

ZÁVĚREČNÁ STUDIJNÍ PRÁCE

dokumentace

Effio - webová aplikace pro vytváření testů



Autor: Matěj Kotrba
Obor: 18-20-M/01 INFORMAČNÍ TECHNOLOGIE
se zaměřením na počítačové sítě a programování
Třída: IT4
Školní rok: 2023/24

Poděkování

Rád bych poděkoval Mgr. Markovi Lučnému za poskytuní konzultace ohledně tohoto projektu.

Prohlášení

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně a uvedl veškeré použité informační zdroje.

Souhlasím, aby tato studijní práce byla použita k výukovým a prezentačním účelům na Střední průmyslové a umělecké škole v Opavě, Praskova 399/8.

V Opavě 1. 1. 2024

.....
Podpis autora

Abstrakt

Výsledkem projektu je funkční webová aplikace pro vytváření a vyplňování testů, které se skládají z různých možností otázek včetně programovací. Aplikace zahrnuje přihlášení přes Google a GitHub. Uživatel vytváří jednotlivé testy výběrem šablony, nebo importem z GIFT formátu, otázek, komentářů a následně upravuje detaily testu. Hotový test může sám zkusit z vlastní kolekce testů a nebo z komunitního centra, kde se nacházejí komunitou vytvořené testy, po vyplnění testu se uživateli objeví výsledky a známka. Kromě tvorby a vyplňování testů aplikace obsahuje také skupiny, v nichž mohou mezi sebou uživatelé např. komunikovat, na dříve vyplněné testy se může podívat v sekci testové historie. Přehled o aktivitě uživatele si může prohlédnout v dashboardu prostřednictvím vizuálních grafů. Dříve vytvořené testy se dají v části kolekce editovat, mazat a také exportovat do GIFT formátu v textovém souboru pro použití například v Moodlu. Aplikace disponuje zcela responsivním designem se světlým a tmavým režimem.

Klíčová slova

webová aplikace, databáze, responsivní design, účty, grafy, tvorba testů, barevné režimy

Abstract

The result of the project is a functional web application for creating and taking tests, which consist of various types of questions, including programming questions. The application supports login via Google and GitHub. Users can create individual tests by selecting a template or importing from the GIFT format, including questions, comments, and then customize the details of the test. The completed test can be tried by the user from their own collection of tests or from the community center, where tests created by the community are available. After completing the test, users will see the results and a grade. In addition to test creation and completion, the application also includes groups where users can communicate with each other. Users can review previously taken tests in the test history section. An overview of user activity can be viewed in the dashboard through visual graphs. Previously created tests can be edited, deleted, and exported to the GIFT format in a text file for use, for example, in Moodle. The application features a fully responsive design with both light and dark modes.

Keywords

web application, database, responsive design, user accounts, graphs, test creation, color modes

Obsah

Úvod	2
1 Architektura a koncepty aplikace	3
1.1 Architektura	3
1.2 Typesafety	4
2 Backend	6
2.1 Založení a konfigurace projektu	6
2.2 Architektura backendu	6
2.3 Autentifikace	7
2.4 Databáze	9
2.5 Využité backendové technologie	10
2.6 SvelteKit	10
3 Frontend	15
3.1 Design	15
3.2 Responsivita	15
3.3 Světlý a tmavý režim	16

3.4	Využité frontendové technologie	16
4	Funkce aplikace	21
4.1	Domovská stránka	21
4.2	Přihlášení	22
4.3	Testy a jejich vlastnosti	22
4.4	Vyplňování testu	23
4.5	Zobrazování testů	25
4.6	Skupiny	26
5	Zhodnocení práce	27
5.1	Splněné a nesplněné cíle	27
A	Databázový model	31

ÚVOD

V dnešní době se běžně využívají webové aplikace, které umožňují vytváření testů/kvízů, které poté jiní uživatelé vyplňují. Prostředí těchto aplikací jsou však často nepřehledné a vytváření testů či kvízů je úporné. S touto myšlenkou jsem se rozhodl vytvořit aplikaci, která by kombinovala možnosti jiných aplikací s přehledným moderním zobrazením a dalšími užitečnými prvky.

Má aplikace by kromě již zmíněné funkcionality pro tvorbu testů a kvízů měla do jisté míry umožňovat prvky sociálních sítí jako třeba skupiny, komunitní místo kde by se mimo jiné zobrazovaly testy ostatních uživatelů. Hlavní myšlenkou bylo vytvořit nejen aplikaci jako takovou ale také využít moderní technologie a postupy, neboli vytvořit ji „typesafe“, bez potřeby vlastního serveru za pomocí cloudové technologie „serverless“ a plně responsivní pro uživatele na kterémkoli zařízení.

V dokumentaci jsou popsány využité technologie, postupy a jednotlivé funkcionality celé aplikace. První část popisuje architekturu a přístup k řešení, následuje popis backendu a frontendu, ve čtvrté části kapitole se potom zmiňují o různých možnostech, které Effio nabízí. Nakonec se poohlížím na dosažené cíle a možné vylepšení.

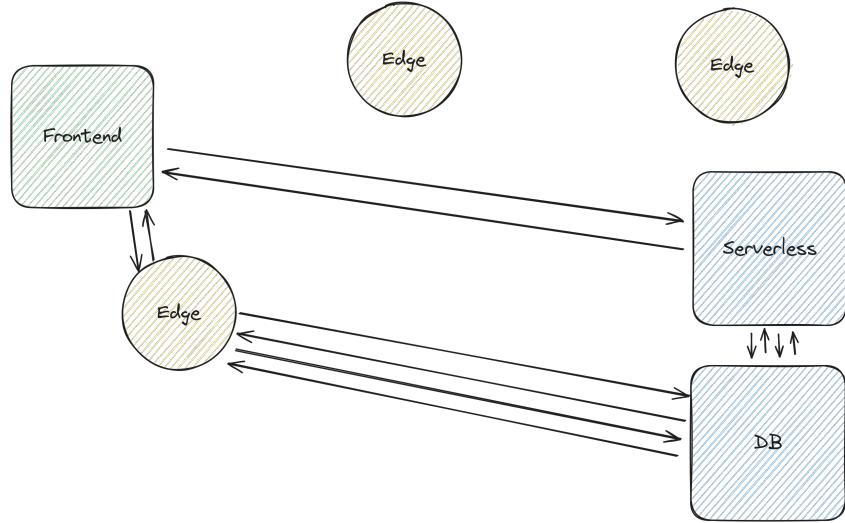
1 ARCHITEKTURA A KONCEPTY APLIKACE

1.1 ARCHITEKTURA

1.1.1 Možnosti řešení

Pro vytvoření webové aplikace je možno využít mnoho postupů, proto zmíním několik variant nad kterými jsem uvažoval s jejich klady se záporou

- Tradiční web server - jedná se o nejběžnější variantu jak vytvářet webové stránky. Jednotlivé stránky jsou vytvářeny na serveru, následně jsou poslány na klienta, poslaný kód může obsahovat také JavaScript pro frontend funkcionality, pro backend je možné využít jazyk dle výběru. Pro veškeré přesměrování a další akce je nutné se obrátit na server.
- Single page application - toto řešení v podstatě odstraňuje server a nechází veškerou zodpovědnost frontendovém frameworku jako např. React, Svelte nebo Solid, toto řešení ale nemá k dispozici žádný způsob jak spouštět kód, která na klientu pouštět nemůžeme jako například SQL dotazy. Další nevýhodou je to, že stránka je generovaná až na klientovi JavaScriptem, proto search enginy nejsou schopny detektovat obsah stránky, což vede k mnohem nižším výsledkům SEO.
- Serverless - je architektura kdy vývojář využívá server poskytovatele, o ten se nemusí starat, a je škálován podle potřeby. Tento koncept ale má i své nevýhody, omezená doba relace odpovědi, menší úložný prostor pro načtení knihoven nebo třeba nemožnost spravovat svůj server.
- Edge runtime - tato technologie je podobná Serverless architektuře, serverový kód ale neběží v jedné lokalitě ale na jednotlivých CDN. Funkce se spouštějí ne skrze Node ale přes Edge runtime, toto obsahuje svou nevýhodu, Edge není Node, a proto nemůžeme používat Node moduly, jako je třeba „fs“. Další nevýhodou je velice malé množství paměti, které je pro dostupnou instanci dostupné. Výhodou je poté velice nízká cena spuštění takové funkce a bezkonkurenční rychlosť odpovědi, tato výhoda je největší u serverových úkolů jako přesměrování, cookies nebo geograficky založených údajů, v případě několikanásobných dotazů do databáze se ale cesta potřebná k získání dat zvětšuje a výhoda rychlosti mizí.



Obrázek 1.1: Rozdíl mezi serverless a edge.

- Metaframework je technologie, která kombinuje výhody „single page application“ a tradičního web serveru. Disponuje možností běhu kódu na serveru, přesměrováním na klientské části a dalšími výhodami. Takových technologií existuje celá řada, jako například populární NextJS, já si pro svůj projekt zvolil SvelteKit. Další fází této architektury je hostování, nejlepší variantou většinou bývá hostování přes providery jako Vercel nebo Netlify, tato architektura se poté spíše primárně aplikuje se „serverless“.

Jednou z hlavních myšlenek bylo hostování Effia na cloudových službách, proto jsem si vybíral hlavně mezi technologiemi „serverless“ a „edge computing“, výhodou providera, kterého jsem si vybral - Vercel je, že kombinace těchto technologií je velice snadná, základní variantou je „serverless“ s jednoduchým přepnutím dané cesty na „edge“. Ve spojení s konceptem metaframeworku nakonec utváří velmi flexibilní, rychlou a příjemnou variantu.

```

about/+page.ts

import type { Config } from '@sveltejs/adapter-vercel';

export const config: Config = {
  runtime: 'edge',
};

```

Obrázek 1.2: Možnost přepnutí dané cesty ze Serverless na Edge. [2]

1.2 TYPESAFETY

Webové aplikace standardně využívají JavaScript, ten ale obnáší signifikantní nevýhodu v podobě nemožnosti „otypovat“ kód, to způsobuje obtíž orientovat se v kódu, velké množství

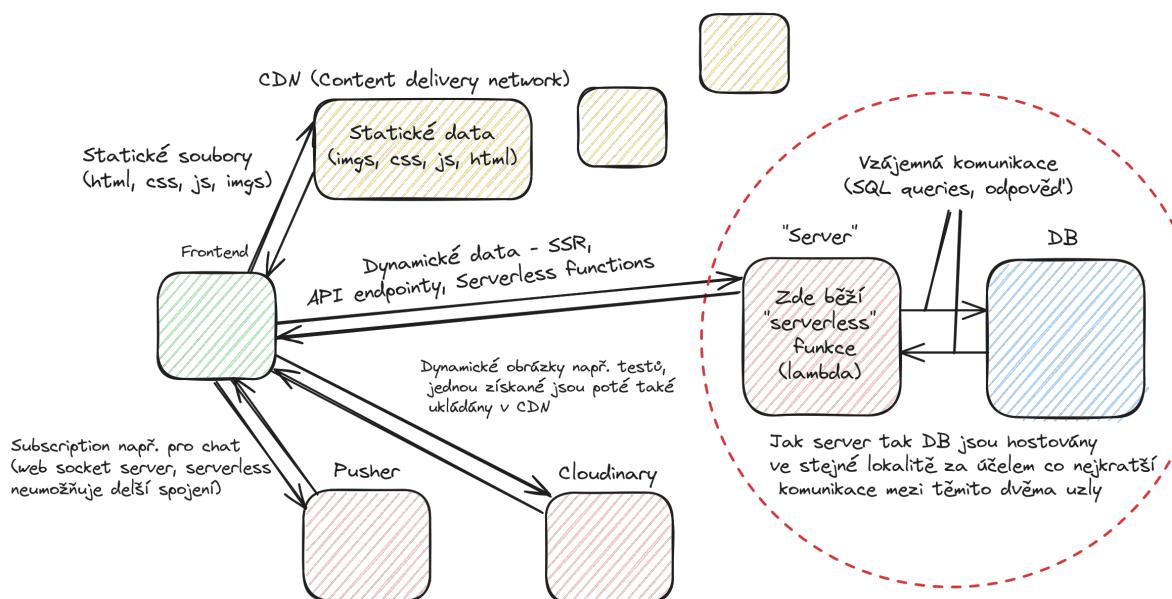
produkčních chyb a také spoustu času stráveného pochopením dříve napsaného kódu. Pro Effio jsem se tedy rozhodl využít moderní technologie a vytvořit tak téměř plně „typesafe“ (otypovanou) aplikaci. TypeScript v Effiu nahrazuje JavaScript, ten do tohoto jazyka přináší typy. To ale nestačí, API endpointy, stejně jako databázové dotazy stále nemůžou být otypované a proto jsem připojil také knihovny tRPC a Prisma.

2 BACKEND

2.1 ZALOŽENÍ A KONFIGURACE PROJEKTU

Prvním krokem bylo založení projektu a stažení potřebných knihoven technologií, které jsem plánoval využít. Kombinace mnou vybraných technologií nebyla kompletně konvenční a proto jsem se musel v některých případech obrátit na komunitou vytvořené adaptéry, příkladem je například knihovna `trpc-sveltekit`, která propojuje SvelteKit a tRPC, které je primárně navrženo buď jako samostatný server a nebo jako implementace do Next.js.

2.2 ARCHITEKTURA BACKENDU



Obrázek 2.1: Jednoduchý přehled backendové části Effia.

- **CDN** - neboli Content Delivery Network je síť serverů, které jsou charakteristické hlavně tím, že jsou rozmištěny po celém světě a ve velkém množství, starají se o distribuci statického obsahu díky čemu nemusí klient data získávat ze vzdáleného server ale právě z tohoto, který je ve většině případů mnohem blíže. Dále se starají o cachování dat, což opět zkracuje dobu pro uživatele aby zobrazil obsah.

- Server - neoznačuje server jako takový ale spíše místo kde se spouštějí instance serverových funkcí, což jsou funkce, které se vytváří podle potřeby na reálných serverech, které ale spravuje provider této služby, v méém případě Vercel, respektive Cloudflare.

Programátora nemusí tyto serveru vůbec zajímat, instance se sami škálují a obecně jsou pro vývojáře velice příjemných řešením. V poslední letech se tato architektura těší významné oblibě nejen malých projektů ale také velikých firem jako je Amazon, ty se ale v poslední době opět začínají vracet k hostování vlastních serverů, které se po dlouholehlých diskuzích znova zdají výhodnější pro takto masivní účely typu Amazonu.

Tento server je v Effiu využíván hlavně pro první zobrazení stránky metodou SSR („server side rendering“) a také získáváním dat, které nemohou být získány na klientu (např. SQL dotazy), pro jednotlivé stránky, formulářovými akcemi nebo jako API endpointy.

- DB - MySQL databáze hostovaná přes službu Planetscale, ta disponuje velice zajímavými možnostmi jako jsou „větve“ podobné verzovacímu systému Git, „Deploy requesty“, které zlepšují práci v týmu nebo možnost distribuce „read-only“ instancí databáze do rozdílných regionů. Pro mě byla veliká výhoda rychlosť a velice štědrý „free tier“.
- Pusher, Cloudinary - jedná se o cloudové služby, které slouží účelům, které s touto architekturou nejsem schopný zařídit.
 - Pusher se stará o web sockety, respektive stále spojení, které se „serverless“ spojením není možné. V Effiu posloužil pro chat v kanálech skupin pro aktualizaci zpráv všech uživatelů pokud nějaký pošle zprávu.
 - Cloudinary slouží pro ukládání obrázků a jejich distribuci do CDN.

2.3 AUTENTIFIKACE

2.3.1 Auth.js

Auth.js je knihovna sloužící pro autentifikaci, poskytuje možnost „session based“, to je použito v Effiu, a JWT autentifikace. Dále knihovna podporuje OAuth s mnohými providery, v tomto projektu je využit GitHub a Google s jednoduchou možností přidat další. Výhodou knihovny je, že data si vývojář spravuje sám, neboli jsou ukládána do jeho vlastní databáze v podobě tabulek (které si také může sám upravit): Account, Session, User a Verification Token, které poskytují naprostou kontrolu nad ověřením uživatelů.

2.3.2 Proces přihlášení

Uživatel se na podstránce /login rozhodne zdali se přihlásit pomocí Google nebo GitHub účtu, je poté přesměrován na stránku těchto providerů, kde potvrdí přístup k informacím o jejich účtu a je navrácen zpět do Effia. V databázi jsou vytvořeny tabulky o tomto uživateli včetně nové relace (Session), díky které je schopen přihlášení. To proběhne automaticky po navrácení se zpět od providera.

2.4 DATABÁZE

Databáze je podrobněji popsaná v backendové architektuře, ale pro shrnutí se jedná o MySQL-compatible databázi hostovaná přes Planetscale. Jako ORM využívám Prismu. Databázový model je umístěn jako příloha A.1.

2.4.1 Stručný popis účelů jednotlivých tabulek

Autorizace a autentifikace

- User - tabulka s údaji o uživateli.
- Account - účet uživatele, typ providera přes kterého je přihlášen a data k relaci.
- Verification Token - ověřovací identifikátor providera.
- Session - relace, do jisté míry propojuje jednotlivé tabulky.

Otázky

- Question - otázka jako taková, obsahuje název, popis, její propojení s testy atd.
- QuestionType - typ otázky, rozhoduje jejím chováním na frontendu.
- QuestionRecord - záznam otázky, po vyplnění testu se vytvoří ke každé otázce jeden nový záznam s výsledky, je sdružován Test Recordem.

Testy

- Test - obsahuje název testu, jeho popis a sdružuje veškeré další podrobnosti testu jako jsou Tagy, Stars, obsahuje množství TestVersion, což je vlastně jednotlivá verze testu.
- TestVersion - verze testu, uchovává odpovědi, body a Mark System, verze se aktualizují při každé změně testu.
- TestRecord - záznam vyplněného testu, obsahuje také Question Records.

Ostatní tabulky týkající se otázek

- TestStar - ohodnocení testu hvězdičkou, každý uživatel mimo majitele může test takto ohodnotit, ekvivalent "liku" na sociálních sítích
- MarkSystem - známkovací systém daného testu, uživatel si ho může sám upravit a při uložení testu je uchován právě v této tabulce.

- Tag/TagOnTest - uchovává štítky týkající se tématu testu vybrané majitelem.

Skupiny

- Group - skupina pro uživatele, obsahuje základní vlastnosti skupiny jako jméno, slug apod.
- GroupSubcategory - jednotlivý kanál skupiny, každý tento kanál má i samostatné zprávy, chat apod.
- GroupSubcategoryMessage - Zpráva v kanálu, vztahuje se k ní také její odesilatel, název, obsah atd. Obsahuje také MessageType, což je druh zprávy, kterou uživatel poslal.

Ostatní

- Template - šablona testu

2.4.2 Cloud hosting

Jak již bylo zmíněno v úvodu tak tato aplikace by se měla obejít bez vlastního serveru, nejde ale jenom o databázi ale také například o ukládání obrázků nebo hostování aplikace jako takové, to je prováděno přes „Cloud hostingy“ jako Vercel pro hostování stránky jako takové, zároveň ale řeší i rozesílání statických dat do CDN a poskytuje serverless lambda funkce, Planetscale pro hostování mojí MySQL databáze, Cloudinary, který slouží jako „bucket“ pro obrázky a také jejich možnost editace přes url parametry, nebo Pusher, který slouží jako web socket server například pro chat.

2.5 VYUŽITÉ BACKENDOVÉ TECHNOLOGIE

2.6 SVELTEKIT

SvelteKit je metaframework postavený na Svelte, jako ostatní metaframeworky typu NextJS nebo SolidStart Jeho hlavní výhody se skládají z:

- Rychlosť - Svelte vytváří velice rychlou aplikaci, v kombinaci s Vitem (bundler) se ale také spojuje s velice rychlým build timem a hot module replacementem. To ale není jediné místo kde se rychlosť projevuje, za pomocí SvelteKitu se aplikace využívá velmi rychle díky velice malému množství „boilerplate“ kódu.

- Flexibilita - Aplikace často potřebuje různé typy vykreslování stránek, SvelteKit dovoluje jednoduše nakonfigurovat jednotlivé stránky či cesty pro specifické způsoby jako SPA, SSR, SSG nebo MPA. Dále také umožňuje různě kombinovat kód, který běží na serveru a ten co se spouští na klientovi, dává nad nimi rozsáhlou kontrolu a jednoduše se jeho chování upravuje.
- Přehlednost - SvelteKit využívá „file based routing“, tedy cesty aplikace jsou generovány podle složek, které vývojář vytvoří, soubory se poté vždy jmenují stejně, `+page.svelte` pro stránku, `+layout.svelte` pro layout apod. Díky tomu je vždy jasné pro co specifický soubor slouží, přehlednosti také přidává, již u Sveltu zmíněná podobnost s JavaScriptem.

V mé aplikaci SvelteKit zastává naprosto zásadní roli, od routování, přes serverové operace jako load funkce a API endpointy, konfiguraci bundleru až po přesměrování. Ve zkratce se jedná o základní stavební blok, bez kterého by aplikace nebyla funkční.

Tento kód ukazuje získání dat z databáze při načtení stránky, ale před zobrazením uživateli, (load funkce) a poté také actions, což jsou formulářové akce vytvářené pro specifickou cestu, které poté můžeme využívat, zde je vidět mazání uživatele ze skupiny

```

1 export const load: ServerLoad = async (event) => {
2   // Mohou se vracet i Promisy, SvelteKit je sám resolvne, mění se ve SvelteKit 2.0
3   const users = prisma.user.findMany({ where: { name: event.params.name } })
4   return users
5 }
6 export const actions: Actions = {
7   deleteUsers: async (event) => {
8     const formData = await event.request.formData()
9     const users: string[] = []
10    formData.forEach((value) => { users.push(value.toString()) })
11    try {
12      await (await trpcServer(event)).groups.kickUsersFromGroup({
13        groupSlug: event.params.name as string,
14        userIds: users,
15      })
16      return { success: true }
17    }
18    catch (e) {
19      if (e instanceof TRPCError) {
20        return fail(getHTTPStatusCodeFromError(e), { message: e.message })
21      }
22      else { return fail(500, { message: "Something went wrong." }) }
23    }
24  }
25}
```

Kód 2.1: Ukázka z `+page.server.ts`

2.6.1 tRPC

V minulé kapitole jsem se zmínil o problémech s otypováním API endpointů, tRPC (Type-script Remote Procedure Call) tento problém řeší tím, že vytváří dynamické typy pro jednotlivé endpointy, podle toho jak si je sami nadefinujeme, ty se potom dají volat pomocí funkcí bez přímého použití fetche nebo třeba axiosu (tyto funkce ve skutněnosti „fetch“ requesty vykonávají ale programátor se o ně nemusí starat přímo), tyto funkce jsou dokonale otypované a v kódu tím pádem fungují jako jakákoli jiná funkce vytvořená programátorem. Další výhodou je také to, že jednotlivé „procedury“, což je vlastně API endpoint, mají k dispozici metody pro kontrolu vstupu nebo třeba middleware, který například může zjišťovat stav přihlášení uživatele, jako zbytek knihovny jsou i tyto metody perfektně otypované.

V Effiu tRPC využívám hlavně jako možnost komunikace klienta s databází, neboli tRPC v build timu vytvoří API endpointy, které poté klient přes zmíněné funkce volá. V těchto endpointech se většinou nachází operace s databází, ty na klientu provádět nemohu. Výhodou je, že tyto funkce mohu využívat i na serveru s trochu odlišnou implementací.

Ukázka kódu zobrazuje vytvoření procedury, neboli endpointu, který se poté dá volat. V druhé části je zobrazena právě zmíněná funkce, díky které získáme potřebné data.

```
1 getTestById: procedure.input(z.object({
2   id: z.string(),
3   includeGroupSubcategories: z.boolean().optional()
4 })) .query(async ({ ctx, input }) => {
5   const test = await ctx.prisma.test.findUnique({
6     where: { id: input.id },
7     include: {
8       subcategories: input.includeGroupSubcategories || false,
9       owner: true,
10      tags: { include: { tag: true } },
11      testVersions: {
12        include: {
13          questions: { include: { type: true } }
14        },
15        orderBy: { version: "desc" },
16        take: 1
17      }
18    },
19  })
20
21  if (!test) return null
22  return test
23 }),
```

Kód 2.2: Endpoint generovaný pomocí tRPC

```

1 const imageUrlToDeleteTest = await trpc(get(page)).getTestById.query({
2   id: props.data.id,
3 })

```

Kód 2.3: Volání funkce pomocí tRPC klienta s metodou getTestById

2.6.2 Prisma

Prisma slouží jako ORM (Object–relational mapping), to znamená pomocí JavaScriptu získavat data z databáze bez přímého použití jazyka SQL, Prisma se skládá z klientské části, která pomocí protokolu založeném na JSON komunikuje se serverovou částí, tam je následně uskutečněn SQL dotaz a odpověď je poslána klientovi. Také řeší již zmíněný problém s otypováním těchto dotazů, pro Prismu je totiž nutné vytvořit schema.prisma soubor kde se definuje model databáze, ten poté můžeme pomocí Prisma CLI nahrávat do databáze ale také vytvářet dynamické typy, které poté využijeme jak v dotazech tak v aplikaci pro data, která dostaneme zpět. Prisma je asi hlavní důvod nemožnosti využívat Edge runtime, tato knihovna totiž potřebuje využít TCP spojení pro komunikaci s databází, které aktuálně Edge runtime nepodporuje, jako řešení by se nabízelo využít jinou knihovnu jako Drizzle ORM nebo Database JS. I přes to se na řešení tohoto problému pro Prismu intenzivně pracuje a ve velmi brzké budoucnosti se očekává řešení (jiné než další balíčky a úprava kódu jako nyní).

Effia Prismu využívá jako ORM, tedy pro veškeré databázové operace, v celé aplikaci se nevyskytuje jediná SQL query, která by nevyužívala Prismu buď pro sestavení dané query nebo minimálně pro komunikaci s databází.

Tato ukázka kódu zobrazuje SQL operaci, kterou Prisma vytvoří podle této objektové struktury, kterou jsem sestavil. Cílem je získat unikátní test pomocí jeho ID společně s přidruženými tabulkami. Druhá ukázka poté vytváření modelu.

```

1 const test = await ctx.prisma.test.findUnique({
2   where: {id: input.id},
3   include: {
4     subcategories: input.includeGroupSubcategories || false,
5     owner: true,
6     tags: { include: {tag: true} },
7     testVersions: {
8       include: { questions: { include: { type: true } } },
9       orderBy: { version: "desc" },
10      take: 1
11    }
12  },
13})

```

Kód 2.4: Získání testu podle id a přidání dat ze spojených tabulek

```

1 model Question {
2     id      String          @id @default(uuid())
3     title   String
4     createdAt DateTime       @default(now())
5     updatedAt DateTime       @updatedAt
6     typeId   String
7     testId   String
8     content  Json
9     points   Int            @default(0)
10    type     QuestionType  @relation(fields: [typeId], references: [id], onDelete:
11        Cascade)
12    test     TestVersion    @relation(fields: [testId], references: [versionId],
13        onDelete: Cascade)
14    records  QuestionRecord[]
15
16    @@index([typeId])
17    @@index([testId])
18}

```

Kód 2.5: Schéma modelu verze testu

2.6.3 Zod

Zod je validační knihovna jako např. Yup. To znamená, že jeho prací je kontrolovat mnou vložené vstupy, knihovna vrací úspěšnost a také chyby na které během kontroly narazí. Jeho výhodou je avšak možnost využít jeho validační schémata jako typy a také samotná validace funguje jako „type guard“ (kontroluje a nastavuje typy u vložené proměnné).

Zod je v Effiu využíván jak na backendu tak na frontendu, jeho umístění do backendové části je pouze z důvodu, že validační operace častěji probíhají na serveru. Díky Zodu jsem schopen přehledným a spolehlivým způsobem kontrolovat data testů, formulářů apod. Zásluhou hezký formátovaných vrácených chyb je poté mohu bezproblémově zobrazovat uživateli.

Tento kód kontroluje zdali vložený input odpovídá struktuře „answerSchema“, popřípadě nastaví error do proměnné, která se poté zobrazí klientovi.

```

1 const answerSchema = z.string().min(ANSWER_MIN, `Answer has to be at least ${ANSWER_MIN} character long.`).max(ANSWER_MAX, `Answer can be max ${ANSWER_MAX} characters long.`)
2 const result = answerSchema.safeParse(content.answers[item].answer)
3 if (result.success === false) {
4     content.answers[item].error = result.error.errors[0].message
5 }

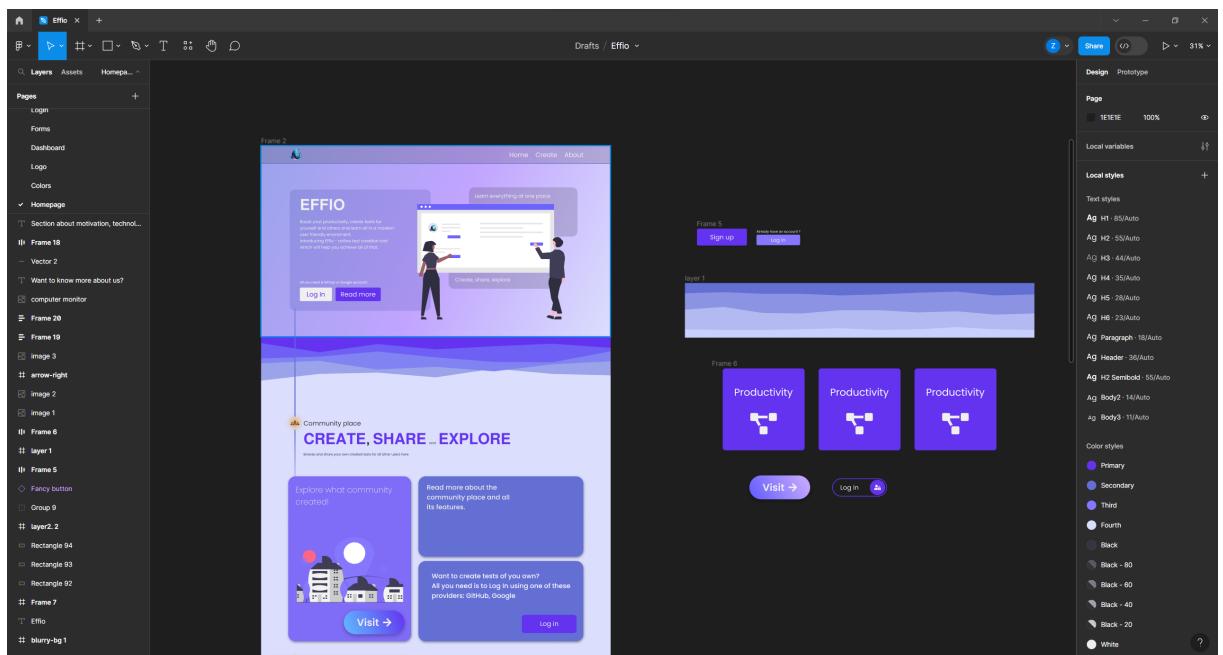
```

Kód 2.6: Validace vstupu pomocí validační knihovny Zod

3 FRONTEND

3.1 DESIGN

Jednou z hlavních myšlenek bylo vytvořit pohledem přívětivou aplikaci, proto se návrh designu stál klíčovou částí pro stylově propracovanější prvky stránky. Pro tvorbu designu, stejně jako vytváření a úpravu potřebných obrázků jsem využil aplikaci Figma.



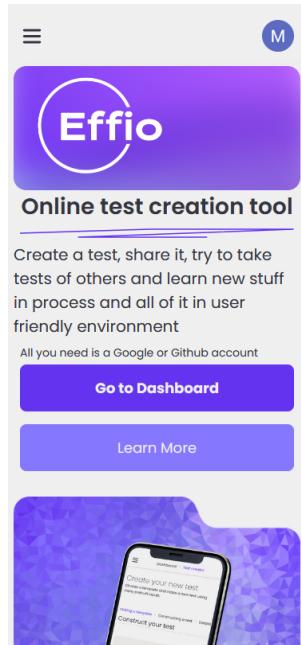
Obrázek 3.1: Prvotní návrh domovské stránky.

3.2 RESPONSIVITA

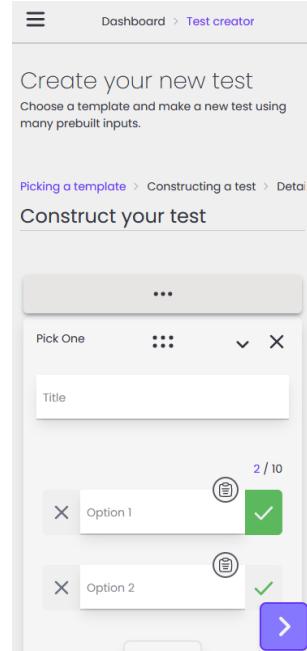
Celá aplikace je uzpůsobená jak pro počítače tak pro mobilní zařízení. Responsivita není vůbec lehká práce, mně ale pomohl Tailwind, ve kterém se CSS media query dělají snadněji společně s moderními CSS containers, které umožňují responsivní breakpointy odvozovat nejen od velikosti stránky ale od rozměrů rodičovských elementů.

Problematika však nespočívá pouze v zobrazení prvků ale také v jejich funkcionalitě, která musí fungovat jak s myší, tak s dotekem, příkladem tohoto problému je například jeden z typů

otázek, a to „Connect“, kde se musí jednotlivé spoje přesouvat jak pohybem myši tak při dotykovém vstupu.



Obrázek 3.2: Domovská stránka na mobilním zařízení



Obrázek 3.3: Generátor testů na mobilním zařízení

3.3 SVĚTLÝ A TMAVÝ REŽIM

S ohledem na uživatele co preferují tmavý režim jsem se také rozhodl pro tvorbu tmavého režimu, ten je možné vidět na obrázku 4.4, oba tyto režimy vyžadovali vlastní paletu barev a byly mnohokrát přepracovány aby k sobě jednotlivé barvy co nejlépe pasovaly.

3.4 VYUŽITÉ FRONTENDOVÉ TECHNOLOGIE

3.4.1 Svelte

Svelte je open source JavaScriptový framework vyvíjený od roku týmem Riche Harrise, jeho hlavní výhodou je rychlosť ale také intuitivita, protože jazyk se snaží vypadat jako JavaScript, zatímco rozšiřuje jeho možnosti, díky tomu se za posledních let těší rostoucí popularitě webových vývojářů. Na rozdíl od ostatních webových frameworků (například React, Angular, Solid nebo Qwik) Svelte disponuje vlastním jazykem, který se zapisuje do .svelte souboru, ten se následně kompiluje do vysoce efektivního JavaScriptu. Kód samotný velice připomíná HTML ale a tvoří jej na tři určité části.

1. Script - jedná se o část, kde se vypisují funkce, proměnné a řeší se reaktivní deklarace. Celá tato část je obklopená <script> tagy jako v HTML dokumentu.

```

1  export let inputValue: HTMLTextAreaElement['value'] = '';
2
3  let setError = getContext('setError');
4
5  let inputRef: HTMLTextAreaElement;
6
7  const dispatch = createEventDispatcher();
8
9  function validateInput() {
10    const result = validationSchema?.safeParse(inputValue);
11    if (!result?.success) {
12      dispatch('error', result?.error.errors[0].message);
13      if (typeof setError === 'function')
14        setError(result?.error.errors[0].message);
15    } else {
16      dispatch('error', null);
17      if (typeof setError === 'function') setError('');
18    }
19  }
20
21  function dispatchInputChange() {
22    dispatch('inputChange', inputRef.value);
23  }
24

```

Kód 3.1: Ukázka Svelte kódu ve script tagu

2. Style - tato část slouží jako CSS pro daný dokument, výhodou je ale lokální rozsah aplikovaných stylů (jako u CSS modules), aplikovat zde lze i globální styly díky :global. Osobně mi vyhovuje, že se styly nacházejí v jednom souboru, a i když jsem tento způsob pro stylování Effia převážně nevyužíval, tak osobně mi připadá velice dobře provedený. Celý úsek tohoto kódu je jako v HTML dokumentu formován do <style> tagů.

```

1  <style>
2    .grid_cover {
3      display: grid;
4      grid-template-rows: auto 1fr;
5    }
6    :global(.dark) .fading {
7      background-color: var(--dark-light_grey);
8    }
9  </style>
10

```

Kód 3.2: Ukázka Svelte CSS kódu

3. Obsahová část - vše co se nenachází v jednom z těchto tagů je HTML reprezentace dané stránky, nejedná se však o čisté HTML ale jeho obohacenou verzi. Jako v podobných frameworcích je možné přidávat „event listeners“ na jednotlivé elementy, podmínkově zobrazovat, pracovat s asynchronním kódem nebo dokonce „svazovat“ element či hodnotu elementu s proměnou ve scriptové části.

```
1  {#await data}
2    <div class="@container h-full">
3      <div class="flex w-full py-1 scroller flex-nowrap">
4        {#each Array(countOfItems).fill('') as _}
5          <div
6            class="min-w-[calc(100%/var(--items-count))] h-full relative aspect-[4/5]"
7            >
8              <!-- Zde se nachází další kód -->
9              </div>
10            {/each}
11            <span class="loading loading-infinity loading-lg" />
12          </div>
13        </div>
14        {:then awaitedData}
15        <div class="@container h-full">
16          <div
17            bind:this={scrollerDiv}
18            class="flex w-full h-full py-1 scroller flex-nowrap"
19            style="--translate-x: 0%;"
20            >
21            {#each awaitedData as item}
22              <div
23                class="min-w-[calc(100%/var(--items-count))] relative aspect-[4/5]"
24                >
25                <CardAlternative
26                  class="mx-auto"
27                  navigationLink={'/tests/' + item.id}
28                  type={item.type}
29                  data={{
30                    ...item
31                  }}
32                  />
33                </div>
34              {/each}
35            </div>
36          </div>
37        {/await}
```

Kód 3.3: Ukázka Svelte kódu

3.4.2 TypeScript

TypeScript se dá považovat jako nadstavba JavaScriptu, poskytuje ale jednu výraznou výhodu - typy, díky nim je možno mnohem snadněji dohledávat chyby, vracet se k dříve napsanému kódu a celkově mnohem zlepší „developer experience“ při vytváření aplikace. Sám o sobě pomůže s otypováním jednotlivých částí kódu, neporadí si však například s API endpointy nebo databázovými dotazy, které se poté musí otypovat ručně, což je ale velice špatný způsob. Proto se v tomto projektu využívají další knihovny jako Prisma, tRPC a Zod.

V Effiu jsem se pro TypeScript rozhodl z důvodů v sekci 1.2, zkráceně šlo ale hlavně o možnost efektivně psát kód, který bude obsahovat méně produkčních chyb, způsob jak se lépe vracet k dřívějšímu kódu, také rychlosť a jistota psaní, díky automatickému doplňování IDE, v mé případě Visual Studio Codem.

Ukázka kódu zobrazuje vytváření různých typů, které poté v aplikaci využívám.

```
1 // Carousel.svelte
2 export type IdCardAlternativeProps = CardAlternativeProps & {
3   id: string;
4   type: TestType;
5 };
6
7 export type CarouselItemInput =
8 | IdCardAlternativeProps[]
9 | Promise<IdCardAlternativeProps[]>;
10
11 // customUtilities.d.ts
12
13 // Ručně vytvořený generic, který mi umožňuje zkombinovat funkcionality Partial a Pick
14 // genericu.
15 type PartialPick<T extends Record<unknown, unknown>, K extends keyof T> = {
16   [Key in Exclude<keyof T, K>]: T[Key];
17 } & {
18   [Key in K]?: T[Key];
19 }
```

Kód 3.4: Ukázka TypeScriptového typu

3.4.3 Tailwind CSS

Tailwind CSS je „CSS utility library“, to znamená, že narozdíl od frameworků jako je třeba Bootstrap nebo Material UI neposkytuje celé předpřipravené komponenty ale připravené CSS classy, které se aplikují na HTML elementy, jeho výhodou je naprostá kontrola nad chováním stylů, které se aplikují, přehlednost a rychlosť se kterou se dá styly vytvářet.

Pro Effia jsem se rozhodl využívat Tailwind hlavně díky jeho flexibilitě a rychlosti použití, pro nějaké komponenty jsem využil Daisy UI, což je komponentová knihovna, která využívá pouze CSS, a slouží jako plugin pro Tailwind.

Tento kód zobrazuje využití Tailwind class na určitý element, pro ukázku jsem záměrně vybral rozsáhlejší, pro většinu případů si ale vystačím s jedním až dvěma řádky class.

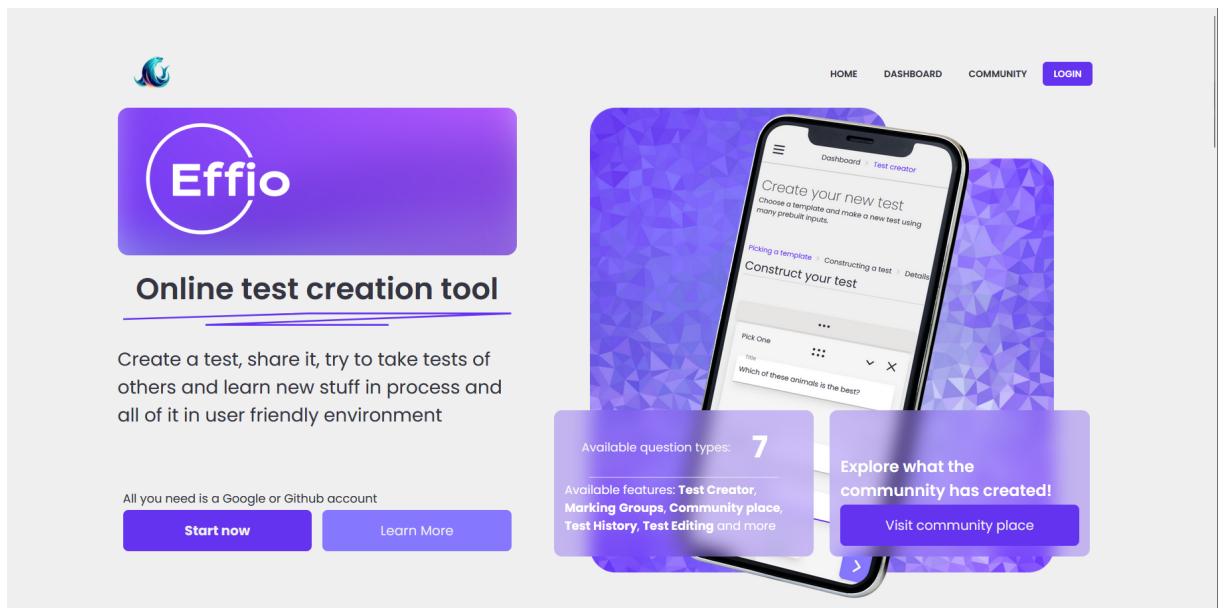
```
1 <button
2   type="button"
3   on:click={starTest}
4   disabled={canStarTest === false || isSubmittingStar === true}
5   class={`absolute flex items-center z-[2] gap-1 px-2 py-1 rounded-lg right-1 top-1 bg
6     -light_white dark:bg-dark_grey shadow-md duration-100 ${'
7       isStarred ? 'bg-yellow-100 dark:bg-yellow-700' : ''
8     } hover:bg-light_secondary dark:hover:bg-dark_secondary disabled:bg-
9     light_grey_dark dark:disabled:bg-slate-600
10    text-light_text_black dark:text-dark_text_white hover:text-light_whiter disabled:
11      hover:text-light_text_black
12      dark:disabled:hover:text-dark_text_white`}
13    >
14  </button>
```

Kód 3.5: Ukázka Tailwind kódu

4 FUNKCE APLIKACE

4.1 DOMOVSKÁ STRÁNKA

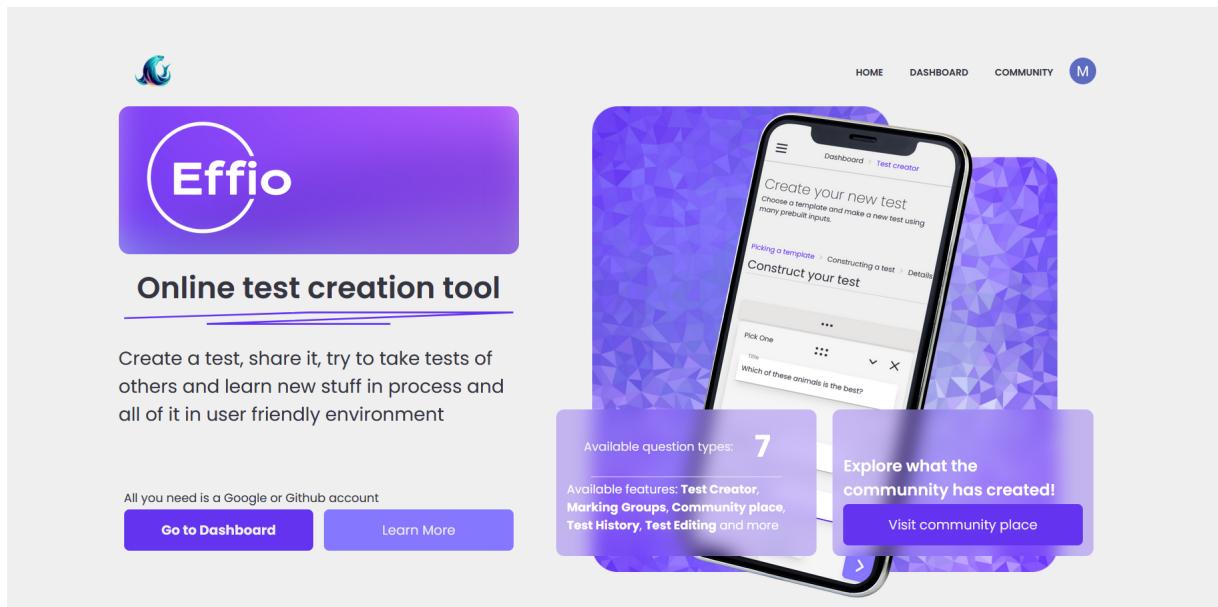
Domovská stránka slouží jako místo pro seznámení návštěvníka s výhodami Effia, rychlá navigace mezi jednotlivými stránkami ale také jako místo nejpečlivěji vytvářeného designu a efektů aby na uživateli zanechala dojem kvalitní aplikace.



Obrázek 4.1: Domovská stránka Effia pro nepřihlášeného uživatele.

4.2 PŘIHLÁŠENÍ

Po kliknutí na tlačítko login se dostaneme na přihlašovací obrazovku kde si můžeme vybrat mezi přihlášením přes Google a GitHub účet, po úspěšném přihlášení se uživatel dostane do dashboardu, což je společně s vytvářením testů, historií a skupinami dostupné pouze pro přihlášené uživatele, pokus načtení stránky pokud je uživatel nepřihlášen vyústí v přesměrování na login page.



Obrázek 4.2: Domovská stránka Effia pro přihlášeného uživatele.

4.3 TESTY A JEJICH VLASTNOSTI

Jednou z esenciálních funkcionalit Effia je možnost vytvořit test, k tomuto existuje mnoho různých postupů, nad danou problematikou jsem se zprvu zamýšlel a až poté napsal funkční nástroj pro jejich vytváření ale to stejně nakonec nezabránilo následné nutnosti přepsat téměř celou funkctionalitu při vytváření testu.

4.3.1 Tvorba testu

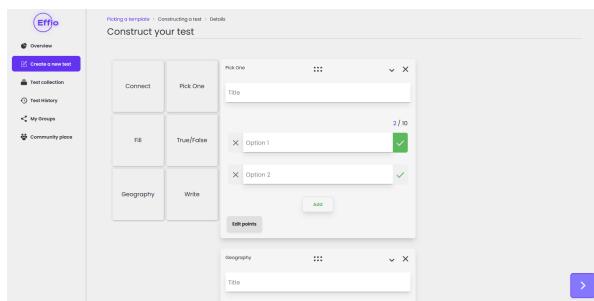
Jako první si uživatel zvolí mezi kvízovým a programovacím testem, u obou si poté vybere šablonu.

- Kvízový - po výběru šablony, kde si uživatel může zvolit i import z GIFT formátu, se uživatel dostane do tvorby testu samotného, vybírat si aktuálně může z 6 typů otázek:

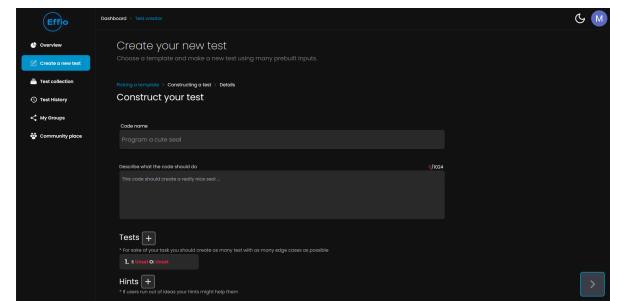
Pick One, True/False, Connect, Write, Fill a Geography, otázkám lze svévolně měnit pořadí, přidávat komentáře k odpovědím a upravovat počet získaných bodů.

- Programovací - po výběru šablony se uživatel dostane do tvorby testu programovacího, kde ho pojmenuje problém, popíše co má uživatel řešit, nadefinuje kontrolní vstupy a očekávané výstupy, poté může zanechat nápovery

Po dokončení těchto úprav se uživatel dostane do konečných úprav testu, což činí jméno, popisek a obrázek testu, volitelné zařazení do skupin, tagy, rozhodne se jestli využít známkovací systém, který si může sám upravit, zvolí si zdali náhodně třídit otázky a následně tvorbu ukončí a rozhodne se zdali test uložit jako návrh nebo ho publikovat.



Obrázek 4.3: Kvízový test



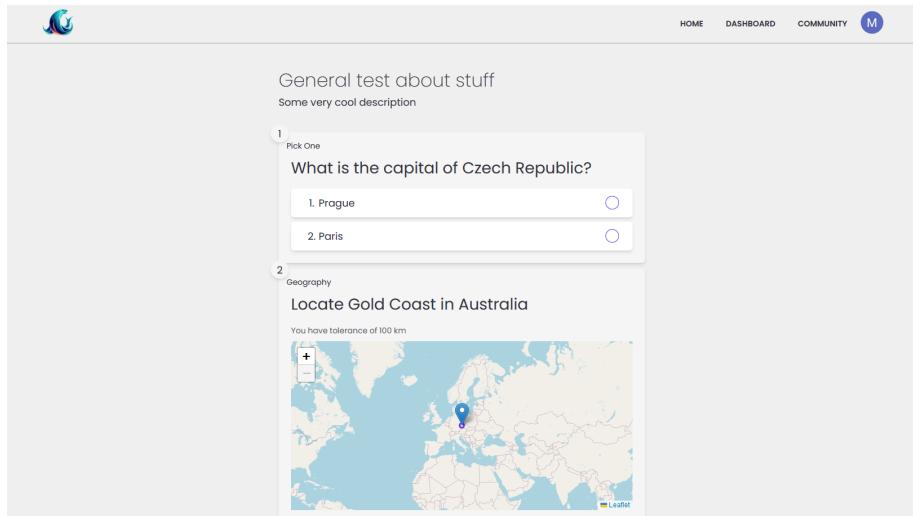
Obrázek 4.4: Programovací test

4.4 VYPLŇOVÁNÍ TESTU

4.4.1 Vyplňování kvízu

Vytvořený test si poté může kdokoliv s přístupem k němu vyplnit (testy jsou základně dostupné pro všechny, po úpravě mohou být zveřejněny pouze pro členy skupin).

Otázky se náhodně seřadí a uživatel je vyplňuje, zamíchané jsou také odpovědi určitých typů otázek. Po vyplnění všech uživatel test odevzdá, zkонтroluje se a vrátí mu správné odpovědi, počet bodů, známku co získal a pokud je uživatel přihlášený tak se záznam a vyplnění uloží do databáze, uživatel si ho poté může zpětně zobrazit v sekci *Test history*.



Obrázek 4.5: Obrázek kvízu.

4.4.2 Plnění programovacího testu

Uživatel dostane popis toho co by měl kód umět, sadu testů, které mají otestovat funkcionality kódu a popřípadě nějaké návodů. Programovací test obsahuje vlastní editor do kterého uživatel píše, pro kontrolu testu můžeme použít tlačítko „Run“ a pokud testy prochází tak je test možné řešení odevzdat.

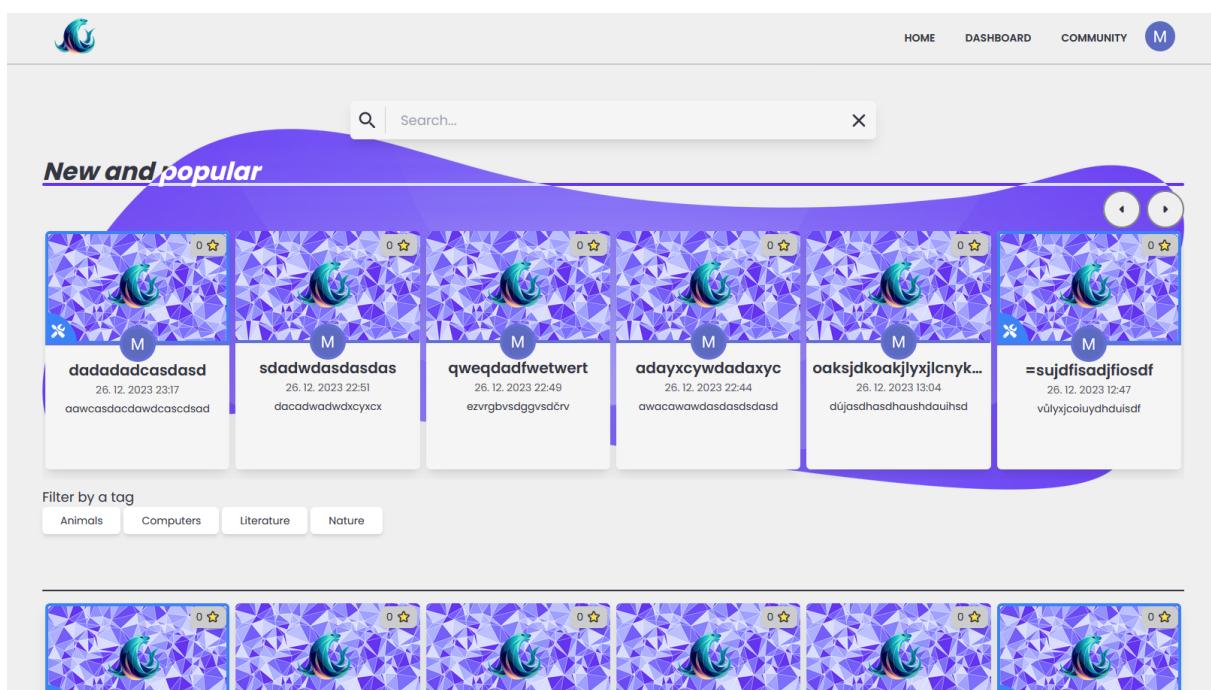
Tests	Input	Output	Logs
1. 16	16	1. Input: 16 Output: Result: Expected Output: "even"	
2. 11	11		
3. 0	0		

Obrázek 4.6: Obrázek programovací úlohy.

4.5 ZOBRAZOVÁNÍ TESTŮ

4.5.1 Komunitní místo

Na této stránce může uživatel najít nové a populární testy včetně všech testů, které existují. Testy jsou zobrazovány postupně podle techniky „infinite scrolling“, na což využívám JavaScript API Intersection Observer, abych zjistil, kdy uživatel dosáhl posledního prvku a vyžádal si tak další testy. Implementované je vyhledávací pole, které filtrouje zobrazované testy, další možností je filtrace pomocí tagů. Vizuálně jsou také rozlišeny kvízy od programovacích testů. Testy se dají ohodnotit hvězdičkou, jejich aplikování využívá principu „optimistic update“, to znamená, že po přidání hvězdičky ji uživatel okamžitě vidí přidanou, zatímco se ověřuje jeho oprávnění a vytváří záznam hvězdičky v databázi. V případě neúspěchu se poté uživateli sama hvězdička opět odebere.



Obrázek 4.7: Komunitní místo.

4.5.2 Kolekce testů

Zde si uživatel může zobrazit jím vytvořené testy, aplikovaná je stejná funkcionality vyhledávacího pole a „infinite scrollingu“. Každý test má ale také další možnosti, a to úpravu, export a smazání.

- Úprava - uživatel se přesune na stránku úprav, tam může celý test přepracovat.

- Export - vytvoří z testu textový soubor ve formátu GIFT se všemi otázkami daného testu, které jsou podporovány Moodlem
- Delete - smazání testu z databáze

4.6 SKUPINY

Každý přihlášený uživatel si může vytvořit vlastní skupinu, do které se můžou pomocí generovaného kódu připojit ostatní uživatelé. Skupina obsahuje kanály, ve kterých je možné psát textové zprávy, taky zde můžeme najít přidané testy, vlastník si poté může procházet grafy výsledku členů skupiny.

5 ZHODNOCENÍ PRÁCE

5.1 SPLNĚNÉ A NESPLNĚNÉ CÍLE

Cíle byly rozdělené jak do rozsahu aplikace tak do kvality implementace jednotlivých prvků, co se týče rozsahu tak ten jsem v určitých místech významně předčil, ne všechny body původních cílů jsou ale aktuálně plně dosaženy.

Hlavním cílem bylo vytvořit rychlou, cloudovou aplikaci pro vytváření a sdílení testů využívající moderních „techstack“, v tomto ohledu jsem cíl kompletně splnil, aplikace je v těchto bodech plně funkční a můj původní plán využití technologií se během vývoje ještě významně rozrostl.

Za úspěch považuji také grafickou část aplikace, která za mě tvoří minimalistický moderní vzhled.

Hlavní neúspěch nebo spíše nedodělanost vidím ve skupinách a přizpůsobení možností testů pro ně, původně jsem zamýšlel vytvořit skupiny jako místo pro sdílení materiálů s více zajímavými možnostmi pro vlastníka, aby sloužili jako funkce vhodná pro výuku. Cíle nejsou zdaleka nerealistické, ale těchto cílů jsem nebyl schopen dosáhnou z nedostatku času, jako vylepšení by ale za mně bylo velice přínosné.

Další částečný neúspěch vidím v programovacích testech, ty se i přes snahu vyladit nepovedly dostat do stavu kdy by byly pro uživatele nezávadné, v aktuálním stavu je naprosto možné přetížit vlastní prohlížeč napsáním např. `while(true)`, JavaScript je „single threaded“ jazyk a proto není možné proces ukončit, protože neskončí ten předchozí (právě zmínění while loop). Řešení nabízí „Web workers“, což je možnost jak JavaScript spouštět na více vláknech, zdálo se tedy, že řešení je na světě. Po bližším přezkoumání jsem ale zjistil, že nejsem schopen web workery v SvelteKit aplikaci v produkci načítat žádným způsobem, daním web workeru do stejné složky dojde k neexistující referenci (web worker se zbundluje se zbytkem kódu a jeho funkcionalita se rozbití), řešení je tedy dát worker do public složky, která se odděluje od zbytku, zde ale dojde k erroru MIME.

ZÁVĚR

Cílem projektu bylo vytvořit webovou aplikaci pro vytváření a vyplňování testů. Aplikace je postavená na frameworku SvelteKit, psaný v jazyce TypeScript. Přihlašování stojí na knihovně Auth.js, MySQL databáze je hostovaná službou Planetscale, jako ORM je použita Prisma. Frontend řeší framework Svelte s CSS utility knihovnou Tailwind, pro validaci využívá Zod.

Základem aplikace je generátor testů, kde uživatel využívá předpřipravené typy otázek. Testy se následně dají vyplnit v rámci komunity nebo vlastníkovi kolekce. Nepřihlášený uživatel může testy pouze vyplňovat, přihlásit se uživatel může pomocí Google nebo GitHub účtu. Uživatelé také mají k dispozici skupiny kde je chat a také možnosti sdílet testy, test historii k zobrazení dříve vyplněných testů a přehled nedávné aktivity v podobě grafů. Využité technologie činí aplikaci lehce škálovatelnou a velice výkonnou. Aplikace je téměř zcela funkční a použitelná na všech zařízeních díky responzivnímu designu.

Aplikace je zálohovaná na GitHubu na adrese <https://github.com/matej-kotrba/effio>. Je také volně přístupná na adrese <https://effio.vercel.app/>

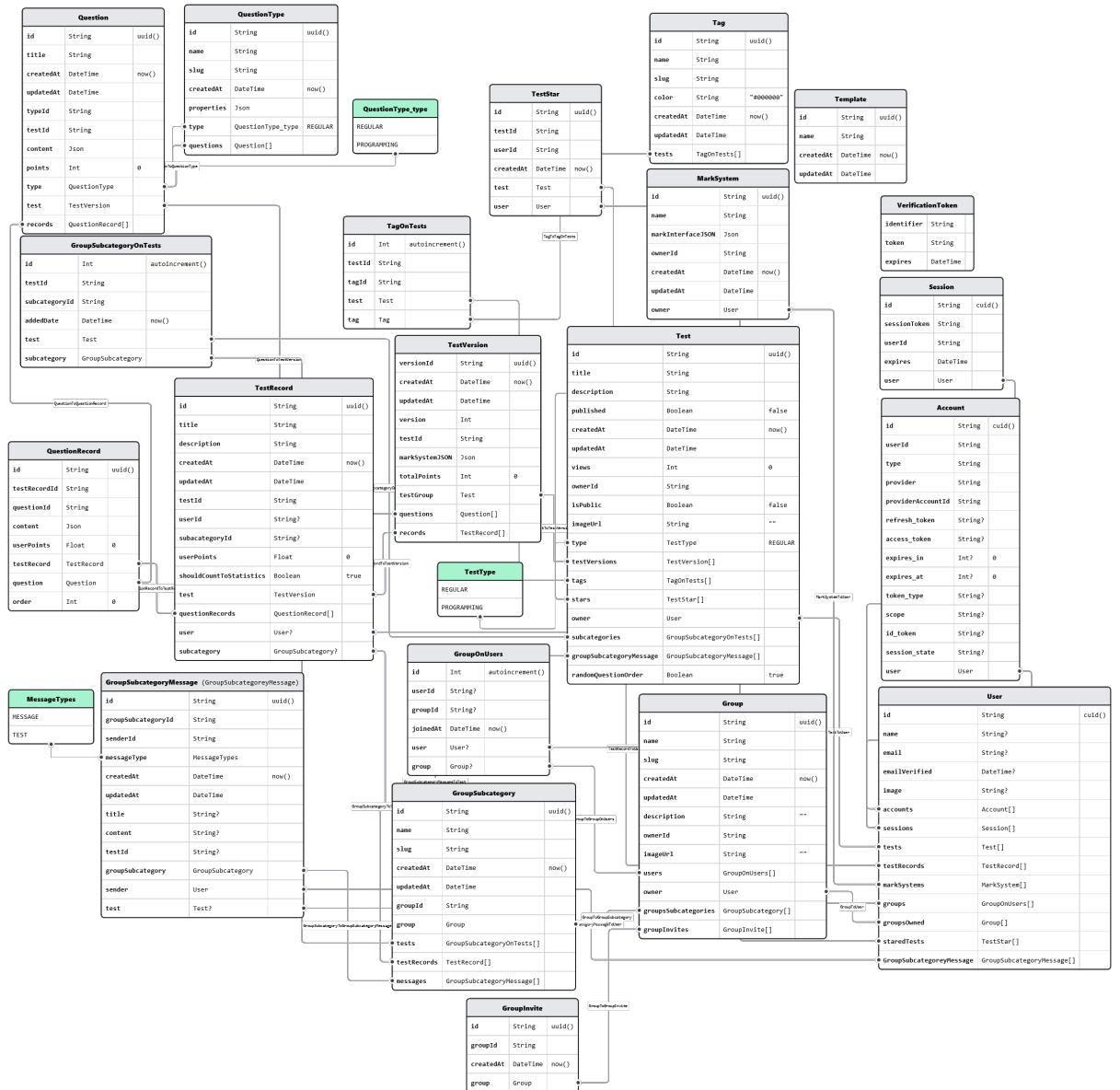
LITERATURA

- [1] Svelte [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://svelte.dev/>
- [2] SvelteKit [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://kit.svelte.dev/>
- [3] tRPC [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://trpc.io/>
- [4] Prisma [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://www.prisma.io/>
- [5] Zod [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://zod.dev/>
- [6] Auth.js [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://authjs.dev/>
- [7] Tailwind CSS [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://tailwindcss.com/>
- [8] tRPC-SvelteKit [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://icflorescu.github.io/trpc-sveltekit/>
- [9] ChatGPT [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://chat.openai.com/>
- [10] Stackoverflow [online]. 2023 [cit. 2023-12-27]. Dostupné z: <https://stackoverflow.com/>
- [11] Joy of Code. Youtube kanál. [Https://www.youtube.com/ \[online\]](https://www.youtube.com/@JoyofCodeDev). 2023 [cit. 2023-12-27]. Dostupné z: <https://www.youtube.com/@JoyofCodeDev>
- [12] Huntabyte. Youtube kanál. [Https://www.youtube.com/ \[online\]](https://www.youtube.com/@Huntabyte). 2023 [cit. 2023-12-27]. Dostupné z: <https://www.youtube.com/@Huntabyte>
- [13] BROWNE, Theo. Youtube kanál. [Https://www.youtube.com/ \[online\]](https://www.youtube.com/@t3dotgg). 2023 [cit. 2023-12-27]. Dostupné z: <https://www.youtube.com/@t3dotgg>
- [14] POWELL, Kevin. Youtube kanál. [Https://www.youtube.com/ \[online\]](https://www.youtube.com/@KevinPowell). 2023 [cit. 2023-12-27]. Dostupné z: <https://www.youtube.com/@KevinPowell>

Seznam obrázků

1.1	Rozdíl mezi serverless a edge.	4
1.2	Možnost přepnutí dané cesty ze Serverless na Edge. [2]	4
2.1	Jednoduchý přehled backendové části Effia.	6
3.1	Prvotní návrh domovské stránky.	15
3.2	Domovská stránka na mobilním zařízení	16
3.3	Generátor testů na mobilním zařízení	16
4.1	Domovská stránka Effia pro nepřihlášeného uživatele.	21
4.2	Domovská stránka Effia pro přihlášeného uživatele.	22
4.3	Kvízový test	23
4.4	Programovací test	23
4.5	Obrázek kvízu.	24
4.6	Obrázek programovací úlohy.	24
4.7	Komunitní místo.	25
A.1	Databázový model Effia. Vytvořeno pomocí Prismaliser	31

PŘÍLOHA A DATABÁZOVÝ MODEL



Obrázek A.1: Databázový model Effia. Vytvořeno pomocí Prismaliser