Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Návrh a implementace webové aplikace s architekturou REST

Bakalářská práce

Autor: Tatiana Buravova

Studijní obor: Informační management

Vedoucí práce: Mgr. Vojtěch Vorel, Ph.D.

Odborný konzultant: Ing. Martin Valenta

Hradec Králové květen 2023

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 22.05.2023 Tatiana Buravova

Poděkování:

Děkuji vedoucímu bakalářské práce panu Mgr. Vojtěchu Vorelovi, Ph.D. za metodické vedení práce a odborné konzultace, které mi poskytl při zpracování mé bakalářské práce.

Anotace:

Cílem práce je navrhnout REST API pro zpracování dat z databáze PostgreSQL. Vyvinutá webová služba nabídne základní vyhledávání subjektů, zadání nových dat do formuláře, modifikaci a odstranění dat.

Klíčová slova:

REST; API; OpenAPI; Java; Spring; databáze.

Annotation:

The aim of the bachelor thesis is to create a web application for processing data from the PostgreSQL database. The resulting data will be displayed on this web application. REST API will offer a basic search for subjects, entering new data into the form, modification and deleting data.

Key words:

REST; API; OpenAPI; Java; Spring; database.

Obsah

1. ÚVOD

1.1. Důvod výběru tématu bakalářské práce

V současné době se pro mnoho úspěšných společností stává trendem vyvíjet aplikace komunikující moderním způsobem pomocí webové služby. Tento způsob komunikace nazýváme *API – Application Programming Interface*. Jde o rozhraní, které umožňuje distribuování dat z jedné strany a použití těchto dat nějakým množstvím jiných objektů z druhé strany. Vývojáři webových služeb mohou využívat rozmanitých možností volně dostupných frameworků, které značně zrychlují tvorbu aplikací komunikujících po síti.

Pracuji v bance, kde se technologie API používá, například v rámci koncepce otevřeného bankovnictví nebo v rámci použití bankovní identity. API služby umožňují bezpečné automatizované zpracování výpisů a pohybů na účtech vedených u banky. Navíc API dovoluje stahování historie transakcí z účtů vedených u jiných bank. Také díky API službám banky se klient může bezpečně přihlašovat do portálů různých firem a státní správy, a to stejně jednoduše jako do internetového bankovnictví. Proto je pro mě toto téma velice zajímavé a aktuální.

1.2. Cíl bakalářské práce

Cílem práce je navrhnout REST API pro zpracování uměle vytvořených údajů klientů banky z databáze PostgreSQL. Výsledná data se zobrazí na webovém portálu, který prostřednictvím REST API nabídne základní vyhledávání subjektů, zadání nových dat do formuláře, modifikaci a odstranění dat. Výsledky poskytne aplikace formou jednoduchého reportu. Serverové řešení bude implementováno v jazyce Java s použitím Spring Frameworku. Frontendová část bude vytvořena s použitím jazyka HTML/CSS a frameworku Thymeleaf. Dokumentace k projektu bude vytvořena pomocí Open API Specification – Swagger. Ten umožňuje popsat REST API pomocí formátu JSON nebo YAML.

Vyvinutá webová služba by mohla být použitá vnitřními API banky za účelem testování. V současnosti se při testování některých nových funkčností používají připravená testovací klientská data. Testovací data jsou uložena do databáze. Vyvinutá aplikace by mohla sloužit pro odstranění, přidání, editaci a vyhledávání některých testovacích klientských údajů. Zmíněné operace jsou voláním endpointů webové služby. Testovací klientská data mohou být přidána do databáze i pomocí frontendové části aplikace.

2. WEBOVÉ SLUŽBY

2.1. Typy webových služeb

S růstem použití internetu a častějším výskytem *HTTP – Hypertext Transfer Protocol* se v současnosti staly webové služby hlavním prostředkem pro vzájemné propojení webových systémů. Webová služba je softwarová součást, ke které uživatel může přistupovat prostřednictvím adresy URL. Webovým obsahem byly původně statické stránky HTML, které se vyvinuly do dynamických webových aplikací poskytujících různou funkčnost koncovému uživateli. Koncept webové služby je o krok napřed před tímto webovým paradigmatem a poskytuje pouze věcnou službu, obvykle ve formě prvotních dat JSON nebo XML. Grafické uživatelské rozhraní a funkce jsou dobře oddělené. Webové služby se nyní považují za samostatné modulární aplikace, které mohou být publikované a vyvolané na webu [1].

Obr. 1. Struktura API.

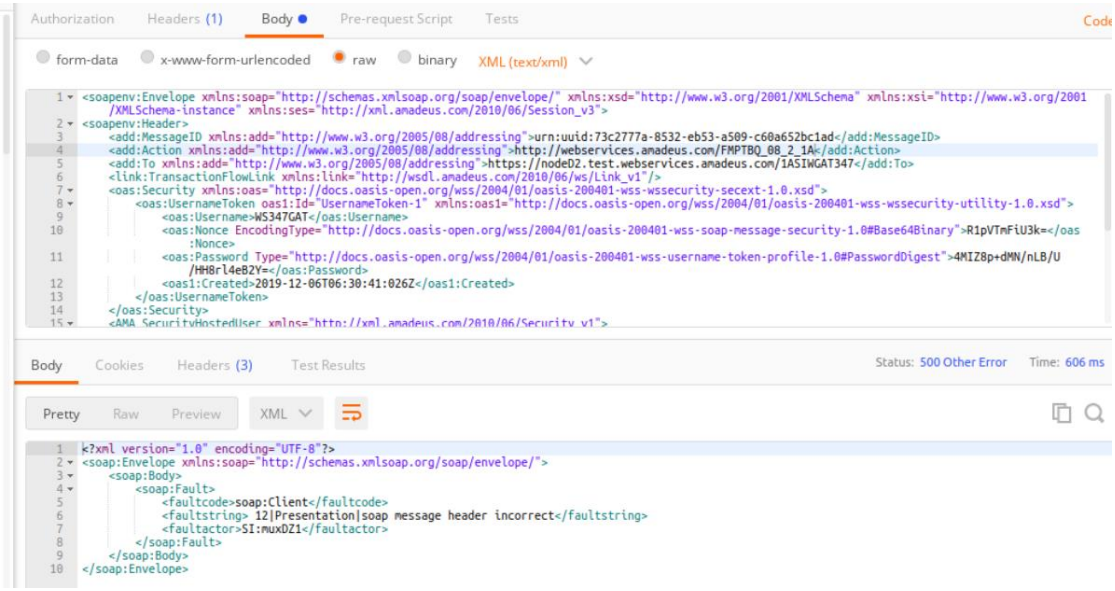
Zdroj: autor.

Největším přínosem webových služeb je interoperabilita. API může být napsané v jiném programovacím jazyce než klientova aplikace, jež k ní přistupuje. Obě aplikace mohou být spuštěné na různých platformách: Linux, Windows, Mac OS atd. V průběhu času byly vydány různé architektonické styly rozhraní API. Každý z nich má své vlastní vzory standardizace výměny dat, své výhody a nevýhody, což bude popsáno v příslušných podkapitolách.

2.1.1. SOAP

Zkratka *SOAP* znamená *Simple Object Access Protocol*. Zprávy posílané pomocí protokolu SOAP jsou obvykle založené na *XML* (*Extensible Markup Language).* XML je značkovací jazyk podobný HTML. REST API je orientovaný na data, SOAP je spíše procedurální. To je vidět i ve způsobu volání. URL při používání SOAPu obsahuje nějaké sloveso, na rozdíl od RESTu, kde je typicky nějaké podstatné jméno [2].

Jedním z nejznámějších použití protokolu SOAP v České republice je odesílání tržeb při Elektronické evidenci tržeb (EET). V českých bankách existují různé starší platební systémy, které také používají SOAP. SOAP služby jsou poměrně náročné, proto jsou aplikované spíše ve státní sféře a finančnictví na velké projekty.



Obr. 2. Volání SOAP služby v Postmanu.

Zdroj: https://blogs.sap.com/2021/08/04/calling-a-soap-api-using-postman/.

2.1.2. GraphQL

GraphQL byl vyvinut v roce 2012 společností *Meta Platforms (Facebook)* pro potřebu flexibility a efektivní komunikace mezi mobilními zařízeními. Hlavní výhodou použití GraphQL je jediný přístupový bod [3]. V případě využití REST API je implementováno několik endpointů. Například pro načtení všech účtů uživatele je vytvořen endpoint `/user/{id}/accounts`. Pro seznam plateb uživatele je nutné vytvořit endpoint `/user/{id}/payments`. Pro zobrazení aktuálního zůstatku na účtu slouží přístupový bod `/user/{id}/balance`. Pro získání dat je zapotřebí se připojit ke všem třem přístupovým bodům. V GraphQL k tomu slouží jeden přesně formulovaný dotaz.

Nevýhodou jediného endpointu je to, že není možné se jednoduchým dotazem doptat na všechna data. Navíc v GraphQL existuje jen jeden HTTP status pro chybu – `400 Bad Request`. Pro všechny ostatní requesty včetně chyb klient dostává HTTP status `200 OK` [4].

Obr. 3. Volání GraphQL endpointu v Postmanu.

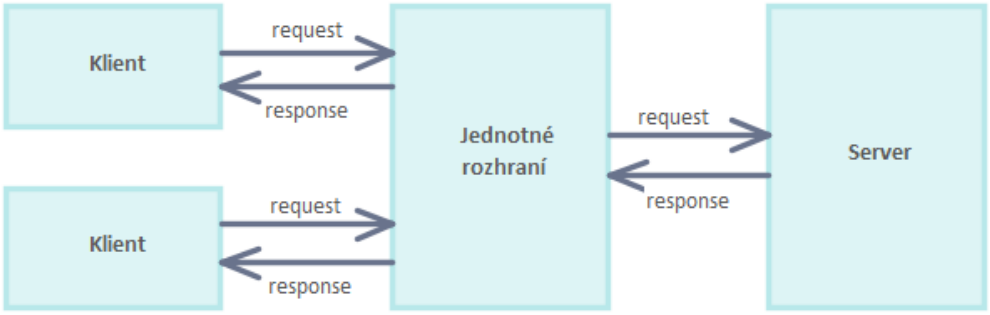
Zdroj: autor.

2.1.3. REST

Název *Representational State Transfer* (*REST*) vytvořil Roy Fielding z University of California, spoluzakladatel projektu *Apache HTTP Server Project*. Fielding se zabýval problémem škálovatelnosti webu a zjistil, že škálovatelnost webu má určitá omezení [5]. Tahle omezení spojil Fielding s kolegy do šesti kategorií, která pojmenovali jako architektonická omezení a rozhodli se je jednotně dodržovat, aby se web mohl dál rozšiřovat:

1. Jednotné rozhraní
2. Architektura klient-server
3. Bezstavový model interakcí
4. Existence vyrovnávací paměti
5. Vrstvený systém
6. Kód na vyžádání

REST architektura předpokládá, že všechny její komponenty musejí sdílet jedno rozhraní. Jednotné rozhraní usnadňuje a rozděluje architekturu, což umožňuje každé komponentě nezávislý vývoj. Jednotným rozhraním a nezávislým vývojem jednotlivých komponent se REST architektura liší od ostatních stylů.



Obr. 4. Jednotné rozhraní.

Zdroj: autor.

Klient-server je architekturou, která předpokládá oddělení úkolů klienta od úkolů serveru. Klient volá konkrétní funkce serveru a pokud dotaz klienta byl validní, server ho zpracuje a odpoví. Výjimky při zaslání dotazu jsou zkoumány a pokud nějaké nastanou, dotaz se vrátí s chybou klientovi. *Bezstavový model interakcí* je založený na požadavcích klienta, kdy požadavek obsahuje veškerá potřebná data pro server. Server neukládá žádná data mezi požadavky. Klient ukládá informaci o stavu požadavku u sebe. Kvůli zmenšení objemu dotazů ze strany klienta na server existuje v REST architektuře tzv. *mezipaměť* neboli *vyrovnávací paměť*. Mezipaměť se tvoří na straně klienta, snižuje počet interakcí do sítě a zmenšuje latenci požadavku. Pro koncového uživatele to přináší větší výkonnost. Vrstvený model REST architektury umožňuje hierarchicky skládat komponenty. V tomto modelu jsou komponenty omezeny svojí komunikací pouze na elementy v jejich vrstvě. Tím se dá dosáhnout vyššího zabezpečení.

Poslední omezení je volitelné a jde o přenesení zátěže ze serveru na klienta. Server může posílat klientovi spustitelné části kódu, kterými dosáhne rozšíření funkcionality u klienta. Jako příklad takových spustitelných kódů lze uvést Java aplety nebo Javascript, které lze spustit na straně klienta. Jedním z původních principů pro REST architekturu je *HATEOAS* princip. Jde o to, že klientovi je známý pouze základní koncový bod. Základní endpoint vrací spolu s daty odkazy na další zdroje. Každý zdroj tak klientovi dává možnosti, jakou akci může provést nebo co dál udělat. Tím pádem klient není závislý na URL adresách, musí znát pouze základní endpoint. Dále lze přidávat nebo měnit další funkčnost, aniž by bylo potřeba cokoli měnit u klienta. Systémy REST API používají čtyři základní funkce perzistentního uložiště, a to CRUD (Vytvořit, Číst, Aktualizovat, Odstranit) [6]. Tyto operace se do HTTP promítají v následujících metodách:

1. GET: používá se pouze pro získání reprezentace zdroje, nikoliv k jeho modifikaci
2. POST: používá se pro vytvoření zdrojů na straně serveru
3. PUT: používá se pro aktualizaci stávajícího zdroje, může se ale používat i pro vytvoření, pokud zdroj neexistuje

4. DELETE: metoda odstraní zdroj ze serveru, pokud bude poskytnuto správné ID zdroje

Metoda	Tělo požadavku	Tělo odpovědi	Bezpečná	Idempotentní
GET	Ne	Ano	Ano	Ano
PUT	Ano	Ano	Ne	Ano
POST	Ano	Ano	Ne	Ne
DELETE	Ne	Ano	Ne	Ano

Tabulka 1. Shrnutí vlastností HTTP metod.
Zdroj: R. Fielding. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. RFC Editor, červ. 1999. url: https://tools.ietf.org/html/rfc2616

Metoda je idempotentní, pokud jejím opakovaným voláním získáme stejný výstup, jako jejím prvním voláním. HTTP požadavky a odpovědi mohou obsahovat hlavičky – *headers*. Hlavičky mají informace o tom, co služba má dělat se zdrojem. Hlavičky se dělí na dvě hlavní kategorie – hlavičky požadavků a hlavičky odpovědí. Při návrhu API lze definovat vlastní hlavičky, které obvykle mají prefix X[7].

Hlavička	Popis	Příklad
Accept	Typ obsahu akceptovatelný v odpovědi	Accept: application/json
Authorization	Údaje pro HTTP autorizaci	Authorization: 208263766
Cache-Control	Nastavení mezipaměti	Cache-Control: no-cache
Connection	Nastavení připojení	Connection: keep-alive
Cookie	HTTP cookie	Cookie: HttpOnly
Content-Type	Typ obsahu v těle požadavku	Content-Type: application/xml
Host	Jméno serveru	Host: ib.xxx.cz

Tabulka 2. Hlavičky požadavků.
Zdroj: R. Fielding. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. RFC Editor, červ. 1999. url: https://tools.ietf.org/html/rfc2616

Hlavička	Popis	Příklad
Cache-Control	Nastavení mezipaměti	Cache-Control: no-cache
Content-Length	Velikost těla odpovědi v bytech	Content-Length: 22
Connection	Nastavení připojení	Connection: keep-alive
Cookie	HTTP cookie	Cookie: HttpOnly
Content-Type	Typ obsahu v těle požadavku	Content-Type: application/xml
Host	Jméno serveru	Host: ib.xxx.cz

Tabulka 3. Hlavičky odpovědí.
Zdroj: Zdroj: R. Fielding. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. RFC Editor, červ. 1999. url: https://tools.ietf.org/html/rfc2616

Protokol HTTP definuje k metodám speciální status kódy. HTTP status kódy lze rozdělit do několika kategorií [8]:

1xx – informační
2xx – úspěšné
3xx – přesměrování
4xx – chyba na straně klienta
5xx – chyba na straně serveru

Nejčastěji se používají následující kódy:

200 OK – úspěšný požadavek
400 Bad Request – server nedokázal splnit požadavek, v požadavku je chyba v syntaxi
401 Unauthorized – pro splnění požadavku je vyžadovaná autorizace klienta
403 Forbidden – server odmítá splnit request, protože klient nemá přístup k obsahu
404 Not Found – daný zdroj neexistuje na poskytnuté URL
500 Internal Server Error – chyba vnitřní logiky serveru

Co největší oddělení klienta a serveru, *cache-friendly* architektura, schopnost podporovat více formátů pro ukládání a výměnu dat jsou důvody, proč je REST v současné době převažující volbou pro vytváření API. API, které používá rozhraní REST, se označuje jako *RESTful*.

2.2. REST frameworky

Framework je sada komponent, která pomáhá vyvíjet rychleji a snadněji. Existuje velké množství frameworků v různých jazycích, které slouží k implementaci webové služby. Moderní frameworky pomáhají vytvářet REST aplikace tím, že umožňují snadno definovat patřičné procedury pro všechny potřebné CRUD metody.

2.2.1. Python: Flask, Django

Django je svobodný a open-source komplexní webový framework napsaný v jazyce Python. Na webových stránkách jsou uvedené důležité vlastnosti daného frameworku [9]:

1. Velké množství knihoven včetně REST nadstavby
2. Autentizační pravidla včetně možnosti použití *OAuth 1* či *OAuth 2*
3. Serializace pro objektově relační zobrazení – *ORM přístup*
4. *Model-Template-View princip*

Dokumentace k frameworku Django obsahuje informaci o použití principu HATEOAS [10]. Hyperlinkované vztahy spolu se stránkováním pomáhají vytvořit rozhraní API zjistitelné a procházetelné.

Nevýhodou frameworku Django je to, že tento framework nemůže poskytnout strojově čitelné formáty hypermédií, jako jsou HAL (*Hypertext Application Language*), JSON. Navíc nelze pomocí Django automaticky vytvářet úplné rozhraní ve stylu HATEOAS. Hlavní předností Django oproti jiným frameworkům je to, že je „*batteries included*“, což znamená přednastavené velké množství funkcionalit připravených k použití. U jiných frameworků je nezbytné vytváření těchto funkcionalit nebo dohledání k tomu dalších knihoven. Takto funguje například framework Flask. Flask je tak zvaný mikro webový framework pro programovací jazyk Python. Flask nemá vlastní ORM pro objektovou práci s databází. Na rozdíl od frameworku Django Flask nenabízí žádnou administraci. Uživatelé si mohou samostatně stáhnout nadstavbu pro práci s databází a pro administraci. Flask je distribuován pod open-source licencí a podporuje obě verze Pythonu.

Do budoucna je plánováno [11][12]:

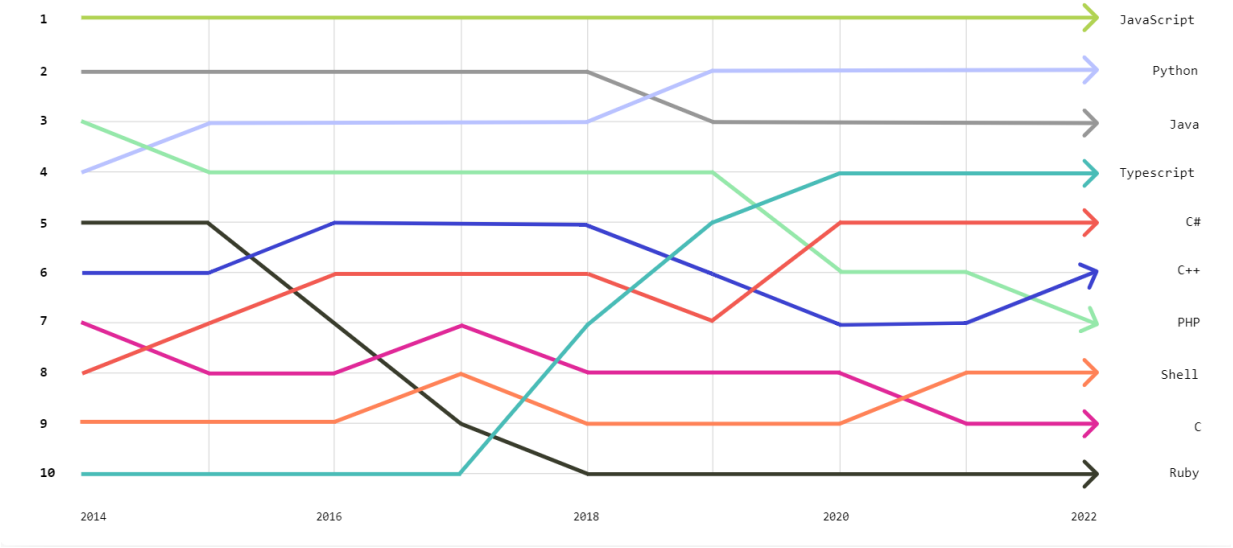
1. limitování počtu požadavků v čase
2. vylepšení procházetelnosti API
3. možnost vlastního zpracování výjimek
4. zdokumentování validace požadavků a linkování.

Flask-RESTful je rozšíření Flasku, pomocí kterého lze rychle vytvářet REST API. Dané rozšíření je závislé na Flasku, dovoluje pracovat s ORM a dalšími knihovnami. [13]

Flask-RESTful zjednodušuje tvorbu REST API oproti použití frameworku Flask, ale nepřináší žádné pokročilé funkce jako podporu autentizace a autorizace, či prolinkování a HATEOAS.

2.2.2. Java: Spring Framework

V roce 2023 je Java třetím nejoblíbenějším programovacím jazykem na světě, je to vidět na obrázku číslo 5 [14]. Popularita Javy spočívá hlavně v tom, že je to jazyk nezávislý na platformě. Tento jazyk také následuje paradigma objektově orientovaného programování a je snadné mu porozumět, psát a ladit.

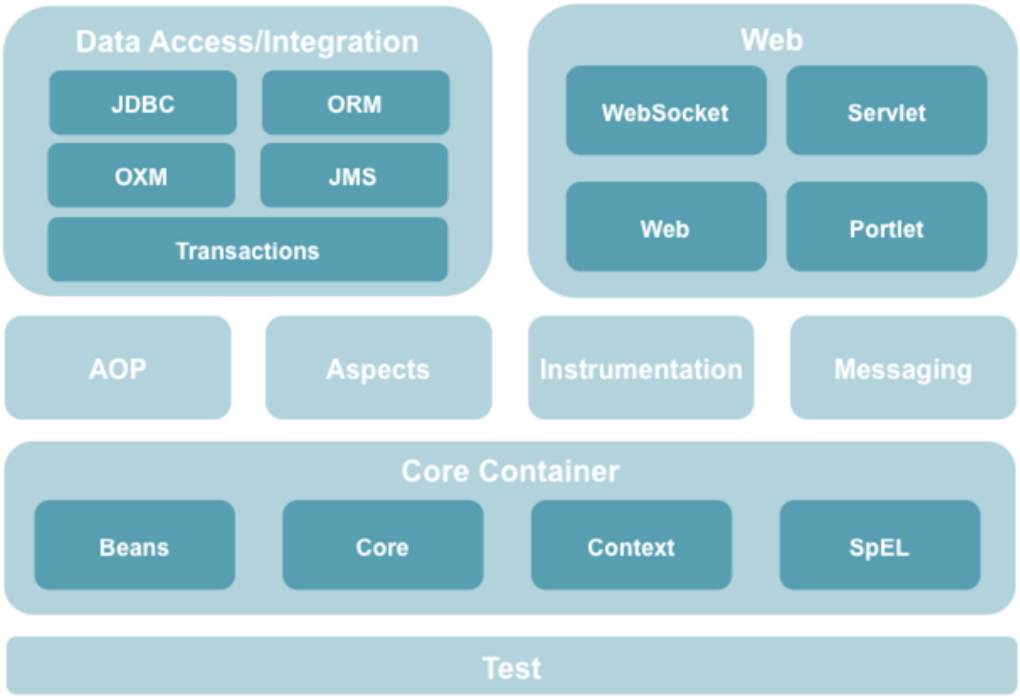


Obr. 5. Popularita programovacích jazyků na GitHub.com.
Zdroj: GitHub.com.

Pro usnadnění a urychlení vytváření aplikací v jazyce Java včetně webových služeb existuje hodně frameworků. Skládají se z předem napsaného kódu Java, tříd, šablon, komponent a dalších struktur, které můžete použít jako základ pro službu. Některé umožňují vytvářet plnohodnotné webové služby, zatímco jiné se zaměřují na frontend nebo backend.

Existují také frameworky pro konkrétní úkoly, jako je například zpracování databázových operací.

Jedním z nejoblíbenějších frameworků pro vytvoření webových služeb je Spring Framework. Spring Framework je open-source frameworkem, jehož první verze byla zveřejněna v roce 2003. Hlavním cílem Spring Frameworku je usnadnění a urychlení práce pro vývojáře. Spring Framework je od začátku modulární a obsahuje mnoho nástrojů a komponent pro různé typy aplikací. Jeho architektura je rozdělena na několik modulů, jak je ukázáno na obrázku 6 [15].



Obr. 6. Architektura Spring Frameworku.
Zdroj: https://docs.spring.io.

Seskupení komponent *Data Access/Integration* pro přístup a práci s daty se skládá z modulů JDBC, ORM, OXM, JMS a Transaction. Modul *JDBC* poskytuje abstraktní JDBC vrstvu, čímž odpadá složité JDBC programování pro získání spojení s databází, formulace dotazů, zpracování výjimek atd. Modul *ORM* umožňuje integraci vrstvy pro objektově-relační mapování objektů z frameworků jako je *Hibernate, MyBatis, Java Persitence API, Java Data Objects*. Modul *OXM* pomáhá implementovat *Object/XML* mapování. XML dokument může být strukturován pomocí modelu DOM (*Document Object Model*) nebo vstupního či výstupního proudu. Modul *JMS* slouží k zpracování komunikací mezi klienty v podobě

asynchronních zpráv, jako požadavky, události, odpovědi apod. Modul *Transactions* poskytuje řízení transakcí pro třídy, jež implementují speciální rozhraní nebo POJO objekty [16].

Skupina komponent s názvem *Web* nabízí funkce využitelné při vývoji webových aplikací. Servlet obsahuje implementaci koncepce *Model – View – Controller*. Tato část architektury Spring Frameworku je známá jako *Spring MVC Framework* a využívá se k tvorbě RESTových rozhraní.

Skupina *AOP* neboli *Aspect Oriented Programming* má za cíl nahradit v kódu opakující se činnosti. Tento modul je jednou z nejpraktičtějších vlastností Spring Frameworku. AOP umožňuje rozdělení kódu na jednotlivé aspekty, které se pak dají použít v celé aplikaci. Vzorovým příkladem takového použití je logování [17].

Součástí architektury Spring Framework je *Core Container*, který se skládá z modulů Core, Beans, Context a Expresion Language. Základnými moduly jsou Core a Beans. Tyto moduly realizují velice důležité vlastnosti Spring Frameworku jako *Inversion of Control (IoC)* a *Dependency Injection (DI).* Základem Core Containeru je implementace BeanFactory. BeanFactory realizuje návrhový vzor Factory a reprezentuje IoC kontejner. BeanFactory řídí životní cyklus POJO objektů: dovoluje vytvářet a inicializovat objekty, nastavuje vazby mezi objekty a pokud se kontejner zastaví, objekty ukončuje [18].

Testování komponent Spring Frameworku pomocí JUnit nebo Mockito se provádí pomocí modulu *Test*. Test umožňuje vytvoření takzvaných mockovacích objektů, které můžeme využít při testování kódu. Díky výše uvedeným vlastnostem komponentů architektury a díky tomu, že se Spring Framework stále vyvíjí a má spoustu nadstaveb, je v současné době jedním z nejpoužívanějších frameworků pro tvorbu webových aplikací v jazyce Java. To dokazuje jeho popularita v rámci online repozitáře GitHub a významný počet knihoven, které tento framework využívají a které jsou dostupné v Maven/Gradle repozitářích, viz tabulka 4.

Framework	Programovací jazyk	Search results GitHub	Search results PyPI (Python), Maven (Java)
Django	Python	649 887	>10 000
Flask	Python	347,685	6 875
Spring	Java	1 275 078	26 129

Tabulka 4. Popularita REST frameworků (březen 2023).

Zdroj: GitHub.com, zpracováno autorem.

Na základě srovnání Spring Framework se jeví v současné době jako nejlepší možnost pro vývoj REST webové aplikace.

2.3. Práce REST API s databází

Významný počet webových služeb komunikuje s databázemi. Frameworky, používané pro vývoj webových aplikací, většinou obsahují komponenty a knihovny, díky kterým lze velmi snadno propojit webovou službu a databázi. Například pro Spring Framework stačí jen přidat do konfiguračního souboru závislost z příslušného repozitáře Maven/Gradle, nastavit potřebné properties pro konfiguraci spojení a nainstalovat databázi. Spolupráce API a databází na rozdíl od práce uživatele přímo s databází má několik očividných výhod.

1. Snadná správa – jakmile na server bude nahraná nová verze API, hned nové možnosti při práci s databázi mohou používat všichni uživatelé.
2. Vyšší bezpečnost – webová aplikace i databáze jsou na serveru, uživatel nemusí nic stahovat a instalovat.
3. Vysoká kompatibilita – uživatelé přistupují k webové aplikaci a databázi přes webový prohlížeč, takže není podstatné, jaký operační systém má klient, aplikace funguje prakticky všude, dokonce i na mobilu.

Integrace REST API, založených například na Spring Frameworku s databází namísto použití vestavěného úložiště, poskytuje uživatelům hodně přínosů.

1. Škálovatelnost, která usnadňuje správu velkých aplikací [19]
2. Minimalizace doby provádění dotazů
3. Snížení nákladů na vývoj nových funkcionalit

Tím pádem je integrace REST API a databáze životaschopným a v mnoha případech ideálním řešením pro úkoly v různých sférách. Nejpopulárnější v současné době databáze s open-source licenci jsou MySQL a PostgreSQL. Obě zmíněné databáze jsou snadno spravovatelné, podporované v jazyce Java a mají velkou uživatelskou komunitu.

2.3.1. MySQL

MySQL, který byl poprvé vydán v roce 1995, je efektivní relační systém správy databází. Tento systém DBMS využívá jazyk *SQL (Structured Query Language)* k provádění manipulace s daty a různých operací souvisejících s daty. MySQL zásadně funguje na open-source modelu a je multiplatformní [20].

MySQL nabízí bezkonkurenční škálovatelnost, o které se psalo v předchozí podkapitole. Použití databáze místo vestaveného úložiště je důležitým přínosem integrace REST API a

MySQL. Aplikace postavené na Spring Frameworku a běžící na MySQL jsou nejen nákladově efektivní pro vývoj, ale také fungují jako dobré řešení pro úkoly na podnikové úrovni. Spring může využít MySQL ke zlepšení své schopnosti zabalit kompletní službu (například ověřování uživatelů) do samostatné formy, která také poskytuje API.

2.3.2. PostgreSQL

PostgreSQL je na rozdíl od MySQL objektově-relační databázový systém s důrazem na rozšiřitelnost a dodržování standardů. PostgreSQL neukládá pouze informace o tabulkách a sloupcích, ale umožňuje definovat i datové typy, typy indexů a funkční jazyky. Jedná se také o open-source projekt, ale hlavní rozdíl je v tom, že PostgreSQL je přizpůsoben spravování spíše velkého počtu databází. PostgreSQL je skvělý pro složité dotazy. Pokud uživatel potřebuje provádět komplikované operace čtení a zápis s použitím dat, která vyžadují ověření, je PostgreSQL správnou volbou.

Ještě jedná důležitá vlastnost PostgreSQL je tak zvaný MVCC – *Multi-Version-Concurrency Control*. MVCC umožňuje uživatelům v různých rolích interagovat a spravovat PSQL databázi současně. To eliminuje potřebu funkce read-write lock pokaždé, když někdo potřebuje interagovat s daty, čímž se zvyšuje účinnost [21]. Pro integraci REST API, například na Spring Frameworku s PostgreSQL databází, stačí provést následující kroky: vytvoření databáze, přidání závislosti Maven/Gradle, mapování dat do objektů Java a vytvoření příslušných tříd.

2.1.1. Cloud vs On-Premise řešení pro databáze

Současným trendem v IT je použití cloudových aplikačních služeb. Národní institut standardů a technologií (*NIST – National Institute of Standards and Technology*) definuje *cloud* jako model, který umožňuje uživateli přístup k výpočetním zdrojům kdekoliv a kdykoliv [22]. Výpočetním zdrojem může být v daném případě server, úložiště, aplikace, databáze apod. Cloudová databáze má licenci na bázi předplatného. Licenční poplatek pro většinu on-premise databází bývá jednorázový nebo v případě open-source licence neexistuje.

Společnosti, které preferují on-premise databáze, musejí samy udržovat hardware a software. To znamená větší náklady a větší časovou náročnost. Cloudové řešení tyto náklady částečně eliminuje. Cloud nabízí velkou rychlost při nasazení databáze, v případě on-premise databází instalace a konfigurace mohou trvat podstatně déle. Ale cloud aplikace nejsou tak plně konfigurovatelné oproti on-premise databázím.

Přestože cloud je považován dnes v mnoha případech za lepší volbu, existuje několik důvodů, proč je někdy výhodněji zůstat u on-premise řešení. Jedním z důvodů je bezpečnost. Hodně firem se zabývá otázkou, jak je bezpečný cloud. Bezpečnostní služby jsou velmi ostražité vůči cloudu, protože data řídí a kontroluje někdo z vnějšku organizace. Pokud společnost pracuje s tak zvanými citlivými daty, může být problémem sdílet tyto údaje v cloudu. Jako nevýhoda cloudu je často zmiňovaná také ztráta kontroly nad vlastní infrastrukturou. Poskytovatelé navíc často nevyvíjí databázové řešení pro specifické potřeby některých firem. Uživatelé tak obvykle nemohou snadno změnit výpočetní infrastrukturu, pokud to situace vyžaduje, naopak poskytovatelé to mohou učinit kdykoliv uznají za vhodné bez uživatelova souhlasu [23].

Vzhledem ke specifickým vlastnostem pro webovou službu, která se bude vyvíjet, bude zvolena databáze PostgreSQL ve variantě on-premise s možností použití cloudového řešení.

2.4. Generování dokumentace k REST API

Dokumentace k REST API je v podstatě její nezbytnou součástí, protože umožňuje všem, kdo pracuje s danou službou, porozumění tomu, jak API funguje. Dokumentaci je vždycky nutné udržovat, aktualizovat a verzovat. Dokumentace REST API by měla obsahovat informace o tom, jaké metody podporuje daná aplikace, jaké musí být vstupní a výstupní parametry, jaké hlavičky služeb jsou k dispozici, jaké chyby a stavy mohou nastat.

2.4.1. Specifikace Open API

V současné době pro dokumentování API často slouží *OpenAPI specifikace*. Společnost OpenAPI Initiative je podporována mnoha společnostmi, mezi nimi jsou například Google, Microsoft, IBM, SAP. Cílem Open API specifikace je poskytnutí jednotného způsobu pro popisování webových aplikací. OpenAPI umožňuje používat několik různých formátů pro dokumentování API. Mezi nejčastěji používané patří JSON a YAML [24].

JSON (*JavaScript Object Notation*) reprezentuje objektový tvar zápisu dat, nezávislý na počítačové platformě. JSON je postavený na dvou strukturách – kolekce párů “název“ : “hodnota“ a uspořádaný seznam hodnot, viz obrázek s ukázkou kódu číslo 7 [25].

```
"balances": [
  {
    "type": {
      "codeOrProprietary": {
        "code": "CLAV"
      }
    },
    "amount": {
      "value": 0,
      "currency": "CZK"
    },
    "creditDebitIndicator": "CRDT",
    "date": {
      "dateTime": "2022-10-26T10:53:18+02:00"
    }
  }
```


Obr. 7. Ukázka formátu JSON.

Zdroj: autor.

YAML (*YAML Ain't Markup Language*) slouží pro serializace strukturovaných dat. Syntaxe YAML je založena na indentaci na rozdíl od JSON [26].


```
invoice: 34843
date   : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
```


Obr. 7. Ukázka YAML.

Zdroj: https://yaml.org/.

V tabulce 5 jsou popsané základní objekty Open API specifikace.

Objekt	Stručný popis
Metadata	Metadata obsahují informace o verzi Open API specifikace. Jedná se především o verzi Open API. Poslední verze je 3.1.
Servery	Jednotlivé bázové URL serverů
Paths	Endpointy, které mohou být použité s bázovými URLs, například relativní cesta `/clients`, která ukazuje seznam klientů
Parametry	Parametry, které mohou být přidané v URL nebo v hlavičce requestu. Může to být například parametr id a uživatel po zavolání

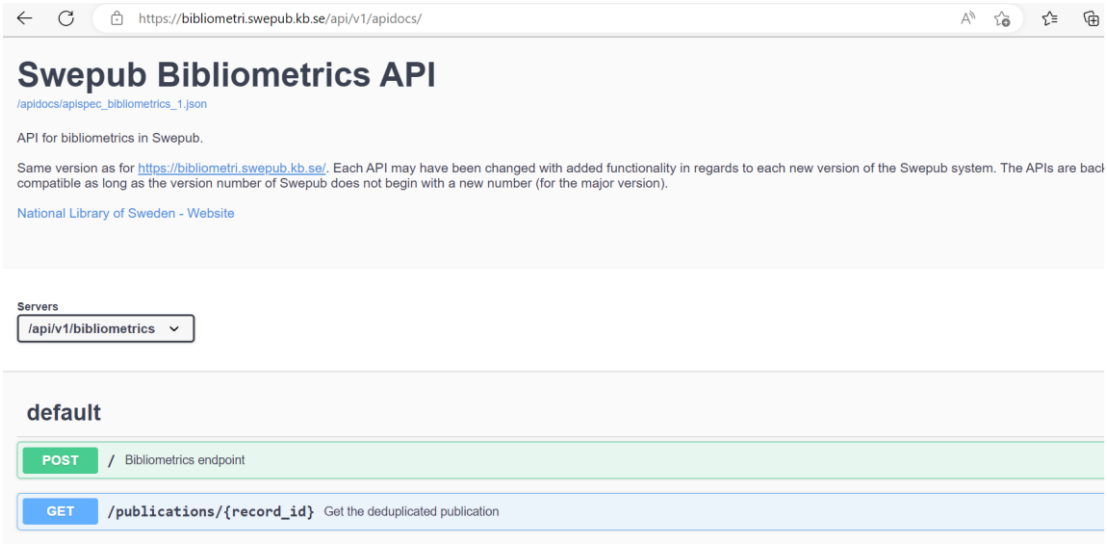
	endpointu `/clients/{id}` dostane informaci o konkrétním klientovi s daným id.
Request	Vstupní parametry pro metody POST, PUT, PATCH
Response	Popis výsledků volání endpointů, zahrnuje také stavy
Schémata	Deklarace struktury, která je předmětem volání metod API
Autentizace	Pokud API používá autentizaci, tento objekt popisuje autentizační metody a jsou tady uvedena například informace o OAuth2, API Key apod.

Tabulka 5. Základní objekty Open API specifikace.

Zdroj: https://swagger.io/specification/, zpracováno autorem.

2.4.2. Nástroje pro práci s Open API: Swagger

Pro práci s Open API existuje řada nástrojů, které jsou schopné editovat JSON a YAML soubory. Také pro psaní může být vhodné použít nástroje generující dokumentaci (např. transformace Open API do podoby HTML), různé textové a visuální editory, API pro implementaci Open API na straně serveru. Je možné použít dokonce online editor od SmartBear Software. [27] Nejsnazším způsobem je využití Swagger UI, který je součástí online editoru a generuje skvěle čitelné HTML. Swagger je open-source editor společnosti SmartBear Software, díky které vznikla Open API specifikace.



Obr. 8. Volně přístupná dokumentace k API švédské národní knihovny.
Zdroj: https://bibliometri.swepub.kb.se/api/v1/apidocs/.

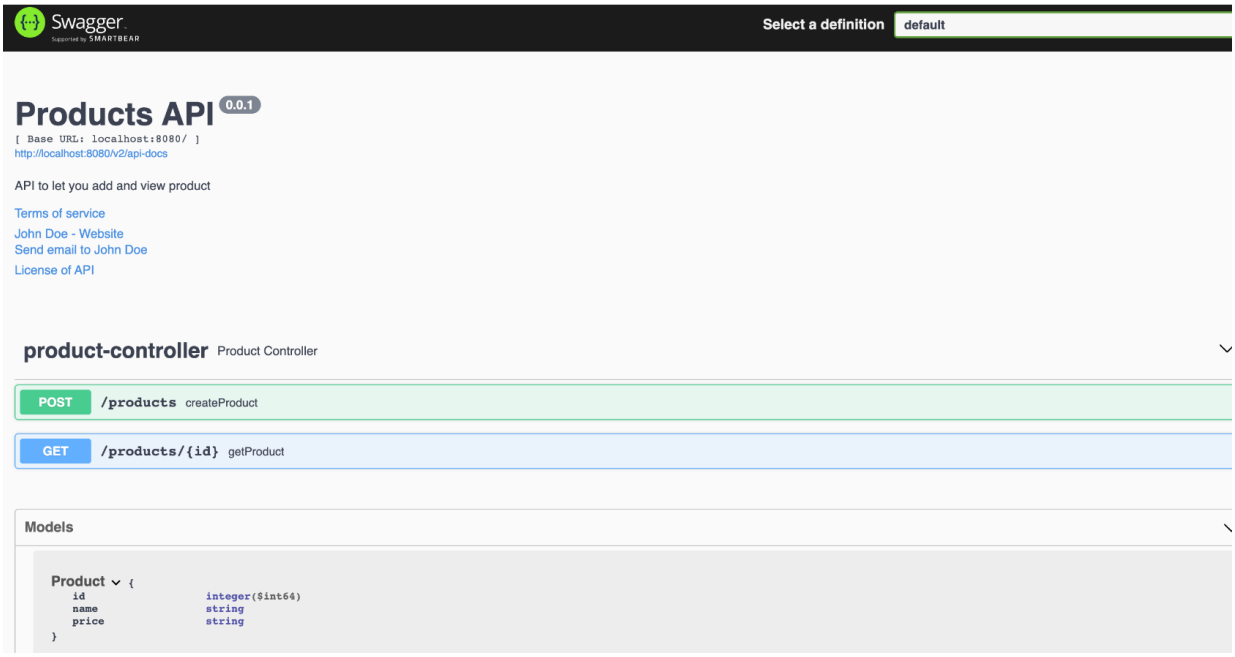
Před vytvořením rozhraní API musí tým vývojářů stanovit protokoly, zpracování chyb a modularitu kódu. Swagger poskytuje nástroje pro rychlou tvorbu prototypů a vytváření funkcí rozhraní API. Místo manuální práce s dokumentací Swagger nabízí řadu řešení pro generování, vizualizaci a údržbu dokumentace k rozhraní API. Pro kontrolu toho, že systém funguje správně, Swagger poskytuje testovací nástroj pro kontrolu požadavků a odpovědí.

Velice užitečné pro popis API jsou anotace, které se používají při tvorbě RESTových rozhraní pomocí Spring Frameworku. Například na obrázku 9 jsou uvedené anotace: `@ApiOperation("Products API"), @PostMapping("/products"), @GetMapping ("/products/{id}"), @RequestBody, @PathVariable.`

```
@RestController
@ApiOperation("Products API")
public class ProductController {

    @PostMapping("/products")
    public ResponseEntity<Void> createProduct(@RequestBody Product product) {
        //creation logic
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    @GetMapping("/products/{id}")
    public ResponseEntity<Product> getProduct(@PathVariable Long id) {
        //retrieval logic
        return ResponseEntity.ok(new Product(1, "Product 1", "$21.99"));
    }
```

Obr. 9. Ukázka kódu, který definuje metody API.
Zdroj: https://www.baeldung.com/swagger-set-example-description.

Když se projekt spustí, Swagger přečte všechny vystavené cesty a vytvoří dokumentaci, která jim odpovídá. Pak se na vytvořenou dokumentaci jde podívat na výchozí adrese URL http://localhost:8080/swagger-ui/index.html.



Obr. 10. Dokumentace s možností testování vytvořená pomocí anotací Swagger.
Zdroj: https://www.baeldung.com/swagger-set-example-description.

Také lze dokumentovat metody, parametry, chybové zprávy a modely pomocí anotací `@ApiParam`, `@ApiOperation`, `@ApiResponses`, `@ApiResponse` a `@ApiModelProperty`.

2.5. Frontend webových služeb

Pojmem frontend je označována ta část webových aplikací, jež je viditelná pro všechny uživatele neboli veřejně přístupná část webových stránek [28]. Vývoj frontendu z velké části znamená vytváření uživatelských rozhraní (UI), se kterými mohou uživatelé pracovat. Vývoj webových aplikací vyžaduje specifickou kombinaci základních nástrojů pro vytvoření frontendové části. Základ ale tvoří HTML, CSS a JavaScript. K nim se přidávají frameworky pro urychlení a usnadnění procesu vývoje.

2.5.1. Použití frameworků pro frontendovou část API

Frontendových frameworků je mnoho a skoro každý rok se objeví nový. Od roku 2017 se ale řadí mezi nejpoužívanější frameworky React, Angular a Vue.js.

React je open-source JavaScript knihovna pro vytváření uživatelských rozhraní a komponent uživatelského rozhraní. React byl vyvinut v roce 2013 a je udržován týmem vývojářů ve Facebooku. Mezi jeho hlavní výhody patří jednoduchost, flexibilita a škálovatelnost. Dalším přínosem je jeho rozsáhlý ekosystém a možnost společného využití s dalšími frameworky, jako například Angular. React je také možné vložit do již existujícího kódu bez nutnosti zbývající kód přepisovat. Navíc React podporuje využití TypeScriptu [29].

Angular je open-source JavaScript framework využívaný v současnosti pro vývoj single-page a progresivních webových aplikací. Angular byl vytvořen v roce 2016 týmem společnosti Google. Angular je druhým nejpoužívanějším frontendovým frameworkem. K jeho výhodam patří strukturovanost a snadná testovatelnost. Na rozdíl od Reactu Angular nevyžaduje stahování dalších knihoven, jelikož většinu potřebných funkcí již obsahuje. Angular také podporuje TypeScript [30].

Vue je progresivní open-source JavaScript framework, který slouží pro vytváření uživatelského rozhraní single-page aplikací. Označení progresivní získal díky své flexibilitě, rychlosti, intuitivní srozumitelnosti. Vue.js dovoluje vytvořit celou aplikaci, ale také vývojář může vkládat části kódu do své již existující aplikace. Vue byl vyvinut bývalým zaměstnancem Google Evanem You v roce 2014. Ačkoli za Vue.js nestojí velká korporace, získal si velkou komunitu podporovatelů. Vue.js je třetí nejpoužívanější z frontendových frameworků a počet jeho uživatelů stále roste [31].

2.5.2. Použití šablonovacího nástroje Thymeleaf

Uživatelské rozhraní na straně prohlížeče, které vývojář může vytvořit pomocí frameworků jako Angular, React, Vue a další, přebírá data ve formátu JSON z aplikace Spring Boot na straně serveru a prezentuje je pomocí JavaScriptu v prohlížeči. Výhodou bude podpora rychlé a plynulé odpovědi uživatelského rozhraní na kliknutí tlačítka v moment, kdy uživatel nepotřebuje další připojení na straně serveru k načtení nových dat.

Existují ale i určité nevýhody takového řešení. Jedna z podstatných – použití jiného programovacího jazyka, než ten, který byl použit pro tvorbu serverové části. Pokud vývojář chce mít jednotnou aplikaci, již může udržovat v jednom jazyce, dobrou volbou bude šablonovací nástroj Thymeleaf.

Thymeleaf je open-source software, licencovaný pod *Apache License 2.0* [32]. Poskytuje plnou integraci se Spring Frameworkem. Ve webových aplikacích Thymeleaf může být kompletní náhradou *JavaServer Pages* (*JSP*) [33] a implementuje koncept *Natural*

Templates: soubory šablon, které lze přímo otevřít v prohlížečích a které se stále zobrazují správně jako webové stránky. Thymeleaf generuje HTML na serveru Spring Boot a vrací ho jako plnou webovou stránku. Thymeleaf oproti JSP má výhodu v tom, že umožňuje rychlé a snadné vytváření dynamických webových stránek bez nutnosti psát složitý kód [34].

Následující ukázka kódu na obrázku 11 vytvoří tabulku HTML s řádky pro každou položku proměnné `List<Product>` s názvem `allProducts`.

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price,1,2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

Obr. 11. Ukázka HTML kódu s použitím Thymeleaf.
Zdroj: autor.

Pro účel této práce – vývoj webové služby, která požaduje, aby klient obdržel webovou stránku, nejen text odpovědi, bude dobrou volbou použití Thymeleaf. Není potřeba, aby byla vyvinutá složitá frontendová část. Šablonovací nástroj Thymeleaf pro danou aplikaci bude úplně dostačující.

2.6. Testování webových aplikací

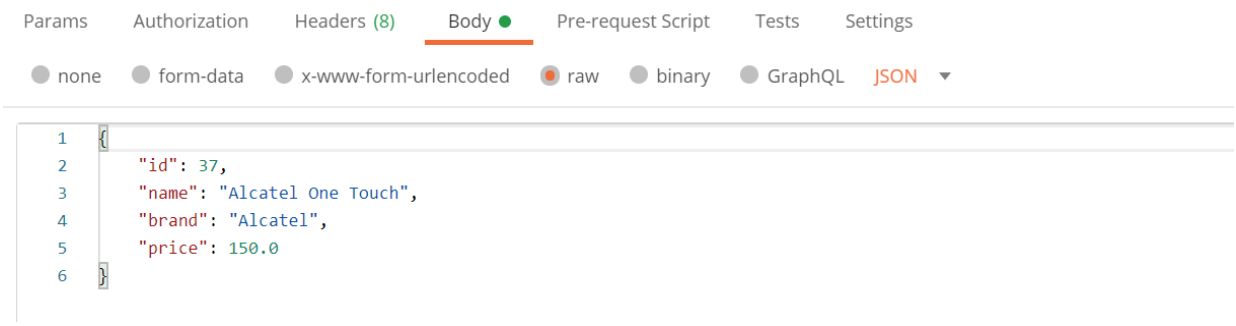
Testování webových služeb je založeno na komunikaci klient-server. Klient odesílá HTTP požadavek na webový server, server zpracuje daný požadavek a vrátí odpověď. Tento druh testů se nazývá *integrační testy*. Pomocí integračních testů se testuje několik komponent webové služby zároveň [35]. To znamená, že k integračnímu testování API je důležité mít aplikaci, která bude odesílat požadavky na server a přijímat odpovědi. Jako základní nástroj pro manuální testování může sloužit i webový prohlížeč nebo různé pluginy. V současné době se ale k integračnímu testování nejčastěji používá open-source testovací nástroj Postman, o kterém bude podrobněji pojednáno níže.

2.6.1. Principy testování REST API

Při testování REST API se postupuje od menších částí k větším. To znamená, že nejdřív se provádí testování komponent – funkcí, tříd, modulů. Takové testy se nazývají jednotkové neboli *unit-testy* [36]. Pokud komponenta, zvolená k testování, komunikuje s jinými komponentami, čili na nich závisí, tak se většinou místo nich používají tak zvané *mocky*. Pro psaní unit-testů v Javě existuje několik frameworků, ze kterých je nejpoužívanější JUnit. Často se JUnit integruje s knihovnou Mockito, což bude popsáno v příslušné kapitole.

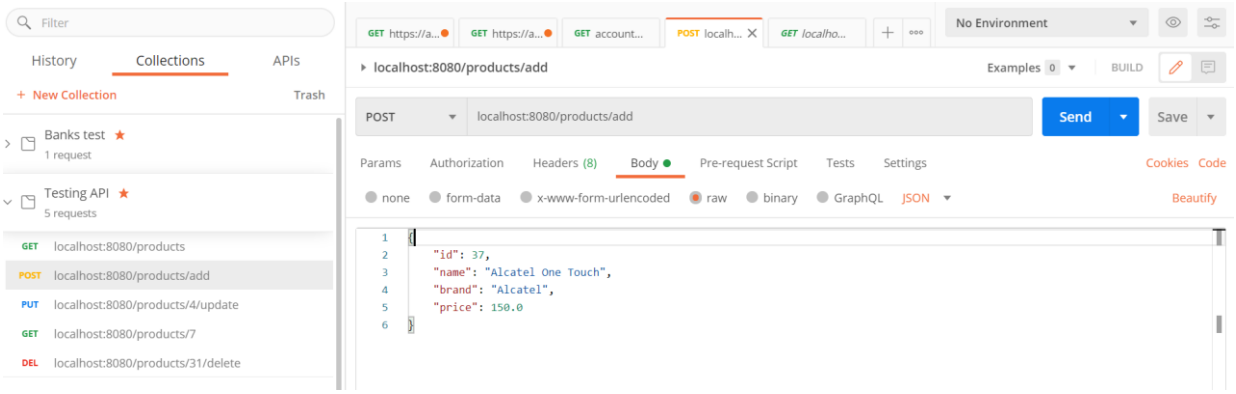
2.6.2. Nástroje pro testování webových API: Postman

Postman je API klient používaný k vývoji, testování a dokumentování API. Postman umožňuje provádět testování zadáním adresy URL endpointu. Požadavek bude poté odeslán na server a ten odpoví zpět aplikaci Postman. Na obrázku 13 se volá POST metoda, která přidává do databáze nový produkt – endpoint `/products/add` s požadavkem na obrázku 12.



Obr. 12. Tělo požadavku.

Zdroj: autor.



Obr. 13. Testování API v Postmanu.

Zdroj: autor.

Postman podporuje různé metody endpointů a POST je jedním z nich. Také vývojář nebo tester může volat GET, PUT, PATCH a DELETE metody. Při odesílání requestu je možné zvolit autorizaci i pomocí metod, jako je OAuth. Všechna volání se ukládají do historie, kde je možné je otevřít, znovu odeslat nebo uložit. Postman nabízí i užitečnou možnost seskupit volání do složek. Důležitá je funkce nastavení pro složku stejné autorizace nebo nastavení proměnných [37]. Proměnné jde pak použít v requestech s dvojitými složenými závorkami, například: `{{url}}/clients`.

2.6.3. Použití jednotkových testů: JUnit, Mockito

Hlavní cíl použití jednotkových testů je detekce chyb a ve výsledku snížení počtu chyb v programu. Existují také i vedlejší cíle, jako explicitní dokumentace chování a zlepšení kvality kódu. Jeden z nejpopulárnějších Java frameworku pro tvorbu jednotkových testů je teď JUnit.

Hlavní vlastností JUnit:

1. Nastavení a odstranění kontextu před a po spuštění testu.
2. Zadání očekávaného výstupu a porovnání ho s přijatým výstupem.
3. Testování výjimky
4. Integrace s Maven nebo Gradle

Některé často používané anotace v JUnit [38]:

`@Test`: ukazuje, že toto je testovací metoda, návratový typ: `public void`

`@Before`: musí se spustit před `@Test`, návratový typ: `public void`

`@BeforeEach`: nastavení před spuštěním každého testu

`@After`: musí se spustit po `@Test`, návratový typ: `public void`

`@Ignore`: blokuje spuštění testovacího případu

```
public boolean isNumberEven(Integer number) {
    return number % 2 == 0;
}
```

Obr. 14. Ukázka kódu pro Junit testy.
Zdroj: https://junit.org.

Tato metoda by měla vrátit `true`, pokud je předaný argument sudé číslo a v opačném případě `false`. Unit test bude ověřovat, zda to funguje tak, jak má [39].

```java
@Test
void givenEvenNumber_whenCheckingIsNumberEven_thenTrue() {
    boolean result = bean.isNumberEven(8);

    Assertions.assertTrue(result);
}

@Test
void givenOddNumber_whenCheckingIsNumberEven_thenFalse() {
    boolean result = bean.isNumberEven(3);

    Assertions.assertFalse(result);
}
```

Obr. 15. Ukázka unit testů k metodě `isNumberEven()`.
Zdroj: https://www.baeldung.com/junit-5-test-annotation.

Pokud je důležitá kontrola časového limitu nebo vyvolanou výjimku, jde použít místo `assertTrue()` a `assertFalse()` výrazy `assertTimeout()` a `assertThrows()`:

```java
@Test
void givenLowerThanTenNumber_whenCheckingIsNumberEven_thenResultUnderTenMillis() {
    Assertions.assertTimeout(Duration.ofMillis(10), () -> bean.isNumberEven(3));
}

@Test
void givenNull_whenCheckingIsNumberEven_thenNullPointerException() {
    Assertions.assertThrows(NullPointerException.class, () -> bean.isNumberEven(null));
}
```

Obr. 16. Ukázka testování vyhození výjimky a timeoutu.
Zdroj: https://www.baeldung.com/junit-5-test-annotation.

Mockito je open-source testovací framework používaný pro jednotkové testování Java aplikací. Mockito se používá k vytváření *mocků* – fiktivních objektů, které nahrazují například skutečnou databázi, jinou API službu nebo nějakou službu, se kterou testovaný objekt má komunikovat [40].

K vytvoření mockovacích objektů slouží metoda `mock()` a anotace `@Mock`. Jako argument mock metody se může používat `anyInt()`, `any()`, `anyString()`. Pro zavolání mock metod se píše kód jako: `doReturn(result).when(mock_Object)`.

`method_Call()`. Pokud se očekává výjimka, používá se metoda `doThrow()` nebo `thenThrow()`. Metoda `verify()` kontroluje, že všechno po zavolání splňuje očekávání.

3. IMPLEMENTACE REST APLIKACE

Tato kapitola se zabývá tím, jak implementovat REST API s použitím frameworku Spring Boot a programovacího jazyka Java. V dané části je podrobněji popsána realizace REST API, použité technologie a řešení. Pro vývoj aplikace je použit programovací jazyk Java a verze 11 JDK, kterou lze zdarma stáhnout z oficiální stránky Oracle. Pro lepší přehlednost a čitelnost reprezentace zdrojů v API je plněna formátem JSON. Pro mapovaní hodnot z databáze do objektu v Java je použit framework Spring JPA, jelikož nabízí dobrou konfigurovatelnost a jednoduchou práci s SQL. Také je podrobněji popsáno logování, vytvoření dokumentace pomocí Swaggeru, testování v Postmanu a frontendová část webové aplikace. Frontendová část je vytvořena s použitím frameworku Thymeleaf.

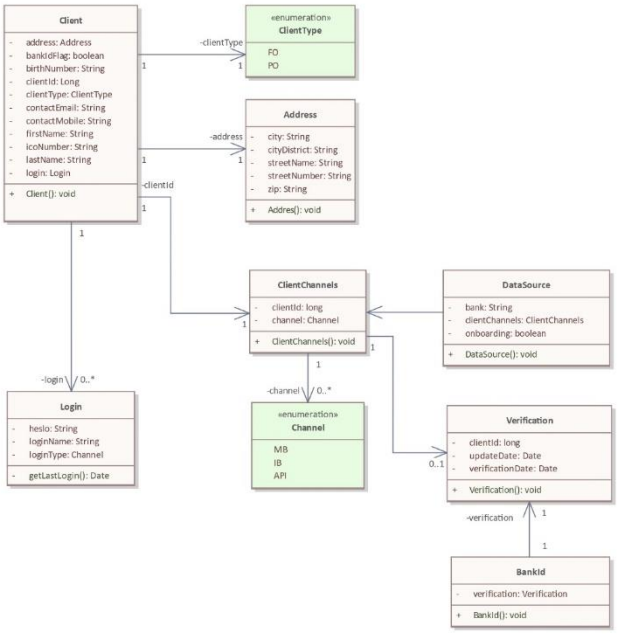
Jako vývojové prostředí je použito IntelliJ Idea 2020. Důvodem dané volby jsou velké nabídky různých pomocných nástrojů zabudovaných přímo uvnitř prostředí. Pro vývoj API byl použit Spring Boot, což je nadstavba pro Spring Framework. Dalším z použitých nástrojů je Apache Maven. Tento nástroj se používá pro řízení a spravování projektu. Projekt obsahuje soubor POM.xml, ve kterém je konfigurace daného nástroje. Maven řídí sestavení projektu a doplňuje závislosti potřebné pro úspěšnou kompilaci aplikace. V této práci je použita databáze PostgreSQL, která je dostupná pro instalaci zdarma. Mezi její výhody patří jednoduchá rozšiřitelnost a stabilita.

Cílem práce je navrhnout rozhraní webové aplikace pro zpracování testovacích údajů klientů banky z databáze PostgreSQL. Výsledná data se musí zobrazit na webovém portálu. Webový portál prostřednictvím REST API nabídne základní vyhledávání subjektů, zadání nových dat do formuláře, modifikaci dat a jejích odstranění. Výsledky poskytne aplikace formou jednoduchého reportu. Dokončená aplikace bude fungovat podle očekávání a bude splňovat všechny druhy požadavků (architektonické, bezpečnostní, uživatelské požadavky atd.).

Při vývoji aplikací je důležitým způsobem, jak zajistit, zda aplikace funguje podle očekávání, testování. Správnost fungování aplikace bude zařízena prostřednictvím Postmanu. Navíc při vývoji budou použity jednotkové testy.

3.1. Architektura projektu: Spring Framework

Cílem této práce je návrh a implementace webové služby, která poskytne k zpracování uměle vytvořené testovací údaje klientů banky. Pro tento účel je důležité, aby záznamy obsahovaly kromě základních údajů informaci o existenci bankovní identity klienta. Údaje budou uložené do databáze PostgreSQL, se kterou REST API bude interagovat. Struktura projektu je vidět na obrázku 17. Pro vytváření diagramu tříd byla použitá aplikace Enterprise Architect.



Obr. 17. Diagram tříd projektu.
Zdroj: autor.

Dané REST API podporuje všechny CRUD metody: GET – pro zobrazení seznamu klientů, PUT – pro update údajů, POST – pro přidání nového záznamu, DELETE – pro odstranění údajů klienta. Pro implementaci API byl použit Spring Boot, což je nadstavba Spring Frameworku. Spring Boot usnadňuje použití frameworku Spring a zrychluje proces vývoje. Výhodou použití Spring Boot je již zmíněná rychlost při startu projektu, snadná konfigurace a integrovaný server *Apache Maven* [41]. To znamená, že není třeba konfigurovat server ve vývojovém prostředí, stačí jen spustit aplikaci z hlavní třídy.

Pro snadnější realizaci projektu byl použit nástroj *Spring Initializr* [42], který generuje strukturu projektu podle požadavků a dovoluje přidat potřebné závislosti. Základním souborem

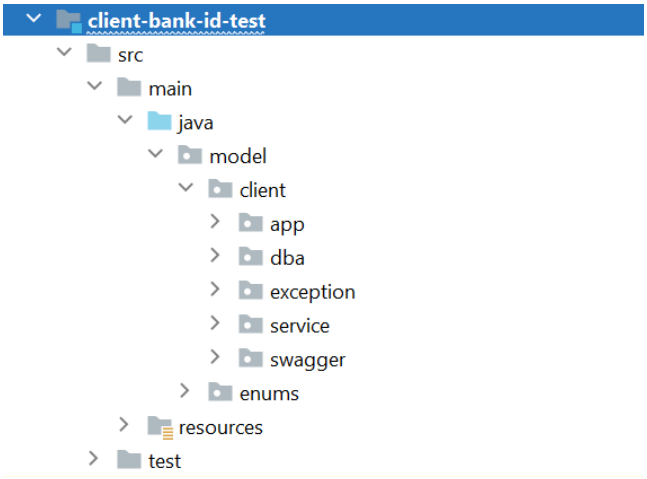
pro Maven je soubor `pom.xml` (POM je zkratka pro *Project Object Model*). V tomto souboru jsou informace o projektu včetně využívaných knihoven, nastavení, verzí apod [43].

```
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.2.6.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
        <version>2.2.6.RELEASE</version>
    </dependency>
```

Obr. 18. Ukázka souboru `pom.xml`.

Zdroj: autor.

Spring Boot je postavený na MVC architektuře. To znamená, že aplikace je rozdělená do 3 logických vrstev, které se jmenují *Model, View, Controller* [44]. V souvislosti s tím byla vytvořena struktura projektu, viz obrázek 19.

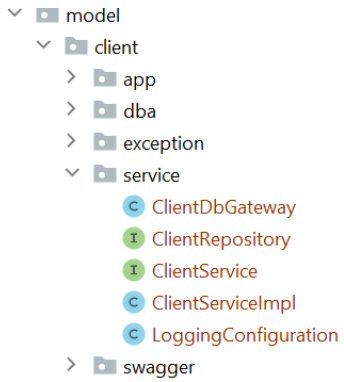


Obr. 19. Struktura projektu.

Zdroj: autor.

3.1.1. Model

Balíček `model` obsahuje logiku aplikace. Jelikož je použité ORM – objektově-relační mapování, třída `Client` v balíčku `model/dba/` koresponduje s tabulkou v databázi. Takže v balíčku `model/dba/` se nachází třídy `Address`, `ClientChannels`, `BankId`, `DataSource`, `Login`, `Verification`. Instance modelů obsahují atributy z databáze. Na obrázku číslo 20 je vidět, že také v balíčku `model` jsou balíčky `service`, `exception`, `swagger`, které slouží pro umístění tříd, spojených se zpracováním vstupních dat, výjimek, s konfigurací Swaggeru atd.



Obr. 20. Struktura balíčku `model/service`.

Zdroj: autor.

Aby uživatelé aplikace mohli přidat nového klienta, smazat údaje nebo údaje klienta změnit, je potřeba mít třídu `Client`, která reprezentuje klienta banky. Třída Client obsahuje jedinečný identifikátor klienta – `clientId` typu `Long`, viz ukázku kódu na obrázku číslo 21.

```
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Data
@Entity
public class Client {
    @ApiModelProperty("Adresa klienta")
    private String address;

    @ApiModelProperty("Priznak zda klient ma aktivni bankovni identitu")
    private boolean bankIdFlag;

    @ApiModelProperty("Rodne cislo klienta v pripade fyzicke osoby")
    private String birthNumber;

    @ApiModelProperty("Identifikator klienta")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long clientId;
```

Obr. 21. Třída `Client`.

Zdroj: autor.

Každý klient v databázi má adresu typu `String`, která pro další účely je rozdělena a namapována na atributy ve třídě `Address`. Metoda `mapAddress()` se nachází ve třídě `ClientServiceImpl`:

```
private Address mapAddress(Long clientId) {
    String address = getClient(clientId).getAddress();
    String[] splittedAddress = address.trim().split( regex: ",");
    Address formattedAddress = new Address();
    formattedAddress.setCity(Optional.ofNullable(splittedAddress[0]).toString());
    formattedAddress.setCityDistrict(Optional.ofNullable(splittedAddress[1]).toString());
    formattedAddress.setStreetName(Optional.ofNullable(splittedAddress[2]).toString());
    formattedAddress.setStreetNumber(Optional.ofNullable(splittedAddress[3]).toString());
    formattedAddress.setZip(Optional.ofNullable(splittedAddress[4]).toString());
    return formattedAddress;
}
```

Obr. 22. Ukázka kódu pro mapování adresy klienta.
Zdroj: autor.

Některé atributy tříd mají *anotace*. Anotace z balíčku `javax.persistence.*` jsou určené pro ORM funkcionality, viz tabulku 6. Na obrázku 21 je vidět take Swagger anotace a anotace knihovny Lombok, které budou podrobně popsané v příslušných kapitolách.

Anotace	Popis
`@Id`	Primární klíč tabulky, které odpovídá entita
`@GeneratedValue`	Pro daný parametr bude automaticky vygenerována hodnota při vkládání objektu do databáze
`@OneToMany`	Určuje *one-to-many* vztah mezi entitami
`@ManyToOne`	Určuje *many-to-one* vztah mezi entitami

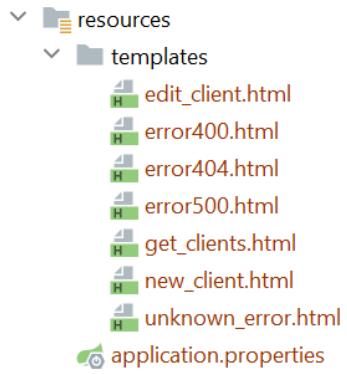
Tabulka 6. Anotace, které jsou používané v entitních třídách.
Zdroj: autor.

3.1.2. View

Vrstva *View* se stará o zobrazení výstupu uživateli. V projektu se nachází složka `resources`, obsahující HTML stránky, jež odpovídají endpointům. Pro implementaci byl

použit šablonovací systém Thymeleaf, který dovoluje vkládat do HTML šablony proměnné. Aplikace obsahuje následující pohledy:

1. zobrazení seznamu klientů – soubor `get_clients.html`
2. stránka, která je určená k editaci údajů – soubor `edit_client.html`
3. stránka, která je určená k přidání nového klienta – soubor `new_client.html`
4. stránky, které zobrazují odpovídající chybový HTTP stav – `error*.html`



Obr. 23. Obsah balíčku `resources`.
Zdroj: autor.

Realizované pohledy obsahují minimální množství logiky, která je pro výpis nutná, například *input field*, které je v režimu *read-only* apod. Podrobněji o šablonovacím systému Thymeleaf bude popsáno v příslušné kapitole.

3.1.3. Controller

Vrstva Controlleru je v podstatě prostředníkem, se kterým komunikuje uživatel. Uživatel zadává do prohlížeče adresu webové aplikace a parametry, které upřesní, kterou stránku je třeba zobrazit, totiž jaký endpoint se musí volat. Vyvinutá webová služba obsahuje dva kontroléry:

1. pro volání uživatelem v prohlížeči – `ClientFrontendController`
2. pro volání jinými webovými službami – `ClientController`.

Aby Spring Framework věděl, že třída `ClientFrontendController` je kontrolérem a správně směroval požadavky, je použitá anotace `@Controller`. Může být použita také anotace `@RestController`, která zajišťuje reprezentaci všech návratových objektů této třídy do formátu JSON nebo XML. Tato anotace je použitá pro `ClientController`. Na obrázku 24 je vidět, že metoda `getClients()` reprezentuje

endpoint, který přijímá GET požadavek s obsahem ve formátu JSON a vrátí odpověď – HTML stránku se seznamem klientů.

```
@ApiOperation(
        value = "Show list of clients for bank id tests.",
        notes = "Intended to return a list of clients in the browser.")
@ApiResponses({
        @ApiResponse(code = 400, message = "Bad request.Try it again please."),
        @ApiResponse(code = 404, message = "We could not this page on our servers."),
        @ApiResponse(code = 500, message = "Something went wrong.")
})
@RequestMapping("/")
public String getClients(@ApiParam(
        name = "keyword",
        type = "String",
        value = "Keyword for searching",
        required = true) Model model, @Param("keyword") String keyword) {
    List<Client> clients = clientService.getClients();
    model.addAttribute( s: "clients", clients);
    model.addAttribute( s: "keyword", keyword);
    log.info("List of clients was downloaded successfully.");

    return "get_clients";
}
```

Obr. 24. GET endpoint `ClientFrontendControlleru`.

Zdroj: autor.

Volání tohoto endpointu vyhledá klienty v databázi a vrátí jejich údaje. Dále se volá další metoda třídy `ClientService`, která například kontroluje, jestli daný klient je fyzickou osobou, má rodné číslo a má aktivovanou bankovní identitu. Tyto údaje se ukládají do proměnných. Nakonec proměnné s příslušnými daty jsou předány do vrstvy `view`. Anotace `@Autowired`, vztahující k třídě `ClientService`, je součástí návrhového vzoru *Dependency Injection*. Tato anotace říká, že do třídy `ClientFrontendController` je vložená instance třídy `ClientService`. Na obrázku 25 je ukázka DELETE endpointu `ClientControlleru`, který je určen k odstranění údajů klienta podle jeho `clientId`.

```
@ApiOperation(
        value = "Delete client by ID.",
        notes = "Intended to delete client by ID when calling other APIs..")
@ApiResponses({
        @ApiResponse(code = 400, message = "Bad request.Try it again please."),
        @ApiResponse(code = 404, message = "We could not this page on our servers."),
        @ApiResponse(code = 500, message = "Something went wrong.")
})
@DeleteMapping("/delete/{id}")
public List<Client> deleteClient(@ApiParam(
        name = "clientId",
        type = "Long",
        value = "Client ID",
        required = true) @PathVariable Long clientId) {
    clientService.deleteClient(clientId);
    log.info("Client ID" + clientId + " was successfully deleted.");
    return clientService.getClients();
}
```

Obr. 25. DELETE endpoint `ClientControlleru`.

Zdroj: autor.

3.1.4. Service

Servisní vrstva aplikace obsahuje zaprvé třídu rozhraní `ClientRepository`, poděděnou z `JpaRepository<Client, Long>`. Rozhraní `JpaRepository<Client, Long>` obsahuje metody, které dovolují pracovat s entitami v databázi: `findAll()`, `findById()`, `save()`, `deleteById()`. Díky tomu, že `ClientRepository` je injektována do třídy `ClientServiceImpl`, můžeme tyto metody použit pro vytváření vlastních metod: `getClients()`, `saveClient()`, `getClient()`, `deleteClient()`. Třída `ClientServiceImpl` implementuje rozhraní `ClientService`, tím se uplatňuje princip zapouzdření, kód je průhlednější, rozšiřitelnější pro budoucnost.

`ClientServiceImpl` má anotace `@Service` a `@Transactional`. Anotace `@Transactional` se používá pro služby související s databázi, což je skupina CRUD-metod vyvinuté aplikace. Použití anotace `@Service` dává možnost toho, že Spring Framework vytvoří instanci dané třídy a bude ji spravovat jako *bean*. Do *bean* Spring vkládá závislosti, čímž mohou být jiné bean nebo literální hodnoty jako String, primitivní typy, kolekce hodnot [45]. Kromě metod, souvisejících s databází, `ClientServiceImpl` obsahuje následující metody: určuje, jestli je klient fyzická nebo právnická osoba, jestli mu přiděleno rodné číslo nebo IČO.

Na obrázku 26 je ukázka kódu ve třídě `ClientServiceImpl`.

```java
@Service
@Transactional
public class ClientServiceImpl implements ClientService {

    @Autowired
    private ClientRepository repo;

    @Autowired
    private ClientDbGateway dbGateway;

    public List<Client> getClients() { return repo.findAll(); }

    public void saveClient(Client client) { repo.save(client); }

    public Client getClient(Long clientId) { return repo.findById(clientId).get(); }

    public void deleteClient(Long clientId) { repo.deleteById(clientId); }
```

Obr. 26. Metody třídy ClientServiceImpl.

Zdroj: autor.

3.1.4. Databázová vrstva

Pro vyvinuté REST API byla použita databáze PostgreSQL. PostgreSQL je objektově-relační databázový systém s důrazem na rozšiřitelnost a dodržování standardů. Má hodně funkcí a je škálovatelný, což je důležité pro budoucnost aplikace. `ClientService` dědí rozhraní `JpaRepository` z modulu *Spring Data JPA*. Spring Data JPA poskytuje podporu databáze pro *Java Persistence API* (JPA), co značně usnadňuje vývoj aplikací, které potřebují přístup ke zdrojům dat JPA [46].

Některé z konfiguračních parametrů přístupů do databáze Spring Boot vytváří automaticky. Ale základní konfiguraci je nutné definovat v speciálním konfiguračním souboru `application.properties`. V tomto souboru jsou uvedené například přihlašovací údaje do databáze:

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.properties.hibernate.dialect=com.example.PostgreSQL94CustomDialect
spring.jpa.properties.hibernate.format-sql=true
spring.jpa.show-sql=true
logging.file.name=logging.txt
logging.file.path=C:/Users/User/git/repository10/.git/ClientBankIdTest
```

Obr. 27. Konfigurace projektu v souboru `application.properties`.

Zdroj: autor.

3.2. OpenAPI Specification – dokumentace k projektu

API dokumentace je jednou z důležitých součástí při vývoji aplikací. Swagger je specifikace, která definuje způsob, jak popsána a zadokumentovaná vyvinutá webová služba. Aby uživatele se mohli seznámit s webovou službou a vyzkoušet jednotlivé endpointy, Swagger vytváří stránku s dokumentací. Pro implementaci byla použitá konfigurace Swagger – nachází se v balíčku `swagger` ve třídě `ClientSwaggerConfiguration` a také byly použity anotace. Pro použití uživatelského rozhraní Swagger UI, do souboru `pom.xml` byla přidaná další závislost Maven – `springfox-swagger-ui`.

```
@Configuration
@EnableSwagger2
public class ClientSwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build()
                .apiInfo(new ApiInfoBuilder().title("ClientBankIdTest application").build());
    }
```

Obr. 28. Konfigurace Swagger UI.

Zdroj: autor.

Na obrázku 28 je ukázka konfigurace Swagger UI. Třída `ClientSwaggerConfiguration` má metodu `api()`, která vyrobí objekt typu *Docket*. Docket bere název balíčku, který obsahuje repository a kontroléry, jež je nutné zobrazit na stránce dokumentace pro uživatelé.

Třídy `ClientFrontendController` a `ClientController` obsahují anotace, viz tabulka 7.

Anotace	Popis
`@ApiOperation`	Definuje vlastnosti metody rozhraní API. Název operace se přidává pomocí vlastnosti value a popis pomocí vlastnosti notes.

`@ApiResponses`	Používá se k určení zpráv, které doprovázejí kódy odpovědí. Pro každou zprávu odpovědi je důležité, aby byla přidaná anotace `@ApiResponse`.
`@ApiParam`	Definuje vlastnosti parametrů metody
`@ApiModelProperty`	Definuje vlastnosti polí

Tabulka 7. Anotace Swagger UI.
Zdroj: swagger.io.

V prohlížeči je možné otevřít Swagger UI na adrese:
`http://localhost:8080/swagger-ui/`.

3.3. Java knihovna Lombok

Podle dokumentace Lombok je knihovnou Java, anotačním procesorem, který při kompilaci přidává do kódu gettery, settery, konstruktory [47]. To je velmi užitečný nástroj, který pomáhá ušetřit hodně času. Ve vyvinuté aplikaci byly použity následující anotace Lombok, viz tabulka 8.

Anotace	Popis

`@Data`	Anotace, která spojuje
	`@RequiredArgsConstructor`
	`@Getter`
	`@Setter`
	`@ToString`
	`@EqualsAndHashCode`

`@NoArgsConstructor`	Generuje konstruktor bez parametrů
`@AllArgsConstructor`	Generuje konstruktor vyžadující argument pro každé pole v anotované třídě

`@RequiredArgsConstructor`	Generuje konstruktor s požadovanými argumenty
`@Builder`	Anotace pro vytváření instance třídy

Tabulka 8. Anotace Lombok, použité v projektu.
Zdroj: https://projectlombok.org.

3.4. Ošetření chyb a logování

REST API vyvinuté pomocí Spring Boot automaticky zpracovávají chyby. Pokud dojde k chybě, bude vrácena odpověď obsahující informace o typu chyby. Tyto informace často mohou být nedostatečné pro uživatele, aby se s chybou správně vypořádali. Proto tato webová služba má implementovanou vlastní logiku zpracování výjimek pomocí třídy s anotací `@ControllerAdvice`, viz obrázek číslo 29:

```
@ControllerAdvice
@Slf4j
public class ClientErrorHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ClientNotFoundException.class)
    public final ResponseEntity<ErrorResponse> handleNotFoundException(ClientNotFoundException ex) {
        ErrorResponse errorResponse = new ErrorResponse("NOT FOUND EXCEPTION");
        log.error("Client was not found");
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(ClientBadRequestException.class)
    public final ResponseEntity<ErrorResponse> handleBadRequestException(ClientBadRequestException ex) {
        ErrorResponse errorResponse = new ErrorResponse("BAD REQUEST EXCEPTION");
        log.error("Request was wrong");
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
```

Obr. 29. Ukázka kódu zpracování chyb pomocí anotace `@ControllerAdvice`.
Zdroj: autor.

Anotace `@ControllerAdvice` umožňuje řešit zpracování chyb v celé aplikaci a centralizovat logiku zpracování. Také tato anotace poskytuje možnost kontroly odpovědi a stavového kódu HTTP. Tak označená `@ControllerAdvice` třída `ClientErrorHandler` obsahuje metody:

- `handleNotFoundException(ClientNotFoundException ex)`
- `handleBadRequestException(ClientBadRequestException ex)`
- `handleInternalServerError(ClientInternalServerErrorException ex)`

Metoda zpracuje příslušnou výjimku a vrátí objekt `ResponseEntity<ErrorResponse>`. Třída `ErrorResponse` obsahuje informaci o chybě, kterou ukládáme do proměnné `message`, viz obrázek číslo 30:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ErrorResponse {

    private String message;
}
```

Obr. 30. Třída `ErrorResponse`.

Zdroj: autor.

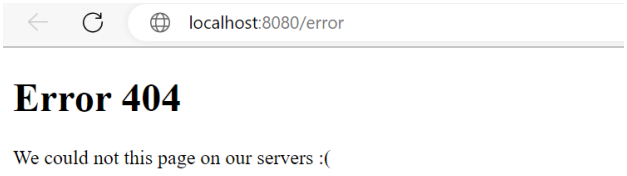
Pro větší pohodlí uživatele pomocí Thymeleaf byly vytvořené vlastní stránky chyb `400 Bad Request`, `404 Not Found`, `500 Internal Server Error` a třída `ClientCustomErrorController` viz obrázek číslo 31:

```
@Controller
public class ClientCustomErrorController {

    public String getErrorPath() { return "/error"; }

    @RequestMapping("/error")
    public String handleError(HttpServletRequest request, Model model) {
        Object status = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        if (status != null) {
            Integer statusCode = Integer.valueOf(status.toString());

            if (statusCode == HttpStatus.NOT_FOUND.value()) {
                return "error404";
            } else if (statusCode == HttpStatus.INTERNAL_SERVER_ERROR.value()) {
                Throwable throwable = (Throwable) request.getAttribute(RequestDispatcher.ERROR_EXCEPTION);
                model.addAttribute( s: "exception", throwable.getCause().getMessage());
                return "error500";
            } else if (statusCode == HttpStatus.BAD_REQUEST.value()) {
                return "error400";
            }
```

Obr. 31. Zpracování zobrazení vlastních chybových stránek.

Zdroj: autor.

Pokud klient například dostane chybu s HTTP-statusem `404 Not Found` – v prohlížeči se zobrazí příslušná chybová stránka, viz obrázek číslo 32:



Obr. 32. Chybová stránka pro HTTP-status `404 Not Found`.

Zdroj: autor.

Uvedené metody zpracovaní výjimek a další metody projektu implementují logování. Vytváření logů z aplikace slouží k zaznamenávání informací o průběhu programu. Tyto informace pak lze použit k ladění API. V projektu pro logování je použitá knihovna Slf4j (*Simple Logging Facade for Java*) [48]. Pro logování do souboru pom.xml byly přidané závislosti slf4j-api, jul-to-slf4j, log4j-to-slf4j. Ve třídě ClientMain je vytvořen tak zvány *logger:*

`LoggerFactory.getLogger(ClientMain.class).` Logger reaguje na události za průběhu programu a klient dostává informaci například o tom, na jakém portu je spuštěna aplikace: `LOGGER.info("Application has started on port = 8080").` Jsou použité dvě úrovně logování: `info` a `error`. Zvlášť důležitá informace o chybách ve vyvinutém API, ukládá se do textového souboru `logging.txt`, informace o kterém je v konfiguračním souboru `application.properties`.

3.5. Frontendová část aplikace

Vyvinuté API umožňuje uživateli obdržet po odesílání požadavku HTML stránku, nikoli pouze text ve formátu JSON. Proto projekt obsahuje složku `resources`, ve které se nachází HTML-soubory, které uživatel dostane, když bude volat odpovídající endpointy.

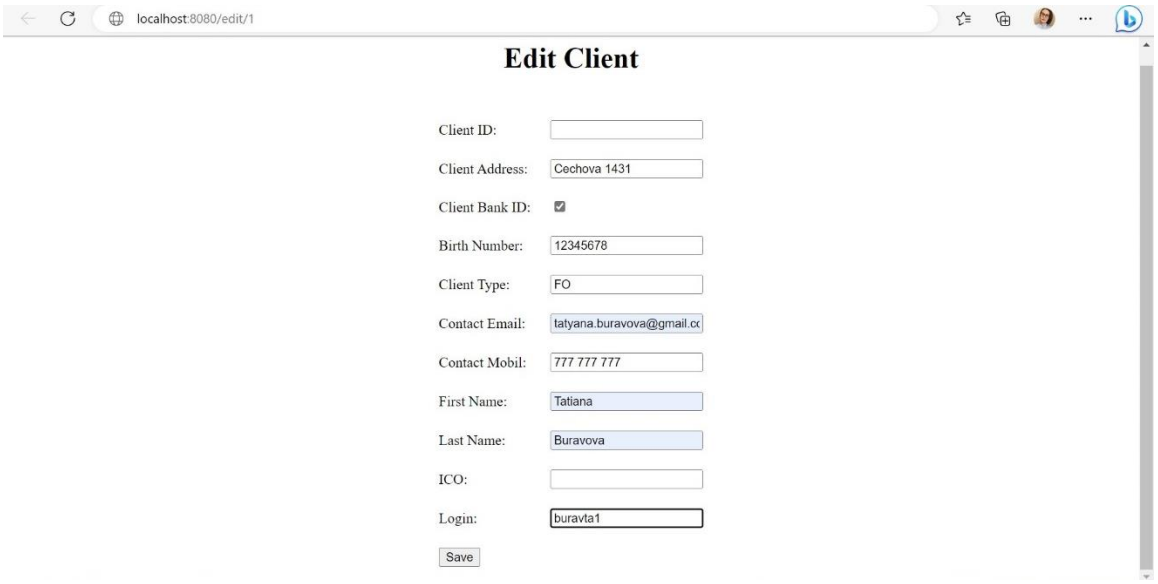
Vyvinutá webová služba obsahuje následující stránky:

1. Zobrazení seznamu klientů – soubor `get_clients.html`. Po spuštění aplikace uživatel dostane v prohlížeči stránku se seznamem klientů, viz obrázek číslo 33.

Obr. 33. Zobrazení seznamu klientů po spuštění aplikace.

Zdroj: autor.

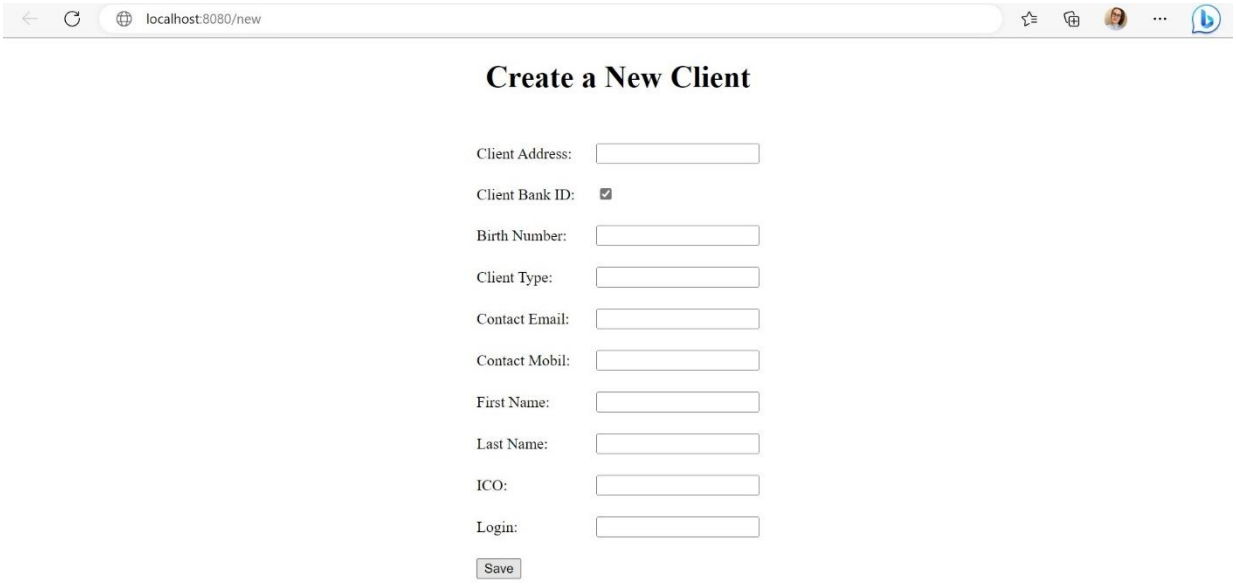
2. Stránka, která je určená k editaci údajů – soubor `edit_client.html`. Pokud uživatel chce změnit údaje klienta, může zavolat endpoint `/edit/{id}` a pro to může stisknout `Edit` na konci řádku. Po volání se uživateli zobrazí stránka editace, viz obrázek číslo 34. Po stisknuti tlačítka `Save` následuje volání endpointu `/save` a údaje budou uloženy. Po tom je implementován přechod na stránky se seznamem klientů.



Obr. 34. Zobrazení stránky pro editaci klientů.

Zdroj: autor.

3. Stránka, která je určená k přidání nového klienta – soubor `new_client.html`. K zobrazení stránky je nutné zavolat endpoint `/new`, což uživatel může uskutečnit, když stiskne odkaz `Create a New Client` na stránce se seznamem klient. Obrázek číslo 35 ukazuje, jak vypadá stránka pro přidání nového klienta.



Obr. 35. Zobrazení stránky pro přidaní nového klienta.

Zdroj: autor.

3.5.1. Šablonovací systém Thymeleaf

Pro implementaci frontendové části aplikace byl použit šablonovací systém Thymeleaf, který dovoluje vkládat do HTML-šablony proměnné například jak je na ukázce kódu níž:

```
<form action="#" th:action="@{/save}" th:object="${client}" method="post">
    <table border="0" cellpadding="10">
        <tr>
            <td>Client ID:</td>
            <td>
                <input type="text" th:field="*{clientId}" readonly="readonly" />
            </td>
        </tr>
        <tr>
            <td>Client Address:</td>
            <td><input type="text" th:field="*{address}" /></td>
        </tr>
        <tr>
            <td>Client Bank ID:</td>
            <td><input type="checkbox" th:field="*{bankIdFlag}" /></td>
        </tr>
```

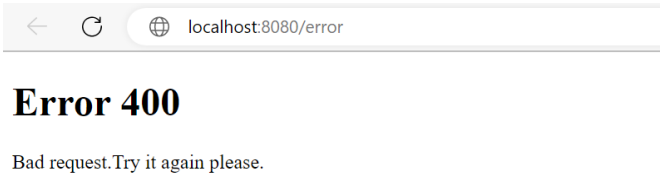
Obr. 36. Ukázka kódu s použitím Thymeleaf.

Zdroj: autor.

Stisknutí uživatelem tlačítka `Save` odešle požadavek POST `/save` s parametrem požadavku `client`. Pro zobrazení textového pole `Address` Thymeleaf potřebuje HTML-kód s atributem `address: <input type="text" th:field="*{address}" />` Pro vytvoření checkboxu pro `bankIdFlag` je použít jiný typ vstupního pole: `<input type="checkbox" th:field="*{bankIdFlag}" />`. Vlastnost `readonly` vztahující se k textovému poli `clientId` znamená, že toto pole uživatel nemůže vyplnit, `clientId` zůstane beze změny. Takže Thymeleaf je velmi užitečným nástrojem, který je snadno integrovat a používat v Spring Framework aplikaci.

3.5.2. Chybové stránky

Pokud se v aplikaci cokoli nepovede, je důležité, aby uživatel dostal informace o chybě na uživatelsky přívětivé stránce. Proto projekt obsahuje soubory, které zobrazují odpovídající chybový HTTP stav – `error*.html`. Aby chybové stránky ukazovaly užitečnou pro uživatele informaci, byly vytvořeny vlastní chybové stránky: `error400.html`, `error404.html`, `error500.html`. Pokud klient dostane chybu s HTTP statusem `404 Not Found`, `400 Bad Request` nebo `500 Internal Error` – v prohlížeči se zobrazí odpovídající stránka, která vypadá například tak, jak je to ukázáno na obrázku číslo 37. V případě jiného HTTP stavu – zobrazí se stránka `unknown_error.html`.



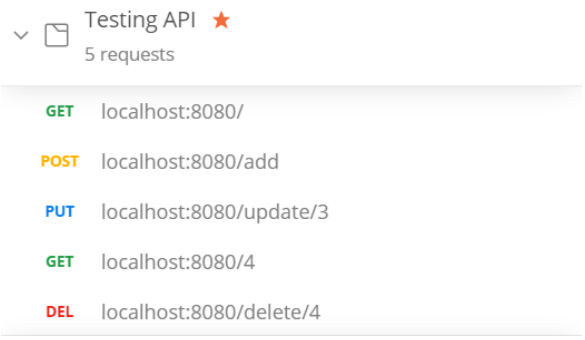
Obr. 37. Chybová stránka pro HTTP-status `400 Bad Request`.
Zdroj: autor.

3.6. Testování projektu

Při vývoji aplikací je testování nepostradatelným krokem, který umožňuje zajistit, že aplikace funguje spolehlivě. Navíc testy poskytují kontrolu toho, že při rozšiřováni aplikace změny v kódu nepoškodí původní funkčnost. V projektu existuje dva typy testů, které nepokrývají všechny možné scénáře, ale kritické časti aplikace: jednotkové testy a integrační testy.

3.6.1. Integrační testy: Postman

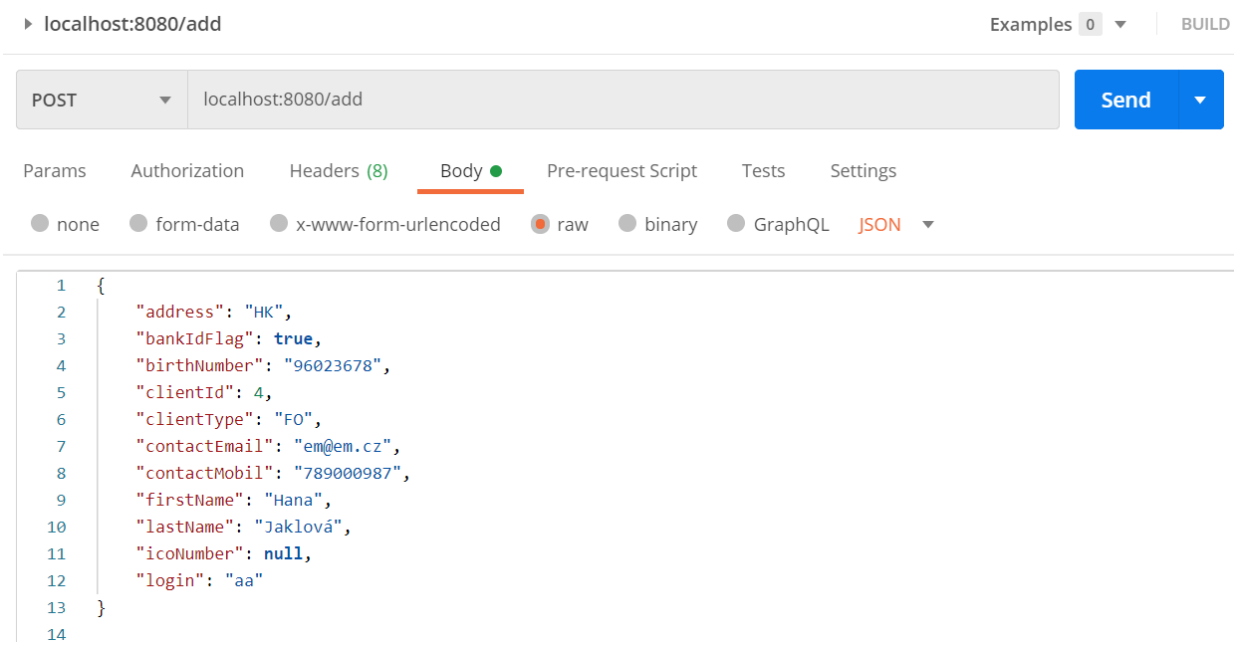
Integrační testování umožňuje testovat interakce několika služeb v aplikaci. Vyvinutá webová služba dosahuje relativně vysoké komplexnosti, je integrací databáze a REST API, které spolu komunikují a jsou vyvíjené zvlášť. Vzhledem k tomu je důležité zjistit, jestli interakce mezi nimi probíhají správně. Pro integrační testování projektu byl použit nástroj Postman. Postman dovoluje vytvořit složku, do které může být uložena sada testovacích endpointů, odpovídajících endpointům Controlleru. Na obrázku číslo 38 je složka Testing API, do které byly uloženy GET, POST, PUT a DELETE endpointy. Aplikace byla spuštěna na portu 8080.



Obr. 38. Sada endpointů pro testování v Postmanu.
Zdroj: autor.

Například vyvinutá webová služba umožňuje vytváření nového klienta, pro to uživatel musí volat endpoint `/add`. Musíme otestovat, zda nový klient opravdu bude uložen do databáze. V Postmanu k tomu byl vyroben nový požadavek:

1. byla zvolena HTTP metoda POST
2. byl zadán odpovídající URL: `localhost:8080/add`
3. byly zadány hodnoty atributů objektu `Client` ve formátu JSON, jak je ukázáno na obrázku 39.



Obr. 39. Požadavek k testování tvorby nového klienta.

Zdroj: autor.

Požadavek byl odeslán na server, server ho zpracoval, v Postmanu se objevil stav odpovědi – `200 OK`, což znamená, že klient byl uložen do databáze. Kontrola databáze potvrzuje správné chování aplikace.

3.6.2. Jednotkové testy: JUnit, Mockito

Jednotkové neboli *unit testy* ověřují správné fungování metod uvnitř třídy. V projektu pro testování pomocí jednotkových testů byl použit framework JUnit. Jiné složité objekty, které by mohly ovlivnit chování testované metody v jednotkových testech vyměněné na tzv. *mocky*. Například, mockované může být jiné API nebo databáze, komunikující s objektem uvnitř aplikace, kterou testujeme. Pokud chceme, aby chování testovaného objektu byl izolované, ostatní objekty by měly být nahrazené simulátory jejich chování. Pro tvorbu mocků byla použita knihovna Mockito.

Při psaní unit testů webové služby byly udělané následující kroky:

1. příprava dat pro navození stavu, který je v zadání
2. volání služby/endpointu,
3. ověření, že se vykonaly akce, které mělo zadání úkolu zajistit [49]

Na obrázku 40 je vidět, že byly splněny tyto kroky, mockované jsou `ClientRepository` (pomocí anotace `@SpyBean`) a `ClientServiceImpl` (pomocí metody `mock()`). Tento test zajistí, že v `ClientServiceImpl` správně funguje metoda,

která najde klienta v databázi podle jeho `clientId` po volání příslušného endpointu v `ClientControlleru`.

```
@SpyBean
private ClientRepository repo;

@BeforeEach
void setUp() { clientService = mock(ClientServiceImpl.class); }

@Test
void getClientTest() {
    client = Client
            .builder()
            .address("address")
            .clientId(1L)
            .clientType(ClientType.FO)
            .bankIdFlag(true)
            .birthNumber("12345678")
            .contactEmail("email@email.cz")
            .contactMobil("+420000000000")
            .firstName("Eva")
            .lastName("Novakova")
            .icoNumber(null)
            .login("login")
            .build();
    Optional<Client> foundClient = repo.findById(client.getClientId());
    assertNotNull(foundClient);
}
```

Obr. 40. Příklad jednotkového testu třídy `ClientController`.

Zdroj: autor.

4. ZÁVĚR

Cílem této práce bylo vytvořit a otestovat REST API, které by mělo být zaměřeno na zpracování testovacích uměle vytvořených údajů klientů banky. Toto API by navíc umožňovalo uložení informací o klientech a případně rozšíření pro různé účely. Vybudovaná aplikace by měla být pokryta jednotkovými a integračními testy. Také vyvinutá webová služba by měla obsahovat pro pohodlí uživatelů webové stránky s možností vyvolat a otestovat konkrétní endpointy.

Všech cílů bylo dosaženo především díky získání informace o REST API, jeho komponentech, různých variantách použití a omezeních. Vyvinuté API je implementováno v programovacím jazyce Java pomocí Spring Frameworku s použitím databáze PostgreSQL. Přestože vytvořené API je plně funkční, jedná se pouze o vzorovou verzi. Chování této webové služby je omezené kvůli tomu, že se nepoužívají údaje skutečných klientů. API lze rozšířit přidáním dalších funkcionalit, například správou zaměstnanců banky nebo uchováváním informací o odděleních banky. Ačkoli je zvolený formát pro tuto práci JSON, lze jej kromě

toho rozšířit na další formáty, například XML, jednoduše přidáním nebo úpravou vrstvy kontroléru v implementaci. Vyvinuté API lze také rozšířit tak, aby bylo možné třídit údaje klientů podle jednoho nebo více sloupců nebo filtrovat podle nějaké podmínky.

5. LITERATURA

[1] DEWAILLY, Ludovic. Building a RESTful Web Service with Spring. Kindle Edition, 2015. 128 s. ISBN 978-1785285714.

[2] NILO, Mitra, Y. L. SOAP Version 1.2 [online]. 2007. [cit. 02-03-2023]. Dostupné z: https://www.w3.org/TR/soap12-part0/.

[3] Facebook. GraphQL [online]. 2023. [cit. 02-03-2023]. Current Working Draft – GraphQL. Dostupné z: https://spec.graphql.org/draft/.

[4] Introduction to GraphQL [online]. Facebook, 2023. [cit. 02-03/2023]. Dostupné z: https://graphql.org/learn/.

[5] FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Irvine, 2000. Disertační práce. University of Califonia.

[6] FIELDING, Roy Thomas, Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230. RFC Editor, červ. 2014, http://www.rfc-editor.org/info/rfc7230.

[7] HTTP request methods [online]. Mozilla Corporation, 2023. [cit. 04-03-2023]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[8] TODD, Fredrich. HTTP Status Codes, 2023 [cit. 04-03-2023]. Dostupné z: http://www.restapitutorial.com/httpstatuscodes.html.

[9] CHRISTIE, Tom. Django REST framework [online]. 2023 [cit. 04-03-2023]. Dostupné z: http://www.django-rest-framework.org/.

[10] CHRISTIE, Tom. Django REST framework: REST, Hypermedia & HATEOAS [online]. 2023 [cit. 04-03-2023]. Dostupné z: http://www.djangorest-framework.org/topics/rest-hypermedia-hateoas/.

[11] CHRISTIE, Tom et al. Flask API [online]. 2023 [cit. 04-03-2023]. Dostupné z: https://github.com/tomchristie/flask-api.

[12] CHRISTIE, Tom. Flask API: Browsable Web APIs for Flask [online]. 2023 [cit. 04-03-2023]. Dostupné z: http://www.flaskapi.org/.

[13] BURKE, Kevin; CONROY, Kyle; HORN, Ryan et al. Flask-RESTful: Flask-RESTful documentation [online]. 2023 [cit. 04-03-2023]. Dostupné z: https://flask-restful.readthedocs.org/en/0.3.5/.

[14] LANGUAGE Trends. [online]. GitHub.com. 2023 [cit. 07-03-2023]. Dostupné z: https://github.blog/.

[15] SPRING Framework. [online]. Spring Documentation. 2023 [cit. 07-03-2023]. Dostupné z: https://docs.spring.io.

[16] HARROP, Rob, Cosmina I. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools 5th Edition, Kindle Edition, 2021, 1717 s. ISBN-13: 978-1484228074.

[17] KHAN, M. Aspect Oriented Programming and AOP in Spring Framework. 2023 [cit. 07-03-2023]. Dostupné z: https://medium.com/@mustafakhansmt/imperative-vsdeclarative-programming-106f99a7ffc5.

[18] CARNELL, John. Spring microservices in action. eBook, 2021, 448 s. ISBN 9781617296956.

[19] ŠKÁLOVATELNOST aplikací. [online]. 2023. [cit. 08-03-2023]. Dostupné z: https://www.webtodate.cz/cz/o-webtodate/technicke-informace/skalovatelnost/489/.

[20] MYSQL, Documentation. [online]. 2023. [cit. 08-03-2023]. Dostupné z: https://dev.mysql.com/doc/.

[21] POSTGRESQL. About [online]. 2023. [cit. 08-03-2023]. Dostupné z: https://www.postgresql.org/about/.

[22] NATIONAL Institute of Standards and Technology. Cloud. [online]. 2023. [cit. 08-03-2023]. Dostupné z: https://www.nist.gov/.

[23] VELTE, Antony; Toby J. Velte; Robert Elsenpeter. Cloud computing - praktický průvodce. Computer Press. 2018. 304 s. ISBN 978-80-2513-333-0.

[24] OPENAPI Specification v3.1.0. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://spec.openapis.org/oas/v3.1.0.

[25] JSON. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://www.json.org/json-en.html.

[26] YAML. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://yaml.org/.

[27] API You Won't Hate. OpenAPI.Tools. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://openapi.tools/.

[28] ANDRUSHKO, Sviatoslav. The Best JS Frameworks for Front End. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://rubygarage.org/blog/bestjavascript-frameworks-for-front-end.

[29] WHAT And Why React.js. In: c-sharpcorner.com. [online]. 2023. [cit. 10-03-2023]. Dostupné z: https://www.c-sharpcorner.com/article/what-and-why-reactjs.

[30] ANGULAR vs React: Which One to Choose for Your App. In: freecodecamp.org. [online]. 2023. [cit. 11-03-2023]. Dostupné z: https://www.freecodecamp.org/news/angular-vs-react-what-to-choose-for-your-app-2.

[31] TEMPLATE Syntax. In: vuejs.org. [online]. 2023. [cit. 11-03-2023]. Dostupné z: https://vuejs.org/guide/essentials/template-syntax.html#raw-html.

[32] APACHE License, version 2.0. [online]. 2023. [cit. 11-03-2023]. Dostupné z: https://www.apache.org/foundation/glossary.html.

[33] JAVA Server Pages Technology. [online]. 2023. [cit. 11-03-2023]. Dostupné z: https://www.oracle.com/java/technologies/jspt.html.

[34] THYMELEAF Documentation. [online]. 2023. [cit. 11-03-2023]. Dostupné z: https://www.thymeleaf.org/documentation.html.

[35] WRITING tests. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://learning.postman.com/docs/writing-scripts/test-scripts/.

[36] JUNIT 5 User Guide. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/.

[37] POSTMAN Documentation. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://learning.postman.com/docs/introduction/overview/.

[38] ANNOTATION JUnit 5. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations

[39] USAGE @Test Annotation. JUnit 5. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://www.baeldung.com/junit-5-test-annotation.

[40] MOCKITO Tutorial. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://www.baeldung.com/mockito-series.

[41] MAVEN Apache Project. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://maven.apache.org/.

[42] SPRING Initializr. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://start.spring.io/.

[43] MAVEN Repository. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://mvnrepository.com/.

[44] MVC Architecture. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://www.w3schools.in/mvc-architecture?utm_content=cmp-true.

[45] WHAT Is a Spring Bean. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://www.baeldung.com/spring-bean.

[46] SPRING. Reference Documentation [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/.

[47] LOMBOK Project. [online]. 2023. [cit. 12-03-2023]. Dostupné z: https://projectlombok.org/.

[48] SIMPLE Logging Facade for Java (SLF4J). [online]. 2023. [cit. 14-03-2023]. Dostupné z: https://www.slf4j.org/.

[49] MARTIN, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship 1st Edition, Kindle Edition, 2008, 1214 s., ISBN-13: 978-0132350884

6. PŘÍLOHY

6.1. Příloha č. 1 - Seznam obrázků

6.2. Příloha č. 2 – Seznam tabulek

6.3. Příloha č. 3 – Zdrojový kód aplikace