VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

MONITORING PRŮBĚHU END-TO-END TESTŮ

END-TO-END TESTS PROGRESS MONITORING

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Maksim Syrvachev

AUTHOR

VEDOUCÍ PRÁCE — Ing. Patrik Kaura

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav:	Ústav automatizace a informatiky
Student:	**Maksim Syrvachev**
Studijní program:	Strojírenství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. Patrik Kaura**
Akademický rok:	2022/23

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Monitoring průběhu end–to–end testů

Stručná charakteristika problematiky úkolu:

Návrh a implementace webového rozhraní pro sledování průběhu end–to–end testů, které testuje stávající rozhraní reklamního systému Sklik.cz. Cílem práce je implementace rozhraní pro přepínání testovacích prostředí prostřednictvím stávající komponenty e2e manager. Dále bude práce řešit problematiku správy testovacích účtů.

Cíle bakalářské práce:

1) Analýza aktuálně dostupných řešení pro monitoring end–to–end testů.
2) Analýza stávajícího řešení běhu end–to–end testů.
3) Návrh a implementace webového rozhraní pro monitoring.

Seznam doporučené literatury:

PORCELLO, Eve. Learning React: Modern patterns for Developing React Apps. 2nd. Sebastopol (Kalifornie): O'Reilly UK, 2020. ISBN 1492051721.

MWAURA, Wawer. End-to-End Web Testing with Cypress. Birmingham, UK: Packt Publishing, 2021. ISBN 9781839213854.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2022/23

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jiří Hlinka, Ph.D. děkan fakulty

ABSTRAKT

Tato bakalářská práce se zaměřuje na vývoj webového rozhraní pro sledování průběhu end-to-end testů. První část práce se věnuje technologiím a programovacím jazykům, které se používají při vývoji moderních webových aplikací, a také způsobům testování těchto aplikací. Následující část popisuje architekturu a princip fungování webové aplikace E2E manager, která slouží k monitorování průběhu end-to-end testů, spolu s technologiemi použitými při jejím vývoji. Poslední část práce se věnuje porovnání webové aplikace E2E manager s dalšími dostupnými řešeními pro sledování E2E testů.

ABSTRACT

This bachelor thesis focuses on the development of a web interface for end-to-end test tracking. The first part of the thesis focuses on the technologies and programming languages that are used in the development of modern web applications, as well as the ways in which these applications are tested. The following section describes the architecture and working principle of the E2E manager web application used to monitor the progress of end-to-end tests, together with the technologies used in its development. The last part of the thesis compares the E2E manager web application with other available solutions for monitoring E2E tests.

KLÍČOVÁ SLOVA

End-to-end testování, webová aplikace, průběh end-to-end testů, sledování testů, JavaScript, TypeScript, React.js.

KEYWORDS

End-to-end testing, web application, end-to-end test flow, test tracking, JavaScript, TypeScript, React.js.

ÚSTAV AUTOMATIZACE A INFORMATIKY

2023

BIBLIOGRAFICKÁ CITACE

SYRVACHEV, Maksim. *Monitoring průběhu end-to-end testů.* Brno, 2023. Dostupné také z: https://www.vut.cz/studenti/zav-prace/detail/149208. Bakalářská práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Vedoucí práce Patrik Kaura.

PODĚKOVÁNÍ

Chtěl bych poděkovat vedoucímu práce Ing. Patriku Kaurovi za cenné rady a odborné vedení práce. Dále bych chtěl poděkovat společnosti Seznam.cz, pro kterou byla tato bakalářská práce napsána, a Ondrovy Zdychovi za podporu při tvorbě práce.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, vypracoval jsem ji samostatně pod vedením vedoucího práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následku porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestně právních důsledků.

V Brně dne 20. 5. 2023 .....

Student Učený

OBSAH

1 ÚVOD

V dnešním světě hrají webové aplikace a webové stránky obrovskou roli. Není žádným tajemstvím, že desktopové aplikace ustupují do pozadí a webové aplikace si získávají stále větší oblibu z několika důvodů. Webové aplikace nevyžadují instalaci objemného softwaru na počítač zákazníka a ke správnému fungování je potřeba pouze prohlížeč a přístup k internetu. Vývoj webové aplikace nevyžaduje speciální konfiguraci a správu, jejich správci jsou vývojáři. Webové aplikace jsou multiplatformní, což znamená že je lze otevřít na jakémkoliv zařízení a zároveň se automaticky aktualizují. [1]

Historie vývoje webových aplikací sahá až do 90. let. Od této doby začal jejich počet narůstat. Stránky však byly statické a zobrazovaly pouze omezené množství informací. Postupem času se však webové technologie vyvíjely a umožňovaly vytváření stále sofistikovanějších a interaktivnějších aplikací. Dnes jsou webové aplikace dynamické, komplexní a často integrované s různými systémy a službami. Na rozdíl od statických webových stránek, které zobrazují všem návštěvníkům stejný obsah ve stejném formátu, dynamické webové stránky zobrazují různým návštěvníkům různé informace. [2]

S narůstající složitostí webových aplikací se zvyšuje jejich potřeba kvalitního testování. Testování webových aplikací hraje klíčovou roli při zajišťování funkčnosti, výkonnosti a uživatelského zážitku. Testování hlavně slouží pro ověření, zda aplikace funguje správně a splňuje stanovené požadavky.

V této bakalářské práci je kladen důraz na end-to-end testování – specifický druh testování, který simuluje reálné interakce uživatele s webovou aplikací a testuje aplikaci od začátku až do konce, včetně všech jejích funkcí.

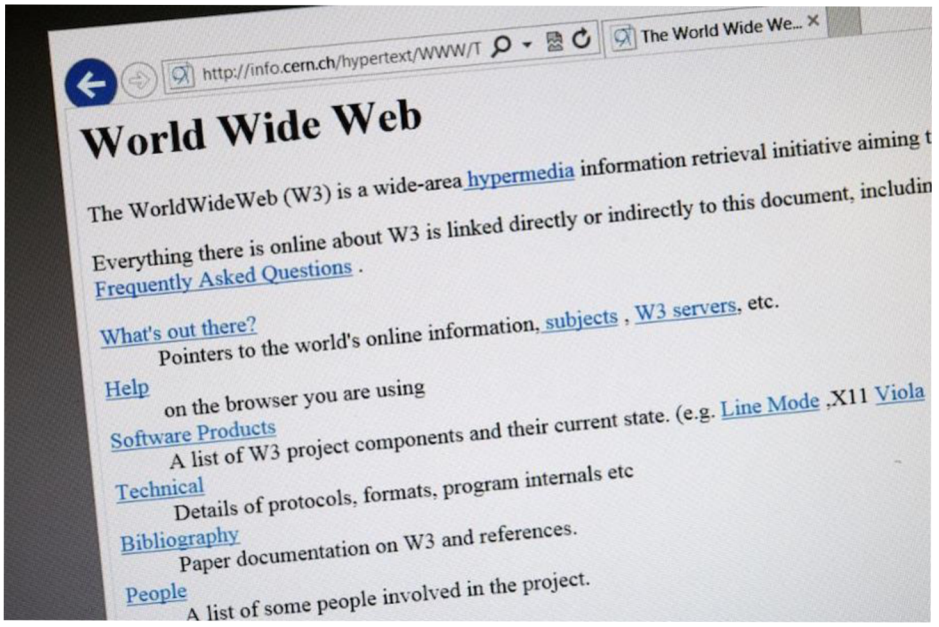
Cílem této práce je navrhnout a implementovat určité části webového rozhraní pro sledování průběhu end-to-end testů, které testuje stávající rozhraní reklamního systému Sklik.cz. Webová aplikace E2E manager bude vytvořena v moderním frameworku React.js s využitím propojení koncových bodů API a načítání dat. V kapitole 5 je popsáno srovnání vytvořené webové aplikace E2E manager s ostatními existujícími řešeními sledování end-to-end testování testů.

2 ÚVOD DO VÝVOJE WEBOVÝCH APLIKACÍ

Tato kapitola pojednává o webových aplikacích a technologiích použitých pro jejich vývoj. V první podkapitole je stručně zmíněna historie rozvoje webových aplikací. V následujících podkapitolách jsou popsány programovací jazyky a nástroje používané při tvorbě moderních webových aplikací. Poslední podkapitola se zaměřuje na testování webových aplikací.

2.1 Historie webových aplikací

Tvůrcem první webové stránky na světě byl Angličan Tim Berners-Lee. První stránka byla vytvořena v roce 1991 a byla dostupná na adrese info.cern.ch, viz obr. 1. Na tomto zdroji, nazvaném World Wide Web, britský vědec zveřejnil data o nejnovější technologii přenosu informací. Tento systém zahrnoval implementaci takových technických prostředků, jako je protokol přenosu dat HTTP, speciální jazyk HTML používaný k označování textů a systém webových adres ve formě URI. Principy, které Tim Berners-Lee implementoval na svůj webový zdroj se staly základem pro provoz a instalaci prohlížečů a serverů. [4]



Obr. 1: První webová stránka na světě [3]

V roce 1994 nastal ve vývoji webu obrovský skok. Objevily se tabulky, které umožnily kvalitativně uspořádat informace, zarovnat text do sloupců a výrazně zkvalitnit uživatelské použití. Vývojáři webových stránek zpracovávali HTML převážně v primitivních textových editorech. [3; 5]

Rok 1995 je rokem, kdy se na webových stránkách začínají objevovat animace a barvy, na úkor jejich čitelnosti a použitelnosti. Pro přidání dynamičnosti stránek bylo

HTML doplněno o rozšíření v podobě programovacího jazyka Javascript, který je dnes nejpopulárnější jazyk pro vývoj webových aplikací. Jeho prostřednictvím mají vývojáři možnost přidávat různé interaktivní prvky a další funkcionality reagující na akce uživatele na stránky pomocí *EventListener*. V tomto roce se taky objevil jazyk PHP, který slouží k psaní aplikací na straně serveru. O rok později vznikl značkovací jazyk CSS, který slouží ke stylizaci HTML dokumentu. Od tohoto roku se tyto nástroje využívají k vytváření vizuálně přívětivých webových stránek se schopnosti uživatelské interakce. Toto lze vidět na webové stránce společnosti Apple a.s v roce 1996, viz obr. 2. [5]

Obr. 2: Webová stránka apple.com v roce 1996 [6]

2.2 JavaScript

JavaScript je multiplatformní, objektově orientovaný, událostmi řízený skriptovací jazyk, jehož autorem je Brendan Eich. Lze ho definovat jako vyšší skriptovací jazyk, který se v posledni dobe používá jak ke psaní frontendových tak i backendových částí webových aplikaci. [7] Vyšší skriptovací jazyk je jazyk se silnou abstrakcí od detailů počítače, který používá prvky přirozeného jazyka pro automatizaci významné oblasti výpočetních systémů a zjednodušení vývoje programu. Javascript je podporován všemi používanými prohlížeči. Na straně frontendu jazyk slouží k přidání dynamičnosti značkování stránek (HTML) a uživatelské funkčnosti (CMS) stránek. Každá akce, vyvolaná uživatelem spustí určitý děj – zobrazování pravidelně aktualizovaného obsahu, animace 2D/3D grafiky, spuštění video souboru, zobrazení interaktivní mapy atd. Jazyk JavaScript se také používá pro backend, prostřednictvím platformy Node.js. Jedná se o rozhraní, které umožňuje spouštět JavaScript kód mimo webový prohlížeč, díky tomu že je rozhraní

postaveno na V8 Javascript engine. Již 6 let je Javascript podle ankety na platformě GitHub nejoblíbenějším jazykem mezi vývojáři. [8]

V následující části bude popsáno, jak Javascript funguje na straně klienta. Jak již bylo zmíněno JavaScript je skriptovací jazyk. Skripty jsou sada instrukcí, které se provádějí při načítání stránky. Tyto skripty mohou přistupovat k aplikacím, protože poskytují „host“ objekty za běhu. Na klientské straně je prostředím zprostředkovávající běh aplikace webový prohlížeč, který umožňuje manipulaci s objekty jako jsou okna a dokumenty HTML. „Host“ objekty nejsou součástí jádra JavaScriptu, jedná se o API webového rozhraní, tedy objekty poskytované prohlížečem. Díky implementaci JavaScript enginu do prohlížeče, může prohlížeč samostatně interpretovat JavaScript kód. Veškeré akce uživatele v okně prohlížeče vytvářejí události (tzv. events), které lze určitým způsobem zpracovávat v jazyce JavaScript. Takto vypadá standardni algoritmus chování webove strance:

1. Uživatel provede nějakou akci na stránce, např. stiskne tlačítko.
2. Prohlížeč definuje novou událost.
3. Aktivuje se JavaScript kód (pokud je pro tuto akci přidán eventListener).
4. Na stránce se projeví žádaná změna.

Shrnutí využití JavaScriptu:

- Manipulace s obsahem webových stránek
- Zpracovávání událostí, vyvolaných uživatelem
- Manipulace s daty
- Asynchronní programování
- Manipulace s dokumentem (DOM)
- Komunikace se serverem

2.3 ECMAScript

Samotný Javascript podléhá standardizaci, která je sdružená pod názvem ECMAScript. ECMAScript (European Computer Manufacturers Association Script) je JavaScriptový standard určený k zajištění interoperability webových stránek napříč různými prohlížeči. Je standardizován společností Ecma International v dokumentu ECMA-262. ECMAScript se běžně používá pro skriptování na straně klienta na webu a stále častěji se používá pro psaní aplikací a služeb na straně serveru pomocí Node.js a dalších běhových prostředí. ECMAScript obsahuje pravidla, informace a pokyny, které musí skriptovací jazyk dodržovat, aby byl považován za kompatibilní s ECMAScriptem. JavaScript lze tedy definovat jako univerzální skriptovací jazyk vyhovující specifikaci ECMAScript.

Zjednodušeně, specifikace ECMAScript stanovuje pravidla pro tvorbu skriptovacího jazyka, a dokumentace jazyka JavaScript říká, jak skriptovací jazyk použít. [9]

Když se mluví o podpoře prohlížečů, obvykle se mluví o „kompatibilitě ECMAScript“ spíše než o „kompatibilitě s jazykem JavaScript“, přestože JavaScriptový engine spouští JavaScriptový kód. ECMAScript je tedy specifikace toho, jak může skriptovací jazyk vypadat. Vznik nové verze ECMAScript automaticky neznamená, že všechny JavaScriptové enginy v prohlížečích budou mít nové funkce, ty se budou přidávat postupně. Proto vývojáři většinou kladou otázku typu „Jakou verzi nebo funkce ECMAScriptu tento prohlížeč podporuje?“. [9]

Každý rok ECMAScript vydává novou edici specifikace obsahující všechny nejnovější funkce a změny (tzv. features), které byly schváleny při jejím vytvoření. Proces provádění změn specifikace ECMAScript provádí TC39. Tento proces se skládá z pěti fází (tzv. stages), začínající nultou fází. Jakýkoli návrh na změnu nebo přidání feature do specifikace bez výjimky prochází těmito fázemi a každý přechod z jedné fáze do druhé musí být schválen komisí. Fáze procesu [10]:

- Stage 0 - Strawman (Plánování a diskuze návrhu).
- Stage 1 - Proposal (V této fázi je vytvořen a posouzen formální návrh, který popisuje konkrétní problém a řešení).
- Stage 2 - Draft (Tato fáze je počátečním návrhem edice ve specifikaci, formulovaným jazykem ECMAScript).
- Stage 3 - Candidate (Návrh je definován jako zcela konečný, pokud se dále nepracuje se specifikací ani se nepracuje s externí zpětnou vazbou).
- Stage 4 - Finished (Návrh je připraven k přidání do nejnovějšího návrhu specifikace se všemi změnami).

2.4 TypeScript

Většina praktické části bakalářské práce byla napsána v jazyce TypeScript, který je „syntaktickou nadmnožinou“ JavaScriptu, která přidává statické typování. TypeScript dělá kód přehlednějším a spolehlivějším a je kompilován do JavaScriptu. Tento jazyk používají vývojáři frontendu i backendu. V následujícím odstavci jsou uvedeny hlavní rozdíly mezi TypeScriptem a JavaScriptem [11]:

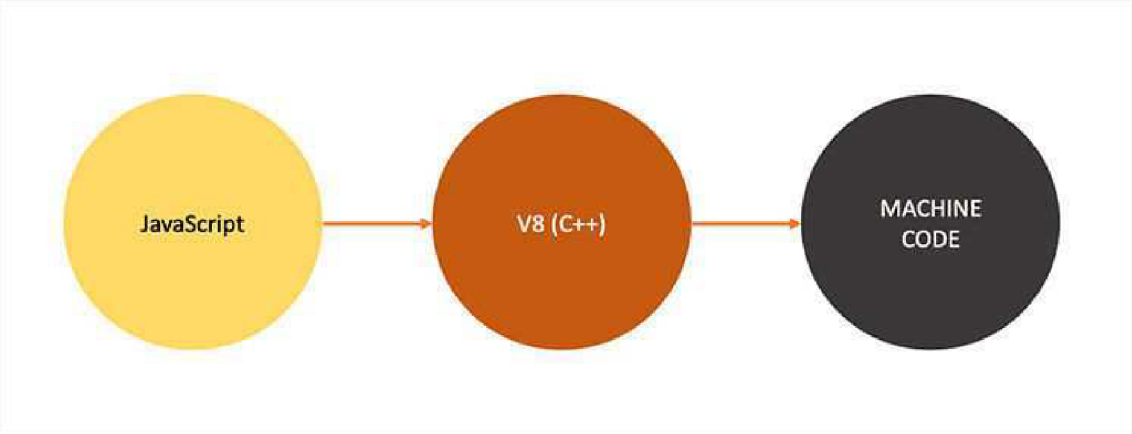
- Prvním rozdílem, jak již bylo zmíněno je typový systém. Data v Javascriptu jsou uložena v proměnných. JavaScript má volné, dynamické typování (např. proměnné lze přiřadit nejprve textovou hodnotu a poté číselnou hodnotu). TypeScript přidává do jazyka typový systém. Při vytváření je každé proměnné přiřazen konkrétní datový typ – primitivní nebo vytvořený vývojářem. To znamená že po přiřazení datového typu proměnné ji nelze

změnit. Kromě primitivních typu, které v JavaScriptu již existují – undefined, null, boolean, number, bigint, string a symbol, má TypeScript taky typy any (lze přiřadit libovolnou hodnotu), never (je často používán jako návratový typ pro funkce, které vyvolávají výjimky nebo nekonečné smyčky) a void (typ pro funkce, které nic nevrací).

- Zlepšené objektově orientované programování: Oba JavaScript i TypeScript mají podporu pro objektově orientované programování: třídy, dědičnost, objekty. TypeScript však rozšiřuje funkcionalitu OPP, například přidáváním rozhraní (tzv. interface). Obsahuje rovněž modifikátory přístupu:
  Public – prvky s tímto modifikátorem jsou přístupné v jakémkoliv místě bez omezení Tento modifikátor je nastaven jako výchozí.
  Private – prvky s tímto modifikátorem jsou přístupné pouze v té třídě, ve které jsou definovány.
  Protected – prvky s tímto modifikátorem jsou kromě třídy, ve které byly definovány dostupné i v odvozených třídách.
- Dalším a možná nejdůležitějším rozdílem mezi JavaScriptem a TypeScriptem je to, že kód napsaný v TypeScriptu nelze spustit v prohlížeči bez transpilace, proto TypeScript není nezávislý jazyk, ale jazykový doplněk JavaScriptu. Transpilace je proces kdy program převede kód napsaný v TypeScriptu na nativní JavaScript. Pro napsání programu v TypeScriptu používáme soubor s příponou .ts nebo. tsx, který se po transpilaci převede na .js soubor (pro čtení prohlížečem). TypeScript navíc na rozdíl od JavaScriptu vyžaduje instalaci balíků třetích stran, protože transpilace vyžaduje modul tsc.

2.5 Node.js

Následující podkapitola pojednává o využití Node.js pro provádění JavaScriptu na straně serveru. Node.js je open source framework, který je postaven na engine V8 vyvinutém společností Google. [12] Tento framework umožňuje psát serverový kód pro webové aplikace. Dříve bylo možné spouštět JavaScriptový kód pouze v prohlížeči. S příchodem Node.js je však možné spouštět JavaScript na serverech jako samostatné aplikace. Engine V8 je rovněž open source a je napsán v jazyce C++. Tento engine zpracovává JavaScriptový kód a překládá ho do rychlejšího strojového kódu, který lze spouštět na počítači bez nutnosti interpretace, viz obr. 3.



Obr. 3: Princip spouštění JavaScript kódu [12]

Vlastnosti Node.js [13]:

- Node.js používá asynchronní programování.
- Node.js je schopný provádět CRUD (Create, Read, Update a Delete) operace, což jsou základní čtyři operace pro manipulaci s daty, které webové aplikace vyžadují. Tato funkcionalita není omezena pouze na souborový systém, ale může být aplikována i na vzdálené servery.
- Node.js umožňuje manipulaci s různými databázovými systémy. Jeho ekosystém balíčků, známý jako npm (Node Package Manager), představuje nejrozsáhlejší open source ekosystém knihoven na světě. Obsahuje více než 500 000 open source modulů a knihoven, které mohou být využity pro různé účely v Node.js aplikacích.
- Soubor package.json představuje centrální bod pro konfiguraci a popis projektu, který umožňuje jednoduchou správu projektových nastavení z jednoho místa. Tento soubor slouží k zaznamenání důležitých metadat o projektu. Zároveň definuje funkční atributy projektu, které jsou využívány npm pro instalaci závislostí (node_modules), spouštění skriptů a určení vstupního bodu do aplikace. Díky tomu existuje možnost centralizovaně spravovat a identifikovat klíčové aspekty aplikace pomocí jednoho souboru.

V následující části je podrobně popsán proces inicializace projektu pomocí Node.js a vytvoření jednoduchého serveru.

Po vytvoření prázdného adresáře je nezbytné provést inicializační proces prostřednictvím balíčku npm specifického pro daný projekt. Tento proces lze spustit zadáním příkazu “npm init” do terminálu ve složce projektu. Tím dojde k vytvoření základního stromu projektu spolu se souborem *package.json*, který slouží k definování skriptů, sledování nainstalovaných modulů a konfiguraci aplikace. Pro instalaci libovolného balíku do projektu stačí provést příkaz “npm i package_name” v terminálu,

přičemž název tohoto balíčku se zobrazí v objektu “dependencies”. Níže uvedený úryvek kódu představuje ukázku souboru package.json:

```
{
  "dependencies": {
    "@koa/router": "^12.0.0",
    "@sklik/remote-applications": "0.207.0",
    "@tsoa/runtime": "^5.0.0",
    "env-var": "^7.3.0",
    "http-errors": "^2.0.0",
  },
  "devDependencies": {
    "@babel/plugin-syntax-decorators": "^7.19.0",
    "@babel/plugin-syntax-typescript": "^7.20.0",
    "eslint-config-prettier": "^8.6.0",
    "eslint-plugin-babel": "^5.3.1",

  },
  "engines": {
    "node": "18.12.1",
    "npm": "8"
  },
  "name": "package_json_example",
  "private": true,
  "scripts": {
    "dev": "npm run clean && tsc --build tsconfig.build.json --watch",
    "prettier:check": "prettier --check .",
    "prettier:write": "prettier --write .",
    "start": "docker compose up",
    "test": "jest",
    "tsoa": "npm run tsoa:v1",
    "tsoa:v1": "cd src/api/v1 && tsoa spec-and-routes"
  },
  "version": "0.16.1"
}
```

Níže uvedený úryvek kódu interpretuje ukázku jednoduchého serveru, napsaneho pomocí Node.js:

```
var http = require('http');

var server = http.createServer(function (req, res) {
   if (req.url == '/data') {
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.write(JSON.stringify({ message: "Hello World"}));
      res.end();
   }
});
server.listen(5000);

console.log('Node.js web server běží na portu 5000');
```

Tímto se vytvoři server, který běží na portu 5000. V rámci funkce createServer je prováděna kontrola URL požadavku a následně je odeslán JSON jako odpověď. Při přístupu na URL http://localhost:5000/data je zobrazen definovaný JSON:

```
{"message":"Hello World"}
```

2.5.1 Alternativa Deno

Node.js je nejstarší a zároveň nejrozšířenější platformou pro vývoj serverového kódu v jazyce JavaScript. Nicméně během své dlouhé existence vzniklo několik konkurentů, mezi které patří i Deno – nové, bezpečnostně zaměřené běhové prostředí pro JavaScript a TypeScript, které se snaží řešit některé problémy spojené s Node.js. Deno využívá stejný V8 engine jako Node.js, ale jeho jádro je napsáno v jazyce Rust. Níže jsou uvedeny klíčové vlastnosti, které odlišují Deno od Node.js [14; 15]:

- Import modulů: V Deno se moduly importují pomocí URL, a od poslední verze je také přidána možnost použití npm. Po spuštění aplikace Deno načte všechny importované moduly a uloží je do mezipaměti (tzv. cache). To znamená, že Deno nemusí stahovat moduly znovu, jakmile jsou již v cache uloženy.
- Kompilace TypeScriptu: Deno převádí TypeScript na JavaScript pomocí zabudovaného kompilátoru jazyka TypeScript a knihovny Rust swc. To znamená, že vývojář nemusí ručně transpilovat svůj TypeScript pro Deno pomocí příkazu tsc, na rozdíl od Node.js nebo prohlížečů.
- Použití ES modulů: Deno používá ES moduly místo modulů typu CommonJS, které jsou běžně využívány v Node.js.

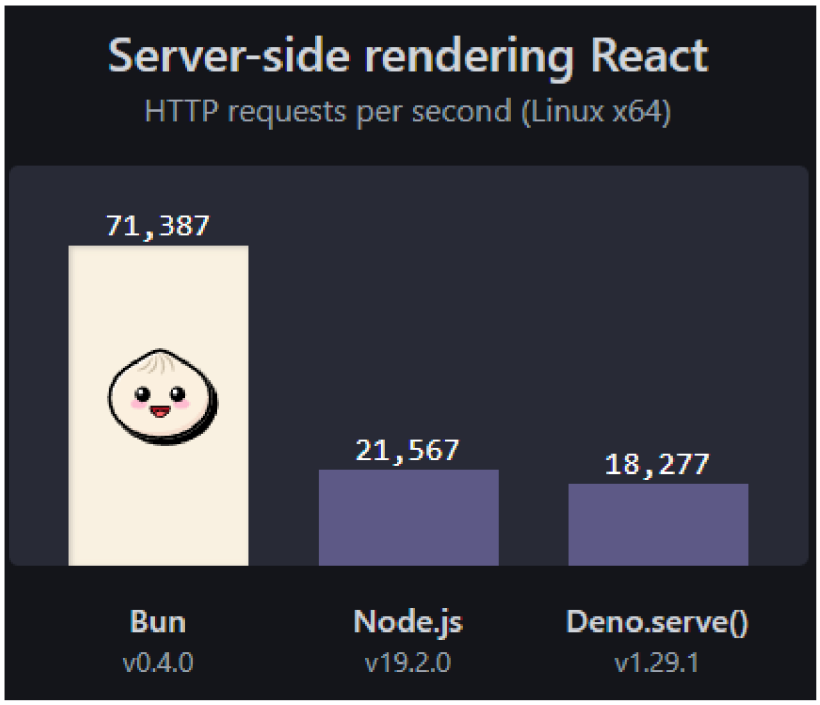
- Zlepšená bezpečnost: Deno disponuje vylepšenými bezpečnostními funkcemi ve srovnání s Node.js. V základní konfiguraci Deno neumožňuje programům přístup k disku, síti, spouštění dalších skriptů a přístupu k prostředí. Pokud je zapotřebí povolit některou z těchto funkcionalit, lze to nastavit pomocí příznaků v příkazovém řádku, například --allow-read=/tmp (umožní čtení souborů) nebo --allow-net=google.com (umožní přístup k síti pro doménu google.com).
- Další bezpečnostní vylepšení v prostředí Deno zahrnují to, že program vždy ukončí svoji činnost při nezachycené chybě (tzv. Uncaught Error). To zabrání pokračování programu po nezachycené chybě, což by mohlo vést k nepředvídatelným výsledkům.

Vývoj Node.js však pokračuje a s každým rokem se posouvá dále, přičemž se neustále pracuje na vylepšování jeho potenciálních nedostatků. Na rozdíl od Deno má Node.js výhodu v rozsáhlém ekosystému balíčků, pro které neexistují ekvivalenty v Deno. Závěrem lze konstatovat, že Deno má stále mnoho nedostatků, aby nahradilo Node.js.

2.5.2 Alternativa Bun

Dalším konkurentem Node.js je Bun. Bun je novým běhovým prostředím pro JavaScript, které bylo vytvořeno s důrazem na rychlost. Obsahuje vestavěný nástroj pro svazování (tzv. bundling), překladač, testovací program a správce balíčků kompatibilní s npm. Hlavním rozdílem oproti Node.js a Deno je jeho vysoká rychlost (viz obr. 4), která je dosažena následujícími faktory [14, 16]:

- Bun používá JavaScriptCore engine místo již zmíněného V8, který je rychlejší při spouštění a vykonávání příkazů.
- Bun je napsán v nízkoúrovňovém programovacím jazyce Zig, který umožňuje manuální správu paměti.
- Autoři prostředí Bun pravidelně optimalizují kód.



Obr. 4: Porovnání počtů HTTP požadavků za sekundu (Linux x64), provedených na straně serveru React pro různé frameworky z oficiální dokumentace Bun [16]

Bun nativně implementuje stovky Node.js API a webových rozhraní, včetně přibližně 90 % funkcí Node-API (nativní moduly), fs, path, Buffer a dalších. Jeho cílem je provádět spouštění JavaScriptu mimo prohlížeč, což přináší výhody v podobě vylepšeného výkonu, spolehlivosti infrastruktury a vyšší produktivity vývojářů pomocí jednodušších nástrojů. [16]

Bun implementuje webová rozhraní API, jako je fetch, WebSocket a ReadableStream. Podporuje práci s npm, umožňuje transpilaci JSX/TS souborů a poskytuje testovací nástroje jako Jest a další. [16]

Závěrem lze konstatovat, že Bun představuje novou generaci běhových prostředí, která může potenciálně nahradit Node.js. Nabízí všechny výhody Node.js a zároveň řeší jeho nedostatky. Umožňuje rychlý a snadný vývoj a spouštění aplikací, což přináší významné vylepšení pro vývojáře.

2.6 Testování webových aplikací

V následující podkapitole se práce bude zabývat teorií testování webových aplikací. Testování webových aplikací zahrnuje proces ověřování webového softwaru s cílem zajistit, že splňuje požadavky a funguje správně. Testování je klíčovým prvkem v procesu vývoje softwaru a pomáhá identifikovat chyby a nedostatky ještě před nasazením aplikace do produkčního prostředí. Nedostatečné testování může vést k problémům, se kterými se mohou koncoví uživatelé setkat a které mohou negativně ovlivnit jejich spokojenost s webovou aplikací. [17]

Existuje čtyři hlavní druhy testování, které je nutné dokončit před uvedením produktu do provozu: testování jednotek, integrační testování, systémové testování a akceptační testování. Každý z těchto typů testování má svůj specifický účel a přispívá k celkové kvalitě webové aplikace. [18]

2.6.1 Testování jednotek

Testování jednotek je klíčovou technikou testování softwaru, která se zaměřuje na testování jednotlivých komponent nebo modulů softwarové aplikace izolovaně od ostatních částí systému. Jeho hlavním cílem je ověřit správnou funkčnost a očekávané chování každé jednotky kódu. Při psaní jednotkových testů vývojáři vytvářejí automatizované testovací scénáře, které kontrolují jednotlivé funkce a chování kódu.
Existují dva přístupy k psaní jednotkových testů. Prvním je přístup zvaný Test-Driven Development (TDD), kdy vývojář nejprve napíše testovací případ definující očekávané chování jednotky kódu a až poté implementuje samotný kód, který musí testem projít. Druhým přístupem je psaní jednotkových testů po napsání samotného kódu, kdy vývojář nejprve implementuje funkční jednotku kódu a následně ji testuje. [18]

Jednotkové testy se obvykle automaticky spouštějí jako součást kontinuální integrace (CI). To znamená, že všechny testy jsou spouštěny při každé změně v kódu a uložení do verzovacího systému. Tím je zajištěno, že při změně nedojde k porušení funkčnosti a nové funkce nebo změny jsou řádně otestovány. Testování jednotek je důležitou součástí vývoje aplikace, protože pomáhá odhalit chyby v rané fázi vývoje a zlepšuje celkovou kvalitu softwaru a kódu. [18]

2.6.2 Integrační testování

Integrační testování je významnou technikou testování softwaru, která se zaměřuje na ověření správné integrace mezi různými komponentami a moduly softwarové aplikace. Jeho hlavním cílem je testovat interakce mezi těmito částmi systému a zajistit, že fungují správně a podle očekávání. [18]

Při integračním testování se vývojáři zaměřují na testování rozhraní a vzájemné interakce mezi jednotlivými moduly či komponentami systému. Tento druh testování se obvykle provádí až po dokončení testování jednotek. [18] Existují dva hlavní přístupy k provádění integračních testů: shora dolů (top-down) a zdola nahoru (bottom-up) [19].

Přístup shora dolů zahrnuje nejprve testování modulů nebo komponent vyšší úrovně, následované testováním modulů nebo komponent nižší úrovně. Naopak přístup zdola nahoru nejprve testuje moduly nebo komponenty nižší úrovně a postupně se posouvá ke komponentám vyšší úrovně. Integrační testování je náročnější než testování jednotek, protože vyžaduje testování vzájemných interakcí mezi různými moduly či komponentami. [19]

Testování integrace pomáhá identifikovat potenciální problémy, které se mohou vyskytnout při kombinaci různých komponent a jejich interakcí. Například se zaměřuje na testování toku dat a komunikace mezi částmi systému. V praxi je dobrým zvykem psát testy, které prověřují například připojení a interakci s databází (takové testy jsou součástí

praktické části této bakalářské práce). Integrační testy často také tvoří součást kontinuální integrace (CI) procesu. [18; 19]

Integrační testování přináší hodnotu tím, že odhaluje případné nedostatky a problémy spojené s propojením různých komponent a modulů. Tím je zajištěna správná integrace a funkčnost systému jako celku. [18; 19]

2.6.3 Systémové testování

Systémové testování představuje klíčovou fázi, kdy se prověřuje chování aplikace jako celku. Jeho hlavním cílem je zajistit, že systém splňuje stanovené požadavky a pracuje tak, jak se očekává v produkčním prostředí. Systémové testování se zaměřuje na testování celého systému včetně end-to-end scénářů, uživatelských rozhraní, integrace s ostatními systémy a interakce s hardwarovým a softwarovým prostředím. Obvykle se provádí po dokončení jednotkového a integračního testování. [18]

Cílem systémového testování je odhalit problémy, které se mohou objevit pouze při spuštění celého systému. Patří sem například výkonnostní problémy, zranitelnost zabezpečení nebo nedostatky v použitelnosti. Pro systémové testování existuje několik přístupů, jako je manuální testování, automatizované testovací nástroje nebo jejich kombinace. [18]

Systémové testování hraje klíčovou roli při zajišťování kvality systému, neboť identifikuje případné nedostatky a problémy, které mohou ovlivnit jeho funkčnost a výkon. Zabezpečuje se tak, že systém je připraven pro nasazení a plně vyhovuje požadavkům uživatelů. [18]

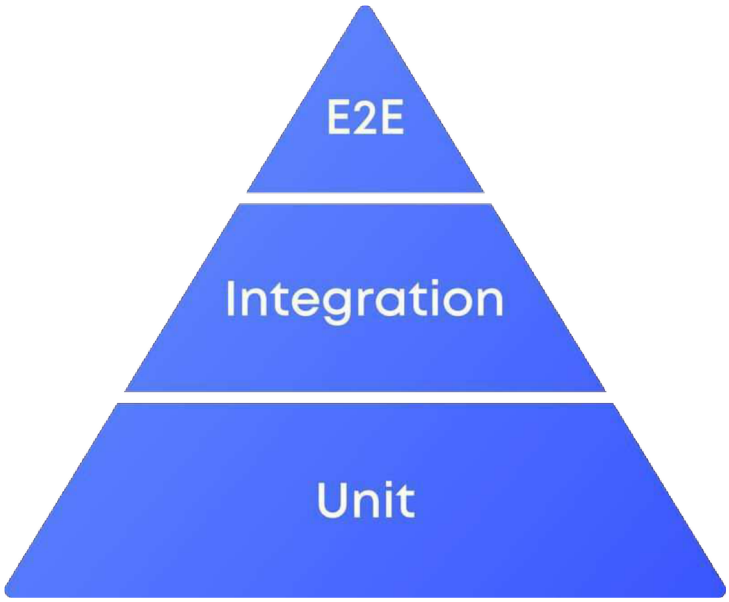
2.6.4 Akceptační testování

Posledním stupněm testování je akceptační testování, které slouží k ověření, zda je produkt (aplikace) připraven k uvedení na trh a splňuje všechny obchodní a uživatelské požadavky. Toto testování se provádí prostřednictvím scénářů z reálného světa, při kterých jsou používána realistická data a vstupy, aby bylo zajištěno, že aplikace funguje správně a plní zamýšlené funkce. [18]

Akceptační testování může být realizováno pomocí různých metod, včetně manuálního testování, automatizovaného testování nebo uživatelského akceptačního testování (UAT). Cílem těchto přístupů je zajistit, že je aplikace schopna splnit požadavky a potřeby obchodních partnerů a koncových uživatelů. Akceptační testování je klíčové pro zajištění spokojenosti uživatelů a úspěchu produktu na trhu. [18]

2.6.5 End-to-end testování

Rozdíl mezi end-to-end (E2E testování) a systémovým testování je v tom že end-to-end testuje celý tok softwarové aplikace od začátku do konce s očekávanými výsledky na výstupu, zatímco testování systému se zaměřuje na chování systému jako celku [20]. Existuje testovací pyramida, viz obr. 5:



Obr. 5: Testovací pyramida [21]

Testovací pyramida na obr. 5 ilustruje, že testování aplikace by mělo obsahovat nejvíce jednotkových testů a nejméně end-to-end testů. To je z důvodu, že end-to-end testy vyžadují více času na spuštění a provedení než ostatní druhy testů. [21; 22]

Cílem end-to-end testování je identifikovat systémové závislosti, zajistit integritu přenosu dat mezi různými komponentami a systémy a simulovat scénáře reálného uživatele. Tento druh testování může zahrnovat testování uživatelského rozhraní, rozhraní API, databází a dalších softwarových součástí, stejně jako externích systémů nebo služeb integrovaných v aplikaci. Obvykle se end-to-end testování provádí po jednotkovém a integračním testování. [22]

Tradičním přístupem k end-to-end testování je použití aplikace Selenium (v jazyce Java). Nicméně, v dnešní době se vývoj frontendu často provádí pomocí JavaScriptových frameworků, jako jsou Angular, React nebo Vue. Proto je pro vývojáře výhodnější psát end-to-end testy s využitím JavaScriptových frameworků. Existuje několik hlavních JavaScriptových frameworků pro end-to-end testování, které se dnes používají [22]:

- WebdriverJS
- Protractor
- WebdriverIO
- NightwatchJS
- Cypress
- TestCafe

Výše uvedené frameworky lze rozdělit do dvou skupin: založené na Selenium (Selenium-based) a nezaložené na Selenium (Non-Selenium-based). Selenium-based frameworky využívají webové ovladače (web driver) pro interakci s prohlížeči, zatímco Non-Selenium-based frameworky mohou komunikovat s prohlížečem přímo, což

představuje významnou výhodu. U těchto frameworků není nutné instalovat ovladače, což přispívá k jejich efektivitě a snadnému nasazení. [22]

End-to-end testy pro webovou aplikaci E2E Manager jsou implementovány pomocí Non-Selenium-based frameworku Cypress. Cypress je open source testovací framework pro end-to-end testování, který je postaven na JavaScriptu. Tento framework umožňuje vývojářům psát automatizované testy pro webové aplikace. [23]

Cypress přináší několik výhod, mezi které patří [23]:

- Flexibilita: Cypress umožňuje psát různé typy testů, včetně end-to-end testů, integračních testů, jednotkových testů a testů pro komponenty.
- Kompletnost: Cypress dokáže testovat všechno, co se děje v prohlížeči. Je schopen ovládat a testovat různé aspekty webové aplikace.
- Integrační možnosti: Po vytvoření sady testů a integraci s CI providerem může Cypress Cloud zaznamenávat testovací běhy a jejich výsledky. Poskytuje také možnost zachytávání snímků obrazovky a videí (tzv. snapshots) během testování.
- Ladění: Cypress umožňuje snadné ladění testů přímo ze známých vývojářských nástrojů, což usnadňuje odhalování a opravování chyb.
- Kontrola chování: Cypress umožňuje kontrolu různých aspektů aplikace, včetně chování funkcí, odpovědí serveru, síťového provozu a časovačů.

Většina testovacích frameworků, včetně Selenium, pracuje tak, že běží mimo prohlížeč a spouští vzdálené příkazy. Framework Cypress funguje opačně. Cypress je testovací nástroj, který pracuje přímo v prohlížeči a při svém spuštění využívá prostředí Node.js. Tento přístup umožňuje neustálou komunikaci a synchronizaci mezi Cypressem a Node.js. Díky této interakci je možné v reálném čase reagovat na události v aplikaci a současně provádět operace mimo prostředí prohlížeče. [22; 23]

Díky tomu, že je Cypress nainstalován přímo na počítači, má také přístup k operačnímu systému. To poskytuje rozšířené možnosti, jako je ukládání snímků obrazovky, nahrávání videí, manipulace se souborovým systémem a síťové operace. [23]

Níže uvedený úryvek kódu představuje ukázku jednoduchého testu z dokumentace Cypress [23]:

```
describe('Simple Test', () => {
  it('clicks the link "type"', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()
  })
})
```

V rámci tohoto end-to-end testu je pomocí nástroje Cypress simulováno reálné navštívení webové stránky na zadané adrese. Test dále vyhledá DOM element s identifikátorem 'type' a provede na něm kliknutí.

3 VÝVOJ E2E MANAGERU

Tato kapitola se věnuje architektuře webové aplikace E2E Manager, která slouží k monitorování end-to-end testů a popisuje technologie použité při její realizaci.

3.1 Použité technologie

V následující podkapitole budou popsány použité technologie při tvorbě frontendu a backendu webové aplikace E2E manager. Budou uvedeny následující technologie: React.js, Material UI, React Query, Koa, Knex.js, Tsoa a OpenAPI (Swagger).

3.1.1 React.js

Pro implementaci webové aplikace E2E Manager na straně klienta byla použita knihovna React.js. Jedná se o JavaScriptovou knihovnu pro tvorbu interaktivních uživatelských rozhraní. React.js byl vyvinut Jordanem Walkem, softwarovým inženýrem ve společnosti Meta (dříve Facebook). Poprvé byl nasazen na Facebook News Feed v roce 2011 a později na Instagramu v roce 2012. V květnu 2013 byl uvolněn pod open source licencí na konferenci JSConf US. Renderování aplikace pomocí React.js se provádí přímo ve webovém prohlížeči (tzv. CSR – Client-Side Rendering). [24]

Hlavní vlastnosti React.js [25]:

- **Komponentní přístup.** React.js umožňuje vytvářet komponenty, které spravují svůj vlastní stav, a tyto komponenty lze dále sestavovat do komplexních uživatelských rozhraní. Data mohou být předávána mezi komponentami a stav aplikace lze udržovat mimo DOM.
- **Deklarativní přístup.** Stačí navrhnout jednoduchá zobrazení pro každý stav aplikace a React.js efektivně aktualizuje a vykresluje pouze potřebné komponenty při změně dat.
- **Virtuální DOM.** React.js vytváří a ukládá do mezipaměti virtuální strom DOM, což je kopie DOM, která se mění rychleji než skutečný model. To umožňuje rychlou aktualizaci stránky. Pokud uživatel provede akci nebo dojde k události, která vyvolá změnu objektu, virtuální DOM se rychle aktualizuje a poté se aktualizuje skutečný DOM. Tím je zajištěno okamžité zobrazení změn na stránce pro uživatele.
- **Částečná aktualizace DOM.** Pro zvýšení rychlosti aktualizace skutečného DOM React.js uchovává dvě zjednodušené kopie – aktuální a předchozí. Při aktualizaci knihovna porovnává tyto verze a změní pouze část stromu, která se skutečně změnila. Tím se minimalizuje opětovné načítání celého DOM a zlepšuje se výkon stránky.
- **Syntaxe JSX.** JSX je rozšířením jazyka JavaScript, které umožňuje popisovat HTML prvky pomocí kódu React.js. Komponenty v Reactu jsou

definovány pomocí JSX, což usnadňuje jejich úpravy a čitelnost. Přestože se syntaxe podobá HTML, stále se jedná o jazyk JavaScript, který umožňuje snadné a efektivní manipulace s DOM pomocí kódu.

- **React Hooks.** Hooks jsou novým prvkem přidaným v Reactu 16.8. Umožňují vyhnout se používání tříd a místo toho využít moderní funkce Reactu, včetně správy stavů a dalších vylepšení.

React.js také podporuje TypeScript. Navíc je možné použít React.js pro renderování na straně serveru (tzv. SSR – Server-Side Rendering) pomocí Node.js. [25]

Níže uvedený úryvek kódu představuje jednoduchý příklad implementace v React.js [26]:

```
function MyButton() {
  return (
    <button onClick={() => { console.log("Button was clicked") }}>
      I'm a button
    </button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```

Spouštění kódu v React.js většinou začíná v komponentě *App*. V ukázce kódu výše hlavní komponenta *App* vrátí tag *div*, do kterého je umístěn jednoduchý nadpis "Welcome to my app" a další komponenta *MyButton*, která vyrendruje tlačítko. Výsledek spuštění tohoto kódu je zobrazen na obr. 6:

Obr. 6: Výsledek renderování jednoduchého React.js kódu [26]

3.1.2 Material UI

Pro tvorbu uživatelského rozhraní (UI) v E2E Manageru byla využita open source knihovna Material UI pro React.js. Tato knihovna implementuje designový systém nazvaný Material Design, který byl vyvinut společností Google. Material UI nabízí

rozsáhlou kolekci předpřipravených komponent, které lze snadno použít při vývoji aplikace. Tato knihovna umožňuje rychlé a jednoduché vytváření vlastních designových komponentů pomocí již existujících komponent knihovny, které lze přizpůsobit podle potřeb uživatele. [27]

Ukázka použití této knihovny z dokumentace vypadá následně [27]:

```
import * as React from 'react';
import Button from '@mui/material/Button';

export default function MyApp() {
  return (
    <div>
      <Button variant="contained">Hello World</Button>
    </div>
  );
}
```

Výše uvedený kód slouží k vykreslení tlačítka z knihovny Material UI, viz obr. 7.

Obr. 7: Tlačítko vyrenderované pomoci knihovny Material UI [27]

Kromě sady komponentů a jejich upravování nabízí knihovna Material UI také možnost vytvářet vlastní témata, která lze použít ve vyvíjené aplikaci. [27]

3.1.3 React Query

Pro načítání dat ze serveru webové aplikace E2E Manager je používána knihovna React Query. Tato knihovna zjednodušuje načítání, cachování, synchronizaci a aktualizaci stavu serveru v aplikacích postavených na Reactu. Obvykle se pro ukládání a manipulaci s asynchronními daty v React aplikacích využívají kombinace useState a useEffect hooků nebo obecnější knihovny pro správu stavu. Tyto přístupy jsou vhodné pro práci se stavem klienta, ale nejsou ideální pro práci se stavem serveru. Stav serveru se liší svými charakteristikami [28]:

- Uchovává data na vzdáleném místě, které nelze přímo ovládat ze strany klienta.
- Vyžaduje asynchronní rozhraní API pro načítání a aktualizaci dat.
- Data v aplikaci se mohou stát zastaralými.

Vývojář musí zajistit cachování, rychlou aktualizaci dat a řešení zastaralých dat. React Query poskytuje nástroje pro řešení těchto problémů a usnadňuje práci se stavem serveru a manipulaci s daty v aplikaci. Z technického hlediska React Query nabízí následující [28]:

- Snížení množství kódu v aplikaci.
- Jednoduché vytváření nových funkcionalit bez nutnosti řešit složitost připojování nových zdrojů dat ze serveru.
- Optimalizaci rychlosti aplikace.
- Optimalizaci využití paměti.

Níže uvedený příklad ilustruje použití React Query v aplikaci E2E manageru:

```
const fetchRuns = () =>
        client.v2.runs
                .list({ limit: 20, orderBy: order.by, orderDir: order.dir })
                .then((runs_: V2ListRunsResponse) => runs_.items);

const { data: runsData, error, isLoading } = useQuery(['runs', order], fetchRuns);
```

Výše uvedený kód umožňuje načítání běhů testů ze serveru a jejich použití ve webové aplikaci E2E manager. Funkce *fetchRuns* vytvoří instanci třídy *Client*, která slouží pro komunikaci se serverem a provolání potřebných koncových bodů API. Funkce *list* objektu *Client* je volána s parametry *limit*, *orderBy* a *orderDir*, které omezují počet vrácených běhů a určují jejich seřazení. Výsledkem je pole běhů (*runs_.items*), které je následně vráceno.

Hook *useQuery* zajišťuje načítání dat ze serveru a jejich správu v aplikaci. Hook je volán s parametry *['runs', order]*, které slouží jako identifikátor pro daný požadavek na data. Funkce *fetchRuns* je předána jako druhý argument a definuje, jak mají být data načítána. Výsledky načítání dat jsou destrukturovány do proměnných *runsData*, *error* a *isLoading*. *runsData* obsahuje načtená data běhů, *error* případnou chybovou zprávu a *isLoading* indikuje, zda probíhá načítání dat.

3.1.4 Koa

Pro implementaci serverové části aplikace E2E Manager byl použit framework Koa. Koa je navržen s cílem poskytnout solidní základ pro vývoj webových aplikací a rozhraní API. Využívá asynchronních funkcí, což umožňuje efektivnější zpracování požadavků a eliminaci tzv. callback hell. Koa poskytuje objektový model aplikace, který obsahuje různé middleware funkce. Tyto funkce jsou skládány a spouštěny postupně na základě příchozích požadavků. Framework Koa také poskytuje různé metody pro časté úkoly, jako je obsluha obsahu, aktualizace mezipaměti (cache), podpora proxy a přesměrování. [29]

Níže uvedený úryvek kódu představuje jednoduchý příklad použití React Query [29]:

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

V ukázce kódu výše je vytvořen server běžící na portu 3000, který vrací text "Hello World". Oproti kódu napsanému pro Node.js (viz kapitola 2.4) poskytuje Koa aplikace kompaktnější a přehlednější zápis.

3.1.5 Knex.js

Aplikace E2E manager využívá interakce s databází pomocí nástroje Knex.js. Knex.js je nástroj pro tvorbu SQL dotazů pro různé SQL databázové systémy, včetně PostgreSQL, MySQL a dalších [30]. Jeho hlavním přínosem je podpora transakčních databází. Transakce jsou důležitou vlastností relačních databází, protože zajišťují správné obnovení databáze v případě poruchy a udržují konzistenci dat. Všechny dotazy prováděné v rámci transakce jsou vykonávány na stejném připojení k databázi a jsou považovány za jednotnou pracovní jednotku. Pokud nastane porucha, databáze vrátí všechny dotazy provedené v rámci této transakce do stavu před zahájením transakce. [30] Příklad použití transakce je uveden výše v ukázce kontroléru. Pro aplikaci E2E manager je používán relační databázový systém PostgreSQL.

3.1.6 Tsoa

Dalším frameworkem pro backendovou část E2E manageru je Tsoa. Jedná se o framework s integrovaným kompilátorem OpenAPI, který umožňuje vytvářet serverové aplikace s použitím Node.js a TypeScript [31]. Tsoa je dobře kompatibilní s frameworky jako express, hapi, koa a dalšími [31]. Slouží k psaní kontrolérů, které jsou součástí návrhového vzoru MVC (Model-View-Controller). Tento vzor rozděluje aplikaci do tří částí [32; 33]:

- **Model:** Model zajišťuje interakci s daty a základní funkce aplikace. Typicky se využívá pro vkládání, načítání nebo aktualizaci dat v databázi pro aplikaci.
- **Pohled (View):** Pohled se stará o zobrazení dat v uživatelském rozhraní. Uživatelé prostřednictvím pohledu komunikují s komponentami aplikace.

- **Kontrolér (Controller):** Kontrolér slouží jako prostředník mezi modelem a pohledem. Řídí tok dat do objektu modelu a aktualizuje pohled při každé změně dat.

Následující příklad demonstruje použití frameworku Tsoa pro implementaci kontroléru v serverové části aplikace E2E manageru:

```
@Route('specs')
export class RunSpecsController extends Controller {
	@Get('{runId}')
	@SuccessResponse(200, 'Specs retrieved')
	async list(
		@Query() runId?: V2ListRunSpecsRequest['runId'],
		@Query() offset?: V2ListRunSpecsRequest['offset'],
		@Query() limit?: V2ListRunSpecsRequest['limit'],
		@Query() orderBy?: V2ListRunSpecsRequest['orderBy'],
		@Query() orderDir?: V2ListRunSpecsRequest['orderDir'],
		@Query() name?: V2ListRunSpecsRequest['name'],
	): Promise<V2ListRunSpecsResponse> {
		const { totalCount, items } = await db.transaction(async (tx) => {
			return RunSpec.load({
				tx,
				offset,
				limit,
				filter: makeFilter({
					runId,
					name,
				}),
				orderBy,
				orderDir,
			});
		});

		return {
			totalCount,
			items,
		};
	}
}
```

V ukázce kódu výše je definována cesta */specs/* pomocí dekorátoru *@Route* nad třídou *RunSpecsController*. Dále je definována metoda list. Dekorátor *@Get* ve spojení

se základní cestou */specs/* říká frameworku Tsoa, že má vyvolat tuto metodu pro každý GET požadavek na cestu */specs/{runId}/*, kde *{runId}* je parametr předávaný do kontroléru prostřednictvím URL. Tato metoda list také přijímá volitelné dotazy (Query). Dále je definováno, co tato metoda musí vrátit. Vrací Promise s typem odpovědi *V2ListRunSpecsResponse*. Následně je vytvořena transakce s databází pomocí knihovny Knex. V této transakci se volá metoda *load* v modelu *RunSpec* a předávají se do ní dotazy z metody *list*, které jsou získány z místa, kde byla tato metoda zavolána. Metoda *list* vrátí odpověď s počtem položek (*totalCount*) a samotnými položkami (*items*).

3.1.7 OpenAPI

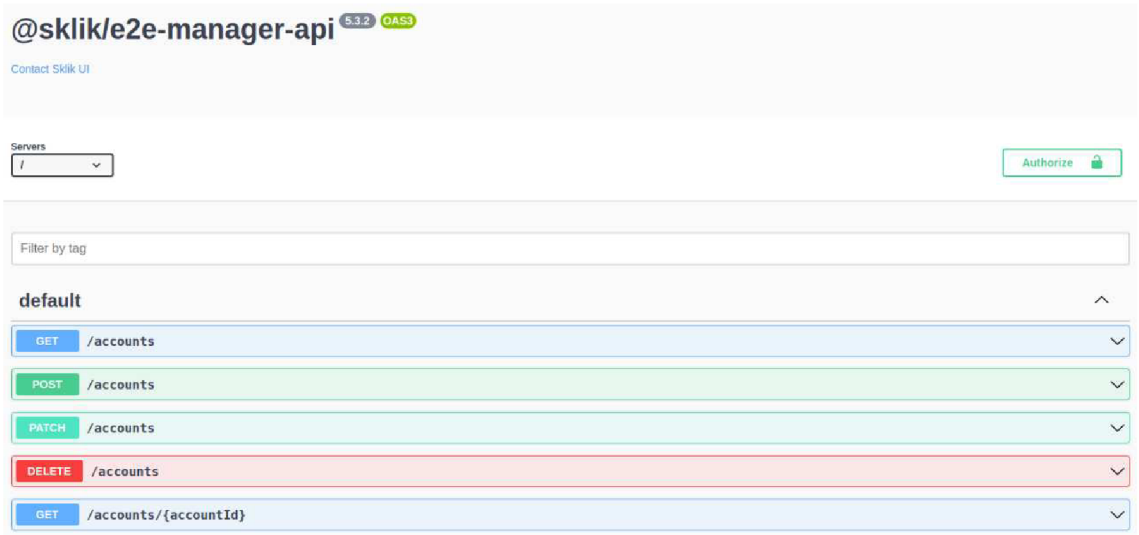
Aplikace E2E manager využívá nástroj pro generování dokumentace API. Specifikace OpenAPI (OAS) definuje standardní rozhraní pro HTTP API, které umožňuje lidem i počítačům porozumět funkcionalitě služby bez přístupu ke zdrojovému kódu. To umožňuje uživatelům komunikovat s vzdálenými službami s minimální implementační logikou. Mezi nejznámější a široce používané nástroje pro implementaci specifikace OpenAPI patří Swagger. Swagger umožňuje popsat strukturu API, aby ji počítače mohli číst. Lze také automaticky generovat klientské knihovny pro API v různých jazycích nebo provádět automatické testování. Swagger žádá API, aby vrátilo YAML nebo JSON soubor obsahující podrobný popis celého rozhraní. Tento soubor je v podstatě seznam zdrojů API, který odpovídá specifikaci OpenAPI. Pro vytvoření specifikace je třeba přidat konfigurační informace, jako jsou [34]:

- Seznam všech operací poskytovaných API
- Parametry dotazů API a formát dat, kterým jsou dotazy vráceny.
- Autorizace pro API, pokud je nutná.

Specifikace Swagger pro API lze napsat ručně, nebo ji nechat vygenerovat automaticky z anotací ve zdrojovém kódu [34]. Ukázka automaticky vygenerovaného souboru ve formátu JSON pomocí frameworku Tsoa pro aplikaci E2E manager vypadá následovně:

```
{
        "components": {
                "examples": {},
                "headers": {},
                "parameters": {},
                "requestBodies": {},
                "responses": {},
                "schemas": {
                        "V2Account": {
                            "allOf": [
                                {
                                    "$ref": "#/components/schemas/V2AccountPartial"
                                },
                                {
                                    "properties": {
                                            "allocation": {
                                               "allOf": [
                                                    {
                                                       "$ref":
"#/components/schemas/V2AccountAllocation"
                                                    }
                                            ],
                                            "nullable": true
                                    }
                        },
                        "required": [
                                "allocation"
                        ],
                        "type": "object"
                }
        ]
},}}}
```

Z výše uvedené ukázky souboru vygeneruje se Swagger dokumentace, viz obr. 8.



Obr. 8: Swagger dokumentace

3.2 Popis architektury webové aplikace E2E manager

Tato podkapitola se zaměřuje na popis architektury a principů fungování webové aplikace E2E Manager. V rámci tohoto popisu budou podrobně popsány dvě verze API, konkrétně APIv1 a APIv2, které jsou využívány webovou aplikací. Následně bude představena samotná aplikace a budou popsány jednotlivé komponenty, které tvoří její strukturu a zajišťují její funkcionalitu.

3.2.1 Popis API

API (Application Programming Interface) je sada pravidel, která definují, jak se aplikace nebo zařízení mohou vzájemně připojit a komunikovat. REST API (také známé jako RESTful API) je rozhraní pro programování aplikací, které splňuje omezení architektonického stylu REST a umožňuje interakci s webovými službami RESTful. Když klient pošle požadavek prostřednictvím rozhraní RESTful API, přenese se stav entity na žadatele nebo koncový bod (endpoint). Tyto entity jsou nejčastěji reprezentovány ve formátu JSON (Javascript Object Notation). Rozhraní REST API komunikují pomocí HTTP požadavků k provádění běžných databázových operací, jako je vytváření, čtení, aktualizace a mazání záznamů. [35]

Webová aplikace E2E manager používá 2 verze API: v1 a v2. Architektura backendu E2E manageru je navržena tak, že existuje třída *Client*, která slouží k typové kontrole požadavků a odpovědí API volání. To znamená, že pomocí této třídy je při volání koncových bodů API známo, jaké HTTP metody jsou v backendu definovány, jaké parametry lze posílat a jakou odpověď po volání konkrétního koncového bodu očekávat.

Třída *Client* obsahuje dvě další vnořené třídy a konstruktor pro tyto třídy:

```
import { ApiV1 } from './v1';
import { ApiV2 } from './v2';
import { Dispatcher } from './types';

export { Dispatcher };

export default class Client {
        v1: ApiV1;
        v2: ApiV2;

        constructor(dispatcher: Dispatcher) {
                this.v1 = new ApiV1(dispatcher);
                this.v2 = new ApiV2(dispatcher);
        }
}
```

Uvnitř každé třídy definice API jsou implementovány metody, které vrací instance nových tříd:

```
export class ApiV2 extends ApiBase {
        constructor(dispatcher: Dispatcher) {
                super(dispatcher);
        }

        get projects() {
                return new ApiProjects(this);
        }

        get runs() {
                return new ApiRuns(this);
        }

        get accounts() {
                return new ApiAccounts(this);
        }

        get runSpecs() {
                return new ApiRunSpecs(this);
        }
}
```

V ukázce výše je vidět třída *APIV2*, která dědí z třídy *ApiBase* a využívá její konstruktor. Dalším příkladem je nově definovaná třída *ApiRuns*, která má následující podobu:

```
class ApiRuns extends ApiBase {
        async list(req: V2ListRunsRequest) {
                const { body } = await this.dispatch<V2ListRunsResponse>({
                        method: 'GET',
                        urlTemplate: '/v2/runs',
                        req: {
                                params: null,
                                query: req,
                                body: null,
                        },
                });

                return body;
        }

        async create(req: V2CreateRunsRequest) {
                const { body } = await this.dispatch<V2CreateRunsResponse>({
                        method: 'POST',
                        urlTemplate: '/v2/runs',
                        req: {
                                params: null,
                                query: null,
                                body: req,
                        },
                });

                return body;
        }
}
```

V ukázce výše jsou definovány 2 asynchronní metody – *list* a *create*. Každá z těchto metod přijímá parametry určitého typu a volá zděděnou metodu *dispatch*, která zajišťuje volání koncových bodů API. Metoda *dispatch* je definována tak, že vyžaduje objekt obsahující následující atributy:

- *method:* Zde se specifikují základní metody HTTP používané při vývoji RESTful API. Mezi nejčastější patří [36]:

- ❖ GET: Nejběžnější metoda, která slouží k získání reprezentace obsahu a dat entity. Používá se pouze pro čtení a udržuje data v bezpečí a entitu idempotentní.
- ❖ POST: Používá se k úpravě nebo vytvoření entity. Je to jediná metoda HTTP RESTful API, která se primárně používá pro práci s kolekcemi entit.
- ❖ PUT: Ekvivalent POST s jednou entitou, který aktualizuje entitu úplnou náhradou jejího obsahu. Je to nejběžnější metoda pro aktualizaci informací o entitách.
- ❖ PATCH: Metoda pro aktualizaci entit. Na rozdíl od metody PUT pouze upravuje obsah entit. Úpravy jsou vyjádřeny ve standardním formátu, jako je například JSON nebo XML.
- ❖ DELETE: Slouží k odstranění uvedené entity.

Ve výše uvedeném příkladu je vidět že metoda list používá GET metodu a metoda create používá POST metodu.

- *urlTemplate*: Jedná se o cestu k souboru, ve kterém jsou definována API pro aplikaci.
- *req*: Zde jsou uvedeny parametry, které jsou předány zvenčí do API.

Server pro aplikaci E2E manager, který je implementován pomocí frameworku Koa, má následující podobu:

```
const server = new Koa()
        .use((ctx, next) => {
                ctx.set('Access-Control-Allow-Origin', '*');
                ctx.set('Access-Control-Allow-Methods', '*');
                ctx.set('Access-Control-Allow-Headers', '*');
                return next();
        })
        .use(createSerializeResponseMiddleware())
        .use(
                koaBody({
                        jsonLimit: '250mb',
                }),
        )
        .use(controllers.routes())
        .use(controllers.allowedMethods())
        .listen(config.port);

    await new Promise((resolve) => server.on('listening', resolve));
return server;
```

V ukázce kódu výše lze vidět vytvoření serveru pomocí příkazu *new Koa()* a následně definici middleware pro tento server. Middleware je softwarová vrstva, která slouží k přenosu informací mezi službami a umožňuje interakci mezi různými aplikacemi. Cesty pro REST API jsou definovány pomocí middleware *.use(controllers.routes())*. Tyto cesty jsou rozděleny na dvě části, a to cesty pro API verze 1 (APIv1) a API verze 2 (APIv2):

```
router.use('/v1', v1Router.routes(), v1Router.allowedMethods());
router.use('/v2', v2Router.routes(), v2Router.allowedMethods());
```

3.2.2 APIv1

Jednotlivé cesty ke koncovým bodům APIv1 jsou ručně definovány pomocí KoaRouter(). Níže je uveden výňatek kódu, který představuje tuto definici:

```
export default new KoaRouter()
        .get('/schema.json', (ctx) => {
                ctx.body = openapiSchema;
        })
        .get('/runs', createHandler(operations, 'listRuns'))
```

V ukázce výše je definována cesta */runs*. Při jejím navštívení se zavolá funkce *createHandler*, které je předán objekt *operations*, obsahující seznam možných operací API, a název požadované operace, která se má vykonat. Jako příklad je zde uvedena operace *listRuns*, která je zároveň funkcí:

```
async listRuns({ query }) {
        const { totalCount, items } = await db.transaction((tx) => {
            return loadRuns({
                tx,
                limit: query.limit,
                offset,
                orderBy: 'createdAt',
                orderDir: OrderDirection.DESC,
            });
        });

        return {
            totalCount,
            limit: query.limit,
            offset,
            items: items.map(mapRun),
        };
    },
```

V ukázce kódu výše je implementována funkce *listRuns*, která obsahuje asynchronní logiku načítání a zpracování dat. V této ukázce se vytváří databázová transakce pomocí knihovny Knex a volá se metoda *loadRuns* z modelu *runs* s parametry pro další zpracování. Model *runs* představuje interakce s daty a implementuje základní funkce nad entitou "*runs*". Funkce *loadRuns* načítá data z databáze na základě dodaných parametrů:

```
export async function loadRuns(options: LoadOptions): Promise<LoadResult> {
    const { items, where } = await loadOrFind(options);
    const totalCount = await queryTotalCount(options.tx('runs').modify(where));

    return { items, totalCount };
}
```

Výše uvedená funkce vrátí položky z databáze a jejich počet.

Funkce *loadOrFind* vypadá takto:

```
async function loadOrFind(options: LoadOptions) {
        const where = getWhere(options.filter, options.forUpdate);
        const items = await options
                .tx('runs')
                .modify(where)
                .modify(applyDisplayOptions(options, { orderBy: 'createdAt',
orderDir: OrderDirection.DESC }));

        return { where, items };
}
```

Tato funkce aplikuje parametry předávané zvnějšku na databázové dotazy. Funkce *loadOrFind* aplikuje parametry předávané zvenčí na databázové dotazy. To umožňuje například filtrovat odpověď na základě hodnoty ve sloupci tabulky. V ukázce výše je vidět použití funkce *where*, která filtruje výsledky na základě zadaného sloupce:

```
qb.where(columnName, filter.value);
```

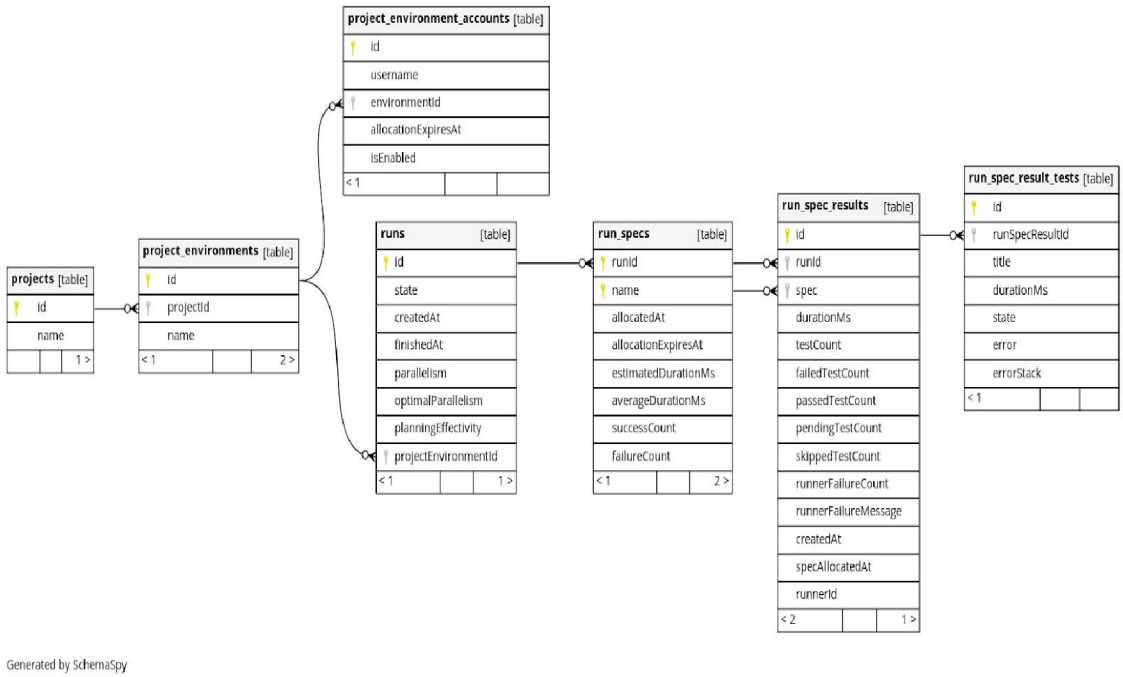
Pomocí query builderu Knex (qb) se vytváří SQL dotaz s parametry, který obsahuje název sloupce v databázi a hodnotu pro filtraci, kterou model dostává zvenčí.

3.2.3 APIv2

Cesty pro APIv2 jsou automaticky generovány pomocí knihovny Tsoa. Toto je dosaženo vytvořením instance KoaRouteru a jeho přidáním do funkce *RegisterRoutes*, která je zodpovědná za vytváření cest pro koncové body APIv2. Funkce obsluhující tyto cesty jsou definovány v samostatné složce *controllers*. Kontroler pro entitu "*runs*" je implementován s využitím knihovny Tsoa (viz kapitola 3.1.6):

```
@Route('runs')
export class RunsController extends Controller {
	@Get()
	@SuccessResponse(200, 'Projects retrieved')
	async list(
		@Query() offset?: V2ListRunsRequest['offset'],
		@Query() limit?: V2ListRunsRequest['limit'],
		@Query() orderBy?: V2ListRunsRequest['orderBy'],
		@Query() orderDir?: V2ListRunsRequest['orderDir'],
		@Query() id?: V2ListRunsRequest['id'],
		@Query() environmentId?: V2ListRunsRequest['environmentId'],
		@Query() state?: V2ListRunsRequest['state'],
		@Query() parallelism?: V2ListRunsRequest['parallelism'],
	): Promise<V2ListRunsResponse> {
		return db.transaction(async (tx) => {
			const { items, totalCount } = await Run.loadRuns({
				tx,
				offset,
				limit,
				filter: makeFilter({
					id,
					environmentId,
					state,
					parallelism,
				}),
				orderBy,
				orderDir,
			});

			return {
				totalCount,
				items,
			};
		});
	}
}
```

Tento kontroler používá stejný model a funkce jako v případě APIv1 (viz kapitola 3.2.2). E2E manager provádí end-to-end testy, které jsou napsané pomocí frameworku Cypress, a výsledky těchto testů jsou uloženy do databáze *e2e-manager*. API následně komunikuje s touto databází. Schéma struktury databáze *e2e-manager*, které bylo vygenerováno pomocí nástroje SchemaSpy, je zobrazeno na obr. 9.



Obr. 9: Schéma databáze *e2e-manager*

3.2.4 Struktura webové aplikace

Jak již bylo zmíněno, frontendová část webové aplikace E2E Manager je implementována pomocí frameworku React.js. Tímto frameworkem dochází k vykreslování aplikace na straně klienta (CSR – Client-side rendering). Pro implementaci dynamického směrování mezi různými komponentami v React aplikacích je v E2E Manageru využíván balíček npm nazvaný React Router Dom [37].

V tradičním přístupu při načítání webové stránky si prohlížeč vyžádá dokument ze serveru, stáhne a vyhodnotí CSS a JavaScript soubory a poté vykreslí HTML obsah odeslaný ze serveru. Při kliknutí uživatele na odkaz dojde k opětovnému načítání celé stránky. S využitím React Router Domu je možné vytvářet jednostránkové aplikace (Single Page Applications), kde se obsah dynamicky aktualizuje na základě změny adresy URL, aniž by bylo nutné obnovovat celou stránku. Při změně adresy URL se pouze aktualizuje obsah aktuální stránky. [38]

Následující ukázka kódu ilustruje směrování mezi komponentami v rámci E2E Manageru:

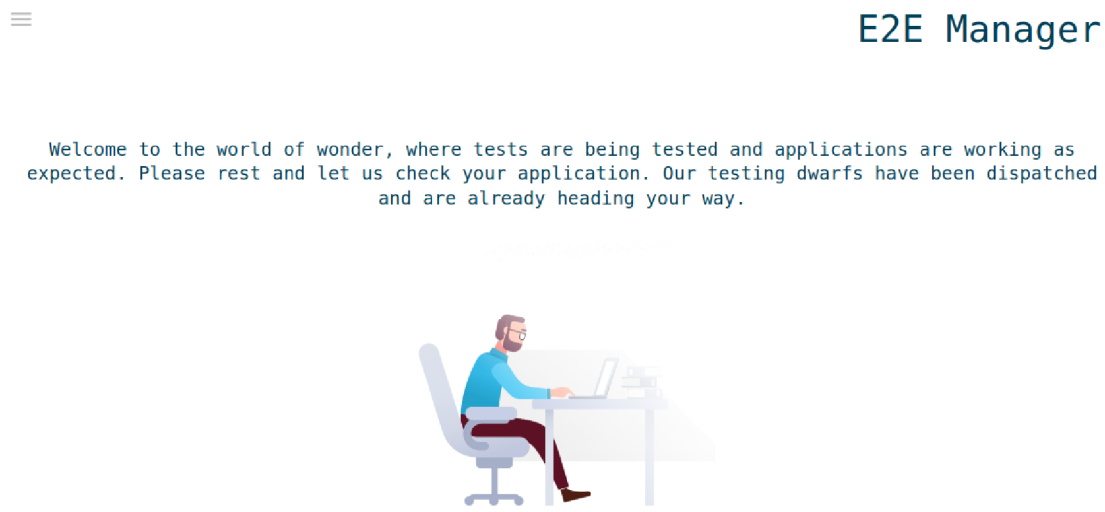
```
<BrowserRouter basename={process.env.NODE_ENV === 'development' ? '/' : '/e2e-manager'}>
    <Routes>
        <Route path="/" element={<Intro />} />
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/runs" element={<Runs />} />
        <Route path="/specs/:runId" element={<Specs />} />
        <Route path="/result/:encodedParams" element={<Results />} />
        <Route path="/accounts" element={<Accounts />} />
    </Routes>
</BrowserRouter>
```

Komponenta *BrowserRouter* slouží jako nadřazená komponenta, která obaluje všechny ostatní komponenty směrování. Vše, co je uvnitř této komponenty, je součástí směrovací logiky. Komponenta *Routes* umožňuje vyhledat první odpovídající trasu a použít ji pro vykreslení odpovídající komponenty. Každá komponenta *Route* zkontroluje aktuální URL a pokud se shoduje s definovanou cestou, vykreslí přiřazenou komponentu.

V ukázce výše lze vidět, že webová aplikace E2E Manager obsahuje šest různých cest, které jsou mapovány na odpovídající komponenty.

Homepage

Domovská stránka aplikace E2E Manager je přístupná na cestě “/”. Grafické rozhraní této stránky je následující:



Obr. 10: Domovská stránka aplikace

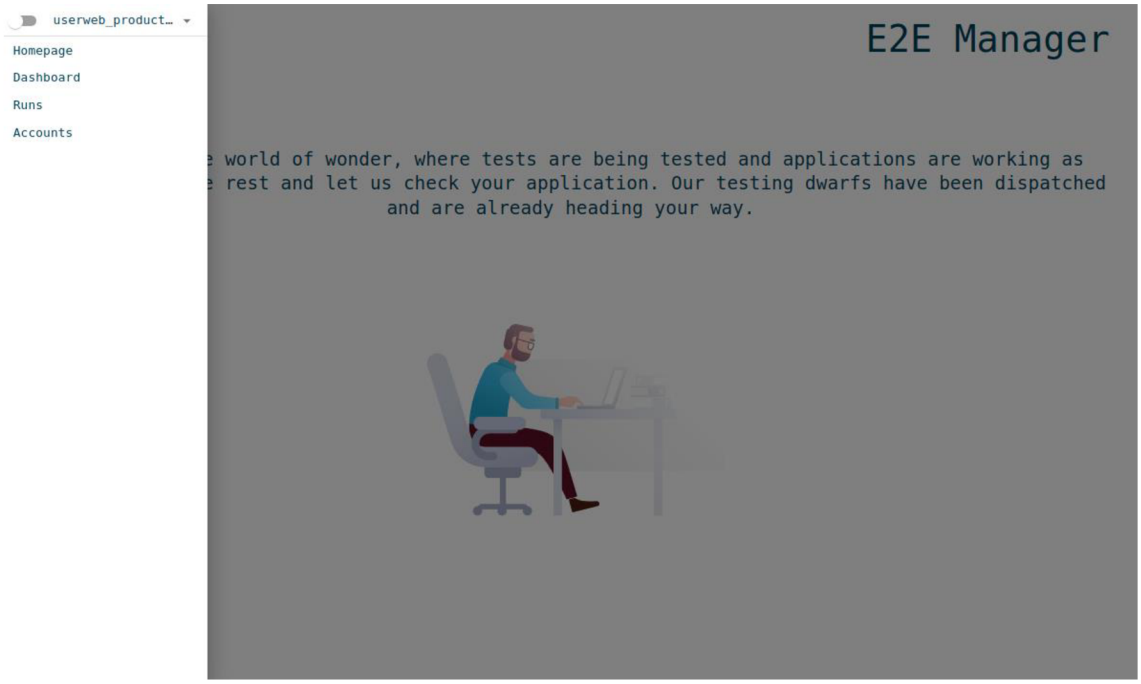
Domovská stránka, viz obr. 10, je reprezentována následující komponentou:

```
export const Intro = () => {
    return (
        <>
            <Title text="E2E Manager" />
            <BoxStyled pt={10}>
                <TypographyStyled variant="h5" ml={5} mr={5} mb={5}>
                    Welcome to the world of wonder, where tests are being tested and applications are working as
                    expected. Please rest and let us check your application. Our testing dwarfs have been dispatched and
                    are already heading your way.
                </TypographyStyled>
                <IntroSvg />
            </BoxStyled>
        </>
    );
};
```

V ukázce výše lze vidět, že komponenta *Intro* obsahuje pouze statický text, který je obalen komponentami z knihovny Material UI. Při prvním načtení webové aplikace E2E Manager se tato komponenta zobrazí, protože je přiřazena k domovské cestě "/".

Navigační menu

Na obr. 10 v levém horním rohu se nachází ikona “hamburger menu”. Po kliknutí na tuto ikonu se zobrazí navigace aplikace, viz obr. 11.

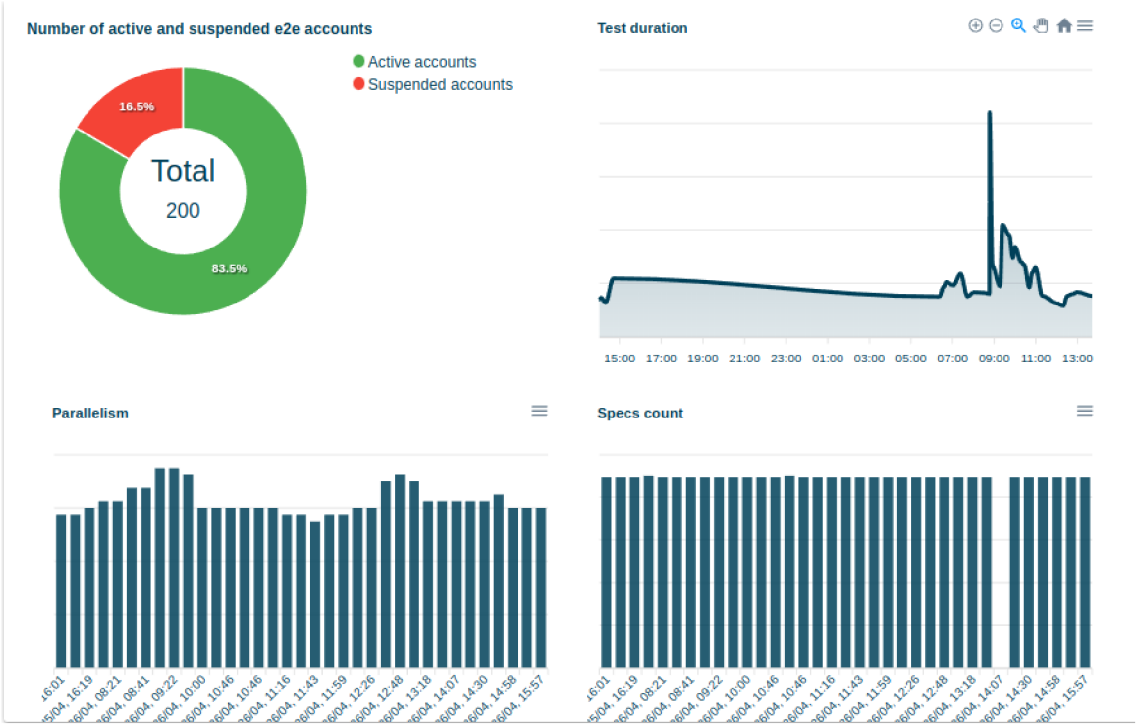


Obr. 11: Navigační menu aplikace

S pomocí této navigace lze procházet aplikací a navštěvovat uvedené cesty. Po navštívení cesty se vykreslí odpovídající část komponenty.

Dashboard

Komponenta *Dashboard* zobrazí veškerou informaci o běhů E2E testů a testovacích účtech ve formě grafů, viz obr. 12.



Obr. 12: Komponenta Dashboard

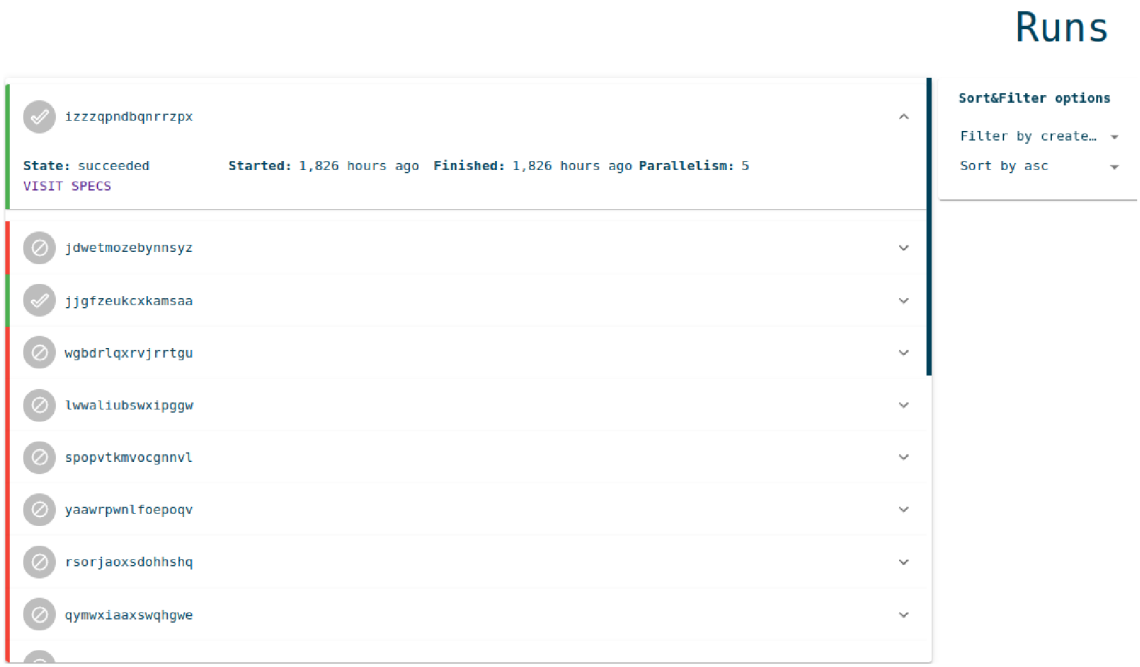
Pro vykreslení těchto grafů byla využita knihovna ApexCharts, která umožňuje vytváření různých typů grafů na základě poskytnutých dat. Komponenta *Dashboard* je navržena tak, že nejprve načítá data z databáze pomocí volání API koncových bodů. Jak již bylo zmíněno výše, pro načítání dat na straně klienta je používána knihovna React Query. Po úspěšném načtení komponenta předá tato data do dalších komponent, které slouží k definici a vykreslení jednotlivých grafů. Například komponenta pro zobrazení grafu aktivních a pozastavených testovacích účtů E2E testů vypadá následujícím způsobem:

```
export function DonutGraph({ accountsData }: Props) {
	const { numberOfEnabledUsers, numberOfDisabledUsers } = accountsData;
	const series = [numberOfEnabledUsers, numberOfDisabledUsers];

	const options = {
		series,
		labels: ['Active accounts', 'Suspended accounts'],
		title: {
			text: 'Number of active and suspended e2e accounts',
		},
		colors: [green[500], red[500]],
		legend: {
			floating: false,
			fontSize: '15px',
		},
		plotOptions: {
			pie: {
				donut: {
					labels: {
						show: true,
						total: {
							show: true,
						},
						value: {
							show: true,
							fontSize: '20px',
						},
					},
				},
			},
		},
	};

	return (
		<DivStyled>
			<Chart options={options} series={series} width={500}
height={400} type="donut" />
		</DivStyled>
	);
}
```

V uvedeném kódu lze vidět využití knihovny ApexCharts a popis "koblihového" grafu (tzv. donut chart) pomoci této knihovny.

Runs

Komponenta *Runs* zobrazí veškeré informace o výsledcích provedených E2E testů, viz obr. 13.

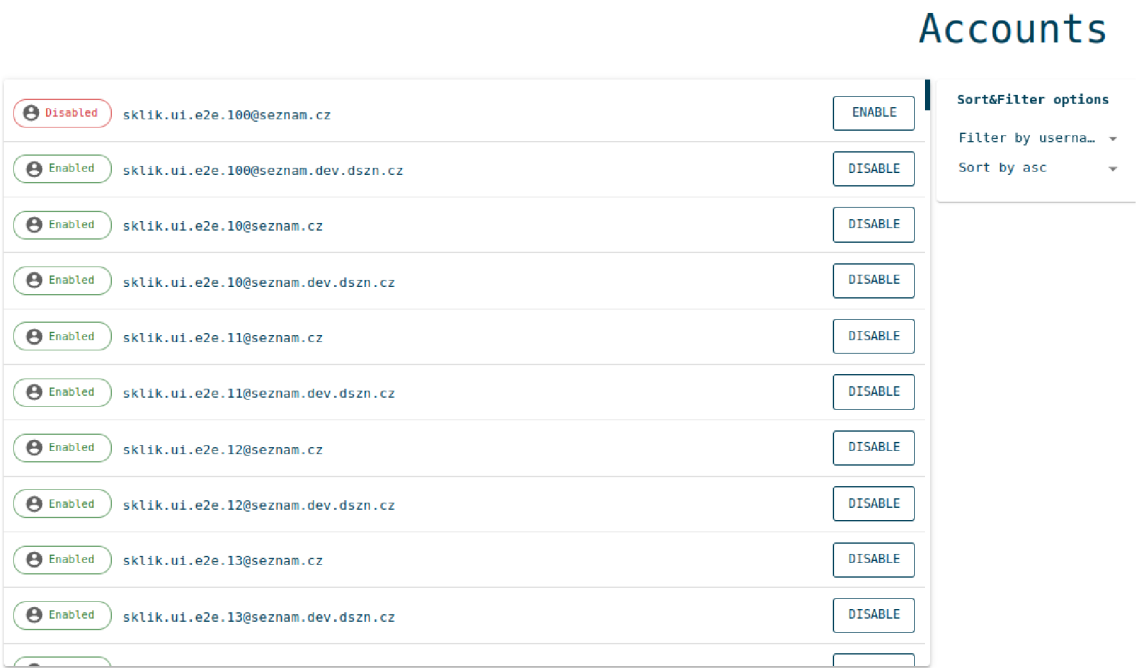


Obr. 13: Výsledky provedených E2E testů

Na obr. 13 je zobrazený výsledek provedených E2E testů a podrobné informace o každém testu pomocí komponenty *Runs*. Po kliknutí na nadpis "VISIT SPECS" se zobrazí stránka s cestou */specs/:runId*, která obsahuje specifikace konkrétně vybraného testu. Komponenta *Specs* má podobný vzhled jako komponenta *Runs*. Na obr. 13 vpravo nahoře je viditelný DOM element, který obsahuje možnosti filtrace a sortování výsledků testů podle různých parametrů.

Accounts

Komponenta *Accounts* slouží k vykreslení a správě informací o uživatelských účtech, které se používají pro provádění E2E testů, viz obr. 14.



Obr. 14: Testovací účty E2E testů

Na obr. 14 je zobrazena stránka “Accounts”, na které jsou zobrazena jména jednotlivých testovacích účtů spolu s informací o jejich aktivním nebo pozastaveném stavu. Kromě toho je u každého účtu umístěné tlačítko, které umožňuje změnit stav jeho aktivity. Na této stránce je také dostupná možnost filtrace a sortování účtů.

4 IMPLEMENTOVANÉ ČÁSTÍ

Každá stránka webové aplikace E2E Manager, s výjimkou domovské stránky, funguje následujícím způsobem: odpovídající komponenta nejprve načte data z backendu pomocí volání API koncových bodů a poté tato data vykreslí na webovou stránku. Příklad načítání dat v komponentě Accounts pomocí knihovny React Query vypadá takto:

```
const fetchAccounts = () =>
        client.v2.accounts
            .list({ limit: 1000, orderBy: order.by, orderDir: order.dir })
            .then((accounts_: V2ListAccountsResponse) =>
accounts_.items);

const { data: accountsData, error, isLoading } = useQuery(['accounts', order],
fetchAccounts);
```

Úryvek kódu výše obsahuje implementaci API volání (viz kapitola 3.2.1) pomoci třídy *Client*:

```
export default new Client({
    async dispatch(options) {
        const response = await fetch('https://e2e-manager.sklik' + options.path,
{
            method: options.method,
            headers: options.headers,
            ...(options.body && { body: options.body }),
        });

        return {
            statusCode: response.status,
            body: await response.arrayBuffer(),
            headers: {
                ['content-type']: 'application/octet-stream',
                ...Object.fromEntries([...response.headers.entries()]),
            },
        };
    },
});
```

V rámci načítání dat v komponentě *Accounts* se vytváří instance třídy *Client*, která slouží k typové kontrole vstupů a výstupů volání API a k přesměrování na příslušný koncový bod API. V kódu je vidět, že v podtřídě *v2* se postupně volá metoda *accounts* a

následně metoda *list*. Metoda *accounts* pouze vytváří instance nové třídy, která obsahuje definici volaných metod API. Metoda *list* je implementována následovně:

```
async list(req: V2ListAccountsRequest) {
        const { body } = await this.dispatch<V2ListAccountsResponse>({
            method: 'GET',
            urlTemplate: '/v2/accounts',
            req: {
                params: null,
                query: req,
                body: null,
            },
        });

        return body;
}
```

Podrobný popis struktury každé definované metody je uveden v kapitolách 3.2.1 - 3.2.3. V kódu výše je patrné, že do parametrů metody *dispatch* zděděné třídy je předán objekt obsahující HTTP metodu GET a cestu *v2/accounts*, která definuje API pro entitu "accounts", spolu s dodanými parametry z vnějšku. Tímto voláním je zajištěno provolání koncových bodů API. Metoda GET RESTful API pro entitu "accounts" v kontroleru vypadá následovně:

```
@Route('accounts')
export class AccountsController extends Controller {
	@Get()
	@SuccessResponse(200, 'Accounts retrieved')
	async list(
		@Query() offset?: V2ListAccountsRequest['offset'],
		@Query() limit?: V2ListAccountsRequest['limit'],
		@Query() orderBy?: V2ListAccountsRequest['orderBy'],
		@Query() orderDir?: V2ListAccountsRequest['orderDir'],
		@Query() id?: V2ListAccountsRequest['id'],
		@Query() isEnabled?: V2ListAccountsRequest['isEnabled'],
		@Query() username?: V2ListAccountsRequest['username'],
	): Promise<V2ListAccountsResponse> {
		const { totalCount, items } = await db.transaction((tx) => {
			return Account.load({
				tx,
				offset,
				limit,
				filter: makeFilter({
					id,
					environmentId,
					isEnabled,
					username,
				}),
				orderBy,
				orderDir,
			});
		});

		return {
			totalCount,
			items,
		};
	}
```

Jak bylo uvedeno v sekci 3.2.3, kontrolery pro API v2 jsou implementovány s využitím knihovny Tsoa. V kontroleru pro entitu “accounts” jsou kromě metody GET také definovány další HTTP metody. Příkladem je metoda POST, která má následující strukturu:

```
@Post()
@SuccessResponse(200, 'Accounts created')
async create(@Body() { items }: V2CreateAccountsRequest): Promise<V2CreateAccountsResponse> {
        const accounts = await db.transaction((tx) => Account.insert({ tx, rows: items }));

        return accounts;
}
```

Každá metoda v tomto kontroleru volá příslušný model Accounts a jeho funkce pro provedení potřebné operace. Například funkce *insert* v modelu Accounts má následující strukturu:

```
export function insert({ tx, rows }: InsertOptions): Promise<Account[]> {
    return tx
        .table('project_environment_accounts')
        .insert(
            rows.map((row) => ({
                allocationExpiresAt: null,
                isEnabled: true,
                ...row,
            })),
        )
        .returning('*');
}
```

V uvedeném kódu dochází k vkládání poskytnutých položek do databázové tabulky "project_environment_accounts" v rámci databáze *e2e-manager* pomocí Knex.js, který je již známý. Po provedení této operace se v tabulce vytvoří nový řádek, do kterého jsou vložena dodaná data do příslušných sloupců (viz obr. 9).

Všechna API pro E2E Manager komunikují s databází *e2e-manager* prostřednictvím SQL transakcí. Například kontroler pro aktualizaci stavu aktivity testovacího účtu má následující strukturu:

```
@Patch()
@SuccessResponse(200, 'Accounts updated')
async update(@Body() { items }: V2UpdateAccountsRequest): Promise<V2UpdateAccountsResponse> {
        const accounts = await db.transaction((tx) => Account.update({ tx, rows: items }));

        return addAllocations(new Date(), accounts);
}
```

V ukázce výše je definována metoda *update*, která vytváří SQL transakci pro databázi a volá metodu *update* v modelu *Account*. Tato metoda je použita pro změnu stavu aktivity testovacího účtu E2E testů na základě dodaných dat z vnějšku. Volání funkce *update* v komponentě *Accounts* je následující:

```
client.v2.accounts.update({ items: changedUser })
```

Samotná metoda *update* modelu *Account* vypadá následujícím způsobem:

```
export async function update({ tx, rows }: UpdateOptions) {
    const items = [];

    for (const account of rows) {
        const { id, ...data } = account;

        items.push(
            ...((await tx
                .table('project_environment_accounts')
                .update(data)
                .where({ id })
                .returning('*')) as Account[]),
        );
    }

    return items;
}
```

V kódu výše je vidět aktualizace dat v tabulce “project_environment_accounts” na základě dodaných požadavků. Výstupem této funkce je seznam aktualizovaných položek z této databázové tabulky.

Po načtení potřebných dat z databáze *e2e-manager* pomocí volání koncových bodů API, komponenta *Accounts* vykreslí obsah testovacích účtů E2E testů do webové stránky

(viz obr. 14). Kód pro vylistování položek s obsahem testovacích účtů E2E testů v komponentě *Accounts* vypadá následovně:

```
return (
        <>
            <Title text="Accounts" />
            <DivStyled>
                <MainContainer>
                    <ListItemStyled>
                        {accountsData?.map((account) => (
                            <Item account={account}
key={account.id} onClick={onClick} />
                        ))}
                    </ListItemStyled>
                </MainContainer>
            </DivStyled>
        </>
);
```

V kódu výše je každá načtená položka postupně předána do komponenty *Item*, která vykreslí data testovacích účtů E2E testů do DOM elementu a výsledek je zobrazen na stránce. Následuje úryvek kódu komponenty *Item*, která slouží k vykreslení jednotlivých položek z databáze do DOM elementu:

```
<DivStyled>
        <ChipStyled
                avatar={<AccountCircleIcon />}
                color={!isEnabled ? 'error' : 'success'}
                label={isEnabled ? 'Enabled' : 'Disabled'}
                size="medium"
                variant="outlined"
        />
        <ListItemTextStyled primary={username} />
        <Box>
                <ButtonStyled
                        color="inherit"
                        disabled={loading}
                        size="large"
                        variant="outlined"
                        onClick={() => showModal(showModalArgs)}
                >
                        {isEnabled ? 'Disable' : 'Enable'}
                </ButtonStyled>
                {loading && <CircularProgressStyled size={24} />}
        </Box>
</DivStyled>
```

V ostatních komponentách je použitý stejný způsob načítání a zobrazování dat z databáze *e2e-manager*. Následuje příklad volání koncových bodů API a načítání dat v komponentě *Runs*:

```
const fetchRuns = () =>
                client.v2.runs
                        .list({ limit: 20, orderBy: order.by, orderDir: order.dir })
                        .then((runs_: V2ListRunsResponse) => runs_.items);

const { data: runsData, error, isLoading } = useQuery(['runs', order], fetchRuns)
```

V kódu výše je provedeno volání metody *list* na serverové straně a výsledek této operace je uložen do proměnné *runsData*. Poté jsou načtená data z tabulky “runs” (viz obr. 9) v databázi *e2e-manager* vykreslena na webové stránce (viz obr. 13). V komponentě *Runs* je následující úryvek kódu pro vylistování položek s obsahem dat běhů E2E testů:

```
<Title text="Runs" />
                    <DivStyled>
                        <MainContainer>
                            <ListItemStyled>
                                {runsData?.map((run) => (
                                    <Item key={run.id} run={run} />
                                ))}
                            </ListItemStyled>
                        </MainContainer>
                    </DivStyled>
```

V kódu výše je vidět, že stejně jako v komponentě *Accounts*, i zde se každá položka s daty běhu E2E testu předává do komponenty *Item*, která vykreslí veškeré informace o tomto testu na stránce:

```
<AccordionStyled expanded={isToggle} onChange={toggle}>
    <AccordionSummary expandIcon={<ExpandMoreIcon />}>
        <Avatar>{renderCorrespondingIcon(run.state)}</Avatar>
        <NameStyled>{run.id}</NameStyled>
    </AccordionSummary>
    <AccordionDetailsStyled>
        <TitleStyled>State:</TitleStyled>
        <Typography>{run.state}</Typography>
        <TitleStyled>Started:</TitleStyled>
        <Typography>{run.createdAt}</Typography>
        <TitleStyled>Finished:</TitleStyled>
        <Typography>{run.finishedAt ? (run.finishedAt) : '---'}</Typography>
        <TitleStyled>Parallelism:</TitleStyled>
        <Typography>{run.parallelism}</Typography>
    </AccordionDetailsStyled>
</AccordionStyled>
```

Všechny použité API lze snadno číst a implementovat pomoci dokumentace Swagger, viz obr. 15.

Obr. 15: Swagger dokumentace, vytvářená pro zobrazení API E2E manager

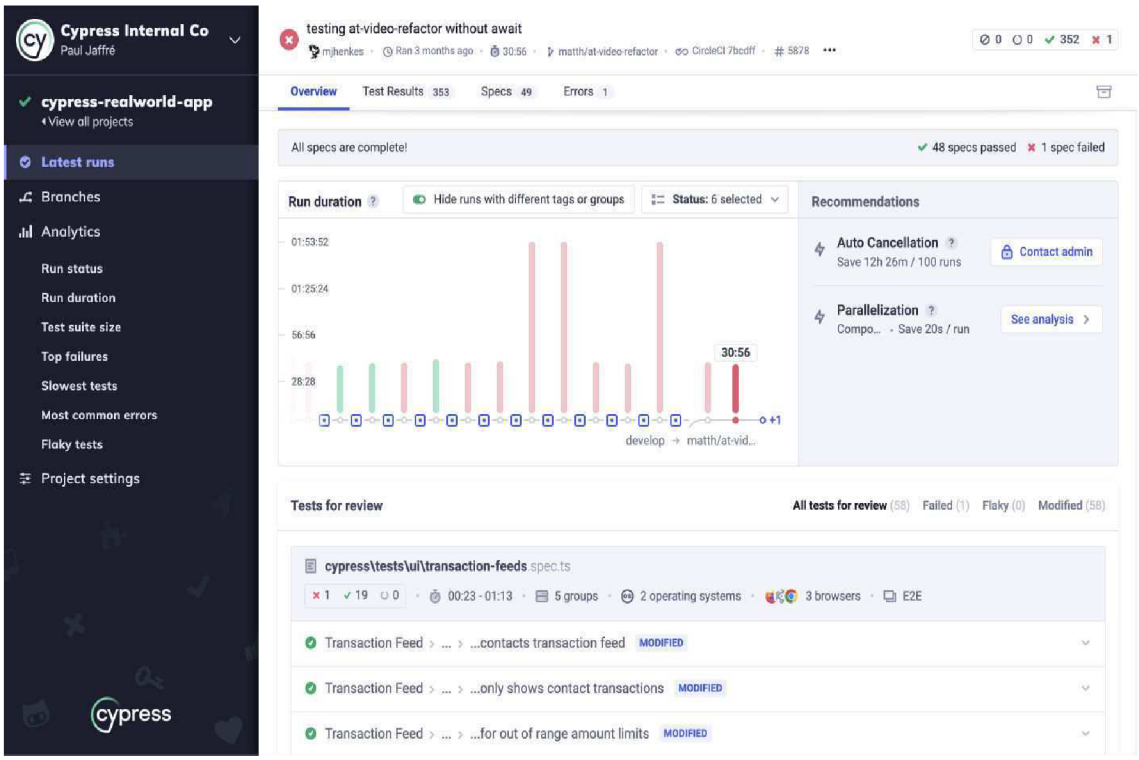
5 SROVNÁNÍ S JINÝMI ŘEŠENÍMI

Tato kapitola se věnuje popisu dalších dostupných řešení sledování E2E testů a porovnání interní webové aplikace E2E manager s těmito řešeními. Mezi tato řešení patří Cypress Cloud a Sorry Cypress (s placenou verzí Currents.dev jako součástí Sorry Cypress).

5.1 Cypress Cloud

Cypress Cloud je webová aplikace od společnosti Cypress, která je dostupná za poplatek. Poskytuje online přístup k zaznamenaným výsledkům E2E testů, analýze a diagnostice provedených testů, viz obr. 16. Mezi výhody Cypress Cloud patří [39, 41]:

- Spuštění testů Cypress v rámci kontinuální integrace (CI) a zaznamenání výsledků testů do Cypress Cloud pro rychlou zpětnou vazbu v případě selhání.
- Chytrá orchestrace umožňuje efektivní paralelní spuštění testů na více virtuálních strojích, což šetří čas potřebný k vykonání testů. Chytrá orchestrace využívá historických dat k optimalizaci pořadí spouštění testů a zajištění předvídatelného chování. V rámci chytré orchestrace je také zajištěno vyvážení zátěže na CI a preferování souborů s testy během paralelního spouštění. V případě prvního selhání testu je jeho spouštění automaticky zrušeno.
- Uložení a vizualizace průběhu testů: Cypress Cloud automaticky ukládá snímky obrazovky při selhání testů a umožňuje vizualizaci výsledků prostřednictvím videozáznamů. Tato funkce je užitečná při ladění problémů, které se vyskytnou v průběhu běhu testů. Když je detekováno nestabilní chování testu (tzv. flaky tests detection) v rámci integračního prostředí (CI), Cypress automaticky označí test s počtem předchozích nestabilních běhů zaznamenaných v historii. Historická data umožňují analyzovat četnost selhání testů, změny v definici testů, dobu trvání testů a výskyt nestabilních běhů.

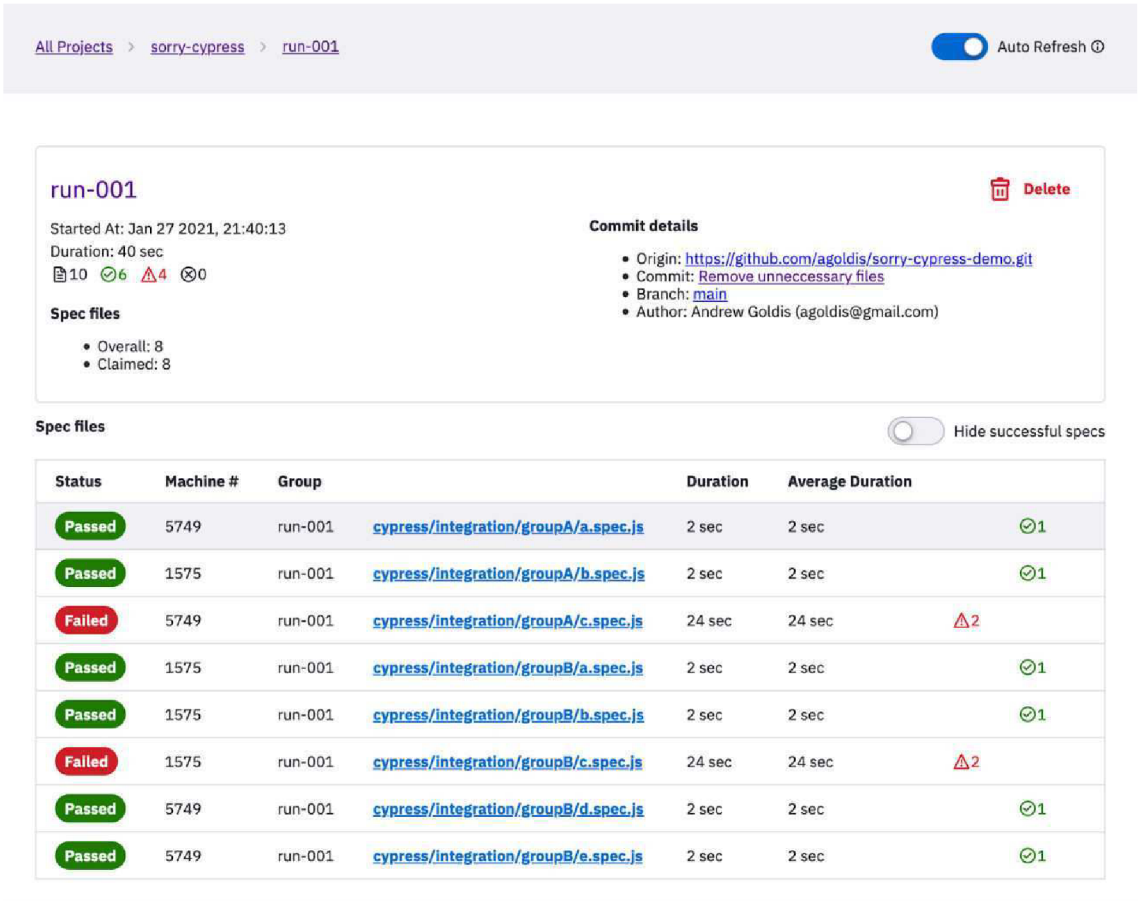


Obr. 16: Cypress Cloud Dashboard [39]

5.2 Sorry Cypress/Currents

Sorry Cypress je open source alternativou k Cypress Cloud a umožňuje provozování na vlastním serveru. Poskytuje neomezenou paralelizaci, záznam a ladění Cypress testů. Jelikož Sorry Cypress je projekt s otevřeným zdrojovým kódem a zdarma dostupný, některé funkce, jako inteligentní orchestrace, správa oprávnění, analýza a určité integrace, chybí. Pro doplnění těchto funkcí existuje placená verze Sorry Cypress nazvaná Currents. Currents je cloudové spravované řešení, které je podobné původnímu Cypress Dashboardu a disponuje téměř identickými funkcemi, viz obr. 17. Mezi funkce poskytované Currents patří [40, 41]:

- Paralelizace a inteligentní orchestrace založená na historických datech, což umožňuje ještě větší zkrácení doby provádění testů.
- Nástroje pro ladění, jako jsou snímky obrazovky a videa s možností zvýraznění zdrojového kódu.
- Historie běhů testů a souborů specifikací, která umožňuje sledování a analýzu předchozích testovacích sezení.
- Integrace a automatizace pomocí REST API, které umožňuje jednoduchou integraci s dalšími nástroji a procesy.
- Statistiky a zprávy poskytující informace o nejčastějších neúspěšných testech, nestabilních bězích, aktuálním stavu běhů, četnosti selhání a dalších metrikách.



Obr. 17: Sorry Cypress Dashboard [40]

5.3 Srovnání E2E manageru s uvedenými řešení

V této podkapitole bude uvedena tabulka, která porovnává E2E manager s výše zmíněnými existujícími řešeními v rámci funkcí E2E testování.

Tab. 1: Porovnání E2E manageru s existujícími řešeními [41]

Funkce	Cypress Cloud	Sorry Cypress	Currents	E2E manager
Základní paralelizace	☑	☑	☑	☑
Optimální paralelizace	☑	✗	☑	☑
Spouštění neúspěšných souborů specifikací jako první	☑	✗	☑	✗
Možnost zrušení běhů testů	☑	✗	☑	☑
CI Logy	☑	✗	☑	☑

Zaznamenávání snímků obrazovky a videí neúspěšných testů	☑	☑	☑	☑
Sledování chyb	☑	☑	☑	☑
Historie testovacích běhů	☑	✗	☑	☑
Výpis provedených akcí pro každý soubor se specifikacemi	✗	✗	☑	☑
Historie zdrojového kódu testu	☑	✗	✗	✗
Detekce nestabilních běhů testů (flaky tests detection)	☑	☑	☑	✗
Informace o prostředí a sestavení	☑	☑	☑	✗
Open API dokumentace	✗	✗	☑	☑
Analytika a statistika	☑	✗	☑	☑
Vlastní práce s uživatelskými účty	✗	✗	✗	☑
Bezplatné použití	✗	☑	✗	☑

6 ZÁVĚR

Cílem práce bylo navrhnout a implementovat určité části webového rozhraní pro sledování průběhu end-to-end testů. Webová aplikace E2E manager byla vytvořena v moderním frameworku React.js s využitím propojení koncových bodů API.

V úvodních dvou kapitolách se práce věnuje historii a základní teorii v oblasti webového vývoje. Významná pozornost je věnována využití jazyka TypeScript na straně klienta a Node.js na straně serveru. Dále jsou podrobně popsány základy testování webových aplikací s důrazem na E2E testy a frameworky pro jejich psaní.

V kapitole číslo 3 jsou podrobně popsány použité technologie při tvorbě E2E monitorovacího dashboardu. Navíc je zde uvedena architektura samotné webové aplikace E2E manager, včetně popisu API, databázového schématu a konceptu webu. Důraz je kladen na principy volání koncových bodů API a vykreslování načtených dat na webových stránkách.

Kapitola číslo 4 pojednává o tvorbě komponent webové aplikace a některých API. Podrobněji je popsána komponenta Accounts, která slouží k zobrazení a správě testovacích účtů, včetně souvisejícího API. Stejným způsobem je popsána komponenta Runs, která slouží k zobrazení podrobných informací o průběhu E2E testů.

Poslední kapitola se zaměřuje na popis existujících řešení sledování E2E testů a porovnání interní webové aplikace E2E manager s těmito řešeními. Výsledky porovnání jsou shrnuty ve formě tabulky.

Zadané cíle práce byly splněny. Webová aplikace E2E manager umožňuje nejen sledování průběhu end-to-end testů, ale také nabízí možnost správy E2E testovacích účtů. Možným vylepšením webové aplikace by mohlo být přidání funkcí, které jsou uvedeny v tabulce 1 a existují u popsaných komerčných řešení

SEZNAM POUŽITÉ LITERATURY

[1] Difference between Website and Web Application (Web App) [online], 2023. Mumnai: Matthew Martin [cit. 2023-05-26]. Dostupné z: https://www.guru99.com/difference-web-application-website.html

[2] From History of Web Application Development, © 2016. Devsaran [online]. Pradeep Saran [cit. 2023-05-25]. Dostupné z: https://www.devsaran.com/blog/history-web-application-development

[3] The World's First Web Site [online]. [cit. 2023-05-25]. Dostupné z: https://www.history.com/news/the-worlds-first-web-site

[4] History of the Web, © 2022. World Wide Web Foundation [online]. Washington DC [cit. 2023-05-25]. Dostupné z: https://webfoundation.org/about/vision/history-of-the-web/

[5] Web application, © 2023. Britannica [online]. London: Adam Volle [cit. 2023-05-25]. Dostupné z: https://www.britannica.com/topic/Web-application

[6] Apple in 1996, © 2023. Web Design Museum [online]. P. Kovář [cit. 2023-05-25]. Dostupné z: https://www.webdesignmuseum.org/gallery/apple-1996

[7] What is JavaScript?, © 1998–2023. MDN [online]. [cit. 2023-05-25]. Dostupné z: https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[8] The top programming languages, © 2023. GitHub [online]. [cit. 2023-05-25]. Dostupné z: https://octoverse.github.com/2022/top-programming-languages

[9] What's the difference between JavaScript and ECMAScript?, 2017. FreeCodeCamp [online]. [cit. 2023-05-25]. Dostupné z: https://www.freecodecamp.org/news/whats-the-difference-between-javascript-and-ecmascript-cba48c73a2b5/

[10] Stages of ECMAScript, © 2021. Proposals [online]. Saad Quadri [cit. 2023-05-25]. Dostupné z: https://www.proposals.es/stages

[11] What is TypeScript?, © 2012-2023. TypeScript [online]. Boston: Microsoft [cit. 2023-05-25]. Dostupné z: https://www.typescriptlang.org/

[12] A Deep Dive Into V8. AppSignal [online]. [cit. 2023-05-25]. Dostupné z: https://blog.appsignal.com/2020/07/01/a-deep-dive-into-v8.html

[13] What is Node.js, © 2009-2023. Simplilearn [online]. [cit. 2023-05-25]. Dostupné z: https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs

[14] Node vs Deno vs Bun, 2022. Patagonian [online]. ARGENTINA: Gonzalo Garro [cit. 2023-05-26]. Dostupné z: https://patagonian.com/blog/node-vs-deno-vs-bun-javascript-runtime/

[15] Deno [online], © 2023. [cit. 2023-05-26]. Dostupné z: https://deno.com/runtime

[16] Bun [online], 2023. Ashcon Partovi [cit. 2023-05-26]. Dostupné z: https://bun.sh/

[17] A Complete Guide to Web App Testing, 2022. HeadSpin [online]. Mousumi Rana [cit. 2023-05-26]. Dostupné z: https://www.headspin.io/blog/a-complete-guide-to-web-app-

testing#:~:text=Web%20app%20testing%20is%20a,environmental%20challenges%2C%20or%20traffic%20load

[18] The Four Levels of Software Testing, 2015. Segue Technologies [online]. Virginia: LaTonya Pearson [cit. 2023-05-26]. Dostupné z: https://www.seguetech.com/the-four-levels-of-software-testing/

[19] Integration Testing, 2023. GURU99 [online]. Mumnai: Thomas Hamilton [cit. 2023-05-26]. Dostupné z: https://www.guru99.com/integration-testing.html

[20] System testing vs End to End Testing, 2023. Testsigma [online]. Vakul Gotra [cit. 2023-05-26]. Dostupné z: https://testsigma.com/blog/system-testing-vs-end-to-end-testing/

[21] Unit, Integration, and End-to-End Testing, 2022. Twilio [online]. MICHELLE TRAN [cit. 2023-05-26]. Dostupné z: https://www.twilio.com/blog/unit-integration-end-to-end-testing-difference

[22] Which E2E testing framework to use for JS-based client applications?, 2020. Medium [online]. A. K. Rajendran [cit. 2023-05-26]. Dostupné z: https://aswinkumar4018.medium.com/which-e2e-testing-framework-to-use-for-js-based-client-applications-fbcac9aab680

[23] Cypress.io [online]. [cit. 2023-05-26]. Dostupné z: https://www.cypress.io/

[24] React (software) [online], 2023. Wikimedia Foundation [cit. 2023-05-26]. Dostupné z: https://en.wikipedia.org/wiki/React_(software)

[25] React [online], © 2023. Meta Platforms [cit. 2023-05-26]. Dostupné z: https://legacy.reactjs.org/

[26] React Quick Start [online], ©2023. MetaOpenSource [cit. 2023-05-26]. Dostupné z: https://react.dev/learn

[27] Material UI [online], © 2023. Material UI SAS [cit. 2023-05-26]. Dostupné z: https://mui.com/

[28] React Query Overview [online], © 2023. Tanner Linsley [cit. 2023-05-26]. Dostupné z: https://tanstack.com/query/v3/docs/react/overview

[29] Koa [online]. Apex Ping [cit. 2023-05-26]. Dostupné z: https://koajs.com/

[30] Knex.js [online]. [cit. 2023-05-26]. Dostupné z: https://knexjs.org/

[31] Tsoa [online]. [cit. 2023-05-26]. Dostupné z: https://tsoa-community.github.io/docs/

[32] Controller Pattern [online], 2022. M. E. Rana [cit. 2023-05-26]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/controller-pattern

[33] Design Patterns – MVC Pattern, © 2023. Tutorials Point [online]. [cit. 2023-05-26]. Dostupné z: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

[34] OpenAPI Specification, © 2023. Swagger [online]. SmartBear Software [cit. 2023-05-26]. Dostupné z: https://swagger.io/specification/

[35] What is a REST API? IBM [online]. [cit. 2023-05-26]. Dostupné z: https://www.ibm.com/topics/rest-apis

[36] The 5 essential HTTP methods in RESTful API development, 2021. TechTarget [online]. Tom Nolle [cit. 2023-05-26]. Dostupné z: https://www.techtarget.com/searchapparchitecture/tip/The-5-essential-HTTP-methods-in-RESTful-API-development

[37] What is react-router-dom? [online], 2023. Noida: GeeksforGeeks [cit. 2023-05-26]. Dostupné z: https://www.geeksforgeeks.org/what-is-react-router-dom/

[38] Feature Overview, 2023. React Router [online]. Remix Software [cit. 2023-05-26]. Dostupné z: https://reactrouter.com/en/main/start/overview

[39] Cypress Cloud, © 2023. Cypress.io [online]. [cit. 2023-05-26]. Dostupné z: https://docs.cypress.io/guides/cloud/introduction

[40] Open Source Cypress Dashboard [online], 2023. Andrew Goldis [cit. 2023-05-26]. Dostupné z: https://sorry-cypress.dev/

[41] Cypress Dashboard vs Currents - Ultimate Comparison Guide, 2022. Cypress [online]. Andrew Goldis [cit. 2023-05-26]. Dostupné z: https://currents.dev/posts/currents-vs-cypres