

Gymnázium Brno, třída Kapitána Jaroše, p. o.

---

# Vývoj sociální sítě pomocí skupiny technologií T3 stack

Závěrečná práce

Vedoucí práce:  
prof. Mgr. Marek Blaha

Albert Pátík

Brno 2023



## **Poděkování**

Rád bych poděkoval Theu Browneovi, který souhrn webových technologií T3 vytvořil a všem dalším vývojářům, kteří se na projektu podíleli. Dále bych rád poděkoval svému spolužákovi Adamu Hrnčárkovi, který mě se souborem technologií T3 stack seznámil a svému panu profesorovi informatiky, Marku Blahovi, který se mnou práci konzultoval a pomáhal koordinovat její dokončení. Na závěr bych chtěl poděkovat svojí rodině a zejména svým rodičům, kteří mě během tvorby práce podporovali.



### Čestné prohlášení

Prohlašuji, že jsem práci **Vývoj sociální sítě pomocí skupiny technologií T3 stack** vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a v souladu s platnou Směrnicí o zveřejňování závěrečných prací.

V Brně dne 28. dubna 2023

.....  
podpis



**Abstract**

This theses investigates how to use many web technologies from T3 stack technologies bundle for creation of simple social media platform. This platform will run only as a web application and will not be deployed for public. The final application will however be full stack, meaning that both server and client side will be developed.

Brno, 2023.

**Key words:** social network, web development, Reactjs

**Abstrakt**

Tato závěrečná práce pojednává o vývoji velmi jednoduché sociální sítě s využitím webových technologií ze souboru webových technologií pro tvorbu robustních stránek – T3 stack. Tvořená aplikace bude běžet pouze jako webová stránka a nebude zveřejněna, každopádně se bude jednat o full stack aplikaci, která bude mít jak klientskou stránku, tak serverovou stránku, kde bude existovat databáze.

Brno, 2023.

**Klíčová slova:** sociální síť, vývoj webových stránek, Reactjs





# Obsah

<b>1</b>	<b>Úvod a cíl práce</b>	<b>11</b>
1.1	Úvod do problematiky . . . . .	11
1.2	Cíl práce . . . . .	11
<b>2</b>	<b>Úvod do T3</b>	<b>12</b>
2.1	create-t3-app . . . . .	12
2.2	Typescript . . . . .	12
2.3	Next.js . . . . .	12
2.4	Tailwind CSS . . . . .	13
2.5	tRPC . . . . .	13
2.6	Prisma . . . . .	13
2.7	NextAuth.js . . . . .	13
<b>3</b>	<b>Inicializace T3 projektu</b>	<b>14</b>
<b>4</b>	<b>Základní struktura T3 projektu</b>	<b>15</b>
4.1	Adresář .next . . . . .	15
4.2	Adresář node_modules . . . . .	16
4.3	Adresář prisma . . . . .	16
4.4	Soubor schema.prisma . . . . .	16
4.5	Adresář public . . . . .	16
4.6	Adresář src . . . . .	16
4.7	Adresář pages . . . . .	16
4.8	Adresář api v adresáři pages . . . . .	17
4.9	Adresář auth . . . . .	17
4.10	Adresář trpc a adresář routers . . . . .	17
4.11	Soubor __app.tsx . . . . .	17
4.12	Adresář server . . . . .	17
4.13	Adresář api v adresáři server . . . . .	17
4.14	Soubor trpc.ts . . . . .	17
4.15	Soubor auth.ts . . . . .	17
4.16	Soubor db.ts . . . . .	17
4.17	Adresář styles . . . . .	18
4.18	Soubor globals.css . . . . .	18
4.19	Adresář utils . . . . .	18
4.20	Soubor api.ts . . . . .	18
4.21	Soubor env.mjs . . . . .	18
4.22	Soubor .env . . . . .	18
4.23	Soubor package.json . . . . .	18
4.24	Soubor tailwind.config.cjs . . . . .	18
4.25	Soubor tsconfig.json . . . . .	18

---

<b>5</b>	<b>Renderovací techniky</b>	<b>19</b>
5.1	SSR a funkce <code>getServerSideProps</code> . . . . .	19
5.2	SSG a funkce <code>getStaticProps</code> . . . . .	20
5.3	SSG a funkce <code>getStaticPaths</code> . . . . .	20
5.4	ISR . . . . .	21
<b>6</b>	<b>Postup vývoje aplikace GatherlyHub</b>	<b>22</b>
6.1	Úprava projektu pro potřeby vyvíjené aplikace . . . . .	22
6.2	Nastavení databáze . . . . .	24
6.3	Tvorba schémata databáze . . . . .	25
6.4	Autentikace přes Google OAuth . . . . .	27
6.5	Implementace funkcionality . . . . .	29
<b>7</b>	<b>Závěr</b>	<b>37</b>
<b>8</b>	<b>Literatura</b>	<b>38</b>

# 1 Úvod a cíl práce

## 1.1 Úvod do problematiky

Vývoj webových aplikací není jednoduchým úkolem. Pokud aplikace vyžaduje autentikaci uživatele, práci s velkým množstvím dat, zobrazování pouze určité podmnožiny všech dat, dynamické filtrování dat, velmi přehledný UX nebo například responsivitu, je potřeba vyřešit otázky bezpečnosti komunikace přes internet, kompatibility a dalších. Navíc je potřeba psát aplikaci tak, aby bylo v budoucnu možné ji dále rozvíjet.

Existuje mnoho technologií a praktik, které řeší dílčí problematiky vývoje robustní webové aplikace, bohužel však jako weboví vývojáři se musíme vypořádat se všemi překážkami, které stojí v cestě. Proto tyto technologie kombinujeme. To ale vždy nemusí být jednoduchým, nebo dokonce proveditelným úkolem. Technologie spolu občas nespolupracují, nebo vyžadují jiné verze stejných externích modulů.

Proto vznikl T3 stack. Jendá se o souhrn technologií, které spolu dokáží spolupracovat a řeší většinu náročných problémů, které s vývojem webových technologií přichází.

## 1.2 Cíl práce

Cílem práce je posoudit schopnosti T3 stacku řešit problematiku vývoje webových aplikací. Pro posouzení bude sloužit testovní aplikace GatherlyHub, jednoduchá sociální síť, která byla pomocí T3 stacku v rámci práce vytvořena. Práce bude posuzovat hlavně:

- implementaci autentifikace uživatelů
- práci s daty
- robustnost výsledné aplikace
- schopnost vyvíjet full stack
- náročnost na hardwarové zdroje
- pohodlnost práce se souborem technologií T3 stack

Dále bude v práci obsažen stručný úvod do technologií T3 stacku a jejich společné používání v rámci celého projektu i stručné vysvětlení postupu vývoje webové aplikace GatherlyHub. Práce nezkoumá proces tvorby vizuálních rozhraní aplikace.

## 2 Úvod do T3

T3 stack, dále jen T3, je označení pro soubor individuálních technologií. Síla T3 stacku vychází z nástroje, který umožňuje tvorbu prázdného T3 projektu, který je nakonfigurovaný tak, aby spolu dílčí technologie fungovaly. Tento nástroj se jmenuje `create-t3-app`, má command line interface a je udržovaný dedikovaným týmem lidí pod vedením Thea Browna. Mezi nejzákladnější technologie, na kterých T3 stojí, patří:

- Typescript
- Next.js
- Tailwind CSS
- tRPC
- Prisma
- NextAuth.js

### 2.1 `create-t3-app`

`create-t3-app` je command line interface nástroj pro vytvoření prázdného projektu webové aplikace. Po spuštění je uživatel vyzván k vybrání technologií, které chce v aplikaci používat. Typescript a Next.js jsou povinné.

### 2.2 Typescript

Typescript je nadstavba javascriptu, která přináší typovou bezpečnost při psaní kódu. Javascript samotný po uživateli žádné typy nevyžaduje a předpokládá, že vývojář ví, co dělá a odvozuje typy z kódu programu. Typescript typy striktně hlídá a nutí vývojáře, aby vše předem definoval a následně dodržoval. Pořád je ale možné Typescript „obejít“ a pracovat s ním, jako by se jednalo o javascript.

### 2.3 Next.js

Next.js je Reactový framework. Narozdíl od populárního CLI `create-react-app`, který vytváří pouze frontendovou aplikaci, generuje CLI `create-next-app` template robustní, webové, full-stack aplikace. Obsahuje vlastní node server a zjednodušuje implementaci často požadovaných procesů, jako například server side rendering, incremental static regeneration nebo routing.

## React

React je javascriptový framewrok pro snadnější tvorbu webových aplikací. Umožňuje tvorbu .jsx komponentů, což usnadňuje tvorbu robustnějších webových aplikací zejména z hlediska UI a UX.

## 2.4 Tailwind CSS

Tailwind CSS je knihovna stylů, která hledá tailwindem nadefinované třídy (class) na všech html elementech v adresáři a aplikuje na ně příslušné styly. Zjednodušuje proces stylování aplikace tím, že zpřehledňuje názvy jednotlivých css stylů, přenáší informace o tom, kde se nachází přímo na konkrétní elementy a definuje vlastní knihovnu stylů. Tyto styly byly vybrány a otestovány tak, aby spolu ladily.

## 2.5 tRPC

tRPC zajišťuje plnou typovou bezpečnost mezi klientskou a serverovou stránkou aplikace. Místo psaní vlastních http requestů na klientovi a jejich vyhodnocování na serveru používá tRPC knihovnu React Query, díky které se komunikace mezi serverem a klientem zajišťuje pomocí volání funkcí, které řeší posílání a vyhodnocování http requestů samy.

## 2.6 Prisma

Prisma umožňuje implementaci ORM (Object Relational Mapping). Dokáže přečíst strukturu databáze a vygenerovat typy, se kterými můžeme při vývoji pracovat. Dále umožňuje definování struktury databáze a interakci s databází.

## 2.7 NextAuth.js

NextAuth.js je knihovna, která výrazně zjednodušuje implementaci autentifikace v projektech Next.js. Má sadu předem nadefinovaných adaptérů, které umožňují komunikaci s databází. Dále má sadu předem nadefinovaných NextAuth.js Providerů, které po troše konfigurace umožňují autentikaci uživatele přes populární OAuth providery, jako například Google, Discord nebo GitHub. Navíc je možné definovat vlastní NextAuth.js Providery, pro autentifikace s jinými OAuth providery, než které NextAuth.js podporuje defaultně. NextAuth.js podporuje i autentifikace přes e-mail či uživatelské jméno a heslo.

## 3 Inicializace T3 projektu

Po spuštění `create-t3-app` jsme vyzváni k výběru technologií, které chceme používat:

- `tailwind`
- `prisma`
- `tRPC`
- `NextAuth`

Při vývoji jednoduchého modelu sociální sítě budeme benefitovat ze všech. Po vybrání dokončíme inicializaci projektu, zejména ozačíme, že chceme stáhnout všechny potřebné moduly a `create-react-app` poté vytvoří základ naší aplikace.

## 4 Základní struktura T3 projektu

Podívejme se na důležité adresáře a soubory vytvořené aplikace. Tato struktura je platná pouze při zvolení všech technologií při inicializaci projektu.

```
adresář projektu
├── .next
├── node_modules
├── prisma
│   └── schema.prisma
├── public
├── src
│   ├── pages
│   │   ├── api
│   │   │   ├── auth
│   │   │   └── trpc
│   │   └── _app.tsx
│   ├── server
│   │   ├── api
│   │   │   ├── routers
│   │   │   ├── root.ts
│   │   │   └── trpc.ts
│   │   ├── auth.ts
│   │   └── db.ts
│   ├── styles
│   │   └── globals.css
│   ├── utils
│   │   └── api.ts
│   └── env.mjs
├── .env
├── package.json
├── tailwind.config.cjs
└── tsconfig.json
```

Kromě těchto souborů a adresářů je v projektu ještě mnoho dalších. Většina z nich jsou konfigurační soubory pro různé moduly nebo vývojové nástroje, jako například postcss, prettier nebo eslint. Dále soubory .gitignore a README.md pocházní z inicializace git repozitáře, který je v projektu automaticky vytvořen. V projektu lze nalézt i ukázkové soubory.

### 4.1 Adresář .next

Tento adresář obsahuje build projektu.

## 4.2 Adresář `node_modules`

Do adresáře `node_modules` se ukládají všechny stažené moduly, které jsou potřeba k běhu aplikace. Zpravidla to bývá největší adresář v projektu a je nastavený na ignoraci gitem v souboru `.gitignore`.

## 4.3 Adresář `prisma`

V adresáři `prisma` je uložený soubor `schema.prisma`. Mimo jiné se do něj ukládají `sqlite` databázové soubory, pokud je aplikace používá a další soubory, které `prisma` použije pro práci s databází (například migrační soubory).

## 4.4 Soubor `schema.prisma`

Soubor `schema.prisma` definuje, jaká bude struktura databáze. Definujeme jednotlivé modely a vztahy mezi nimi. Zároveň se zde definují některá nastavení databáze, jako například jaký typ databáze používáme, nebo jak zacházet s relacemi v databázi.

## 4.5 Adresář `public`

Soubory v adresáři `public` jsou viditelné z celé aplikace bez nutnosti specifikovat plnou nebo relativní cestu. Defaultně zde najdeme soubor `favicon.ico`, který slouží jako logo aplikace.

## 4.6 Adresář `src`

Adresář `src` je pomyslný „root“ naší aplikace. Obsahuje jednotlivé stránky, API endpointy, `tRPC` routery, soubory zajišťující autentikaci a mnoho dalších. Když vytváříme nové soubory a adresáře, které zajišťují funkčnost aplikace nebo její vzhled, budeme je vytvářet v `src` adresáři.

## 4.7 Adresář `pages`

Adresář `pages` slouží jako základ `file base routing` systému aplikace a pochází z `Nextjs`. Každý soubor v něm představuje nezávislou stránku, jejíž adresa bude totožná s názvem souboru. Jednotlivé složky, které v adresáři vytvoříme budou přidávat jejich názvy oddělené znakem „/“ do cesty k cílovým stránkám. Můžeme vytvořit stránku, jejíž obsah a název bude vyhodnocen za běhu aplikace. Když název souboru nebo adresáře zabalíme do hranatých závorek, bude obsah závorek dostupný jako proměnná na stránce jako takové. To bychom mohli použít například v kombinaci s tvorbou statických stránek, jejichž počet se odvíjí od dat z databáze.



## 4.8 Adresář api v adresáři pages

Tento adresář obsahuje endpointy naší aplikace. Zde můžeme zpracovávat http požadavky. Automaticky obsahuje adresáře auth a trpc.

## 4.9 Adresář auth

Tento adresář obsahuje soubor [...nextauth].ts, který zprostředkovává komunikaci mezi Nextjs api a NextAuth požadavky, které voláme z klienta.

## 4.10 Adresář trpc a adresář routers

Adresář trpc obsahuje soubor [trpc].ts. Na klientovi běží react-query, který posílá requesty na generický endpoint, který reprezentuje právě tento soubor. V adresáři routers definujeme jednotlivé routery, které obsahují funkce, jež chceme na serveru podporovat. Soubor [trpc].ts bere požadavky z react-query a volá nadefinované funkce. Sám se přitom chová jako Nextjs endpoint.

## 4.11 Soubor \_\_app.tsx

Tento soubor je základ klienta aplikace. Bere stránku a její parametry jako argumenty a vrací je obalené v kontextu, který má informace o aktuálním session.

## 4.12 Adresář server

V adresáři server definujeme funkce přístupné na serveru naší aplikace a také konfigurujeme jednotlivé technologie (viz soubory trpc.ts, auth.ts a db.ts).

## 4.13 Adresář api v adresáři server

Zde existují jednotlivé routery v adresáři routers. Navíc obsahuje soubory root.ts a trpc.ts.

## 4.14 Soubor trpc.ts

Soubor trpc.ts obsahuje konfiguraci a tvorbu některých objektů z tRPC.

## 4.15 Soubor auth.ts

Soubor auth.ts obsahuje konfiguraci a tvorbu některých objektů z NextAuth.

## 4.16 Soubor db.ts

Soubor db.ts obsahuje částečnou konfiguraci Prisma.

## 4.17 Adresář styles

Adresář styles obsahuje styly naší aplikace. Zejména je zde důležitý soubor `globals.css`.

## 4.18 Soubor `globals.css`

Soubor `globals.css` obsahuje styly, které se aplikují v souboru `__app.tsx`. Automaticky zde `tailwind` registruje svoje styly, které jsou pak dostupné ve všech komponentách, které ze `globals.css` čerpají.

## 4.19 Adresář `utils`

V tomto adresáři se vytváří objekty externích modulů, které mají být přístupné celé aplikaci. Samotná složka nemá žádné modifikátory viditelnosti, jde jenom o organizační záležitost. Defaultně obsahuje soubor `api-ts`.

## 4.20 Soubor `api.ts`

Soubor `api.ts` exportuje nakonfigurovaný objekt, na kterém jsou přístupné funkce, které lze na serveru volat.

## 4.21 Soubor `env.mjs`

Soubor `env.mjs` zajišťuje typování enviromentálních proměnných.

## 4.22 Soubor `.env`

Soubor `.env` slouží k uložení enviromentálních proměnných.

## 4.23 Soubor `package.json`

Soubor `package.json` obsahuje seznam modulů a jejich verzí, které je potřeba před kompilací webové aplikace stáhnout.

## 4.24 Soubor `tailwind.config.cjs`

Soubor `tailwind.config.cjs` obsahuje konfiguraci modulu `tailwindcss`.

## 4.25 Soubor `tsconfig.json`

Soubor `tsconfig.json` obsahuje konfiguraci `typescriptu`.

## 5 Renderovací techniky

Je mnoho způsobů, jak dostat naši aplikaci ke klientovi. Všechny mají svoje výhody i nevýhody, zejména v náročnosti implementace. Naštěstí Nextjs a tedy i T3 stack několik takových způsobů přímo podporuje a snižuje náročnost jejich implementace na minimum. Nextjs používá techniku SSR – server side rendering. Prvně server vygeneruje html, které pošle klientovi a ten ho zobrazí. Aby se stránka stala interaktivní, stáhne si klient ze serveru javascript a provede hydrataci – doplnění staženého javascriptu do staženého html.

### 5.1 SSR a funkce `getServerSideProps`

Pokud exportujeme funkci `getServerSideProps`, každá komponenta ve stejném souboru, která čeká na inicializační parametry, dostane tyto parametry právě od této funkce. Funkce `getServerSideProps` a její obsah se vyhodnotí na serveru, zároveň s tvorbou html, které se později pošle klientovi. Až klient html obdrží, dojde k hydrataci javascriptem. Právě proto bychom mohli funkci `getServerSideProps` použít. Kdybychom například na stránce zobrazovali hodnotu `Date.now()`, DOM, který se vyhodnotí na serveru, bude jiný od toho, který se vyhodnotí na klientovi, což znemožní hydrataci a stránka spadne. Mohli bychom proto získat hodnotu `Date.now()` na serveru a předat ji jako parametr komponentě. Další využití se týkají například otázky bezpečnosti. Cokoliv se děje na serveru, je pod naší kontrolou.

```
import { NextPage, GetServerSideProps } from "next";

type MyPageProps = {
  time: number;
};

const myPage: NextPage<MyPageProps> = ({ time }) => {
  return <p>{time}</p>;
};

export default myPage;

export async function getServerSideProps {
  const time = Date.now();

  return {
    props: {
      time: time,
    },
  };
};
```

Obr. 1: Ukázka použití funkce `getServerSideProps`

## 5.2 SSG a funkce `getStaticProps`

Exportováním funkce `getStaticProps` říkáme, že chceme vygenerovat stránku staticky. To má mnoho výhod, zejména rychlost, se kterou můžeme stránku na klienta dostat a lepší SEO. Bohužel však je tato metoda nevhodná, pokud se data na stránce často mění, protože je potřeba celý projek znovu sestavit.

```
import { NextPage, GetServerSideProps } from "next";

type MyPageProps = {
  time: number;
};

const myPage: NextPage<MyPageProps> = ({ time }) => {
  return <p>{time}</p>;
};

export default myPage;

export async function getStaticProps(context) {
  const time = Date.now();
  return {
    props: {
      time: time,
    },
  };
}
```

Obr. 2: Ukázka použití funkce `getStaticProps`

## 5.3 SSG a funkce `getStaticPaths`

Co když bychom chtěli vytvořit statické stránky z dynamické adresy? Pak nelze zavolat pouze funkci `getServerSideProps`, protože obsah stránky bude záviset na datech, která se určí až za běhu aplikace. Proto exportováním funkce `getStaticPaths` můžeme určit všechny dynamické hodnoty, pro které chceme stránku staticky vygenerovat. Když si o ni potom klient zažádá, dostane právě tu, která byla vytvořena se stejnými parametry. Opět je ale potřeba znovu sestavit aplikaci pokaždé, když by se měly jednotlivé cesty nějak měnit.

```
import { NextPage, GetServerSideProps } from "next";

type MyPageProps = {
  val: number;
};

const myPage: NextPage<MyPageProps> = ({ val }) => {
  return <p>{val}</p>;
};

export default myPage;

export async function getStaticProps() {
  const myprops... // some props obtaining logic
  return { props: {val: myprops} };
}

export async function getStaticPaths() {
  return {
    paths: [{ params: { id: "sejfb864sg86dgs" } }, { params: { id: "sdfiosgbf654sdg" } }],
    fallback: false,
  };
}
```

Obr. 3: Ukázka použití funkce getStaticPaths

## 5.4 ISR

Incremental static regeneration (ISR) je metoda, se kterou lze používat výhody generování statických stránek i v aplikacích, kde se často mění data. Koncept je založený na zneplatnění některých staticky vygenerovaných stránek, které se pak na serveru přegenerují znova. Pro stránku můžeme určit podmínku, jejíž naplnění způsobí, že se daná staticky vygenerovaná stránka vygeneruje znova. Nejjednodušší podmínkou, kterou takhle můžeme určit, je čas (můžeme například říci, že se stránka má přegenerovat každou hodinu). Implementace v T3 stacku (a Nextjs jako takovém, odkud ISR pochází) je velmi jednoduchá. Stačí ve funkci getStaticProps vrátit kromě objektu props i hodnotu revalidate.

```
export async function getStaticProps() {
  const myprops... // some props obtaining logic
  return {
    props: {val: myprops},
    revalidate: 60 // will revalidate every minute
  };
}
```

Obr. 4: Ukázka ISR

## 6 Postup vývoje aplikace GatherlyHub

V této kapitole se podíváme na to, jak se z nového T3 projektu dostat k funkční aplikaci. Začneme úpravou projektu, kdy smažeme všechny nepotřebné části a upravíme některé soubory tak, aby fungovaly pro potřeby naší aplikace. Dále nastavíme databázi a vytvoříme schéma modelů, se kterými budeme chtít pracovat. Poté implementujeme autentifikaci přes Google OAuth. Nakonec se podíváme na způsob, jakým ukládat a načítat data z databáze a jak zabezpečit stránky před nepřihlášenými uživateli.

### 6.1 Úprava projektu pro potřeby vyvíjené aplikace

Vygenerovaný projekt v sobě má spoustu ukázkového kódu, který pro naši aplikaci není přínosný. Je proto potřeba tento kód přepsat nebo úplně vymazat. Provedeme proto následující úpravy:

- odstraníme soubor `.env.example`, který slouží jako ukázka toho, jak by soubor `.env` mohl vypadat a přepíšeme proměnné, které v něm byly do souboru `.env`.
- odstraníme soubor `example.ts`, který slouží jako ukázkový router.
- v adresáři `root.ts` odstraníme `import` a registraci `example` routeru.
- v naší aplikaci budeme používat OAuth od Google jako způsob autentifikace. Proto v souboru `auth.ts` přepíšeme `import DiscordProvider` na `GoogleProvider` a i když by to pro funkčnost aplikace nebylo potřeba, změníme název proměnných prostředí, které importujeme `DISCORD_CLIENT_ID` a `DISCORD_CLIENT_SECRET` na `GOOGLE_CLIENT_ID` a `GOOGLE_CLIENT_SECRET`.
- změníme název a hodnoty proměnných prostředí dle předchozího kroku i v souborech `.env` a `env.mjs`.
- odstraníme `favicon.ico` z `public` adresáře a nahrajeme tam vlastní logo aplikace.
- vygenerujeme si hodnotu proměnné prostředí `NEXTAUTH_SECRET` pomocí příkazu „`openssl rand -base64 32`“ a uložíme ho do souboru `.env`.
- odstraníme model `Example` ze souboru `schema.prisma`.

#### Element Head

Nextjs exportuje element `Head`, který umí specifikovat chování stránky, které se normálně definuje v hlavičce (`<head></head>`) html souboru. Stačí element použít na libovolné stránce a jsme schopni přepsat například ikonku nebo název webové záložky v prohlížeči.

```
import { Head } from "next/document";

const myPage: NextPage = () => {
  return (
    <div>
      <Head>
        <title>My Page</title>
        <link rel="icon" href="/Logo.png" />
      </Head>
      <p>My Page</p>
    </div>
  );
};
```

Obr. 5: Použití elementu Head

V našem případě element Head použijeme na každé stránce, abychom specifikovali její název.

### Soubor `__document.tsx`

V naší aplikaci jsme nahradili soubor `favicon.ico` obrázkem našeho loga. Abychom zajistili, že se na všech stránkách použije právě toto logo, vytvoříme soubor `__document.tsx` v adresáři `pages`, kde rozšíříme defaultní `Document` objekt z `Nextjs`. Na rozdíl od ostatních souborů v adresáři `pages` nebude dostupný jako stránka. Většinu souboru zkopírujeme z dokumentace `Nextjs`. Navíc přidáme funkci `render`, která bude nastavovat logo na každé instanci `Document` objektu. Použijeme k tomu element `Head`.

```
import type { DocumentContext, DocumentInitialProps } from "next/document";
import Document, { Html, Head, Main, NextScript } from "next/document";

class MyDocument extends Document {
  static async getInitialProps(
    ctx: DocumentContext
  ): Promise<DocumentInitialProps> {
    const initialProps = await Document.getInitialProps(ctx);

    return initialProps;
  }

  render() {
    return (
      <Html>
        <Head>
          <link rel="icon" href="/Logo.png" />
        </Head>
        <body>
          <Main />
          <NextScript />
        </body>
      </Html>
    );
  }
}

export default MyDocument;
```

Obr. 6: Soubor \_\_document.tsx

## 6.2 Nastavení databáze

Aplikace GatherlyHub bude používat lokální sqlite datatabázi. Vše, co pro to musíme udělat, je změnit hodnotu proměnné provider v souboru schema.prisma z mysql na sqlite a upravit hodnotu enviromentální proměnné DATABASE\_URL v souboru .env na „file:./dev.db“.

```
datasource db {
  provider = "sqlite"
  url = env("DATABASE_URL")
}
```

Obr. 7: Změna v souboru schema.prisma



```
DATABASE_URL='file:./dev.db'  
NEXTAUTH_SECRET="fuAgf16h3jYxdHvvE5L75PB91GT6J286"  
NEXTAUTH_URL="http://localhost:3000"  
GOOGLE_CLIENT_ID="..."  
GOOGLE_CLIENT_SECRET="..."
```

Obr. 8: Změna v souboru .env

Aplikace bude nyní ukládat data do souboru dev.db, který najdeme v adresáři prisma.

## 6.3 Tvorba schémata databáze

Nyní máme databázi, ve které jsou modely, které NextAuthjs potřebuje pro správu autentikace. Naše aplikace bude ale vyžadovat úschovu dalších dat, zejména:

- uživatelů
- příspěvků od uživatelů
- jaký příspěvek se jakému uživateli líbí
- jaký uživatel jakého uživatele sleduje

Model uživatele už vytvořil NextAuthjs. Stačí tedy do něj dopsat hodnoty, o kterých chceme mít přehled. To budou:

- posts – List modelů Post, které uživateli náleží.
- likes – List modelů Like, které uživatel vytvořil.
- followers – List modelů Follow, kde je hodnota „follow\_to“ nastavena na id uživatele.
- following – List modelů Follow, kde je hodnota „follow\_from“ nastavena na id uživatele.

```
model User {  
  id          String    @id @default(cuid())  
  name        String?  
  email       String?   @unique  
  emailVerified DateTime?  
  image       String?  
  accounts    Account[]  
  sessions    Session[]  
  posts       Post[]  
  likes       Like[]  
  followers   Follow[]  @relation(name: "follow_to")  
  following   Follow[]  @relation(name: "follow_from")  
}
```

Obr. 9: Model User

Model příspěvku musíme vytvořit od nuly. Musí obsahovat unikátní id, musí uchovávat text, který je na něm napsaný a musí uchovávat bod v čase, kdy byl vytvořen. Normálně bychom pravděpodobně využili i údaj o tom, kdy byl příspěvek naposledy upraven. Proto tuto hodnotu přidáme i když s ní v rámci této práce pracovat nebudeme. Dále musí příspěvek vědět, komu patří. K tomu potřebujeme dvě hodnoty. Id uživatele (ownerId), kterému patří a relaci k modelu User, která páruje ownerId na id. Zároveň v relaci nastavíme, že pokud je uživatel smazaný, má se smazat i každý post, který mu patří. Poslední hodnotou bude list modelů Like, který zaznamenává, které Likey příspěvku náleží.

```
model Post {  
  id          String    @id @default(cuid())  
  text        String  
  createdAt   DateTime   @default(now())  
  updatedAt   DateTime   @updatedAt  
  owner        User      @relation(fields: [ownerId], references: [id], onDelete: Cascade)  
  ownerId     String  
  likes       Like[]  
}
```

Obr. 10: Model Post

Model Like je o trochu náročnější. Kromě unikátního id, relace s uživatelem a relace s příspěvkem musíme totiž zajistit, aby jeden uživatel nemohl dát like stejnému příspěvku dvakrát. To uděláme přes klíčový argument „@@unique()“, kterému předáme seznam hodnot, které musí být unikátní. V našem případě to budou hodnoty ownerId a postId.

```
model Like {  
  id          String    @id @default(cuid())  
  ownerId     String  
  owner        User      @relation(fields: [ownerId], references: [id])  
  postId      String  
  post        Post       @relation(fields: [postId], references: [id], onDelete: Cascade)  
  @@unique([ownerId, postId])  
}
```

Obr. 11: Model Like

Model Follow přináší další drobnou komplikaci. Nese dvě relace s modelem uživatele. Aby se od sebe daly rozlišit, přidělíme každé relaci jiné jméno. Když ale potom budeme na modelu uživatele tvořit relaci s modelem Follow, budeme muset jméno specifikovat. Opět je žádané, aby jeden uživatel nemohl sledovat jiného dvakrát, a aby se v případě odstranění uživatele smazaly všechny modely Follow, které vytvořil.

```
model Follow {  
  id      String      @id @default(cuid())  
  fromId   String  
  from     User        @relation(name: "follow_from", fields: [fromId],  
    references: [id], onDelete: Cascade)  
  toId     String  
  to       User        @relation(name: "follow_to" , fields: [toId], references:  
    [id], onDelete: Cascade)  
  @@unique([fromId, toId])  
}
```

Obr. 12: Model Follow

Nyní když jsou všechny modely, se kterými budeme chtít pracovat, nadefinované, stačí zavolat příkaz „npx prisma db push“, který upraví schéma databáze a vygeneruje nové typy pro typescript.

## 6.4 Autentikace přes Google OAuth

Aby autentifikace pomocí Google OAuth provideru mohla fungovat, potřebujeme od Googlu získat client-id a client-secret klíče. Ty potom uložíme do proměnných prostředí `GOOGLE_CLIENT_ID` a `GOOGLE_CLIENT_SECRET`. To samo o sobě zprovozní možnost přihlášení se přes Google OAuth provider. NextAuth vytváří automaticky stránku, kde se můžeme zkusit autentifikovat všemi způsoby, které jsme v naší aplikaci nastavili. My si na to ale raději vytvoříme naši vlastní stránku. Bude sídlit v adresáři pages s názvem signin.

NextAuth exportuje funkci, která se jmenuje `signIn`. Pokud ji zavoláme bez argumentů, přesune nás na stránku, kterou má NextAuth registrovanou jako sloužící k autentifikaci. Přidáme proto do objektu `authOptions` v souboru `auth.ts` objekt `pages`, který bude přepisovat, kde jsou jednotlivé autentifikační stránky v naší aplikaci.

```
export const authOptions: NextAuthOptions = {
  callbacks: {
    session({ session, user }) {
      if (session.user) {
        session.user.id = user.id;
      }
      return session;
    },
  },
  adapter: PrismaAdapter(prisma),
  providers: [
    GoogleProvider({
      clientId: env.GOOGLE_CLIENT_ID,
      clientSecret: env.GOOGLE_CLIENT_SECRET,
    }),
  ],
  pages: {
    signIn: "/signin",
  },
};
```

Obr. 13: Úprava defaultní signIn adresy

Nyní stačí zajistit, že na stránce `signin.tsx` bude možnost přihlásit se. Cokoliv, co se děje uvnitř funkce `getServerSideProps`, bude vyhodnoceno ještě na serveru. Funkce může vrátit objekt `props`, který se předá komponentě, která takové parametry bere. Tato funkcionality pochází z Nextjs. V našem případě takhle získáme seznam všech providerů, který předáme stránce jako parametr. Funkce `getProviders`, kterou k načtení dostupných providerů používáme ale vrací 4objekt záznamů. Jednoduše ho tedy převedeme na list pomocí funkce `values` na třídě `Object`. Pak můžeme celý seznam projet a pro každého providera vykreslit tlačítko, které po stisknutí volá funkci `signIn` a předává jí id providera a url adresu, na kterou se vrátit po skončení OAuth autentikace.

```
import { getProviders, signIn } from "next-auth/react";

const signin: FC<signinProps> = ({ providers }) => (
  {Object.values(providers).map((provider) => (
    <button
      key={provider.id}
      onClick={() => {
        signIn(provider.id, { callbackUrl: "/" });
      }}
    >
    {`Sign in with ${provider.name}`}
  </button>
  )})
);

export default signin;

export const getServerSideProps = async () => {
  const providers = await getProviders();
  return {
    props: { providers: providers },
  };
};
```

Obr. 14: Zjednodušený model stránky signin, který ukazuje pouze logiku přihlašování.

## 6.5 Implementace funkcionality

Veškerá funkcionalita aplikace se točí okolo efektivního získávání a zapisování dat z databáze. Tato kapitola se tedy věnuje:

- middlewaru a bezpečnému přístupu k databázi
- tvorbě tRPC routerů
- práce s databází

Jak již bylo několikrát uvedeno, práce se nezabývá tvorbou konkrétních komponent a stránek. Bylo by ale vhodné zmínit, že každá stránka v naší aplikaci vyjma stránky `signIn` brání přístupu neautentikovaných uživatelů. V projektu T3 by to nebylo potřeba. Jediný způsob, jak by se někdo mohl dostat na stránku, kam nemá mít přístup by byl, že by manuálně napsal url adresu, což většina uživatelů nedělá. I když by se ale na stránku dostal, nemohl by nijak interagovat s databází (viz sekce 7.5, middleware a bezpečný přístup k databázi) a tudíž by tím poškodil pouze svůj „zážitek“ z používání aplikace. I přesto ale implementujeme funkcionalitu, která zamezí uživatelům dostat se na stránky, kam nemají mít přístup, pokud nejsou přihlášení.

Toho docílíme pomocí funkce `getServerSideProps`. Ještě na serveru zjistíme, jestli existuje objekt `sezení` (to znamená, že je uživatel přihlášený) a pokud ne, pošleme ho na jinou adresu. Trik je v tom, že se vyhodnocení a přesměrování odehraje na serveru, uživatel tedy nemá možnost mu zabránit přepsáním javascriptu, který

obdrží na klientovi a nevidí ani divné problíknutí před přesměrováním.

```
import type { GetServerSidePropsContext, NextPage } from "next";
import { getServerAuthSession } from "../server/auth";

const Home: NextPage = () => {
  return <main>...</main>;
};

export default Home;

export async function getServerSideProps(context: GetServerSidePropsContext) {
  const session = await getServerAuthSession(context);

  if (!session) {
    return {
      redirect: {
        destination: "/signin",
        permanent: false,
      },
    };
  }

  return {
    props: {},
  };
}
```

Obr. 15: Přesměrování v případě nepřihlášeného uživatele

Nadefinujeme tedy jednoduchou proceduru „post“, která nahraje příspěvek do databáze. Prvně vytvoříme router jako takový. Ten bude takřka totožný s root routerem, až na to, že nebudeme exportovat typ tohoto routeru.

Pak vytvoříme zod schéma, které bude určovat, jaké hodnoty k tvorbě příspěvku potřebujeme. Náš příspěvek potřebuje pouze text, který je jeho obsahem. Zbytek dat odvodíme na serveru, přímo v proceduře. Tento text bude dlouhý minimálně jeden znak a maximálně 1000 znaků.

```
export const postPostInput = z.object({
  post_text: z.string().min(1).max(1000),
});
```

Obr. 17: Zod schéma pro input procedury post v aplikaci GatherlyHub

Přidejme proceduru post na routeru postRouter se vstupními hodnotami určenými zod schématem, které jsme proto vytvořili. Zavoláme postupně metody input a pak mutation. Metoda input bere zod schéma jako parametr. Metoda mutation, která značí, že funkcí plánujeme data v databázi měnit, bere jako parametr námi definovanou funkci.

```
export const postRouter = createTRPCRouter({
  post: protectedProcedure
    .input(postPostInput)
    .mutation(({ ctx, input }) => {}),
});
```

Obr. 18: Procedura post na routeru postRouter v aplikaci GatherlyHub

Taková procedura zatím ale nic nedělá. Pokud chceme v databázi vytvořit novou instanci modelu Post, musíme na objektu prisma, který získáme z objektu ctx, zavolat metodu create a předat jí vše, co potřebuje k tvorbě objektu. Objekt ctx a input procedura naší funkci předá automaticky. Metoda create bere jako parametr objekt, ve kterém je klíč „data“. Tam se poté snaží najít všechny potřebné parametry. Kromě objektu data můžeme předat i jiné objekty, které specifikují, co má metoda dělat (například objekt „where“ filtruje, kde se má metoda uplatnit). To ale v případě metody create nemá smysl.

Abychom vytvořili relaci mezi příspěvkem a uživatelem, který ho vytvořil, vytáhneme si z objektu session id uživatele a uložíme ji do proměnné userId. Potom místo pevného nastavení hodnoty owner předáme objekt, ve kterém bude vnořený objekt s názvem „connect“. Ten na sobě bude mít property id s hodnotou userId. Tím prisma spáruje příspěvek právě s tím uživatelem, který má id rovné userId. Vzhledem k tomu, že když příspěvek vytváříme, nemá ještě žádné lajky, můžeme tuto hodnotu ignorovat a seznam relací se tak vytvoří prázdný. Ostatní hodnoty na objektu se vytvoří automaticky. Všimněme si, že metodu create nevoláme přímo na objektu prisma, ale na modelu, který chceme vytvořit. Ten je přístupný na objektu prisma.

```
export const postRouter = createTRPCRouter({
  post: protectedProcedure.input(postPostInput).mutation(({ ctx, input }) => {
    const { post_text } = input;
    const { prisma, session } = ctx;

    const userId = session.user.id;

    return prisma.post.create({
      data: {
        text: post_text,
        owner: {
          connect: {
            id: userId,
          },
        },
      },
    });
  }),
});
```

Obr. 19: Procedura post na routeru postRouter v aplikaci GatherlyHub

Nyní stačí proceduru na klientovy zavolat. Naimportujeme si objekt `api` z adresáře `utils/api`. Na něm můžeme volat všechny procedury na všech routerech, které jsme k hlavnímu routeru `appRouter` připojili. Naše procedura je na routeru, který jsme pojmenovali `post` a jmenuje se `post`. Zároveň se jedná o mutaci (`mutation`). Budeme tedy volat metodu `useMutation`, která může nastavit nějaké dodatečné okolnosti volání procedury (například co se má stát, když se procedura úspěšně dokončí). Teď jen stačí zavolat metodu `mutate`, která bere zodem specifikované parametry jako vstup.

```
const { mutate: post } = api.post.post.useMutation();

post({
  post_text: postTextInput,
});
```

Obr. 20: Volání procedury `post` v aplikaci `GatherlyHub`

Nyní vytvořme proceduru `listPosts`, která nám vrátí všechny příspěvky v databázi. Podobně jako u procedury `post` se jedná o akci, kterou nemůže uživatel provést pokud není přihlášený. Bude se tedy jednat o `protectedProcedure`. Opět vytvoříme zod schéma vstupních parametrů pro funkci `input`. Toto schéma ale zatím necháme prázdné, použijeme ho až později. Narozdíl od procedury `post` se však jedná o proceduru, která má data z databáze pouze získávat. Místo metody `mutation` tedy budeme volat metodu `query`. Ta opět bere funkci, které předá objekty `ctx` a `input`. Místo volání metody `create` ale budeme volat metody `findMany`, která vrátí všechny záznamy, které odpovídají zadání. To jsou v našem případě všechny posty, které existují. Zároveň potřebujeme dostat informaci o vlastníkovi daného příspěvku (například abychom mohli získat jeho profilový obrázek, když příspěvek zobrazujeme) a o všech lajcích, které příspěvek obdržel. Automaticky nám `prisma` tyto hodnoty nevrátí, aby se snížila náročnost požadavku. Můžeme ale `prismu` donutit vrátit i záznamy uživatele a lajků předáním objektu `include`. Výsledného efektu docílíme napsáním názvu relace a přidělením hodnoty `true`. Dále předáme seznam s názvem `orderBy`. Každý objekt v něm bude představovat jednu hodnotu, podle které se bude řadit. My budeme chtít řadit pouze podle data vytvoření tak, aby nejnovější příspěvky byly v listu první.



```
listPosts: protectedProcedure
  .input()
  .query(async ({ ctx, input }) => {
    const { prisma } = ctx;

    return await prisma.post.findMany({
      orderBy: [{ createdAt: "desc" }],
      include: {
        owner: true,
        likes: true,
      },
    }),
  });
```

Obr. 21: Volání procedury post v aplikaci GatherlyHub

Teď stačí proceduru zavolat na klientovi. Narozdíl od mutace se queries zavolají automaticky, když na ně dojde řada při provádění kódu. Stačí tedy vzít hodnotu data, kterou procedura vrátí. Musíme ale volat metodu useQuery a ne useMutation.

```
const { data } = api.post.listPosts.useQuery();
```

Obr. 22: Volání procedury listPosts v aplikaci GatherlyHub

Nyní ale vzniká problém. V databázi sice přibyl nový příspěvek. Na klientovi ale vidíme pořád stará data. Přímou tRPC nabízí funkci „invalidate“, která označí procedury typu query jako zastaralé a donutí je provést se znova. Vytvoříme si objekt utils, který získáme zavoláním metody useContext() na objektu api. Podobně, jako bychom na objektu api volali proceduru, zavoláme metodu invalidate na objektu utils. My ale nebudeme chtít označit pouze proceduru listPosts. V budoucnu by mohl počet procedur, které získávají příspěvky z databáze růst (například bychom mohli mít proceduru, která bere pouze příspěvky, které patří uživateli). Zavoláme tedy invalidate na celém routeru post a tím označíme všechny procedury typu query, které jsou na něm a na routerech na něm navázaných, definované. Mohli bychom zároveň zavolat invalidate přímo na objektu utils, čímž bychom označili každou query v aplikaci za neplatnou. V našem případě to ale nedává smysl.

```

const utils = api.useContext();

const { mutate: post } = api.post.post.useMutation({
  onSuccess: async () => {
    await utils.post.invalidate();
  },
});

...

post({
  post_text: postTextInput,
});

```

Obr. 23: Invalidace procedur v aplikaci GatherlyHub

Pojďme vyřešit poslední problém, který aktuální implementace způsobuje. Kdyby v databázi byly desetitisíce příspěvků, budeme pokaždé, když uživatel přidá příspěvek, upraví příspěvek, klikne do jiného okna a pak zpátky do prohlížeče nebo obnoví stránku, stahovat nepřehledné množství dat. Nejen že by to mělo nepříjemný dopad na rychlost aplikace, zároveň bychom potřebovali výrazně výkonnější hardware, abychom takovou aplikaci mohli provozovat. Naštěstí existuje jednoduché řešení. Metoda `useInfiniteQuery`. Narozdíl od metody `useQuery` dostaneme přístup k metodám a datům, které nám pomůžou načítat pouze tolik příspěvků, kolik potřebujeme. Myšlenka stojí na principu, že budeme brát určitý počet příspěvků a pamatovat si, kde jsme skončili. Zároveň budeme všechny příspěvky, které jsme už načetli, ukládat do cache paměti. K vysvětlení implementace bude snazší prvně ukázat výsledný kód a poté popsat, co se v něm děje. Začneme tvorbou zod schématu pro teď již potřebný input procedury a pak úpravou procedury jako takové.

```

export const listPostsInput = z.object({
  cursor: z.string().nullish(),
  limit: z.number().min(1).max(100).default(10),
});

```

Obr. 24: Zod schéma procedury `listPosts` v aplikaci GatherlyHub

Hodnota „`cursor`“ bude představovat id příspěvku, u kterého budeme příště chtít začít načítat. Hodnota `limit` bude představovat počet příspěvků, který při každém volání vrátíme.

```
listPosts: protectedProcedure
  .input(listPostsInput)
  .query(async ({ ctx, input }) => {
    const { prisma } = ctx;
    const { cursor, limit } = input;

    const posts = await prisma.post.findMany({
      take: limit + 1,
      orderBy: [{ createdAt: "desc" }],
      cursor: cursor ? { id: cursor } : undefined,
      include: {
        owner: true,
        likes: true,
      },
    });

    let nextCursor: typeof cursor | undefined = undefined;

    if (posts.length > limit) {
      const nextPost = posts.pop() as typeof posts[number];

      nextCursor = nextPost.id;
    }

    return {
      posts,
      nextCursor,
    };
  }),
```

Obr. 25: Procedura listPosts upravená pro fungování s useInfiniteQuery v aplikaci GatherlyHub

Všimněme si, že objektu, který definuje požadavek findMany, přidáváme hodnotu „take“ a nastavujeme ji na limit + 1. To proto, že chceme získat id příspěvku, od kterého příště začneme příspěvky vracet. Předáváme také string cursor, který bude mít buď hodnotu undefined, což znamená, že se příspěvky mají čerpat od začátku nebo hodnotu id příspěvku. Příspěvky ale rovnou nevracíme. Prvně odstraníme poslední příspěvek z listu, pokud je delší než hodnota limit. To proto, že kdybychom došli až na konec databáze, mohlo by se stát, že bude zbývat méně příspěvků, než limit + 1. Jinak nastavíme proměnnou nextCursor na hodnotu id posledního příspěvku. Pak jen vrátíme objekt, ve kterém je list příspěvků a cursor. Nyní musíme výsledek nějak zpracovat na klientovi.

```
const { data, hasNextPage, fetchNextPage, isFetching } =  
  api.post.listPosts.useInfiniteQuery(  
    { limit: 5 },  
    {  
      getNextPageParam: (lastPage) => lastPage.nextCursor,  
    }  
  );  
  
const posts = data?.pages.flatMap((page) => page.posts) ?? [];
```

Obr. 26: Volání procedury listPosts s useInfiniteQuery v aplikaci GatherlyHub

Z volání procedury nyní očekáváme více výsledků. Data opět reprezentují naše příspěvky. Tentokrát jsou ale seskupeny do listu pages. Každý list příspěvků v listu pages odpovídá jednomu volání procedury. Hodnota „hasNextPage“ je typu boolean a říká, jestli jsou v databázi ještě nějaká data, která jsme nenačetli. Metoda „fetchNextPage“ opětovně volá proceduru listPosts s daty, které jsme upřesnili v parametrech metody useInfiniteQuery a vrací další „page“. Hodnota „isFetching“ je typu boolean a říká, jestli ještě probíhá načítání (mohli bychom například zablokovat tlačítko „načíst další“, pokud by ještě nebylo načítání dokončeno). Předávání hodnoty limit je vcelku jasné. Předávání hodnoty cursor je ale trochu náročnější. Metoda useInfiniteQuery očekává, že obdrží nějaký kurzor. Má proto přístup k mnoha funkcím, které by mohly usnadnit jeho zapsání. Funkce „getNextPageParam“ nastavuje kurzor (hodnotu cursor, protože ví, že se jedná o kurzor). Zároveň pokud předáme jako hodnotu funkci, dostane poslední načtenou stránku jako parametr. Každá stránka na sobě má hodnotu „nextCursor“, kterou spolu s listem „posts“ vracíme v proceduře listPosts. Zápisem tedy říkáme, že chceme vzít hodnotu „nextCursor“ z poslední načtené stránky a nastavit ji jako kurzor pro useInfiniteQuery, který ji pak předá jako input „cursor“ proceduře listPosts. Ukládání dat do chace paměti nemusíme řešit, děje se automaticky za nás.

Nakonec zbývá jenom dostat všechny příspěvky do jednoho listu. Toho můžeme docílit pomocí metody flatMap.

Tvorba ostatních procedur, které aplikace GatherlyHub, je takřka totožná.

## 7 Závěr

Cílem práce bylo posoudit schopnosti ekosystému technologií známého jako „T3 stack“. Konkrétně jsme se zaměřili na tvorbu webové aplikace, která slouží jako sociální síť. V průběhu vývoje jsme narazili na mnoho náročných úkolů, se kterými se vývojáři webových aplikací často potýkají. Problémy spojené s autentikací uživatelů, bezpečným přístupem k databázi, efektivní práci s daty a mnoha dalšími. Naštěstí technologie T3 stacku poskytují silné nástroje, které umožnily tyto problémy relativně snadno vyřešit. Čím více se aplikace zvětšovala, tím více jsem z pohledu vývojáře oceňoval přínosy T3 stacku. Výsledná aplikace byla robustní a jednoduše by se dala rozšířit o novou funkcionalitu. T3 stack jako takový je ale velmi komplikovaný ekosystém technologií. Bylo náročné se v něm zorientovat a jakožto u relativně nové technologie je možné dohledat menší množství studijních materiálů. Struktura projektu je komplikovaná, aby uspokojila potřeby velkých aplikací. Proto bych T3 stack doporučil pro tvorbu středních a velkých projektů. Na malé projekty je zbytečně komplexní. Zejména ve velkých projektech ale T3 stack přináší mnoho prostředků, které pomáhají překonávat překážky spojené s vývojem webových aplikací.

## 8 Literatura

[T3] *T3* [online] [vid. 6. 4. 2023].Dostupné z: <https://create.t3.gg/en/introduction>.

[Next.js] *Next.js* [online] [vid. 6. 4. 2023]. Dostupné z: <https://nextjs.org/>.

[Tailwind CSS] *Tailwind CSS* [online] [vid. 6. 4. 2023].Dostupné z:  
<https://tailwindcss.com/docs/installation>.

[React] *React* [online] [vid. 7. 4. 2023].Dostupné z: <https://legacy.reactjs.org/>.

[tRPC] *tRPC* [online] [vid. 7. 4. 2023].Dostupné z: <https://trpc.io/docs/>.

[Prisma] *Prisma* [online] [vid. 8. 4. 2023].Dostupné z: <https://www.prisma.io/>.

[NextAuth.js] *NextAuth.js* [online] [vid. 8. 4. 2023].Dostupné z:  
<https://next-auth.js.org/>.