Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Zoran Špec

RAZVOJ RESTFUL STORITEV NA GOOGLE APP ENGINE IN AMAZON WEB SERVICES

diplomsko delo

univerzitetni študijski program računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01920/2013
Datum: 02.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ZORAN ŠPEC**

Naslov: **RAZVOJ RESTFUL STORITEV NA GOOGLE APPENGINE IN AMAZON WEBSERVICES**

RESTFUL SERVICES DEVELOPMENT USING GOOGLE APPENGINE AND AMAZON WEBSERVICES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Analizirajte RESTful spletne storitve in jih primerjajte s SOAP storitvami v kontekstu storitvene usmerjenosti. Podrobno proučite in opišite RESTful storitveni model. Analizirajte koncepte PaaS (Platform-as-a-Service) in primerjajte med seboj ključne ponudnike PaaS, kot sta Google in Amazon. Izdelajte praktični primer RESTful storitev in pripadajoče aplikacije z uporabo Google AppEngine in Amazon WebServices ter jih primerjajte. Pri tem uporabite najboljše prakse razvoja.

Mentor:

prof. dr. Matjaž B. Jurič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Zoran Špec, z vpisno številko **63060296**, sem avtor diplomskega dela z naslovom:

RAZVOJ RESTFUL STORITEV NA GOOGLE APP ENGINE IN AMAZON WEB SERVICES

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 1. oktobra 2013 Podpis avtorja:

Na tem mestu bi se rad zahvalil mentorju za pomoč pri izdelavi diplomskega dela. Posebej pa bi se rad zahvalil svoji družini, ki mi je vsa leta študija stala ob strani in me podpirala. Zahvalil bi se tudi svojim prijateljem in sošolcem, ki so verjeli vame in me spodbujali.

Kazalo

Povzetek

V diplomskem delu obravnavamo RESTful spletne storitve in analiziramo koncept računalništva v oblaku, natančneje PaaS (Platform as a Service). Spoznamo tipične tehnologije in standarde, ki so potrebni za razvoj RESTful spletnih storitev. Podrobneje analiziramo in primerjamo najpogosteje uporabljena javanska ogrodja: Jersey, Restlet in RESTEasy. Podrobneje preučimo PaaS model, v okviru tega pa tudi primerjamo štiri ponudnike: Google App Engine, AWS Elastic Beanstalk, Windows Azure in Heroku. V nadaljevanju razvijemo dve aplikaciji na platformi Google App Engine in AWS Elastic Beanstalk ter jima dodamo funkcionalnosti, ki jih morajo imeti tipične RESTful storitve.

Ključne besede

RESTful spletne storitve, PaaS, JAX-RS ogrodja, Google App Engine, AWS Elastic Beanstalk.

Abstract

The thesis discusses RESTful web services and analyses platform as a service (PaaS) as a cloud computing concept. Typical technologies and standards, needed to develop RESTful web services are presented. Java frameworks such as Jersey, Restlet and RESTEasy are analysed in detail and compared. A detailed study of PaaS model is provided. Furthermore four providers of PaaS model are presented including Google App Engine, AWS Elastic Beanstalk, Windows Azure and Heroku. Finally we develop two applications on Google App Engine and AWS Elastic Beanstalk platforms, which we extend with functionalities that are usually included in typical a RESTful web service.

Key words

RESTful web services, PaaS, JAX-RS frameworks, Google App Engine, AWS Elastic Beanstalk.

Poglavje 1

Uvod

V zadnjem obdobju se je pojem računalništva v oblaku razširil v vse sloje družbe. Gre za koncept, ki ga IT javnost ne more in ne sme prezreti. Prinaša nov koncept aplikacij, ki se iz standardnih namiznih programov selijo v navidezne stroje, o katerih ne vemo praktično ničesar.
V tem delu bomo poskušali spoznati, kaj oblak je, kako deluje in predvsem zakaj ga izbrati ter kako uporabljati. Spoznali bomo tri modele, pri čemer si bomo podrobneje ogledali samo PaaS (Platform as a Service), ker je ta za razvoj spletnih aplikacij oziroma storitev najprimernejši model. S tem bomo prišli še do drugega ključnega koncepta tega dela - spletnih storitev. Storitve si najlažje predstavljamo kot funkcionalnosti, ki jih je razvila neka tretja oseba, mi pa bi jih radi koristili. Še več, pogosto bi radi te funkcionalnosti uporabili v svojih rešitvah in jih s tem dopolnili ter izboljšali. Sama integracija spletnih storitev niti ni tako enostavna. Obstajata dva večja tipa spletnih storitev. Mi bomo pozornost posvetili RESTful storitvam, ki so morda intuitivno lažje razumljive, saj so sestavljene iz tehnologij in protokolov, ki jih uporabljamo vsak dan, pa se morda tega niti ne zavedamo. Že navadna spletna stran predstavlja neke vrste RESTful storitev. Cilj diplomske naloge bo usmerjen na storitvene sisteme in njihovo delovanje. Spoznali bomo temeljne zakonitosti spletnih storitev, ki jih bomo kasneje podrobno pregledali še v okviru RESTful. Preleteli bomo osnovne tehnologije za izgradnjo RESTful storitev ter si pogledali ogrodja, ki nam lajšajo sam razvoj. V drugem delu se bomo posvetili pregledu računalništva v oblaku in njegovih temeljnih značilnosti. Podrobneje bomo opisali PaaS model oblaka, hkrati pa tudi spoznali glavne PaaS ponudnike na trgu. Zaradi omejenosti prostora bomo opisali samo bistvene značilnosti posameznih platform in nekaj najpomembnejših storitev, ki jih ponujajo. V zaključnem delu si bomo ogledali razvoj aplikacij na dveh v tem trenutku najbolj popularnih platformah, in sicer platformi Google App Engine in AWS Elastic Beanstalk. Še več, tem apli-

kacijam bomo dodali funkcionalnosti RESTful spletnih storitev, kjer bomo pokazali, kako lahko aplikacija na Google platformi uporablja storitev, ki jo nudi aplikacija na Amazon platformi in obratno. Aplikacije oziroma storitve bodo enostavne in namenjene zgolj razumevanju zgoraj opisanih konceptov, saj bomo implementirali samo ključne elemente, kot so avtentikacija, osnovna uporaba HTTP metod, upravljanje s statusnimi kodami itd.

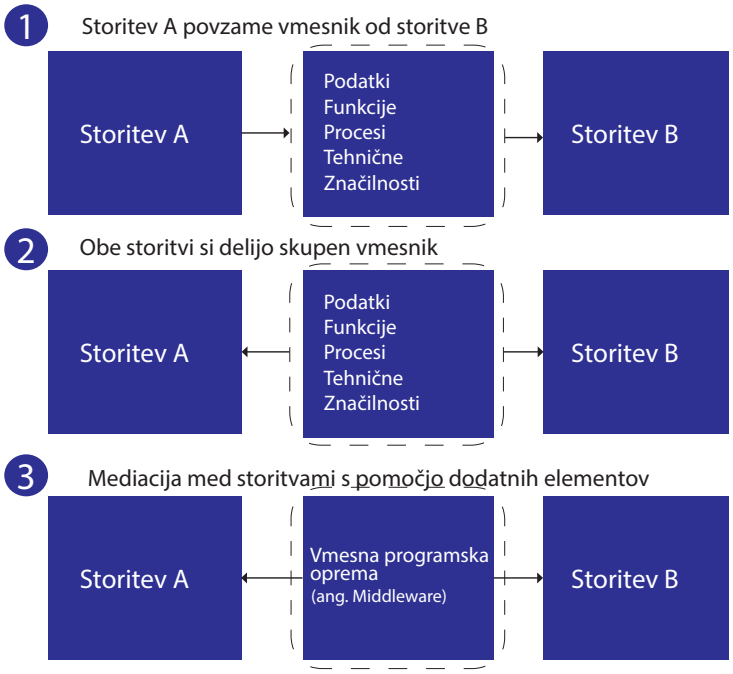
Poglavje 2

Usmerjenost sistemov

Pomembna koncepta v računalništvu sta **integracija** in **interoperabilnost** naprav, ki dajeta slutiti, da govorimo o ključnih elementih informacijskih tehnologij ter računalništva nasplošno.
Integracijo se tako najpogosteje opisuje kot proces povezovanja aplikacij in naprav v namen medsebojnega izmenjevanja funkcionalnosti, informacij ter podatkov. Interoperabilnost pa kot medsebojno razumevanje dveh ali več naprav oziroma aplikacij. Slika 2.1 prikazuje možne načine doseganja interoperabilnosti storitev: storitev povzame vmesnik druge storitve, storitve si lahko delijo skupen vmesnik, lahko pa v tak sistem vključimo še tretjo komponento, in sicer vmesno programsko opremo (ang. Middleware), ki skrbi za interoperabilnost.
Interoperabilnost in integracijo je možno doseči na različnih nivojih oziroma stopnjah razvoja. Na najnižji točki lahko govorimo o tehnološki stopnji (protokoli, nižjenivojska logika itd.). Naslednja stopnja je procesni nivo, kjer gre za splet procesov, ki omogočajo integracijo in interoperabilnost. Na najvišji stopnji pa najdemo informacijski nivo, kjer gre za izmenjavanje informacij in podatkov. Največ pozornosti bomo namenili slednjemu nivoju, ki bo tudi osrednja tema tega dela.

2.1 Podatkovna usmerjenost

Primer na sliki 2.2 prikazuje interoperabilnost in integracijo dveh podatkovno usmerjenih aplikacij [1]. Komunikacija v takšnem sistemu poteka na sledeči način. Aplikacijski strežnik dostopa do podatkovne baze. Dostop poteka neposredno preko ukazov za manipulacijo s podatkovno bazo, pri čemer je potrebno natančno poznavanje podatkovne baze in njenih omejitev. Pri tem se lahko zgodi, da aplikacija in podatkovna baza nista interoperabilna zaradi

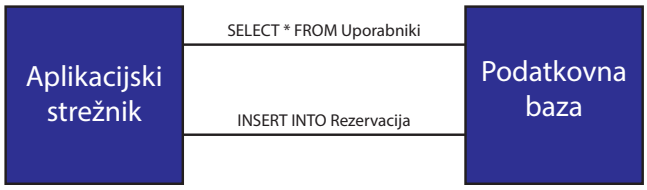


Slika 2.1: Rešitve za doseganje interoperabilnosti.

različnih razlogov (različne platforme, uporaba različnih tehnologij, zastareli standardi itd.). V ta namen je prišlo do razvoja storitveno usmerjenih sistemov.

2.2 Storitvena usmerjenost

Storitvena usmerjenost je paradigma, ki v ospredje postavlja storitve. Aplikacija, ki teče na strežniku, nudi storitve, ki jih ostale aplikacije oziroma odjemalci lahko koristijo. Implementacija samih storitev je ostalim nevidna, znan je le vmesnik za integracijo, ki mora biti natančno opisan. Problem takšne usmerjenosti je, da jo je težko vnaprej pravilno načrtovati in da je interoperabilnost precej omejena. Komunikacija v takšnem sistemu poteka na način, kot prikazuje slika 2.3. Aplikacija na strežniku A posreduje aplikaciji na strežniku B natančno opisano zahtevo z vsemi zahtevanimi parametri. Aplikacija na strežniku B prejme zahtevo, jo obdela in pošlje ustrezne rezultate nazaj aplikaciji na strežniku A. Tako smo se izognili problemu interoperabilnosti, ki se pojavi pri podatkovno usmerjenih sistemih, saj implementacija storitev navzven ni vidna. V tem delu bomo pozornost posvetili storitveni



Slika 2.2: Primer podatkovno usmerjenega sistema.

Slika 2.3: Primer storitveno usmerjenega sistema.

usmerjenosti sistemov.

2.2.1 SOA - Storitveno usmerjena arhitektura

Kratica SOA (Service Oriented Architecture) ali storitveno usmerjena arhitektura [2] je model, katerega glavni namen je večanje zmogljivosti, učinkovitosti ter prožnosti procesov znotraj podjetij. Glavno sredstvo za doseganje teh ciljev so storitve in storitveno usmerjen razvoj aplikacij. Ključni element SOA je modularizacija, saj dodajanje novih funkcionalnosti ne zahteva ponovnega razvoja sistemov. SOA implementacija lahko sestoji iz kombinacije tehnologij, produktov, API-jev, infrastrukturnih razširitev itd. Splošno sprejete definicije, ki bi opisala SOA, ni, čeprav so to poskušali mnogi. Med drugimi jo IBM [3] poskuša definirati kot:

> Način razvoja aplikacij, ki teži k čimvečjemu ujemanju poslovnim modelom. Za dosego tega cilja se SOA trudi v smeri zagotavljanja medsebojnega sodelovanja, koordiniranja in prilagajanja med poslovnim okoljem in IT subjekti. Pod to spada upravljanje nalog in procesov v okviru storitev, s katerimi je SOA podprta.

Open Group skupnost dodaja, da je SOA arhitekturni stil, ki podpira storitveno usmerjenost.

2.2.2 SOA principi

Za lažje razumevanje, razvoj in vzdrževanje storitveno usmerjenih arhitektur je potrebno slediti arhitekturnim principom. V nadaljevanju bomo opisali 8 principov, ki so povzeti po Thomasu Earlu [4]:

- **Abstrakcija** - značilnosti, ki zagotavlja nevidnost storitev navzven oziroma storitev predstavlja kot črno škatlo (ang. Black Box), kjer implementacija ni vidna odjemalcem.
- **Ponovna uporaba** (ang. Service Reusability) - osrednja značilnost storitveno usmerjenih arhitektur je ponovna uporaba že napisanih storitev. Za dosego te značilnosti morajo biti storitve neodvisne od tehnologij in poslovnih procesov.
- **Sestavljanje storitev** (ang. Service Composability) - sestavljanje storitev je tesno povezano z njihovo ponovno uporabo. Gre za sestavljanje storitev iz več neodvisnih delov (storitev).
- **Avtonomnost storitev** - avtonomnost storitev od izvajalnega okolja povečuje zanesljivost sistema. Zanesljivost je v največji meri odvisna ravno od kontrolne logike, ki upravlja z viri, ki so potrebni za delovanje (npr. skupna podatkovna baza).
- **Šibka sklopljenost** (ang. Loose Coupling) - princip govori, da je povezava (sklopljenost) med storitvami in odjemalcem šibka. To omogoča nadaljni razvoj tako storitve kot tudi aplikacij, ki to storitev uporabljajo, ne da bi se ob tem podrla razmerja med njimi.
- **Odkrivanje storitev** (ang. Service Discoverability) - gre za množico protokolov, ki omogočajo avtomatsko odkrivanje storitev v omrežju. Tipičen primer je UDDI (Universal Description Discovery and Integration).
- **Storitve brez stanja** (ang. Service Statelessness) - v primeru, da pri interakciji uporabljamo stanja, se lahko zgodi, da za novo zahtevo potrebujemo odgovor prejšnjih interakcij. S porazdeljenostjo arhitektur in večanjem števila zahtev postane upravljanje z informacijami zamudno.
- **Standardizirani formati** (ang. Standardized Service Contract) - zahteva po standardiziranem mediju, ki predstavlja sredstvo interakcije med storitvami. Kot primer lahko navedemo XML shemo.

Poglavje 3

Spletne storitve

3.1 Storitev

Beseda storitev predstavlja sredstvo za prinašanje vrednosti oziroma funkcionalnosti končnim uporabnikom, ne da bi si storitve lastili ali imeli z njimi stroške vzdrževanja ali tveganja. V računalniški terminologiji bi storitev lahko označili kot sistemski proces, ki teče neodvisno od programa.

Open Group skupnost [5] storitev opisuje kot:

- logično predstavitev ponavljajoče aktivnosti z določenim rezultatom,
- sestavljeno strukturo iz več delov (lahko tudi iz drugih storitev),
- črno škatlo, ki skriva implementacijo.

3.2 Spletna storitev

Za razumevanje tega dela je nujno definirati, kaj spletna storitev je in kaj predstavlja. Gre za kompleksen koncept, ki je težko opisljiv in nima enotne definicije [6].

- Spletna storitev je lahko del programske opreme, ki je na voljo v vsakem trenutku in uporablja standardiziran XML sistem sporočil. XML je standardizirana tehnologija, ki kodira podatke v sporočila za lažjo komunikacijo.
- Spletne storitve so samostojne, modularne, distribuirane in dinamične aplikacije, ki so opisljive in jih je mogoče odkriti v omrežju ter jih izvršiti. Njihov namen je pridobitev podatkov oziroma procesov. Te aplikacije so lahko lokalne, distribuirane ali pa na voljo preko spleta.

Spletne storitve temeljijo na odprtih standardih, kot so TCP/IP, HTTP, Java, HTML in XML ter drugih tehnologijah.

- Spletne storitve so sistemi, ki temeljijo na standardizirani XML izmenjavi podatkov in ki uporabljajo internet za neposredno interakcijo med aplikacijami. Ti sistemi so lahko grajeni iz aplikacij, objektov, sporočil ali dokumentov.
- Spletna storitev je zbirka odprtih protokolov in standardov, ki se uporabljajo za izmenjavo podatkov med aplikacijami ali sistemi. Programske aplikacije, ki so napisane v različnih programskih jezikih in delujejo na različnih platformah, jih lahko uporabljajo spletne storitve za izmenjavo podatkov preko omrežja.

3.2.1 Uradna definicija spletne storitve

Uveljavljena je definicija, ki jo navaja W3C (World Wide Web Consortium) [7]:

> Spletna storitev je abstrakten sistem, ki podpira medsebojno interakcijo in komunikacijo naprav preko omrežja. Vsebuje vmesnik, ki je opisan v računalniško opisljivem jeziku (natančneje WSDL). Sistemi (naprave) med seboj komunicirajo preko spletnih storitev s SOAP sporočili, tipično preko HTTP protokola, kjer si izmenjujejo XML datoteke.

Spletna storitev je tako abstraktna množica funkcionalnosti, ki jih mora implementirati konkreten agent (programska ali strojna oprema).

Spletna storitev je torej vsaka storitev, ki:

- je na voljo preko interneta ali zasebnega omrežja (intranet),
- uporablja standardizirano izmenjavo sporočil,
- je neodvisna od operacijskega sistema in programskega jezika,
- jo je enostavno najti preko iskalnega mehanizma.

Spletne storitve delimo v 2 veliki skupini: RESTful spletne storitve in SOAP spletne storitve (poznane tudi kot "Big Web Services").

3.3 SOAP spletne storitve

SOAP (Simple Object Access Protocol) [8] je protokol, ki je namenjen izmenjavi strukturiranih informacij v porazdeljenih sistemih. Izmenjava sporočil

poteka z uporabo XML tehnologije preko različnih aplikacijskih protokolov. Glavni cilj SOAP protokola je enostavnost in razširljivost.

SOAP je definiran z naslednjim ogrodjem (ang. Framework):

- **SOAP procesni model** (ang. SOAP Processing Model) - tu se nahaja definicija pravil za obdelavo SOAP sporočil.
- **SOAP razširljivostni model** (ang. SOAP Extensibility Model) - tu se nahaja definicija SOAP značilnosti in definicija modulov.
- **SOAP aplikacijsko povezovalno ogrodje** (ang. Underlying Protocol Binding Framework) - opis in definicija pravil za povezovanje aplikacijskih protokolov in izmenjavo SOAP sporočil med vozlišči.
- **SOAP sporočilo** (ang. SOAP Message) - definicija zgradbe SOAP sporočila.

3.3.1 Zgradba SOAP sporočila

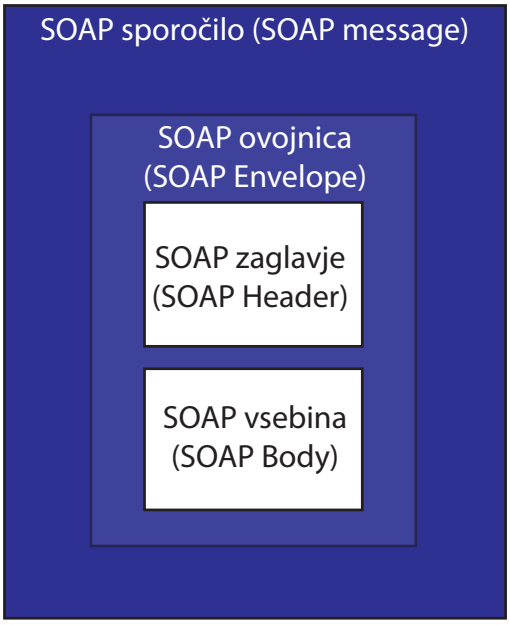
SOAP sporočilo je XML dokument, ki je sestavljen iz elementov prikazanih na sliki 3.1.

- **Ovojnica** (ang. Envelope) - definira začetek in konec sporočila.
- **Zaglavje** (ang. Header) - vsebuje atribute (neobvezno), ki so namenjeni za procesiranje sporočila.
- **Telo** (ang. Body) - vsebuje podatke.
- **Napaka** - vsebuje informacijo o napakah, ki so se zgodile tekom procesiranja sporočila.

3.3.2 Prednosti SOAP spletnih storitev

Odločitev, zakaj izbrati SOAP spletne storitve, ni enostavna, smo pa zbrali nekaj najpogosteje navajanih prednosti [9].

- **Heterogeno okolje** - SOAP lahko deluje na poljubni platformi oziroma na poljubnem operacijskem sistemu. Implementiran je lahko v poljubnem programskem jeziku preko različnih protokolov.
- **XML osnova** - ker je zastavljen na XML tehnologiji, omogoča dobro strukturiranost, človeško berljivost in enostavno obdelavo.



Slika 3.1: Zgradba SOAP sporočila.

- **Transportno neodvisen** - SOAP je sporočilni protokol in lahko poteka preko katerega koli transportnega protokola (HTTP, SMTP).
- **Paketi v obliki čistega teksta** (ang. Plain Text Packages) - SOAP sporočila pošiljamo v obliki čistega teksta z metapodatki v zaglavju (preko Content-type atributa). S tem povečamo transparentnost sporočil, saj dovolimo varnostnim mehanizmom (npr. požarnemu zidu), da preverijo varnost in veljavnost paketkov.
- **Atribut mustUnderstand** - globalni *mustUnderstand* atribut preprečuje ponovno pošiljanje SOAP sporočil, če niso bila razumljiva. Namesto tega se pošlje sporočilo o napaki.

Poglavje 4

REST spletne storitve

REST ali REpresentational State Transfer je prvi opisal Roy Fielding v svoji doktorski disertaciji, ki je bila objavljena leta 2000 [10].
V njej navaja, da REST spletna storitev ni arhitektura temveč arhitekturni stil. Je množica arhitekturnih omejitev, ki jasno določajo funkcije in vloge podatkov, komponent, hiperpovezav, komunikacijskih protokolov in podatkovnih odjemalcev.

Ključni cilji RESTful storitev so [11]:

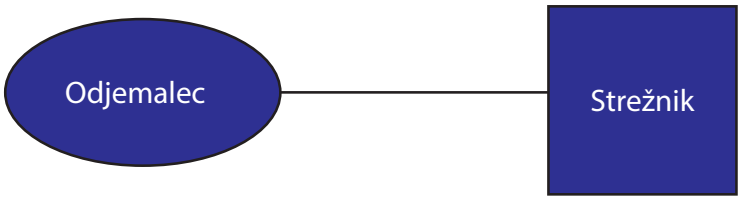
- skalabilnost,
- posplošenost vmesnika,
- neodvisen razvoj in vzpostavitev komponent,
- dodajanje komponent, ki zmanjšujejo latenco, povečujejo varnost in enkapsulirajo že implementirane rešitve.

4.1 REST omejitve

R. Fielding v svoji disertaciji navaja, da REST spletne storitve definirajo naslednje arhitekturne omejitve (ang. Constraints) [10]:

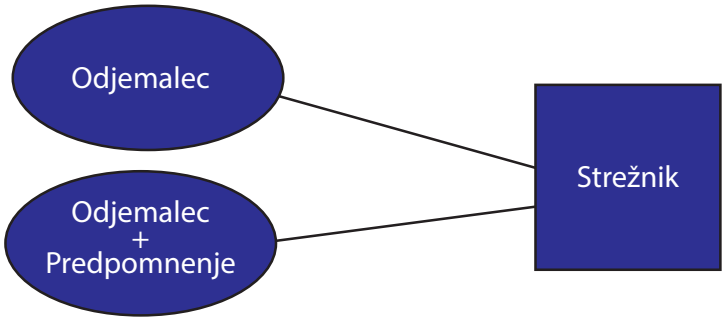
- **Strežnik - odjemalec** - kot je prikazano na sliki 4.1 sta strežnik in odjemalec ločeni strukturi. Imata ločen uporabniški vmesnik in ločene podatkovne baze, kar omogoča večjo prenosljivost, skalabilnost ter neodvisen razvoj posameznih komponent.

- **Predpomnenje** (ang. Caching) - slika 4.2 prikazuje odjemalca, ki hrani odgovor strežnika (ang. Server Response) v predpomnilnik za kasnejšo uporabo. Potrebno je opredeliti, kdaj si naj odjemalec zapomni



Slika 4.1: Model strežnik - odjemalec.

nove podatke. S tem povečamo učinkovitost in zmanjšamo interakcijo med strežnikom ter odjemalcem.



Slika 4.2: Odjemalec s predpomnenjem in strežnik brez stanja.

- **Sistem brez stanja** (ang. Statelessness) - med prenosi zahtev strežnik ne hrani podatkov o odjemalcu. Vsaka HTTP zahteva se zgodi v popolni izolaciji. Seja se lahko shrani na strani odjemalca. Vsaka zahteva, poslana strežniku, mora vsebovati vse potrebne informacije za razumevanje in izvedbo operacije. S tem izboljšamo zanesljivost, saj je lažje obnoviti stanje ob morebitnih napakah. Omogočimo lahko tudi lažje izenačevanje bremena (ang. Load Balancing), saj se lahko zahteve neodvisno obdelujejo na različnih strežnikih.
- **Enoten vmesnik** - namenjen je poenostavitvi in neodvisnosti arhitekture spletnih storitev. Sledi naslednjim konceptom:
  - **Sredstva** (ang. Resources) - so konceptualna preslikava v množico entitet. Vsaka informacija je lahko sredstvo, ki je odvisno od konteksta. Namenjena so izmenjavi med odjemalcem in strežnikom. Vsebujejo lahko metapodatke.

  - **Razločevanje sredstev** - vsako sredstvo ima enolično določen naslov URI (Uniform Resource Identifier). Ti naslovi se navadno ne spreminjajo. URI naslov ne sporoča ničesar drugega kot lokacijo, kjer se sredstvo nahaja.
  - **Predstavitev sredstev** (ang. Resource Representation) - predstavitev je trenutno ali želeno stanje sredstev. Vsako sredstvo ima lahko več predstavitev. Običajno jih najdemo v formatih HTML, XML, JSON. Redkeje pa tudi v ostalih podatkovnih tipih (png, jpg, pdf ...).
  - **Samo opisljiva sporočila** - vsaka zahteva in odgovor vsebujeta svoj opis sheme. Opisani so z XML shemo in imenskim XML prostorom.
  - **Hypermedia kot izbiralec stanj aplikacije (HATEOAS)** - HATEOAS je logična razširitev hiperteksta, kjer se prepletajo audio, video, tekstovni in grafični tokovi, da se ustvari nelinearni tok informacij.

- **Večplasten sistem** (ang. Layered System) - sistem mora biti sestavljen iz plasti, ker želimo, da so posamezne komponente na plasteh nevidne ostalim. S tem omogočimo neodvisen razvoj in nadgradnjo obstoječih storitev.

- **Koda na zahtevo** (ang. Code on Demand) - funkcionalnost odjemalca lahko razširimo tako, da naložimo in izvajamo kodo v majhnih programih (ang. Applets) ali skriptah.

4.2 URI

Ko govorimo o RESTful storitvah, se ne moremo izogniti standardu URI (Uniform Resource Identifier), ki predstavlja hiperpovezavo do sredstev. URI določa, kako intuitivna bo REST spletna storitev in ali bo uporabljena tako, kot je bila zastavljena. V ta namen lahko UR naslov sestavimo v obliki hierarhične strukture (npr. drevesna struktura). Gre za znakovni niz, ki identificira sredstvo. Opredelimo ga kot:

- **Lokatorje** (ang. URL - Unifrom Resource Locator) - način, kako najti sredstvo.
- **Imena** (ang. URN - Uniform Resource Name) - definicija identitete sredstva.

Izvorna koda 4.1: Prikaz zgradbe URI naslova.

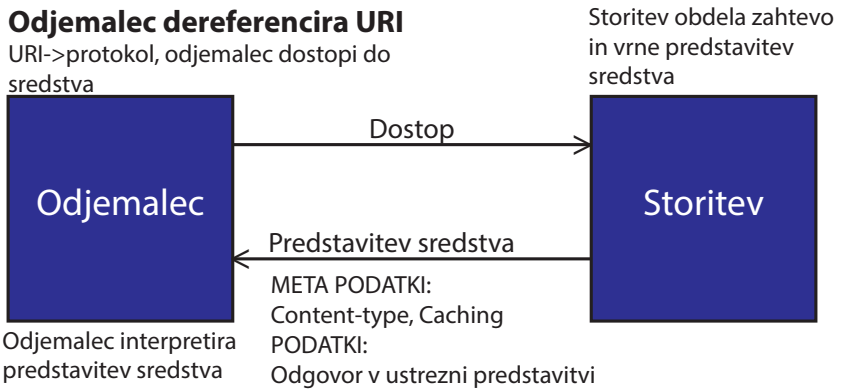
```
scheme://host/path/query#fragment
```

Element	Primer	Pomen
scheme	http	*Shema (ang. Scheme)* predstavlja protokol, ki se uporablja za komunikacijo s sredstvi. To je ponavadi http ali https.
host	www.domain.com	*Gostitelj (ang. Host)* določi strežnik, na katerem se nahaja sredstvo. Host je lahko del URI naslova, lahko pa je izpuščen. Če se izpusti, URI vsebuje le relativne podatke, ki jih je mogoče interpetirati le v absolutnem kontekstu.
path	/ime/resource	*Pot (ang. Path)* določa lokacijo sredstva na strežniku.
query	resource?date=10-09&time=15-00	*Poizvedba (ang. Query)* je navodilo strežniku za uporabo dodatnih parametrov pri izstavljanju zahtev ali pa navodilo, v kakšni obliki naj vrne predstavitev sredstva.
fragment	bookmark	*Fragment* se ne podaja strežniku in se nahaja samo na strani odjemalca.

Tabela 4.1: Razlaga elementov URI naslova.

4.2.1 URI dereferenciranje

URI dereferenciranje [13] je proces pridobivanja protokola iz URI-ja in sestavljanje zahteve. Proces dereferenciranja URL naslova je shematično prikazan na sliki 4.1.



Slika 4.3: Dereferenciranje URI naslova.

4.2.2 Verzije storitev

Dobra praksa razvoja spletnih storitev je vzdrževanje verzij. Kljub temu da se spletne storitve lahko nadgrajujejo, je potrebno vzdrževati povezljivost z odjemalci. V ta namen se lahko poslužujemo verzioniranja (ang. Versioning) preko URI naslovov. V idealnem primeru bi obdržali stare verzije storitev, dokler jih uporablja vsaj en odjemalec. Realno jih obdržimo, dokler arhitekturne spremembe to dopuščajo oziroma dokler stroški vzdrževanja ne presegajo stroškov prehoda odjemalcev na novo storitev. Primer verzij preko URL naslova kaže izvorna koda 4.2.

Izvorna koda 4.2: Primer različnih verzij za dostop do istega sredstva.

```
myapplication.com/V1/resource1
myapplicaiton.com/V2/resource1
```

4.3 Predstavitev sredstev

Predstavitev sredstva reflektira stanje sredstva in njegovih atributov v času, ko ga odjemalec zahteva. Format, v katerem so sredstva predstavljena, je

bistvenega pomena za izmenjavo podatkov med odjemalcem in strežnikom v HTTP telesu (ang. HTTP Body).

4.3.1 WADL

WADL (Web Application Description Language) je primeren za opisovanje spletnih aplikacij, ki uporabljajo protokol HTTP in XML sporočila. Te aplikacije so tipične za spletne storitve REST. WADL ni standardiziran s strani W3C, zato tudi ni tako pogosto uporabljen. Kot kaže izsek izvorne kode 4.3, WADL za opis storitev uporablja *resource* elemente v značkah (ang. Tag). Vsak *resource* element vsebuje parametre, ki so namenjeni označevanju vhodnih podatkov, in metode, ki opisujejo HTTP zahtevo ter odgovor.

Izvorna koda 4.3: Primer WADL strukture.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
        <resources base="http://example.com/api">
                <resource path="books">
                        <method name="GET"/>
                        <method name="POST"/>
                </resource>
        </resources>
</application>
```

4.3.2 XML

XML (Extensible Markup Language) [14] je označevalni jezik, ki definira pravila kodiranja dokumentov v format, ki je berljiv tako stroju kot tudi človeku. Definiran je v XML 1.0 specifikaciji W3C organizacije. Cilj XML jezika je poenostaviti in posplošiti predstavitev podatkov preko omrežja. Je znakovni format (glej izvorno kodo 4.4), v katerem se lahko pojavi vsak unicode znak. Podatek je opisan tako, da ga obdamo z obeh strani z oznako (ang. Tag). Oznake so prilagodljive in niso vnaprej določene. Dajejo nam informacijo o podatkih in strukturi dokumenta.

Izvorna koda 4.4: Primer XML dokumenta.

```
<uporabnik>
        <ime>Zoran</ime>
        <priimek>Spec</priimek>
        <Starost>50</starost>
        <kraj>Lasko</kraj>
```

```
        <fakulteta>FRI</fakulteta>
</uporabnik>
```

4.3.3 JSON

JSON (JavaScript Object Notation) [16] je znakovni format (glej izvorno kodo 4.5) za izmenjavo podatkov in je, podobno kot XML, berljiv tako stroju kot človeku. JSON temelji na skriptnem programskem jeziku javascript, kjer so podatki predstavljeni kot objekti.

Izvorna koda 4.5: Primer JSON dokumenta.

```
{"uporabnik":
        {
        "ime": "Zoran",
        "priimek": "Spec",
        "starost": "50",
        "kraj": "Lasko",
        "fakulteta": "FRI";
        }
}
```

4.3.4 Tipi internetnih medijev

Tip internetnih medijev (ang. Internet Media Type) [17] je dvodelni tekstovni identifikator podatkovnih formatov na internetu. Izhaja iz MIME tipov (Multipurpose Internet Mail Extension Types), prvotno definiranih za definicijo ne-ASCII delov elektronskih sporočil.

Kot prikazuje izsek iz kode 4.6 je internetni media tip sestavljen iz **tipa** (ang. Type) in **podtipa** (ang. Subtype) ter nič ali več parametrov.

Izvorna koda 4.6: Primer tipa internetnega medija.

```
type/subtype; parameter=vrednost
text/html; charset=UTF-8
```

Najpogostejši tipi so **application** (za večnamenske podatke), **audio** (za podatke v obliki zvokovnega formata), **image** (za podatke v obliki slikovnega), **model** (za 3D modele), **message**, **text** (za berljiv tekst in izvorno kodo), **video** (za podatke v obliki video formata).

4.3.5 Pogajanje o predstavitvi resourcev

Pogajanje o predstavitvi sredstev (ang. Content Negotiation) je mehanizem, ki je potreben, kadar lahko za podani URI naslov uporabimo različne predstavitve sredstev. Kot bomo videli v nadaljevanju, lahko v glavi zahteve s parametrom *Accept* določimo, v kakšni obliki želimo prejeti vsebino. Enakovredno lahko v glavi odgovora s parametrom *Media-type* napovemo, v kakšnem formatu pošiljamo vsebino.

4.4 HTTP protokol

Pravzaprav je bil REST koncept sprva opisan v kontekstu **HTTP protokola**, vendar z njim ni omejen. Je pa to protokol, ki si ga bomo zaradi tesne povezanosti z glavno tematiko tega dela pogledali podrobneje.
HTTP [20] je komunikacijski protokol med odjemalci in strežniki, ki deluje po principu zahteva-odgovor. **HTTP odjemalec** (tipično spletni brskalnik) prične z zahtevo, tako da vzpostavi TCP povezavo na oddaljenem gostitelju (strežniku). Strežnik obdela zahtevo in vrne odgovor odjemalcu. Odgovor lahko vsebuje informacije o zahtevi in vsebino.

4.4.1 HTTP zahteva

Je zbirka informacij, ki je poslana strežniku. Ključni elementi HTTP zahteve so prikazani v izvorni kodi 4.7. Sestavljajo jo:

Izvorna koda 4.7: Primer HTTP zahteve.

```
GET http://myexample.com/ HTTP/1.0
Accept: text/html
If-Modified-since: Saturday, 15-January-2000 14:37:11 GMT
User-Agent: Mozilla/4.0
```

- **Vrstica zahteve** (ang. Request Line) - sestavljena je iz:
  - metode, ki jo bo zahteva izvedla,
  - URI naslova, kjer se bo zahteva izvedla,
  - verzija protokola, ki ga uporablja odjemalec.
- **Neobvezni parametri**

  - Accept
  - Accept-Charset
  - Accept-Encoding
  - Accept-Language
  - Authorization
  - Expect
  - From
  - Host
  - If-Match
  - If-Modified-Since
  - If-None-Match
  - If-Range
  - If-Unmodified-Since
  - Max-Forwards
  - Proxy-Authorization
  - Range
  - Referer
  - User-Agent

- **Vsebina zahteve** - gre za predstavitev sredstva v ustreznem formatu, ki jo pošljemo kot del zahteve (pri uporabi HTTP PUT in HTTP POST metod).

4.4.2 HTTP odgovor

HTTP odgovor (ang. HTTP Response) je zbirka informacij, ki je poslana odjemalcu. Ključni elementi HTTP odgovora so prikazani v izvorni kodi 4.8. Sestavljajo ga:

Izvorna koda 4.8: Primer HTTP odgovora.

```
HTTP/1.0 200 OK
Date: Sat, 15 Jan 2000 14:37:12 GMT
Server: Microsoft-IIS/2.0
Content-Type: text/HTML
Content-Length: 1345
If-Modified-since: Fri, 14 Jan2000 08:37:11 GMT
```

- **Statusna vrstica** - sestavljena je iz:
  - verzije uporabljenega protokola,
  - statusne kode,
  - pomena statusne kode.
- **Neobvezni parametri**

  - Content-Encoding
  - Content-Language
  - Content-Length
  - Content-Type
  - Date
  - Expires
  - Forwarded
  - Location
  - Server

- **Vsebina odgovora** - gre za predstavitev sredstev v ustreznem formatu, ki jo pošljemo kot del odgovora.

4.5 HTTP metode

Kot je bilo že omenjeno, izmenjava predstavitev sredstev poteka preko HTTP protokola. Pri REST spletnih storitvah se želimo izogniti dvoumnosti dizajna in implementacije, zato nad sredstvi izvajamo CRUD operacije (Create, Retrieve, Update, Delete), ki imajo svoje ekvivalente v HTTP metodah [21]:

- create (HTTP POST),
- retrieve (HTTP GET),
- update (HTTP PUT),
- delete (HTTP DELETE).

4.5.1 Lastnosti HTTP metod

4.5.1.1 Varne metode

Razvijalci spletnih aplikacij in storitev se morajo zavedati, da lahko uporabnikove interakcije preko spleta spreminjajo obnašanje elementov in sredstev. V izogib konfliktov je bil sprejet dogovor, da GET metoda ne sme spreminjati sredstev, lahko pa jih kliče iz podatkovnih skladišč. Zato takšnim metodam pravimo **varne metode** [21]. Nasprotno pa metode PUT, POST, DELETE spreminjajo sredstva oziroma njihovo stanje in zato te metode niso varne. Ni odveč omeniti, da varnost GET metode ni moč popolnoma zagotoviti zaradi možnosti napak strežnika.

4.5.1.2 Idempotentne metode

Še ena pomembna lastnost metod je idempotentnost [21]. Idempotentne metode lahko neodvisno ponavljamo n-krat in pri tem vedno dobimo isti odgovor. Poudarimo lahko, da ni nujno, da je odziv enak. Pomemben je končni učinek na sredstvu. Idempotente metode so GET, PUT, DELETE, OPTIONS, medtem ko POST ni idempotentna metoda.

Primeri obnašanja metod:

- Če na URL naslovu izvršimo GET metodo, pridobimo predstavitev sredstva, ki se je tam nahajal. Pri večkatnem klicanju istega naslova dobimo vedno isto predstavitev sredstva.

- Z DELETE metodo brišemo sredstvo. Če jo kličemo večkrat, je sredstvo še vedno izbrisano, le odziv strežnika je različen (404 – Not Found, 200 – OK, 204 – No Content).

- Metodo POST najpogosteje uporabljamo za ustvarjanje novih sredstev. Pošiljanje iste zahteve večkrat bo ustvarilo več istih sredstev.

4.5.2 HTTP GET

GET metoda je namenjena pridobivanju informacij (v obliki entitet), ki smo jih zahtevali preko URI naslova. Pomembno je poudariti, da je GET operacija **varna**. To pomeni, da jo je mogoče večkrat izvesti, ne da bi spremenili stanje sredstva. Ta lastnost je pomembna zaradi več razlogov. Eden pomembnejših je indeksiranje vsebin, saj ne želimo, da se sredstvo ob izvedbi operacije spremeni. Pomembna je tudi pri predpomnenju (ang. Caching) rezultatov operacij, saj tako pospešimo celoten proces pri dostopanju istega sredstva.

4.5.3 HTTP POST

POST metodo uporabljamo za kreiranje novega sredstva. URI za kreiranje novega študenta v seznam je http://restfuljava.com/studenti/luka. Potek procesa kreiranja predstavitve sredstva preko HTTP POST metode:

1. Odjemalec pošlje HTTP zahtevo s POST metodo na URI naslov http://restfuljava.com/studenti/luka.

2. Zahteva ima dodano tudi XML predstavitev sredstva študent z imenom Luka.

3. Strežnik sprejme zahtevo, kjer se izvršijo potrebne operacije za shranjevanje sredstva.

4. Ko se sredstvo shrani, strežnik pošlje odgovor nazaj odjemalcu z ustrezno numerično kodo.

4.5.4 HTTP PUT

PUT metodo uporabljamo za posodobitev sredstev. Za izvedbo operacije posodobitve najprej potrebujemo predstavitev sredstva s strani odjemalca, ki jo pošljemo preko HTTP PUT metode do strežnika. Predpostavimo, da želimo v seznamu študentov študentu z imenom Janez spremeniti starost. Potek procesa posodobitve sredstva:

1. Odjemalec pošlje PUT zahtevo na URI http://restfuljava.com/studenti/janez z ustreznim posodobljenim XML sredstvom.
2. Strežnik obdela zahtevo in posodobi bazo.
3. Strežnik pošlje odgovor z ustrezno numerično kodo.

4.5.5 HTTP DELETE

DELETE metoda je namenjena brisanju sredstva. Če želimo brisati sredstvo, pošljemo zahtevo preko HTTP DELETE metode na ustrezni URI naslov npr. http://restfuljava.com/studenti/tina. Potek procesa brisanja sredstva:

1. Odjemalec pošlje DELETE zahtevo na URI http://restfuljava.com/studenti/tina.
2. Strežnik obdela zahtevo in izbriše sredstvo z imenom Tina.
3. Strežnik pošlje odgovor z ustrezno numerično kodo.

4.5.6 HTTP HEAD in HTTP OPTIONS

S HTTP HEAD metodo pošljemo zahtevo po metapodatkih sporočila. Deluje enako kot HTTP GET, le da ne vrača celotne predstavitve sredstva.
Z OPTIONS metodo preverjamo, katere HTTP metode sredstvo podpira. Odgovor na OPTIONS zahtevo vsebuje HTTP *Allow header* z naštetimi metodami: npr. *Allow GET* ali *Allow PUT*.

4.6 Statusne kode

Pomemben REST koncept so tudi statusne kode (ang. Status Code), kjer strežnik v odgovoru pošlje numerično vrednost, ki sovpada z dogajanjem aplikacije. Numerične kode delimo na 5 sklopov:

- informativna obvestila: 1xx,
- obvestila o uspešnosti: 2xx,

- obvestila o dodatnih možnostih: 3xx,
- obvestila o napakah na odjemalcu: 4xx,
- obvestila o napakah na strežniku: 5xx.

Nekaj najpogosteših statusnih kod:

- 200 (OK) - sredstvo je bilo posodobljeno.
- 201 (Created) - sredstvo je bilo ustvarjeno.
- 202 (Accepted) - sredstvo je bilo sprejeto za obdelavo, vendar obdelava še ni končana.
- 400 (Bad Request) - nepravilna zahteva (navadno gre za napačno navajanje parametrov).
- 404 (Not Found) - sredstvo ne obstaja.
- 406 (Not Acceptable) - strežnik ne podpira zahtevane predstavitve sredstva.
- 415 (Unsupported Media Type) - sprejeta predstavitev sredstva ni podprta.
- 500 (Generic Error Message) - splošno sporočilo o napaki, kjer ne moremo natančneje opisati izvora napake.

Poglavje 5

Programska ogrodja

V računalništvu programsko ogrodje (ang. Software Framework) [23] predstavlja abstrakcijo generičnih funkcionalnosti za razvoj aplikacij. Vključuje podporne programe, prevajalnike, knjižnice, orodja in programske vmesnike (ang. API) ter jih združuje v celoto za lažji razvoj rešitev. Od navadnih programskih knjižnic jih loči:

- **Inverzija kontrole** (ang. Inversion of Control) - za razliko od knjižnic kontrolo nad tokom programa izvaja programsko ogrodje in ne uporabnik.
- **Privzeto obnašanje** - vsako ogrodje ima privzeto obnašanje.
- **Razširljivost** (ang. Extensibility) - razvijalci lahko dodajajo funkcionalnosti.
- **Nespremenljivost kode** - programska koda ogrodja je v osnovi nespremenljiva. Razvijalci lahko dodajo funkcionalnosti, vendar ne smejo spreminjati obstoječe kode.

Programska ogrodja pogosto diktirajo strukturo naših aplikacij. Nekatera nudijo toliko kode, da razvoj lastne skoraj ni potreben. Največja prednost programskih ogrodij je krajši čas razvoja aplikacij, saj razvijalcem omogoča, da se posvetijo razvoju same aplikacije in zahtevanim funkcionalnostim, ogrodje pa poskrbi za nižje nivoje (priprava okolja, orodij). Ima pa programsko ogrodje tudi slabost in ta je, da je težko razumljiv koncept, ki se ga je potrebno naučiti in razumeti. Še več, programska ogrodja ne ustrezajo v celoti vsem tipom aplikacij, zato jih je potrebno razširiti in dodati svoje lastne funkcionalnosti. V nadaljevanju si bomo ogledali tipe programskih ogrodij. Podrobneje bomo spoznali tudi JAX-RS in tri njegove implementacije.

5.1 Tipi programskih ogrodij

Spletno aplikacijsko ogrodje ali WAF (Web Aplication Framework) [24] je univerzalna programska platforma, ki je namenjena razvoju dinamičnih spletnih strani, spletnih aplikacij in spletnih storitev. Navadno vključuje podporne programe, prevajalnike, programske knjižnice, API-je in ostala orodja, ki omogočajo lažji razvoj oziroma implementacijo. Ključen element spletnih vmesnikov so programske knjižnice, ki služijo kot centralni repozitorij funkcij in metod za upravljanje baz podatkov ter za zagotavljanje varnosti. Ločimo naslednje tipe spletnih ogrodij:

- **MVC (Model View Controler)** - gre za ločitev podatkovnega modela od uporabniškega vmesnika. Prednost takšnega vmesnika je modularizacija programske kode, ki omogoča ponovno uporabo. Posledično lahko takšno kodo uporablja več uporabniških vmesnikov. Poznamo "Push based"vmesnike, ki uporabljajo akcije za obdelavo podatkov in jih nato prikažejo (push action). Druga vrsta so "Pull based"vmesniki, ki priskrbijo prikazane rezultate za nadaljnjo obdelavo.

- **Three Tier** - gre za ločitev odjemalca, aplikacije in podatkovne baze na tri dele. Aplikacija vsebuje poslovno logiko, teče na strežniku in z odjemalcem komunicira preko HTTP protokola. Odjemalec je navadno spletni brskalnik, ki prikazuje HTML elemente na zaslon.

- **CMS (Content Managing Systems)** - gre za organiziranje in kategoriziranje informacij za njihovo lažje upravljanje. Namenjen je zbiranju, upravljanju, objavljanju in shranjevanju vsebin, hkrati pa vzdržuje dinamične povezave med njimi.

5.2 Aplikacijski programski vmesnik

Aplikacijski programski vmesnik (API) določa, kakšna je medsebojna interakcija programskih komponent. Poleg dostopa do baz podatkov oziroma računalniške strojne opreme (trdi diski, grafične kartice) lahko API uporabljamo za enostavnejše programiranje grafičnih komponent uporabniškega vmesnika. V praksi je API knjižnica, ki vključuje rutinska opravila, podatkovne strukture, objektne razrede in spremenljivke.
Programsko ogrodje lahko razumemo kot implementacijo več knjižnic oziroma več API vmesnikov. Poudarimo lahko še enkrat, da za razliko od API vmesnikov, v primeru programskega ogrodja, kontrolo nad tokom programa izvaja ogrodje samo.

5.2.1 Spletni API

V okviru razvoja spletnih aplikacij spletni API tipično opredeljujemo kot HTTP zahteve, ki jim dodajamo strukturirana sporočila (XML, JSON). V preteklosti je bil spletni API sinonim za spletne storitve, z razvojem Web 2.0 pa se ta termin vse bolj nanaša na RESTful spletne storitve in ROA arhitekturo (ang. Resource Oriented Architecture).
Spletni API lahko razumemo iz dveh vidikov. Z vidika strežnika API predstavlja programski vmesnik za sistem tipa zahteva-odgovor, kjer so dostopne standardne HTTP metode. Z vidika odjemalca API predstavlja vmesnik, ki deluje znotraj brskalnika.

5.3 Nivoji API vmesnikov

Ko govorimo o API vmesnikih, obstajajo štirje različni nivoji [26]. Vsak nivo od razvijalca zahteva pozornost na različne elemente.

- **Prenosni nivo** (ang. The Wire) - na tej stopnji razvijalec piše neposredno v prenosni format zahteve. V primeru RESTful spletnih storitev razvijalec ustvari HTTP zahtevo z ustrezno glavo, vključno z vsebino (ang. Payload), ter odpre HTTP povezavo. RESTful spletna storitev nato odgovori z ustrezno vsebino in pripadajočo statusno kodo. V primeru SOAP spletnih storitev pa razvijalec ustvari SOAP ovojnico, doda ustrezno SOAP zaglavje in vsebino. SOAP spletna storitev se odzove z ustreznim odgovorom v SOAP ovojnici. Delo s SOAP sporočili zahteva razčlenjevanje (ang. Parsing) XML vsebine, za kar navadno skrbijo višje nivojski API vmesniki.
- **Nivo jezikovno specifičnega nabora orodij** (ang. Language Specific Toolkit) - na tem nivoju razvijalci RESTful in SOAP spletnih storitev za delo z obliko in strukturo podatkov uporabljajo nabor razpoložljivih orodij.
- **Nivo storitveno specifičnega nabora orodij** (ang. Service Specific Toolkit) - razvijalci uporabljajo naprednejše nabore orodij za delo s storitvami. Na tem nivoju se osredotočamo na poslovne objekte in poslovne procese. Nivo temelji na paradigmi, da smo precej bolj učinkoviti, če se ukvarjamo z vsebinami, ki so pomembne in ne z nižje ležečimi protokoli.
- **Nivo storitveno neodvisnega nabora orodij** (ang. Service Neutral Toolkit) - gre za najvišji API nivo. Razvijalec uporablja skupen vmesnik za različne ponudnike storitev. Kot na tretjem nivoju je tudi tu poudarek na poslovnih objektih in procesih, vendar pa se mu tu ni

potrebno ukvarjati s tem, na katero konkretno storitev se nanašajo. Storitve, napisane z neodvisnim naborom orodij, je mogoče prenesti na poljuben oblak brez spreminjanja programske kode oziroma so te spremembe minimalne.

5.4 REST Ogrodja

5.4.1 Zrelostni model REST ogrodij

V tem delu si bomo ogledali neuradni zrelostni model REST ogrodij. Gre za hierarhično lestvico značilnosti, ki jim ogrodje mora zadostiti, da lahko govorimo o stopnji zrelosti. Leonard Richardson je v svoji razpravi govoril o štirih nivojih REST modela [27], ki so ga kasneje razširili na 6 nivojev [28]:

- **Nivo 0** - ogrodje na tem nivoju ni RESTful ogrodje, saj ne zagotavlja nikakršne podpore RESTful storitvam.
- **Nivo preslikav/preusmeritev in HTTP/URI enkapsulacija** (ang. Mapping/Routing and HTTP/URI Encapsulation) - ogrodje na prvem nivoju nudi dve funkcionalnosti, in sicer preprosto preslikavo operacij in sredstev v ustrezen jezik ter enkapsulacijo HTTP in URI upravljanja.
- **Nivo podpore medijskim tipom in odjemalcem** - na tem nivoju ogrodje zagotavlja podporo različnim formatom za predstavitev sredstev (XML, JSON) in podporo odjemalcem. Tu mislimo predvsem na knjižnice za delo s HTTP zahtevami, mehanizmi za pogajanje o tipih internetnih medijev itd.
- **Nivo modeliranja RESTful elementov v programski jezik** - na tem nivoju bi morali biti RESTful koncepti vdelani (ang. Embedded) v samo semantiko jezika.
- **Nivo podpore HATEOAS in semantiki** - na tem nivoju bi moralo ogrodje nuditi polno podporo HATEOAS.
- **Nivo podpore kodi na zahtevo in plastem** - za dodatne funkcionalnosti mora ogrodje dopuščati kodo na zahtevo (v obliki skriptnih programov) ter prepletenost plasti.

5.4.2 JAX-RS

Za razvoj spletnih storitev je na voljo več ogrodij. V tem delu nas zanimajo ogrodja za razvoj RESTful spletnih storitev, in sicer JAX-RS [29]. JAX-RS predstavlja ogrodje za implementacijo javanskih vmesnikov. Ponuja mehanizme za dodeljevanje javanskih anotacij POJO objektom (Plain Old Java

Objects). POJO objekti so javanski objekti, ki nimajo omejitev in niso vezani na nobeno ogrodje ali ostale modele. JAX-RS omogoča povezovanje specifičnih URI vzorcev in HTTP operacij z metodami naših javanskih razredov. Glavni cilji JAX-RS ogrodij so [30]:

- Zagotovljanje nabora anotacij in razredov za uporabo upravljanja POJO objektov. Definira tudi življenjski cikel in obseg objekta.
- Zagotavljanje podpore pogostim HTTP vzorcem na višjem nivoju in podpora raznolikim HTTP aplikacijam.
- Podpora različnim formatom podatkov v telesu HTTP entitet (ang. Entity Body Content-type) preko razširitev (ang. Add-ons) ali priključkov (ang. Plugins).
- Definira, kako so razredi razporejeni v okoljih Servlet Container in JAX–WS Provider razredu.
- Definira okolje za razrede, ki predstavljajo sredstva in so nameščeni v Java EE vsebnik (ang. Container). Opisuje, kako uporabiti funkcije Java EE in njene razrede.

5.4.2.1 Anotacije

JAX-RS uporablja anotacije (ang. Annotations) za poenostavitev razvoja in postavitev (ang. Deployment) spletnih storitev. Anotacije so povezovalniki, ki avtomatično generirajo kodo za povezovanje javanskih razredov z ogrodjem (ang. Framework). Nekaj tipičnih anotacij in primerov:

- **@Path** - določa relativno pot do sredstva oziroma metode (primer je prikazan z izvorno kodo 5.1).

Izvorna koda 5.1: Primer uporabe @Path anotacije.

```
@Path("/users")
public class UserResource {
       //TODO
}
```

- **@GET, @POST, @PUT, @DELETE, @HEAD** - določajo tip HTTP zahteve (primer je prikazan z izvorno kodo 5.2).

Izvorna koda 5.2: Primer uporabe @GET anotacije.

```
@Path("/users")
public class UserResource {
```

```
        @GET
        public String handleGETRequest() {
        //TODO
        }
}
```

- @**Produces** - določa internetni medijski tip HTTP odgovora namenjenega pogajanju o vsebini (glej izvorno kodo 5.3).

Izvorna koda 5.3: Primer uporabe @Produces anotacije.

```
@Path("/users")
public class UserResource {
        @GET
        @Produces("application/xml")
        public String getXML(String representation) {
        //TODO
        }
}
```

- @**Consumes** - določa internetni medijski tip HTTP zahteve. Prav tako je namenjen pogajanju o vsebini (glej izvorno kodo 5.4).

Izvorna koda 5.4: Primer uporabe @Consumes anotacije.

```
@Path("/users")
        public class UserResource {
        @PUT
        @Consumes("application/xml")
        public void updateUser(String representation) {
        //TODO
        }
}
```

- @**PathParam** - poveže parametre metod z deli URI naslova (glej izvorno kodo 5.5).

Izvorna koda 5.5: Primer uporabe @PathParam anotacije.

```
@Path("/users/{username}")
public class UserResource {
        @GET
        public String handleGET(@PathParam("username")
```

```
                                String Uime) {
        // del URLja se shrani v spremenljivko Uime
        }
}
```

- **@QueryParam** - poveže parametre metode z URI poizvedbenimi parametri.
- **@CookieParam** - poveže parametre metod z vrednostjo piškotkov.
- **@FormParam** - poveže parametre metod z vrednostjo polj obrazcev (ang. Form).
- **@DefaultValue** - privzeta vrednost, če ključ ne obstaja.
- **@Context** - vrne celoten kontekst objekta (npr. HTTPServlet Request).

5.4.2.2 JAX-RS razširitve

JAX-RS ogrodje nudi tudi podporo razširitvam. Funkcionalnosti razširja preko ponudnikov (ang. Provider). Ponudniki prestrezajo vezavo med predstavitvami sredstev, prestrezajo izjeme in zagotavljajo vsebino. JAX-RS ima na voljo tri ponudnike:

- **Ponudnik entitet** (ang. Entity Provider) - povezuje predstavitve sredstev z njihovimi povezanimi javanskimi tipi. Poznamo bralca sporočil (ang. *MessageBodyReader*) in zapisovalca sporočil (ang. *MessageBodyWritter*). Bralec sporočil predstavlja pogodbo med JAX–RS v času izvajanja in komponentami, ki zagotovijo povezavo med predstavitvami in javanskimi tipi. Zapisovalec sporočil predstavlja pogodbo med JAX–RS v času izvajanja in komponentami, ki zagotovljajo povezavo med javanskimi tipi in predstavitvami.
- **Ponudnik konteksta** (ang. Context Provider) - razredom in ostalim ponudnikom omogoča dostop do konteksta.
- **Ponudnik za preslikavo izjem** (ang. Exception Mapping Providers) - omogoča lovljenje izjem in povezovanje z ustreznim HTTP odgovorom.

5.5 JAX-RS implementacije

V nadaljevanju bomo na kratko predstavili področja, ki jim bomo posvetili pozornost pri implementaciji JAX-RS. Pregledali bomo tri najpogosteje uporabljena javanska RESTful ogrodja, in sicer:

- **Jersey**,

- **Restlet**,
- **RESTEasy**.

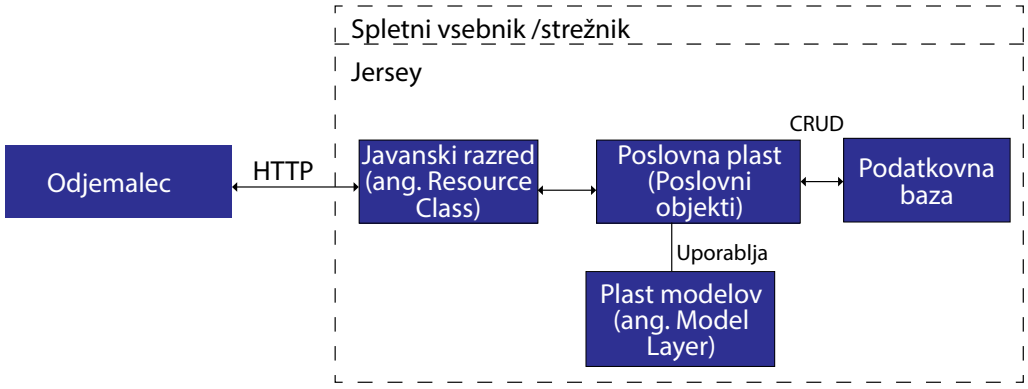
Za primerjavo bomo vzeli analizo podjetja IBM [31], v kateri so analizirali:

- **Vdelane vsebnike** (ang. Embedded Containers) - večina JAX-RS implementacij lahko deluje v servlet vsebniku. Včasih pa želimo, da delujejo tudi na aplikacijah, ki ne temeljijo na servlet tehnologiji.
- **API odjemalca** (ang. Client API) - JAX-RS opredeljuje zapletene povezevalne standarde za API odjemalce, vendar prepušča njihovo implementacijo ogrodju.
- **Prestrezanje HTTP zahtev** (ang. Interceptor Framework) - razvijalci spletnih storitev želijo pogosto obdelati HTTP zahteve pred njihovo izvršitvijo ali pa celo po njej. To je koristno pri prijavi v aplikacije, pri predpomnenju in validaciji ter avtorizaciji.
- **Podatkovni tipi** - JAX-RS omogoča enostavno dodajanje različnih vrst podatkov z uporabo razredov *MessageBodyReader* in *MessageBodyWriter*. Zanimala nas bo podpora najpogostejšim formatom, vključno z Atom, JSON in MIME tipi.
- **Integracija komponent** - povezovanje z drugimi ogrodji je pomembno pri razvoju RESTful storitev. Pogosto bomo za delo z uporabniškimi vmesniki uporabljali druga ogrodja (SPRING ali ostala MVC ogrodja). Zadnji vidik bo torej namenjen integraciji teh ogrodij.

5.5.1 Jersey

Jersey [32] je odprtokodno JAX-RS (JSR 311 & JSR99) ogrodje za implementacijo in razvoj RESTful spletnih storitev podjetja Sun. Ponuja lasten API in razširjen JAX-RS nabor orodij z dodatnimi zmogljivostmi za še enostavnejše delo z RESTful storitvami. Jersey med svoje cilje uvršča sledenje razvoju JAX-RS API-ja in izdaje rednih posodobitev, nudenje API vmesnikov za razširitev Jersey ogrodja in enostavno gradnjo RESTful spletnih storitev z uporabo Jave in javanskega navideznega stroja. Za razvoj in delovanje Jersey storitev potrebujemo najmanj Java SE 6. Jersey najdemo kot del Apache Maven projektno-razvojnega in upravljalnega orodja. Trenutno je na voljo verzija 2.2.

Vdelani vsebniki
Jersey običajno deluje v servlet vsebnikih, vendar pa podpira tudi vgrajeni način delovanja znotraj javanskih aplikacij. JAX-RS storitev v vdelanem (ang. Embedded) načinu je enostavno implementirati s kodo. V ta namen



Slika 5.1: Jersey arhitektura.

lahko uporabimo paket *com.sun.jersey.test.framework.spi.container.embedded*. Lahko uporabimo tudi testne enote ogrodja. Jersey test nudi podporo za Grizzly, JDK *(com.sun.net.httpserver.HttpServer)* in Simple HTTP Container.

API odjemalec

API odjemalec je zapleten, višje nivojski API za sklicevanje na RESTful spletne storitve. Zajema tri točke:

- **Enkapsulacijo RESTful omejitev**, še posebej uniformnega vmesnika na strani odjemalca.
- **Lažje upravljanje s spletnimi storitvami** na strani strežnika.
- **Vsebuje vzvode za JAX-RS API koncepte** na strani odjemalca. API odjemalec omogoča implementacijo HTTP povezav (razred HttpURLConnection). V splošnem pa ta API omogoča učinkovito izvajanje RESTful rešitev na strani odjemalca.

Ogrodje za prestrezanje zahtev

Jersey zagotavlja ogrodje za prestrezanje, ki temelji na filtrih. Filtri so mehanizmi, ki skrbijo za spreminjanje parametrov zahtev ali odgovorov. Na strani strežnika poznamo dva različna filtra: *ContainerResponseFilter* in *ContainerRequestFilter*.

Podatkovni tipi

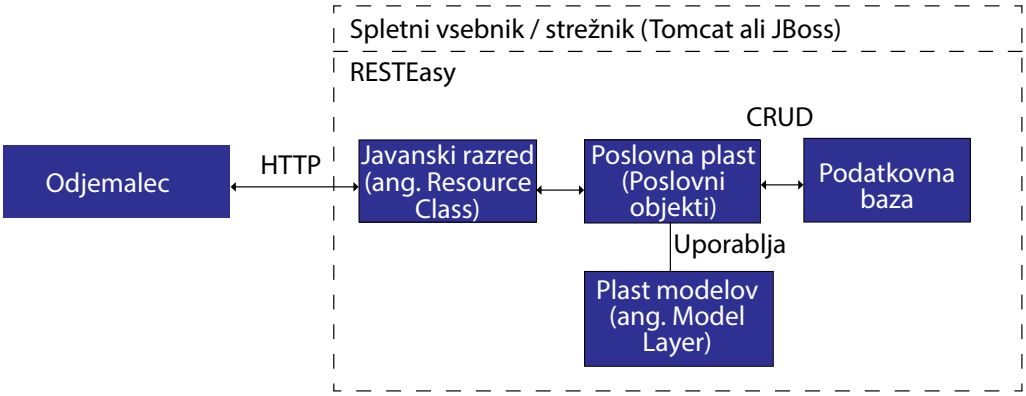
Jersey, tako kot druge JAX-RS implementacije, določa tudi JAX-RS razširitvene module za podporo skupnim formatom, vključno z Atom, JSON in MIME tipi. Za JSON lahko uporabimo MOXy, Jackson, Jettison. Vsak od omenjenih nudi vsaj tri pristope za delo s predstavitvami in sicer POJO pove-

zovalni pristop, JAXB povezovalni pristop (uporaba JAXB elementov) in nižjenivojsko razčlenjevanje in obdelavo JSON formata. Za XML uporablja podobne principe kot JSON, vključno z nižje nivojsko podporo, JAXB, POJO pristop in MOXy.

Integracija komponent
Jersey trenutno zagotavlja podporo dvema ogrodjema za injeciranje odvisnosti (ang. Dependency Injection) Spring in Google Guice. Spring za delovanje potrebuje Spring modul, ki ga lahko omogočimo v spletnem deskriptorju web.xml. Google Guice podpora je na voljo pri sklicevanju na Guice filter, ki ga prav tako omogočimo v web.xml datoteki.

5.5.2 RESTEasy

RESTEasy [33] je JBoss projekt, ki vsebuje različna ogrodja za izdelavo REST spletnih storitev in REST javanskih aplikacij. Model delovanja je podoben ogrodju Jersey.



Slika 5.2: RESTEasy arhitektura.

Vdelani vsebnik
JBoss RESTEasy ponuja enostaven, vdelan servlet TJWS, ki ga lahko uporabimo za testiranje. TJWS lahko uporabimo, ne da bi zagnali celoten servlet. Prav tako podpira javanski JDK vsebnik in Netty.

API odjemalec
RESTEasy zagotavlja API, s katerim lahko oddajamo HTTP zahteve. Vsebuje tri razrede: *Client*, *WebTarget* in *Response*. *WebTarget* razred predsta-

vlja enoličen URL naslov. RESTEasy ima tudi vgrajeno Proxy ogrodje, ki nudi drugačen način razvoja odjemalcev. Namesto JAX-RS anotacij Proxy ogrodje ustvari HTTP zahtevo in jo uporabi pri klicu RESTful spletne storitve, ki niso nujno implementirane z JAX-RS. Komunikacija med strežnikom in odjemalcem poteka preko *HTTPClient* razreda.

Ogrodje za prestrezanje zahtev
RESTEasy za prestrezanje uporablja poslušalce, ki se imenujejo prestrezniki (ang. Interceptors), ki lahko prestrežejo JAX-RS klice in jih preusmerijo. Obstajajo štiri različne vrste prestreznikov na strani strežnika: *MessageBodyReader* prestrezniki, *MessageBodyWriter* prestrezniki in pred ter po obdelovalni prestrezniki. Tudi odjemalec ima svoje prestreznike.

Podatkovni tipi
Kot pri drugih implementacijah JAX-RS tudi RESTEasy podpira večino priljubljenih formatov, vključno z XML in JSON. RESTEasy omogoča tudi enostavno JAXB podporo Atom objektnim modelom.

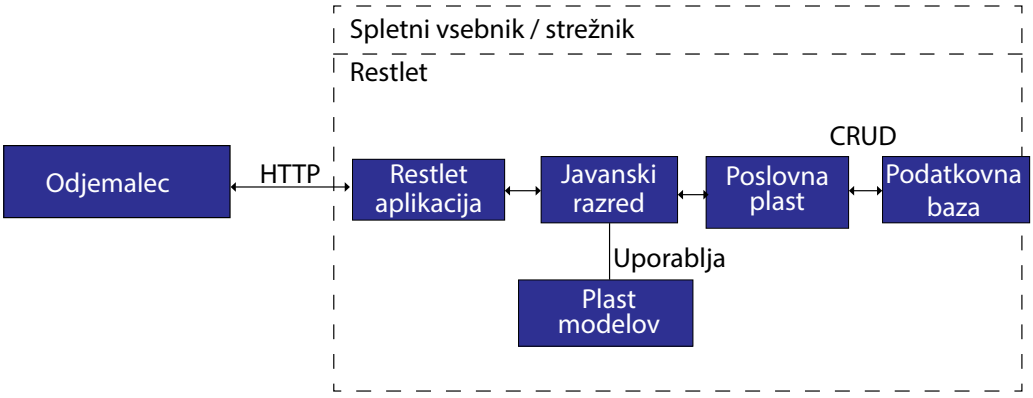
Integracija komponent
Kot del JBoss projekta je razumljivo, da RESTEasy podpira tesno integracijo z JBoss Seam komponento. Prav tako podpira integracijo z drugimi priljubljenimi ogrodji in standardi, kot so Enterprise JavaBean (EJB), Spring, Spring MVC in Google Guice.

5.5.3 Restlet

Restlet [34] je enostavno, odprtokodno ogrodje za Javo. Podpira vse večje podatkovne formate, prenosne protokole in standarde za opis storitev. Omenimo lahko, da Restlet 1.0 za razliko od Restlet 2.0 ne uporablja tipičnih anotacij za delo s HTTP metodami.

Vdelani vsebovalniki
Restlet je bil vedno neodvisen od protokolov, kar je omogočilo razvoj drugačnih postavitvenih nastavitev, vključno s samostojnimi JAR datotekami (Java Archive files), servlet vsebniki, Spring vsebniki in nabori orodij. Restlet podpira tudi lastno XML konfiguracijo za podporo Spring konfiguracijskih mehanizmov XML.

API odjemalec
Restlet API odjemalec omogoča preprosto uporabo storitev na osnovi HTTP-ja in ne samo JAX-RS storitev. Temelji na arhitekturi, sestavljeni iz pove-



Slika 5.3: Restlet arhitektura.

zovalnikov in komponent.

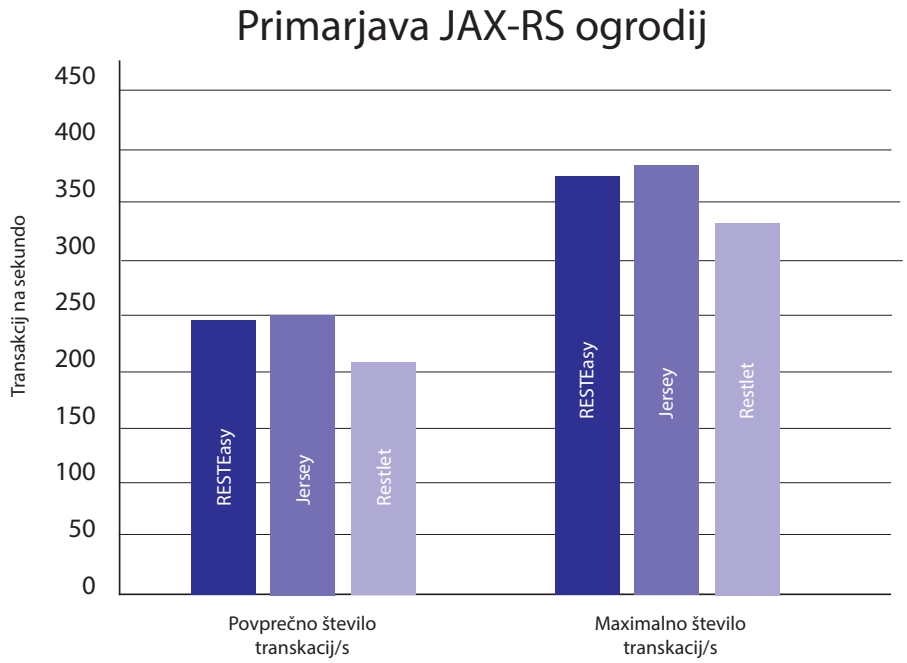
Ogrodje za prestrezanje zahtev
Restlet uporablja zapletene usmerjevalnike za preusmerjanje URI klicev znotraj aplikacije. Usmerjevalniki povezujejo URI naslove z vsemi sredstvi, ki se navezujejo na dani naslov. Z razširitvijo abstraktnega razreda *Filter* lahko prestrežemo preusmeritve, ki jih izvršujejo usmerjevalniki. Filtri podpirajo obdelavo klicev pred ali po njihovi izvršitvi.

Podatkovni tipi
Tudi Restlet podpira glavne podatkovne formate, kot so XML, JSON in Atom. Vgrajena XML podpora preko JAXB, JiBX, SAX in DOM.

Integracija komponent
S svojo bogato knjižnico razširitev Restlet podpira integracijo z različnimi ogrodji in standardi, kot so Spring, Jetty, Grizzly, JAXB, JiBX in Velocity.

5.6 Primerjava REST ogrodij

Za kriterij primerjave bomo vzeli število transkacij v sekundi, ki se izvršijo pri klicu GET zahteve na Apache Tomcat strežniku. Na vsakem ogrodju smo zahtevo poslali 1500-krat in s tem dobili statistično zanesljive podatke. Graf na sliki 5.4 prikazuje, da je učinkovitost vseh testiranih ogrodij približno enaka - z izjemo Restlet ogrodja. V povprečju se je največ transakcij pri klicanju GET zahteve zgodilo pri Jersey ogrodju. Prav tako je to ogrodje zabeležilo tudi največji modus. Že prejšnji pregled nam je dal sliko, da za-



Slika 5.4: Primerjava REST ogrodij.

snova RESTEasy in Jersey temeljita na istih mehanizmih, medtem ko Restlet uporablja strategijo preusmerjanja. Odstopanja seveda niso takšna, da bi se odpovedali Restlet ogrodju. Primerjavo izbranih ogrodij si lahko, glede na izhodiščne kriterije predstavljene na začetku, ogledate v tabeli 5.1.
Kot je bilo slutiti že na začetku, vsa obravnavana ogrodja zadoščajo JAX-RS in nudijo odlično podporo gradnji RESTful spletnih storitev. Vsi delujejo vsaj v Servlet vsebniku. Prav tako vsi nudijo podporo najpogostejšim predstavitvam sredstev (XML, JSON) in integracijo s Spring ogrodjem. Kriterija, po katerih se razlikujejo, sta API odjemalec in implementacija prestreznikov, vendar pa funkcionalnosti ostajajo podobne. Odločitev, katero ogrodje izbrati, je prepuščena razvijalcem glede na njihove pretekle izkušnje s tehnologijami in komponentami, ki so podprte v izbranih ogrodjih. V okviru tega dela bomo za samo implementacijo uporabili Jersey ogrodje, ki nudi odlično dokumentacijo in aktivno podporo skupnosti. Omeniti velja, da opisana ogrodja še zdaleč niso edina, ki so namenjena za JAX-RS.

	Jersey	**RESTEasy**	**Restlet**
Vdelan vsebnik	JDK, Grizzly, Simple HTTP, Servlet	JDK, Netty, TJWS, JBossWeb, Servlet	JDK, Grizzly, Jetty, Netty, Simple HTTP, Servlet
API odjemalec	Enkapsulacija RESTfull omejitev - uniformen vmesnik, implementacija JAX-RS 2.0	Implementacija JAX-RS 2.0, proxy framework	Razširitev Simple razreda in uporaba usmerjevalnikov
Prestrezanje zahtev	ContainerRequestFilter, ContainerResponseFilter	MessageBodyWriter, MessageBodyReader, interceptor pred in po izvršitvi zahtev	Razred Filter - pred in po izvrševanju zahtev
Podatkovni tipi	XML, JSON, Atom	XML, JSON, Atom	XML, JSON, Atom
Integracija komponent	Guice, Spring	EJB, Spring, Spring MVC, Guice	Spring, Velocity, Jetty

Tabela 5.1: Primerjava JAX-RS implementacij.

Poglavje 6

Računalništvo v oblaku

Razvoj aplikacij je običajno zapleten, drag in počasen proces. Vsaka aplikacija je v preteklosti za nemoteno delovanje potrebovala strojno opremo, operacijski sistem, podatkovne baze, vmesno programsko opremo in spletne strežnike. Še več, potrebna je bila ekipa strokovnjakov za upravljanje z omrežjem, podatkovnimi bazami, operacijskim sistemom itd. Poleg tega velika podjetja pogosto potrebujejo posebne aplikacije in funkcionalnosti za zadovoljitev svojih potreb, za kar potrebujejo lastne podatkovne centre in ekipo za njihovo vzdrževanje. Za delovanje in hlajenje strežnikov so bile potrebne tudi ogromne količine električne energije. Dodatno pa je bilo potrebno vzdrževati redundanco sistema v primeru težav ali nesreč. V ta namen se je pojavilo računalništvo v oblaku.
Računalništvo v oblaku (ang. Cloud Computing) [35] je pojem, ki pomeni izrabo zmogljivosti računalniških virov, ki so v obliki storitev dosegljivi preko omrežja (tipično preko interneta). Računalništvo v oblaku je iskalo navdihe in vzporednice v nekaterih drugih sistemih:

- **Avtonomno računalništvo** - glavna značilnost takšnega sistema je samoupravljanje, saj se z večanjem kompleksnosti sistema povečuje tudi število enot in parametrov za upravljanje. Takšno računalništvo uporabniku in skrbniku lajša uporabo.

- **Model strežnik odjemalec** (ang. Server-Client Model) - gre za trenutno prevladujočo obliko v računalniških omrežjih. Med strežnikom in odjemalcem poteka izmenjava sporočil preko komunikacijskega protokola, pri čemer je odjemalec pobudnik takšne izmenjave.

- **Mrežno računalništvo** (ang. Grid Computing) - gre za porazdeljeni

sistem računalnikov, tipično na različnih fizičnih lokacijah, ki opravljajo zahtevno delo. Glavna strategija mrežnega računalništva je uporaba vmesne programske opreme za porazdelitev nalog na več računalnikov.

- **Glavni računalniki** (ang. Mainframe Computer) - to so visoko zmogljivi in zanesljivi računalniki, ki so zmožni sočasno upravljati in izvrševati več operacijskih sitemov, zmožni so upravljanja z velikim številom vhodnih in izhodnih parametrov. Uporabljajo jih predvsem korporacije in vladne organizacije.

- **Storitveno računalništvo** (ang. Utility Computing) - govorimo o "On Demand"računalništvu. Računalniški viri so na voljo v najem uporabnikom, pri čemer se najem zaračuna po porabi.

- **P2P (Peer to Peer)** - model povezovanja sistemov, kjer je lahko vsak računalnik v omrežju strežnik ali odjemalec, brez uporabe osrednjega sistema.

Navadno se v takšnih omrežjih prenaša digitalna vsebina (avdio, video, tekst).

Podjetje IBM [36] računalništvo v oblaku opredeljuje kot celostno rešitev, pri čemer so vsi računalniški viri (strojna oprema, programska oprema, podatkovne baze itd.) dostopne uporabnikom glede na trenutno povpraševanje.

Splošno sprejeta definicija s strani NIST-a (National Institute of Standards and Technology) [37] je:

> Računalništvo v oblaku je model, ki omogoča nenehen in enostaven "On Demand"dostop do poljubno nastavljivih računalniških virov, kot so računalniško omrežje, strežniški sistemi, podatkovne baze, aplikacije in storitve. Takšen sistem je mogoče hitro vzpostaviti z minimalnim stroškom upravljanja in minimalno interakcijo s ponudnikom storitve oblaka. Model računalnika v oblaku sestoji iz petih glavnih značilnosti, treh storitvenih modelov in štirih vzpostavitvenih modelov.

6.1 Značilnosti računalništva v oblaku

NIST [37] v svojem dokumentu opredeljuje pet glavnih značilnosti računalništva v oblaku:

- **Storitev na zahtevo** (ang. On Demand Self Service) - uporabnik lahko razpoložljive vire koristi samodejno brez interakcije s ponudnikom storitve.
- **Širok mrežni dostop** (ang. Broad Network Access) - računalniški viri so dosegljivi preko omrežja z uporabo standardnih tehnologij in protokolov.
- **Združevanje virov** (ang. Resource Pooling) - lokacija računalniških virov uporabnikom ni znana, prav tako jih ne morejo nadzorovati. Viri se dinamično dodeljujejo uporabnikom glede na njihove potrebe.
- **Rastoča elastičnost** (ang. Rapid Elasticity) - računalniški viri se lahko samodejno povečajo glede na prihajajoče število zahtev za obdelavo, nasprotno pa se v manjšem številu zahtev samodejno zmanjšajo. Uporabnik ima v vsakem trenutku na voljo neomejeno količino računalniških virov.
- **Merjenje storitev** (ang. Measured Service) - sistem oblaka samodejno nadzira, meri in optimizira izrabo virov glede na tip zahtev.

6.2 Storitveni modeli

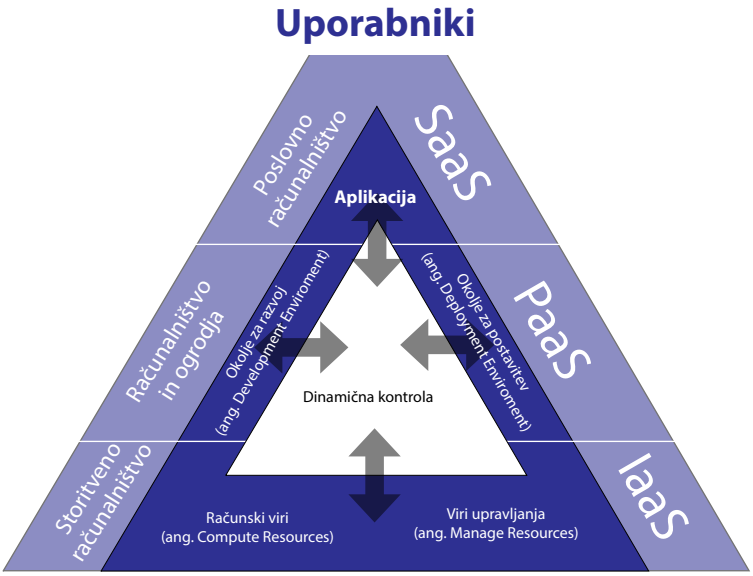
NIST v svojem dokumentu prav tako opisuje tri različne storitvene modele, ki jih bomo spoznali v nadaljevanju. Njihov obseg si lahko ogledamo na sliki 6.1.

SaaS (Software as a Service) je storitev, ki je preko oblaka na voljo uporabnikom. Takšna aplikacija je dostopna preko različnih odjemalcev (spletni brskalnik, programski vmesniki). Odjemalec v nobenem trenutku ne upravlja in nadzira infrastrukture oblaka (omrežje, strežniki, operacijski sistem, podatkovne enote) z izjemo določenih uporabniških nastavitev. Za izrabo SaaS administrativne operacije niso potrebne, prav tako ne namestitve in posodobitve programske opreme. Tipičen primer predstavljajo storitve spletnih pošt (Gmail, Hotmail itd.).

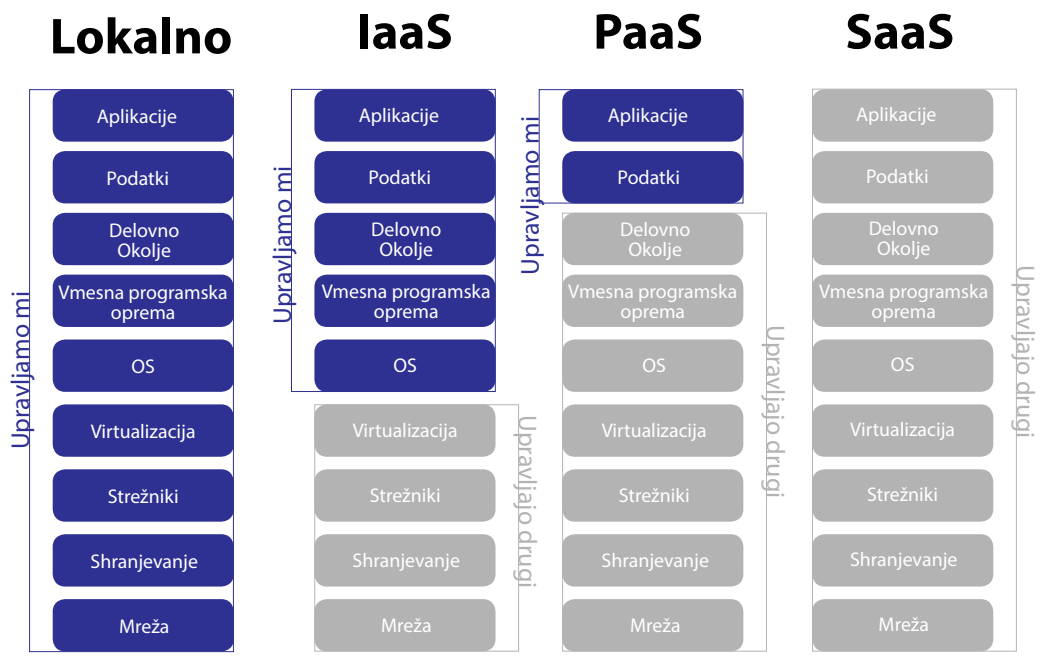
PaaS (Platform as a Service) je storitev, ki uporabnikom nudi razvoj in postavitev aplikacij, razvitih v programskih jezikih (vključno z vsemi knjižnicami, ogrodji in ostalimi orodji, ki so na voljo) na oblak. Uporabnik nima nadzora nad nižjo infrastrukturno plastjo (nastavitve strežnika, operacijski sistem, podatkovne enote), lahko pa upravlja s postavljenimi aplikacijami in nastavitvami okolja, kjer teče aplikacija.

IaaS (Infrastructure as a Service) je storitev, ki uporabnikom nudi celovito rešitev za obdelavo zahtev, možnost različnega shranjevanja podatkov, mreženja in ostale osnovne računalniške vire za zagotavljanje kakovostnih storitev (npr. izenačevanje bremena (ang. Load Balancing). Uporabnik ne nadzoruje infrastrukturnega sloja, upravlja pa z operacijskim sistemom, podatkovnimi bazami, aplikacijami ter nastavitvami mreženja (npr. požarni zid). Prednost takšnega koncepta je v času, ki je potreben za postavitev aplikacij.

Vsak model storitev je namenjen določenemu tipu uporabnikov. Za IaaS je to sistemski inženir, za PaaS je razvijalec, SaaS pa je storitev, ki je na voljo končnemu uporabniku. Vsi trije modeli storitev se razlikujejo tudi glede na razmerje med računalniškimi viri, s katerimi lahko upravljajo uporabniki storitev, ter viri, ki jih lahko upravlja le ponudnik storitev v oblaku. Primerjavo med tremi modeli in lokalnim sistemom (ang. On Premisis) prikazuje slika 6.2.



Slika 6.1: Obseg storitvenih modelov.



Slika 6.2: Primerjava storitvenih modelov.

6.3 Modeli vzpostavitve oblaka

NIST [37] za vzpostavitev oblaka navaja štiri različne modele. Z njimi opisuje različne lastniške scenarije in lokacijsko porazdeljenost oblaka. Vzpostavitveni modeli so:

Zasebni oblak (ang. Private Cloud) - infrastruktura oblaka je na voljo izključno posamezni organizaciji, ki sestoji iz več uporabnikov. Takšna infrastruktura je lahko last same organizacije, tretje osebe ali pa kombinacije obeh. Lahko se nahaja na mestu organizacije ali pa na poljubni lokaciji (ang. Off Premises).

Skupni oblak (ang. Community Cloud) - infrasktruktura oblaka je na voljo skupnosti organizacij, ki imajo podobne interese in cilje. Takšen oblak je lahko v lasti ene ali več organizacij, tretje osebe ali pa kombinacij med njimi. Lahko se nahaja na mestu organizacije ali pa na poljubni lokaciji.

Javni oblak (ang. Public Cloud) - infrastruktura oblaka je na voljo za splošno uporabo. Lahko je v lasti poslovnih, izobraževalnih ali vladnih organizacij oziroma kombinacij med njimi. Lahko se nahaja na mestu organizacije (ang. On Premises) ali pa na poljubni lokaciji (ang. Off Premises).

Hibridni oblak (ang. Hybrid Cloud) - infrastruktura oblaka je sestavljena iz dveh ali več različnih infrastruktur, ki delujejo kot posamezni deli, vendar ostajajo medsebojno povezani. Tako še vedno omogočajo prenosljivost aplikacij in podatkov.

6.4 Prednosti računalništva v oblaku

Prednosti računalništva v oblaku [38] izhajajo že iz samih karakteristik, opredeljenih v uvodu. K temu lahko dodamo še nekatere, ki so pomembne tudi iz poslovnega vidika:

- **Agilnost** (ang. Agility) - nudi hitro dostavo zahtevanih storitev in s tem hitrejši ter učinkovitejši odziv organizacij na vsakodnevne poslovne situacije, kot so pritiski tekmecev, tržni izzivi, omejitve sredstev itd.

- **Osredotočenost na poslovanje** (ang. Business Focus) - organizacije se lahko osredotočajo na poslovanje in ne na iskanje primernih informacijskih sistemov in virov ter kako in kdo jih bo zagotavljal.

- **Nadzor stroškov in razpoložljivih sredstev** (ang. Cost And Budget Control) - storitve v oblaku omogočajo večji nadzor nad stroški in boljše upravljanje s sredstvi. Ponudniki storitev v oblaku storitve zaračunavajo na podlagi naročnine ali pa na podlagi količine uporabljenih virov (ang. Pay As You Go). Zmanjša se tudi strošek IT kadra in vzdrževanja strojne opreme.

- **Skalabilnost in upravljanje kapacitet** (ang. Scalability And Capacity Management) - skalabilnost je sposobnost sistema, da se prilagaja številu zahtev in tako zagotavlja normalno delovanje tudi v primeru povečane obremenitve. Deluje tudi v obratni smeri znižanja števila zahtev.

- **Mobilnost** (ang. Mobility) - z računalništvom v oblaku lahko uporabniki dostopajo do storitev kjerkoli in s pomočjo katerekoli naprave.

Slika 6.3: Prednosti računalništva v oblaku.

6.5 Slabosti računalništva v oblaku

Kljub številnim prednostim pa ima računalništvo v oblaku tudi nekaj pomankljivosti [39]. Takšna oblika računalništva je **namenjena predvsem srednje velikim in večjim organizacijam**, ki imajo svoje IT oddelke in lahko razvijajajo IT rešitve po svoji meri. **Nastavitve oblaka in virov niso poljubno nastavljive**, kot to omogočajo drugi modeli računalništva. Prav tako oblak ni najbolj primeren za prenos večjega števila podatkov (latenca), na drugi strani pa je zelo primeren za procesorsko zahtevne naloge. Oblak je **sistem brez stanja**. Za ohranitev komunikacije na takšnem porazdeljenem sistemu je nujno potrebna dvosmerna komunikacija (model zahteva-odgovor), kar pa takšna arhitektura ne predvideva. Posledica tega so različne, nepredvidljive poti sporočil, podatki lahko prihajajo v nepravilnem zaporedju itd. Glavna in največkrat izpostavljena slabost pa je **zasebnost in varnost podatkov**. Podatki v oblaku niso več v naši kontroli, ampak pod nadzorom ponudnika oblaka. Obstajajo različne regulative in zakoni o varstvu podatkov na strežnikih in transkacijah za različne države. Primer je Kitajska, ki je s svojo restriktivno in samovoljno politiko prisilila Google, da je prestavil svoje strežnike v Hong Kong. Tudi EU zakonodaja obvezuje ponudnike oblakov, da obveščajo svoje odjemalce o prenosu podatkov izven svojih meja itd. Če se torej naša aplikacija nahaja na strežnikih v različnih državah z različnimi zakoni, se je potrebno prilagoditi vsem [40].

6.6 Načrtovanje kapacitet

V osnovi gre pri planiranju kapacitet [39] za prilagajanje računalniških virov glede na obremenjenost in je najpomembnejši faktor pravilnega načrtovanja

računalništva v oblaku. Pogosto ta pojem napačno enačimo z optimizacijo delovanja sistema. Pri načrtovanju kapacitet gre za pravilno in optimalno razporeditev delovne obremenitve. Z drugimi besedami - maksimiziramo delo glede na trenutne vire, ki jih zasedamo. Za uspešno nastavitev sistema je potrebno poznati omejitve sredstev, ki jih imamo na voljo in poiskati ozko grlo (ang. Bottleneck). Vsak sistem lahko skaliramo **vertikalno** (povečamo računalniško moč virov) ali **horizontalno** (povečamo število računalniških virov). Horizontalno skaliranje lahko vodi do tako imenovanih strežniških farm (ang. Server Farms). Prednosti enega ali drugega skaliranja niso jasno opredeljive in so odvisne od zahtev posamezne aplikacije.

6.7 PaaS model

PaaS model računalništva v oblaku predstavlja storitev, kjer imamo znotraj okolja na voljo orodja za vzpostavitev SaaS. Aplikacije, razvite na PaaS sistemih, so navadno kompleksne poslovne aplikacije, podatkovni portali ali pa porazdeljene aplikacije iz različnih podatkovnih virov. PaaS okolja lahko ponudijo celovito upravljanje in kontrolo nad življenjskim ciklom aplikacij. Aplikacijsko ogrodje je osnovno orodje za izgradnjo aplikacij na oblaku. Takšno ogrodje vsebujejo danes vsi bolj znani ponudniki PaaS storitev, kot so Google App Engine, Windows Azure, AWS in drugi. Med pomembnimi elementi za razvoj in vzpostavitev aplikacij bi lahko omenili še virtualni stroj (ang. Virtual Machine). PaaS sisteme bi lahko idejno primerjali s sodobnimi CMS sistemi (ang. Content Managing System), kjer so na voljo osnovne nastavitve, nato pa je od razvijalca odvisno, v kateri smeri bo razvoj potekal. Bistvo PaaS je, da ponudnik takšnih storitev v oblaku v zameno za plačilo ponuja razvojno in razvijalno okolje.

6.7.1 Razpon PaaS storitev

Za razvoj spletnih aplikacij mora platforma v oblaku nuditi vsaj naslednje storitve [39]:

- **Storitve za razvoj aplikacij** - PaaS platforma nudi storitve in vire za razvoj aplikacij v podprtem programskem okolju. Nekateri celo nudijo vizualne vmesnike, ki nato kodo generirajo samodejno.
- **Upravljanje s podatki** - PaaS mora nuditi storitve in orodja za učinkovito upravljanje in shranjevanje podatkov v oblaku.
- **Meritve delovanja aplikacij in testiranje** - PaaS nudi orodja za merjenje potreb naših aplikacij. Prav tako nudijo orodja za testiranje aplikacij, kot je iskanje točk preloma obremenitve.

- **Upravljanje s transakcijami** - večina PaaS ponudnikov omogoča ali celovito upravljanje s transakcijami ali pa posredne storitve za upravljanje z njimi.

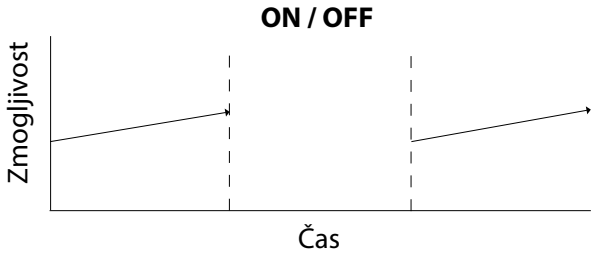
6.7.2 Značilnosti PaaS

Za razvoj robustnih, skalabilnih in zanesljivih aplikacij na PaaS je zaželjeno, da ima platforma v oblaku naslednje značilnosti [39]: ločeno podatkovno upravljanje od uporabniškega vmesnika, upošteva standarde računalništva v oblaku, nudi možnost uporabe IDE (Integrated Development Enviroment), nudi orodja za upravljanje življenjskega cikla aplikacij, podpira večnajemniško arhitekturo, biti mora fleksibilen in storitveno-integriran model in mora vsebovati orodja za kontrolo, testiranje ter optimizacijo.

6.7.3 Vzorci za uporabo PaaS

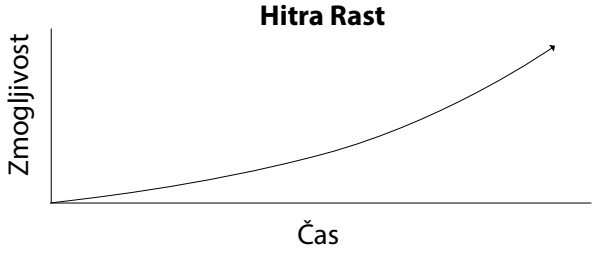
V tem delu vam bomo prikazali tipične probleme [41], ki jih platforme uspešno premagujejo:

ON/OFF vzorec
Obremenitve prihajajo v paketih, kot prikazuje slika 6.4. Obstaja čas večje obremenitve sistema in čas, ko sistem ni obremenjen oziroma je v stanju pripravljenosti. V tem primeru je prekomerna količina računalniških virov v obdobju neobremenjenosti popolnoma neizkoriščena.



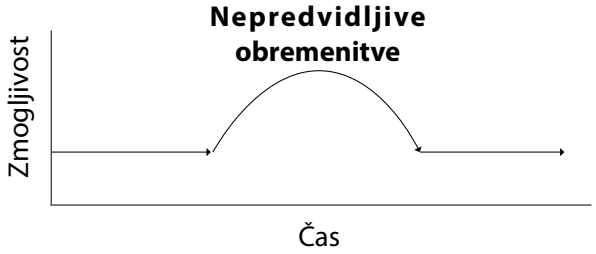
Slika 6.4: Prikaz razpoložljive računske moči.

Vzorec hitre rasti (ang. Fast Growth)
Uspešne storitve s časom zahtevajo več virov (glej sliko 6.5). Usklajevanje rasti storitev z rastjo virov je kompleksno, ki pa je premostljivo z uporabo PaaS.
Nepredvidljive obremenitve (ang. Unpredictable Bursting)
Delovanje storitev lahko prestresejo različni nepredvidljivi dogodki, ki lahko



Slika 6.5: Prikaz razpoložljive računske moči v časovnem obdobju v primeru hitre rasti.

povišajo obremenitev sistema. Takšna nenadna obremenitev in neprilagajanje anomalijam nižajo uporabno vrednost storitev.

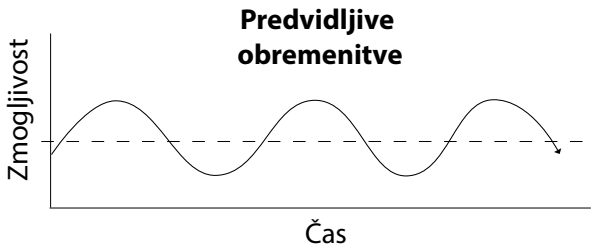


Slika 6.6: Prikaz razpoložljive računske moči v časovnem obdobju v primeru nepredvidljivih obremenitev.

Predvidljive obremenitve (ang. Predictable Bursting)
Gre za storitve, ki imajo mikro sezonske obremenitve. To so obremenitve, ki jih je možno napovedati in se jim tudi prilagoditi. Podobno kot pri ON/OFF vzorcu tudi tu prihaja do neizkoriščenja vseh razpoložljivih računalniških virov. Aplikacije je v takšnih primerih najlažje načrtovati.

6.7.4 Večnajemniški model

Vsaka prava storitev v oblaku uporablja večnajemništvo (ang. Multi Tenancy) [42]. Večnajemništvo je zmožnost, da več strank (najemnikov) izkorišča in uporablja iste aplikacije in/ali računalniške vire. Skozi večnajemništvo se kaže glavna prednost računalništva v oblaku, in sicer visoka stroškovna učinkovitost. Takšen model arhitekture mora uskladiti stroškovne prednosti z zahtevo po varnosti podatkov in zanesljivim delovanjem aplikacij.



Slika 6.7: Prikaz razpoložljive računske moči v časovnem obdobju v primeru predvidljivih obremenitev.

Obstajata dva načina implementacije večnajemniškega modela [43]: **Deljenje vmesnega sloja programske opreme** (pod vmesni sloj programske opreme spadajo npr. aplikacijski strežniki in podatkovne baze) in **virtualizacija**. Pri delitvi vmesnega sloja programske opreme se poslužujemo treh pristopov:

- **Ena sama instanca aplikacije** - najemniki si delijo operacijski sistem, strežnike, vmesni sloj programske opreme in eno samo instanco aplikacije.
- **Več instanc aplikacije v deljenem naslovnem prostoru** - najemniki si delijo operacijski sistem, strežnike, vmesni sloj programske opreme, vsak pa uporablja svojo instanco aplikacije. Ker se vmesni sloj programske opreme deli med najemnike, vsi uporabljajo isti proces operacijskega sistema oziroma isti naslovni prostor.
- **Več instanc aplikacije v ločenem naslovnem prostoru** - najemniki si delijo operacijski sistem in strežnike, vsak pa uporablja svojo instanco vmesnega sloja programske opreme in svojo instanco aplikacije. Ker vsak najemnik uporablja svojo instanco vmesnega sloja programske opreme, mu pripada lastna množica procesov operacijskega sistema oziroma lastni naslovni prostor.

O virtualizaciji govorimo, ko si najemniki delijo fizične strežnike, vsak pa uporablja svojo navidezno napravo, na kateri ločeno tečejo aplikacije, ločen vmesni sloj programske opreme in ločen operacijski sistem.

6.7.5 Ponudniki PaaS rešitev na trgu

Uporaba PaaS rešitev že od pojavitve leta 2009 do danes beleži strmo rast tako odjemalcev PaaS storitve kot tudi ponudnikov. Rast se kaže v štirih

[44] segmentih:

- **PaaS postaja vodilni model** - skupina analitikov iz Synergy Research Group je ugotovila, da je uporaba PaaS in IaaS modelov samo v letu 2012 zrasla za 65 %. Samo v drugi polovici leta so prihodki v industriji računalništva v oblaku znašali krepko preko 2,75 milijarde dolarjev. Ker se vrednost PaaS povečuje, se povečuje tudi konkurenca za tržni delež. Ob koncu leta 2012 je bil Salesforce.com prevladujoč igralec v tej panogi z 21 % tržnega deleža. Danes je zaznati predvsem vzpon Amazonovih rešitev in Googla. Kupci PaaS rešitev lahko pričakujemo cenovno vojno med ponudniki PaaS, ki bo podobna tisti iz leta 2012 med Amazonom, Microsoftom in Googlom za prevlado v IaaS.

- **Prevzemi manjših podjetij** - večja konkurenca bo večje ponudnike PaaS prisilila k prevzemu manjših podjetij. Podjetja, kot so Amazon in Google, bodo odkupila manjša podjetja ali pa jih bodo z nižjimi cenami izrinili iz trga. Trend kaže, da bodo podjetja, ki ponujajo samo PaaS rešitve, težko preživela.

- **Razvoj odprtokodnih rešitev in zasebna programska oprema** - z večanjem konkurence in uporabnikov se bo sprostila zahteva po odprtokodnih rešitvah na področju PaaS. Te bodo prevladujoča izbira malih in srednje velikih podjetij, ki bodo želeli svoje stroške minimizirati. Večja podjetja, ki si lahko privoščijo, da razvijajo svoje lastne rešitve, pa bodo sklenila pogodbo z zasebnimi razvijalci programske opreme. Za njih bodo dodatni stroški opravičljivi z dodatno komponento varnosti in prilagojeno programsko opremo, ki bo popolno sovpadala s poslovnimi procesi. PaaS industrija bo tako rasla skozi odprtokodne in zasebne projekte.

- **Platforme, namenjene posebnim zahtevam** (ang. Special Purpose Platforms) - pričakovati je, da se bo raznolikost PaaS ponudnikov večala in v končni fazi ne bo vodila do "de facto"platforme, kot se je to zgodilo s fenomenom operacijskega sistema Microsoft Windows v primeru osebnih računalnikov. Razmere so sedaj popolnoma drugačne, saj trg obvladuje ogromno razvijalcev programske opreme, ki ima razpon skozi različne industrije (od avtomobilske do zabavne itd.). Prav tako so danes potrebe po konkretnih rešitvah z izpostavljenimi funkcijami nujne za delovanje podjetij.

Med množico različnih ponudnikov rešitev na platformah v oblaku, se je težko odločiti za le enega. Držimo se lahko nekaj splošnih smernic za pravilno iz-

biro:

Programski jezik in tehnologije na strani strežnika - izbira programskega jezika je prvi in pogosto najpomembnejši korak v začetni fazi projekta. Izbira programskega jezika močno vpliva na programerske paradigme in komponente, ki jih postavljamo. Tudi tehnologije, ki jih uporablja strežnik, močno vplivajo na razvoj aplikacij. Najpogosteje uporabljamo tehnologije, kot so .NET, PHP in Java, ki so tudi splošno podprte med večino PaaS ponudnikov.

Shranjevanje podatkov - možnosti shranjevanja podatkov je danes vse več. V preteklosti so razvijalci tipično izbirali med prilagojeno datotečno shrambo in relacijskimi bazami. Če so se odločili za relacijske baze, so lahko izbirali med množico odprtokodnih in komercialnih baz. Pri izbiri PaaS moramo torej predvideti potrebe shranjevanja, ki jih bo naša aplikacija zahtevala.

Orodja in integracija aplikacij ter podpora - podpora razvijalskim orodjem je še en pomemben faktor pri izbiri PaaS ponudnika. Nekateri ponudniki za integracijo ponujajo orodja, kot so Eclipse, NetBeans ali Visual Studio. Zanesljiva integracija zmanjšuje čas in administracijo z upravljanjem kode in prenosom aplikacij na strežnik. Podobno velja za orodja za upravljanje s kodo (npr. Git). Vidik, ki ga moramo upoštevati, je tudi, ali lahko naše podatke shranjene na strežniku integriramo z ostalimi aplikacijami na oblaku. Vprašati se moramo, ali je potrebno podatke za delo z drugimi aplikacijami izvoziti v drugačen format ali je enostavneje podvojiti in kopirati podatke v drugo bazo itd.

Stroški in proračun - k zgoraj naštetim tehničnim vidikom moramo prišteti še stroške za delovanje aplikacij na PaaS. Prav tako moramo opredeliti količino sredstev, ki smo jih pripravljeni vložiti.

PaaS ponudniki ponujajo svoje rešitve v različnih oblikah in velikostih. V nadaljevanju si bomo ogledali glavne med njimi: Google App Engine, AWS Elastic Beanstalk, Heroku in Microsoft Azure.

6.7.6 Google App Engine

Google App Engine (GAE) je PaaS, ki uporablja podobne tehnologije, ki delujejo ali gostujejo na isti infrastrukturi kot Google. Ta storitev razvijalcem omogoča, da razvijejo in postavijo spletne aplikacije, nato pa prepustijo Goo-

glu, da upravlja z infrastrutkurnimi potrebami, kot je kontrola, upravljanje z instancami itd. Da bi aplikacije delovale na platformi App Engine, morajo zadoščati Googlovim standardom, kar močno omeji prenosljivost aplikacij.

6.7.6.1 Shranjevanje podatkov

App engine uporablja različne načine shranjevanja podatkov: **App Engine Datastore**, **Google Cloud SQL**, **Google Cloud Storage**.

Datastore
Google App Engine [45] zagotavlja porazdeljeno storitev za shranjevanje podatkov. Ko strežnik raste s številom uporabnikov in opravili, ki se izvajajo, hkrati raste tudi Datastore (podatkovno skladišče). Na voljo imamo dve možnosti za shranjevanje podatkov, ki se razlikujeta po dostopnosti in doslednosti: **visoko-replikacijska shramba** in **shramba tipa gospodar/suženj** (ni več podprta). Datastore ni tradicionalni model relacijske baze podatkov. Podatkovni objekti oziroma entitete imajo svojo množico lastnosti. S pomočjo poizvedb lahko pridobimo entite in jih uporabimo za nadaljnjo obdelavo. Entitete nimajo lastne sheme (ang. Schemaless Objects) in so določene s programsko kodo. Značilnosti Datastore so: ni načrtovanih izpadov, uporabljajo atomske transakcije, visoka razpoložljvost pisanj in branj, visoka konsistentnost pri branju in poizvedbah po prednikih in konsistentnost tudi za ostale poizvedbe. Za delo z Datastore Google nudi vmesnike Objectify, Twig in Slim3. Pri razvoju naše storitve na platformi Google App Engine bomo uporabljali Objectify vmesnik za poenostavljeno upravljanje z JDO (Java Data Objects).

BigTable
BigTable je visoko porazdeljena in skalabilna podatkovna baza, namenjena shranjevanju ogromnih količin podatkov, porazdeljenih po več tisoč strežnikih. Gre za multidimenzionalno in razpršeno polje. Tabela sestoji iz vrstic in stolpcev. Vsaka celica ima svoj časovni žig. BigTable si lahko predstavljamo kot objektno podatkovno bazo. Podatkovni objekt se imenuje entiteta (ang. Entity). Vsaka entiteta je določena z enoličnim identifikatorjem ali ključem.

Google Cloud SQL
Google Cloud Storage je MySQL podatkovna baza na Googlovem oblaku. Ima vse zmožnosti in značilnosti MySQL baze z nekaj izjemami. Za uporabo ne potrebuje posebne programske opreme in je tako idealna za manjša podjetja.

Google Cloud Storage
Google Cloud Storage je uporaben za shranjevanje in obdelavo velikih datotek. Prav tako pa nudi sezname s podatki za dostop (ACL's).

Blobstore
Blobstore omogoča shranjevanje podatkovnih objektov imenovanih Blobi (ang. Blob), ki močno presegajo dovoljeno velikost v Google Datastore. Navadno so to video ali slikovne datoteke. Blobe ustvarimo tako, da datoteko naložimo na strežnik preko HTTP zahteve. Za dostop do Blobov je potrebna uporaba Blobstore API-ja.

6.7.6.2 Google App Engine storitve

- **App Entity** - storitev, ki identificira aplikacijo, v kateri se izvaja.
- **Capabilities** - storitev, s katero lahko zaznamo izpade in načrtovane prekinitve.
- **Channel** - storitev, ki ustvari vztrajno povezavo med našo aplikacijo in strežnikom. Omogoča, da pošiljamo sporočila JavaScript odjemalcem, ne da bi uporabili izprševanje (ang. Polling). Uporabno za hitro obveščanje uporabnikov.
- **Google clouds endpoints** - storitev, ki sestoji iz orodij in knjižnic za poenostavljen dostop odjemalcev do podatkov iz ostalih aplikacij. Končne točke olajšajo kreiranje spletne hrbtenice med mobilnimi odjemalci.
- **Images** - storitev za upravljanje s slikovnimi datotekami. Storitev lahko med drugim spreminja dimenzije slikam, jih vrti okoli osi, zrcali itd.
- **Logs** - storitev nudi dostop do aplikacije in pripadajočega dnevnika (ang. Log). Obstajata dve vrsti vpisov v dnevnik: vpisi zahtev in vpisi, ki jih izvrši aplikacija.
- **Mail** - storitev lahko pošlje elektronsko sporočilo v imenu administratorja aplikacije ali v imenu uporabnika z Googlovim računom.
- **Memcache** - storitev, ki zagotavlja učinkovito in hitro delovanje skalabilnih aplikacij s shranjevanjem objektov v predpomnilnik.
- **Multitenancy** - omogoča implementacijo večnajemniškega modela.
- **Oauth** - protokol, ki dovoljuje aplikacijam uporabo tretje aplikacije, ne da bi s tem delili uporabniško ime in geslo.
- **Prospective search** - storitev za spremljanje podatkov, ki prihajajo v našo aplikacijo v realnem času. Razvijalec mora predhodno definirati poizvedbe, ki se lahko tekom izvajanja aplikacije istočasno izvedejo nad

podatki, ki prihajajo v aplikacijo.
- **Search** - storitev, ki nudi model za indeksiranje strukturiranih dokumentov.
- **Task Queues** - storitev, ki omogoča, da aplikacija opravi nalogo, ne da bi to zahteval uporabnik. Aplikacija v ozadju izvede neko delo, ki ga običajno imenujemo opravilo (ang. Task) glede na čakalno vrsto, ki smo jo pripravili. Obstajata dve različni čakalni vrsti: "push queues"in "pull queues".
- **URL fetch** - storitev, ki omogoča komunikacijo med aplikacijami preko URL naslova oziroma preko HTTP in HTTPS zahtev.
- **Users** - storitev, ki omogoča avtentikacijo uporabnikov na enega izmed treh načinov: preko Googlovih računov, lastnih računov v domeni ali OpenID identifikatorjev. Storitev lahko uporabnika vpiše, izpiše iz aplikacije ali pa preusmeri na ustrezno mesto v aplikaciji.
- **XMPP** - omogoča aplikacijam takojšnje pošiljanje in sprejemanje sporočil od in k uporabnikom, ki uporabljajo XMPP združljive storitve za takojšnje sporočanje (npr. Google Talk).

6.7.6.3 Zaračunavanje storitev

Z uporabo platforme Google App Engine plačamo toliko, kot porabimo (ang. Pay As You Go). Ni začetnih stroškov. Viri, ki jih naša aplikacija lahko zasede, se kontrolirajo iz naše strani. Uporaba do 1GB podatkov in pasovne širine za zadovoljitev potreb približno petih miljonov uporabnikov mesečno je brezplačna. Za večjo uporabo virov pa Google uporablja shemo zaračunavanja, ki jo najdete na njihovi uradni spletni strani. Če omenimo bistvene postavke:

- Shranjeni podatki v Datastore merjeni v GB/mesec = 0.18 $.
- Shranjeni podatki v Blobstore merjeni v GB/mesec = 0.15 $.
- Odhodna pasovna širina, merjena v GB = 0,12 $.
- Prejeta/odposlana elektronska sporočila = 0,0001 $ na prejemnika.

6.7.6.4 Zagotavlanje varnosti

Google App Engine ponuja sledeče varnostne mehanizme:

- Service Level Agreement (SLA) - da.
- Avtentikacija - uporabniško ime in geslo (Google Accounts), povezana identifikacija z uporabo OpenID, možnost uporabe lastnega ogrodja za avtentikacijo.

- Avtorizacija - uporabniške vloge (samo vlogi *User* ali *Administrator*), možnost uporabe odprtega standarda za avtorizacijo OAuth (Open Authorization).
- Integriteta - zagotovljena s pomočjo avtentikacije in avtorizacije ter enkripcije (SSL).
- Zaupnost - definirana v okviru varnostne politike in zagotovljena s pomočjo avtentikacije, avtorizacije ter enkripcije.
- Razpoložljivost - ponudnik v okviru SLA zagotavlja razpoložljivost 99.95 %.

6.7.7 AWS Elastic Beanstalk

AWS Elastic Beanstalk [46] je PaaS rešitev podjetja Amazon v sklopu Amazon Web Services. Amazon.com je ena najpomembnejših in najbolj obremenjenih spletnih strani na svetu. Zagotavlja veliko izbiro izdelkov s pomočjo infrastrukture, ki temelji na spletnih storitvah. Leta 2006 je Amazon svojo platformo spletnih storitev predstavil razvijalcem. S pomočjo virtualnih strojev Xen Hypervisors je Amazon omogočil uporabo zasebnih virtualnih strežnikov po celem svetu. Na teh strežnikih lahko delujejo skoraj vse vrste programske opreme, ki jo spremljajo podporne storitve s strani Amazona. Amazon ima za svojo rešitev na voljo več regij. Trenutno so na voljo naslednje regije: US West (Oregon), US West (Northern California), EU Ireland, Asia Pacific (Singapore), Asia Pacific (Tokyo), Asia Pacific (Sydney), South America (Sao Paulo) in GovCloud (US).

6.7.7.1 Shranjevanje podatkov

AWS v sklopu svojih storitev nudi shranjevanje podatkov na različne načine. Odločitev je prepuščena uporabnikom glede na potrebe aplikacij in cenovne zmožnosti. V nadaljevanju si bomo ogledali enote, kamor lahko shranjujemo podatke in baze, ki jih lahko uporabljamo.

Amazon Simple Storage Service (Amazon S3)
Amazon S3 je shranjevalni prostor, ki je namenjen internetnim aplikacijam. Zasnovan je tako, da razvijalcem lajša razvoj skalabilnih aplikacij. Amazon S3 nudi enostaven spletni vmesnik, ki ga lahko uporabimo za shranjevanje ali pridobivanje poljubne količine podatkov kadarkoli in od koderkoli. Značilnosti Amazon S3: podpora objektom velikost 1B–5TB na objekt, vsak objekt je shranjen v tako imenovanem vedru (ang. Bucket), vedro je lahko shranjeno v eno izmed več regij za optimizacijo latence in minimizacijo stroškov, obstajajo mehanizmi za avtentikacijo za preprečevanje nedovolje-

nega dostopa, uporaba enkripcije podatkov, uporaba standardnih SOAP in REST vmesnikov za delo s spletnimi aplikacijami.

Amazon Glacier
Amazon Glacier je nizkocenovni shranjevalni prostor, ki nudi varno in trajno shranjevanje arhivskih podatkov ter rezervnih kopij. Optimalen je za podatke, do katerih ni potreben pogost dostop in kjer čas za pridobivanje podatkov ni bistvenega pomena. Zakup 1 GB podatkov stane do 0,01 $ na mesec.

Amazon Elastic Block Store (EBS)
EBS nudi shranjevanje na nivoju blokov v povezavi z Amazon EC2 instancami. EBS shranjevalni prostor je vezan na omrežje, vendar neodvisen od instanc. Tako je zagotovljena visoka razpoložljivost, zanesljivost in predvidljivost prostora, ki ga zahteva instanca na napravi.

AWS Import/Export
AWS Import/Export pospešuje premik večjih količin podatkov iz oziroma na AWS. Za ogromne količine podatkov je ta način hitrejši in bolj učinkovit kot navaden prenos po omrežju. Je tudi cenejši kot nadgradnja omrežja.

Amazon Relational Database Service (Amazon RDS)
Amazon RDS je storitev, ki omogoča shranjevanje podatkov po principu relacijskih podatkovnih baz. Nudi enostavno upravljanje, postavljanje in skaliranje v oblaku. Temelji na modelu podobnem MySQL oziroma Microsoft SQL. To pomeni, da lahko orodja in knjižnjice, ki jih uporabljamo pri razvoju MySQL aplikacij, uporabimo tudi na Amazon RDS.

Amazon DynamoDB
Amazon DynamoDB je trenutno na voljo v beta verziji in še ni uradno predstavljena. Gre za izjemno hitro storitev, ki je enostavna in lahko shrani ali pridobi neomejeno količino podatkov ne glede na promet. Vsi podatki so shranjeni na SSD diske, ki so porazdeljeni preko treh regij za večjo razpoložljivost in zanesljivost. DynamoDB tabele nimajo predpisane sheme, torej ima lahko vsak objekt različno število lastnosti. Več podatkovnih tipov še dodatno povečuje fleksibilnost podatkovnega modela.

Amazon ElasticCache
Amazon ElasticCache je spletna storitev, ki lajša delo s predpomnilnikom v oblaku. Tudi ta storitev je zaenkrat še v beta razvojni stopnji. Storitev zvišuje učinkovitst aplikacij tako, da omogoča promet podatkov v in iz

predpomnilnika. Amazon ElasticCache trenutno podpira 5 različnih tipov, znotraj pa se delijo še na manjše enote: mikro tip, standarden tip, povečan tip, velik spominski tip (ang. High Memory), velik CPU tip.

Amazon SimpleDB (beta)
Amazon simple DB je fleksibilna nerelacijska podatkovna baza, ki ne potrebuje administracije. Razvijalci samo izvajajo poizvedbe in shranjujejo podatke, vse ostalo opravi Amazon SimpleDB sama. Za razliko od relacijskih podatkovnih baz Amazon SimpleDB omogoča prilagodljiv podatkovni model, ki se ne podreja vnaprej definirani shemi. To omogoča enostavno dodajanje novih podatkov, ki se tudi samodejno indeksirajo. Storitev se lahko uporablja tudi v navezi z Amazon S3, na primer za shranjevanje kazalcev na objekte, ki so shranjeni v Amazon S3.

6.7.7.2 Ostale AWS storitve

Amazon Elastic Compute Cloud (Amazon EC2)
Amazon EC2 je storitev, ki omogoča računanje v oblaku. Razvijalec lahko kreira navidezne naprave oziroma instance, na katerih se izvajajo njegove aplikacije. Izbiramo lahko med različnimi konfiguracijami navideznih naprav, izberemo lahko operacijski sistem, ki se bo izvajal na navidezni napravi, podatkovno bazo in različne zmogljivosti računalniških virov (procesorska moč, pomnilnik itd.). Amazon EC2 omogoča samodejno skalabilnost aplikacij, in sicer glede na pogoje, ki jih postavi razvijalec. Posebej si velja pogledati, kaj je EC2 instanca. Gre za osnovni element računanja na AWS oblaku. Instanco si lahko zamislimo kot navidezni strežnik, kjer delujejo naše aplikacije. Amazon Machine Image (AMI) je prednastavljena slika z operacijskim sistemom in ostalimi nastavitvami, preko katere ustvarjamo instance in delovno okolje. Poznamo več skupin instanc: splošno-namenske, optimizirane za računanje, optimizirane za shranjevanje, optimizirane za pomnenje, mikro instance in GPU instance (za paralelno računanje).

Amazon Simple Queue Service (SQS)
Amazon Simple Queue Service (Amazon SQS) je storitev, ki razvijalcem omogoča sporočilne vrste za zanesljivo shranjevanje sporočil, ki potujejo med komponentami njihovih aplikacij. Te komponente so lahko med seboj neodvisne in porazdeljene po omrežju, prav tako ta storitev ne zahteva, da so vedno razpoložljive.

Amazon Simple Notification Service (SNS)
Amazon SNS je storitev, ki omogoča pošiljanje obvestil v realnem času upo-

rabnikom, ki so prijavljeni na obveščanje preko različnih sporočilnih protokolov. Sporočila so poslana na mobilne naprave (npr. Android, Kindle in ostale naprave, povezane na internet) v obliki SMS ali pa elektronske pošte.

6.7.7.3 Zaračunavanje storitev

Amazon uporabo platforme Elastic Beanstalk zaračunava glede na količino porabljenih virov. V prvem letu je na voljo Free Tier opcija, ki omogoča brezplačno uporabo platforme z omejenimi sredstvi. AWS nudi ogromno nastavitev, zato ima tudi skrbno dodelan cenik zaračunavanja, ki se razlikujejo glede na regijo, v kateri se nahajajo in na katerem operacijskem sistemu delujejo.

- Mikro instance na zahtevo na EC2 lahko stanejo 0,02 $ na uro. Izjemno zmogljive instance pa vse do 4931 $ na uro.
- Uporaba platforme AWS Elastic Beanstalk na prednastavljenih (ang. Default) nastavitvah stane 35,7 $ na Linux sistemu in 42,47 $ na Windows sistemu.
- Uporaba Amazon S3 znaša 0,095 % na mesec za uporabo 1 GB prostora. Več prostora uporabljamo, manjša je cena na GB.
- Shranjevanje na Amazon Glacier stane 0,01 $ na GB.

6.7.7.4 Zagotavljanje varnosti

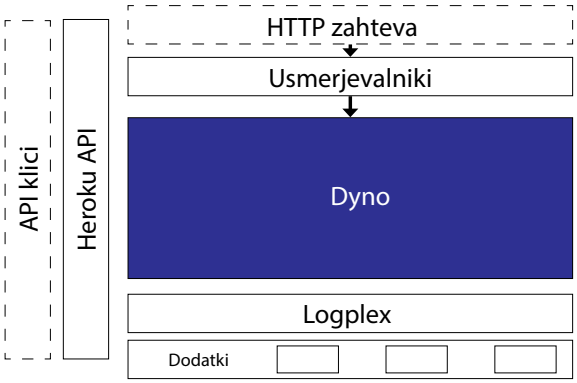
AWS Elastic Beanstalk ponuja sledeče varnostne mehanizme:

- Service Level Agreement (SLA) - opredeljen je za nekatere storitve.
- Avtentikacija - več načinov avtentikacije, ki jih nudi storitev IAM (avtentikacija na podlagi uporabniškega imena in gesla, uporaba generiranih parov ključev, varnostnih žetonov, povezane identifikacije (Federated Identity) itd.
- Avtorizacija - več načinov avtorizacije, ki jih nudi storitev IAM (npr. definiranje varnostnih skupin (Security Groups).
- Integriteta - zagotovljena s pomočjo avtentikacije in avtorizacije ter enkripcije (SSL).
- Zaupnost - definirana je v okviru varnostne politike in zagotovljena s pomočjo avtentikacije, avtorizacije ter enkripcije.
- Razpoložljivost - podana v je okviru SLA (večinoma 99.9 %), vendar ne za vse storitve.

6.7.8 Heroku

Heroku [47] je relativno nov PaaS ponudnik. Podjetje je bilo ustanovljeno leta 2007, danes pa gosti že preko tri milijone aplikacij. Za razvoj podpirajo sodobne programske jezike, in sicer Javo, Ruby, Scala, Play, Node.js in Clojure. Za nemoteno delovanje je Heroku platforma razdeljena na dva ključna elementa:

- **Usmerjevalnike** (ang. Routers) - zagotovljajo, da aplikacije prejemajo pravilne zahteve od uporabnikov.
- **Dyno vsebnike** – vsebovalni prostor kjer aplikacije delujejo.



Slika 6.8: Shema Heroku platforme.

6.7.8.1 Dyno

Ko postavimo aplikacijo na Heroku, ta deluje v vsebniku (ang. Container) imenovanem Dyno. Več kot ima naša aplikacija vsebnikov, več instanc lahko obdeluje zahteve. Vsak Dyno je povsem izoliran od ostalih. Dyno vsebniki so medsebojno neodvisni in jih lahko dodajamo in odstranjujemo poljubno. Značilnosti Dyno vsebnikov:

- **Elastičnost in skalabilnost** - število vsebnikov lahko v trenutku povečamo ali zmanjšamo, ne da bi vplivali na delovanje aplikacije.
- **Usmerjanje** - usmerjevalniki sledijo vsem razpoložljivim vsebnikom v omrežju in jim tudi dostavljajo HTTP zahteve.
- **Upravljanje** - omogočen je nadzor nad delujočimi vsebniki. Če pride do napake, se vsebnik izbriše in ustvari nov.

- **Porazdeljenost in redundanca** - vsebniki so porazdeljeni preko elastičnega okolja. Aplikacija, ki deluje na več Dyno vsebnikih na različnih lokacijah, bo v primeru okvare enega še vedno nemoteno delovala.
- **Izolacija** - vsak vsebnik je izoliran od ostalih, kar povečuje varnost, robustnost in zanesljivost.

6.7.8.2 Shranjevanje podatkov

PostgreSQL
Heroku za svoje delovanje nudi PostgreSQL odprtokodno aplikacijo za upravljanje s podatkovnimi bazami, ki jo sicer uporabljamo kot dodatek (ang. Add-on). PostgreSQL podpirajo vsi programski jeziki, ki so tudi sicer podprti. V zadnjih različicah se PostgreSQL lahko meri tudi z najbolj zmogljivimi komercialnimi rešitvami. Izjemna je v zanesljivosti delovanja, hitrosti in zmožnosti obdelave zelo velikih količin podatkov (več milijon zapisov). Z nekaterimi dodatki je mogoče obdelovati tudi prostorske podatke (PostGIS), slike (PostPic), indeksirati besedila (OpenFTS) itd.

ClearDB
ClearDB je zmogljiva, na napake odporna storitev, ki omogoča shranjevanje MySQL baz v oblak. Namesto razvijalca poskrbi za upravljanje s strežnikom, podvajanje podatkov, napredno shranjevanje in še posebno za napake. ClearDB deluje na geografsko ločenih zrcalnih podatkovnih gručah v kombinaciji z MySQL gospodar-gospodar (ang. Master-master) podvajanjem in z lastno SQL usmerjevalno tehnologijo, ki zaznava izpade in napake ter jih odpravlja s preusmerjanjem na nove instance.

6.7.8.3 Ostale storitve

Heroku omogoča množico dodatkov za razvoj aplikacij, med katerimi so nekateri tudi plačljivi.

Logplex
Logplex je odprtokodno orodje, ki nadomešča koncept dnevnika, saj zagotavlja stalen tok informacij, ki teče iz naše aplikacije od začetka do konca izvajanja. Še več, ta tok vsebuje kanoničen vir informacij o vsakem delu aplikacije, pa naj gre za usmerjevalnike ali enega izmed naših dyno vsebovalnikov. To pomeni, da lahko kot razvijalec lahko vidimo vse aktivnosti svoje aplikacije na enem mestu v trenutku. Po želji lahko ta tok tudi shranimo v datoteke.

6.7.8.4 Zaračunavanje storitev

Heroku zaračunava storitve po modelu Pay As You Go. Za vpogled v stroškovno strukturo naj navedemo, da:

- 1 Dyno stane 0,05 $ na uro.
- Začetna baza do 10 000 vrstic je brezplačna, do 10 milijonov vrstic stane 9,00 $.
- Produkcijske baze za naprednejše aplikacije stanejo od 50,00 $ – 6400,00 $ glede na potrebe.
- Dodatki se zaračunavajo posebej.

6.7.8.5 Zagotavljanje varnosti

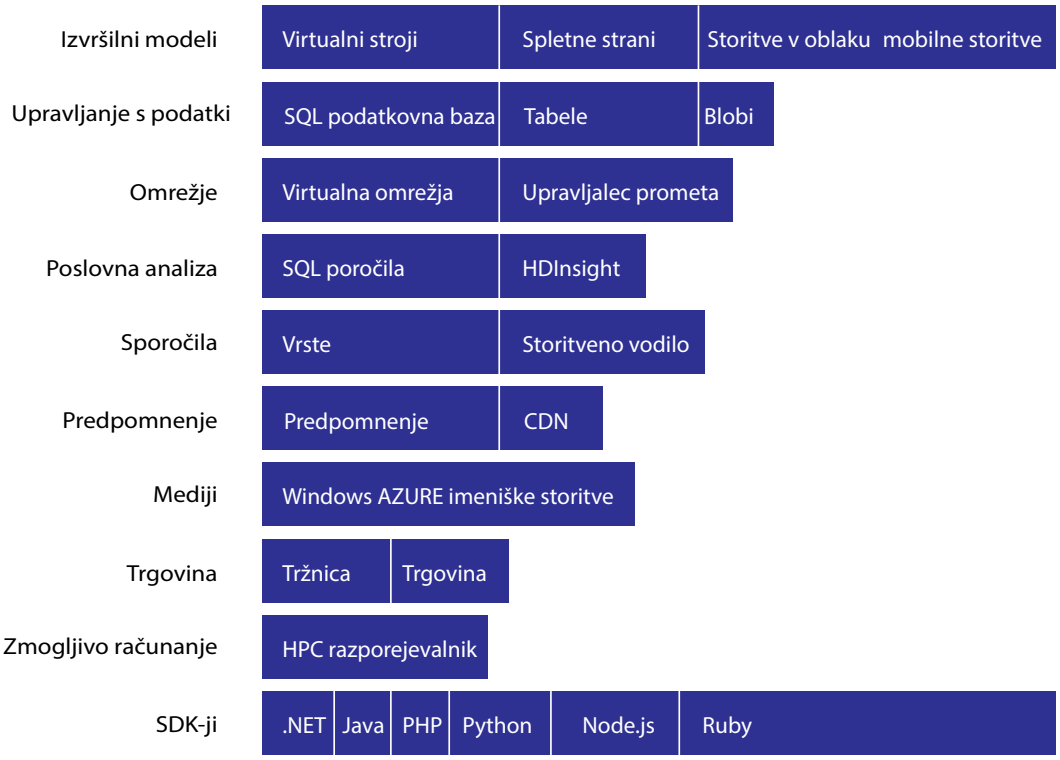
- Service Level Agreement (SLA) - ni objavil SLA sporazuma, objavljene so zaobljube uporabnikom. Strmijo k modelu, ki ga vodi Amazon.
- Avtentikacija - več načinov avtentikacije: e-pošta in geslo, API žeton ali SSH ključ.
- Avtorizacija - mogoča je Oauth avtorizacija tretjih aplikacij.
- Integriteta - zagotovljena s pomočjo avtentikacije in avtorizacije ter enkripcije (SSL).
- Zaupnost - definirana je v okviru varnostne politike in zagotovljena s pomočjo avtentikacije, avtorizacije ter enkripcije.
- Razpoložljivost - podana je v okviru zaobljube, kjer strmijo k 99,5 % razpoložljivosti.

6.7.9 Windows Azure

Windows Azure [48] je platforma v oblaku, ki jo je ponudil Microsoft za izgradnjo, uvajanje in upravljanje aplikacij ter storitev prek svetovnega omrežja z upravljanjem Microsoftovih podatkovnih središč. Omogoča enostavno, neomejeno skaliranje naših aplikacij. Je popolnoma avtomatizirana platforma, ki lahko zagotovi dodatne računalniške vire v trenutku. Za razumevanje, kaj vse nam Azure ponuja, si poglejmo sliko 6.9. Za potrebe tega dela bomo kratko opisali samo storitve za shranjevanje in računanje.

6.7.9.1 Shranjevanje podatkov

Windows Azure ponuja različne načine shranjevanja podatkov. Eden izmed njih je delovanje SQL strežnika na virtualnih strojih. Še več, nismo omejeni zgolj na relacijske baze, pač pa lahko uporabljamo tudi NoSQL tehnologije, kot je MongoDB. Imamo možnost razviti tudi lastno podatkovno bazo s



Slika 6.9: Sklopi Azure storitev.

pomočjo binarnega shranjevanja podatkov.

SQL podatkovna baza

SQL Azure oziroma SQL Database ponuja vse značilnosti tipičnih relacijskih podatkovnih baz. Gre za storitev, kjer upravljamo z našimi podatki, platforma pa poskrbi za upravljanje s strojno opremo in operacijskim sistemom. Podatki v SQL Database so porazdeljeni na več strežnikov za boljše delovanje (krajše latence).

Tabele

Za aplikacije, ki potrebujejo hiter dostop do vnešenih podatkov, vendar ne potrebujejo zapletenih SQL poizvedb, je Azure pripravil model tabel. Gre za nerelacijski ključ-vrednost (ang. Key-value) model baze. V tabele lahko shranimo množico lastnosti objekta, ki jih preko enolično določenega ključa lahko tudi preberemo. Gre za skalabilen, enostaven in predvsem cenejši model kot SQL shranjevanje.

Blob (Binary Large Object)
Blob je cenovno ugoden prostor za shranjevanje nestrukturiranih binarnih podatkov. En blob lahko doseže do 1 TB podatkov. V blobe shranjujemo multimedijske datoteke (v obliki video in avdio formatov).

6.7.9.2 Računske storitve

Za izvrševanje aplikacij ponuja Azure tri tipe storitev. Imenuje jih **navidezni stroji, spletne strani in storitve v oblaku.**
Pri navideznih strojih gre za slike Microsoft ali Linux operacijskega sistema. V osnovi gre za IaaS. Primerni so za uporabo REST spletnih storitev. Enostaven vmesnik za nastavitev navideznih strojev in intuitivna kontrola z REST API-ji omogočata hiter razvoj REST spletnih storitev.
Azure nudi okolje tudi za gostovanje spletnih strani. Gre za strežnike, kjer še vedno upravljamo z navideznimi stroji, vsa ostala administracija pa se dogaja avtomatsko (vzdrževanje, posodobitve). Dodajanje oziroma odstranjevanje navideznih strojev se dogaja dinamično glede na obremenitve.
Storitve v oblaku so namenjene za PaaS. Namenjene so storitvam, ki so zanesljive, skalabilne in cenovno ugodne. Tudi tu glavno vlogo igrajo navidezni stroji, ki pa jih ne ustvarjamo sami, ampak to storimo preko nastavitvene datoteke, platforma pa sama generira instance virtualnih strojev.

6.7.9.3 Zaračunavanje storitev

Microsoft ponuja uporabnikom dva načina zaračunavanja storitev svoje platforme: naročnine, kjer naročnik zakupi določeno količino virov in prihrani (za obdobje 6 ali 12 mesecev) ter standarden model zaračunavanja na podlagi porabe. Ob vpisu v Windows Azure dobimo 200 $ dobroimetja na naš račun. Kot pri platformi Google App Engine in Amazon je tudi tu stroškovna struktura skrbno razdelana. Če samo omenimo nekaj postavk:

- Izjemno nizko zmogljiva instanca navideznega stroja v oblaku stane 0,02 $ na mesec.
- Kapacitete diska za SQL baze se cenovno gibajo od 4,995 $ za 100 MB vse do 125,88 $ za 150 GB.

Na voljo je tudi enomesečna, omejena, brezplačna uporaba Windows Azure storitev.

6.7.9.4 Zagotavljanje varnosti

- Service Level Agreement (SLA) - da.

- Avtentikacija - multiFaktor avtentikacija, podpora Oauth 2.0, Sign-on Sign off protokol, Windows Live ID, certifikati.
- Avtorizacija - uporaba ObjectId identifikatorja in uporaba skupin uporabnikov.
- Integriteta - zagotovljena je s pomočjo avtentikacije in avtorizacije ter enkripcije (SSL).
- Zaupnost - definirana je v okviru varnostne politike in zagotovljena s pomočjo avtentikacije, avtorizacije ter enkripcije.
- Razpoložljivost - podana je v okviru SLA, in sicer 99,9 % razpoložljivosti.

6.8 Primerjava obravnavanih PaaS ponudnikov

Iz tabele 6.1 lahko vidimo razliko med manjšim ponudnikom PaaS, kot je Heroku, in tremi velikimi ponudniki. Razlika je opazna predvsem pri naboru dodatnih storitev, ki jih ostali ponujajo. Pri Amazonu je to razumljivo, saj Elastic Beanstalk obravnavamo v okviru AWS, ki v osnovi predstavlja IaaS in tako nudi ogromen nabor storitev. Podobno nudita tudi Google App Engine in Windows Azure. Heroku jih nudi precej manj. Večina storitev je dodanih oziroma dosegljivih preko dodatkov. Prav tako je razlika v podatkovnih bazah, kjer večji ponudniki nudijo podporo različnim vrstam podatkovnih baz. Heroku se osredotoča zgolj na PostgreSQL. Zanimivo je, da med vsemi ponudniki le Windows Azure nudi poleg zaračunavanja glede na porabo tudi naročnino. Gledano s stroškovnega vidika je za samo preizkušanje platform najprimernejši Google App Engine, saj nudi omejene vire za časovno nedoločeno obdobje. Izbira na prvi pogled ni enostavna. Po opravljeni podrobnejši analizi pa lahko ugotovimo, da večji ponudniki nudijo bolj prilagodljivo platformo. Izbira med njimi pa je odvisna še od ostalih kriterijev, ki jih tu nismo pregledali (npr. razvojno okolje, programski jeziki, možnost povezovanja z ostalimi platformami itd.)

	Google App Engine	**AWS**	**Heroku**	**Windows Azure**
Storitve za shranjevanje podatkov	Google Cloud Storage, BlobStore	Amazon S3, Glacier, EBS	ClearDB	Blob, Table Storage
Relacijska baze	Da - Cloud SQL	Da - Amazon RDS	DA - PostGre SQL	Da - SQL Azure
Nerelacijske baze	Da - Datastore, BigTable	Da - SimpleDB, DynamoDB	Ne	Da - MongoDB, Windows Azure Storage Table
Ostale storitve	Širok nabor storitev	Širok nabor storitev	Storitve v obliki dodatkov	Širok nabor storitev
Brazplačna uporaba	Da - omejeni viri, časovno neomejeno	Da - omejeni viri, 1 leto	Da - omejeni viri	Da - omejeni viri, 1 mesec
Zaračunavanje storitev	Po porabi	Po porabi	Po porabi	Po porabi, naročnina
SSL	Da	Da	Da	Da
Avtentikacija	Google računi, OpenID, lastno ogrodje	Ime/geslo, par ključev, žetoni, povezana identifikacija	Ime/geslo, žeton, SSH ključ	Multifaktor, Windowslive ID, certifikati

Tabela 6.1: Primerjava PaaS ponudnikov.

Poglavje 7

Razvoj RESTful spletnih storitev na platformah v oblaku

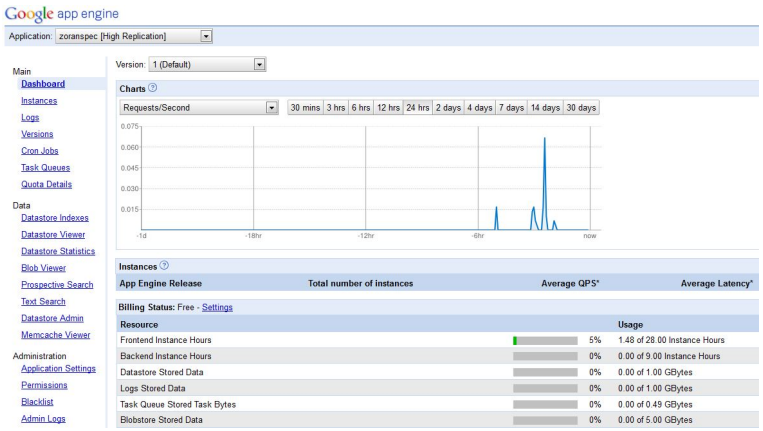
7.1 Uvod

V praktičnem delu bomo razvili dve podobni spletni aplikaciji na platformi Google App Engine in AWS Elastic Beanstalk ter ju razširili in jima dodali funkcionalnosti spletnih storitev. Na vsaki aplikaciji bomo implementirali tudi odjemalca, ki te storitve kliče. Razlog za izbiro omenjenih platform je, da sta trenutno med vodilnimi ponudniki PaaS rešitev in ker za osnovno ter enostavno uporabo omogočata brezplačno uporabo omejenih računalniških virov.
Ker gre za dokaj podobni aplikaciji s podobnimi funkcionalnostmi, bomo v nadaljevanju opisovali pristop, ki smo ga uporabili pri obeh aplikacijah oziroma spletnih storitvah. Na točke v katerih se implementacije razlikujejo, bomo posebej opozorili.

7.2 Opis aplikacije in storitev na platformi Google App Engine

Aplikacija na platformi Google App Engine bo predstavljala zelo enostaven **sistem za rezervacijo kart** (glej sliko 7.2). Aplikacija dovoli dva tipa uporabnikov, in sicer uporabnika Admin in uporabnika User. Admin bo lahko vnesel nove dogodke v sistem, jih posodobil ali izbrisal, torej tipične CRUD operacije. Uporabnik User pa bo lahko pregledal dogodke, ki so na voljo, in rezerviral določeno število kart. Ustvarjene rezervacije bo lahko po želji tudi brisal. Storitve sistema bodo nudile API-je za pregled dogodkov,

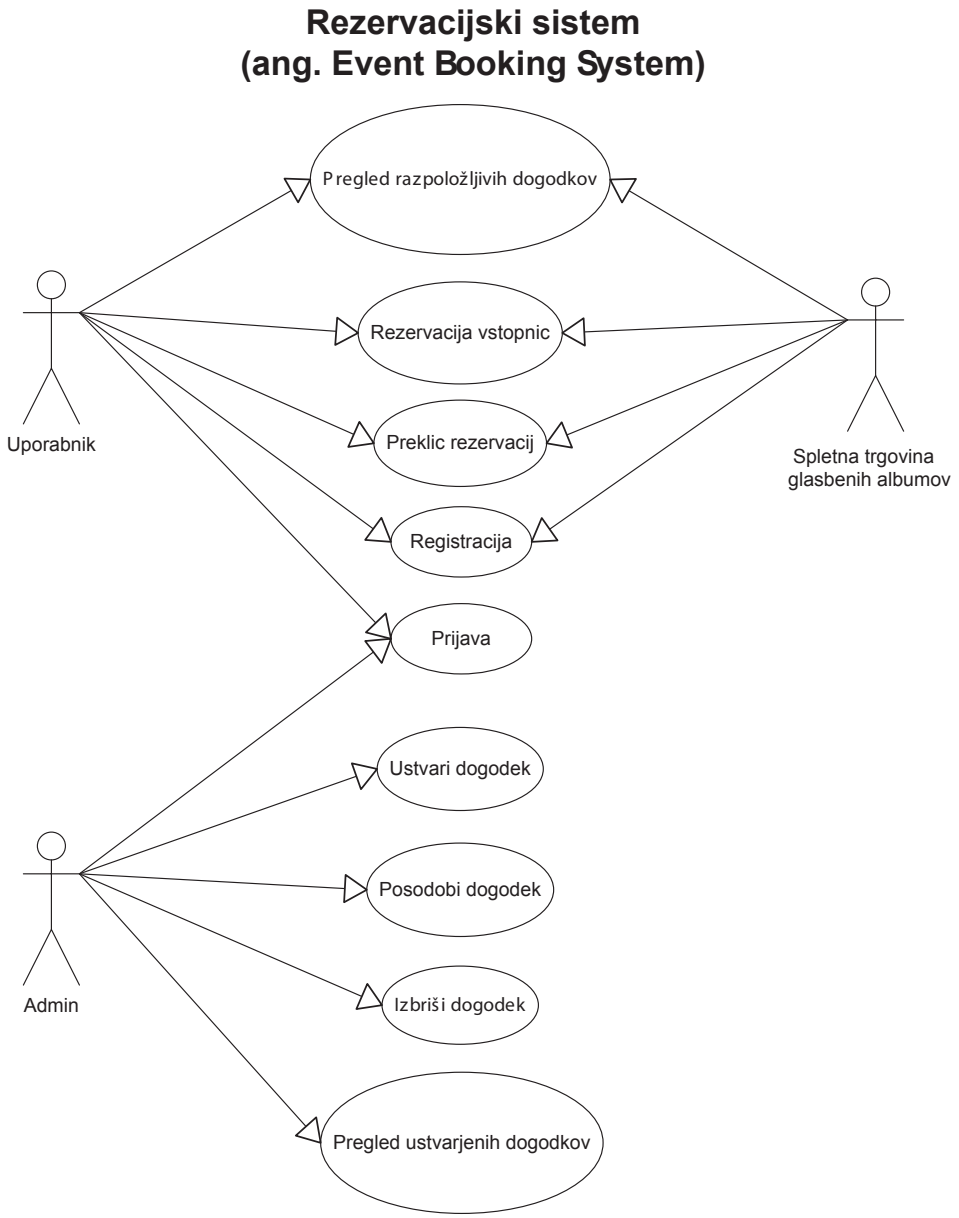
ki so na voljo, pregled dogodkov glede na izbranega izvajalca. Nudile bodo tudi možnost rezerviranja kart za dogodek, prav tako pa tudi brisanja. Slika 7.1 prikazuje stran za upravljanje z nastavitvami na platformi Google App Engine.



Slika 7.1: Stran za administracijo na platformi Google App Engine.

7.3 Opis aplikacije in storitev na AWS Elastic Beanstalk

Aplikacija na platformi AWS Elastic Beanstalk bo tehnično zelo podobna aplikaciji na platformi Google App Engine in bo nudila podobne storitve, vendar z vsebinsko razliko. Predstavljala bo enostaven **sistem za nakup glasbenih albumov**. Tudi v tem primeru imamo na voljo dva tipa uporabnikov. Admin bo lahko v sistem vnašal, posodabljal in brisal nove albume. Uporabnik User bo lahko pregledoval ponujene albume, lahko jih bo kupil ali pa preklical nakup. Storitve trgovine z albumi bodo nudile API-je za pregled vseh albumov, ki so na voljo, pregled albumov glede na izvajalca. Nudile bodo možnost nakupa albuma, prav tako pa tudi preklica. Slika 7.3 prikazuje okno za upravljanje z nastavitvami na platformi AWS Elastic Beanstalk.

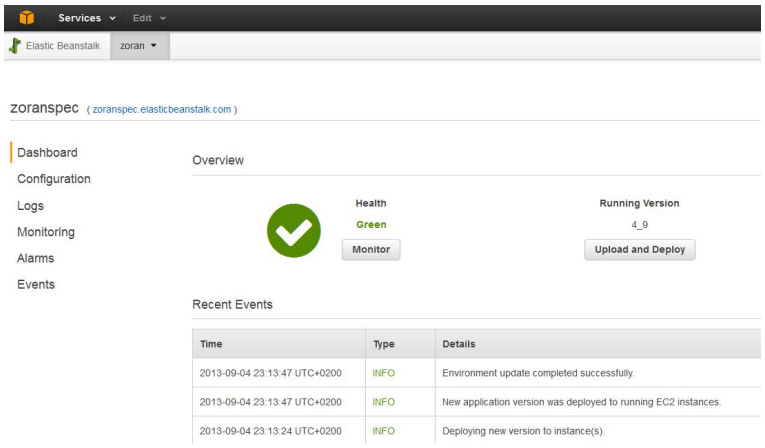


Slika 7.2: Diagram primerov uporabe aplikacije in storitev na platformi Google App Engine.

7.4 Uporabljene tehnologije

7.4.1 Programski jezik

Za razvoj obeh spletnih aplikacij smo uporabili programski jezik Java, in sicer verzijo 6. Za razvoj in nalaganje aplikacij na Google App Engine imamo na



Slika 7.3: Stran za administracijo na platformi AWS Elastic Beanstalk.

voljo nabor orodij App Engine Java SDK, za razvoj in postavljanje aplikacij pa AWS Java SDK. Oba nabora nudita vtičnik za IDE orodje Eclipse.

7.4.2 Razvojno okolje

Za razvoj aplikacij na platformi Google App Engine smo uporabili Eclipse 4 Juno z integriranim vtičnikom Google Plugin for Eclipse, ki omogoča razvoj, testiranje in nalaganje aplikacije na Google App Engine. Za razvoj aplikacij na platformi AWS Elastic Beanstalk smo uporabili NetBeans IDE 7.3, ker smo imeli probleme z integriranim vtičnikom za Eclipse.

7.4.3 Poslovno okolje

Za komunikacijo med spletno aplikacijo in spletnim strežnikom na platformi Google App Engine in AWS Elastic Beanstalk uporabljamo servlete (ang. Servlets). To so razredi, ki lahko procesirajo spletne zahteve in nanje odgovarjajo.
Posamezne sklope funkcionalnosti naše aplikacije smo implementirali v ločenih servletih, in sicer v paketu za prikaz obrazcev in paketu za obdelavo zahtev. Če pripišemo nekaj servletov za prikaz obrazcev:

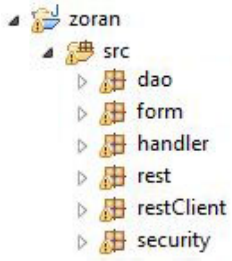
- CreateEventForm,
- UpdateEventForm,
- DeleteEventForm,
- MakeReservationForm,

- DeleteReservationForm,
- itn.

Servleti za procesiranje zahtev pa so:

- CreateEventHandler,
- UpdateEventHandler,
- DeleteEventHandler,
- MakeReservationHandler,
- DeleteReservationHandler,
- itn.

Slika 7.4 prikazuje drevesno strukturo javanskih razredov naše aplikacije. Omenili smo že paketa form in handler. Poleg imamo tudi paket DAO, ki vsebuje javanske razrede objektov in razrede za manipulacijo s podatkovno bazo in objekti. V paketu rest imamo razred, kjer so implementirane spletne storitve. Paket *restClient* vsebuje servlete za odjemalca, ki izkorišča storitve in jih ponuja aplikacija na platformi AWS Elastic Beanstalk. V paketu security najdemo dva razreda. Prvi razred je razširitev razreda *ContainerRequestFilter*, drugi razred pa služi kot pomožni razred za delo s filtri. Struktura javanskih razredov na platformi AWS Elastic Beanstalk je podobna.



Slika 7.4: Drevesna struktura javanskih razredov aplikacije za rezervacijo dogodkov.

7.4.4 Implementacija spletnih storitev

Za implementacijo spletnih storitev smo uporabili Jersey ogrodje. Za nudenje spletnih storitev imamo na voljo en razred, kjer so na voljo vse storitve. Izvorna koda 7.1 prikazuje implementacijo spletne storitve z GET metodo.

Izvorna koda 7.1: Primer spletne storitve z uporabo GET metode.

```
@GET
@Path("getEvent")
@Produces({ "application/xml" })
public Response getAllEvents(@QueryParam("artist") String
   artist) {
       Response.ResponseBuilder builder = Response
                              .status(Response.Status.
                                 BAD_REQUEST);

       DAOEvent ev = new DAOEvent();
       List<Event> list = new ArrayList<Event>();

       try {

              list = ev.showAllEventsl(artist);
              if (!list.isEmpty()) {
                     builder.status(Response.Status.OK);
                     GenericEntity<List<Event>> entity = new
                        GenericEntity<List<Event>>(
                                   list) {
                     };
                            return builder.entity(entity).
                               build();
              } else {
                     builder.status(Response.Status.NOT_FOUND)
                        ;
                     return builder.build();
              }
       } catch (NullPointerException e) {
              builder = Response.status(Response.Status.
                 NO_CONTENT);
              return builder.build();
       }
}
```

Na začetku imamo tri anotacije, in sicer z @GET napovemo, da se bo storitev izvršila ob klicu zahteve preko HTTP GET metode. @Path anotacija sporoča relativno pot za klic storitve, @Produces pa opredeljuje, v kakšni obliki bo predstavitev sredstva poslana (v našem primeru bo sredstvo predstavljeno kot XML dokument). Omeniti velja še anotacijo @QueryParam, s katero

napovemo, da lahko ob klicu storitve dodamo še poizvedbeni paramater (v našem primeru parameter *artist*).
Telo storitve je sestavljeno na enostaven način. Na začetku napovemo uporabo razreda *DAOEvent*, ki skrbi za manipulacijo podatkovne baze in objekta *Event* (ustvarjanje, brisanje in posodobitev dogodkov). Metoda *showAllEventsl(artist)* vrne seznam vseh dogodkov, ki ustrezajo poizvedbenemu parametru *artist*. V kolikor seznam dogodkov ni prazen, ga s pomočjo razreda GenereciEntity zgradimo v ustrezno predstavitev (*builder.entity(entity).build()*). V primeru da je seznam prazen, vrnemo ustrezno statusno kodo NOT FOUND. V celotnem procesu lovimo tudi izjeme, in sicer *NullPointerException*. V primeru da jo ujamemo, vrnemo statusno kodo NO CONTENT.

Izvorna koda 7.2: Primer spletne storitve z uporabo POST metode.

```
@POST
@Consumes({ "application/xml" })
@Path("/booking")
public Response addBooking(JAXBElement<Booking> booking) {

        Booking b = booking.getValue();
        URI ticketUri = uriInfo.getAbsolutePathBuilder()
                        .path(Long.toString(b.getEventID())).
                           build(new Object[0]);
        Response.ResponseBuilder builder = Response
                        .status(Response.Status.BAD_REQUEST);

        if (!b.equals(null)) {
                DAOBooking db = new DAOBooking();
                db.makeBooking(b.getEventID(), b.getUserName(),
                   b.getNrOfTickets());
                builder = Response.status(Response.Status.
                   CREATED).location(
                                   ticketUri);
        } else {
                builder.status(Response.Status.BAD_REQUEST);
        }
        return builder.build();
}
```

Izvorna koda 7.2 prikazuje primer spletne storitve s HTTP POST metodo. Anotacije pred metodo so podobne tistim pri izvorni kodi 7.1. Izjema je anotacija @Consumes, ki opredeljuje obliko predstavitve sredstva, ki jo naša

metoda lahko prejme. Ta metoda kot del zahteve prejme parameter JAXBElement. JAXBElement je del JAXB paketa za delo z XML dokumenti. Izvorna koda 7.3 prikazuje del Booking razreda, kjer smo uporabili JAXB anotacije. Metoda *getValue()* poskrbi za ustrezno preslikavo vrednosti iz XML predstavitve v dejanski razred. Razred URI omogoča gradnjo naslova, kjer se bo ustvarjeno sredstvo nahajalo in ki ga pošljemo kot del odgovora. V primeru da smo uspešno ustvarili sredstvo, vrnemo statusno kodo CREATED, v nasprotnem primeru pa BAD REQUEST.

Izvorna koda 7.3: Razred Booking.

```
@Entity
@XmlRootElement(name = "booking")
public class Booking implements Serializable {
        @Id
        private Long id;
        private String userName;
        private long eventID;
        private int nrOfTickets;

        public Booking(long id, String userName, long eventID,
            int nrOfTickets) {
                this.id = Long.valueOf(id);
                this.userName = userName;
                this.eventID = eventID;
                this.nrOfTickets = nrOfTickets;
        }

        public Booking() {
                this.id = null;
        }
```

Brisanje sredstva prikazuje izvorna koda 7.4. Tu je prikazana uporaba anotacije @PathParam, ki iz URL naslova pridobi uporabnikovo ime, preko poizvedbene anotacije @QueryParam pa pridobimo ID rezervacije (ang. Booking ID), ki jo želimo preklicati.

Izvorna koda 7.4: Brisanje sredstva z uporabo DELETE metode.

```
@DELETE
@Path("/deleteBooking/{usr}")
public Response deleteBooking(@PathParam("usr") String usr,
```

```
            @QueryParam("id") String id) {
        DAOBooking db = new DAOBooking();
        Long idl = Long.valueOf(id);
        DAOUser du = new DAOUser();
        Response.ResponseBuilder builder = Response
                        .status(Response.Status.BAD_REQUEST);

        try {
                User u = du.getUser(usr);
                if (u.getUsr_name().equals(null)) {
                        builder.status(Response.Status.
                            BAD_REQUEST);

                } else if (db.getBooking(idl).equals(null)) {
                                builder.status(Response.Status.
                                    NOT_FOUND);
                } else {
                                builder.status(Response.Status.OK)
                                    ;
                                db.deleteBooking(idl);
                }

        } catch (NullPointerException e) {
                        builder = Response.status(Response.Status
                            .NOT_FOUND);
                        return builder.build();
        } catch (java.lang.NumberFormatException e) {
                        builder = Response.status(Response.Status
                            .BAD_REQUEST);
                        return builder.build();
        }

        return builder.build();
}
```

7.4.5 Testiranje spletnih storitev

Testiranje RESTful spletnih storitev ni enostavna naloga. Na srečo pa obstajajo orodja tudi za to. Naše spletne storitve smo testirali z orodjem CURL, poizkusili pa smo tudi vtičnikom za Mozillo RestClient.

7.4.5.1 CURL

CURL [49] je odprtokodno konzolno orodje za prenašanje podatkov preko URL naslovov. Podpira vse bolj znane protokole, med drugimi FTP, HTTP, Gopher, HTTP, HTTPS, SMTP, SMTPS, telnet itd. Podpira tudi SSL certifikate, piškotke in različne načine avtentikacije.

Izvorna koda 7.5: Primer CURL ukaza za pridobivanje predstavitve sredstva.

```
curl -v http://zoranspec.appspot.com/api/v1/getEvent?artist=mi2
    -H "Accept: application/xml" -u test:test
```

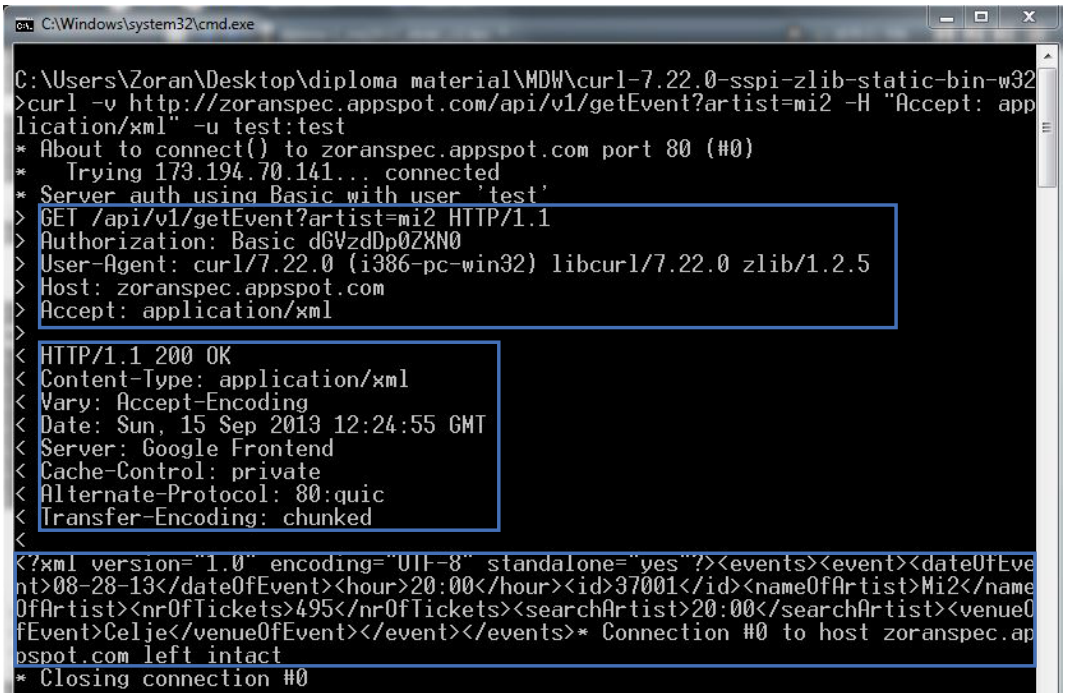
Izvorna koda 7.5 prikazuje sestavljen ukaz za testiranje spletne storitve, ki uporablja HTTP GET metodo, predstavljeno v izvorni kodi 7.1. Parameter -v omogoča lažje berljiv izpis. Sledi URL naslov, kjer se sredstva nahajajo. Parameter -H omogoči izpis zaglavja zahteve in odgovora. Sledi mu parameter *Accept: application/xml*, ki opredeljuje, v kakšni obliki bomo sprejeli predstavitev sredstva. Na koncu imamo še parameter -u, ki služi za avtentikacijo uporabnika s podanimi argumenti (uporabniško ime in uporabniško geslo). Na sliki 7.5 vidimo, kako orodje CURL deluje in kako izpiše rezultate glede na vnešen ukaz iz kode 7.5. Na izpisu smo označili tudi pomembne dele izpisa. Prvi obrobljen pravokotnik prikazuje HTTP zahtevo, ki smo jo tvorili. Drugi obrobljen pravokotnik prikazuje HTTP odgovor, ki smo ga prejeli. Tretji obrobljen pravokotnik pa prikazuje vsebino telesa, v našem primeru XML predstavitev sredstva. Z izvorno kodo 7.6 je prikazan še CURL ukaz za brisanje sredstva. V primerjavi s pridobivanjem sredstva imamo dodaten parameter, in sicer -X DELETE, ki napove, da bomo kot del ukaza uporabili metodo HTTP DELETE. Odziv orodja na ukaz je pričakovan (glej sliko 7.6). Ponovno smo zaradi lažje preglednosti označili, kaj je zahteva (zgornji pravokotnik) in kaj odgovor (spodnji pravokotnik).

Izvorna koda 7.6: Primer CURL ukaza za brisanje sredstva.

```
curl -v -X DELETE http://zoranspec.appspot.com/api/v1/
   deleteBooking/test?id=42001 -H "Accept: application/xml" -u
    test:test
```

7.4.5.2 RestClient vtičnik

Pri testiranju smo uporabljali tudi vtičnik RestClient [50], ki je na voljo za spletni brskalnik Mozilla. Podobno kot CURL je tudi RestClient zmogljivo orodje za testiranje in razhroščevanje RESTful spletnih storitev. Ima lepo dodelan in razumljiv uporabniški vmesnik. Največja pomankljivost je, da

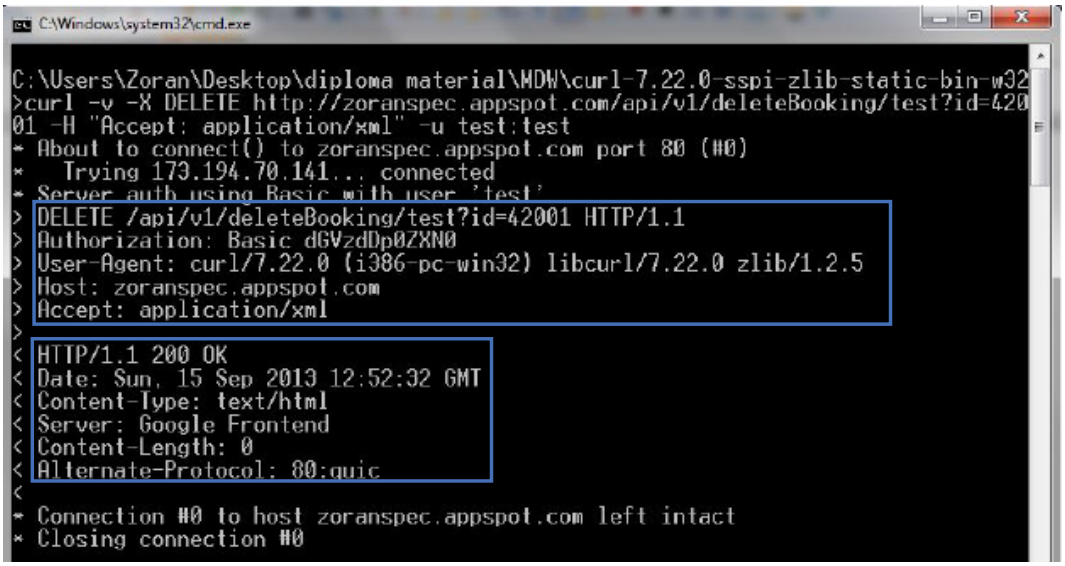


Slika 7.5: Obnašanje orodja CURL pri klicanju metode HTTP GET.

ob izvrševanju zahteve (slika 7.8) ne prikaže glave zahteve, prav tako ima na voljo manjši nabor mehanizmov za avtorizacijo in avtentikacijo. Slika 7.8 prikazuje odgovor v RestClient vtičniku. Kot vidimo, lahko izbiramo med predstavitvijo glave in tremi različnimi predstavitvami vsebine (raw, highlight in preview).

7.4.6 Odjemalec

Za klicanje spletnih storitev je potrebno implementirati odjemalca. Iz dela kode 7.7 vidimo, da je potrebno najprej definirati URL naslov, kjer se sredstvo, ki ga želimo prejeti, v našem primeru nahaja na naslovu http://zoranspec.appspot.com/api/v1/getEvent. URL naslovu smo dodali še ustrezen poizvedbeni parameter artist. Ker je za delovanje same aplikacije potrebna prijava v sistem, se podatki o uporabniku shranijo v sejo. Za tvorbo zahteve uporabniške podatke najprej pridobimo iz seje in jih s pomočjo *HTTPBasicAuthFiltra* kodiramo in dodamo odjemalcu v zahtevo. *WebResource* je razred, kamor vnesemo izbrani URL naslov. Razred *ClientResponse* pa predstavlja odgovor, ki ga dobimo, ko pokličemo spletno storitev.



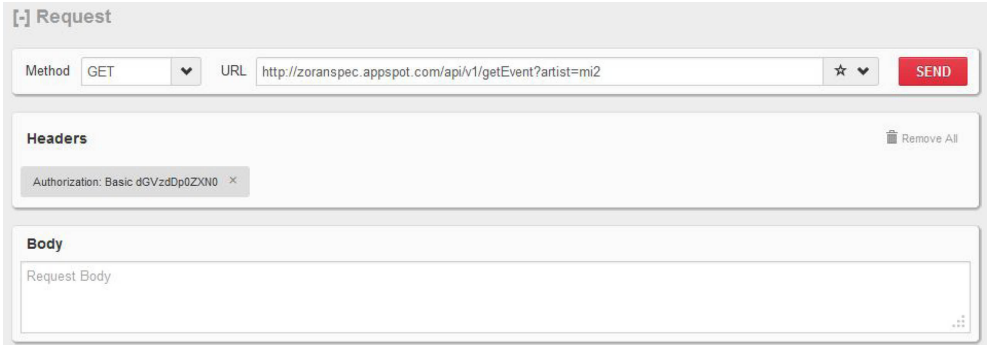
Slika 7.6: Obnašanje orodja CURL pri klicanju metode HTTP DELETE.

Izvorna koda 7.7: Izsek kode za ustvarjanje odjemalca.

```
    HttpSession session = request.getSession(true);
User u=(User) session.getAttribute("user");
String un=u.getUsr_name();
String psw=u.getPsw();
String URL="http://zoranspec.appspot.com/api/v1/getEvent?
   artist=";
String search=request.getParameter("search");
search=search.replaceAll(" ", "%20");
URL=URL.concat(search);
Client client = Client.create();
     client.addFilter(new HTTPBasicAuthFilter(un, psw));
WebResource webResource = client.resource(URL);
ClientResponse cr = webResource.accept("application/xml")
              .get(ClientResponse.class);
String xml=cr.getEntity(String.class);
```

7.5 Avtentikacija

Za avtentikacijo spletnih storitev smo uporabljali HTTP Basic Authentication metodo. Na strani strežnika to implementiramo tako, da v aplikacijo s



Slika 7.7: Obnašanje orodja RestClient pri klicanju zahtev.

Slika 7.8: Obnašanje orodja RestClient pri branju odgovora.

pomočjo deskriptorja web.xml vpeljemo in implementiramo razred filter, ki je razširitev razreda *ContainerRequestFilter* za prestrezanje zahtev. Izvorna koda 7.8 prikazuje tak filter. Bistvo takšnega filtra je, da preverja glavo zahteve in išče avtorizacijski ključ ter njegovo vrednost. Ta vrednost predstavlja BASE_64 kodirano uporabniško ime in geslo, ki ga v filtru dekodiramo, nato pa pošljemo poizvedbo v našo bazo uporabnikov za validacijo in avtentikacijo. V kolikor zahteva ne vsebuje zahtevane glave (Authorization vrednost) ali pa vrednosti, ki pripadajo avtorizacijskemu ključu ne zadoščajo kriterijem oziroma uporabniško ime in geslo ne opravita avtentikacije, se v odgovor spletne storitve zapiše status UNAUTHORIZED.

Izvorna koda 7.8: Implementacija razreda ContainerRequestFilter.

```
public ContainerRequest filter(ContainerRequest
    containerRequest) throws WebApplicationException {
        //GET, POST, PUT, DELETE, ...
        String method = containerRequest.getMethod();
        String path = containerRequest.getPath(true);
        //pridobivanje avtentikacijskih parametrov iz glave
            HTTP zahtev
        String auth = containerRequest.getHeaderValue("
            authorization");
        // uporabnik nima pravic
        if(auth == null){
            throw new WebApplicationException(Status.
                UNAUTHORIZED);
        }
        //lap : uporab. ime in geslo
        String[] lap = BasicAuth.decode(auth);
        //prijava ne uspe
        if(lap == null || lap.length != 2){
            throw new WebApplicationException(Status.
                UNAUTHORIZED);
        }
        DAOUser da=new DAOUser();
        Boolean valid=da.validate(lap[0], lap[1]);
        //prijava ne uspe
        if(valid.equals(false)){
            throw new WebApplicationException(Status.
                UNAUTHORIZED);
        }
        return containerRequest;
    }
```

7.6 Nastavitvene datoteke

7.6.1 Postavitveni deskriptor spletne aplikacije

Postavitveni deskriptor spletne aplikacije se nanaša na nastavitveno datoteko aplikacij ali sistema, ki jih postavimo na strežnik. V Java EE postavitveni spletni deskriptor (web.xml datoteka) opisuje, kako so komponente, moduli ali aplikacije postavljeni. Uporabljamo ga za opis nastavitev vsebovalnika,

kjer naša aplikacija deluje, za nastavitve varnosti ali pa za kakšne druge nastavitve. Za opis uporabljamo XML sintakso. V primeru spletnih aplikacij ga vedno najdemo v WEB-INF imeniku. Primer izseka takšnega deskriptorja si lahko ogledamo na sliki 7.9. Iz deskriptorja vidimo, da bo naša aplikacija delovala v *ServletContainerju*, ki ga nudi Jersey, prav tako smo definirali filter, ki se bo sprožil ob vsaki zahtevi. Spodnji del prikazuje, kako opišemo servlet v deskriptorju. Vsakemu servletu dodelimo ime, opredelimo, v katerem paketu se nahaja, nato pa imenujemo še njegovo URL preslikavo (ang. URL Pattern).

Izvorna koda 7.9: Izsek iz web.xml deskriptorja.

```
<servlet>
        <servlet-name>ServletAdaptor</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.
           ServletContainer</servlet-class>
        <init-param>
                <param-name>com.sun.jersey.config.property.
                   packages</param-name>
                <param-value>rest</param-value>
        </init-param>
            <init-param>
                        <param-name>com.sun.jersey.spi.container.
                           ContainerRequestFilters</param-name>
                        <param-value>security.AuthFilter</param-
                           value>
                </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletAdaptor</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>

  <servlet>
    <description/>
    <display-name>LoginForm</display-name>
    <servlet-name>LoginForm</servlet-name>
    <servlet-class>form.LoginForm</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginForm</servlet-name>
```

```
    <url-pattern>/LoginForm</url-pattern>
  </servlet-mapping>
```

7.6.2 Appengine-web.xml

Ta datoteka med drugim vsebuje registrirano identifikacijsko številko naše aplikacije, številko verzije aplikacije, nastavitve seje itd. Datoteka appengine-web.xml je specifična za Google App Engine in ni del standarda Java Servlets. Koda 7.10 prikazuje takšno datoteko.

Izvorna koda 7.10: Appengine-web.xml datoteka.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>zoranspec</application>
  <version>1</version>
  <threadsafe>true</threadsafe>
  <static-files>
    <include path="**" />
    <include path="**.nocache.*" expiration="0s" />
    <include path="**.cache.*" expiration="365d" />
    <exclude path="**.gwt.rpc" />
  </static-files>
  <system-properties>
    <property name="java.util.logging.config.file" value="WEB-
       INF/logging.properties"/>
  </system-properties>
  <sessions-enabled>true</sessions-enabled>
</appengine-web-app>
```

7.7 Podatkovna baza

7.7.1 Podatkovni model

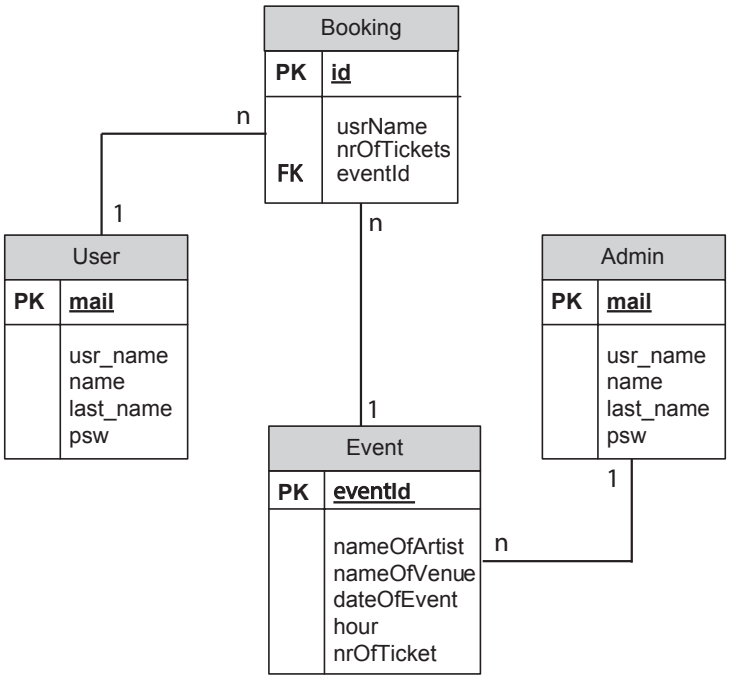
Za delo z aplikacijo na platoformi Google App Engine smo potrebovali naslednje entitetne tipe, ki so prikazani v podatkovnem modelu na sliki 7.9:

- Event,
- Booking,
- User,

- Admin(razširjen razred User).

Za delo z aplikacijo na platformi AWS Elastic Beanstalk pa:

- Album,
- Purchase,
- User,
- Admin(razširjen razred User).



Slika 7.9: Podatkovni model aplikacije na platformi Google App Engine.

7.7.2 Podatkovna baza

Točki, v katerih se aplikaciji razlikujeta, sta podatkovni bazi in upravljanje z njimi.

7.7.2.1 Google App Engine

Za delo z **Google App Engine** Datastore smo uporabili Objectify knjižnico, ki omogoča delo z JPA (Java Persistance Architecture). Gre torej za presli-

kavo objektov v entitete in lažje upravljanje z njimi. Koda 7.11 prikazuje primer uporabe Objectify z GET metodo:

Izvorna koda 7.11: Uporaba knjižnice Objectify za pridobivanje sredstev.

```
public Event getEvent(Long id) {
        Objectify ofy;
        ObjectifyService.register(Event.class);
        ofy = ObjectifyService.begin();
        Event e = (Event) this.ofy.get(Event.class, id.
           longValue());
        return e;
}
```

Pred izvršitvijo same metode moramo sprva v razredu ObjectifyService registrirati objekt, nad katerim bomo izvajali poizvedbe. V zgornjem primeru torej *ObjectifyService.register(Event.class)*. V nadaljevanju pa samo kličemo metode. V primeru na sliki gre za klic GET metode nad razredom Event, kjer je primarni ključ dogodka enak vrednosti id.

7.7.2.2 AWS Elastic Beanstalk

Za delo z **AWS SimpleDB** nismo uporabili JPA knjižnic, ker niso bile podprte oziroma stabilne. SimpleDB uporablja sintakso, ki je precej podobna SQL. Sestoji iz domen, ki vsebujejo elemente (ang. Items). Vsak element pa je sestavljen iz imena elementa in atributov. Pred začetkom dela s SimpleDB bazo je potrebno ustvariti domeno. *sdb = new AmazonSimpleDBClient(credentials); sdb.createDomain(new CreateDomainRequest("Album"));* nato pa lahko z njo upravljamo. Slika 7.12 prikazuje ustvarjanje objekta in pisanje v bazo.

Izvorna koda 7.12: Primer pisanja v simpleDB.

```
public void createAlbum(Album a){

        this.getDomain();
        PutAttributesRequest patr=new PutAttributesRequest();
        List<ReplaceableAttribute> atr= new ArrayList<
           ReplaceableAttribute>();
        String name=UUID.randomUUID().toString();
        patr.setItemName(name);
        patr.setDomainName("Album");
```

```
                atr.add(new ReplaceableAttribute().withName("
                    Artist").withValue(a.getArtist()));
                atr.add(new ReplaceableAttribute().withName("
                    Title").withValue(a.getName()));
                atr.add(new ReplaceableAttribute().withName("
                    ReleaseDate").withValue(a.getReleaseDate()));
                atr.add(new ReplaceableAttribute().withName("
                    Price").withValue(Float.toString(a.getPrice()
                    )));
                atr.add(new ReplaceableAttribute().withName("
                    NrOfAlbums").withValue(Integer.toString(a.
                    getNrOfproducts())));

                patr.setAttributes(atr);
            sdb.putAttributes(patr);
}
```

Koda 7.13 prikazuje nekaj primerov poizvedb v simpleDB.

Izvorna koda 7.13: Primeri poizvedb v simpleDB.

```
String q = "select * from Album";
String q = "select Artist from Album";
String q = "select * from Purchase where User='test'";
```

Orodje SimpleDB Tool
SimpleDB Tool [51] je orodje za testiranje in razhroščevanje programske kode pri delu s podatkovno bazo simpleDB. Na voljo je kot vtičnik za spletni brskalnik Mozilla.

7.8 REST API

7.8.1 REST API aplikacije na platformi Google App Engine

Vsi API-ji zahtevajo uporabo HTTP Basic avtentikacije. Osnovni URL naslov je http://zoranspec.appspot.com/api/v1. V nadaljevanju bomo predstavili nekaj API-jev, ki so povzeti v tabeli 7.1.

- **Prikaži dogodke glede na izvajalca** - namenjen je pridobivanju sredstva Event.
  Primer URL naslova: *zoranspec.appspot.com/api/v1/getEvent?artist=Mi2*

Metoda	Pot	Poizvedbeni parametri	Statusne kode	Predstavitev sredstva
GET	getEvent	artist	OK, NOT FOUND, NO CONTENT	XML
GET	getEventID	id	OK, NOT FOUND, NO CONTENT	XML
GET	getBooking	usr	OK, NOT FOUND	XML
POST	booking	usr	OK, NOT FOUND	XML
DELETE	delete Booking	usr, id	OK, NOT FOUND, BAD REQUEST	XML

Tabela 7.1: Primeri API-jev za rezervacijski sistem.

- **Prikaže dogodek glede na ID** - namenjen je pridobivanju dogodka glede na podani ID.
  Primer URL naslova: *zoranspec.appspot.com/api/v1/getEventID?id=500*
- **Prikaži vse rezervacije uporabnika** - namenjen je pridobivanju rezervacij glede na uporabnika.
  Primer URL naslova: *zoranspec.appspot.com/api/v1/getBookings?usr=test*
- **Ustvari rezervacijo** - namenjen je ustvarjanju rezervacije.
  Primer URL naslova: *zoranspec.appspot.com/api/v1/getBookings?usr=test*
- **Izbriši rezervacijo** - namenjen je brisanju rezervacije.
  Primer URL naslova: *zoranspec.appspot.com/api/v1/deleteBooking/test/id=5003*

7.8.2 REST API aplikacije na platformi AWS Elastic Beanstalk

Vsi API-ji zahtevajo uporabo HTTP Basic avtentikacije. Osnovni URL naslov je http://zoranspec.elasticBeanstalk.com/api/v1. V nadaljevanju bomo predstavili nekaj bolj uporabnih API-jev (glej tabelo 7.2).

- **Prikaži albume glede na izvajalca** - namenjen je pridobivanju sredstva Album.
  Primer URL naslova: *zoranspec.elasticBeanstalk.com/api/v1/getAlbum?artist=Mi2*
- **Prikaže albume glede na ID** - namenjen je pridobivanju albuma glede na podani ID.
  Primer URL naslova: *zoranspec.elasticBeanstalk.com/api/v1/getAlbumID?id=5001*
- **Prikaži vse nakupe uporabnika** - namenjen je pridobivanju informacij o nakupu uporabnika.

Metoda	Pot	Poizvedbeni parametri	Statusne kode	Predstavitev sredstva
GET	getEvent	artist	OK, NOT FOUND, NO CONTENT	XML
GET	getAlbumID	id	OK, NOT FOUND	XML
GET	getPurchases	usr	OK, NOT FOUND	XML
POST	purchase	usr	CREATED, NO CONTENT	XML

Tabela 7.2: Primeri API-jev za nakup albumov.

Primer URL naslova: *zoranspec.elasticBeanstalk.com/api/v1/getPurchases?usr=test*

- **Prikaži vse nakupe uporabnika** - namenjen je pridobivanju informacij o nakupu uporabnika.
  Primer URL naslova: *zoranspec.elasticBeanstalk.com/api/v1/getPurchases?usr=test*

Poglavje 8

Sklepne ugotovitve

Računalništvo v oblaku postaja tisti aspekt IT tehnologije, ki presega vsa pričakovanja. Vsa večja podjetja, ki se ukvarjajo z razvojem komercialne programske opreme za široko rabo, v zadnjem letu ali dveh selijo svoje rešitve v oblak. Če vzamemo Google in Gmail kot tipično storitev, lahko vidimo, da je to že stalnica, ki se je niti nismo zavedali. Danes temu procesu sledijo tudi drugi večji igralci v tej industriji, npr. podjetje Adobe s svojimi orodji za obdelavo multimedijskih vsebin ali pa podjetja, ki razvijajo igre. Na drugi strani pa najdemo kup poslovnih rešitev, ki niso namenjene množici "per se", ampak delovanju organizacij na splošno (razni zdravstveni informacijski sistemi, interni sistemi korporacij, razne e-storitve itd.).
V diplomskem delu smo obravnavali tisti del računalništva v oblaku, ki se nanaša na platforme. Danes imamo v zasebnem PaaS oblaku možnost minimizirati stroške tako razvijalcev kot tudi podjetij na nivoju javnih oblakov. Prvi imajo za razvoj končno na voljo skoraj neomejeno količino virov za razvoj zmogljivih aplikacij. Podjetja pa se prvič lahko posvetijo le svojim poslovnim modelom, medtem ko je IT struktura prepuščena ponudniku oblaka. To prinaša prihranke, tako vzpostavitvene, kot tudi nadaljne (stroški razvoja, vzdrževanja, nadgrajevanja itd.). V okviru PaaS smo preleteli bistvene ponudnike na trgu. Za razvoj aplikacij smo uporabili Google App Engine in AWS Elastic Beanstalk. Ugotovili smo, da nekatere knjižnice niso popolnoma stabilne oziroma smo imeli probleme s postavitvijo aplikacij v oblak. Pri platformi Google App Engine je največji problem predstavljala najnovejša knjižnica Objectify za delo z JPA, ki kot kaže, ni bila popolnoma integrirana s platformo, zato smo uporabili objectify 3.0.1. Razvoj aplikacije za platformi Google App Engine je sicer potekal brez večjih problemov, saj je dokumentacija precej dobro dodelana, skupnost pa velika in aktivna. Nekoliko več problemov nam je povzročala aplikacija na AWS Elastic Beanstalku, saj je Amazon med našim razvojem spremenil svoj vmesnik in s tem tudi

ukinil vse instance virtualnih strojev, ki smo jih že nastavili. Prav tako je model zaračunavanja tako podrobno razdeljen, da je težko pravilno nastaviti parametre za brezplačno uporabo.
V splošnem lahko ocenimo, da je sam razvoj na obeh platformah enostaven, hkrati pa tudi precej omejujoč. Vsaka platforma nudi svoje storitve preko API-jev, ki jih lahko uporabljamo, vendar s tem povečujemo svojo odvisnost od ponudnika. Sama narava naših aplikacij je bila precej osnovno zasnovana, zato kompleksnejših problemov niti nismo pričakovali, saj smo uporabljali večinoma le CRUD operacije in enostavne poizvedbe v bazah. Dotaknili smo se tudi RESTful spletnih storitev. Gre za tip storitev, ki razvijalcem z dokaj jasnimi arhitekturnimi principi lajša razvoj spletnih storitev. Spoznali smo, da REST arhitekturni stil nudi uniformen vmesnik, preko katerega dostopamo do storitev. Prav tako smo podrobneje preleteli HTTP protokol, metode, ki ga spremljajo, in pa URI standard. Zgoraj naštete tehnologije so nepogrešljiv del praktično vsakega zahteva-odgovor sistema še posebej pa RESTful storitev. V nadaljevanju smo si ogledali tudi tri pogosteje uporabljana JAX-RS ogrodja za izdelavo javanskih REST storitev. Kot smo pričakovali, razlike med njimi niso velike, saj vsebujejo iste funkcionalnosti, le implementacije so različne. Tudi sam razvoj spletnih storitev je potekal brez večjih problemov, saj tako Google App Engine kot AWS Elastic Beanstalk podpirata RESTful spletne storitve (v našem primeru s pomočjo Jersey ogrodja).
Ugotovimo lahko, da razvoj aplikacij in spletnih storitev v oblaku ni težak, vendar pa v primeru naše aplikacije manj primeren, saj nismo izkoriščali zmogljivosti, ki jih oblak nudi. Takšna oblika je precej bolj primerna za aplikacije z velikim številom zahtev oziroma za poslovne aplikacije.
Vse kaže, da računalništvo v oblaku ni modna muha in se v prihodnosti pričakuje njegovo povečanje. Varnostna komponenta je edina večja pomankljivost, ki jo avtorji navajajo. V kolikor bo v bodoče prišlo do izboljšave te komponente oziroma do večjega zaupanja, pa lahko celo pričakujemo popolno prevlado. Z računalništvom v oblaku je nepremostljivo povezan tudi storitveno orientiran razvoj sistemov in s tem razvoj spletnih storitev, bodisi SOAP ali RESTful. Vsi omenjeni koncepti pa skupaj prinašajo kombinacijo, ki je začetek 21. stoletja obrnila na glavo in porušila model računalništva, ki smo ga uporabljali do sedaj.

Slike

Tabele

Izvorna koda

Literatura

[1] (2013) Rajive Joshi. *"Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA"*. Dostopno na:
http://www.rti.com/whitepapers/Data-Oriented_Architecture.pdf

[2] (2013) Arcitura. *"Service-Oriented Architecture"*. Dostopno na:
http://serviceorientation.com/whatissoa/service_oriented_architecture

[3] (2013) IBM. *"Introduction to SOA governance"*. Dostopno na:
http://www.ibm.com/developerworks/library/ar-servgov/

[4] (2013) Arcitura. *"Service orientation principles"*. Dostopno na:
http://serviceorientation.com/serviceorientation/index

[5] (2013) OpenGroup. *"Service orientated architecture"*. Dostopno na:
http://www.opengroup.org/soa/source-book/soa/soa.htm

[6] (2013) TutorialsPoint. *"What are Web Services"*. Dostopno na:
http://www.tutorialspoint.com/webservices/why_web_services.htm

[7] (2013)W3C. *"Web Services Architecture"*. Dostopno na:
http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis

[8] (2013)W3C. *"SOAP: Messaging Framework"*. Dostopno na:
http://www.w3.org/TR/soap12-part1/

[9] (2013)Brian Suda. *"SOAP Web Services"*. Dostopno na:
http://suda.co.uk/publications/MSc/brian.suda.thesis.pdf

[10] (2013)Roy Fielding. *"Representational State Transfer (REST)"*. Dostopno na:
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[11] (2013) Wiki. *"Representational state transfer"*. Dostopno na: http://en.wikipedia.org/wiki/Representational_state_transfer

[12] (2013)Wiki. *"Uniform resource identifier"*. Dostopno na: http://en.wikipedia.org/wiki/Uniform_resource_identifier

[13] (2013) W3C. *"Dereferencing HTTP URIs "*. Dostopno na: http://www.w3.org/2001/tag/doc/httpRange-14/2007-05-31/HttpRange-14

[14] (2013) Wiki. *"XML"*. Dostopno na: http://en.wikipedia.org/wiki/XML

[15] (2013) W3C. *"Web Application Description Language"*. Dostopno na: http://www.w3.org/Submission/wadl/

[16] (2013) Json. *"Introducing JSON"*. Dostopno na: http://www.json.org/

[17] (2013) Wiki. *"Internet media type"*. Dostopno na: https://en.wikipedia.org/wiki/Internet_media_type

[18] (2013) IANA. *"MIME Media Types"*. Dostopno na: http://www.iana.org/assignments/media-types

[19] (2013) W3C. *"Content negotiation"*. Dostopno na: http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html

[20] (2013) Kioskea. *"The HTTP protocol"*. Dostopno na: http://en.kioskea.net/contents/273-the-http-protocol

[21] (2013) W3C. *"Method definitions"*. Dostopno na: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[22] (2013) W3C. *"Status Code definitions"*. Dostopno na: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[23] (2013) Wiki. *"Software Framework"*. Dostopno na: http://en.wikipedia.org/wiki/Software_framework

[24] (2013) Wiki. *"web application Framework"*. Dostopno na: http://en.wikipedia.org/wiki/Web_application_framework

[25] (2013) 3Scale. *"Your guide to the Internet Business (R)evolution"*. Dostopno na:
`http://www.3scale.net/wp-content/uploads/2012/06/What-is-an-API-1.0.pdf`

[26] (2013) Cloud Computing Use Case Discussion Group. *"Cloud Computing Use Cases White Paper"*. Dostopno na:
`http://opencloudmanifesto.org/Cloud_Computing_Use_Cases_Whitepaper-4_0.pdf`

[27] (2013) Martin Fowler. *"Richardson Maturity Model"*. Dostopno na:
`http://martinfowler.com/articles/richardsonMaturityModel.htmlf`

[28] (2013) William Martinez Pomares. *"Evaluating REST Frameworks Part 1: A Maturity Model"*. Dostopno na:
`http://martinfowler.com/articles/richardsonMaturityModel.html`

[29] (2013) JAX-RS. *"Java API for RESTful Services (JAX-RS)"*. Dostopno na:
`http://jax-rs-spec.java.net/`

[30] (2013) Oracle. *"JAX-RS: Java API for RESTful Web Services"*. Dostopno na:
`download.oracle.com/otn-pub/jcp/jaxrs-1.1-mrel-eval-oth-JSpec/jax_rs-1_1-mrel-spec.pdf`

[31] (2013) IBM. *"RESTful Web services with Apache Wink, Part 3: Apache Wink and the REST"*. Dostopno na:
`http://www.ibm.com/developerworks/library/wa-apachewink3/`

[32] (2013) Sun. *"Jersey"*. Dostopno na:
`https://jersey.java.net/`

[33] (2013) JBoss. *"RESTEasy"*. Dostopno na:
`http://www.jboss.org/resteasy)`

[34] (2013) Restlet. *"RESTLet"*. Dostopno na:
`http://restlet.org/`

[35] (2013) Wiki. *"Cloud Computing"*. Dostopno na:
`en.wikipedia.org/wiki/Cloud_computing`

[36] (2013) IBM. *"Cloud computing for the enterprise: Part 1: Capturing the cloud"*. Dostopno na:
http://www.ibm.com/developerworks/websphere/techjournal/0904_amrhein/0904_amrhein.html

[37] (2013) The National Institute for Standards and Technology. *"The NIST Definition of Cloud"*. Dostopno na:
http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[38] Archie Reed, Stephen G. Bennet. *"Dark Linings: A Concise Guide to Cloud Computing"*, Prentice Hall, 2010, pogl. 1. Dostopno na:
http://ptgmedia.pearsoncmg.com/images/9780131388697/samplepages/013138869X.pdf

[39] Barrie Sosinsky. *"Cloud Computing Bible"*, Wiley Publishing, 2011, pogl. 1. Dostopno na:
http://www.gbv.de/dms/ilmenau/toc/638282322.PDF

[40] (2013) Technet. *"Cloud Computing: Data Privacy in the Cloud"*.Dostopno na:
http://technet.microsoft.com/en-us/magazine/jj554305.aspx

[41] (2013) SharePoint. *"Pay-as-you-Go with PaaS"*. Dostopno na:
http://blog.creative-sharepoint.com/2012/10/pay-as-you-go-with-paas/

[42] (2013) Force.com. *"Force.com"*. Dostopno na:
http://www.salesforce.com/platform/

[43] (2013) IBM. *"Approaches for enabling multi-tenancy"*. Dostopno na:
http://www.ibm.com/developerworks/webservices/library/ws-multitenantpart2/index.html

[44] (2013) CloudProvidersUSA. *" Rise of PaaS: Trends to Watch in 2013"*. Dostopno na:
http://www.cloudproviderusa.com/rise-paas-trends-watch-2013/

[45] (2013) Google. *"Google App Engine"*. Dostopno na:
https://appengine.google.com/

[46] (2013) Amazon. *"Amazon Web Service"*. Dostopno na:
aws.amazon.com

[47] (2013) Heroku. *"Heroku"*. Dostopno na:
`https://www.heroku.com`

[48] (2013) Microsoft. *"Microsoft Azure"*. Dostopno na:
`http://www.windowsazure.com`

[49] (2013) Curl. Dostopno na:
`http://curl.haxx.se/`

[50] (2013) RestClient. Dostopno na:
`http://restclient.net/`

[51] (2013) SimpleDB Tool. Dostopno na:
`http://code.google.com/p/sdbtool/`