Univerzita Pardubice

Fakulta elektrotechniky a informatiky

RESTful API pro správu provozu letiště

Bakalářská páce

2020

Andrii Lupenko

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Andrii Lupenko**
Osobní číslo:	**I17119**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Restful API pro správu provozu letiště**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem bakalářské práce je vybudovat a otestovat RESTful API se zaměřením na provoz letiště. Konkrétně bude API umožňovat správu příletů, odletů, rezervaci letenek, databázi letů, případně některé další technologické aspekty. Vytvořené API bude pokryto jednotkovými a integračními testy. API bude také vhodně doplněno vizualizační stránkou s přehledem všech endpointů a použitých datových modelů. Webová stránka bude umožňovat vyvolat a otestovat konkrétní endpointy. Text bakalářské práce bude obsahovat aktuální informace o REST API, případně jeho alternativách, informace o testování, technologiích a frameworcích pro vytváření REST API.

Rozsah pracovní zprávy: **30 normostran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. BIEHL M. RESTful API Design: Best Practices in API Design with REST. API-University Press; 1 edition (August 28, 2016). ASIN B01L6STMVW.
2. FARCIC V., GARCIA A. Test-Driven Java Development. Packt Publishing; 1 edition (August 27, 2015). ISBN-13: 978-1783987429.

Vedoucí bakalářské práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **15. listopadu 2019**
Termín odevzdání bakalářské práce: **7. května 2020**

L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

Ing. Lukáš Čegan, Ph.D.
pověřený vedením katedry

V Pardubicích dne 17. prosince 2019

Prohlašuji:

Tuto práci 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., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, 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, s tím, že pokud dojde k užití této práce mnou nebo k poskytnutí licence o užití jinému subjektu, je Univerzita Pardubice oprávněna požadovat ode mne 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. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 2. 5. 2020 ....

Andrii Lupenko

Rád bych zde poděkoval vedoucímu bakalářské práce doc. Ing. Michaelu Bažantovi, Ph.D. za jeho podnětné rady a trpělivost při vytváření práce. Také děkuji všem respondentům, kteří mi poskytli potřebné informace pro dopracování mé práce.

ANOTACE

Práce se zabývá návrhem a implementací prototypu API v architektonickém stylu REST. API se zaměřuje na provoz letiště. Výsledná aplikace umožňuje správu příletů, odletů, rezervaci letenek, databázi letadel a letišť. Teoretická část obsahuje aktuální informaci o REST API, alternativách architektonického stylu REST a implementačního prostředí. Realizace prototypu ukazuje implementaci různých vrstev aplikací a demonstruje jejich funkcionalitu. Po realizaci jsou uvedeny integrační a jednotkové testy výsledné funkcionality. V závěru jsou uvedena doporučení pro vylepšení prototypu.

KLÍČOVÁ SLOVA

REST, API, Representational State Transfer, Java, Spring MVC, Letiště

TITLE

RESTful API for airport management

ANNOTATION

The goal of this thesis was to design and implement a prototype of an API in the REST architectural style. The API focuses on airport managing. The result application can manage arrivals, departures, ticket reservations, aircraft and airport database. The theoretical part contains current information about the REST API, alternatives to the REST architectural style and the implementation environment. The implementation of the prototype shows the different layers of application and demonstrates their functionality. After the implementation part, integration and unit tests of the resulting functionality are presented. Finally, recommendations for improving the prototype are given.

KEYWORDS

REST, API, Representational State Transfer, Java, Spring MVC, Airport

OBSAH

SEZNAM ILUSTRACÍ A TABULEK 8
SEZNAM ZKRATEK A ZNAČEK 10
ÚVOD 11
1 APPLICATION PROGRAMMING INTERFACE 12
1.1 API webových služeb 12
2 REPRESENTATIONAL STATE TRANSFER 14
2.1 REST Přístup 15
2.2 Architektonická omezení 15
2.2.1 Jednotné rozhraní 16
2.2.2 Klient-server 17
2.2.3 Bezstavovost 19
2.2.4 Vyrovnávací paměť 20
2.2.5 Vrstvený systém 22
2.2.6 Kód na vyžádání 25
3 VÝVOJ RESTFUL API NA PLATFORMĚ JAVA 26
3.1 Populární frameworky 26
3.1.1 JAX-RS 26
3.1.2 Spring MVC 27
4 REALIZACE CÍLOVÉ APLIKACE 29
4.1 Vrstva modelu 30
4.2 Databázová vrstva 33
4.3 Vrstva kontroléru 34
4.4 Servisní vrstva 36
4.5 Swagger UI 37
4.6 Konfigurace 39
4.7 Testování 39
4.7.1 Integrační testy 40
4.7.2 Jednotkové testy 41
5 ZÁVĚR 44
6 POUŽITÁ LITERATURA 45
PŘÍLOHY 47

SEZNAM ILUSTRACÍ A TABULEK

Seznam obrázků

Obrázek 1 – Web API (3) 12
Obrázek 2 – Ukázkový formát zprávy SOAP obsahující informace o adrese 13
Obrázek 3 – Diagram komunikace v bezstavové architektuře (7) 19
Obrázek 4 – Diagram komunikace s využitím vyrovnávací paměti (7) 21
Obrázek 5 – Znázornění abstraktního vrstveného systému (7) 23
Obrázek 6 – Příklad komunikace ve vrstveném systému (7) 23
Obrázek 7 – Ukázka implementací webové služby pomocí JAX-RS 27
Obrázek 8 – Ukázka implementací webové služby pomocí Spring MVC 28
Obrázek 9 – Struktura balíčku 29
Obrázek 10 – Entitně vztahový diagram 30
Obrázek 11 – Zjednodušená ukázka třídy `Flight` 31
Obrázek 12 – Zjednodušená ukázka třídy `Booking` 32
Obrázek 13 – Zjednodušená ukázka třídy `Passenger` 32
Obrázek 14 – Zjednodušená ukázka třídy `Airport` 33
Obrázek 15 – Zjednodušená ukázka třídy `Aircraft` 33
Obrázek 16 – Obsah konfiguračního souboru `application.properties` 34
Obrázek 17 – Ukázka třídy `AircraftController` 35
Obrázek 18 – Ukázka třídy `BookingDto` 36
Obrázek 19 – Ukázka třídy-mapovače `AirportMapper` 36
Obrázek 20 – Ukázka třídy-služby B`ookingServiceImpl` 37
Obrázek 21 – Ukázka konfigurace `Swagger UI` 38
Obrázek 22 – Ukázka seznamu kontroleru na vizualizační stránce 38
Obrázek 23 – Ukázka seznamu endpointu ve třídě AircraftController na vizualizační stránce 39
Obrázek 24 – Ukázka enpointu `/aircraft/{id}` 39
Obrázek 25 – Výstup v logu oznamující aktivní profil 39
Obrázek 26 – Ukázka konfiguračního souboru `application-dev.properties` 40
Obrázek 27 – Ukázka testovací třídy pro `AircraftController` 41
Obrázek 28 – Příklad jednotkových testů třídy `AirportMapper` 42
Obrázek 29 – Příklad jednotkových testůs použitím „mockování“ 42

Seznam tabulek

Tabulka 1 – Různá záhlaví ovládacích prvků mezipaměti 22

Tabulka 2 – Popis anotací používané v entitách projektu 31

SEZNAM ZKRATEK A ZNAČEK

API	Application Programming Interface
COD	Code On Demand
DAO	Data Access Object
DTO	Data Transfer Object
ER	Entity–Relationship
GUI	Graphical User Interface
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICAO	International Civil Aviation Organization
JAX-RS	Java API for RESTful Web Services
JDK	Java Development Kit
JPA	Java Persistence API
JPA 2	Java Persistence API 2
JSON	JavaScript Object Notation
J2EE	Java 2 Platform, Enterprise Edition
MVC	Model–View–Controller
ORM	Object-Relational Mapping
REST	Representational state transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

ÚVOD

Webové technologie se rychle vyvíjejí, takže aby se programátoři dokázali vypořádat s vývojem složitých webových aplikací, a pro zjednodušení vývoje dochází k oddělení klientské a serverové části aplikace. V poslední době jsou serverovými aplikacemi webové služby. Klientská část komunikuje s webovou službou prostřednictvím svého API.

Díky své lehké povaze, adaptabilitě a škálovatelnosti se REST stal preferovaným stylem pro tvorbu webových API. S příchodem REST se webové API stalo základem webových aplikací, cloudových technologií, mobilních aplikací a dokonce mnoha iniciativ v oblasti otevřených dat zaměřených na usnadnění přístupu k informacím. Ve skutečnosti je nyní webové API pro každou firmu kriticky důležité a jeho implementace roste. Pokud v telefonu používáte mobilní aplikaci, může váš telefon tajně hovořit s mnoha cloudovými službami o načtení, aktualizaci nebo odstranění dat pomocí REST. REST služby mají obrovský dopad na náš každodenní život.

Cílem bakalářské práce je vybudovat a otestovat REST API se zaměřením na provoz letiště. Konkrétně API umožňuje správu příletů, odletů, rezervaci letenek, databázi letů a některé další technologické aspekty. Vytvořená aplikace je pokryta jednotkovými a integračními testy. API je také vhodně doplněno vizualizační stránkou s přehledem všech endpointů a použitých datových modelů. Webová stránka umožňuje vyvolat a otestovat konkrétní endpointy. Při implementaci byl použit programovací jazyk Java. Účelem práce je také uvést čtenáře do problematiky implementací REST API, seznámit s aktuální informací o REST API, jeho alternativách, technologiích a frameworcích pro vytváření REST API na platformě Java.

1 APPLICATION PROGRAMMING INTERFACE

Lidé obvykle používají software prostřednictvím uživatelského rozhraní. Přesto s narůstajícím vývojem programového vybavení používají software nejen lidé, ale i jiné softwarové aplikace. To vyžaduje jiný typ rozhraní, Application Programming Interface API. (1)

API nabízí jednoduchou cestu pro propojení, integraci a rozšíření softwarového systému. Zde studovaná API jsou webová API, která jsou realizována jako webové služby a dodávají data prostřednictvím webových technologií. Typickými aplikacemi používajícími API jsou mobilní, cloudové a webové aplikace nebo inteligentní zařízení. Kouzlem API je, že je jednoduché, čisté, jasné a přístupné. Opakovaně poskytuje použitelné rozhraní, ke kterému se různé aplikace mohou snadno připojit. (1)

1.1 API webových služeb

S významem internetu a všudypřítomností HTTP v dnešním světě se webové služby staly hlavním prostředkem pro vzájemné propojení webových systémů. (2)

Webová služba je softwarová součást, ke které lze přistupovat prostřednictvím adresy URL. Webové služby jsou webové aplikace, moduly nebo komponenty, které lze považovat za službu poskytovanou prostřednictvím webu. Tradičně jsme měli jako webový obsah statické stránky HTML, které se vyvinuly do dynamičtějších, plně funkčních webových aplikací poskytujících různou funkčnost koncovému uživateli. Komponenta webové služby je o krok napřed před tímto webovým paradigmatem a poskytuje pouze obchodní službu, obvykle ve formě prvotních dat JSON nebo XML. GUI a obchodní funkce jsou dobře oddělené. Webovou službu lze považovat za samostatnou samopisující modulární aplikaci, která může být publikována, lokalizována a vyvolána na webu. (3)

Obrázek 1 – Web API (3)

Největší výhodou, kterou webové služby poskytují, je interoperabilita. Webové služby mohou být přenášeny na jakékoli platformě a mohou být psány v různých programovacích jazycích. Podobně i klientova aplikace přistupující k webové službě může být napsána v jiném jazyce a spuštěna na jiné platformě než samotná služba. (3)

Protokol Simple Object Access Protocol (SOAP) byl de facto volbou pro budování takových služeb. (2) SOAP je komunikační protokol založený na XML, který využívá otevřené standardy. Formát zprávy SOAP se skládá z obálky SOAP, která uzavírá všechny informace o požadavcích. SOAP Envelope je pak vyroben z volitelných záhlaví a těla. Záhlaví volitelně obsahují kontextové informace, jako je zabezpečení nebo transakce, zatímco tělo obsahuje skutečné užitečné údaje nebo aplikační data. (3)

```
<soapenv:Envelope
    xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
    xmlns:ns1=http://apress.com/beginjava6/address
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soapenv:Header></soapenv:Header>
    <soapenv:Body>
        <ns1:Address>
            <ns1:addressLine1>1501ACity</ns1:addressLine1>
            <ns1:addressLine2>UCity</ns1:addressLine2>
            <ns1:city>SFO</ns1:city>
            <ns1:state>CA</ns1:state>
            <ns1:country>US</ns1:country>
        </ns1:Address>
    </soapenv:Body>
</soapenv:Envelope>
```

Obrázek 2 – Ukázkový formát zprávy SOAP obsahující informace o adrese

V posledních letech se však REST stal velmi populární alternativou k tradičním webovým službám SOAP. (2)

2 REPRESENTATIONAL STATE TRANSFER

Název Representational State Transfer (REST) vytvořil Roy Fielding z University of California (16). Jedná se o velmi zjednodušenou a lehkou webovou službu ve srovnání se SOAP. Výkon, škálovatelnost, jednoduchost, přenositelnost a modifikovatelnost jsou hlavními principy návrhu REST. (5)

REST není ani technologie, ani standard; je to architektonický styl, soubor pokynů pro odhalení zdrojů na webu. Architektonický styl REST souvisí se zdrojem, což je reprezentace identifikovaná pomocí URI (Uniform Resource Indicator). Zdrojem může být jakákoli informace, jako je kniha, objednávka, zákazník, zaměstnanec atd. Klient se dotazuje nebo aktualizuje zdroj prostřednictvím URI výměnou reprezentací zdroje. Reprezentace obsahují skutečné informace ve formátu jako je HTML, XML nebo JavaScript Object Notation (JSON), který je zdrojem akceptován. Klient si musí být vědom reprezentace vrácené klientem. Ten obvykle specifikuje, jaké reprezentace chce a server vrátí požadovaný zdroj, například požadovanou stránku s obsahem HTML. Všechny zdroje sdílejí jednotné rozhraní pro přenos stavu mezi klientem a prostředkem. Všechny informace potřebné pro zpracování požadavku na zdroj jsou obsaženy v samotném požadavku. (3)

World Wide Web je klasický příklad architektonického stylu REST. Jak je implementováno na World Wide Web, URI identifikují zdroje a HTTP je protokol, pomocí kterého jsou přístupné zdroje. HTTP poskytuje jednotné rozhraní a sadu metod pro manipulaci se zdrojem. Klientský program, například webový prohlížeč, může přistupovat, aktualizovat, přidávat nebo odebírat webový zdroj pomocí URI a různých metod HTTP. HTTP poskytuje standardní metody, jako je `GET, POST, PUT, DELETE, HEAD, TRACE` a `Options`. Každá z těchto metod představuje akci, kterou lze nad zdrojem provést. Například `HTTP GET` se používá pouze pro získávání informací a neměl by nikdy měnit stav zdroje, zatímco metody jako `PUT, POST, DELETE` ovlivňují změnu stavu v jeho reprezentaci. (3)

Webové služby postavené na principech architektury REST se nazývají RESTful webové služby. Webové služby vyvinuté pomocí přístupu REST lze představit jako zdroje identifikované podle jejich URI. Webová služba odhaluje seznam operací pomocí standardních metod HTTP, jako je `GET` nebo `POST`. Klienti webových služeb volají jednu z metod definovaných ve zdrojích pomocí URI přes protokol HTTP. (3)

Příkladem RESTful API může být služba, která poskytuje podrobnosti o zaměstnancích v rámci oddělení:

URI RESTful služby – http://<host>/department/deptname/employee:

- GET – vrací seznam zaměstnanců v oddělení,
- POST – vytvoří záznam zaměstnance v oddělení,
- DELETE –smaže záznam zaměstnance v oddělení.

URI RESTful služby – http://<host>/department/deptname/employee/jan:

- GET – vrátí informaci o zaměstnanci Jan,
- PUT – obnoví informaci o zaměstnanci Jan,
- DELETE – smaže informaci o zaměstnanci Jan. (3)

REST API umožňuje různým systémům komunikovat a odesílat/přijímat data velmi jednoduchým způsobem. Každé volání REST API má vztah mezi metodou HTTP a URL. (5)

2.1 REST Přístup

Na konci roku 1993 se Roy Fielding, spoluzakladatel projektu Apache HTTP Server Project, začal zabývat problémem škálovatelnosti webu. Po analýze Fielding zjistil, že škálovatelnost webu byla řízena sadou klíčových omezení. On a další se rozhodli zlepšit implementaci webu pragmatickým přístupem: jednotně splnit všechna omezení, aby se web mohl dále rozšiřovat. (4)

2.2 Architektonická omezení

Omezení, která Fielding seskupil do šesti kategorií a společně označovaných jako architektonický styl webu, jsou:

1. Jednotné rozhraní,
2. Klient-server,
3. Bezstavovost,
4. Vyrovnávací paměť,

5. Vrstvený systém,
6. Kód na vyžádání.

Každá kategorie omezení je shrnuta v následujících podkapitolách. (4)

2.2.1 Jednotné rozhraní

Jádrem REST jsou zdroje. Zdroje jsou identifikovány pomocí URI. Z koncepčního hlediska jsou zdroje oddělené od jejich reprezentace (formátu, ve kterém jsou poskytovány klientům). REST nenabízí žádný specifický formát, ale obvykle zahrnuje XML a JSON. (2)

Reprezentace zdrojů jsou navíc sebepopisné. Konkrétně to znamená, že pro úspěšné zpracování odpovědí je třeba vrátit dostatečné informace. (2)

Interakce mezi komponentami webu — jeho klienty, servery a síťová intermédia – závisí na jednotnosti jejich rozhraní. Pokud některá z komponent vybočí ze zavedených standardů, komunikační systém webu se zhroutí. (2)

Webové komponenty konzistentně interagují v rámci čtyř omezení jednotného rozhraní, které Fielding definuje jako:

1. Identifikace zdrojů,
2. Manipulace se zdroji pomocí jejich reprezentací,
3. Sebe popisné zprávy,
4. Hypertext As The Engine Of Application State (HATEOAS).

Čtyři omezení rozhraní jsou shrnuté v následujících podkapitolách.

Identifikace zdrojů

Každý odlišný webový koncept je známý jako zdroj a může být adresován jedinečným identifikátorem, jako je URI. Například konkrétní URI stránky, jako `http://www.example.com`, jedinečně identifikují koncept kořenového zdroje konkrétního webu. (4)

Manipulace se zdroji prostřednictvím reprezentací

Klienti manipulují se znázorněním zdrojů. Stejný zdroj lze prezentovat různým klientům různými způsoby. Například dokument může být reprezentován jako HTML pro webový

prohlížeč a jako JSON pro automatizovaný program. Klíčovou myšlenkou je, že pohled je způsob interakce se zdrojem, ale není to samotný zdroj. Tento koncepční rozdíl umožňuje reprezentovat zdroj různými způsoby a v různých formátech, aniž by se změnil jeho identifikátor. (4)

Sebe popisné zprávy

Požadovaný stav zdroje může být reprezentován ve zprávě s požadavkem klienta. Aktuální stav prostředku může být reprezentován ve zprávě s odpovědí, která je vrácena ze serveru. Jako příklad může klient editoru wiki stránek použít zprávu s požadavkem k přenosu pohledu, který nabízí aktualizaci stránky (nový stav) pro webovou stránku (zdroj) spravovanou serverem. Je na serveru, aby přijal nebo odmítl požadavek klienta. (4)

Samopopisující zprávy mohou zahrnovat metadata, která zprostředkují další podrobnosti týkající se stavu zdroje, formátu a velikosti prezentace a samotné zprávy. Zpráva HTTP poskytuje záhlaví pro uspořádání různých typů metadat do jednotlivých polí. (4)

HATEOAS

Lze přeložit jako „hypertext jako prostředek pro aplikační stav“. Zobrazení stavu zdroje obsahuje odkazy na související zdroje. Odkazy jsou vlákna, která propojují síť a umožňují uživatelům smysluplně a účelně prohlížet informace a aplikace. Přítomnost nebo nepřítomnost odkazu na stránce je důležitou součástí aktuálního stavu zdroje. (4)

2.2.2 Klient-server

Model klient-server, na který se vztahuje REST, umožňuje oddělit zájmy klientů, jako je interakce uživatelů nebo zkušenosti uživatelů, od problémů se serverem, jako je ukládání dat a škálovatelnost. (4)

Toto oddělení zajišťuje, že s konzistentním rozhraním lze vývoj klientů a serverů provádět nezávisle. Mohou být implementovány a rozmístěny nezávisle pomocí jakéhokoli jazyka nebo technologie, pokud odpovídají jednotnému rozhraní webu (4). Pomáhá také snížit složitost a zlepšit vyladění výkonu. (2)

Architektura nebo model klient-server pomáhá při oddělení problémů mezi uživatelským rozhraním a ukládáním dat:

Klient je komponenta, která je žadatelem služby a odešle serveru požadavky na různé typy služeb.

- Server je komponenta, která je poskytovatelem služeb a nepřetržitě poskytuje služby klientovi podle požadavků.
- Klienti a servery obvykle obsahují distribuované systémy, které komunikují po síti. (7)

Klient v architektuře klient-server

Neexistuje žádná horní hranice počtu klientů, kteří mohou být obsluhováni jediným serverem. Není také povinné, aby klient a server sídlili v samostatných systémech. Klient i server mohou sídlit ve stejném systému na základě hardwarové konfigurace systému a typu funkčnosti nebo služby poskytované serverem. Komunikace mezi klientem a serverem probíhá výměnou zpráv pomocí vzoru požadavek-odpověď. Klient v zásadě odešle požadavek na službu a server vrátí odpověď. Tento vzorec komunikace-požadavek je vynikajícím příkladem mezi-procesové komunikace. Aby tato komunikace probíhala efektivně, je nutné mít dobře definovaný komunikační protokol, který stanoví pravidla komunikace, jako je formát žádostí, odpovědních zpráv, zpracování chyb atd. Všechny komunikační protokoly používané pro komunikaci klient-server fungují v aplikační vrstvě zásobníku protokolů. Aby se dále zefektivnil proces komunikace mezi klientem a serverem, server někdy implementuje specifické API, které může klient použít pro přístup k jakékoli konkrétní službě ze serveru. (7)

Služba v architektuře klient-server

Termín služba používaná v kontextu architektury klient-server se týká abstrakce zdroje. Zdroj může být jakéhokoli typu a založen na zdroji, který poskytuje server (služba); server je odpovídajícím způsobem pojmenován. Pokud například server poskytuje webové stránky, nazývá se webový server a pokud poskytuje soubory, nazývá se souborový server atd. Server může přijímat požadavky od libovolného počtu klientů v určitém časovém okamžiku. Každý server však bude mít svá vlastní omezení, pokud jde o jeho možnosti zpracování. Často je nutné, aby server upřednostňoval příchozí požadavky a obsluhoval je podle jejich priority. Plánovací systém přítomný na serveru pomáhá serveru stanovit priority. (7)

Výhody klient-server modelu jsou kromě oddělení zájmů a pomoci také:

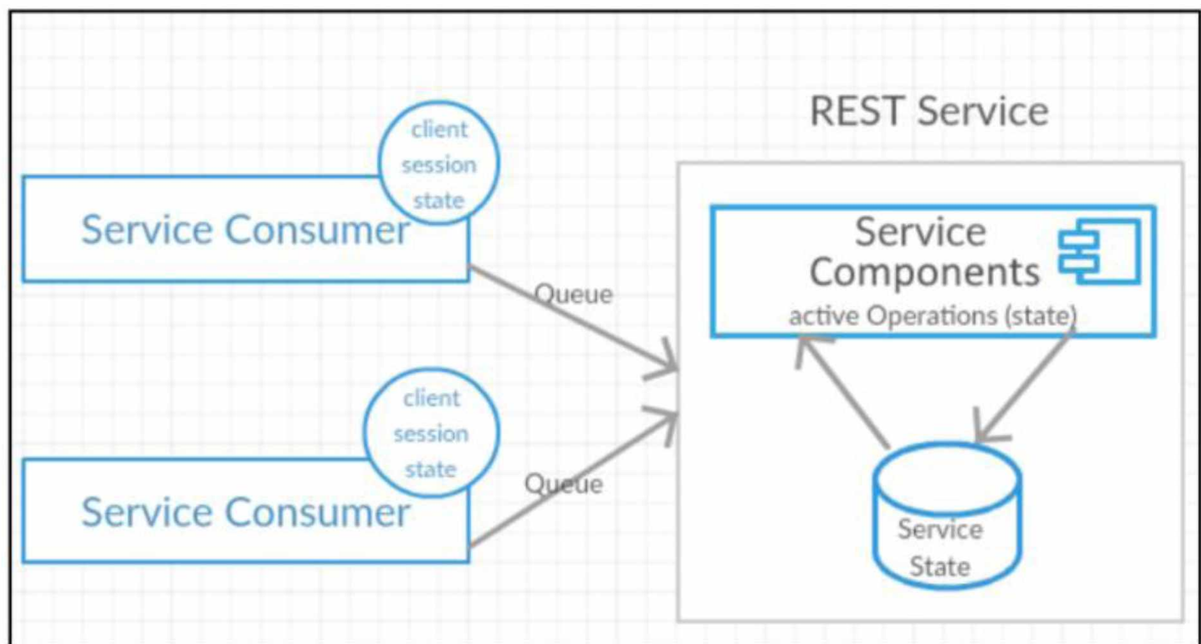
- Zlepšení přenositelnosti uživatelského rozhraní,

- Zlepšení škálovatelnosti zjednodušením implementací serveru,
- Vývoj pomocí samostatných nezávislých testovatelných komponent. (7)

2.2.3 Bezstavovost

Omezení bezstavovosti pomáhá službám být škálovatelnější a spolehlivější. Bezstavovost v kontextu REST znamená, že všechny klientské požadavky na server přenášejí všechny informace jako explicitní, takže server zpracovává požadavky jako nezávislé a tyto klientské požadavky udržují server nezávislý na uložených kontextech. Udržování stavu sezení (angl. *„session“*) v klientovi je důležité pro správu tohoto omezení ve službách. (7)

Následující diagram ukazuje, že stav klienta je nezávislý a je spravován na straně klienta.



Obrázek 3 – Diagram komunikace v bezstavové architektuře (7)

Omezení pro dosažení bezstavovosti:

- Za uložení a zpracování všech stavů aplikace a souvisejících informací na straně klienta je plně zodpovědný klient.
- Klient je zodpovědný za odesílání jakýchkoli informací o stavu na server, kdykoli je to potřeba.
- Server musí také obsahovat veškeré nezbytné informace, které může klient potřebovat k vytvoření stavu na své straně.

- Interakce HTTP zahrnují dva druhy stavů, stav aplikace a stav zdroje. Bezstavovost se týká obou.
    - **Stav aplikace**: Data uložená na straně serveru pomáhají identifikovat příchozí klientský požadavek pomocí předchozích podrobností o interakci s aktuálními informacemi o kontextu.
    - **Stav zdroje**: Je označován jako reprezentace zdroje, která je nezávislá na klientovi. Jeto aktuální stav serveru v kterémkoli daném okamžiku. (7)

Výhody a nevýhody bezstavovosti

Níže jsou uvedeny některé výhody bezstavovosti:

- Vzhledem k tomu, že server nepotřebuje spravovat žádné sezení, je možné nasazení služeb na libovolný počet serverů, takže škálovatelnost nebude nikdy problémem.
- Protože požadavky lze ukládat do mezipaměti, bezstavovost zkracuje dobu odezvy serveru, tj. zlepšuje výkon s ohledem na dobu odezvy.
- Je možná bezproblémová integrace/implementace s protokoly HTTP, protože HTTP je sám o sobě protokolem bez státní příslušnosti. (7)

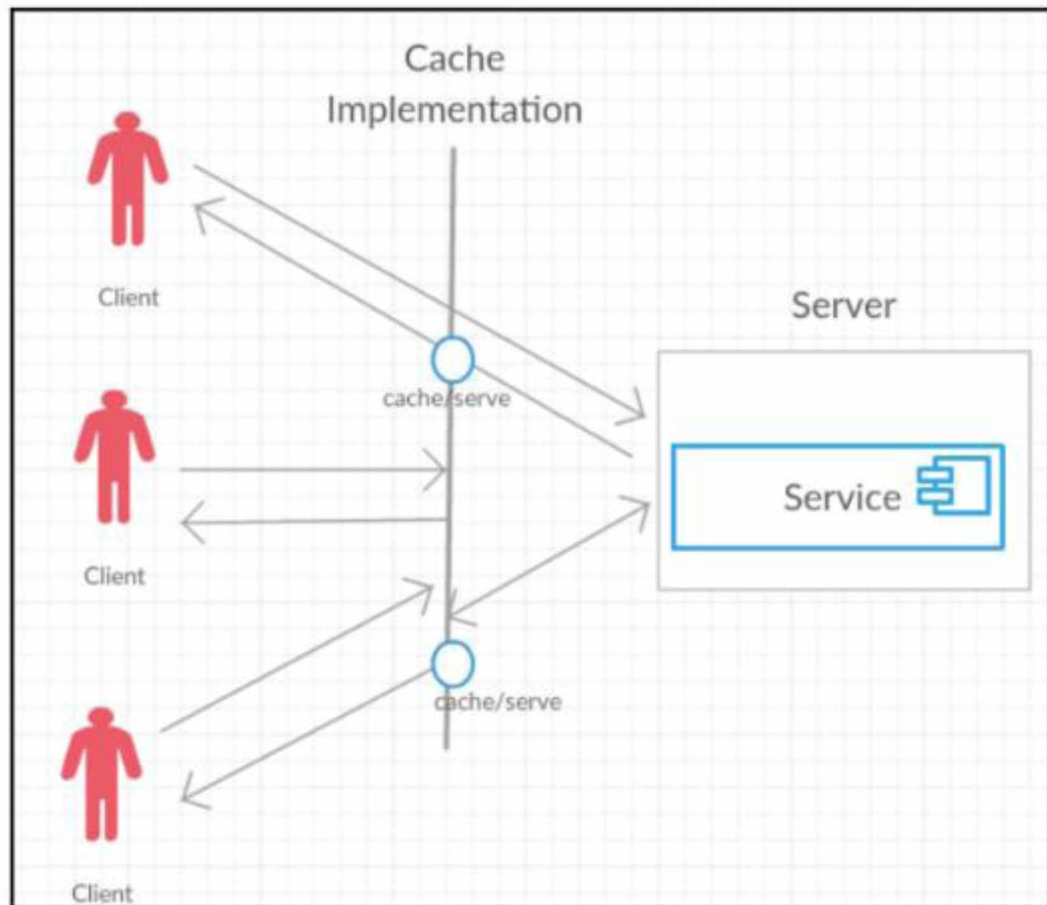
Nevýhody bezstavovosti:

- Zvýšení režijních nákladů na jednu interakci
- Každý požadavek webových služeb musí získat další informace, aby byl analyzován, aby server porozuměl stavu klienta z příchozí žádosti a v případě potřeby se staral o sezení klienta/serveru. (7)

2.2.4 Vyrovnávací paměť

Ukládání do mezipaměti je schopnost ukládat často přístupná data (v tomto kontextu odpověď), která slouží požadavkům klienta a nikdy nemusí generovat stejnou odpověď více než jednou, pokud to není nutné. Správně řízené ukládání do mezipaměti eliminuje částečné nebo úplné interakce mezi klientem a serverem a stále mu poskytuje očekávanou odpověď. Je zřejmé, že ukládání do mezipaměti přináší škálovatelnost a také výhody výkonu s rychlejšími časy odezvy a sníženým zatížením serveru. (7)

Z následujícího diagramu je zřetelné, že spotřebitel služby (klient) obdrží odpověď z mezipaměti a ne ze samotného serveru a několik dalších odpovědí je pak přímo ze serveru. Ukládání do mezipaměti tak pomáhá s částečným nebo úplným odstraněním některých interakcí mezi spotřebiteli služeb a pomáhá tak zlepšit účinnost a výkon:



Obrázek 4 – Diagram komunikace s využitím vyrovnávací paměti (7)

K dispozici jsou různé strategie nebo mechanismy ukládání do mezipaměti, jako jsou mezipaměti prohlížeče, vyrovnávací paměti proxy a mezipaměti brány (reverzní proxy), a existuje několik způsobů, jak můžeme řídit chování mezipaměti, například prostřednictvím pragmy, značek vypršení platnosti atd.

Následující tabulka poskytuje představu o různých záhlavích ovládacích prvků mezipaměti, které lze použít k doladění chování mezipaměti. (7)

Tabulka 1 – Různá záhlaví ovládacích prvků mezipaměti

Záhlaví	**Popis**	**Vzorky**
Expires	Atribut záhlaví představuje datum/čas, po kterém je odpověď považována za zastaralou	Expires: Fri, 12 Jan 201818:00:09 GMT
Cache-control	Záhlaví, které definuje různé směrnice (pro požadavky i odpovědi)	Max age=4500, cache-extension
E-Tag	Jedinečný identifikátor pro stav zdroje serveru	ETag: amiv2309u324klm
Last-modified	Záhlaví odpovědi pomáhá identifikovat čas, kdy byla odpověď vygenerována	Last-modified: Fri, 12 Jan 2018 18:00:09 GMT

Další informace o cache-control direktivách https://tools.ietf.org/html/rfc2616#section-14.9.

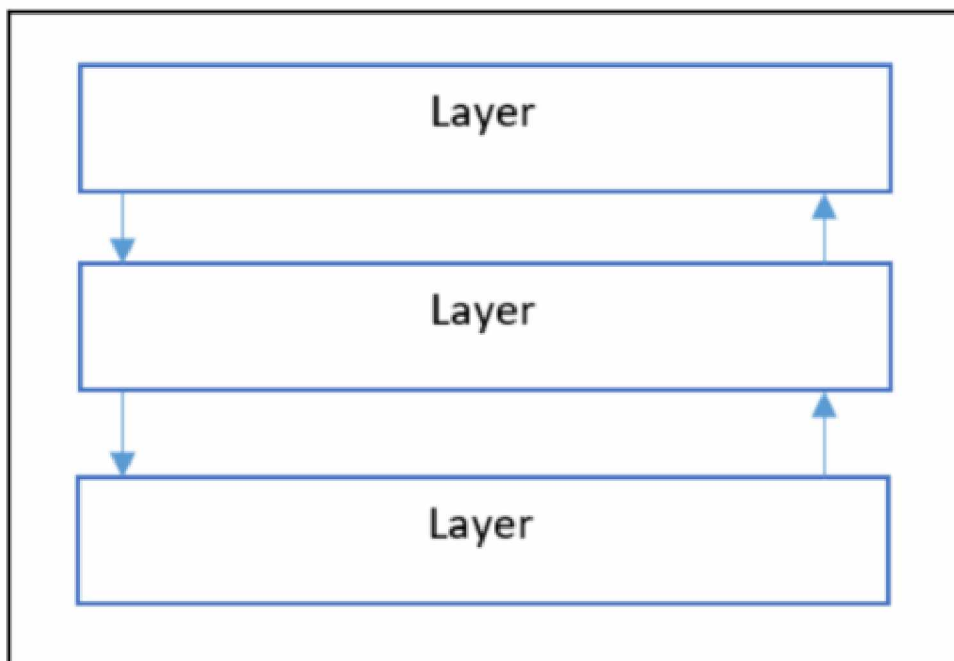
Výhody ukládání do mezipaměti

Je zřejmé, že ukládání do mezipaměti často přístupných dat má mnoho výhod, z nichž následující jsou významné:

- Snížená šířka pásma,
- Snížená latence (rychlejší doba odezvy),
- Snížená zátěž na serveru,
- Skrytí selhání sítě a obsluhování klienta s odezvou. (7)

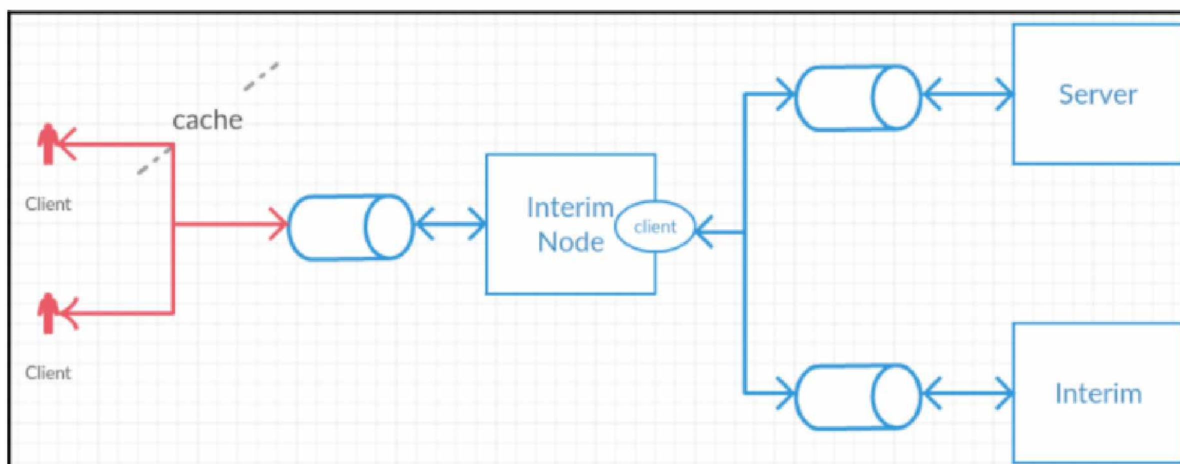
2.2.5 Vrstvený systém

Vrstvený systém se obecně skládá z vrstev s různými jednotkami funkčnosti. Základními charakteristikami vrstevnatých systémů je to, že vrstva komunikuje pomocí předem definovaných rozhraní a komunikuje pouze s vrstvou nad nebo vrstvou pod a vrstvy výše se spoléhají na vrstvy pod ní, aby mohly vykonávat své funkce. Vrstvy lze přidávat, odebírat, upravovat nebo měnit pořadí podle vývoje architektury. (7)



Obrázek 5 – Znázornění abstraktního vrstveného systému (7)

Například nasazujeme rozhraní REST API na server A, ukládáme data na server B a ověřujeme se serverem C. Klient, který volá rozhraní REST API, nemá žádné znalosti o serverech využití služeb. (7)



Obrázek 6 – Příklad komunikace ve vrstveném systému (7)

Architektonický styl REST naznačuje, že služby se mohou skládat z více architektonických vrstev. Vrstvy budou mít zveřejněné servisní kontrakty nebo zprostředkovatele. Zprostředkovatelé jsou vrstvy přítomné mezi klientem a serverem a mohou být přidány nebo odebrány, což je důležitější, bez změny rozhraní mezi součástmi. (7)

Zprostředkovatelé mají následující vlastnosti:

- Zprostředkovatelé mohou být součástmi middlewaru zaměřeného na události a vytvářet vrstvy zpracování mezi spotřebiteli a službami.
- Mohou to být servery proxy (vybrané klientem, aby poskytovaly rozhraní se službami přenosu dat, zvýšeným výkonem nebo ochranou zabezpečení).
- Mohou to být také brány (vybrané serverem nebo sítí a použity pro překlad dat, vynucení zabezpečení a zvýšení výkonu). (7)

Klient nemusí být schopen zjistit, zda je připojen ke službám přímo s koncovým bodem serveru nebo před dosažením skutečného serveru. Zprostředkující servery pomáhají dosáhnout lepší škálovatelnosti systému tím, že mají vyrovnávače zatížení a sdílené mezipaměti. Vrstvy mohou také vynucovat bezpečnostní zásady pro své volající klienty. (7)

Další vlastnosti vrstveného systému:

- Služba volaná klientem nezveřejňuje žádné informace o jiných službách, které používá ke zpracování požadavků klienta. Jinými slovy, spotřebitel služby (klient) ví pouze o službě, kterou přímo volá, a neví o dalších službách, které volaná služba používá ke zpracování svých požadavků.
- Zprávy mezi klientem a serverem jsou zpracovávány zprostředkovateli, kteří pomáhají klientům osvobodit se od logiky zpracování zpráv za běhu a zbavují je porozumění tomu, jak jsou tyto zprávy zpracovávány na jiných úrovních.
- Pro stabilitu a škálovatelnost je velmi důležité přidávat nebo odstraňovat vrstvy ve vrstveném systému bez jakýchkoli změn pro uživatele služeb.
- Zprávy s požadavky a odpověďmi nesdělují příjemcům žádné informace o tom, z jaké vrstvy zpráva pochází. (7)

Další výhody vrstveného systému:

- Zapouzdřuje starší služby,
- Zavádí zprostředkovatele,
- Omezuje složitost systému,
- Zlepšuje škálovatelnost. (7)

2.2.6 Kód na vyžádání

V distribuovaných počítačích je kód na vyžádání (angl. *„code on demand (COD)"*) jakákoli technologie, která umožňuje serveru odeslat klientský programový kód pro provedení v klientském počítači na žádost z klientského softwaru. Některé známé příklady paradigmatu COD na internetu jsou applety Java, jazyk Adobe ActionScript pro přehrávač Flash a JavaScript. Výhody COD lze také nazvat:

- COD je volitelné omezení REST a je navrženo tak, aby umožňovalo obchodní logiku v klientském webovém prohlížeči, appletech, JavaScriptu a ActionScript (Flash).
- COD umožňuje klientům být flexibilní, protože server rozhoduje o tom, jak by měly být konkrétní prvky zpracovány na straně klienta. Například pomocí COD může klient načíst akční skripty, jako je JavaScript, applety (v těchto dnech se příliš nepoužívají), Flex skripty, k šifrování komunikace mezi klientem a serverem, takže základní servery nebudou vědět o žádných konkrétních metodách šifrování, používaných v procesu.
- COD se může vztahovat také na služby a spotřebitele služeb. Například navrhování služby může serverům umožnit dynamicky odkládat části logiky pro programy klientských služeb. Tento přístup zpožděného provádění kódu na straně klienta je oprávněný, když může spotřebitel logiku spotřebitele provádět efektivněji.
- RESTful aplikace mohou velmi dobře využívat klienti, kteří podporují COD. Například webové prohlížeče umožňují serverům vrátit skripty nebo odkazy, které lze provést na straně klienta. Tento druh zpožděného provádění kódu pomáhá rozšířit možnosti klienta, aniž by uživatel musel instalovat nový klientský software.
- Ve stylu COD má klientská komponenta přístup k sadě zdrojů, ale nikoli k know-how, jak je zpracovat. Odesílá požadavek vzdálenému serveru na kód představující toto know-how, přijímá tento kód a provádí jej lokálně. (7)

Na druhou stranu, používání COD však sníží viditelnost základního API a ne každé API upřednostňuje tento druh flexibility. (7)

COD je klasifikován jako nepovinný; architektury, které tuto funkci nepoužívají, lze stále považovat za RESTful. (7)

3 Vývoj RESTful API na platformě Java

Tato kapitola obsahuje alternativní přístupy k implementaci RESTful API pomocí programovacího jazyku Java. Java je jeden z nejpopulárnějších jazyků co se týče vývoje backendu.

Java je programovací jazyk a výpočetní platforma, která byla poprvé vydána společností Sun Microsystems v roce 1995. Existuje mnoho aplikací a webů, které nebudou fungovat, pokud nemáte nainstalovanou Javu, a každý den se vytvoří další. Java je rychlá, bezpečná a spolehlivá. (8)

3.1 Populární frameworky

Nejpopulárnějšími frameworky pro vytváření RESTful API v Java prostředí jsou specifikace JAX-RS a Spring MVC.

3.1.1 JAX-RS

Java API pro RESTful Web Services (JAX-RS) je specifikace API pro programovací jazyk Java, která poskytuje podporu při vytváření webových služeb podle architektonického vzoru REST. JAX-RS je kolekce rozhraní a anotací Java, které zjednodušují vývoj aplikací REST na straně serveru. (10)

JAX-RS je jednou z nejnovějších generací Java API, které používají anotace Java, aby se snížila potřeba standardních základních tříd. Anotace se používají k nasměrování požadavků klientů na vhodné metody třídy Java a pro deklarativní mapování dat požadavků na parametry těchto metod. Anotace se také používají k poskytování statických metadat pro vytváření odpovědí. JAX-RS také poskytuje tradičnější třídy a rozhraní pro dynamický přístup k datům požadavků a pro přizpůsobení odpovědí. (9)

```java
@Path("/show-on-screen")
public class JerseyHelloWorldService
{
    @GET
    @Path("/{message}")
    public Response getMsg(@PathParam("message") String msg)
    {
        String output = "Message requested : " + msg;
        return Response.status(200).entity(output).build();
    }
}
```

Obrázek 7 – Ukázka implementací webové služby pomocí JAX-RS

Na obrázku 7 je zobrazen příklad implementací webové služby pomocí frameworku JAX-RS.

Anotace `@Path` nad třídou `JerseyHelloWorldService` identifikuje tuto třídu jako příjemce požadavků, které kontejner servletu obdrží na URL `/show-on-screen`. Anotace `@GET` určuje typ požadavků, jež obdrží metoda `getMsg`. Anotace `@Path` zase nad danou metodou upřesňuje URL požadavků.

Tělo metody vytváří objekt typu `Response` pomocí návrhového vzoru stavitel (anglicky Builder). Nejdřív se pomocí metody `status(200)` předá HTTP stavový kód 200, který znamená úspěšný HTTP požadavek. Pak se pomocí metody `entity(output)` stanoví tělo odpovědi, což v tomto případě je řetězec.

3.1.2 Spring MVC

Typicky je Spring popsán jako framework pro vytváření Java aplikací. Spring Framework vznikl z knihy Expert One-on-One: Design a vývoj J2EE od Rod Johnson. V posledním desetiletí se Spring Framework dramaticky rozrostl v základních funkcích, přidružených projektech a podpoře komunity. (11)

Modul Spring Web MVC poskytuje implementaci tradičního vzoru Model View Controller. Zatímco REST nenabízí použití žádného specifického vzoru, použití vzoru MVC je zcela přirozené. (2)

```java
@RestController
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map < String, Appointment > get() {
        return appointmentBook.getAppointmentsForToday();
    }
}
```

Obrázek 8 – Ukázka implementací webové služby pomocí Spring MVC

Na obrázku 8 je zobrazen příklad implementací webové služby pomocí frameworku Spring a jeho modulu Spring MVC.

Anotace `@RestController` je pohodlná anotace pro vytváření Restful kontrolérů. Identifikuje danou třídu jako kontrolér. Převádí odpověď metod na JSON nebo XML.

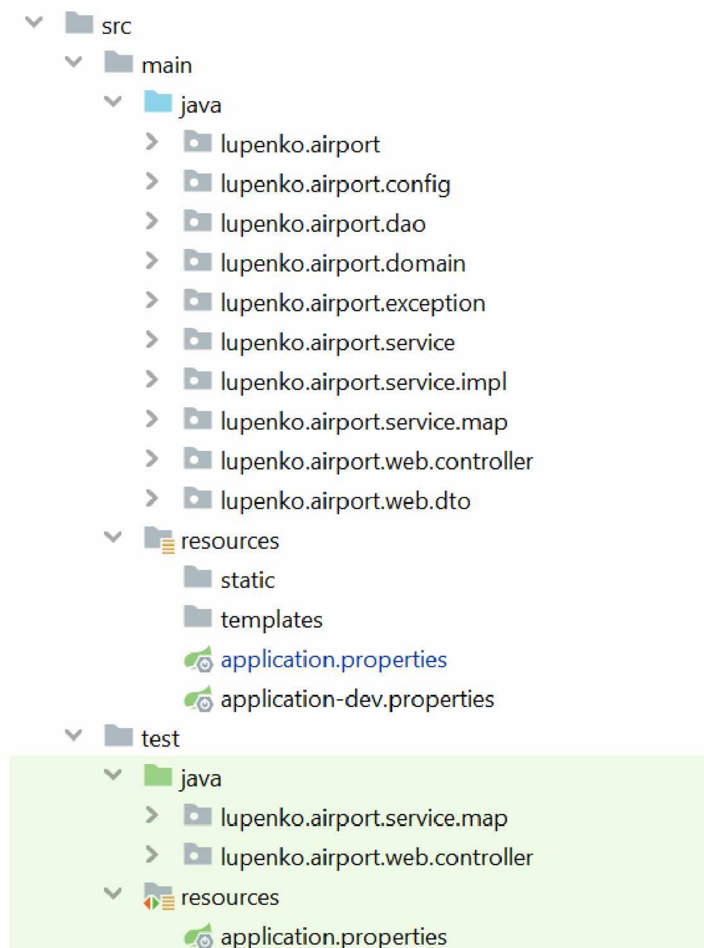
Anotace `@RequestMapping` se používá pro mapování adresy URL na celou třídu nebo konkrétní metodu. V tomto příkladu je @RequestMapping používán na několika místech. První použití je na úrovni třídy, což znamená, že všechny metody zpracování na tomto kontroléru patří do cesty`/appointments`. Metoda `get()` má další upřesnění `@RequestMapping`: přijímá pouze požadavky GET, což znamená, že HTTP GET požadavek pro `/appointments` volá tuto metodu.

4 Realizace cílové aplikace

Cílem bakalářské práce je vybudovat a otestovat RESTful API se zaměřením na provoz letiště. Konkrétně bude API umožňovat správu příletů, odletů, rezervaci letenek, databázi letů, případně některé další technologické aspekty. Vytvořené API bude pokryto jednotkovými a integračními testy. API bude také vhodně doplněno vizualizační stránkou s přehledem všech endpointů a použitých datových modelů. Webová stránka umožní vyvolat a otestovat konkrétní endpointy.

Pro vývoj aplikace popsané v této práci se používá programovací jazyk Java a verze 8 JDK, kterou lze zdarma stáhnout z oficiální stránky Oracle. Po instalaci je nutné nastavit systémovou proměnnou PATH, přidáním cesty adresáře bin nainstalovaného JDK.

K realizaci REST API byl zvolen framework Spring a jeho nadstavba Spring Boot. Protože autor má nejvíc zkušeností z prostředí Spring, a většina existujících REST API je v Java prostředí, je v současné době napsána pomoci frameworku Spring.

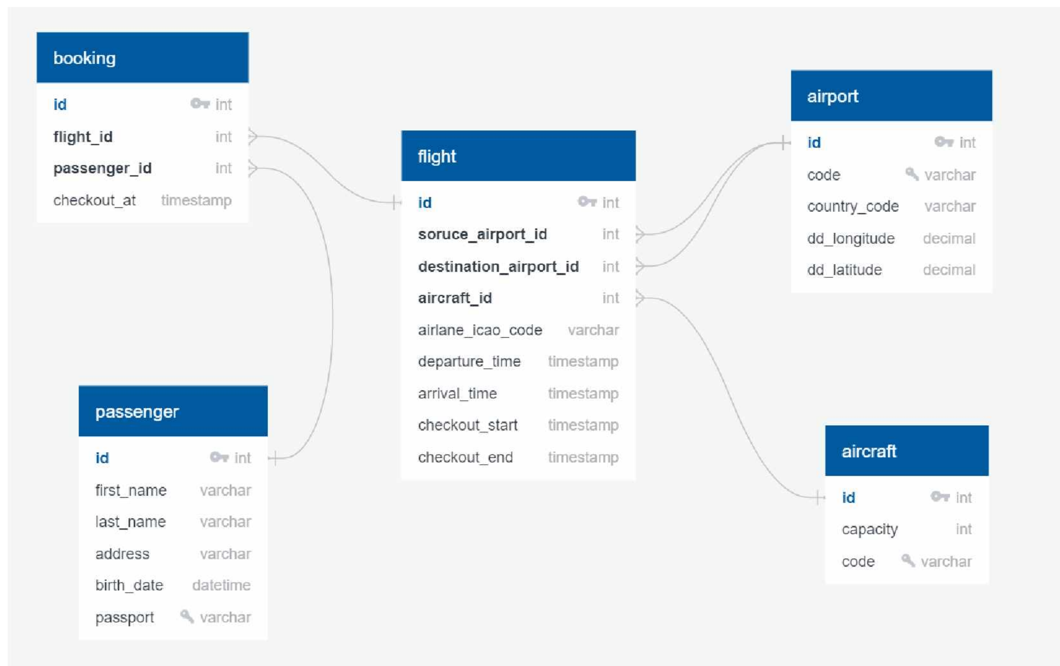


Obrázek 9 – Struktura balíčku

4.1 Vrstva modelu

Balíček `lupenko.airport.domain` obsahuje třídy, které reprezentují vrstvu modelu. Vrstva modelu je doménově specifická reprezentace informací, s nimiž aplikace pracuje.

Všechny entity v aplikaci lze znázornit na následujícím ER diagramu:



Obrázek 10 – Entitně vztahový diagram

Entita `flight`

Aby aplikace umožňovala správu příletu a odletu, je potřeba mít třídu, která reprezentuje let. Každý let se navazuje na cílové letiště, výchozí letiště a letadlo. Let taky obsahuje důležité parametry jako jsou: ICAO kód letecké společnosti, čas odletu a příletu, čas zahájení a ukončení registrace.

```java
public class Flight {
 @Id
 @GeneratedValue
 private Long id;
 @ManyToOne(cascade = CascadeType.PERSIST)
 private Airport sourceAirport;
 @ManyToOne(cascade = CascadeType.PERSIST)
 private Airport destinationAirport;
 @ManyToOne(cascade = CascadeType.PERSIST)
 private Aircraft aircraft;
 private String airlaneIcaoCode;
 private Instant departureTime;
 private Instant arrivalTime;
 private Instant checkoutStart;
 private Instant checkoutEnd;

 @OneToMany(fetch = FetchType.LAZY, mappedBy = "flight",
           cascade = CascadeType.ALL, orphanRemoval = true)
 private List < Booking > bookings = new ArrayList < > ();

}
```

Obrázek 11 – Zjednodušená ukázka třídy `Flight`

Na obrázku 11 je zjednodušená ukázka třídy Flight. Nad některými parametry třídy jsou anotace z balíčku `javax.persistence.*`. Anotace z tohoto balíčku jsou určené primárně pro ORM funkcionality. Následující tabulka obsahuje popis anotací používané v entitách projektu.

Tabulka 2 – Popis anotací používané v entitách projektu

Anotace	Popis
@Id	Určuje primární klíč entity.
@GeneratedValue	Určuje, že pro tento parametr bude automaticky vygenerována hodnota při vkládání objektu do databáze. Toto je primárně určeno pro parametry primárních klíčů.
@OneToMany	Určuje one-to-many vztah mezi entitami. Znamená, že jeden řádek v tabulce je mapován na více řádků v jiné tabulce. Může obsahovat parametry: • fetch — typ načtení asociace • mappedBy — jméno parametru, který vlastní vztah • cascade — typ operací, které musí být prováděny nad entity • orphanRemoval — má-li JPA smazat entity při odstranění hlavní entity
@ManyToOne	Určuje many-to-one vztah mezi entitami. Znamená, že mnoho instancí entity je mapováno na jednu instanci jiné entity. Je opačným případem one-to-many vztahu.

Entita `booking`

Jednou ze základních operací v cílové aplikaci je rezervace letenky. Entita `booking` reprezentuje rezervaci. Cestující si může rezervovat letenku na určitý let. Rezervace vytváří vazbu mezi letem a cestujícím. Obsahuje parametry: čas registrace cestujícího na let a číslo místenky.

```
public class Booking {
 @Id
 @GeneratedValue
 private Long id;
 @ManyToOne(cascade = CascadeType.PERSIST)
 private Flight flight;
 @ManyToOne(cascade = CascadeType.PERSIST)
 private Passenger passenger;
 private Instant checkout;
}
```

Obrázek 12 – Zjednodušená ukázka třídy `Booking`

Entita `passenger`

Reprezentuje cestujícího. Obsahuje parametry: jméno, příjmení, adresa, datum narození, číslo pasu. V aplikaci je zajištěno, aby číslo pasu bylo unikátní.

```
public class Passenger {
 @Id
 @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private String address;
 private LocalDate birthDate;
 private String passport;
}
```

Obrázek 13 – Zjednodušená ukázka třídy `Passenger`

Entita `airport`

Reprezentuje letiště. Letiště obsahuje následující parametry: unikátní kód letiště, kód státu, zeměpisnou délku a šířku. V aplikaci a databázi je zajištěno, aby kód letiště byl unikátní.

```
public class Airport {
 @Id
 @GeneratedValue
 private Long id;
 private String code;
 private String countryCode;
 private BigDecimal ddLongitude;
 private BigDecimal ddLatitude;
}
```

Obrázek 14 – Zjednodušená ukázka třídy `Airport`

Entita `aircraft`

Reprezentuje letadlo. Má dva parametry: kapacita míst a kód letadla. V aplikaci a databázi je zajištěno, aby kód letadla byl unikátní.

```
public class Aircraft {
 @Id
 @GeneratedValue
 private Long id;
 private Long capacity;
 private String code;
}
```

Obrázek 15 – Zjednodušená ukázka třídy `Aircraft`

4.2 Databázová vrstva

Pro úložiště byl zvolen databázový systém PostgreSQL. PostgreSQL je výkonný, otevřený, zdrojový objektově relační databázový systém, který používá a rozšiřuje jazyk SQL v kombinaci s mnoha funkcemi, které bezpečně ukládají a škálují nejsložitější pracovní zatížení dat. Původ PostgreSQL sahá do roku 1986 jako součást projektu POSTGRES na University of California v Berkeley a má více než 30 let aktivního rozvoje na základní platformě. (12)

Balíček `lupenko.airport.dao` obsahuje rozhraní, která poskytují základní operace nad úložištěm. Všechna rozhraní nedefinují žádné metody, ale dědí základní rozhraní `JpaRepository` z modulu Spring Data JPA. Spring Data JPA poskytuje podporu úložiště pro Java Persistence API (JPA). Usnadňuje vývoj aplikací, které potřebují přístup ke zdrojům dat JPA. (13)

Většina konfiguračních parametrů přístupů do úložiště se vytváří automaticky pomocí Spring Boot. Ale základní konfiguraci je stejně potřeba definovat manuálně. Přihlašovací údaje jsou uvedené v externím konfiguračním souboru `application.properties`.

```
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/airport
spring.datasource.username=airport
spring.datasource.password=airport
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```

Obrázek 16 – Obsah konfiguračního souboru `application.properties`

Pro režim vývojáře, když nepotřebujeme sahat do reálného úložiště, se používá in-memory databázový systém H2. Jeho konfigurace se nachází v souboru `application-dev.properties`.

4.3 Vrstva kontroléru

Balíček `lupenko.airport.web` obsahuje dva podbalíčky `controller` a `dto`.

Balíček `lupenko.airport.web.controller`

Obsahuje třídy – kontroléry. Funkce kontroléru je zpracovat požadavky HTTP. Jinými slovy, HTTP obdrží požadavek od uživatele a předá informace službám, které jsou logické pro splnění požadavku. Nakonec kontrolér doručí odpověď uživateli.

Aby framework Spring věděl, jaká třída je kontrolérem a mohl směrovat požadavky z webového serveru přímo k potřebné třídě, nad třídou musí být anotace `@Controller` nebo `@RestController`. `@RestController`, na rozdíl od `@Controller`, zajišťuje převádění všech návratových objektů dané třídy na JSON nebo XML reprezentaci. Nakonec musíme naprogramovat metody kontroléru. Každá metoda reprezentuje jednotlivé endpointy. Na obrázku 17 je vidět, že metoda `createAircaft` reprezentuje endpoint, který přijímá POST požadavky s obsahem ve formátu JSON a vrátí odpověď taky ve formátu JSON.

```java
@RestController
@RequestMapping(value = "/aircraft")
public class AircraftController {

 private final AircraftService aircraftService;

 public AircraftController(final AircraftService aircraftService) {
  this.aircraftService = aircraftService;
 }

 @RequestMapping(method = RequestMethod.POST, consumes = MediaType.APPLICATION_JSON_VALUE,
                 produces = MediaType.APPLICATION_JSON_VALUE)
 @ResponseStatus(HttpStatus.CREATED)
 @ApiOperation("Creates a new aircraft")
 public AircraftDto createAircraft(@RequestBody AircraftDto dto) {
  return aircraftService.createAircraft(dto);
 }

 @RequestMapping(path = "/{id}", method = RequestMethod.DELETE)
 @ResponseStatus(HttpStatus.NO_CONTENT)
 @ApiOperation("Deletes aircraft")
 public void deleteAircraft(@PathVariable Long id) {
  aircraftService.deleteAircraft(id);
 }

 // ...

}
```

Obrázek 17 – Ukázka třídy `AircraftController`

Na obrázku 17 je ukázka třídy `AircraftController`. Pro mapování požadavků jsou zde použité anotace `@RestController` a `@RequestMapping` vysvětlené v kapitole 3.1.2. Navíc třída obsahuje anotace `@ApiOperation` a `@ResponseStatus`. `@ApiOperation` je anotace z nástroje Swagger UI, která přidává popis určitému endpointu. Anotace `@ResponseStatus` určuje stavový kód odpovědi.

Balíček `lupenko.airport.web.dto`

Obsahuje DTO třídy. DTO – Ze zkratky *„Data Transfer Object"*. (14) Lze přeložit jako „objekt pro přenos dat".

Když pracujeme se vzdáleným rozhraním, je každé volání drahé. V důsledku toho musíme snížit jejich počet, což znamená, že při každém volání musíme přenést více dat. Jedním ze způsobů, jak toho dosáhnout, je použít spoustu parametrů. To je však často nepříjemné programovat. Řešením je vytvoření DTO, který dokáže uchovat všechna data pro volání. (14)

```java
public class BookingDto {
 @JsonProperty(access = JsonProperty.Access.READ_ONLY)
 private Long id;
 @NonNull
 private Long flightId;
 @NonNull
 private Long passengerId;
 private Instant checkout;
}
```

Obrázek 18 – Ukázka třídy `BookingDto`

4.4 Servisní vrstva

Balíčk `lupenko.airport.web.service` obsahuje třídy služeb. Lze rozlišit dvě vrstvy: třídy DAO, které pracují s databází, kde jsou data umístěna, a třídy služeb. Tato střední vrstva je zodpovědná za transformaci požadavků kontroléru tak, aby byly pochopeny třídami DAO a naopak. Pro účely transformace mezi DTO a databázovým objektem používají služby třídy-mapovače z balíčku `lupenko.airport.web.service.map.` Mapovače jsou generovány během kompilace projektu pomocí anotačního procesoru MapStruct.

Obrázek 19 ukazuje příklad mapovače, který transformuje DAO letiště v DTO a naopak. Metody jsou označené jako abstraktní, implementace jednotlivých metod se generuje během kompilace.

```java
@Mapper(componentModel = "spring")
public abstract class AirportMapper {

 @Mapping(target = "id", ignore = true)
 public abstract Airport dtoToEntity(AirportDto dto);

 public abstract AirportDto toDto(Airport entity);

 @Mapping(target = "id", ignore = true)
 public abstract void copyParametersFromDto(AirportDto dto, @MappingTarget Airport entity);

}
```

Obrázek 19 – Ukázka třídy-mapovače `AirportMapper`

Kromě toho jsou služby zodpovědné za kontrolu, že data z controlleru jsou zpracovatelná a zahrnují všechna povinná pole. Dojde-li k některému z výše uvedených problémů, je vyvolána řádná výjimka a přestane se vyhovovat žádosti. Příkladem ošetření nezpracovatelných dat může býtslužba `BookingServiceImpl`, která je zodpovědná za zpracování požadavků spojených s rezervací letenek. Tato služba obsahuje metodu

`checkAircraftCapacityByFlight`, která ošetřuje, jestli letadlo má dostatečný počet míst, aby cestující mohl udělat rezervaci.

```
@Service
@Slf4j
public class BookingServiceImpl implements BookingService {

 private final BookingRepository bookingRepository;
 private final BookingMapper bookingMapper;

 @Autowired
 public BookingServiceImpl(final BookingRepository bookingRepository,
                           final BookingMapper bookingMapper) {
  this.bookingRepository = bookingRepository;
  this.bookingMapper = bookingMapper;
 }

 private void checkAircraftCapacityByFlight(final Flight flight) {
  Long capacity = flight.getAircraft().getCapacity();
  int bookings = flight.getBookings().size();
  if (capacity == bookings) {
   throw new AircraftCapacityException(capacity, flight.getAircraft().getId());
  }
 }

 @Override
 public void deleteBooking(final Long id) {
  log.info("Deleting booking {}", id);
  bookingRepository.deleteById(id);
 }

 // ...
}
```

Obrázek 20 – Ukázka třídy-služby `BookingServiceImpl`

4.5 Swagger UI

Cílem vizualizační stránky je nabídnout možnost uživateli tohoto API seznámit se s obsahem a interaktivně vyzkoušet jednotlivé endpointy.

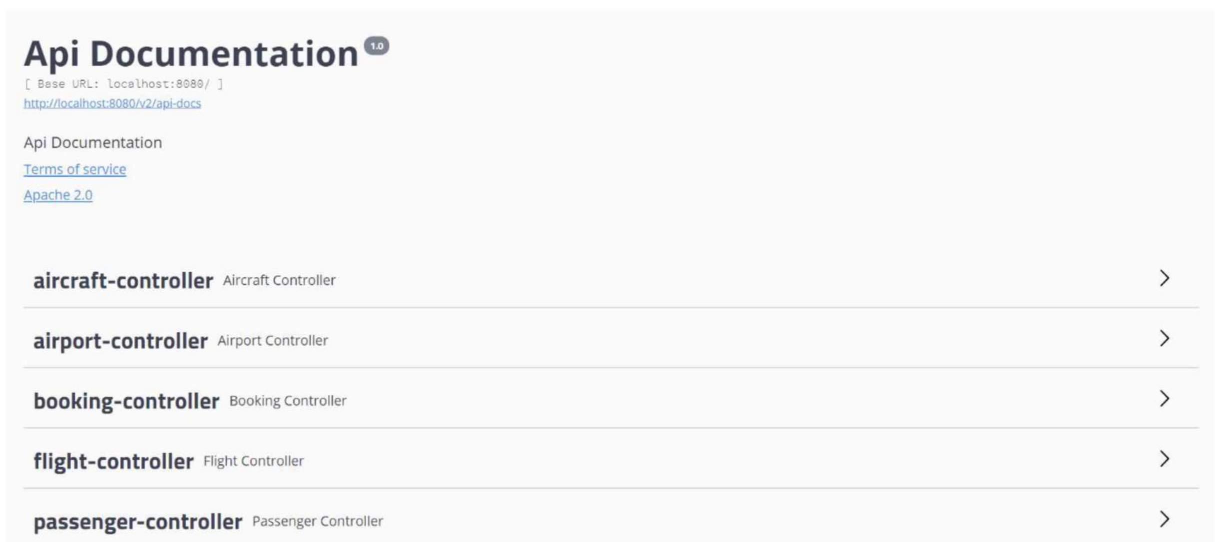
Pro implementaci vizualizační stránky byl zvolen nástroj Swagger UI. Swagger UI je projekt s otevřeným zdrojovým kódem, který vizuálně vykresluje dokumentaci pro API. (15) Swagger UI pracuje tak, že za běhu prozkoumá aplikaci, aby odvodil sémantiku API založenou na konfiguracích Spring, struktuře tříd a různých anotacích. Proto je potřeba nastavit v konfiguraci Swagger balíček, který obsahuje controllery, jež chceme vizualizovat.

```java
@Configuration
@EnableSwagger2
public class SpringFoxConfig {
 @Bean
 public Docket api() {
  return new Docket(DocumentationType.SWAGGER_2)
    .select()
    .apis(RequestHandlerSelectors.basePackage("lupenko.airport.web"))
    .paths(PathSelectors.any())
    .build();
 }
}
```

Obrázek 21 – Ukázka konfigurace `Swagger UI`

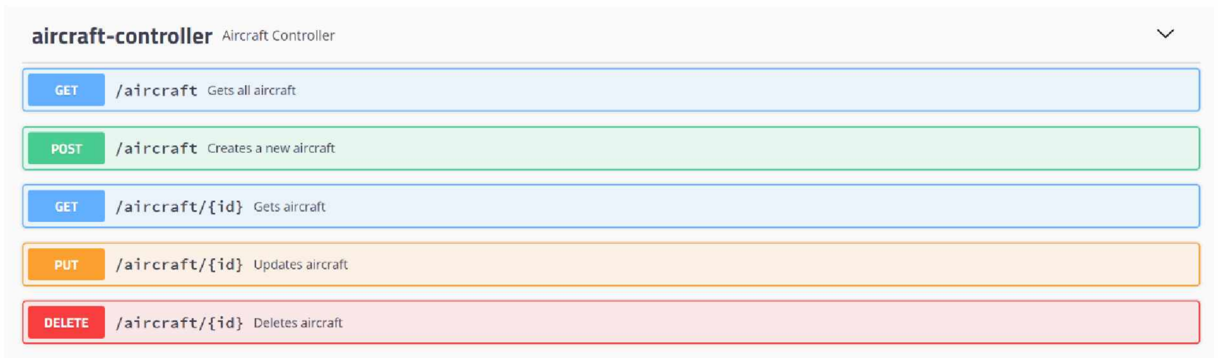
Na obrázku 21 je ukázka konfigurace nástroje Swagger UI. Třída SpringFoxConfig obsahuje metodu `api()`. Tato metoda vybuduje objekt typu Docket. Objekt přijímá název balíčku, který obsahuje DTO a kontrolery, jež potřebujeme zobrazit na vizualizační stránce. Metoda `api()` je označena anotací `@Bean`, což znamená, že Spring při spuštění programu vyvolá tuto metodu.

Výsledkem je vizualizační stránka s možností vyvolat libovolný endpoint v kontroleru a zadat různé parametry požadavku.



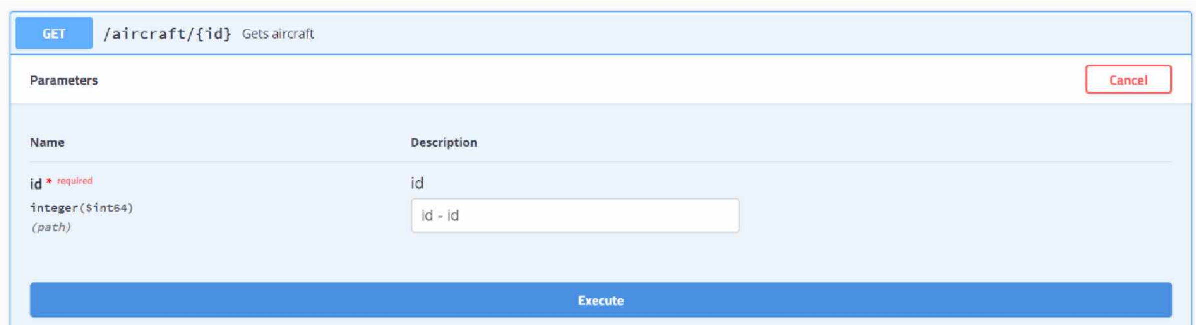
Obrázek 22 – Ukázka seznamu kontroleru na vizualizační stránce

Příklad controlleru pro letadla:



Obrázek 23 – Ukázka seznamu endpointu ve třídě AircraftController na vizualizační stránce

Příklad endpointu pro získání určitého letadla pomocí jeho identifikátoru:



Obrázek 24 – Ukázka enpointu `/aircraft/{id}`

4.6 Konfigurace

V aplikaci jsou k dispozici dva profily pro spuštění aplikací default a dev. Jak bylo uvedeno v kapitole 4.2, konfigurace dev profilu se nachází v souboru application-dev.properties. A obsahuje alternativní databázovou konfiguraci pro testování a vývojářské účely. Při spouštění čte Spring konfigurační soubor a nastavuje databázový systém na in-memory databázový systém H2 v případě dev profilu a na PostgreSQL v případě default profilu. Spustit aplikaci v dev profilu můžeme pomocí příkazu `mvn spring-boot:run -Dspring-boot.run.profiles=dev`.

```
[main] lupenko.airport.AirportBpApplication: The following profiles are active: dev
```

Obrázek 25 – Výstup v logu oznamující aktivní profil

4.7 Testování

Při vývoji aplikací je testování důležitým způsobem, jak zajistit, aby dokončená aplikace fungovala podle očekávání a splňovala všechny druhy požadavků (architektonické, bezpečnostní, uživatelské požadavky atd.). Při každé změně máme zajistit, aby provedené

změny neovlivnily existující logiku. Udržování trvalého prostředí pro vytváření a testování je rozhodující pro zajištění vysoce kvalitních aplikací. Opakovatelné testy s vysokým pokrytím celého kódu umožňují nasazovat změny v aplikacích s vysokou mírou důvěry. V prostředí podnikového vývoje existuje mnoho druhů testování, která se zaměřují na každou vrstvu v podnikové aplikaci, a každý druh testování má své vlastní charakteristiky a požadavky. (11)

4.7.1 Integrační testy

V rámci podnikového testování se integrační testování týká testování interakce skupiny tříd v různých aplikačních vrstvách pro konkrétní logiku podnikání. V prostředí pro testování integrace by se obvykle měla vrstva služeb testovat s vrstvou perzistence a měla by být k dispozici databáze. Avšak s vývojem aplikační architektury a vývojem lehkých in-memory databází se nyní běžně praktikují „jednotkové testy“ servisní vrstvy s vrstvou perzistence a databází typu back-end jako celek. (11) Pro tyto účely se používá modul Spring Test a základní třída `MockMvc`, která umožní volat deklarované endpointy v kontextu.

V integračním testování cílové aplikace používáme JPA 2 s Hibernate a Spring Data JPA jako poskytovatele perzistence a H2 jako databázi. V této architektuře je při testování servisní vrstvy méně důležité „mockovat“ Hibernate a Spring Data JPA. Testovací konfigurace kontextu je uvedena ve zvláštním konfiguračním souboru v adresáři src/test/resources.

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

Obrázek 26 – Ukázka konfiguračního souboru `application-dev.properties`

Třídy pro integrační testování se nacházejí v balíčku:

`lupenko.airport.web.controller`.

Například třída `AircraftControllerTest` obsahuje metody pro testování jednotlivých endpointů controlleru `AircraftController`.

```java
@AutoConfigureMockMvc
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class AircraftControllerTest {

 @Autowired
 private MockMvc mockMvc;

 @Autowired
 private AircraftRepository aircraftRepository;

 @Autowired
 private ObjectMapper objectMapper;

 @Test
 public void getAllAircraftDefaultPaging_oneAircraftInDb_aircraftReturned()
    throws Exception {
  Aircraft aircraft = new Aircraft(10 L, "TEST");
  aircraftRepository.save(aircraft);

  mockMvc.perform(MockMvcRequestBuilders.get("/aircraft")
    .accept(MediaType.APPLICATION_JSON))
   .andDo(print())
   .andExpect(status().isOk())
   .andExpect(jsonPath("$.content[0].id", is(aircraft.getId())))
   .andExpect(jsonPath("$.content[0].code", Matchers.is(aircraft.getCode())))
   .andExpect(jsonPath("$.content[0].capacity", is(aircraft.getCapacity())))
   .andExpect(jsonPath("$.totalElements", Matchers.is(1)));
 }
 // ...
}
```

Obrázek 27 – Ukázka testovací třídy pro `AircraftController`

Na obrázku 27 je jedna z testovacích metod:

`getAllAircraftDefaultPaging_oneAircraftInDb_aircraftReturned`

třídy `AircraftControllerTest`, která otestuje návratovou hodnotu endpointu `GET/aircraft`. V uvedeném příkladu úložiště obsahuje jedno letadlo, proto se ověřuje, zda endpoint vrací jenom jeden záznam.

4.7.2 Jednotkové testy

Správný jednotkový test ověřuje pouze logiku metod v rámci třídy, přičemž všechny ostatní závislosti jsou „mockované“ správným chováním. (11) Mockování je replika nebo napodobení něčeho. Objekt, který potřebujeme otestovat, může mít závislosti na jiných složitých objektech. Chceme-li izolovat chování objektu, který máme otestovat, nahradíme ostatní objekty falešnými simulátory chování skutečných objektů. Jednoduše řečeno, mockování je vytváření objektů, které simulují chování skutečných objektů.

Pro mockování používáme knihovnu Mockito. V cílové aplikaci se jednotkové testy používají pro ověření logiky mapovačů z balíčku `lupenko.airport.web.service.map`. Chceme si být jisti, že všechny transformace mezi DTO a DAO probíhají správně.

```
@Test
public void dtoToEntity() {
 AirportDto dto = new AirportDto(22 L, "CODE", "US", new BigDecimal("2.4"), new BigDecimal("2.7"));

 Airport entity = airportMapper.dtoToEntity(dto);

 assertNull(entity.getId());
 assertEquals(dto.getCountryCode(), entity.getCountryCode());
 assertEquals(dto.getCode(), entity.getCode());
 assertEquals(dto.getDdLongitude(), entity.getDdLongitude());
 assertEquals(dto.getDdLatitude(), entity.getDdLatitude());
}

@Test
public void toDto() {
 Airport entity = new Airport(22 L, "CODE", "US", new BigDecimal("2.4"), new BigDecimal("2.7"));

 AirportDto dto = airportMapper.toDto(entity);

 assertEquals(entity.getId(), dto.getId());
 assertEquals(entity.getDdLongitude(), dto.getDdLongitude());
 assertEquals(entity.getDdLatitude(), dto.getDdLatitude());
 assertEquals(entity.getCountryCode(), dto.getCountryCode());
 assertEquals(entity.getCode(), dto.getCode());
}
```

Obrázek 28 – Příklad jednotkových testů třídy `AirportMapper`

Na obrázku 28 je příklad jednotkových testů třídy `AirportMapper`.

Metoda `dtoToEntity()` ověřuje správnost převádění objektu `AirportDto` na objekt `Airport`. Naopak metoda `toDto()` ověřuje správnost převádění objektu `Airport` na `AirportDto`.

```
@BeforeAll
public static void mockRepositories() {
    when(PASSENGER_MOCK.getId()).thenReturn(PASSENGER_ID);
    PassengerRepository passengerRepositoryMock = mock(PassengerRepository.class);
    when(passengerRepositoryMock.findById(PASSENGER_ID)).thenReturn(Optional.of(PASSENGER_MOCK));
    ReflectionTestUtils.setField(bookingMapper, "passengerRepository", passengerRepositoryMock);

    when(FLIGHT_MOCK.getId()).thenReturn(FLIGHT_ID);
    FlightRepository flightRepositoryMock = mock(FlightRepository.class);
    when(flightRepositoryMock.findById(FLIGHT_ID)).thenReturn(Optional.of(FLIGHT_MOCK));
    ReflectionTestUtils.setField(bookingMapper, "flightRepository", flightRepositoryMock);
}

@Test
public void dtoToEntity() {
    BookingDto dto = new BookingDto(22L, FLIGHT_ID, PASSENGER_ID, Instant.EPOCH);

    Booking entity = bookingMapper.dtoToEntity(dto);

    assertNull(entity.getId());
    assertEquals(FLIGHT_MOCK, entity.getFlight());
    assertEquals(PASSENGER_MOCK, entity.getPassenger());
    assertEquals(Instant.EPOCH, entity.getCheckout());
}
```

Obrázek 29 – Příklad jednotkových testů s použitím „mockování“

Na obrázku 29 je příklad jednotkových testů třídy `BookingMapper`. Tady se používá dříve vysvětlený koncept mockování. Metoda `mockRepositories()`, se díky anotaci `@BeforeAll` spouští před všemi testy z této třídy a zajišťuje falešné chování úložiště, jež používá třída `BookingMapper`. Metoda `dtoToEntity()` ověřuje správnost převádění objektu `BookingDto` na objekt `Booking`.

5 ZÁVĚR

Cílem této práce bylo vybudovat a otestovat REST API, přičemž toto API mělo být zaměřeno na provoz letiště s možností správy příletů, odletů a rezervaci letenek. API navíc umožňuje uchování informací o cestujících a dalších objektech potřebných pro zmíněné účely. Vytvořená aplikace měla být pokryta jednotkovými a integračními testy. Také výsledná aplikace měla obsahovat webovou stránku s možností vyvolat a otestovat konkrétní endpointy.

Všech cílů bylo dosaženo především získáním aktuální informace o REST, jeho prvcích, alternativách a omezeních. API je implementováno v programovacím jazyce Java pomoci frameworku Spring.

Přestože vytvořené API lze plně využívat, jedná se pouze o prototypovou verzi, protože jeho chování je velmi omezené. API lze rozšířit přidáním dalších funkcionalit, například správu zaměstnanců letiště nebo uchovávání informací o hangárech. 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.

6 POUŽITÁ LITERATURA

(1) BIEHL, Matthias. *API Architecture: The Big Picture for Building APIs*. Kindle Edition, 2015. API-University Series #2. 192 s. ISBN 978-1508676645.

(2) DEWAILLY, Ludovic. *Building a RESTful Web Service with Spring*. Kindle Edition, 2015. 128 s. ISBN 978-1785285714.

(3) BALANI, Naveen. *Apache CXF Web Service Development*. Packt Publishing, 2009. 336 s. ISBN 978-1847195401.

(4) MASSE, Mark. *REST API Design Rulebook*. O'Reilly Media, 2011. 114 s. ISBN 978-1449310509.

(5) YELLAVULA, Naren. *Building RESTful Web services with Go: Learn how to build powerful RESTful APIs with Golang that scale gracefull*. Kindle Edition, 2017. 316 s. ISBN 978-1788294287.

(6) UZAYR, Sufyan. *Learning WordPress REST API*. Kindle Edition, 2016. 218 s. ISBN 978-1786469243.

(7) SUBRAMANIAN, Harihara. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*. Packt Publishing, 2019. 378 s. ISBN 978-1788992664.

(8) JAVA. *What is Java technology and why do I need it?* [online]. [cit. 2020-04-07]. Dostupné z: http://moodle.cz.https://java.com/en/download/faq/whatis_java.xml

(9) BURKE, Bill. *RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services*. 2. vydání. O'Reilly Media, 2013. 392 s. ISBN 978-1449361341.

(10) PATNI, Sanjay. *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS*. Apress, 2017. 148 s. ISBN 978-1484226643.

(11) COSMINA, Iuliana a kol. *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools*. 5. vydání. Apress, 2017. 849 s. ISBN 978-1484228074.

(12) PostgreSQL. *About* [online]. [cit. 2020-04-07]. Dostupné z: https://www.postgresql.org/about/

(13) SPRING. *Reference Documentation* [online]. [cit. 2020-04-07]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

(14) FOWLER, Martin. *Data Transfer Object* [online]. [cit. 2020-04-07]. Dostupné z: https://martinfowler.com/eaaCatalog/dataTransferObject.html

(15) Swagger. *Download Swagger UI* [online]. [cit. 2020-04-07]. Dostupné z: https://swagger.io/tools/swagger-ui/download/

(16) FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, 2000. Disertační práce. University of California.

PŘÍLOHY

PŘÍLOHA A – SPUŠTĚNÍ APLIKACE......48

Příloha A – Spuštění aplikace

Příloha popisuje postup spuštění aplikace.

Aplikace vyžaduje nainstalovanou databázi PostgreSQL, běžící na portu 5432, s vytvořeným uživatelem `airport`, a heslem `airport`. Databáze musí být taky pojmenovaná jménem `airport`.

Pro spuštěni samotné aplikace je potřeba v příkazovém řádku spustit skript `mvnw` s parametrem `spring-boot:run`. Skript se nachází v kořenovém adresáři projektu.

Pro spuštěni aplikace v testovacím režimu je nutné přidat navíc parametr `-Dspring-boot.run.profiles=dev`