UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Nástroj pro automatizované testování API

Daniel Myška

Bakalářská práce
2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Daniel Myška**
Osobní číslo: **I21196**
Studijní program: **B0688A140009 Informační technologie**
Téma práce: **Nástroj pro automatizované testování API**
Zadávající katedra: **Katedra informačních technologií**

Zásady pro vypracování

Cílem této práce je vytvořit nástroj k automatickému testování API. Mezi základní funkci této aplikace bude patřit vytvoření testovacích scénářů, kde budou definované jejich kroky a vyhodnocen průběh. Všechny scénáře budou poté v aplikaci uloženy. Dále se bude ukládat historie výsledků daných scénářů, což umožní uživateli porovnávat jednotlivé výsledky. K aplikaci bude možné přistupovat přes webové rozhraní.
Aplikace bude vytvořena pomocí jazyka Kotlin a frameworku Spring Boot a dostupných knihoven pro jazyk Kotlin. Webové rozhraní bude vytvořeno pomocí jazyka TypeScript a dostupných frameworků a knihoven. Jako databáze pro tuto aplikaci bude sloužit PostgreSQL a bude spuštěna pomocí nástroje Docker.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

ECKEL, Bruce a Svetlana ISAKOVA, 2021. Atomic kotlin. Crested Butte, CO: Mindview LLC. ISBN 978-0--9818725-5-1.

WALLS, Craig, 2016. Spring Boot in action. Shelter Island: Manning. Java. ISBN 978-1-61729-254-5.

JIN, Brenda, Saurabh SAHNI a Amir SHEVAT, 2018. Designing Web APIs: building APIs that developers love. First edition. Beijing Boston Farnham: O'Reilly. ISBN 978-1-4920-2692-1.

Vedoucí bakalářské práce: **Ing. Jan Panuš, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. prosince 2023**
Termín odevzdání bakalářské práce: **10. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
Děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2024

Prohlášení autora

Prohlašuji:
Práci s názvem Nástroj pro automatizované testování API jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.
Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 17. 05. 2024 Daniel Myška

Poděkování

Děkuji Ing. Janu Panušovi Ph.D. za odborné vedení práce, poskytování rad a vstřícnost. Dále bych rád poděkoval své rodině a přítelkyni, za podporu při psaní této bakalářské práce.

Anotace

Tato bakalářská práce se věnuje tvorbě nástroje pro automatické testování API. V rámci této práce byla vytvořena serverová aplikace pro účeli testování API spolu s grafickým uživatelským rozhraním. Aplikace umožňuje uživateli nastavit testovací scénáře, kde v jednotlivých krocích nastaví dotaz na API a zpracování výsledků odpovědi dané API. Aplikace umožní uživateli grafické zobrazení výsledků jednotlivých scénářů. Veškeré výsledky, nastavení scénářů a jednotlivých kroků budou uloženy v databázi.

Klíčová slova

Testování, API, Kotlin, Spring boot, Docker, Automatizace testování

Title

Tool for automated API testing

Annotation

This bachelor's thesis focuses on the development of a tool for automatic API testing. Within this thesis, a server application was created for the purpose of API testing along with a graphical user interface. The application allows the user to set up testing scenarios where in each step they set up a query to the API and process the results of the response of that API. The application allows the user to graphically display the results of each scenario. All results, scenario settings and individual steps will be stored in the database.

Keywords

Testing, API, Kotlin, Spring boot, Docker, Test automation

Obsah

Seznam zkratek

API	Application Programming Interface
JVM	Java Virtual Machine
EJB	Enterprise JavaBean
POJO	Plain Old Java Objects
JPA	Jakarta Persistence
SQL	Structured Query Language
ORM	Object-Relational Mapping
NoSQL	Non-Relational
ACID	Atomicity, Consistency, Isolation and Durability
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
RPC	Remote Procedure Call

Seznam obrázků

Úvod

V současném světě softwarového vývoje je testování API klíčovým prvkem, který zajišťuje správnou funkčnost a integraci různých aplikací a systémů. Moje bakalářská práce se zaměřuje na vytvoření aplikace, která usnadňuje a automatizuje proces testování API. Hlavním problémem, který tato aplikace řeší, je zajištění spolehlivosti a funkčnosti API v různých scénářích, což je často náročné při manuálním testování. V teoretické části je popsaná problematika testování API a technologie použité k vytvoření aplikace. V praktické části je pak popsaná samotná implementace aplikace, vzhled uživatelského rozhraní a práce s databází.

Aplikace je navržena tak, aby umožňovala automatizované testování API pomocí předdefinovaných testovacích scénářů. Uživatelé mohou definovat jednotlivé testy, které se následně spouštějí a vyhodnocují automaticky. Tento přístup nejenže zvyšuje efektivitu testování, ale také snižuje riziko lidských chyb a zajišťuje konzistenci testů. Aplikace poskytuje přehledné výsledky testování, které uživateli umožňují rychle identifikovat a řešit případné problémy.

Pro implementaci serveru byl použit programovací jazyk Kotlin a framework Spring Boot. Pro tvorbu uživatelského rozhraní byl pak použit framework React s jazykem TypeScript. Serverová část komunikuje s grafickým uživatelským rozhraním pomocí GraphQL a pro ukládání dat je pak použita SQL databáze PostgreSQL. Všechny tři části aplikace jsou pak spuštěny pomocí docker-compose.yml.

Teoretická část

1 Testování API

1.1 Co to je API

API neboli „application programming interface“ je seznam specifik a pravidel, které umožňují různým aplikacím komunikaci mezi sebou. API se dělí na několika druhů podle toho, jaký typ softwaru je využívá a specifikuje. Může se tedy jednat například o API operačního systému, datové, vzdálené nebo webové. Pro účely mé práce je nejdůležitější web API. To je druh vzdálené API, která se používá pro přenos data a zpřístupnění funkcionality aplikace přes internet pomocí protokolu http. Web API lze rozdělit do čtyř kategorií podle dostupnosti. (1)

Otevřená API je veřejně dostupná pro všechny uživatele. Partnerská API je potom dostupná pouze pro vývojářem nebo společností, kteří vlastní danou API, autorizované uživatele. Vnitřní nebo také soukromá API slouží ke vnitřní komunikaci mezi aplikacemi organizace. Tyto API pak nejsou dostupné žádnému externímu uživateli. Složené API kombinují několik API od různých aplikacích do jednoho požadavku. Tento druh se hlavně používá pro komunikace s mikro službami. (1)

1.2 Proč testovat API

Testování API je klíčové pro zajištění kvality a spolehlivosti softwarových systémů. Ať už jde o jakýkoliv druh API, musí být zajištěno správné fungování jejich požadovaných funkcí a kvalita dat. Jelikož API zpřístupňuje data nebo funkcionalitu ostatním uživatelům, je nutné zajistit, aby uživatel vždy obdrželi očekávaná data nebo aby proběhlo správné provedení volané funkce. Nekvalitní až nefunkční API může pak zdržet vývoj aplikace, která závisí na jejich funkcích, nebo může odradit uživatele od používání dané API, což může u majitele dané API vést ke ztrátě zisku. (2)

2 Analýza požadavků

Nástroj by měl být spustitelný jak v lokálním prostředí vývojáře, tak v cloudové službě pro umožnění vzdáleného přístupu. Serverová část by pak měla být přístupná přes API. Ta by měla být navržena tak, aby bylo možné se serverem pracovat jak skrze uživatelské rozhraní, tak by měl být možný přístup k serveru z nástrojů třetích stran, například testovací „pipelines“.

Nástroj by měl být schopen poslat dotaz na testovanou API a uložit výsledek daného dotazu. Následně by měl zpracovat výsledek podle uživatelem nastavené konfigurace a uložit je. Jednotlivé výsledky by měly být uloženy s časovou stopou, aby bylo možné se podívat na ty předchozí výsledky. Veškerá práce a konfigurace by měla být proveditelná skrze grafické uživatelské rozhraní.

3 Použité Technologie

Tato kapitola obsahuje popis jednotlivých technologií a případné porovnání s alternativami a zdůvodnění proč byla vybrána.

3.1 Backend

Hlavní částí nástroje bude backendová, tedy serverová část aplikace. Tato část bude zodpovědná za zpracování všech klíčových funkcí. Jejím úkolem je například provádět dotazy na testované API, ukládat výsledky, validovat je, a spravovat testovací scénář a jejich kroky. Backendová část funguje jako jádro systému, které zajišťuje provádění automatických testů, uchovávání historie testování a umožňuje správu testovacích scénářů.

3.1.1 Porovnání programovacích jazyků Java a Kotlin

Java je jedním z nejpoužívanějších programovacích jazyků na světě. Byla poprvé vydána společností Sun Microsystems v roce 1995. (3) Java programy se kompilují do bajt kódu („bytecode“), který je pak interpretován pomocí JVM (Java virtual machine). Díky tomu může Java kód běžet na jakékoliv platformě, která podporuje JVM. Java podporuje „garbage collection“, tedy místo manuálního alokování a uvolňování paměti programátorem, aplikace následně sama alokuje paměť a následně ji uvolní. (4)

Kotlin byl vyvinutý společností JetBrains jako lepší verze jazyka Java. Java jsi nese spoustu neduhů ze svých raných fází vývoje. Vývojáři se rozhodli tyto nedostatky odstranit při vývoji jazyka Kotlin. Stejně jako Java i Kotlin se kompiluje do bajt kódu a poté běží nad JVM. To umožňuje interoperabilitu s jazykem Java, tedy kód napsaný v Javě může být bez problému použit v Kotlinu stejně tak většina knihoven a frameworků, které byli původně napsané pro Javu. (5)

Kotlin nabízí čistší a jednodušší syntaxi. Mezi hlavní rozdíly těchto jazyků patří inference typů, kdy V Kotlinu není nutné explicitně uvádět typ proměnné. Kompilátor sám zjistí typ z přiřazené hodnoty. Dále Kotlin podporuje funkce na nejvyšší úrovni, což eliminuje potřebu vkládat funkce do tříd. Také přináší rozšířené možnosti řízení průtoku, jako je výraz `when`. Díky integrované „null safety“ programátor ví, které hodnoty mohou být null, a obstarat jejich správné obsloužení. Další syntaktickým rozdílem je, pokud uvedeme lamda výraz jako poslední parametr v deklaraci funkce, můžeme jeho hodnotu předat mimo závorky volané

funkce, ale pomocí složených závorek na konci funkce. Lamda výrazy mají také základní proměnou `it,` při jejich volání není tedy potřeba deklarovat proměnnou. (6) (5)

Funkční rozdíly oproti jazyku Java. Kotlin podporuje přetěžování operátorů, to znamená že můžeme upravit chování operátorů pro jednotlivé objekty. V Kotlinu také můžeme vytvořit rozšiřující funkce („extension functions“) na již existujících třídách, což nám pak umožní volat dané funkce pomocí tečkové notace tedy `foo.funkce`. Chytré přetypování („smart casting“) je funkce Kotlinu, kdy programátor nemusí manuálně kontrolovat typ proměnné a přetypovávat ji, ale tento úkol je udělán automaticky kompilátorem. Hlavní výhodou jazyka Kotlin je pak podpora „coroutines“. (6)

3.1.2 Spring boot

Spring Boot je rozšíření frameworku Spring, kde Spring boot usnadňuje konfiguraci potřebnou k používání frameworku Spring. Spring byl vyvinut jako alternativa k JEE neboli Java Enterprise Edition. Spring nabídnul jednodušší vývoj oproti JEE, ta využívá Enterprise JavaBeans zkráceně EJBs. Spring zvolil přístup přes „dependency injection“ a EJBs nahradil za POJOs „Plain Old Java Objects“. To ale kde Spring získal na jednoduchosti kódových komponent, tak získal na složitosti a konfigurace. Spring se původně konfiguroval pomocí „Extensible Markup Language“ zkráceně XML, následně od verze 2.5 bylo možné konfigurovat přímo v Java kódu pomocí anotací. Verze 3.0 poté umožnila použít místo XML přímo jazyk Java.

Spring Boot tedy minimalizuje množství konfiguračního kódu potřebného k zahájení projektu, což zjednodušuje proces vývoje a nasazení aplikace. Spring Boot toho dosahuje pomocí automatizace konfigurace a nabízením výchozích nastavení pro projektovou strukturu a závislost („dependencies“), čímž umožňuje programátorům rychleji a efektivněji tvořit aplikace. Další významnou vlastností Spring Boot je vložený server, jako je Tomcat nebo Jetty, který eliminuje potřebu externího nástroje pro spuštění aplikace. (7)

3.1.3 JPA

Jakarta Persistence dříve známá jako Java Persistence API, zkráceně JPA, je standardní specifikace, která slouží pro práci s relačními databázemi. Hlavním využitím je vývoj databáze, která je přímo závislá na aplikaci a její podoba by se měla shodovat s podobou aplikace. K tomu JPA používá objektově-relační-mapování zkráceně ORM. To umožňuje s databázovými tabulkami pracovat jako s objekty, kde jednotlivé atributy třídy jsou pak

sloupci dané tabulky. Programátor tedy vytvoří třídu a pomocí anotací definuje podobu tabulky, které bude následně použita v databázi. JPA následně umožňuje vytvoření rozhraní pro danou tabulku, kde je pak možné definovat metody pro práci s danou tabulkou. Díky tomu je možné pracovat s databází bez potřeby psát čisté SQL. Samozřejmě v JPA existuje anotace, která umožní k dané metodě navázat čistý SQL dotaz napsaný programátorem. JPA také nabízí standardní rozhraní pro čtení, zápis a mazání záznamů pomocí „EntityManager“, kdy tyto základní metody jsou již implementované v základu. (8) (9)

JPA je nezávislé na konkrétní databázi, v případě potřeby je tedy možné tedy použít jiný databázový ovladač („driver“), a aplikace může pracovat s jinou databází, bez nutnosti měnit kód. (9)

3.1.4 Fasterxml jackson

FasterXML Jackson je open-source knihovna pro práci s JSON daty na JVM platformě, vytvořená a udržovaná společností FasterXML. Jackson poskytuje nástroje pro serializaci a deserializaci objektů do JSON formátu a naopak, což usnadňuje práci s JSON daty v programech. Knihovna podporuje různé formáty JSON, umožňuje práci s různými datovými typy a nabízí široké možnosti konfigurace. Jeho hlavními výhodami jsou jednoduché API, a možnosti rozšíření pomocí modulů, které umožňují přizpůsobit jeho funkčnost specifickým potřebám aplikace. (10)

3.2 API

Uživatelské rozhraní neboli „UI“ potřebuje rozhraní, kterým může komunikovat se serverovou částí aplikace, a právě k tomuto účelu slouží API, pro potřeby testovacího nástroje je to konkrétně webová API. Ta umožňuje uživatelskému rozhraní odesílat požadavky na server a přijímat od něj odpovědi. Existuje několik různých implementací webových API, z nichž každá má své specifické výhody a nevýhody. (1) (2)

3.2.1 REST

Representational state transfer zkráceně REST je definice standardu pro vývoj webových služeb. Za pomoci bez stavového protokolu a předdefinovaných bezstavových operacích REST, definuje šest specifikací, které když služba splní může být nazvána „RESTful Web Services“ neboli RWS.

Klient-server architektura je návrhový vzor, kdy je aplikace rozdělena na uživatelské rozhraní a datové úložiště. To umožňuje nezávislý vývoj jednotlivých částí, zlepšuje škálovatelnost a zjednodušuje serverovou část. Zároveň toto umožňuje přístup k serveru z různých zdrojů.

Požadavek on klienta na server musí být bezstavový, což znamená, že server neuchová žádné informace mezi jednotlivými požadavky klienta. Každý požadavek na server musí obsahovat všechny potřebné informace k vykonání dotazu. V případě potřeby zachování stavu, je buď tento stav udržován klientem, nebo je serverem přiřazená jiné službě například databázi, v případě autentizace.

Možnost ukládání do mezipaměti „Cacheability" je schopnost klienta dočasně ukládat výsledky požadavků na server. Při opakovaném dotazování se stejnými parametry může klient použít výsledek předchozího požadavku. Tím se následně eliminuje část potřebné komunikace mezi klientem a serverem a tím se tedy zlepšuje výkon. U ukládání do mezipaměti je důležité hlídat, jaké dotazy mohou mít „cacheable". Odpovědi by tedy měli uvést buď explicitně nebo implicitně jestli mohu být uloženy do mezipaměti. To by mělo zabránit tomu, aby klient nedostal zastaralá nebo nekompletní data.

Vrstvení systému slouží k tomu, aby klient nepoznal, jestli mluví přímo se serverem nebo s prostředníkem. Díky tomu nedojde, v případě umístění proxy nebo vyrovnávače „load balancer" mezi klienta a server, k ovlivnění komunikace mezi nimi.

Kód na vyžádání „Code on Demand" umožňuje rozšířit možnosti serveru o vlastní kód. Tato specifikace není striktně vyžadována.

Stejnorodý interface neboli „Uniform Interface" je specifikace, kdy by vývojář měl následovat podobný design všech APIs v rámci aplikace.

V rámci REST API je informace na serveru je považovaná za zdroj, ke kterému může klient přistoupit přes URIs „Uniform Resource Identifiers". REST pak používá http protokol pro komunikaci, kde jsou definované čtyři základní metody. Těmi jsou POST, GET, PUT a DELETE. (11)

3.2.2 GraphQL

GraphQL je dotazovací jazyk určený pro API vytvořený společností Facebook (dnes známé jako META) v roce 2012. Tento jazyk umožňuje klientovi v požadavku určit jaká data chce

získat od serveru. Tím se zabraňuje posílání zbytečných dat po síti takzvaný „over fetching“. Na rozdíl od RESTu, kde jsou přesně definované koncové body „endpointy“ a jaká přesně data vracejí. GraphQL má typicky jeden koncový bod, a klient specifikuje jaká data chce v dotazu. K tomu GraphQL využívá schéma. (12) (13)

Schéma se skládá z několika částí. První dvě základní části jsou typ „Query“ a typ „Mutation“. Zde se definují základní operace, které může GraphQL provádět. Kdy Query slouží pro operace na čtení dat, zatím co Mutation slouží k zapsání nebo úpravě dat a následnému přečtení dat. (14)

Syntaxe pro schéma následuje jednoduchou konvenci. V schématu můžeme pomocí klíčového slova type definovat objektové typy. Tyto typy pak musí obsahovat minimálně jedno pojmenované pole, které je buď skalárního typu `Int`, `Float`, `String` nebo `Boolean` či jiný objektový typ. Tyhle typy se pak používají jako návratové hodnoty. Pro definici vstupních hodnot se pak používá stejný princip jenom se místo klíčového slova `type` použije klíčové slovo `input`. (15)

GraphQL nabízí možnost introspekce, což znamená, že se klienti mohou dynamicky dotazovat na schéma API pro zjištění, jaké operace, typy a pole jsou dostupné. (16)

3.2.3 gRPC

gRPC je moderní open source framework pro vzdálené volání procedur „Remote Procedure Call“ zkráceně RPC, který byl původně vyvinutý společností Google. Využívá HTTP/2 jako transportní protokol, což mu umožňuje efektivně zpracovávat multiplexované streamy dat. Framework gRPC je navržen tak, aby podporoval vysoký výkon a škálovatelnost v mikro službách a distribuovaných systémech. To usnadňuje vývojářům práci v těchto systémech.

V gRPC se na serveru pomocí „protocol buffers“ zkráceně Protobuf souborů definuje, jaké metody mohou klienti volat, jaké jsou vstupní parametry a jak vypadá odpověď. Základní myšlenka tedy je, že klienti mohou volat metody serveru, jako kdyby to byl lokální objekt. Tento přístup značně zjednodušuje vývoj mikroslužeb.

Protobuf soubory jsou systém pro serializaci strukturovaných dat. Protobuf umožňuje definovat jednoduchou syntaxi pro strukturování dat. Tyto soubory jsou potom zkompilovány do zdrojového kódu v cílovém programovacím jazyce pomocí kompilátoru

„protoc“. Tento zdrojový kód pak obsahuje třídy nebo struktury s metodami pro serializaci, deserializaci a manipulaci s daty v daných strukturách. (17)

3.3 Databáze

Databáze je kolekce vzájemně propojených informací uložených v počítačovém systému. Databáze lze rozdělit na dvě základní kategorie SQL a NoSQL. Kdy NoSQL databáze se pak dělí na další poddruhy. SQL databáze se pak hlavně liší v implementaci a fungování některých vlastností. Všechny SQL databáze pak rozumí jazyku SQL „Structured Query Language“. Když to NoSQL databáze vzhledem ke své různorodosti nemají žádný společný dotazovací jazyk. (18)

3.3.1 PostgreSQL

Pro implementaci testovacího nástroje byla vybrána databáze PostgreSQL. PostgreSQL je open source relační databáze, původně vytvořena v roce 1986 jako součást projektu POSTGRES na University of California. PostgreSQL splňuje 170 ze 179 povinných specifikací pro SQL. Stejně tak PostgreSQL plně podporuje ACID. ACID neboli „Atomicity, Consistency, Isolation, Durability“, je sada pravidel pro zachování validity dat.

Ačkoliv je PostgreSQL skoro plně splňuje SQL standard, jsou zde stále rozdíly v syntaxi, kde některé operace používají jiný zápis, nebo upravují chování některých funkcích. Mezi hlavní výhody PostgreSQL patří velká škála datových typů s možností vytvořit vlastní datové typy. (19)

3.4 Front end

Front end neboli uživatelské rozhraní, je druhou důležitou částí tohoto nástroje. Přes uživatelské rozhraní, může uživatel interagovat se serverem a přistupovat k jeho funkcím. To vše přes jednoduché a srozumitelné rozhraní.

3.4.1.1 Desktopové vs Webové rozhraní

Desktopová aplikace běží nativně na operačním systému počítače. Taková aplikace musí být nejdříve nainstalována na uživatelův počítač. Desktopové aplikace jsou většinou vázané na operační systém, tudíž musí být buď separátně vyvinuty nebo modifikovány, aby mohli běžet na různých operačních systémech.

Webové aplikace pak používají webový prohlížeč, a může k nim být přistupováno z jakéhokoliv systému přes internet. Hlavní nevýhodou webových aplikacích je značně menší výkonnost oproti desktopovým aplikacím. Naopak webové aplikace mají výhodu větší dostupnosti a není je potřeba instalovat na uživatelský systém.

Pro účely vývoje testovacího nástroje, bude webová aplikace lepší variantou. Nástroj by měl být totiž spustitelný jak na lokálním prostředí, tak v cloudu. Proto dává větší smysl vytvořit uživatelské rozhraní jako webovou aplikaci. (20)

3.4.2 Porovnání jazyků JavaScript a TypeScript

JavaScript je skriptovací a programovací jazyk, který byl vytvořen pro potřeby vývoje dynamických webových aplikacích. Spolu s HTML a CSS se JavaScript stal de facto standardem pro vývoj webových aplikacích. JavaScript běží přímo v prohlížeči, prohlížeč interpretuje JavaScriptový kód například pomocí V8 „engine“ od společnosti Google nebo SpiderMonkey od společnosti Mozilla. (21)

JavaScript je dynamicky typovaný, to znamená že datový typ proměnné je určen až během průběhu programu. To potom vede k větší chybovosti a k takzvaným „runtime“ chybám, což jsou chyby, které se projeví až za běhu programu.

Tento problém se snaží vyřešit TypeScript. Ten byl vytvořený společností Microsoft jako rozšíření JavaScriptu o statické datové typy. TypeScript je superset JavaScriptu, tedy validní JavaScript kód je validní TypeScript kód. (22)

Pro účely vývoje testovacího nástroje byl zvolen jazyk TypeScript. Právě kvůli přidání statických datových typů.

3.4.3 React

React je jedním z nejpoužívanějších JavaScriptových frameworků vytvořený společností Facebook (dnes META) v roce 2013. React funguje na principu komponent. Každá komponenta si uchovává vlastní stav, a informace o svém stavu pak může předávat potomkům. React pracuje s virtuálním DOM „Document Object Model“, když se změní stav komponenty, virtuální DOM se porovná se skutečným DOM stránky v prohlížeči a překreslí se pouze ta část, která se změnila. React přidává podporu JSX „JavaScript XML“, jedná se obdobu HTML, ale není to přímo HTML. Jednou z hlavních výhod JSX je kód vložený do hranatých závorek je JavaScript. (23)

3.4.4 MaterialUI

Material-UI je populární open-source knihovna pro vývoj uživatelských rozhraní v Reactu. Tato knihovna implementuje designový systém Material Design, který byl vyvinut společností Google. Material-UI poskytuje širokou škálu předem navržených komponent, jako jsou tlačítka, formuláře, dialogová okna, ikony a další prvky. (24)

3.4.5 Apollo

Apollo je open-source knihovna pro práci s GraphQL. Vytvořená společností Apollo GraphQL. Apollo poskytuje komplexní sadu nástrojů pro správu stavu dat, dotazování se pomocí query a mutation typů na GraphQL server. Apollo poskytuje klienta, který má v sobě funkce jako automatické „cechování" dat, aktualizace UI při změnách dat, a jednoduchou konfiguraci autentizace a chybové zpracování. Knihovna také nabízí různé React „hooky", jako `useQuery`, `useMutation`, `useLazyQuery` atd., které usnadňují práci s GraphQL přímo v komponentách Reactu. (25)

3.5 Docker

Docker je platforma pro vývoj, distribuci a provozování aplikací pomocí kontejnerů. Byla vytvořena v roce 2013 firmou Docker, Inc. Základní jednotkou je „Docker image", což je soubor, který obsahuje vše potřebné ke spuštění aplikace. Tím může například být kód, knihovny, „runtime" atd. Docker pak umožní jednotlivé soubory „image" pouštět. Těmto spuštěným souborům se pak říká kontejner. Kontejnery jsou izolované od hostitelského systému i od sebe navzájem, ale mohou sdílet jádro operačního systému. To je odlišuje od tradičních virtuálních strojů, které mají vlastní operační systém a jsou pomalejší a výkonnostně náročnější na provoz. Díky tomu je pak možné aplikace jednoduše pouštět na různých systémech, bez nutnosti instalovat knihovny či cokoliv nastavovat.

K vytvoření „Docker image" se používá „Dockerfile" což je skript, který obsahuje kroky, jako je instalace závislostí a kopírování souborů, příkaz pro spuštění aplikace atd. (26)

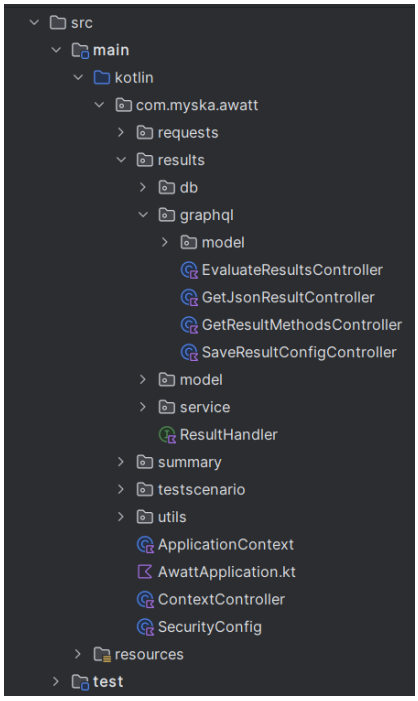
Praktická část

Testovací nástroj je rozdělen do 3 částí: databáze, serverová část a uživatelské rozhraní. Všechny tři části jsou spolu spustitelné pomocí docker-compose. Serverová část je závislá na databázi a nemůže bez ní být spuštěna. Uživatelské rozhraní může běžet samostatně bez serveru, ale nemá zpřístupněné veškeré funkce, které vyžadují výsledky ze serveru. Uživatelské rozhraní komunikuje se serverem přes GraphQL API. Server využívá pro práci s databází JPA.

Samotné testování je pak rozděleno do několika kroků. Nejdříve uživatel vytvoří testovací scénář. U toho si zvolí jméno a popis toho, co by měl testovací scénář dělat. V testovacím scénáři následně může přidat jednotlivé kroky. Krok bude mít v sobě specifikovanou URL adresu na kterou se bude dotazovat. Poté je zde výběr metody a možnost přidaní obsahu, který může dotaz poslat na danou URL. Dále v sobě bude mít daný krok konfiguraci, podle které se vyhodnotí výsledek daného dotazu. Testovací nástroj budu do databáze ukládat jednotlivé scénáře, kroky, výsledky dotazů a výsledky vyhodnocení dotazu podle konfigurace.

4 Architektura serveru

4.1 Rozdělení kódu do balíčků



Obrázek 1: Struktura Kotlin projektu

V Kotlinu stejně jako v Javě se kód rozděluje do balíčků „package“. V každý balíček obsahuje třídy a rozhraní, které zajišťují funkcionalitu aplikace. Každý balíček je pak rozdělen na další balíčky o obecném rozpoložení. Balíček „db“ obsahoval třídy pro reprezentaci tabulky a rozhraní implementující metody pro práci s danou tabulkou. V balíčku „graphql“ jsou třídy, které obsahují metody z GraphQL schématu. V balíčku „model“ jsou datové třídy, které vytvářejí datové objekty používané v kódu. Jako poslední je balíček „service“. V tomto balíčku jsou pak třídy a rozhraní, které implementují logiku aplikace a vykonávají jednotlivé funkce. V případě potřeby jsou v jednotlivých balíčcích další podbalíčky, které seskupují určitou funkcionalitu.

4.2 Práce s databází

JPA mně umožnilo definovat jednotlivé tabulky databáze jako třídy v Kotlinu. Pomocí anotací `Entity` a `Table`. `Entity` anotace, specifikuje že daná třída může být uložená do databáze a `Table` anotace nám umožní nastavit jméno tabulky, pod kterým bude entita uložená do databáze. Jednotlivé atributy dané třídy pak reprezentují sloupce v dané tabulce. Pomocí anotace `Id` určíme že daný atribut bude použit jako primární klíč. JPA automaticky

přiřadí jméno a datový typ sloupce podle jména a datového typu atributu. Pokud je potřeba bližší specifikace nebo úprava, může být použita anotace `Column`.

Pro veškerá id v rámci aplikace je použito `UUID`. O samotné přidělení `UUID` se pak stará databáze. V Kotlinu se toto nastaví pomocí anotace `GeneratedValue` s nastavenou hodnotou `strategy` na `GenerationType.UUID`. Všechny `id` jsou pak pomocí anotace `Column` nastavené, že nejsou „nullable“ tudíž nepřijímají hodnotu `null`, jako validní.

Pro každou entitu je pak vytvořeno rozhraní, které implementuje dodatečné metody pro práci s tabulkou.

```
interface EvalResultEntityRepository : JpaRepository<EvalResultEntity,
UUID>
```

Toto rozhraní je rozšířeno o `JpaRepository`, kde první argument určuje, pro jakou entitu je rozhraní implementováno. Druhý argument určuje, jaký datový typ je pak použitý pro primární klíč.

Test Scenario table

```
@Entity
@Table(name = "test_scenario")
class TestScenario(
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", nullable = false)
    val id: UUID? = null,
    var name: String,
    @Column(columnDefinition = "TEXT") var description: String?,
)
```

Tato tabulka slouží k uložení informací o testovacím scénáři. V této tabulce máme sloupce `id` typu `UUID`, sloupe `name` a `description`. Atributy `id` je nastavený jak „nullable“ se základní hodnotou `null`. To je kvůli tomu že Kotlin vyžaduje při vytváření objektu v konstruktoru uvést hodnoty pro všechny atributy třídy. Není to však vyžadováno pro atributy, které mají základní hodnotu.

Test Case table

```
@Entity
@Table(name = "test_case")
class TestCase(
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", nullable = false)
    val id: UUID? = null,
    @ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "scenario_id")
val scenario: TestScenario,
    var name: String,
    @Column(columnDefinition = "TEXT") var description: String?,
    @Column(columnDefinition = "TEXT") var url: String,
    var method: String,
    @Column(columnDefinition = "TEXT") var payload: String?,
    @Column(columnDefinition = "TEXT") var resultConfig: String,
    @Column(nullable = false) var caseOrder: Int? = 0,
    @ElementCollection
    @CollectionTable(name = "test_case_headers", joinColumns =
[JoinColumn(name = "test_case_id")])
    @MapKeyColumn(name = "header_name")
    @Column(name = "header_value")
    var headers: Map<String, String> = emptyMap()
)
```

Tato tabulka slouží k ukládání jednotlivých kroků scénáře. Zde atribut `scenario` odkazuje na třídu `TestScenario`. To říká JPA aby vytvořilo relaci mezi entitami. Anotace `ManyToOne` specifikuje, že vztah mezi entitami je takový, že k jednomu `TestScenario` patří mnoho `TestCase`. Tedy že `TestCase` může mít pouze jeden `TestScenario`. `FetchType.LAZY` určuje, že entita má být načtená z databáze až v okamžiku, kdy je skutečně potřeba. Anotace `JoinColumn` pak říká, že jméno sloupce má být „`scenario_id`“. V tabulce pak jsou ještě sloupce pro jméno, popis, url, metodu a payload. Payload jsou data, která se mají poslat s dotazem na URL. V kontextu REST API by to bylo „requestBody“. Sloupec `resultConfig` má pak v sobě uloženou konfiguraci pro zpracování výsledku dotazu. Kromě toho je zde atribut headers, který je anotován `ElementCollection`. Tato anotace říká JPA, že jde o kolekci základních typů, které by měly být mapovány jako samostatná tabulka. Anotace `CollectionTable` s parametrem „`name`“ "test_case_headers" specifikuje název tabulky, kde budou hlavičky uložené, a „`joinColumns`“ s hodnotou „test_case_id“ určuje název sloupce, který slouží jako cizí klíč pro propojení s tabulkou `TestCase`. Anotace `MapKeyColumn` s hodnotu parametru „`name`“ „header_name“ označuje, že klíče mapy (jména hlaviček) budou uloženy ve sloupci

„header_name“. `Column` s hodnotou parametru „`name`“ „header_value“ označuje sloupec, kde budou uloženy hodnoty mapy (hodnoty hlaviček).

Eval results table

```
@Entity
@Table(name = "eval_results")
class EvalResultEntity (
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", nullable = false)
    val id: UUID? = null,
    @ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "case_id") val
case: TestCase,
    val date: Date,
    @Column(columnDefinition = "TEXT") val result: String,
    @Column(columnDefinition = "TEXT") val error: String? = ""
)
```

Tato třída specifikuje tabulku „eval_results“. Ta slouží k ukládání výsledku po vyhodnocení podle konfigurace. Tabulka je spojená s `TestCase` proto, aby bylo možné určit k jakému kroku scénáře výsledek patří. Tabulky jsou spojeny ve vztahu `ManyToOne`. Tedy jeden `TestCase` může mít mnoho `EvalResultEntity`. V případě, že se při zpracování dotazu vyskytla chyba, bude uložená v sloupci error.

Responses table

```
@Entity
@Table(name = "responses")
class ResponseEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", nullable = false)
    val id: UUID? = null,
    @Column(columnDefinition = "TEXT") val response: String,
    val date: Date,
    @ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "case_id") val
case: TestCase
)
```

Tabulka pro ukládání odpovědí z testovaných API je pak definovaná pomocí třídy `ResponseEntity`. Zde je uložena odpověď ve sloupci „`response`“, která má dodatečnou definici jako `TEXT`. To říká databázi, že má sloupec datový typ `TEXT`, ten slouží pro ukládání velkého textu. `ResponseEntity` je pak spojená vztahem `ManyToOne` k `TestCase`.

4.3 Implementace GraphQL serveru

Pro implementaci GraphQL serveru nabízí Spring boot „starter“. Tento starter v sobě obsahuje základní konfiguraci, třídy a anotace pro práci s GraphQL. Součástí tohoto „starteru“ je Graphiql. To je webové rozhraní, které umožňuje pracovat s GraphQL API. Tento nástroj umožňuje psát dotazy na GraphQL API a zobrazovat výsledky daných dotazů. To velice usnadňuje vývoj, jelikož není potřeba vytvářet ihned klienta, který by GraphQL API používal, a není potřeba žádného nástroje třetí strany. Při vývoji lze tedy rovnou vyzkoušet, zda daná API funguje.

4.3.1 Definice GraphQL schématu

Aby bylo možné GraphQL používat musí se nejdříve definovat schéma. Při vytváření Spring boot projektu s GraphQL, je hned ve složce „resources“ pod složka graphql. Do této složky se pak vloží soubor s názvem „schema.graphqls“. Je důležité zachovat jméno souboru, protože jinak Spring tento soubor nenajde, pokud nezměníte konfiguraci.

```
type Query {
    getContext: context!
    executeTestCase(param: requestParameters): ResponseResult!
    executeTestScenario(param: requestParameters): [ResponseResult]!
    getScenarioResults(scenarioId: String!): ScenarioResultDto
    getCaseResults(caseId: String!): CaseResultDto
    getCaseResultForDate(caseId: String!, date: String!): CaseResultDto
    getScalarResultMethods: [String]!
    getArrayResultMethods: [String]!
    getObjectResultMethods: [String]!
    runEvaluation(scenarioId: String!): ScenarioResultDto
    getAllRunsDate(caseId: String!): [String]!
    getAllScenarios: [testScenarioDto]
    getAllCases(scenarioId: String!): [testCaseDto]
    getCase(caseId: String!): testCaseDto
    getScenario(scenarioId: String!): testScenarioDto
    getSummary: SummaryDto
    getRunningScenarios: [String]
}
```

V aplikaci je definováno 17 query typů. Ty se navážou v kódu na metody, které se zavolají, pokud přijde dotaz na dané query. Většina query typů slouží k získání dat ze serveru. Například getAllScenarios, vrátí pole všech scénářů uložených v databázi. Všechny query typy, které pouze získávají data ze serveru začínají prefixem „get“. Pokud query spouští nějakou další akci, jako například že říká serveru, že má zpracovat testovací scénář. Tyhle query typy začínají prefixem „run“ nebo „execute“. V schématu si můžeme všimnout, že

kromě skalárních typů jako `String`, tak jsou zde použity i vlastní typy pro vstup a výstup. Ty jsou rovněž definované ve schématu například takto:

```
type testScenarioDto {
    id: String,
    name: String!,
    description: String,
    cases: [testCaseDto]
}
```

Zde je definice výstupního datového objektu pro testovací scénář. Tento objekt obsahuje skalární pole typu `String` pro id, jméno a popis. V hranatých závorkách, ty v schématu určují pole, se nachází další objektový datový typ pro `TestCase`. Ve schématu je takto možné řetězit jak skalární, tak další objektové datové typy pro vytvoření komplexnějších typů.

```
type Mutation {
    saveTestScenario(args: saveTestScenarioArguments!): saveScenarioDto
    saveResultConfig(args: saveResultConfigArguments!): Boolean
    addCaseToScenario(scenarioId: String!, args: testCaseArguments!):
Boolean
    saveTestCase(caseId: String!, args: testCaseArguments!): Boolean
    deleteCase(caseId: String!): Boolean
    deleteScenario(scenarioId: String!): Boolean
}
```

V schématu se nachází definice pouze pro šest mutací. Stejně jako query typy tak i mutace mají pak navázanou metodu v kódu.

4.3.2 Metody pro obsluhu GraphQL dotazů

V kódu se pak používají anotace pro spojení metod s GraphQL schématem. Anotace `QueryMapping` slouží ke spojení query typu s metodou. `MutationMapping` pak spojuje metody s mutation typem ze schématu. Tyto metody musí být definované ve třídě s anotací Controller. Ta říká Springu, že daná třída slouží k obsluze http dotazů.

Metody pro obsluhu dotazů potom vypadají takto:

```
@QueryMapping("getAllCases")
fun getAllCasesByScenarioId(@Argument scenarioId: String):
List<TestCaseDto> {
    try {
        return
service.getAllCasesByScenarioId(UUID.fromString(scenarioId))
    } catch (e: Exception) {
        throw GraphqlQueryError(e.message ?: "Error while getting all
test cases")}
}
```

V závorkách u anotace `QueryMapping` je textová hodnota „getAllCases“, která říká, že daná metoda slouží pro obsluhu GraphQL Query typu „getAllCases“. Tato hodnota je zde explicitně uvedena, protože by bez ní Spring použil jméno funkce k přiřazení k typu v GraphQL schématu. Takto se může jméno funkce lišit oproti tomu co je ve schématu. Dále pak anotace `Argument` určuje že vstupní parametr metody se získá přímo z dotazu. Zde už není explicitně uvedené jméno, tudíž se musí jméno parametru shodovat se jménem uvedených v schématu. Stejně tak se návratový typ musí shodovat jak jménem tak strukturou tomu co je uvedené ve schématu. Co se týče `List`, v schématu je jako návratový typ seznam. V Kotlinu tedy můžeme použít jakoukoliv kolekci, pro návratový typ. V tomto případě `List`, protože se jedná o nejběžněji používanou kolekci. Pokud používáme složitější struktury ať už jako vstupní nebo návratové hodnoty, musí být zajištěno, že struktura daného objektu v Kotlinu bude odpovídat struktuře uvedené ve schématu. Zde je příklad `TestCaseDto`, které bylo použito jako návratový typ v metodě.

```
data class TestCaseDto(
    val id: String? = null,
    val name: String,
    val description: String,
    val url: String,
    val method: String,
    val payload: String? = null,
    val resultConfig: String,
    val order: Int,
    val headers: List<Headers>,
)
```

Definice ve schématu:

```
type testCaseDto {
    id: String,
    name: String!,
    description: String,
    url: String!,
    method: String,
    payload: String,
    resultConfig: String!,
    order: Int,
    headers: [headers]
}
```

Z ukázky je vidět, že datová třída v Kotlinu má stejnou strukturu jako typ definovaný ve schématu. Hlavní jsou zde jméno a datový typ. Spring totiž při konstrukci objektu použije hodnoty z polí dotazu se stejným jménem. Pokud by se nějaké jméno neshodovalo nebo mělo jiný datový typ, či dokonce chybělo, Spring by pak nedokázal vytvořit daný objekt z dotazu a vyhodil by chybu, což by vedlo ke selhání dotazu.

4.4 Implementace

4.4.1 Testovací kroky

Testovací kroky jsou základní složky testovacího nástroje. Zde je definovaná URL testované API. Dále je zde definovaná metoda dotazu a popřípadně data, která se mají poslat s dotazem. Následně je zde uložená konfigurace pro zpracování jednotlivých výsledků.

4.4.2 Testovací scénáře

Testovací scénář je hlavní složkou tohoto testovacího nástroje. Zde jsou uvedeny jednotlivé kroky, jméno a popis daného scénáře. Scénáře jsou pak uloženy v tabulce „test_scenario“, která je definovaná ve třídě `TestScenario`.

4.4.3 Poslání dotazu na testovanou API

K dotazování se na testovanou API slouží třída `RequestService`. Hlavní část je metoda `executeRequest(case: TestCase)`. Tato metoda pomocí standartní balíčku java.net vytvoří dotaz na požadovanou URL. Vstupním parametrem je `TestCase`, což je entita uložená v databázi reprezentující krok v scénáři.

Metoda může provádět dotazy na REST API pomocí http protokolu. Na začátku metody se vytvoří objekt URL z URL adresy zadané v testovacím kroku `TestCase`. Následuje nastavení HTTP metody (GET, POST, PUT apod.), která je specifikována v `TestCase`. Po nastavení HTTP metody, následuje konfigurace hlaviček požadavku. Pokud metoda vyžaduje zaslání dat (POST, PUT, PATCH), metoda zapíše data ze vstupního parametru do `outputStream`, a nastaví proměnnou `doOutput` na hodnotu `true`.

Dále metoda inicializuje síťové spojení pomocí metody `connect` a zkontroluje HTTP odpověď. Pokud server odpoví chybovým kódem, zaznamená se chyba do logu a metoda vrátí null, což indikuje, že požadavek selhal. Pokud je odpověď úspěšná, metoda přečte data z `inputStream` a výsledek uloží do databáze pro další zpracování.

4.4.4 Konfigurace pro zpracování výsledků

Konfigurace má formát JSON a definuje, jak má nástroj zpracovat jednotlivé výsledky dotazů na testovanou API. Konfigurace se potom „serializuje“ a je uložená v databázi jako TEXT. Základní konfigurace vypadá potom takto.

```
"results" : [
  {
    "method": "Compare",
    "path": "city/name",
    "dataType": "STRING",
    "extendedDataType": null,
    "expectedValue": ["Town"]
  }
```

V poli „`results`“ se nachází jednotlivé konfigurace. Za použití JSON pole je zde možné uvést více jednotlivých konfigurací. Konfigurace pak nástroji říká, jakou metodu má použít pro zpracování výsledku. Ta je uvedena v položce „method“. Položka „path“ potom určuje cestu k výsledku. Následně položka „dataType“ určuje datový typ hodnoty co se očekává od odpovědi testované API. K tomu je zde položka „extendedDataType“. Ta se používá v případě, že by bylo třeba určit dodatečný datový typ. Kupříkladu pokud by očekávaný datový typ bylo JSON pole, a metoda by chtěla určit, jestli jsou všechny položky pole stejného datového typu, který uživatel očekává. Poslední položkou je pak „expectedValue“ jedná se o přesnou hodnotu, která se očekává v odpovědi testované API. Tato položka je typu JSON pole z důvodu, kdyby byla očekávaná hodnota JSON pole hodnot.

Pro práci s konfigurací se v nástroji používá třída `JsonResultBuilder`. Tato třída používá `objectMapper` z knihovny FasterXML Jackson. Ten sestaví z dané JSON konfigurace datový objekt `JsonResultConfig`.

```
data class JsonResultConfig(
    val metadata: JsonResultMetadata,
    val resultConfigs: List<JsonObjectResult>
)
```

Kde atribut „resultConfig“ je datový objekt s listem samotných konfiguracích.

```
data class JsonResult @JsonCreator constructor(
    @JsonProperty("method") val method: String,
    @JsonProperty("path") val path: String,
    @JsonProperty("dataType") val dataType: String,
    @JsonProperty("extendedDataType") val extendedDataType: String?,
    @JsonProperty("expectedValue") val expectedValue: List<String>
)
```

Anotace `JsonCreator` poté určuje, že se daný konstruktor by měl být použit při deserializaci JSON do objektu. Anotace `JsonProperty` pak určuje, jaké položky s JSON mají být přiřazené k jakým atributům. V třídě `JsonResultBuilder` je pak ještě použit:

```
objectMapper.typeFactory.constructCollectionType(List::class.java,
JsonResult::class.java)
```

To slouží pro vytvoření `JavaType`, který Jackson používá ke správné deserializaci JSON dat do specifického typu kolekce. V tomhle případě se jedná o Kotlin `List`.

4.4.5 Vyhodnocení výsledku dotazů z API

Zpracování výsledků dotazů z testovaných API podle konfigurace probíhá v třídě `JsonResultHandlerService`. Tato třída implementuje rozhraní `ResultHandler`, kde je definovaná metoda `checkResult` se vstupními parametry „response“ která reprezentuje odpověď z testované API a „resultConfig“. Ten představuje konfiguraci podle, které se má zpracovat daná „response“. Třída je anotovaná jako Service, což umožňuje Springu zařadit danou třídu jako „bean“. To znamená, že Spring spravuje jejich životní cyklus a závislosti. Tyto servisní „beany“ mohou být poté injektovány do jiných komponent pomocí anotace `Autowired` nebo konstruktorové injekce.

Samotné zpracovaní pak probíhá tak, že se nejdříve třídou `JsonResultBuilder` sestaví konfigurace z JSON objektu do datových tříd. Ty se následně začnou procházet jednotlivé konfigurace pomocí cyklu `foreach`. Nejdříve se získá hodnota z „response“ pomocí metody `getValueFromPath`. V této metodě se z konfigurace načte cesta k hodnotě. V případě JSON objektu má cesta formát „*cesta/k/hodnotě*“, kde je „/“ použit pro rozdělení jednotlivých částí JSON objektu.

```
{
  "address": {
    "city": "Sample City"
  }
}
```

Takže pokud bychom chtěli získat hodnotu položky „city“ cesta by vypadalo takto „*address/city“*. Potom co se získá hodnota na uvedené cestě, se zavolá metoda `checkValueFromField`. Ta má vstupní parametry hodnotu získanou z cesty, název JSON položky a konfiguraci pro zpracování výsledku. V metodě se podle typu hodnoty získá třída pro zpracování dané hodnoty. V nástroji jsou k dispozici tři takové to třídy podle typu JSON hodnoty. Pro hodnoty typu JSON pole je zde třída `ArrayValueHandler`. Pro JSON hodnoty typu objekt je zde třída `ObjectValueHandler`. Pro ostatní hodnoty je zde třída `ScalarValueHandler`. Každá tato třída pak implantuje rozhraní `JsonValueChecker`, které definuje metodu `checkValue`.

Po získání příslušné třídy pro zpracování hodnoty se tedy zavolá metoda `checkValue` dané třídy. Každá třída pak používá `ResultMethodFactory` pro získání třídy implementující metodu pro zpracování výsledku, která je uvedená v konfiguraci. Toho je docíleno tak, že jednotlivé metody jsou implementované v samostatných třídách. Tyto třídy jsou rozdělené do tří kategorií podle JSON typu. Tedy metody pro skalární hodnoty jako je `STRING`, `NUMBER`, `BOOLEAN` atd. poté pro JSON pole a JSON objekt. Třídy, které pak obsahují jednotlivé metody implementují buď rozhraní `ScalarResultMethod`, `ArrayResultMethod` a `ObjectResultMethod`. Tyto rozhraní pak implementují rozhraní `ResultMethod`, ve kterém je definovaná metoda execute. Tohle rozdělení je pak kvůli tomu, že v třídě `ResultMethodFactory` jsou pomocí anotace Autowired inicializovány tři kolekce obsahují jednotlivé třídy s implementací dané metody. Když je následně potřeba získat konkrétní metodu stačí zavolat metodu `getMethod`.

```
fun getMethod(clazz: KClass<*>, method: String): ResultMethod? {
    return when(clazz) {
        ScalarValueHandler::class -> getScalarMethod(method)
        ArrayValueHandler::class -> getArrayMethod(method)
        ObjectValueHandler::class -> getObjectMethod(method)
        else -> null
    }
}
```

Ta má jako vstupní parametr typ Kotlin třídy v proměnné `clazz`, a název požadované metody v proměnné `method`. Podle typu třídy se pak zavolá metoda, která projde danou kolekci a pokusí se najít požadovanou metodu. Pokud se název metody nebude shodovat, s žádnou metodou v kolekci vrátí `null`. Stejně tak je hodnota `null` vrácena, pokud se hodnota `clazz` neshoduje s možnostmi v metodě.

Jak již bylo řečeno, výše uvedené jednotlivé metody jsou implementovány ve třídě, která má jméno podle dané metody. Pokud je v konfiguraci metoda s názvem „`Compare`“ tak třída implementující tuto metodu bude nést název Compare. To ale vytváří problém, pokud budeme chtít mít metodu „`Compare`“ jak pro skalární JSON typy, tak i pro typ JSON pole. Proto jednotlivé třídy mají prefix podle toho, pro jaký jsou typ. Pro skalární metody budou mít implementující třídy název `ScalarCompare`, pro pole pak bude název `ArrayCompare` a pro JSON objekty to pak bude `ObjectCompare`.

Pomocí GraphQL je pak možné získat jednotlivá jména pro dostupné metody.

```
getScalarResultMethods: [String]!
getArrayResultMethods: [String]!
getObjectResultMethods: [String]!
```

Ve schématu definujeme query pro každý typ metod. V obslužné metodě pro každé query vrátíme seznam hodnot, který obsahuje jednotlivé názvy dostupných metod. Protože jméno pro metodu se bere ze jména třídy, která danou metodu implementuje, je předtím, než jsou jména vrácena z GraphQL dotazu, je od nich odstraněn prefix (scalar, array, object).

Tento systém umožňuje velice jednoduché přidávání a odebírání metod. Kdy pouze stačí vytvořit třídu se správným prefixem a implementovat příslušné rozhraní. Metoda pak bude hned k dispozici bez nutnosti upravovat již vytvořený kód. Díky GraphQL query má pak klient vždy dostupné všechny metody, které se v nástroji používají.

4.4.6 Metody pro Skalární hodnoty

Třída `ScalarValueHandler` slouží k vyhodnocení skalárních typů dat získaných z JSON. Když metoda checkValue obdrží skalární hodnotu, jméno pole a konfiguraci výsledku. Nejprve se pomocí `factory.getMethod` pokusí získat odpovídající metodu pro vyhodnocení dle specifikované konfigurace výsledku. Pokud metoda není nalezena, vrátí se `JsonResultDto` s informací o neshodě. V opačném je zavolána případě metoda execute. Ta pak vrátí `Pair` hodnot, které indikují, zda došlo ke shodě mezi očekávanou a skutečnou hodnotou a v případě neshody vrátí vysvětlení.

Metoda Compare

```
resultConfig.expectedValue.firstOrNull()?.let { expected ->
    val result = value.asText().lowercase() == expected.lowercase()
    val explanation = "Values don't match"
    return Pair(
        first = result,
        second = if(result) "" else explanation
    )
}
return Pair(
    first = false,
    second = "No value for Expected value"
)
```

Metoda compare je určena pro porovnávání skalárních hodnot z JSON s očekávanými hodnotami specifikovanými uživatelem. Tato metoda pracuje tak, že nejprve zkontroluje, zda existuje očekávaná hodnota v `resultConfig`. Pokud je očekávaná hodnota

k dispozici, porovná ji s aktuální hodnotou z JSON, přičemž obě hodnoty jsou převedeny na malá písmena pro eliminaci rozdílů ve velikosti písmen.

Metoda CompareStrict

```
if(!value.nodeType.isType(resultConfig.dataType)){
    return Pair(
        first = false,
        second = "Data types doesn't match expected:
${resultConfig.dataType}, but was ${value.nodeType}"
    )
}
if(value.asText() != resultConfig.expectedValue.firstOrNull()) {
    return Pair(
        first = false,
        second = "Values doesn't match expected:
${resultConfig.expectedValue[0]}, but was ${value.asText()}"
    )
}
return Pair(
    first = true,
    second = ""
)
```

Metoda `ScalarCompareStrict` je implementována pro striktní porovnávání skalárních hodnot z JSON, kde se kromě samotné hodnoty kontroluje i datový typ. Tato metoda nejprve ověřuje, zda datový typ aktuální hodnoty odpovídá očekávanému datovému typu specifikovanému v `resultConfig`. Pokud typy nesouhlasí, metoda vrací `Pair` s hodnotou false a vysvětlením rozdílu mezi očekávaným a skutečným datovým typem. Následuje kontrola, zda textová reprezentace hodnoty přesně odpovídá očekávané hodnotě. Pokud se hodnoty neshodují, vrací se opět `Pair` s false a vysvětlením rozdílu mezi očekávanou a skutečnou hodnotou. V případě, že obě kontroly projdou, metoda vrací `Pair` s true a prázdným řetězcem, signalizující úspěšné porovnání.

Metoda IsType

```
val result = value.nodeType.isType(resultConfig.dataType)
val explanation = "Values weren't of same type"
return Pair (
    first = result,
    second = if(result) "" else explanation
)
```

Metoda `IsType` poskytuje funkčnost pro ověření, zda datový typ JSON hodnoty odpovídá očekávanému datovému typu definovanému v `resultConfig`. Tato metoda využívá pomocnou funkci isType, aby zkontrolovala shodu mezi typy. Pokud typy souhlasí, metoda

vrací `Pair` s hodnotou true a prázdným řetězcem, což indikuje úspěšnou shodu typů. V případě neshody mezi typy vrací false a textové vysvětlení.

Implementace rozšiřující metody `isType` pro `JsonNodeType`.

```
fun JsonNodeType.isType(type: String): Boolean = this.name.uppercase() ==
type.uppercase()
```

Metoda Matches

```
resultConfig.expectedValue.firstOrNull()?.let {
    val regex =  Regex(it)
    val result = regex.matches(value.asText())
    val explanation = "Value doesn't match regex $regex"
    return Pair (
        first = result,
        second = if(result) "" else explanation
    )
}
return Pair (
    first = false,
    second = "No value found for regex"
)
```

Metoda `matches` je určena pro kontrolu, zda textová hodnota z JSON uzlu odpovídá regulárnímu výrazu specifikovanému v očekávaných hodnotách `resultConfig`. Tato metoda nejprve vytvoří objekt `Regex` z první očekávané hodnoty definované v konfiguraci výsledku. Následně použije tento regulární výraz pro testování, zda textová reprezentace hodnoty `value` odpovídá regulárnímu výrazu. V případě neshody mezi hodnotou a regulárním výrazem vrací false a vysvětlení. Stejně tak pokud není v konfiguraci hodnota pro regulární výraz, vrací se hodnota false a vysvětlení.

Metoda NonNull

```
val result = value.nodeType != JsonNodeType.NULL
val explanation = "Value was null"
return Pair (
    first = result,
    second = if(result) "" else explanation
)
```

Metoda `nonNull` je určena pro kontrolu, zda JSON hodnota nemá hodnotu `null`. V metodě se provede porovnání typu hodnoty s `JsonNodeType.NULL`. Pokud hodnota není nulová, vrací metoda `Pair` s hodnotou true a prázdným řetězcem. V opačném případě, pokud je hodnota nulová, vrací false a vysvětlení, proč je výsledek invalidní.

Metoda NonNullOrEmpty

```
if(value.nodeType == JsonNodeType.NULL) {
    return Pair(first = false, second = "Value was null")
}
val result = value.asText() != ""
return Pair(
    first = result,
    second = if(result) "" else "Value was empty"
)
```

Metoda `nonNullOrEmpty` zajišťuje, že hodnota JSON není nulová a ani prázdný řetězec. V metodě se nejdříve kontroluje, zda je datový typ hodnoty `JsonNodeType.NULL`. Pokud ano, metoda ihned vrací `Pair` s hodnotou false a vysvětlením, že hodnota byla `null`. Pokud hodnota není nulová, metoda pokračuje kontrolou, zda textová reprezentace hodnoty není prázdný řetězec. Pokud je hodnota prázdná, vrací `Pair` s false a vysvětlením, že hodnota byla prázdná. V případě, že hodnota není ani nulová ani prázdná, vrací metoda `Pair` s true a prázdným vysvětlením.

4.4.7 Metody pro hodnoty pole

Třída `ArrayValueHandler` slouží k implementaci zpracování JSON polí. Nejdříve se s využitím `factory.getMethod` získá potřebná metoda pro zpracování výsledku. Pokud se příslušná metoda nenajde, vrací se `JsonResultDto` s informací o neshodě. Pokud je metoda dostupná, je zavolána funkce execute, která vyhodnotí celé pole a vrátí `Pair`, který obsahuje výsledek dané metody. Poté se vrátí z metody `JsonResultDto`, který obsahuje celkový výsledek dané operace.

Metoda Compare

```
val result = containsAllElements(value.elements().asSequence().toList(),
resultConfig.expectedValue.toMutableList())
val explanation = "Results are not the same"
return Pair(
    first = result,
    second = if (result) "" else explanation
)
```

Metoda `compare` je určena pro porovnání obsahu JSON pole s očekávanými hodnotami definovanými v `JsonResult`. Principem této metody je ověřit, zda všechny prvky z očekávaného seznamu hodnot jsou přítomny v hodnotách JSON pole. Výsledný `Pair`

obsahuje `boolean` hodnotu indikující úspěch porovnání a v případě neshody poskytuje vysvětlení.

```
private fun containsAllElements(actual: List<JsonNode>, expected:
List<String>): Boolean {
    val actualStrings = actual.map { it.asText() }
    return areListsEqual(actualStrings, expected)
}

private fun <T : Comparable<T>> areListsEqual(list1: List<T>, list2:
List<T>): Boolean {
    return list1.sorted() == list2.sorted()
}
```

Metoda `containsAllElements`, přijímá dva seznamy a to „actual“ a „expected“. Dále využívá funkci `areListsEqual`, která oba seznamy nejprve seřadí a poté porovná, zda jsou identické. Pokud jsou seřazené seznamy stejné, znamená to, že všechny očekávané prvky jsou přesně reprezentovány v poli, a metoda vrací true.

Metoda CompareStrict

```
val result = resultConfig.extendedDataType?.let { datatype ->
    containsAllElements(value.elements().asSequence().toList(),
resultConfig.expectedValue.toMutableList(), datatype)
} ?: false

val explanation = "Results are not the same"

return Pair(
    first = result,
    second = if (result) "" else explanation
)
```

Metoda `CompareStrict` je rozšířenou verzí metody `Compare`, která navíc zajišťuje, že všechny prvky v JSON poli odpovídají nejen hodnotou, ale i datovým typem specifikovaným v `resultConfig` jako `extendedDataType`.

```
private fun containsAllElements(actual: List<JsonNode>, expected:
MutableList<String>, type: String): Boolean {
    val actualStrings = mutableListOf<String>()

    for (element in actual) {
        if (!element.nodeType.isType(type)) return false
        actualStrings.add(element.asText())
    }

    return areListsEqual(actualStrings, expected)
}

private fun <T : Comparable<T>> areListsEqual(list1: List<T>, list2:
List<T>): Boolean {
    return list1.sorted() == list2.sorted()
}
```

V metodě `containsAllElements` se nejdříve zkontroluje, jestli každý prvek odpovídá specifikovanému typu. Pokud ne, metoda ihned vrací false. Jestliže všechny prvky odpovídají danému typu, tak se jejich textová reprezentace srovnává s očekávanými hodnotami pomocí funkce `areListsEqual`.

Metoda Contains

```
val elements = resultConfig.expectedValue
val result = value.elements().asSequence().map { it.asText()
}.toList().containsAll(elements)
return Pair(
    result,
    if(result) "" else "Array didn't contain all set elements "
)
```

Metoda `contains` je navržena tak, aby ověřila, zda JSON pole obsahuje všechny očekávané prvky specifikované v konfiguraci `resultConfig`. Tato metoda umožňuje zjistit, zda všechny prvky uvedené v `expectedValue` jsou přítomny v JSON poli. Pokud jsou všechny očekávané prvky v poli nalezeny, metoda vrací `Pair` s hodnotou true a prázdným řetězcem jako vysvětlením. Pokud některá z očekávaných hodnot chybí v poli, vrací se false a vysvětlením, že ne všechny očekávané hodnoty byli obsaženy v poli.

Metoda isType

```
val result = resultConfig.extendedDataType?.let { datatype ->
    value.elements().asSequence().filter { it.nodeType.isType(datatype) }
        .toList().size == value.elements().asSequence().toList().size
} ?: false
val explanation = "Results are not same"
return Pair(first = result, second = if(result) "" else explanation)
```

Metoda `isType` slouží k ověření, že všechny prvky v JSON poli, jsou stejného datového typu specifikovaném v `resultConfig`. V metodě se projdou pomocí metody `filter` všechny prvky JSON pole, kde se pomocí rozšiřující metody `isType` z `JsonNodeType` zjistí, jestli hodnota je stejného datového typu, jaký je uveden v `resultConfig`. Výsledek je poté vrácen jako `Pair`, kde první hodnota udává, zda všechny prvky pole splňují požadovaný datový typ a druhá hodnota obsahuje vysvětlení v případě, že se ne všechny prvky shodují.

Metoda Matches

```
resultConfig.expectedValue.firstOrNull()?.let {
    val regex = Regex(it)
    val result = allElemMatchesRegex(value.elements(), regex)
    val explanation = "Value doesn't match regex $regex"

    return Pair(
        result,
        if(result) "" else explanation
    )
}
return Pair(false, "No value found for expected value")
```

Metoda `matches` zajišťuje, že všechny prvky JSON pole odpovídají specifikovanému regulárnímu výrazu uvedenému v `resultConfig`. Na začátku metoda vytvoří `Regex` objekt z první očekávané hodnoty definované v konfiguraci výsledku. Poté kontroluje, zda každý prvek v poli splňuje tento regulární výraz pomocí funkce `allElemMatchesRegex`.

Výsledek této kontroly je vrácen jako `Pair`, kde první hodnota indikuje, zda všechny prvky pole odpovídají regulárnímu výrazu, a druhá hodnota poskytuje vysvětlení, v případě že ne všechny prvky odpovídají regulárnímu výrazu.

```
private fun allElemMatchesRegex(actual: Iterator<JsonNode>, regex: Regex
): Boolean {
    actual.forEach { if(!regex.matches(it.asText())) return false }
    return true
}
```

V této funkci se iteruje přes prvky pole a aplikuje regulární výraz na textovou reprezentaci každého prvku. Pokud kterýkoli prvek neodpovídá regulárnímu výrazu, funkce ihned vrací false.

Metoda NonNull

```
if (value.nodeType != JsonNodeType.NULL) return Pair(false,"Value was
null")
value.elements().forEach {
    if(it.isNull) return Pair(false, "Value was null")
}
return Pair (true,"")
```

Metoda `nonNull` slouží k zjištění, jestli žádný z prvků v JSON pole není nulový. Tato metoda nejprve ověří, zda samotné pole není nulové. Pokud je pole nulové, metoda okamžitě vrátí `Pair` s false a vysvětlením, že hodnota byla nulová. Pokud pole není nulové, metoda pokračuje iterací přes všechny prvky pole. Pro každý prvek zkontroluje, zda není nulový. Jakmile narazí na nulový prvek, vrátí `Pair` s false a opětovným vysvětlením, že byla hodnota nulová.

Metoda NonNullOrEmpty

```
if (value.nodeType != JsonNodeType.NULL) return Pair(false,"Value was
null")
val elements = value.elements().let {
    if (it.asSequence().toList().isEmpty()){
        return Pair(false, "Array is empty")
    } else {
        it
    }
}
elements.forEach {
    if(it.isNull) return Pair(false, "Value was null")
}
return Pair (true,"")
```

Metoda `nonNullOrEmpty` je zaměřena na zajištění, že JSON pole není nulové, prázdné, ani neobsahuje nulové prvky. Tato metoda provádí několik kontrol, aby zajistila validaci obsahu pole. Metoda začíná kontrolou, zda není celé pole nulové. Pokud je pole nulové, metoda okamžitě vrátí `Pair` s false a vysvětlením. Pokud pole není nulové, metoda dále ověřuje, zda pole neobsahuje žádné prvky. To je zjištěno tak, že se transformuje na iterátor prvků pole a poté na seznam a kontroluje se, zda je seznam prázdný pomocí metody `isEmpty`. Pokud je pole prázdné, vrátí se `Pair` s false a vysvětlením. Pokud pole obsahuje prvky, metoda pokračuje kontrolou každého prvku pole. Pro každý prvek zkontroluje, zda není nulový. Pokud jakýkoliv prvek v poli je nulový, vrátí se `Pair` s false a vysvětlením. V případě že všechny kontroly projdou úspěšně vrátí se `Pair` s true a prázdný, řetězcem pro vysvětlením.

4.4.8 Metody pro JSON objekt

Třída `ObjectValueHandler` je navržena pro zpracování JSON objektů v rámci validace výsledků. Tato třída využívá třídu `ResultMethodFactory` k nalezení specifických metod pro validaci JSON objektů založených na konfiguraci specifikované v `JsonResult`. Nejdříve se z konfigurace získá hodnota pro expectedValue. Potom se z ResultMethodFactory pokusí získat metoda pro zpracování výsledku. Pokud není metoda nalezena vrátí se `JsonResultDto` s informací o neshodě.

Pro JSON objekty je definována pouze metoda `Compare`, která efektivně zajišťuje, zda je celý objekt shodný s očekávanou hodnotou.

```
resultConfig.expectedValue.firstOrNull()?.let{
    val expectedValue = it.replace("\\s".toRegex(), "")
    val actualValue = value.toString().replace("\\s".toRegex(), "")

    val result = actualValue == expectedValue

    return Pair(
        first = result,
        second = if (result) "" else "Objects aren't same"
    )
}
return Pair(
    first = false,
    second = "No value found for expected value"
)
```

Tato metoda nejprve normalizuje jak očekávanou, tak aktuální hodnotu objektu odstraněním bílých znaků a následně je porovnává. Pokud jsou hodnoty identické, vrací `Pair` s hodnotou true a s prázdným vysvětlením. V opačném případě vrací `Pair` false a vysvětlením, že objekty nebyli stejné.

Důvod, proč je pro objekty definována pouze metoda `Compare`, spočívá v tom, že pro detailnější validaci jednotlivých prvků objektu je efektivnější použít skalární metody. Tyto metody umožňují specifičtější a flexibilnější kontrolu jednotlivých hodnot v rámci objektu.

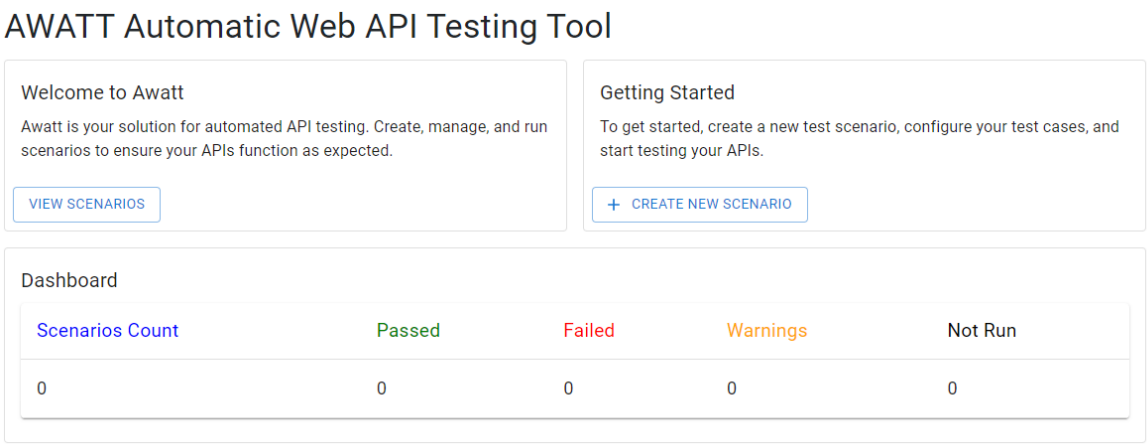
5 Uživatelské rozhraní

Uživatelské rozhraní, které je součástí mého testovacího nástroje, bylo navrženo tak, aby umožňovalo intuitivní a efektivní interakci s funkcemi serverové části aplikace. Toto rozhraní je postaveno na moderních technologiích, konkrétně Reactu, TypeScriptu a knihovně MaterialUI, což uživatelům zajišťuje hladký a vizuálně příjemný zážitek. MaterialUI je využito pro stylizování komponent. To nabízí bohatou sadu předdefinovaných stylů a komponent jako jsou tlačítka, dialogová okna a snackbar notifikace, které zlepšují uživatelskou přívětivost a zároveň zajišťují konzistentní designový jazyk napříč aplikací.

Uživatelské rozhraní nabízí přehledné panely a formuláře pro zadávání testovacích případů, zobrazení výsledků a analýzu dat. Díky reaktivním vlastnostem Reactu se uživatelé mohou těšit na okamžité aktualizace UI v reakci na změny na serveru bez nutnosti manuálního obnovování stránky.

Celkově, toto uživatelské rozhraní zásadně usnadňuje práci se serverovou částí testovacího nástroje, umožňuje rychlé nastavení testů a snadné vyhodnocování výsledků.

5.1 Domovská stránka



Obrázek 2: Domovská obrazovka

Domovská obrazovka je navržena tak, aby poskytovala základní přehled o testovacích scénářích prostřednictvím „dashboardu“. Tato obrazovka je základním středobodem pro uživatele, kde může vidět statistiky pro všechny scénáře.

Na obrazovce „Dashboard“ jsou zobrazeny klíčové statistiky testovacích scénářů ve formě tabulky. Tato sekce zobrazuje celkový počet scénářů, počet úspěšně dokončených scénářů, počet scénářů, u kterých selhaly všechny kroky označené jako „failed“, počet scénářů, u kterých se některé kroky nezdařily označené jako „Warnings“ a také počet scénářů, které ještě nebyly spuštěny.

Další dvě hlavní části na domovské obrazovce jsou karty „Welcome to Awatt“ a „Getting Started“. Karta "Welcome to Awatt" uvádí základní popis testovacího nástroje a jeho schopnost automatizace testování API. Zde je také tlačítko „View Scenarios“, které uživatele přesměruje na stránku se seznamem všech scénářů. Karta "Getting Started" je zaměřena jako průvodce pro nové uživatele, s tlačítkem „Create New Scenario“, které přesměruje uživatele na stránku pro tvorbu nového scénáře.

5.2 Scenarios stránka

Obrázek 3: Seznam všech scénářů

Stránka „Scenarios“ poskytuje centrální místo pro správu a zobrazení testovacích scénářů. Ta umožňuje uživatelům prohlížet seznam všech scénářů, spouštět testy, vytvářet nové scénáře, nebo mazat existující. Každý scénář je prezentován v tabulce s detaily jako název, popis a souhrn výsledků testování.

Nad seznamem se nacházejí tři tlačítka pro práci se scénáři. Tlačítko „New Scenario“ přesměruje uživatele na stránku pro vytváření nových scénářů. Tlačítko „Run“ umožňuje spuštění vybraných testovacích scénářů. Pokud není vybrán žádný scénář, aplikace upozorní uživatele, že je potřeba nejdříve nějaké scénáře vybrat. Tlačítko „Delete“ slouží k odstranění vybraných scénářů a otevírá dialogové okno pro potvrzení akce. Stejně jako u spuštění testů, pokud nejsou scénáře vybrány, informuje aplikace uživatele o nutnosti výběru.

V pravé části tabulky, ve sloupci „Results“, jsou zobrazeny výsledky posledního průběhu scénáře. Ty jsou reprezentovány počtem úspěšných a neúspěšných kroků, doprovázených ikonami, které vizuálně indikují výsledky. Zelená ikona značí úplný úspěch, červená ukazuje

na selhání všech kroků, oranžová pak signalizuje smíšené výsledky, a absence ikony naznačuje, že testy nebyly ještě spuštěny. Pokud je scénář aktuálně testován, místo výsledku je zde text „Running…“ vedle kterého je ikona načítání, která ukazuje, že výsledky ještě nejsou dostupné.

5.3 Stránka scénáře

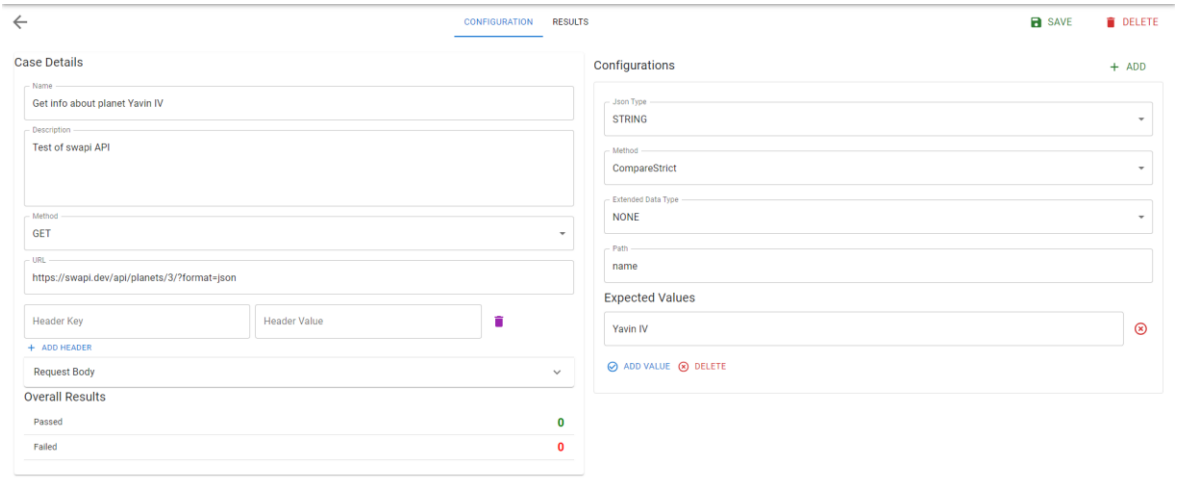
Obrázek 4: Stránka pro scénář

Stránka „Scenario Details“ poskytuje zobrazení a správu jednotlivých testovacích scénářů. Na této stránce mohou uživatelé vidět detaily jako název scénáře, jeho popis a souhrn výsledků testů. K dispozici jsou tlačítka pro ukládání změn, spouštění testů a mazání scénáře. Tyto tlačítka jsou pak deaktivovány během provádění testů, aby nedošlo k narušení procesu.

V sekci „Cases“ jsou uvedeny jednotlivé testovací kroky, které patří ke scénáři. Každý krok je pak zobrazen v komponentě „CaseCard“, která ukazuje název kroku, jeho popis a výsledky testování, zahrnující počet úspěšných a neúspěšných testů. Tlačítko „Add“ umožňuje přidání nového testovacího případu k danému scénáři.

Sekce „Overall Results“ poskytuje přehled o celkovém počtu úspěšných a neúspěšných kroků, což dává rychlý přehled o výsledku průběhu scénáře. Vizuální indikátory jako zelená, červená a oranžová ikona signalizují pak celkový úspěch, selhání nebo smíšené výsledky kroků. Stránka také zahrnuje navigační prvky pro návrat na seznam scénářů a dialogová okna pro potvrzení kritických akcí jako je mazání scénáře, čímž zvyšuje uživatelskou kontrolu a předchází nechtěným akcím.

5.4 Stránka kroku scénáře



Obrázek 5: Stránka kroku scénáře (zdroj vlastní)

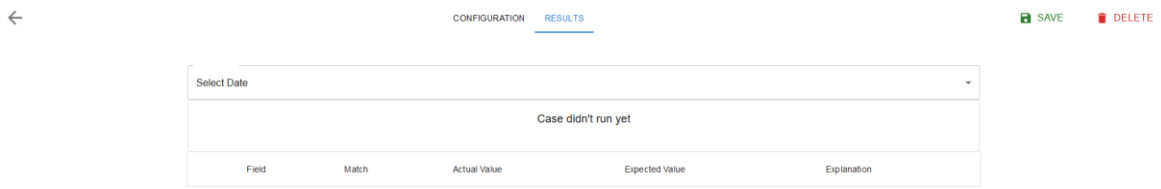
Stránka „case“ pro zobrazení jednotlivých kroků scénáře nabízí uživatelům rozhraní pro správu a úpravu detailů specifického testovacího kroku. Na této stránce je možné editovat název, popis, metodu HTTP požadavku, URL, a tělo požadavku. Stránka rovněž obsahuje funkce pro správu konfigurací výsledků, kde uživatelé mohou definovat a přidávat očekávané výsledky testů. Tlačítka jsou na stránce dvě. Kde tlačítko „Save“ slouží pro uložení změn a tlačítko "Delete" pak, po potvrzení v dialogovém okně, smaže krok ze scénáře.

Pro editaci „Request Body“ je ve stránce zabudovaný editoru kódu. Ten dovoluje uživatelům přímo upravovat tělo HTTP požadavku ve formátu JSON. Editor nabízí funkce jako automatické formátování, zvýraznění syntaxe a možnost překlopení sekcí pro lepší organizaci a přehlednost kódu.

Výsledky jsou pak zobrazeny v tabulce, která ukazuje počet úspěšných a neúspěšných testů podle zadaných konfigurací. Tyto výsledky poskytují přehled o úspěšnosti testovacího kroku a jsou aktualizovány po každém spuštění scénáře.

Na stránce je pak menu, které uživateli umožňuje přepnout mezi zobrazením konfigurace a detailech kroku položka „Configuration“, tak detailnějším zobrazením výsledků položka „Results“.

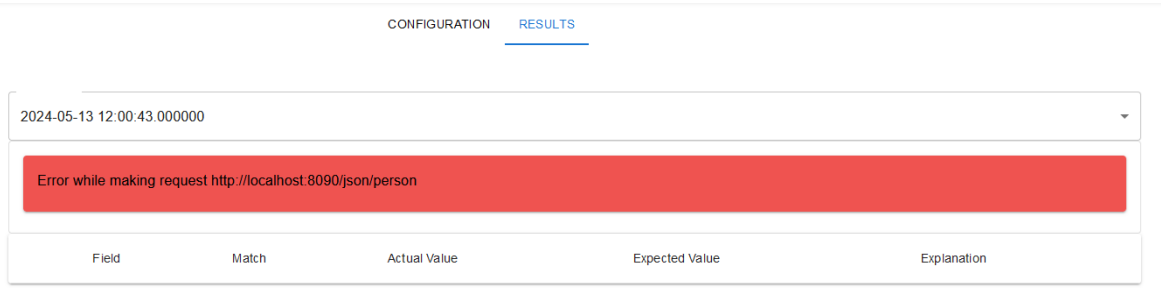
5.5 Stránka pro zobrazení detailů výsledků



Obrázek 6: Stránka pro zobrazení výsledků prázdná

Stránka pro detailní zobrazení výsledků testovacího kroku umožňuje uživatelům přehledně vidět výsledky jednotlivých běhů testů na základě výběru datumu. Uživatel může pomocí rozbalovacího menu vybírat mezi dostupnými daty průběhu scénářů. Po výběru konkrétního data se automaticky načtou výsledky pro toto datum. Pokud jsou nějaké výsledky dostupné, zobrazí se v tabulce, která ukazuje jednotlivé testované položky, jejich očekávané hodnoty, skutečné hodnoty, a zda test prošel či nikoliv. Jednotlivé položky je pak možné rozkliknout, čímž se rozbalí karta s detailnějším popisem.

V případě, že dojde k chybě během testování, stránka zobrazí error v rámci informačního boxu, kde uživatelé mohou vidět chybovou zprávu přímo spojenou s problémem, který nastal během testu. Toto umožňuje rychlé identifikování a řešení problému.



Obrázek 7: Stránka pro zobrazení výsledků s chybovou hláškou

5.6 Kartička pro konfiguraci zpracování výsledků

Obrázek 8: Karta pro nastavení konfigurace pro zpracování výsledku

Kartička pro konfiguraci zpracování výsledků je určena k nastavování a úpravám parametrů specifického výsledku v rámci kroku scénáře. Uživatel zde může specifikovat několik důležitých atributů, jako jsou typ JSON dat, metoda porovnání, rozšířený datový typ, cesta v datové struktuře a očekávané hodnoty. Každý z těchto atributů je možné dynamicky upravovat dle potřeb uživatele.

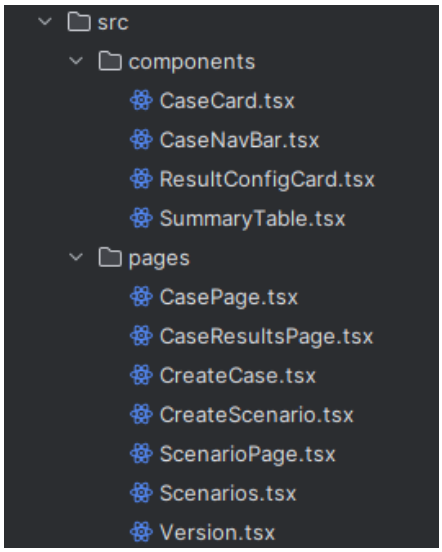
Na kartičce je dostupný výběr z předdefinovaných metod pro zpracování skalárních typů (string, number atd.), polí a objektů, což závisí na vybraném typu JSON dat. Uživatelé mohou dále definovat cestu k datům, které určuje, kde v JSON struktuře se daná data nacházejí.

Pro detailnější specifikaci lze nastavit rozšířený datový typ, což umožňuje přesněji určit, jakého typu data uživatel očekává na dané cestě. Kdy hlavní použití je pro JSON typ pole. Kde metoda „`isType`“ určuje, jestli jsou všechna data v poli stejného datové typu, určeného právě v „Extended Data Type “.

Zásadní částí jsou očekávané hodnoty, kde uživatel může přidávat nebo odebírat jednotlivé hodnoty. Každá hodnota je reprezentována editovatelným textovým polem, doplněným o ikonu pro její odstranění. Přidání nové hodnoty je realizováno pomocí tlačítka, které přidá nové textové pole do seznamu.

6 Implementace uživatelského rozhraní

6.1 Rozdělení komponent



Obrázek 9: Rozdělení React projektu

Kód je strukturovaný do dvou základních složek, „components“ a „pages“, což usnadňuje organizaci kódu a zvyšuje jeho přehlednost. Složka „components“ obsahuje jednotlivé komponenty, které jsou znovupoužitelné části uživatelského rozhraní, jako jsou `CaseCard`, `CaseNavBar`, `ScenarioCard`, a další. Tyto komponenty jsou designovány tak, aby byly použitelné v různých částech aplikace, což zvyšuje modularitu a flexibilitu vývoje.

Druhá složka „pages“ zahrnuje soubory představující jednotlivé stránky uživatelského rozhraní. Každá stránka slouží jako kontejner pro různé komponenty a představuje ucelený uživatelský pohled, jako jsou `CasePage`, `ScenarioPage` nebo `CreateScenario`. Tyto stránky jsou propojeny s konkrétními cestami v aplikaci, což umožňuje efektivní navigaci a správu různých pohledů uživatele.

Ostatní soubory jako `App`, `index` jsou pak uloženy v kořenové složce „src“.

6.2 Routing

V uživatelském rozhraní je „Routing“ řízen pomocí knihovny „*react-router-dom*“, která umožňuje efektivní správu navigace v React aplikacích. Všechny „routovací“ operace jsou obaleny komponentou `<Router>`, která slouží jako hlavní kontejner pro interní správu změn URL a navigace mezi stránkami. V rámci `<Router>` se nachází komponenta `<Routes>`, která obsahuje definice jednotlivých cest `<Route>`. Ty specifikují cesty na

odpovídající komponenty, jež mají být zobrazeny, když je daná cesta aktivní. Některé cesty využívají dynamické parametry, jako je například „`/scenario/:id`“, kde `id` představuje proměnnou část URL adresy, což umožňuje aplikaci zobrazovat data specifická pro konkrétní scénář. Tento systém „routing“ poskytuje aplikaci schopnost efektivně spravovat vícestránkový uživatelský zážitek v rámci jednostránkové aplikace SPA neboli „Single Page Application“.

6.3 GraphQL klient

Pro práci s GraphQL používám Apollo klienta, který je integrován do React aplikace. Toto řešení umožňuje efektivní a flexibilní zpracování dat z GraphQL serveru. Využití Apollo klienta v aplikaci nabízí několik výhod, včetně automatizovaného „cáchování“, jednoduchého spravování stavu dat a podpory pro integraci aktualizací.

6.3.1 Konfigurace Apollo klienta

```
const client = new ApolloClient({
    uri: "http://localhost:8080/graphql",
    cache: new InMemoryCache()
});
```

Nastavení Apollo klienta v kódu začíná vytvořením instance ApolloClient s konfigurací, která specifikuje uri adresu GraphQL serveru a používá InMemoryCache pro „cáchování“ dat a zvýšení efektivity aplikace.

```
<ApolloProvider client={client}>
    <App />
</ApolloProvider>
```

Tato instance klienta je poté předána komponentě ApolloProvider, která obaluje celou React aplikaci. Tím se umožní všem komponentám v aplikaci přístup k funkcím Apollo klienta pro zasílání dotazů query a mutation na GraphQL server.

6.3.2 Použití dotazů a mutací

Dotazy a mutace jsou definovány pomocí šablony „gql“, což je tagovaný šablonový literál, který umožňuje zápis GraphQL dotazů přímo v JavaScriptu. Tyto dotazy jsou poté použity jako argumenty v příslušných Apollo funkcích.

Pro volání jednotlivých dotazů se používají funkce `useQuery` a `useLazyQuery`. Pro volání mutací jsou pak k dispozici funkce `useMutation` a `useLazyMutation`. Tyto funkce se starají o provedení dotazu na GraphQL server. Rozdíl mezi klasickou verzí funkce

a „lazy“ je ten, že klasická funkce vykoná dotaz okamžitě když je query zavoláno. Zatím co „lazy“ varianta vrátí funkci, která pak umožní zavolat dotaz na server manuálně.

6.3.3 Zpracování výsledků dotazů

Pro zpracování chyb se v implementaci UI používá funkce `onError`, kterou Apollo nabízí. Funkce `onError` je využívána k zachycení a zpracování chyb, které mohou při dotazech nastat.

```
onError: (err) => {
    setAlert({
        open: true,
        message: err.message,
        type: 'error'
    });
    getRunningScenarios();
}
```

Zde je pak ukázka zpracování chyby pomocí funkce `onError`. Aplikace reaguje na chyby získané z GraphQL dotazů tím, že aktivuje uživatelské rozhraní pro zobrazení chybových zpráv. Konkrétně je stav „alert“ nastaven na „open: true“, což otevírá snackbar komponentu s chybovou zprávou získanou z chybové hlášky, kterou vrátí GraphQL dotaz. Tento přístup ihned informuje uživatele o problému, který nastal, a poskytuje specifický důvod selhání operace.

Často je po dokončení GraphQL dotazu potřeba aktualizovat data v uživatelském prostředí. V implementaci uživatelského prostředí k tomu se používají dva hlavní způsoby. Prvním je použití funkce `onCompleted`, která je dostupná z Apollo klienta. Funkce `onCompleted` je „callback“, který se spustí ihned po úspěšném dokončení dotazu. Tento mechanismus je užitečný pro provádění operací, které by měly následovat bezprostředně po získání dat, jako je aktualizace stavu aplikace nebo zobrazení notifikací uživatelům. Zde je možné vidět na příkladu je vidět použití funkce `onCompleted`.

```
onCompleted: (data) => {
    getRunningScenarios();
    const newCaseResults: CaseResults =
data.runEvaluation.casesResults.reduce(
        (acc: CaseResults, curr: { caseId: string; overAllResult:
CaseResult; error?: string }) => {
            acc[curr.caseId] = {
                passed: curr.overAllResult.passed,
                failed: curr.overAllResult.failed,
                error: curr.error
            };
            return acc;
        }, {});
    setCaseResults(newCaseResults);
    setPassed(data.runEvaluation.overAllResult.passed || 0);
    setFailed(data.runEvaluation.overAllResult.failed || 0);
    setAlert({
        open: true,
        message: 'Evaluation completed successfully!',
        type: 'success'
    });
}
```

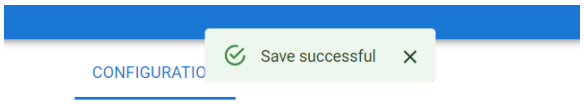
Kde se provede logika pro získání výsledků dotazu. Poté se použijí set funkce pro jednotlivé části UI, které se mají aktualizovat.

Druhý způsob je pak použití „hooku“ `useEffect`. Tento „hook“ je základní „hook“ v React, který umožňuje provádět vedlejší efekty v komponentě, jako jsou API volání nebo práce s DOM, reagující na změny ve stavu nebo „props“.

```
useEffect(() => {
    if (data) {
        const { name, description, url, method, payload, resultConfig,
headers } = data.getCase;
        setName(name);
        setDescription(description);
        setUrl(url);
        setMethod(method);
        setRequestBody(parseRequestBody(payload));
        setResultConfigs(parseResultConfigs(resultConfig));
        setHeaders(headers);
    }
}, [data]);
```

V uvedeném příkladu, `useEffect` sleduje změny v „data“, která obsahují informace o kroku scénáře. Jakmile jsou data k dispozici, `useEffect` aktualizuje lokální stavy komponenty (jako jméno, popis, URL, metodu atd.) na základě nově získaných dat.

6.4 Zpětná vazba uživateli



Obrázek 10: Zobrazení alertu pro úspěšnou operaci

Zpětná vazba uživatelům poskytována pomocí komponenty Snackbar z knihovny MaterialUI. Tato komponenta zobrazuje Alert, který informuje uživatele o různých stavech operací, jako jsou úspěchy, chyby nebo varování. Například, když je evaluace scénáře spuštěna a dokončena, Snackbar se objeví s úspěšnou zprávou o dokončení evaluace, nebo s chybovou zprávou, pokud evaluace selže kvůli technickému problému.

```
interface AlertType {
    open: boolean;
    message: string;
    type: "success" | "error" | "info" | "warning" | undefined;
}
```

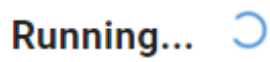
Samotná implementace je realizována tak, že komponenta Snackbar je aktivována stavem „open“, který je uložený v proměnné alert, která implementuje rozhraní `AlertType`. Alert pak má v sobě informace o zprávě „message“, a o jaký typ alertu se jedná. Tato hodnota může nabývat stavů „succes“, „error“, „info“, „warnig“ a kvůli tomu, jak je Alert implementován v knihovně MaterialUI, tak i stav „undefined“. Informace pro alert se pak mění v závislosti na tom, jaká událost Alert vyvolala. V případě úspěchu se zobrazí Alert typu „success“ se zprávou, která informuje uživatele o úspěšném provedení operace. Například když uživatel stiskne tlačítko „Save“ tak se mu v případě úspěchu zobrazí Alert se zprávou, že položka byla úspěšně uložena.

Kromě zobrazování zpráv v Snackbar. Uživatelské rozhraní reaguje dynamicky na stavy aplikace. Když uživatel spustí evaluaci scénáře pomocí tlačítka "Run", zobrazí se načítací

Obrázek 11: Zobrazení ikony v běžícím scénáři

„spinner“ (neboli komponenta `CircularProgress` z MaterialUI) přímo v rozhraní, což signalizuje uživateli, že se právě daný scénář vyhodnocuje.

Pokud pak uživatel spustí evaluaci scénáře přímo ze seznamu scénářů, tak se na místě pro výsledky zobrazí text „Running…“ a ikona načítání.



Obrázek 12: Označení běžícího scénáře v seznamu

Tento stav je pak přenesen mezi stránky, tedy když uživatel spustí evaluaci scénáře ze seznamu. A následně přejde na stránku pro daný scénář bude zde viditelný stav, že scénář právě běží.

Závěr

Tato bakalářská práce pro mě představovala novou a významnou výzvu, při které jsem musel navrhnout celou architekturu aplikace od základu sám. Před zahájením mé práce bylo nezbytné pečlivě promyslet, jak budou jednotlivé části systému spolupracovat, aby bylo dosaženo efektivního fungování celé aplikace. Tento proces zahrnoval nejen technické aspekty, jako je volba technologií a způsob jejich integrace, ale také design uživatelského rozhraní a optimalizaci uživatelského zážitku. Tato zkušenost byla nesmírně cenná, protože mi umožnila prohloubit mé znalosti v oblasti softwarového inženýrství, naučila mě řešit komplexní problémy a zlepšila mé schopnosti v oblasti návrhu a implementace složitých systémů.

I když se mi podařilo dosáhnout všech zadaných cílů, vidím prostor pro další vylepšení a rozšíření funkcionality aplikace. Do budoucna bych chtěl přidat podporu pro další typy API, jako jsou GraphQL a gRPC, což by rozšířilo možnosti testování a zlepšilo flexibilitu aplikace. Kromě toho bych rád upravil uživatelské rozhraní, zejména co se týče nastavení jednotlivých kroků scénářů a konfigurace pro zpracování výsledků. Tato vylepšení by uživatelům umožnila snadnější a intuitivnější práci s aplikací a poskytla by jim více možností pro přizpůsobení testovacích scénářů a zpracování výsledků podle jejich potřeb.

7 Literatura

1. IBM. IBM. *What is an API (application programming interface)?* [Online] [Citace: 09. 05 2024.] https://www.ibm.com/topics/api.

2. Fitzgerald, Anna. API Testing: What It Is, Why It's Important & How to Do It. *HubSpot.* [Online] 9. 8 2023. [Citace: 09. 05 2024.] https://blog.hubspot.com/website/api-testing#why-is-api-testing-important.

3. Oracle. What is Java. *Java.* [Online] [Citace: 09. 05 2024.] https://www.java.com/en/download/help/whatis_java.html.

4. —. About the Java Technology. *Java Documentation.* [Online] [Citace: 09. 05 2024.] https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.

5. *ECKEL, Bruce a Svetlana ISAKOVA, 2021. Atomic kotlin. Crested Butte, CO: Mindview LLC. ISBN 978-0-9818725-5-1.*

6. Bansal, Anshul. Java vs. Kotlin. *Baeldung.* [Online] 19. 3 2024. [Citace: 15. 05 2024.] https://www.baeldung.com/kotlin/java-vs-kotlin.

7. *WALLS, Craig, 2016. Spring Boot in action. Shelter Island: Manning. Java. ISBN 978-1-61729-254-5.*

8. Eclipse Foundation. Introduction to Jakarta Persistence. *Jakarta EE.* [Online] [Citace: 09. 05 2024.] https://jakarta.ee/learn/docs/jakartaee-tutorial/current/persist/persistence-intro/persistence-intro.html.

9. Cambi, Luca. Difference Between JPA and Spring Data JPA. *Baeldung.* [Online] 8. 1 2024. [Citace: 10. 05 2024.] https://www.baeldung.com/spring-data-jpa-vs-jpa.

10. FasterXML. Jackson. *GitHub.* [Online] [Citace: 10. 05 2024.] https://github.com/FasterXML/jackson.

11. Visual Paradigm. What is REST API? *Visual Paradigm.* [Online] [Citace: 10. 05 2024.] https://www.visual-paradigm.com/guide/development/what-is-rest-api/.

12. Facebook, Inc. GraphQL. *spec.graphq.* [Online] 7 2015. [Citace: 10. 05 2024.] https://spec.graphql.org/July2015/.

13. GraphQL Foundation. Introduction to GraphQL. *GraphQL.* [Online] [Citace: 10. 05 2024.] https://graphql.org/learn/.

14. —. Queries and Mutations. *GraphQL*. [Online] [Citace: 10. 05 2024.] https://graphql.org/learn/queries/.

15. —. Schemas and Types. *GraphQL*. [Online] [Citace: 10. 05 2024.] https://graphql.org/learn/schema/.

16. —. Introspection. *GraphQL*. [Online] [Citace: 10. 05 2024.] https://graphql.org/learn/introspection/.

17. gRPC Authors. Introduction to gRPC. *gRPC*. [Online] 16. 2 2023. [Citace: 11. 05 2024.] https://grpc.io/docs/what-is-grpc/introduction/.

18. Microsoft. Co jsou databáze? *azure.microsoft*. [Online] [Citace: 11. 05 2024.] https://azure.microsoft.com/cs-cz/resources/cloud-computing-dictionary/what-are-databases/.

19. PostgreSQL Global Development Group. About. *PostgreSQL*. [Online] [Citace: 11. 05 2024.] https://www.postgresql.org/about/.

20. Gomez, Jose. Web Apps Vs. Desktop Apps: Understanding the Differences. *koombea*. [Online] 16. 11 2023. [Citace: 11. 05 2024.] https://www.koombea.com/blog/web-apps-vs-desktop-apps/.

21. Mozilla Corporation. What is JavaScript? *mdm web docs*. [Online] [Citace: 11. 05 2024.] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

22. Baxter-Reynolds, Matt. Microsoft TypeScript: Can the father of C# save us from the tyranny of JavaScript? *ZDNET*. [Online] 1. 10 2012. [Citace: 11. 05 2024.] https://www.zdnet.com/article/microsoft-typescript-can-the-father-of-c-save-us-from-the-tyranny-of-javascript/.

23. Miller, Stephan. What Is React? *Codecademy*. [Online] 13. 09 2021. [Citace: 11. 05 2024.] https://www.codecademy.com/resources/blog/what-is-react/.

24. MUI. Material UI - Overview. *MUI Core*. [Online] [Citace: 12. 05 2024.] https://mui.com/material-ui/getting-started/.

25. Apollo Graph Inc. Client (React). *Apollo Docs*. [Online] [Citace: 11. 05 2024.] https://www.apollographql.com/docs/react.

26. Docker Inc. Docker overview. *Docker.docs*. [Online] [Citace: 11. 05 2024.] https://docs.docker.com/get-started/overview/.

Seznam příloh

PŘÍLOHA A – docker-compose.yml

Soubor pro spuštění aplikace pomocí technologie docker.

```
version: '3'

services:
  db:
    container_name: postgres
    image: postgres
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    volumes:
      - db:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - db
    restart: always
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin"]
      interval: 10s
      timeout: 5s
      retries: 5

  awatt_be:
    container_name: awatt_be
    image: salt0ut/awatt_be:1.0.1
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/postgres
      SPRING_DATASOURCE_USERNAME: ${POSTGRES_USER}
      SPRING_DATASOURCE_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "8080:8080"
    depends_on:
      - db
    networks:
      - db

  awatt:
    container_name: awatt
    image: salt0ut/awatt_fe:1.0.1
    ports:
      - "3000:3000"
    depends_on:
      - awatt_be
    networks:
      - db

networks:
  db:
    driver: bridge

volumes:
  db:
  minio_data:
```