenty může vývojář ztrácet pojem o tom, co se v aplikaci děje. Samotné operace načítání tak přímo nevidí ve zdrojovém kódu komponenty, která data požaduje.

```
export const App = () => {
  const userListQuery = await client.user.list.useQuery({
    filter: {...},
  });

  return (
    <Page
      isLoading={userListQuery.isFetching}
      isError={userListQuery.error}
    >
      <p>Data: {JSON.stringify(userListQuery.data)}</p>
    </Page>
  )
}
```

Výpis kódu 41 Dotaz pomocí PTSQ React klienta

4.20.7 Invalidace dotazů

Klienti pro React nebo Svelte podporují invalidaci dotazů, která je implementována uvnitř knihovny Tanstack query.

Klíče dotazů jsou v případě PTSQ klientů tvořeny automaticky pomocí specifikovaného endpointu a vstupů požadavku. Díky tomu je možné po úspešné mutaci snadno invalidovat všechny související dotazy. Invalidace dotazů je tak velmi snadná a vývojář se nemusí zabývat manuální aktualizací dat. V komplexní aplikaci představuje invalidace dotazů obrovské usnadnění a deklarativní způsob aktualizace. Díky cachování, optimalistickým aktualizacím a dalším výhodám Tanstack query nedochází k neoptimálním dotazům na server. PTSQ dále umožňuje specifikovat dodatečné klíče dotazu `additionalQueryKey`, které mohou sloužit k větší specifikaci, čímž lze zvýšit výkon aplikace a snížit overfetching.

4.20.8 Vlastní klient

Základní modul klienta (`@ptsq/client`) umožňuje vytvořit vlastní adaptér pro frontendový framework. To vyžaduje znalosti typového systému jazyka TypeScript a knihovny PTSQ. Modul klienta poskytuje funkce na vysoké úrovni abstrakce a umožňuje díky nim vytvořit jakýkoliv adaptér do jiných než dvou vybraných frontendových frameworků. React i Svelte klient využívají těchto funkcí pro tvorbu klientů s pomocí Tanstack query.

4.20.9 Přerušení požadavku

V některých aplikacích může být vyžadováno mít možnost přerušit nějaký, již odeslaný požadavek. Klient knihovny umožňuje přerušení takového požadavku pomocí standardního rozhraní `AbortController`. Funkce volající dotaz nebo mutaci umožňuje připojit k požadavku `AbortSignal`, který vychází z rozhraní AbortControlleru (kód 42).

```
const abortController = new AbortController();

const response =
  await client.user.list.query(..., { signal: abortController.signal });

abortController.abort();
```

Výpis kódu 42 Přerušení dotazu

Přerušení požadavku je podporováno jak základním klientem knihovny, tak i klienty pracujícími s frontend frameworky React a Svelte.

Některé frameworky pro tvorbu HTTP serveru umožňují reagovat na toto přerušení požadavku. Reakce ze strany serveru však není příliš častá. Přerušení požadavku se využívá spíše při dotazování, při mutacích nemá příliš velký význam. Pokud totiž server nereaguje na přerušení, není možné vrátit již provedené změny.

4.20.10 Chybová odpověď na klientovi

Jelikož se i chybová odpověď přenáší mezi serverem a klientem pomocí formátu JSON, musí být na klientovi transformována opět do třídy reprezentující PTSQ chybu. Samotná třída chybové odpovědi pak obsahuje PTSQ kód chyby, zprávu chyby a vlastnost `cause`, která blíže popisuje daný problém.

Problém může nastat v datovém typu chybové odpovědi. Při použití klasického klienta je odpověď vrácena v objektu Promise, který však nepodporuje možnost definovat typ výjimky. Chybová odpověď je tak vždy neznámého datového typu. K automatickému přetypování může dojít při kontrole samotné výjimky. TypeScript totiž umožňuje pro funkci vracející `boolean` nadefinovat predikát. To je datový typ parametru funkce, který se nastaví, pokud je z této funkce vrácena hodnota `true` (kód 43). Jinak řečeno, ve větvi kódu při splnění dané podmínky této funkce, je argument přetypován na příslušný datový typ.

```
try {
  const response = await client.user.list.query(...);
} catch(error) {
  if(PtsqClientError.isPtsqClientError(error)) {
    console.error(error.code);
  }
}
```

Výpis kódu 43 Kontrola chybové odpovědi

Uvnitř podmínky definované v `catch` bloku je typ výjimky přetypován z datového typu `unknown` na typ `PtsqClientError`. Po přetypování je tedy možné bezpečně přistoupit k vlastnostem PTSQ chybové odpovědi.

Při použití klientů pracujících s Tanstack query se datový typ chybové odpovědi automaticky nastaví na `PtsqClientError`. Protože Tanstack query nepoužívá Promise pro vrácení výsledků ze serveru, není potřeba typ chybové odpovědi dále kontrolovat nebo přetypovávat. Zmíněný problém tam tedy nenastává.

4.21 Použití knihovny v komplexním projektu

Knihovna PTSQ slouží jako základ aplikace. Kombinováním s ostatními nástroji ekosystému jazyka JavaScript lze pomocí knihovny vytvářet velmi komplexní webové aplikace. Tato podkapitola slouží k představení části tohoto ekosystému a k vyobrazení zasazení knihovny mezi ostatní nástroje a frameworky.

PTSQ lze kombinovat na straně serveru s téměř jakýmkoliv frameworkem, který vytváří HTTP server. Na straně klienta podporuje jakýkoliv framework díky možnosti vytvořit vlastního klienta. Vzhledem k obrovské podpoře různých prostředí je možné projekt nasadit bez nutnosti správy vlastního serveru v prostředích jako AWS lambda a Cloudflare Workers nebo službách jako Vercel[15], DigitalOcean[16] nebo Render[17].

Při samotném procesu vývoje je důležité, aby netrval příliš dlouho, je tak možné rychleji nasadit nové funkcionality pro koncové uživatele. Jedním způsobem pro zrychlení vývoje je usnadnění promítnutí aktualizací zdrojových kódů. Při změnách kódu dojde k okamžitému obnovení aplikace bez nutnosti neustále manuálně restartovat server. Mnoho nástrojů a frameworků poskytuje speciální režim vývoje, kdy při aktualizaci souborů zdrojového kódu dojde k automatickému obnovení buď celé aplikace, nebo její upravené části. Mezi nejznámější takové nástroje patří Nodemon[18] nebo téměř jakýkoliv full-stack framework. Tato technika aktualizací velmi zefektivňuje proces vývoje a díky tomu, že knihovna podporuje širokou škálu různých frameworků a prostředí, je možné tuto techniku aplikovat i na vývoj aplikace vytvořené pomocí PTSQ.

Většina webových aplikací uchovává data v různých databázích, nejběžněji se však využívají relační. TypeScript ekosystém nabízí mnoho ORM frameworků, přičemž většina z nich podporuje mapování databázových modelů i do samotných datových typů TypeScriptu. PTSQ lze jednoduše kombinovat s jakýmkoliv ORM jako je Prisma[19], MikroORM[20] nebo Drizzle[21], přičemž díky typově bezpečným argumentům a výstupům PTSQ endpointu má vývojář neustále přehled o správnosti přijatých i vrácených dat z databáze. Pokud by databáze vrátila nekompatibilní datový typ s tím, který definuje validační schéma výstupu, TypeScript by nedovolil kód zkompilovat a ve vývojovém prostředí by zobrazil daný problém. Celková práce s databází se díky těmto typovým kontrolám stává velmi elegantní. Drizzle ORM dokonce nabízí možnost vytvořit Typebox schémata přímo z databázových modelů bez nutnosti generování kódu, čímž je pak lze efektivně využít uvnitř PTSQ endpointu jako definice výstupu.

Díky možnosti sdílení kódu mezi serverem a klientem lze využít dvojí validace. Protože jsou validační schémata nutně definována pro tvorbu serverové části, použití této optimalizace je téměř bez práce. Pro tvorbu formulářů na frontendu pak lze použít například knihovnu React Hook Form.

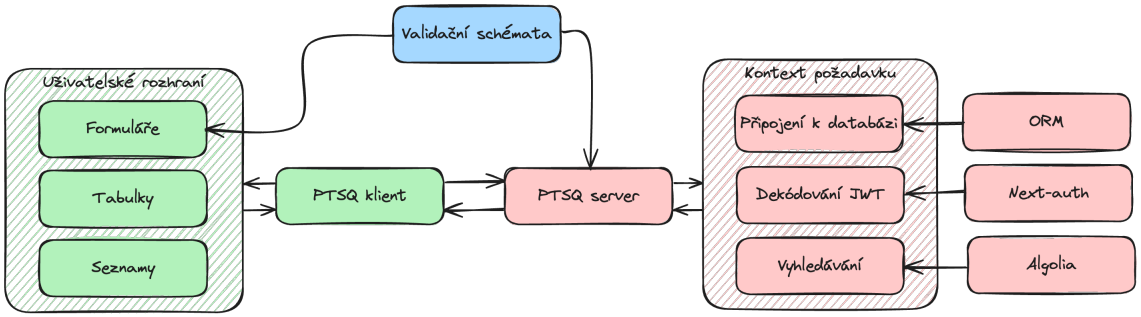
Pro optimalizaci v oblasti indexování stránek pro vyhledávače je možné využít vykreslování webové stránky na straně serveru. Tuto funkci podporuje téměř každý full-stack framework.

Invalidace dotazů a cachování na straně klienta pak umožňují vytvořit komplexní reaktivní uživatelské rozhraní s jednoduchou správou dat mezi komponentami při zajištění optimalizace síťového provozu.

[15] Služba Vercel je dostupná na webové stránce https://vercel.com/.
[16] Služba DigitalOcean je dostupná na webové stránce https://www.digitalocean.com/.
[17] Služba Render je dostupná na webové stránce https://render.com/.
[18] Nástroj NodeMon je dostupný na webové stránce https://nodemon.io/.
[19] Prisma ORM je dostupná na webové stránce https://www.prisma.io/.
[20] MikroORM je dostupné na webové stránce https://mikro-orm.io/.
[21] Drizzle ORM je dostupné na webové stránce https://orm.drizzle.team/.

Možnost tvorby vlastního kontextu požadavku na straně serveru zas umožňuje snadnou integraci knihoven třetích stran.

Díky možnosti integračních i jednotkových testů bez nutnosti tvorby HTTP serveru knihovna podporuje tvorbu plně otestovaných webových aplikací.



Obrázek 4.8 Komplexní webová aplikace s využitím PTSQ knihovny

4.22 Perspektivy a možnosti rozvoje knihovny

Tato podkapitola analyzuje a popisuje vlastnosti, které jsou buď potenciálními rozšířeními stávající funkcionality nebo klíčovými prvky, které by měla knihovna PTSQ obsahovat.

Jedna z klíčových funkcí je schopnost definovat vlastní chybové kódy, které může endpoint vrátit. Tato vlastnost PTSQ je v současné době ve fázi vývoje a vyžaduje především praktické ověření a testování návrhu. Jedním z hlavních přínosů by byla možnost definovat vlastní chybové kódy, na něž by mohly existovat překlady na straně klienta. Protože by tyto kódy chyb byly dostupné i v samotném schématu, umožnilo by to plně pokrýt překlady všech chybových odpovědí i externím klientům.

```
resolver
  .error({ code: 'CUSTOM_ERROR', httpStatus: 500 })
  .use(({ PtsqError, ctx, next }) => {
    if(!ctx.user) throw new PtsqError({ code: 'CUSTOM_ERROR' });

    return next();
  });
```

Výpis kódu 44 Vlastní kódy chybových odpovědí serveru

Další perspektivou je zdokonalení typové bezpečnosti linků na klientské straně. Podpora bezpečných linků z hlediska TypeScript typů prostřednictvím výběru endpointů by zamezila špatnému přístupu k objektu vstupu nebo výstupu. Linky by se vytvářely stejně jako middleware na serveru, tedy pomocí speciálního builderu klienta. Ten by automaticky znal datový typ všech endpointů a jejich vstupů i výstupů, to plyne z podstaty tvorby silně otypovaného klienta, jaký je v knihovně implementován již v této fázi.

Díky vytvoření podmínky na daný endpoint uvnitř linku by TypeScript dokázal napovídat správné datové typy. Typovaní vstupů a výstupů pomocí podmíněné selekce endpointu lze zajistit spojením všech variant objektu `meta` (kód 45). Získání takového typu není nijak složitý proces a nevyžaduje příliš úprav z hlediska implementace, stačí jen aplikovat několik typových transformací na typ kořenového routeru nebo schématu z introspekce. Metoda `castResponse` by pak mohla sloužit k efektivnímu přetypování odpovědi v dané selekci názvu endpointu.

```
type Meta = {
  route: 'user.create';
  input: {
    username: string;
  };
  type: 'mutation'
  castResponse: (Response) => UserCreateResponse;
} | {
  route: 'user.update';
  input: {
    id: string;
    username?: string;
  };
  type: 'mutation';
  castResponse: (Response) => UserUpdateResponse;
};
```

Výpis kódu 45 Typově bezpečné linky – Nový typ objektu meta

Další funkcionalitou by mohla být tvorba vlastních terminating linků. Aktuálně knihovna v modulu klienta poskytuje pouze jeden předdefinovaný terminating link a v plánu tak je možnost pro tvorbu vlastního. Ten by vždy byl v řetězci linků specifikován jako poslední a staral by se o posílání požadavků na HTTP server a přijímání odpovědí, čímž by toto chování šlo upravovat. Mohl by zároveň nahradit aktuální způsob definování vlastní `fetch` funkce.

Poslední možnou perspektivou by mohlo být vytvoření vlastního builderu validačních JSON schémat. V aktuální podobě PTSQ využívá pro tvorbu JSON schémat knihovnu Typebox. Ta sice umožňuje odvodit datový typ validních dat na základě schématu, avšak je k tomu zapotřebí využít přímo Typebox typů. Na straně klienta se při odvozování typů ze schématu z introspekce využívá jiné knihovny. Typebox totiž nedokáže odvodit datový typ JSON schémat jiných než těch, který formuje. Vytvořením vlastního builderu validačních schémat by se tak docílilo konzistencí mezi odvozením typů z kořenového routeru a ze schématu z introspekce.

4.23 Nevhodné návrhy knihovny

V některých případech se očekávání a skutečnost návrhů knihovny neshodovaly a v této podkapitole rozebereme nejzásadnější nevhodné návrhy při tvorbě knihovny PTSQ.

Validační schémata Zod

Při prvotním návrhu knihovny bylo využito validačních schémat Zod. Tato volba se bohužel ukázala jako nevhodná, protože Zod nativně neposkytuje možnosti pro převod schématu do formátu JSON pro možnost introspekce. Schémata tak byla převáděna knihovnou PTSQ na základě jejich definic, to však vedlo k nedostatečnému pokrytí všech validačních schémat Po důkladném testování bylo zjištěno, že plně podpořit převod Zod schémat do JSON struktury není možné. To vedlo k průzkumu validačních knihoven a nahrazení knihovny Zod knihovnou Typebox, která pro validace vytváří přímo JSON schémata.

Model bez introspekce

Původní myšlenka byla silně inspirována projekty Zodios a TS-REST. Zahrnovala psaní tří vrstev včetně schématu a distribuci schématu jako NPM balíčku. Po implementaci tohoto přístupu bylo dospěno k závěru, že takový způsob je velmi náročný a neefektivní. Projekt byl přepracován do modelu s podporou introspekce a automaticky generované vrstvy schématu.

Transformace mimo validační schéma

PTSQ využívá k transformacím, tedy možnosti automatické serializace a deserializace hodnot, validačních schémat. Tento přístup není ideální zejména kvůli možnosti sdílení schématu mezi klientem a serverem pro tvorbu dvojí validace. Omezení v podobě transformací ve většině případů není zdrojem problémů, knihovny pro tvorbu formulářů na straně klienta většinou data pouze validují, nikoliv transformují. V některých případech to však může být limitující a následkem může být nutnost definovat separátní validační schéma pro formulář na frontendu.

Knihovna se nejdříve snažila o podobný přístup, který využívá tRPC. Využitím transformátorů ale API ztrácí možnost být otevřené, protože transformátory na straně serveru i klienta musejí být totožné. Sdílení transformátorů na úrovni introspekce schématu pak nelze docílit.

Další možností, která byla vyzkoušena a následně také nevyužita, byla tvorba transformací mimo validační schémata. To bylo velmi neefektivní a přidělávalo to mnoho práce při psaní backendu (kód 46).

```
resolver
  .args(
    Type.Object({
      url: Type.String()
    })
  )
  .transform({
    url: (value) => new URL(value)
  })
  .query(...)
```

Výpis kódu 46 Ukázka transformací mimo validační schémata

Po zvážení a vyzkoušení těchto různých možností bylo usneseno, že definování transformací na úrovni validačních schémat je nejlepší, avšak neideální možnost.

TypeScript typy s výchozími hodnotami

Tento problém se týká pouze vývoje knihovny a na samotné používání nemá žádný vliv. Knihovna při definici generických typů využívala výchozích hodnot. Tento přístup v projektu s typovaným kódem těchto rozměrů není příliš vhodný, protože výrazně snižuje čitelnost kódu. Dovoluje totiž definovat jen některé generické typy a ostatní vynechat, čímž se jim nastaví výchozí hodnota. Při vývoji tento způsob velmi znesnadňoval orientaci mezi datovými typy. Řešením byly generické typy bez výchozích hodnot a obecné objekty byly definovány s předponou „Any“, například `AnyQuery` nebo `AnyMiddleware`.

Kapitola 5

Porovnání s ostatními nástroji

Porovnávání s již existujícími nástroji poukazuje na výhody a přednosti knihovny, zároveň však zobrazuje její možné nedostatky. Takové porovnání je velmi důležité, jelikož knihovna se snaží být konkurenceschopná. Při srovnání je věnována pozornost především nástrojům, knihovnám nebo frameworkům v ekosystému jazyka TypeScript nebo JavaScript. Porovnání nevěnuje příliš velkou pozornost architekturám jako takovým. Zaměřuje se především na možnosti a náročnost tvorby silně otypovaného API a možnosti nasazení aplikace v různých prostředích.

Porovnání s frameworky pro tvorbu REST API

Architektura REST ani nástroje pro tvorbu REST API většinou přímo nepodporují silné typování. Pro přehled endpointů, povolených datových typů na vstupu a typů výstupů se většinou vytváří webová dokumentace popisující rozhraní služby. PTSQ také podporuje možnost vytvořit webovou dokumentaci. Na rozdíl od frameworků pro tvorbu REST API ale vytváří schéma API a umožňuje jeho následný export pomocí introspekce. Tímto schématem přináší typování a popis služby přímo do vývojového prostředí. Samotná webová dokumentace bez typových předpokladů pak není příliš praktická. Vývojáři třetích stran ji musí neustále konzultovat a vývoj se tak stává neplynulý a zdlouhavý. Typováním uvnitř vývojových prostředí se docílí plynulosti a efektivity s dotazováním se služby.

Mezi nejznámější frameworky pro tvorbu REST API uvnitř Node.js patří zejména Express, Koa nebo Fastify. Žádný z těchto frameworků nepodporuje přímo možnost vytvoření silně otypovaného API. Většina pak dokonce neposkytuje ani možnost vytvořit vlastní kontext požadavku. Některé frameworky podporují vytvoření middleware, protože však neposkytují kontext požadavku, úpravy uvnitř middleware jsou většinou aplikovány nestandardním způsobem přímo na standardní objekt požadavku. Žádný ze zmíněných frameworků pak nepodporuje typově bezpečnou úpravu kontextu jako podporuje PTSQ.

I ve srovnání možnostech snadného nasazení aplikace předčí knihovna PTSQ všechny zmíněné frameworky pro tvorbu REST API. Ani jeden z nich totiž nepodporuje snadné nasazení v jiném prostředí než v Node.js, zatímco PTSQ podporuje širokou škálu frameworků a prostředí. Express nebo Koa lze sice také nasadit v různých prostředích, to však vyžaduje velmi odlišný přístup v každém tomto prostředí a může to tak být zdrojem komplikací.

Samotná náročnost tvorby API je pak velmi srovnatelná. Tvorba REST API a PTSQ API je podobně náročná. PTSQ ale nabízí automatickou validaci vstupů i výstupů a typové předpoklady jak na straně serveru, tak i klienta. Při vyžadování validací vstupních hodnot by se dokonce dalo konstatovat, že tvorba PTSQ API je jednodušší než tvorba REST API pomocí vyjmenovaných frameworků. Knihovna totiž umožňuje validace velmi snadno abstrahovat z finálního zpracování

požadavků.

Knihovny jako Zodios nebo TS-REST umožňují přidat validační schémat existujícímu REST API. Tím sice vytváří typové předpoklady na klientovi i serveru, schémata služeb se však distribuují jako NPM balíčky. To není pro tvorbu otevřeného API příliš praktické, protože externí klienti potřebují znalost daného NPM balíčku. PTSQ podporuje introspekci a tím vytváří možnost získat schéma rozhraní přímo ze zdroje serveru.

Jedním z nejmodernějších nástrojů pro tvorbu REST API je feTS. Ten využívá, stejně jako PTSQ, Typebox JSON validačních schémat, čímž vytváří typové předpoklady pro klienta i server. Umožňuje také vytvoření silně otypovaného otevřeného API díky exportu schématu rozhraní ve formě Open API specification (OAS). Knihovna feTS podporuje, stejně jako PTSQ, nasazení aplikace uvnitř několika prostředí, jako je Node.js, Bun, Deno nebo serverless prostředí. Hlavní nevýhodou feTS je náročnost tvorby otypovaného API. Je třeba definovat jak úspěšné výstupy endpointů tak i všechny chybové, a to bez možnosti tvorby předpřipravených komponent, které PTSQ podporuje ve formě resolveru. Tvorba API je tak velmi zdlouhavá a náročná. Další nevýhodou je nemožnost definovat middleware pouze jednoho endpointu. Knihovna feTS poskytuje middleware ve formě pluginů, které jsou aplikovatelné pouze na celý server.

Porovnání s GraphQL

Je důležité zdůraznit, že GraphQL se nezaměřuje pouze na vytváření silně typovaného API. Nabízí mnohem více a jeho flexibilní model a možnosti definování dotazů na klientovi jsou téměř nepřekonatelné. Samotné GraphQL má nevýhody, zejména co se týče pomalé validace polí oproti GraphQL schématu nebo velmi komplexní tvorby, která často vyžaduje znalost a nastavení spousty dalších nástrojů.

Zatímco GraphQL jako takové se zaměřuje na kompatibilitu s většinou programovacích jazyků, PTSQ podporuje pouze programovací jazyk TypeScript nebo JavaScript. PTSQ je určené pro full-stack vývoj v jazyce TypeScript, který je v poslední době velmi populární a umožňuje výhody, kterých při použitích jiných jazyků na straně serveru nelze dosáhnout.

PTSQ je zaměřené na snadné vytvoření silně typovaného otevřeného API při poskytnutí typové bezpečnosti při psaní kódu uvnitř knihovny. Nemá tedy smysl srovnávat model explicitního výběru dat nebo vnořování entit, které poskytuje GraphQL. PTSQ tyto možnosti nenabízí a ani se o to žádným způsobem nesnaží.

Protože standardní JavaScriptová knihovna GraphQL nepodporuje typově bezpečný kód při vytváření serveru, bude se srovnání zaměřovat především na generátory schémat, které byly popsány v podkapitole 3.10.

Nejpoužívanějšími takovými generátory v ekosystému jazyka TypeScript jsou Nexus a Pothos. Oba tyto generátory schémat poskytují typově bezpečný kód na straně serveru. Ovšem pro vytvoření GraphQL schématu je nutné v obou případech využít generování kódu. PTSQ nevyžaduje generování kódu při vytváření serveru, ani pro vytvoření schématu API. Při lokálním vývoji jsou tak změny ze serveru ihned reflektovány na stranu klienta díky sdílení TypeScript typu kořenového routeru aplikace. Generování GraphQL schémat neumožňuje takto plynulý přenos datových typů nebo schématu.

PTSQ i GraphQL umožňuje vytvářet vlastní kontext požadavku. Standardní knihovna GraphQL ani jeden z generátorů schémat ale nepodporují typově bezpečné přenášení kontextu mezi middleware, které umožňuje PTSQ. Kontext požadavku GraphQL tak nemusí vždy nabízet odpovídající datový typ skutečné hodnoty. To často vyžaduje manuální přetypování s čímž může být spojena vyšší chybovost.

Oba přístupy pak umožňují abstrahovat validaci vstupů i výstupů z finálního zpracování požadavku. Jak v případě PTSQ, tak i v případě GraphQL generátorů, poskytují validační schémata nejen validace v běhové úrovni, ale i v typové. PTSQ vykonává validace pomocí JSON validačních schémat, která jsou navíc kompilovaná, krok validace je tak velmi efektivní. GraphQL

vždy provádí validace na základě GraphQL schématu. To sice umožňuje validovat data oproti datovému typu GraphQL, taková validace je však ve většině případů nedostatečná. Je tedy většinou třeba definovat vlastní validační schéma, které umožňuje komplexní validaci dat. Tím ale dochází k opakování validačního kroku, protože část validace, kterou provede GraphQL schéma, musí poté provést znovu validace na úrovni validačního schématu. JSON schémata, která jsou využita v PTSQ, umožňují rovnou definovat vcelku komplexní validace. Hlavní výhodou pak je možnost sdílet toto JSON schéma mezi serverem a klientem pro vytvoření dvojí validace.

GraphQL i PTSQ umožňují transformovat data před přijetím nebo před odesláním. Tyto transformace často serializují nebo deserializují data pro možnosti přenosu. Oba nástroje podporují tyto transformace mimo samotné finální zpracování požadavku, čímž se toto zpracování stává velmi čitelným a méně chybovým.

Hlavními výhodami PTSQ je rychlé a jednoduché vytvoření silně otypovaného API bez nutnosti generování kódu, a to i při spojení s PTSQ klientem. GraphQL díky nutnosti generování schématu takové možnosti neposkytuje, a to ani při použití jednotného repozitáře.

V jazyce JavaScript existuje několik poskytovatelů GraphQL serverů, mezi nejznámější patří GraphQL Apollo nebo GraphQL Yoga. Většina serverů umožňuje nasadit API v mnoha prostředích podobně jako knihovna PTSQ.

Porovnání s tRPC

Hlavními rozdíly mezi tRPC a PTSQ je především otevřenost API. Zatímco PTSQ umožňuje vytvořit silně otypované otevřené API, tRPC to nepodporuje. Typování je v případě neveřejného tRPC projektu pouze proprietární a tím je znemožněno vytváření otypovaných externích klientů využívajících API. Nevyžaduje sice generování kódu, ale právě toto omezení tRPC bylo jedním z hlavních motivací vytvoření knihovny PTSQ. Nemožnost vytvoření silně otypovaného otevřeného API se projevuje nejen na samotné otevřenosti pro externí klienty, ale i na striktní struktuře projektu, kterou tRPC vyžaduje. Jednotný repozitář je pro použití tRPC téměř nevyhnutelný. PTSQ nabízí silně otypované otevřené API, čímž všechny tyto popsané nevýhody tRPC eliminuje. Zároveň však zachovává jednoduchost tvorby projektu a nevyžaduje, stejně jako tRPC, generování kódu pro vytvoření silně otypovaného API.

Oba nástroje pak umožňují tvorbu transformací dat. To se hodí pro serializaci a deserializaci dat přenášených mezi klientem a serverem. PTSQ využívá k transformacím validačních schémat, a to především kvůli otevřenosti API. Transformátor se totiž nedá sdílet pomocí introspekce schématu ve formátu JSON. Klient i server ale musí definovat totožný transformátor pro správou serializaci a deserializaci hodnot, transformátory tak pro otevřená API nedávají smysl.

Knihovna tRPC umožňuje využívat různá validační schémata pro definici vstupů i výstupů. PTSQ pak podporuje pouze Typebox jakožto builder validačních schémat.

Obě knihovny je pak možné nasadit v mnoha prostředích a obě podporují tvorbu vlastních klientů pro různé frontendové frameworky. Zatímco tRPC nativně vytváří pouze React klienta, PTSQ vytváří jak React, tak i Svelte klienta.

Kapitola 6
Implementace

Nástroje použité pro samotnou implementaci knihovny a správu repozitáře. Kapitola popisuje strukturu projektu, testování a pokrytí, tvorbu webové dokumentace a nasazení na službu NPM.

Repozitář

Repozitář celého projektu knihovny PTSQ byl strukturován jako monorepozitář. Tato struktura umožňuje v takovém projektu vytvořit několik balíčků (modulů), které jsou na sobě závislé a vzájemně provázané. To je pro tuto práci velmi vhodné, protože knihovna vytváří několik adaptérů pro frontendové frameworky, vytváří modul serveru a pak další nástroje, jako je CLI pro introspekci. Monorepozitář tak umožňuje všechny tyto moduly spravovat v jediné kódové základně. Zároveň umožňuje snadné přidávání a integraci nových modulů, například pro tvorbu dalších frontendových adaptérů. Tím se repozitář stává velmi dobře rozšiřitelným. Uvnitř tohoto repozitáře je pak také možné uchovávat zdrojový kód webové dokumentace knihovny a různé ukázkové projekty. Pro uživatele knihovny je tak velmi snadné najít jakýkoliv zdrojový kód ukázkového projektu, webové dokumentace nebo samotné knihovny, vše je totiž obsaženo na jediném místě.

Tato struktura repozitáře zároveň velmi výrazně zrychluje vývoj a zvyšuje flexibilitu při práci. Tvorba testů napříč celým projektem a správa závislostí mezi jednotlivými lokálními moduly je velmi snadná. Monorepozitář zároveň zefektivňuje refaktoring kódu, nebo nasazení několika modulů knihovny najednou.

Pro správu jednotného repozitáře bylo využito nástroje Turborepo. Na rozdíl od alternativy NX poskytuje jednodušší konfiguraci projektu a správu lokálních závislostí uvnitř jednotného repozitáře. Turborepo především ukládá již zkompilované moduly monorepa do mezipaměti. Při úpravě jednoho z modulů tak není nutné znovu kompilovat všechny jeho závislosti nebo celý obsah repozitáře.

Verzování kódu bylo zajištěno verzovacím systém GIT a pro správu vzdálených repozitářů byla využita platforma GitHub. GitHub poskytuje širokou škálu nástrojů pro spolupráci a správu kódové základny.
Vzdálený repozitář knihovny PTSQ se pak nachází na webové stránce https://github.com/lewis-wow/ptsq.

GitHub nabízí různá omezení pro nahrávání kódu do hlavní vývojové větve master. Bylo tak vytvořeno několik pravidel, jako je zákaz násilného odeslání (force push) do hlavní větve, vyžadování vytvoření pull requestu před sjednocením kódu do hlavní větve nebo úspěšné vyhodnocení všech testů.

Pro nasazení knihovny na službu NPM, nasazení webové dokumentace a spouštění testů

včetně pokrytí bylo využito GitHub Actions. Ty poskytují automatizace ve formě kontinuální integrace. Při nahrání kódu nebo vytvoření pull requestu do hlavní větve master tak byla knihovna automaticky sestavena a byly spuštěny všechny testy a změřeno pokrytí kódu. Pokud se navíc nejednalo o vytvoření pull requestu do hlavní větvě, nýbrž o nahrání kódu, všechny moduly knihovny byly, v závislosti na nastavených verzích, automaticky publikovány na službu NPM.

Při vývoji byly pull requesty squashovány, což znamená, že při sjednocení pull requestu se do master větve přidal pouze jeden commit zahrnující všechny změny. Pro přehlednost pak byly pull requesty pojmenovány pomocí vlastního pravidla `[package, ...package[]] description`. To umožňuje jednoduchou orientaci mezi jednotlivými commity v hlavní větvi a zároveň mezi otevřenými i uzavřenými pull requesty.

Testování a pokrytí

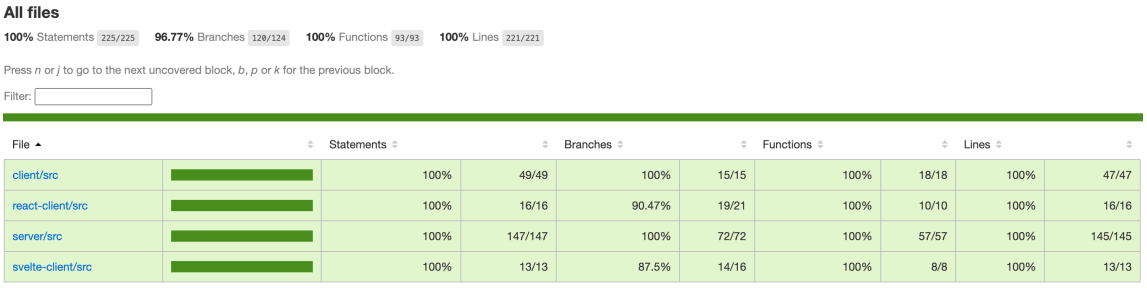
Testování je jedním z nejdůležitějších procesů při programování knihovny. Díky testování je možné určit, zda je knihovna funkční. Lze také na základě automatického testování vyhodnotit pokrytí kódu, tedy řádky zdrojového kódu, které byly během testů spuštěny. Během implementace byly vytvořeny jak jednotkové testy pro ověření základních funkcionalit různých komponent, tak i integrační testy pro testování komunikace klienta a serveru.

Pro testování byl využit nástroj Vitest[1]. Tato moderní alternativa Jestu[2] umožňuje jednoduše vytvářet jednotkové i integrační testy, a to jak v JavaScriptu, tak i TypeScriptu. Oproti Jestu nevyžaduje velké množství konfigurace, především pak při testování v jazyce TypeScript uvnitř jednotného repozitáře.

Pokrytí bylo měřeno nástrojem Istanbul[3]. To je jeden z nejznámějších a nejspolehlivějších nástrojů pro měření pokrytí v JavaScriptu. Jelikož Vitest poskytuje plugin pro integraci Istanbulu, bylo spojení těchto nástrojů velmi snadné.

Vitest pak umožňuje spouštění testů ve watch módu. To znamená, že při úpravě kódu nebo testového souboru jsou testy v dotčeném souboru spuštěny znovu. Proces testování a hledání chyb v kódu je díky tomuto módu velmi efektivní. Zároveň Vitest umožňuje spouštět pouze vybrané testy a podporuje strukturu projektu jakožto monorepozitáře. Pomocí Vitest workspace lze spravovat a spouštět testy a pokrytí napříč všemi balíčky a moduly pomocí jednoho příkazu.

Samotná úroveň pokrytí pak byla dosažena na 100 %, to znamená, že téměř veškerý kód byl během testování spuštěn. Je důležité podotknout, že TypeScript typy nelze testovat nástroji jako je Vitest a je potřeba je testovat pomocí speciálních typů. Datové typy a typový kód tedy nijak neovlivňují úroveň pokrytí ani samotné testy. Pro reprezentaci výsledků pokrytí ve vzdáleném repozitáři na GitHubu pak bylo využito nástroje Codecov[4]. Ten přehledně a veřejně zobrazuje výsledné pokrytí aktuální verze knihovny na webové stránce.



File		Statements		Branches		Functions		Lines	
client/src		100%	49/49	100%	15/15	100%	18/18	100%	47/47
react-client/src		100%	16/16	90.47%	19/21	100%	10/10	100%	16/16
server/src		100%	147/147	100%	72/72	100%	57/57	100%	145/145
svelte-client/src		100%	13/13	87.5%	14/16	100%	8/8	100%	13/13

Obrázek 6.1 Výstup pokrytí modulů

[1] Vitest je dostupný na webové stránce https://vitest.dev/.
[2] Jest je dostupný na webové stránce https://jestjs.io/.
[3] Istanbul je nástroj pro měření pokrytí testů a je dostupný na webové stránce https://istanbul.js.org/.
[4] Codecov je dostupný na webové stránce https://about.codecov.io/.

Pro testování byl vytvořen speciální interní modul, který není dostupný na službě NPM a slouží k vytvoření jednoduchého testovacího HTTP serveru. Server je vytvořen tak, aby ho bylo možné zavřít. Metoda HTTP serveru `Server.close` sice umožňuje blokovat nová příchozí spojení, čeká však na ukončení všech aktuálních spojení se serverem. Testovací server si ukládá všechna spojení a při zavření serveru tyto spojení ukončí pomocí `Socket.destroy`. Tím je docíleno, že po konci testu využívajícího server, nebude tento server stále běžet a blokovat tak ostatní testy.

Při testování nebyla ověřována efektivita knihovny prostřednictvím benchmark testů. Efektivita totiž velmi závisí na samotné službě a její implementaci, případně pak na implementaci ostatních nástrojů jako databázových ORM nebo autentizačních knihoven. Knihovna Typebox, která byla využita pro definování validačních schémat a validací, poskytuje přehled testů efektivity v její dokumentaci.

Nástroje pro vývoj projektu

Pro správu závislostí a balíčků byl zvolen nástroj PNPM, který umožňuje efektivní ukládání NPM balíčků a zároveň podporuje strukturu jednotného repozitáře. Díky PNPM workspace umožňuje spravovat závislosti mezi všemi moduly uvnitř monorepa a to jak lokálními, tedy v jiné složce monorepozitáře, tak i vzdálenými externími závislostmi. Umožňuje vytvářet selektivní verze závislostí mezi různými moduly repozitáře a podporuje vynucení instalace balíčku redundantně. To je vhodné zejména pro vytváření ukázkových projektů knihovny, které musejí fungovat i jako samostatné projekty bez celkového monorepozitáře.

Tvorba samotné knihovny pak vyžadovala použití bundleru. Úlohou bundleru je, mimo jiné, transformovat kód mezi různými systémy pro tvorbu a správu modulů. Využití bundleru je tak velmi důležité pro podporu obou modulových systémů, CommonJS i ES modules, které se v JavaScriptu běžně používají. Podpora obou systémů modulů dává uživateli knihovny možnost zvolit si, jakým stylem chce kód psát. Zároveň některé ostatní frameworky nebo projekty podporují pouze jeden typ modulů. Vytvořením kódu pro oba typy se tak rozšiřuje možnost použití knihovny uvnitř různých prostředí. Jako bundler byl zvolen nástroj Rollup. Kromě transformování na oba typy modulů byl Rollup v projektu využit k sjednocení všech souborů do jednoho, tím se výrazně redukuje velikost projektu v běhovém prostředí. Hlavním důvodem zvolení Rollupu jakožto bundleru je jednoduchost konfigurace a spousta pluginů, které Rollup nabízí.

Pro nasazení jednotlivých modulů na službu NPM bylo využito nástroje Changesets. Ten umožňuje spravovat a publikovat více NPM balíčků najednou. Možnost nasazení několika balíčků najednou pak zjednodušila vytvoření CI skriptu pro automatické nasazení při nahrání kódu do hlavní vývojové větve vzdáleného repozitáře.

Při vývoji také byly využity další nástroje jako je ESlint pro statickou analýzu kódu, a to jak JavaScriptového, tak i TypeScriptového kódu. Pro automatické formátování souborů byl využit nástroj Prettier s pluginem pro řazení importů podle názvu a priorit, jako jsou externí a interní závislosti. Prettier i ESlint představují základ nástrojů při vývoji většího open-source projektu.

Webová dokumentace

Webová dokumentace byla vytvořena pomocí nástroje Nextra a je nasazena na webové stránce https://ptsq.vercel.app. Dokumentace je rozsáhlá a pokrývá většinu vlastností knihovny a vysvětlení různých funkcionalit pro pochopení ostatními vývojáři. Zároveň dává vývojářům vodítko, jak tvořit aplikace pomocí knihovny, když zobrazuje základní myšlenky a vzory.

Nástroj Nextra umožňuje jednoduché nasazení i tvorbu dokumentace ve formátu MDX. MDX je nadstavba značkovacího jazyka Markdown, která umožňuje vkládat do kódu React komponenty a obecně jakékoliv JSX elementy, tedy i HTML. Nextra je knihovna postavená nad frameworkem Next.js a kromě možnosti psát webovou dokumentaci ve formátu MDX podporuje

jednoduché tvoření cest webové stránky pomocí file-system routeru, překlady, verzování a vyhledávání v dokumentaci. Díky Next.js pak automaticky podporuje vykreslování na straně serveru, což je přívětivé pro SEO. Alternativou nástroje Nextra je větší projekt Docusaurus od společnosti Facebook, který poskytuje mnohem více funkcí a pluginů. Pro líbivý vzhled a funkcionality, jako je vyhledávání, je však potřeba Docusaurus správně nastavit. V tomto ohledu je tak Nextra jednodušší. Nextra při sestavování dokumentace automaticky indexuje webové stránky pro pozdější možnosti vyhledávání.

Při tvorbě dokumentace bylo nahráno několik krátkých videí, které jsou součástí samotné webové dokumentace. Videa pokrývají různé vlastnosti knihovny jako je tvorba middleware, vytváření endpointů nebo testovacího calleru. Tato videa mají za cíl poukázat na typovou bezpečnost, plynulost a efektivní sdílení schématu PTSQ API.

Webová dokumentace poskytuje playground prostředí jednoho z mnoha ukázkových projektů knihovny na adrese https://ptsq.vercel.app/tryit.

Pro nasazení dokumentace bylo využito služby Vercel, která umožňuje snadné nasazení frameworku Next.js a podporuje monorepo s nástrojem Turborepo.

Kapitola 7
Závěr

V rámci této bakalářské práce byl úspěšně řešen problém vytvoření služby generující silně otypovaného otevřeného API. Práce detailně analyzovala existující metody řešení pomocí různých nástrojů pro tvorbu API a silně otypovaného rozhraní. Díky výzkumu byly identifikovány jak výhody, tak nedostatky různých technologií.

Na základě této analýzy byla vyvinuta knihovna PTSQ, která poskytuje uživatelům jednoduché rozhraní pro tvorbu silně otypovaného otevřeného API a současně se snaží eliminovat nedostatky existujících nástrojů. PTSQ je navržena tak, aby reflektovala výhody analyzovaných nástrojů a podporovala jednoduchost tvorby projektů. Dále umožňuje vytvoření schématu služby bez nutnosti generování kódu a poskytuje export tohoto schématu ve formátu JSON pro externí klienty.

Rozsah knihovny PTSQ je značný, což dokládá repozitář s více než 150 sjednocenými pull requesty a téměř s 9 500 řádky kódu TypeScriptu. Její vývoj vyžadoval značné praktické zkušenosti s různými nástroji pro tvorbu silně otypovaných API, a to jak pro rozbor a eliminaci jejich nedostatků, tak pro inspiraci a využití jejich předností. Dále bylo nezbytné mít až nadprůměrné znalosti jazyka TypeScript a jeho typového systému, které byly klíčové pro úspěch této práce. Velmi rozsáhlé praktické zkušenosti v tvorbě komplexních dynamických webových aplikací a systému pomocí různých přístupů. Práce vyžadovala povědomí o moderních technologiích, které se používají na straně frontendu i backendu. Reaktivní frameworky, nástroje pro tvorbu REST API, full-stack frameworky, alternativní runtime prostředí, GraphQL, tRPC, databázové ORM, formuláře, validační schémata, dvojí validace, TypeScript a JavaScript, architektury a principy webové komunikace, Git a GitHub, kontinuální integrace a nasazení, dokumentace, testování, obecné požadavky na webové služby, praktické znalosti s tvorbou webových aplikací, to vše a mnohem více bylo potřeba pro vypracování této bakalářské práce.

V rámci práce byla vytvořena webová dokumentace, která detailně popisuje funkcionality knihovny a názorně ukazuje tvorbu silně typovaného API a možnosti propojení s ostatními nástroji pro tvorbu komplexní webové aplikace. Webová stránka dokumentace také poskytuje velmi jednoduchý projekt postavený na knihovně PTSQ, který lze vyzkoušet, upravit i spustit přímo v prohlížeči. Tento malý ukázkový projekt tak ukazuje základní principy a promítá typově bezpečnou práci s PTSQ.

V rámci testování bylo vytvořeno několik dalších větších ukázkových projektů s využitím knihovny PTSQ, přičemž vývoj a používání API bylo plynulé, velmi jednoduché a intuitivní. Knihovna byla také úspěšně testována v různých ekosystémech a ve spojení s mnoha jinými knihovnami nebo frameworky pro vytváření uživatelských rozhraní, správu databáze, vytvoření autentizace nebo tvorbu dynamických formulářů. Zejména s knihovnami kompatibilními s Next.js nebo React, jako jsou React Hook Form, Next-auth nebo Drizzle ORM, kde prokázala vysokou efektivitu a jednoduchost při tvorbě full-stack aplikace. Zapadá tak do celkového vývoje komplex-

ních dynamických webových aplikací a je kombinovatelná se spoustou nástrojů, které se v praxi velmi často používají, samotná knihovna pak poskytuje velmi kvalitní základ pro efektivní vývoj.

Závěrem lze konstatovat, že knihovna PTSQ prošla důkladným testováním návrhu rozhraní, a to jak prostřednictvím praktických aplikací, které testovaly zejména použitelnost knihovny v různých prostředích a s různými nástroji, tak automatických testů, přičemž byla dosažena 100% úroveň pokrytí. Jednotlivé moduly knihovny byly publikovány na službě NPM a jsou dostupné pro většinu prostředí počínaje Node.js, přes full-stack frameworky až po jiná alternativní prostředí a externí frameworky. Repozitář knihovny je veřejně dostupný na službě GitHub a je nastaven a připraven pro open-source vývoj.

Samotná knihovna pak představuje nejmodernější přístupy pro tvorbu silně otypovaného API a přichází s převratným modelem sdílení schématu webové služby pro interní i externí klienty. Umožňuje až nezvykle snadno a plynule propojit kód serveru a klienta, zároveň však neporušuje jejich oddělitelnost a nezávislý vývoj, a poskytuje tak typově bezpečné a efektivní rozhraní pro dotazování se API. Zároveň vytváří rozsáhlý a jednoduchý způsob integrace do systému, a to jak kompatibilitou nasazení, tak i podporou ostatních nástrojů v ekosystému.

Příloha A

Ukázkové projekty

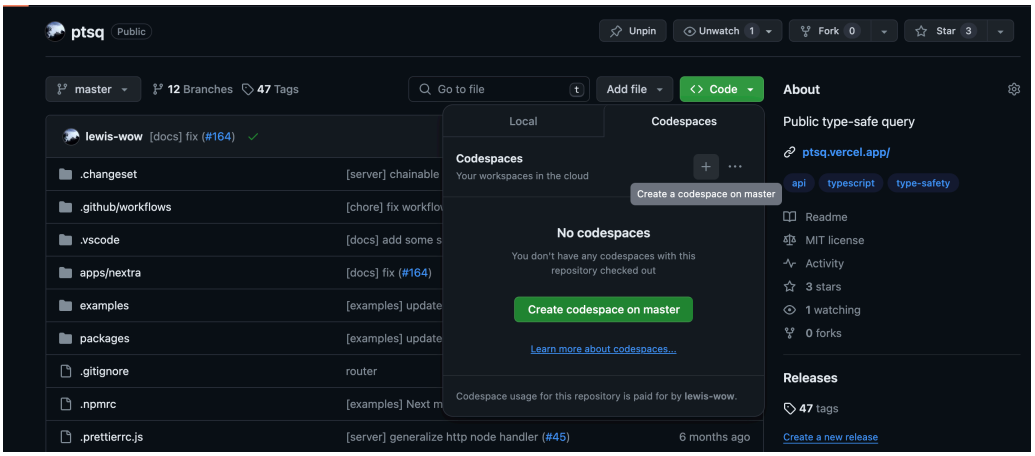
Tato příručka popisuje, jak lze spustit ukázkové projekty knihovny PTSQ a na co se při zkoumání těchto projektů zaměřit.

A.1 Prerekvizity

GitHub codespace

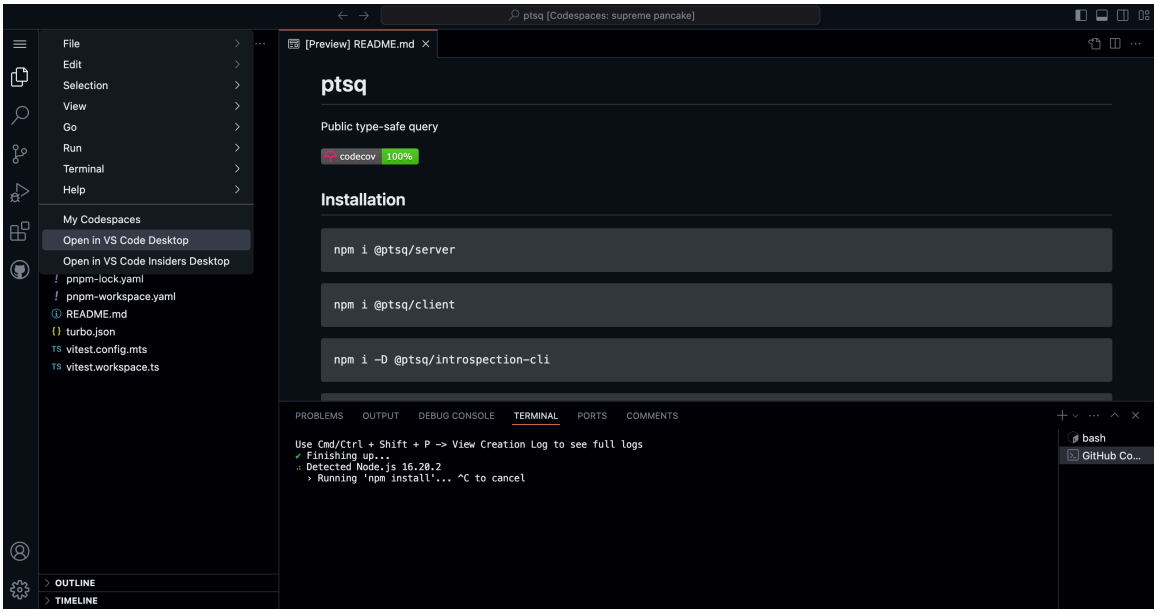
Pro spuštění a vyzkoušení ukázkových projektů doporučuji vytvořit codespace na službě GitHub. V takovém případě je potřeba mít nainstalované pouze Visual Studio Code, které je dostupné na adrese https://code.visualstudio.com/. Není již potřeba nic dalšího instalovat nebo konfigurovat lokálně.

Pro vytvoření codespace je potřeba přejít na webové stránky knihovny na GitHubu, která se nachází na adrese https://github.com/lewis-wow/ptsq. Kliknutím na tlačítko „Code“ a přejitím na záložku „Codespaces“ lze vytvořit codespace nad větví master (obrázek A.1).



Obrázek A.1 Vytvoření codespace

Codespace automaticky instaluje nejnovější verzi Node.js, Dockeru i Docker compose, není tak třeba nic dalšího nastavovat. Vytvořený virtuální stroj můžeme pomocí rozbalení menu otevřít přímo ve Visual Studio Code (obrázek A.2).



Obrázek A.2 Otevření codespace ve Visual Studio Code

Lokální počítač

Druhou, lehce náročnější, variantou je spustit projekt lokálně. Jediným požadavkem pro spuštění jednoduchých ukázek je Node.js ideálně poslední verze. Pro spuštění komplexnější ukázky použití knihovny je nutné mít nainstalovaný Docker compose verze 2. Doporučená cesta pro spuštění je instalace Docker desktop, kde je tato verze Docker compose nainstalována automaticky.
Docker desktop je dostupný na adrese https://www.docker.com/products/docker-desktop/.

Pro stažení samotných ukázkových projektů je možné využít verzovacího systému GIT a naklonovat repozitář se zdrojovými kódy knihovny i ukázkových projektů, který je na adrese https://github.com/lewis-wow/ptsq.git.
Pro naklonování lze použít příkaz: `git clone https://github.com/lewis-wow/ptsq.git`.
Druhou možností je využít příloh, kde je celý repozitář uložen v adresáři `ptsq`.

Protože je repozitář jednotný, jsou v něm uloženy nejen ukázkové projekty, ale i implementace samotné knihovny nebo webová dokumentace. Ukázkové projekty se nachází v adresáři `examples`.

Jednotlivé ukázkové projekty fungují zcela bez kompletního repozitáře, je tak možné a doporučené instalovat závislosti přímo v adresáři ukázkového projektu.

A.2 Ukázkový projekt basic

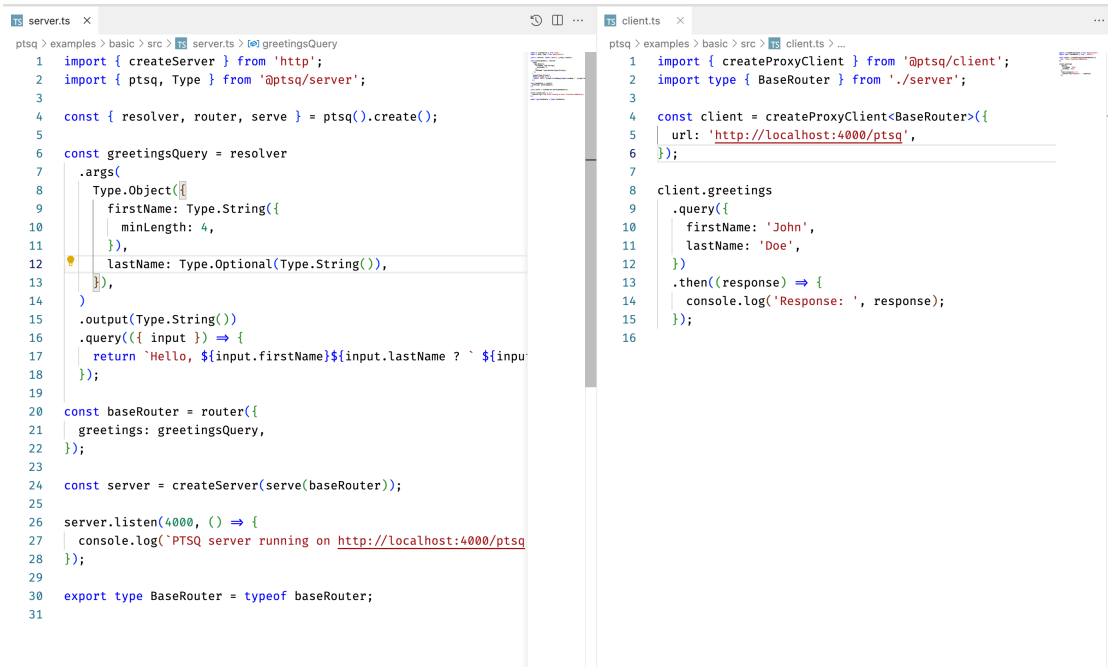
Tento projekt zobrazuje základní nastavení a fungování knihovny v co možná nejmenší a nejjednodušší formě.

Zde jsou uvedeny jednotlivé kroky pro instalaci a spuštění ukázkového projektu:

1. v příkazové řádce přejdeme do adresáře `examples/basic`.
2. Pro instalaci závislostí ukázkového projektu spustíme příkaz `npm install`.
3. Spuštění serverové části vyžaduje příkaz `npm run start:server`.
4. v jiné příkazové řádce ve stejném adresáři spustíme klienta `npm run start:client`.

Spuštění klienta by mělo vypsat do konzole odpověď serveru na klientův dotaz.

Asi nejdůležitější na tomto ukázkovém projektu je zkusit kód upravit, jedině tak lze poznat co všechno knihovna dokáže. Zdrojové kódy tohoto projektu se nacházejí v adresáři `src`. Při úpravě kódu doporučuji rozdělený editor, kdy na jedné straně bude soubor klienta (`src/client.ts`) a na druhé soubor serveru (`src/server.ts`), jako na obrázku A.3.



Obrázek A.3 Ukázka úpravy kódu – projekt basic

Po změně kódu je tak nejlépe vidět, jak je tvorba silně otypovaného API, které nevyžaduje generování kódu, plynulá. Pro zachycení změn v běhovém prostředí je nutné server vypnout a spustit ho znovu pomocí příslušného příkazu.

Je také možné vyzkoušet introspekci schématu pomocí příkazu `npm run generate:schema`. Místo typu kořenového routeru ve funkci `createProxyClient` pak lze typ dodat ze schématu z introspekce (kód 47).

```
import { BaseRouter } from './generated/schema.generated';

const client = createProxyClient<typeof BaseRouter>({
  url: 'http://localhost:4000/ptsq',
});
```

Výpis kódu 47 Vytvoření klienta se schématem z introspekce

Projekt basic nasazuje server v prostředí Node.js přímo pomocí standardního Node.js http modulu.

A.3 Ukázkový projekt next

Tento projekt vyobrazuje použití knihovny v ekosystému frameworku Next.js. Využívá React klienta a vytváří základní CRUD operace pro manipulaci s příspěvky, které jsou uloženy v PostgreSQL databázi. Projekt používá další nástroje, jako kontrolu proměnných prostředí pomocí Zod schématu nebo Prisma ORM.

Zde jsou uvedeny jednotlivé kroky pro instalaci a spuštění ukázkového projektu:

1. v příkazové řádce přejdeme do adresáře `examples/next`.
2. Pro instalaci závislostí ukázkového projektu spustíme příkaz `npm install`.
3. Spuštění databáze PostgreSQL lze provést pomocí příkazu `docker compose up`.
4. Databázi, která je nyní prázdná, je potřeba nejprve zmigrovat nástrojem Prisma ORM pomocí příkazu `npm run prisma:migrate`.
5. Protože Prisma vytváří databázového klienta na základě schématu databáze, je nutné tohoto klienta vygenerovat příkazem `npm run prisma:generate`.
6. Pro spuštění celé aplikace spustíme příkaz `npm run dev`.

Aplikace umožňuje vytvářet, upravovat, mazat a dotazovat se na příspěvky. Na straně klienta je zajímavé si povšimnout invalidování dotazů. Při vytvoření nebo úpravě příspěvku jsou data automaticky znovu dotázána a stránka je překreslena. Mutace dat se tak deklarativně šíří celou aplikací.

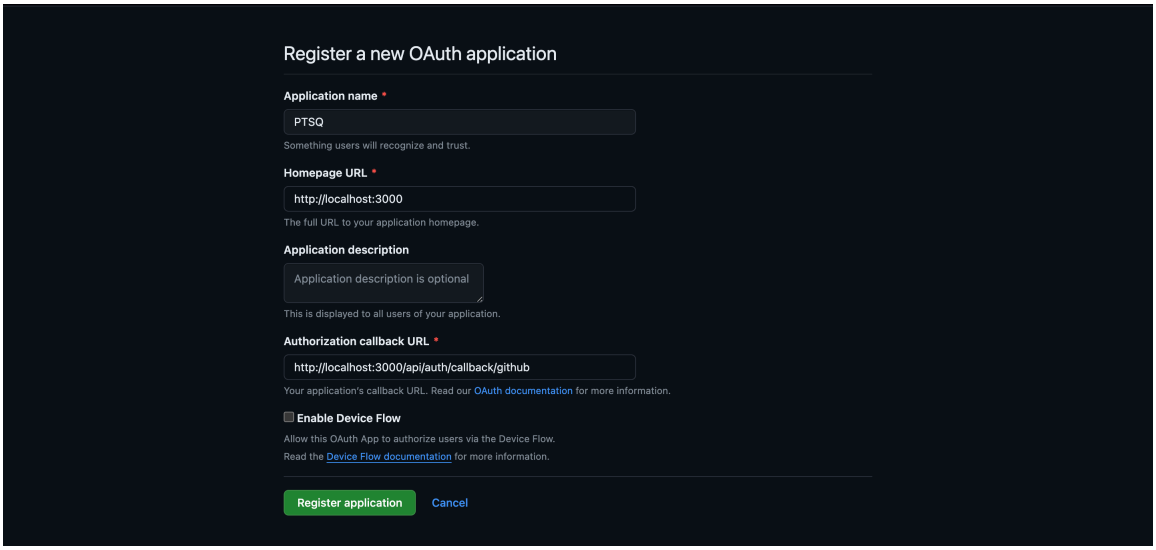
Další vlastností tohoto ukázkového projektu, které stojí za povšimnutí, je využití dvojí validace. Formuláře na straně klienta využívají stejných validačních schémat, která jsou využita na serveru pro definování argumentů endpointů. Nevalidní data se pak vůbec neodesílají na server a nezatěžují ho. Pro tvorbu formulářů byla použita knihovna React Hook Form a samotnou dvojí validaci lze v kódu vidět v komponentách v adresáři `src/components`, například `CreatePost.tsx`.

V souboru `src/pages/api/ptsq.ts` pak lze vidět snadné nasazení PTSQ aplikace uvnitř Next.js.

A.4 Ukázkový projekt next-auth

Dalším zajímavým ukázkovým projektem může být next-auth. Ten ukazuje základní práci s kontextem požadavku, kdy pomocí knihovny Next-auth autentizuje uživatele pomocí přihlašování přes GitHub.

Pro funkčnost tohoto projektu je nejprve potřeba nastavit ID a klíč (secret) aplikace pro přihlašování na platformě GitHub. Na webové stránce https://github.com/settings/developers lze zaregistrovat aplikace pro autentizaci přes OAuth[1]. Při registraci aplikace je nutné nastavit „Homepage URL“ na hodnotu `http://localhost:3000` a „Authorization callback URL“ na hodnotu `http://localhost:3000/api/auth/callback/github` (obrázek A.4). Parametr „Homepage URL“ na GitHubu opravňuje tuto URL pro vytváření požadavků k autentizaci, „Authorization callback URL“ zase nastavuje, na jakou adresu má GitHub přesměrovat v případě dokončení autentizace, v tomto případě využíváme cestu, kterou poskytuje knihovna Next-auth.



Obrázek A.4 Registrace OAuth aplikace – GitHub

Zde jsou pak uvedeny jednotlivé kroky pro instalaci a spuštění ukázkového projektu po nastavení OAuth aplikace:

1. v příkazové řádce přejdeme do adresáře `examples/next-auth`.
2. Pro instalaci závislostí ukázkového projektu spustíme příkaz `npm install`.
3. Spuštění databáze PostgreSQL lze provést pomocí příkazu `docker compose up`.
4. Databázi, která je nyní prázdná, je potřeba nejprve zmigrovat nástrojem Prisma ORM pomocí příkazu `npm run prisma:migrate`.
5. Protože Prisma vytváří databázového klienta na základě schématu databáze, je nutné tohoto klienta vygenerovat příkazem `npm run prisma:generate`.
6. Pro spuštění celé aplikace spustíme příkaz `npm run dev`.

Tato jednoduchá aplikace umožňuje přihlašovat, registrovat a odhlašovat uživatele pomocí OAuth při připojení uživatele ke kontextu požadavku PTSQ.

[1] OAuth je autorizační framework pro přihlašování se přes aplikace třetích stran bez poskytnutí citlivých údajů jiným aplikacím [66].

A.5 Ukázkový projekt next-full-application

Tento ukázkový projekt pak kombinuje přihlašování a registrace s vytvářením a manipulací s příspěvky. Vyobrazuje jednoduché použití kontextu uvnitř handlerů když dochází k úpravě nebo vylistování jednotlivých příspěvků. API vytvořené knihovnou dodržuje omezení přihlášeného uživatele a vrací jen takové příspěvky, které daný uživatel vytvořil. Zároveň jiným uživatelům neumožňuje jakkoliv měnit cizí příspěvky.

Projekt používá framework Next.js, Prisma ORM a Next-auth pro přihlašování a registraci. Stejně jako u projektu Ukázkový projekt next, lze i tady pozorovat chování invalidace dotazů a jejich významné usnadnění práce. Vytvořením kontextu požadavku zobrazuje projekt jednoduché zasazení dalších nástrojů v PTSQ serveru.

Pro kompletní funkčnost tohoto projektu je nejprve potřeba nastavit ID a klíč (secret) aplikace pro přihlašování a registraci pomocí OAuth na platformě GitHub, stejně jako tomu bylo u projektu Ukázkový projekt next-auth. Toto nastavení však není nutné, protože ukázkový projekt poskytuje i lokální přihlašování a registraci bez využití OAuth.

Zde jsou uvedeny jednotlivé kroky pro instalaci a spuštění ukázkového projektu:

1. v příkazové řádce přejdeme do adresáře `examples/next-full-application`.
2. Pro instalaci závislostí ukázkového projektu spustíme příkaz `npm install`.
3. Spuštění databáze PostgreSQL lze provést pomocí příkazu `docker compose up`.
4. Databázi, která je nyní prázdná, je potřeba nejprve zmigrovat nástrojem Prisma ORM pomocí příkazu `npm run prisma:migrate`.
5. Protože Prisma vytváří databázového klienta na základě schématu databáze, je nutné tohoto klienta vygenerovat příkazem `npm run prisma:generate`.
6. Pro spuštění celé aplikace spustíme příkaz `npm run dev`.

A.6 Ostatní ukázkové projekty

Ostatní ukázkové projekty v adresáři `examples` ukazují nasazení PTSQ v různých prostředích nebo frameworcích jako je Koa, Express, Fastify nebo Bun.

Všechny tyto projekty lze spustit stejnými příkazy jako projekt basic:

1. v příkazové řádce přejdeme do adresáře `examples/*`.
2. Pro instalaci závislostí ukázkového projektu spustíme příkaz `npm install`.
3. Spuštění serverové části vyžaduje příkaz `npm run start:server`.
4. v jiné příkazové řádce ve stejném adresáři spustíme klienta `npm run start:client`.
5. Pro spuštění introspekce lze využít příkaz `npm run generate:schema`.

Příloha B

Dodatečné výpisy kódu

```
type IsString<T> = T extends string ? true : false;
```

Výpis kódu 48 Podmínka na úrovni typů

```
type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

Výpis kódu 49 Cyklus na úrovni typů

```
type inferReturnType<T> = T extends (...args: any[]) =>
infer R ? R : never;
```

Výpis kódu 50 Odvození typů

```
ptsq({
  plugins: [
    useCORS({
      origin: ['http://localhost:3000', 'http://localhost:4000'],
      credentials: true,
      allowedHeaders: ['X-Custom-Header'],
      methods: ['POST'],
      maxAge: 300,
      exposedHeaders: ['Content-Type'],
    }),
  ]
});
```

Výpis kódu 51 Ukázka nastavení CORS

```
resolver.use(({ ctx, next }) => {
  console.log('before handler');

  return next();
});
```

Výpis kódu 52 Definování middleware, která je spuštěna před spuštěním handleru

```
resolver.use(async ({ ctx, next }) => {
  const response = await next();

  console.log('after handler');

  return response;
});
```

Výpis kódu 53 Definování middleware, která je spuštěna po spuštěním handleru

```
const { resolver, router, serve } = ptsq(...).use(async ({ next }) => {
  const response = await next();

  if(
    process.env.NODE_ENV === 'development' &&
    !response.ok
  ) {
    console.error(response.error);
  }

  return response;
}).create();
```

Výpis kódu 54 Vypisování chyb ve vývojovém prostředí

```
const { resolver, router, serve } = ptsq(...).use(async ({ next }) => {
  const response = await next();

  if(
    process.env.NODE_ENV === 'production' &&
    !response.ok
  ) throw new PtsqError({ ...response.error, cause: undefined });

  return response;
});
```

Výpis kódu 55 Maskování vlastnosti cause u chyby v produkčním prostředí

```
export const createContext = async (options: {
  req: NextApiRequest;
  res: NextApiResponse;
}) => {
  const session =
    await getServerSession(options.req, options.res, nextAuthOptions);

  return {
    ...options,
    session,
  };
};
```

Výpis kódu 56 Autentizace pomocí Next-auth

```
ptsq(...).use(async ({ meta, next, ctx }) => {
  const response = await next();

  ctx.logger.logAction({
    route: meta.route,
    payload: {
      meta,
      response
    }
  });

  return response;
});
```

Výpis kódu 57 Ukázka logování pomocí middleware serveru

```
export const App = () => {
  const userListInfiniteQuery =
    client.user.list.useInfiniteQuery(
      {
        filters: {...}
      },
      {
        getNextPageParam: (lastPage) => lastPage.nextCursor,
      },
    );

  return (
    <Page
      isLoading={userListInfiniteQuery.isFetching}
      isError={userListInfiniteQuery.error}
    >
      <p>Data: {JSON.stringify(infiniteQuery.data)}</p>
      <button onClick={() => infiniteQuery.fetchNextPage()}>
        Load next page
      </button>
    </Page>
  );
};
```

Výpis kódu 58 Stránkování pomocí React klienta

Bibliografie

1. BIEHL, Matthias. *API architecture*. Sv. 2. API-University Press, 2015.
2. W3C. *Web Services Glossary* [online]. 2004-02. [cit. 2024-04-04]. Dostupné z: https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice.
3. BERNERS-LEE, Tim; FIELDING, Roy T.; MASINTER, Larry M. *Uniform Resource Identifier (URI): Generic Syntax* [RFC 3986]. RFC Editor, 2005-01. Request for Comments, č. 3986. Dostupné z DOI: 10.17487/RFC3986.
4. NIELSEN, Henrik; MOGUL, Jeffrey; MASINTER, Larry M; FIELDING, Roy T.; GETTYS, Jim; LEACH, Paul J.; BERNERS-LEE, Tim. *Hypertext Transfer Protocol – HTTP/1.1* [RFC 2616]. RFC Editor, 1999-06. Request for Comments, č. 2616. Dostupné z DOI: 10.17487/RFC2616.
5. RESCORLA, Eric. *HTTP Over TLS* [RFC 2818]. RFC Editor, 2000-05. Request for Comments, č. 2818. Dostupné z DOI: 10.17487/RFC2818.
6. BARTH, Adam. *HTTP State Management Mechanism* [RFC 6265]. RFC Editor, 2011-04. Request for Comments, č. 6265. Dostupné z DOI: 10.17487/RFC6265.
7. PŘISPĚVATELÉ MDN. *Cross-Origin Resource Sharing (CORS) – HTTP | MDN* [online]. 2023-12. [cit. 2024-04-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.
8. WHATWG. *Fetch Standard* [online]. 2024-04. [cit. 2024-04-06]. Dostupné z: https://fetch.spec.whatwg.org/.
9. BAKKEN, David. Middleware. *Encyclopedia of Distributed Computing*. 2001, roč. 11.
10. PŘISPĚVATELÉ MDN. *Middleware – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-06. [cit. 2024-04-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Middleware.
11. PŘISPĚVATELÉ MDN. *JavaScript – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-12. [cit. 2024-04-07]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/JavaScript.
12. PŘISPĚVATELÉ MDN. *JavaScript language overview – JavaScript | MDN* [online]. 2023-11. [cit. 2024-05-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview.
13. PŘISPĚVATELÉ NODEJS. *Node.js* [online]. 2024-03. [cit. 2024-04-08]. Dostupné z: https://nodejs.org/en/about.
14. PŘISPĚVATELÉ MDN. *Polyfill – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-06. [cit. 2024-04-08]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.

15. GARRETT, Jesse James et al. Ajax: A new approach to web applications. 2005.
16. PŘISPĚVATELÉ MDN. *Ajax – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-11. [cit. 2024-04-07]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Glossary/AJAX`.
17. PŘISPĚVATELÉ MDN. *WHATWG – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-07. [cit. 2024-04-05]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Glossary/WHATWG`.
18. WHATWG. *WHATWG* [online]. 2024. [cit. 2024-05-02]. Dostupné z: `https://whatwg.org/faq`.
19. PŘISPĚVATELÉ MDN. *Promise – JavaScript | MDN* [online]. 2023-11. [cit. 2024-05-04]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.
20. PŘISPĚVATELÉ MDN. *Blob – Web APIs | MDN* [online]. 2023-12. [cit. 2024-05-02]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Blob`.
21. PŘISPĚVATELÉ MDN. *Using the Fetch API – Web APIs | MDN* [online]. 2023-08. [cit. 2024-04-04]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`.
22. KOISHYBAYEV, Igibek; KAPRAVELOS, Alexandros. Mininode: Reducing the attack surface of node. js applications. In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 2020, s. 121–134.
23. PŘISPĚVATELÉ MDN. *JavaScript modules – JavaScript | MDN* [online]. 2024-03. [cit. 2024-04-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`.
24. MICROSOFT. *TypeScript language specification – version 1.8* [online]. 2016-01. [cit. 2024-04-07]. Dostupné z: `https://javascript.xgqfrms.xyz/pdfs/TypeScript%20Language%20Specification.pdf`.
25. PŘISPĚVATELÉ MDN. *TypeScript – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2024-04. [cit. 2024-04-07]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Glossary/TypeScript`.
26. PROKOPEC, Ludvík. *Kiq/Dokumentace.pdf at main · lewis-wow/Kiq* [online]. 2023-01. [cit. 2024-04-08]. Dostupné z: `https://github.com/lewis-wow/Kiq/blob/main/Dokumentace.pdf`.
27. ZHANG, Leng. *REACT APPLICATION OPTIMIZATION*. Diss. California State Polytechnic University, Pomona, 2021.
28. META PLATFORMS, INC.; PŘISPĚVATELÉ REACT. *React* [online]. 2022-06. [cit. 2024-04-08]. Dostupné z: `https://react.dev/`.
29. HARRIS, Rich; PŘISPĚVATELÉ SVELTE. *Svelte • Cybernetically enhanced web apps* [online]. 2023-03. [cit. 2024-04-08]. Dostupné z: `https://svelte.dev/`.
30. FIELDING, Roy Thomas. *REST: Architectural Styles and the Design of Network-based Software Architectures*. 2000. Dostupné také z: `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. Doctoral dissertation. University of California, Irvine.
31. MASSE, Mark. *REST API design rulebook: designing consistent RESTful web service interfaces*. "O'Reilly Media, Inc.", 2011.
32. SUN MICROSYSTEMS, Inc. *RPC: Remote Procedure Call Protocol specification: Version 2* [RFC 1057]. RFC Editor, 1988-06. Request for Comments, č. 1057. Dostupné z DOI: `10.17487/RFC1057`.

33. BRAY, Tim. *The JavaScript Object Notation (JSON) Data Interchange Format* [RFC 8259]. RFC Editor, 2017-12. Request for Comments, č. 8259. Dostupné z DOI: 10.17487/RFC8259.
34. PHILIPPE, THOMY. *JSON Schema extension to NTV data*. Internet Engineering Task Force, 2024-02. Internet-Draft, draft-thomy-ntv-schema-00. Internet Engineering Task Force. Dostupné také z: https://datatracker.ietf.org/doc/draft-thomy-ntv-schema/00/. Work in Progress.
35. YERGEAU, François; PAOLI, Jean; SPERBERG-MCQUEEN, Michael; MALER, Eve; BRAY, Tim. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008-11. W3C Recommendation. W3C. https://www.w3.org/TR/2008/REC-xml-20081126/.
36. LLC, TanStack. *Overview | TanStack Query Docs* [online]. 2024-04. [cit. 2024-04-11]. Dostupné z: https://tanstack.com/query.
37. PŘISPĚVATELÉ NODEJS. *HTTP | Node.js v22.1.0 Documentation* [online]. 2024-04. [cit. 2024-05-05]. Dostupné z: https://nodejs.org/en/about.
38. TANRIKULU, Arda. *Home (FETS)* [online]. 2024-04. [cit. 2024-04-09]. Dostupné z: https://the-guild.dev/openapi/fets.
39. FACEBOOK, INC.; PŘISPĚVATELÉ GRAPHQL. *GraphQL* [online]. 2021-10. [cit. 2024-04-08]. Dostupné z: https://spec.graphql.org/October2021/.
40. FOUNDATION, The GraphQL. *Execution | GraphQL* [online]. 2024-03. [cit. 2024-04-04]. Dostupné z: https://graphql.org/learn/execution/.
41. INC., Apollo Graph. *Context and contextValue – Apollo GraphQL Docs* [online]. 2023-02. [cit. 2024-05-05]. Dostupné z: https://www.apollographql.com/docs/apollo-server/data/context/.
42. FOUNDATION, The GraphQL. *Introspection | GraphQL* [online]. 2024-03. [cit. 2024-05-16]. Dostupné z: https://graphql.org/graphql-js/.
43. GRIESSER, Tim; PŘISPĚVATELÉ NEXUS. *GraphQL Nexus · Declarative, Code-First GraphQL Schemas for JavaScript/TypeScript* [online]. 2023-03. [cit. 2024-04-04]. Dostupné z: https://nexusjs.org/.
44. HAYES, Michael; PŘISPĚVATELÉ POTHOS. *Pothos GraphQL Overview and documentation* [online]. 2024-03. [cit. 2024-04-04]. Dostupné z: https://pothos-graphql.dev/.
45. FOUNDATION, The GraphQL. *GraphQL Clients | GraphQL* [online]. 2024. [cit. 2024-04-04]. Dostupné z: https://graphql.org/graphql-js/graphql-clients/.
46. INC., Apollo Graph. *Apollo Link overview* [online]. 2024-01. [cit. 2024-04-04]. Dostupné z: https://www.apollographql.com/docs/react/api/link/introduction.
47. JOHANSSON, Alex; PŘISPĚVATELÉ TRPC. *tRPC – Move Fast and Break Nothing. End-to-end typesafe APIs made easy.* [Online]. 2024-04. [cit. 2024-04-11]. Dostupné z: https://trpc.io/.
48. PŘISPĚVATELÉ MDN. *Proxy – JavaScript | MDN* [online]. 2023-11. [cit. 2024-04-08]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.
49. PŘISPĚVATELÉ ZODIOS. *Zodios* [online]. 2023-10. [cit. 2024-04-04]. Dostupné z: https://www.zodios.org/.
50. BUTLER, Oliver; PŘISPĚVATELÉ TS-REST. *TS-REST* [online]. 2024-04. [cit. 2024-04-11]. Dostupné z: https://ts-rest.com/.
51. IBM CLOUD EDUCATION. *SDK vs. API: What's the Difference?* [Online]. 2021-07. [cit. 2024-05-09]. Dostupné z: https://admin01.prod.blogs.cis.ibm.net/blog/sdk-vs-api/.

52. ALEXANDER SHVETS. *Design Patterns* [online]. [cit. 2024-05-09]. Dostupné z: https://refactoring.guru/design-patterns.

53. HUNT, John. Gang of Four Design Patterns. In: *Scala Design Patterns: Patterns for Practical Reuse and Design*. Cham: Springer International Publishing, 2013, s. 135–136. ISBN 978-3-319-02192-8. Dostupné z DOI: 10.1007/978-3-319-02192-8_16.

54. OLUWATOSIN, Haroon Shakirat. Client-server model. *IOSR Journal of Computer Engineering*. 2014, roč. 16, č. 1, s. 67–71.

55. PŘISPĚVATELÉ MDN. *MVC – MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online]. 2023-12. [cit. 2024-05-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/MVC.

56. BRITO, Gleison; TERRA, Ricardo; VALENTE, Marco Tulio. Monorepos: a multivocal literature review. *arXiv preprint arXiv:1810.09477*. 2018, roč. 18.

57. VERCEL, Inc. *Turborepo* [online]. 2024-04. [cit. 2024-04-04]. Dostupné z: https://turbo.build/repo.

58. PŘISPĚVATELÉ NX. *Nx and Turborepo | Nx* [online]. 2024-04. [cit. 2024-05-08]. Dostupné z: https://nx.dev/concepts/more-concepts/turbo-and-nx.

59. PŘISPĚVATELÉ PNPM. *Fast, disk space efficient package manager | pnpm* [online]. 2024-02. [cit. 2024-04-04]. Dostupné z: https://pnpm.io/.

60. JONES, Michael B.; BRADLEY, John; SAKIMURA, Nat. *JSON Web Token (JWT)* [RFC 7519]. RFC Editor, 2015-05. Request for Comments, č. 7519. Dostupné z DOI: 10.17487/RFC7519.

61. SINCLAIRZX81. *sinclairzx81/typebox* [online]. 2024-03. [cit. 2024-04-04]. Dostupné z: https://github.com/sinclairzx81/typebox.

62. AMBLER, Scott W. Mapping objects to relational databases. *White Paper, Ronin International*. 2000.

63. HU, Vincent C; FERRAIOLO, David; KUHN, D Richard et al. *Assessment of access control systems*. US Department of Commerce, National Institute of Standards a Technology …, 2006.

64. SHAHIN, Mojtaba; BABAR, Muhammad Ali; ZHU, Liming. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*. 2017, roč. 5, s. 3909–3943.

65. YALÇIN, Nursel; KÖSE, Utku. What is search engine optimization: SEO? *Procedia-Social and Behavioral Sciences*. 2010, roč. 9, s. 487–493.

66. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [RFC 6749]. RFC Editor, 2012. Request for Comments, č. 6749. Dostupné z DOI: 10.17487/RFC6749.

Obsah příloh