Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Sandi Šemrov

Razvoj migracijskih adapterjev za aplikacije SaaS

diplomsko delo

univerzitetni študijski program prve stopnje
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: 00123/2013
Datum: 09.04.2013

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

Kandidat: **SANDI ŠEMROV**

Naslov: **RAZVOJ MIGRACIJSKIH ADAPTERJEV ZA APLIKACIJE SAAS**
MIGRATION ADAPTER DEVELOPMENT FOR SAAS APPLICATIONS

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Preučite metode in postopke za migracijo in integracijo aplikacij in se pri tem osredotočite na aplikacije SaaS (Software-as-a-Service). Na nivoju integracije podatkov analizirajte pristope za migracijo podatkov med različnimi aplikacijami. Pripravite zasnovo ogrodja za izvajanje migracijskih adapterjev in na primeru realizirajte migracijo podatkov med dvema SaaS aplikacijama.

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 Sandi Šemrov, z vpisno številko **63090142**, sem avtor diplomskega dela z naslovom :

Razvoj migracijskih adapterjev za aplikacije SaaS

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaž Branko Jurič,
- 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 23. septembra 2013 Podpis avtorja:

Kazalo

Povzetek

V diplomskem delu so predstavljene metode in postopki za migracijo in integracijo aplikacij, pri čemer je glavni poudarek na aplikacijah, ki so ponujene kot storitve – SaaS (Software-as-a-Service). Na nivoju integracije podatkov so analizirani pristopi za migracijo podatkov med različnimi aplikacijami tipa SaaS. Pripravljena je zasnova ogrodja za izvajanje migracijskih adapterjev in na primeru realizirana migracija podatkov med dvema aplikacijama tipa SaaS. Razložen in prikazan je tudi razvoj adapterjev za konkretni aplikaciji tipa SaaS oz. storitvi, Salesforce in Do. Pri vsakem konkretnem primeru razvoja migracijskega adapterja za aplikacijo tipa SaaS je prikazan postopek za uvoz in izvoz podatkov.

Ključne besede: migracijski adapter, integracija, migracija, aplikacija tipa SaaS

Abstract

This thesis presents methods and procedures for migration and applications integration, with the main focus on applications that are offered as a service – SaaS (Software-as-a-Service). At the level of data integration we analyze approaches for data migration between different types of SaaS applications. We design a framework for executing migration adapters and show an example of data migration between two SaaS applications. We then describe the development of adapters for two specific SaaS applications, Salesforce and Do. Finally, we demonstrate the procedure for importing and exporting data from SaaS applications.

Keywords: migration adapter, integration, migration, SaaS application

Poglavje 1

Uvod

Računalništvo v oblaku postaja vedno bolj pomemben koncept v računalništvu in s tem dobivajo vedno večjo veljavo aplikacije, infrastrukture in platforme v obliki storitev (SaaS – Software as a Service, IaaS – Infrastructure as a Service in PaaS – Platform as a Service; splošno storitev XaaS).

V diplomskem delu se osredotočamo na razvoj migracijskih adapterjev za aplikacije tipa SaaS. Migracijski adapter omogoča uvoz in izvoz podatkov na posamezno aplikacijo tipa SaaS. Za uvoz in izvoz podatkov uporablja programske vmesnike (ang. Application Programming Interface) posameznih aplikacij tipa SaaS. Preko programskih vmesnikov se adapter najprej avtenticira na aplikaciji, in nato dostopa do podatkov uporabniškega računa avtenticiranega uporabnika.

Migracije podatkov s posameznih aplikacij lahko povežemo tako, da aplikacija uporabi podatke drugih aplikacij. Takrat govorimo o integraciji aplikacij.

Migracijski adapterji so bili razviti v okviru projekta Sintesis. Aplikacija je dostopna kot e-storitev in omogoča:

- pregled, ocenjevanje in izbiro najustreznejše aplikacije SaaS oz. storitve XaaS,

- integracijo dveh aplikacij SaaS oz. storitev XaaS,

- migracijo podatkov med dvema aplikacijama SaaS oz. storitvama XaaS,
- izdelavo varnostne kopije podatkov (ang. back-up) iz aplikacije SaaS oz. storitve XaaS.

V diplomskem delu je predstavljeno ogrodje, ki skrbi za nalaganje in izvajanje migracijskih adapterjev. Prikazana sta tudi dva primera razvoja adapterja za konkretni aplikaciji tipa SaaS, Salesforce in Do. Salesforce spada v skupino aplikacij CRM (Customer Relationship Management), Do pa v skupino aplikacij za vodenje projektov.

Vedno več aplikacij se izvaja v oblaku in se jih ponuja kot storitev preko lahkih oz. tankih odjemalcev (ang. thin client). Običajno lahko do njih dostopamo preko spletnih brskalnikov.

Vsi migracijski adapterji so bili razviti v programskem jeziku Java. Podatki s posameznih aplikacij tipa SaaS oz. storitev so v oblikah JavaScript Object Notation (JSON) ali Extensible Markup Language (XML).

V poglavju 2 je razložena splošna integracija aplikacij in nato še aplikacij tipa SaaS. Predstavljeni so izzivi pri migraciji in integraciji, ki so se pojavili med razvojem aplikacije. Tu je omenjen skupni podatkovni model, ki omogoča integracijo aplikacij tipa SaaS. V poglavju 3 je predstavljen načrtovalski vzorec adapterjev. Prikazano je ogrodje adapterjev, ki poskrbi za izvajanje adapterjev. Opisana je tudi implementacija adapterjev, ki prikaže kako se pripravi razred z metodama za uvoz in izvoz podatkov. V poglavju 4 je predstavljen konkreten razvoj migracijskih adapterjev za dve aplikaciji tipa SaaS, in podan je primer, ki prikazuje spremembe strukture XML, ki so potrebne za migracijo podatkov z ene aplikacije tipa SaaS na drugo.

Poglavje 2

Integracija

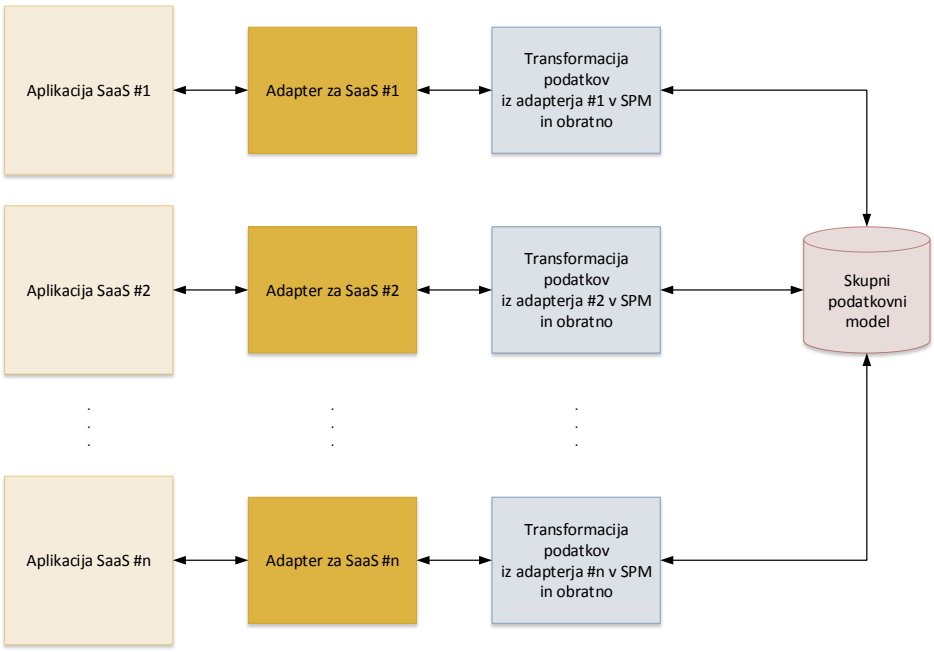
Besedo integracija razumemo, kot "povezovanje posameznih enot, delov v večjo celoto" [1]. Pod izrazom integracija aplikacij mislimo na integracijo podatkov in funkcij med dvema aplikacijama. To pomeni, da združujemo podatke, funkcije posameznih aplikacij v eno, večjo, skupno količino podatkov in funkcij. Angleški izraz za integracijo aplikacij je Enterprise Application Integration (EAI) in predstavlja integracijsko ogrodje, sestavljeno iz mnogih tehnologij in storitev [9]. Tudi za aplikacijami, ki uporabljajo migracijske adapterje, stoji veliko tehnologij. Uporabljene javanske tehnologije so opisane v poglavju 3.2, tehnologije za pretvorbo in strukturiranje podatkov pa v poglavju 2.1.

2.1 Integracija aplikacij SaaS

Software as a Service pomeni ponujanje programske opreme oz. aplikacije kot storitev. To pove, da sta sama aplikacija in podatki, ki jih aplikacija uporablja, centralizirana v oblaku (ang. Cloud Computing). Za dostop do aplikacije oz. podatkov se običajno uporablja lahki oz. tanki odjemalec (ang. thin client), in sicer prek spletnega brskalnika [13].

Integracijo aplikacij tipa SaaS smo realizirali, kot je prikazano na sliki

[1] Povzeto po SSKJ.



Slika 2.1: Shema integracije aplikacij in migracije podatkov.

2.1. Uvedli smo skupni oz. enotni podatkovni model, kjer se zberejo podatki s posameznih aplikacij tipa SaaS oz. storitev. Skupni podatkovni model je sestavljen iz posameznih modelov, shem, v katere se preslikajo podatki s posameznih aplikacij tipa SaaS oz. storitev. Na ta način lahko podatke z različnih aplikacij oz. storitev predstavimo v enotni obliki.

Posamezni adapter je napisan za vsako aplikacijo tipa SaaS oz. storitev posebej, saj se aplikacije oz. storitve in njihovi programski vmesniki razlikujejo med seboj. Migracija podatkov poteka tako, da jih adapter pridobi s posamezne storitve in jih pošlje v transformacijo. Transformacije se lahko vršijo na več načinov. Eden najpomembnejših in najpogostejših načinov je izvajanje transformacij s pomočjo jezika Extensible Stylesheet Language (XSL). Ta omogoča transformacijo dokumenta v obliki XML v drugi dokument, ki je prav tako zapisan v obliki XML, vendar ima (vsaj po večini)

drugačno strukturo. Druga dva načina sta še poizvedovalni jezik za XML – XQuery in pa programske transformacije. Transformacije poskrbijo, da se podatki, ki ustrezajo shemi adapterja, pretvorijo v sheme skupnega podatkovnega modela. Na ta način so podatki predstavljeni enotno. Nato lahko izvedemo transformacijo iz sheme skupnega podatkovnega modela v shemo nekega drugega adapterja, ki podatke naloži na drugo aplikacijo oz. storitev. Tako omogočimo prenos oz. migracijo podatkov z ene aplikacije tipa SaaS na drugo.

2.2 Izzivi pri integraciji in migraciji podatkov

Največji izziv pri migraciji in integraciji predstavlja prenos nekompatibilnih podatkov z ene aplikacije oz. storitve na drugo. Zato smo se domislili enotnega podatkovnega modela, kjer so podatki z različnih aplikacij predstavljeni na enak način. V enotnem oz. skupnem podatkovnem modelu so podatki shranjeni v obliki XML. Predstavljeni so s shemami XML - XML Schema Definition (XSD). Prednost shem XML je ta, da se lahko razširjajo. Tako lahko shemo, ki predstavlja strukturo npr. naslovov elektronske pošte, vključimo v shemo, ki predstavlja uporabnikove kontakte, in v shemo, ki predstavlja uporabnikove organizacije.

Adapterjev vhod in izhod je tudi predstavljen s shemo XML, ki je za oba enaka. Vhod adapterja pomeni podatke v obliki XML, ki pridejo preko transformacij iz skupnega podatkovnega modela. Izhod adapterja pa pomeni podatke v obliki XML, ki jih adapter pridobi iz storitve in vrne v sistem. Tako se transformacije vršijo iz izhodne sheme XML adapterja v shemo skupnega podatkovnega modela in iz skupnega podatkovnega modela v vhodno shemo adapterja, ki pa je ista kot izhodna.

V želji, da bi bilo čim več aplikacij tipa SaaS oz. storitev med seboj kompatibilnih, smo transformacije razdelili na več delov. Tako sta lahko aplikaciji, ki nista istega tipa[2] kompatibilni. Pri večini vseh vrst aplikacij tipa

[2]Tip aplikacije lahko opredelimo kot skupino aplikacij, v katero spada ta aplikacija;

SaaS oz. storitev najdemo uporabnikove kontakte. V kontakte spadajo kontaktne informacije oseb, ki jih je uporabnik zabeležil v svojem uporabniškem računu. Pri aplikacijah CRM so to npr. stranke, s katerimi smo opravili posel. Na podlagi delov transformacij lahko tako migriramo oz. integriramo le del podatkov, ki jih ponuja aplikacija tipa SaaS oz. storitev.

Pri migraciji je morda edini od večjih izzivov prenos podatkov z ene aplikacije tipa SaaS oz. storitve na drugo. Pri integraciji pa obstaja več izzivov. Eden glavnih je vprašanje, kdaj prenesti podatke, in drugi, katere podatke prenesti. Integracijo smo med dvema aplikacijama oz. storitvama realizirali tako, da so se najprej uvozili podatki z obeh storitev. Oba uvoza morata biti uspešna. Če nista, celotna integracija ne uspe, enako velja za izvoz. Če ta na eno od storitev ne uspe, tudi celotna integracija ne uspe. V sistemu morata torej najprej biti oba uspešna uvoza podatkov in nato se izvede izvoz podatkov na drugo aplikacijo. Uvoz iz prve aplikacije gre na izvoz druge in uvoz iz druge aplikacije gre na izvoz prve. Pred izvozom podatkov na posamezno aplikacijo tipa SaaS oz. storitev mora adapter nujno preveriti, če zapisi na storitvi že obstajajo. Če adapter tega ne bi počel, bi se ob vsaki izvedbi migracije podatki podvajali. Pri konfiguraciji migracije oz. integracije imamo možnost periodičnega izvajanja, da se podatki avtomatsko prenašajo med aplikacijami oz. storitvami in nam shranjenih konfiguracij ni potrebno izvajati ročno.

npr. aplikacije CRM (Customer Relationship Management), aplikacije za podporo uporabnikom.

Poglavje 3

Adapterji

Besedo adapter si lahko razlagamo kot "pripravo za prilagoditev dveh, za medsebojno delovanje sicer neprilagojenih priprav"[1]. Če besedo adapter postavimo v kontekst integracije aplikacij tipa SaaS oz. storitev, si jo lahko razlagamo kot program, algoritem, ki poskrbi za prilagoditev teh dveh aplikacij. V okviru projekta Sintesis smo razvili štiri vrste adapterjev:

- integracijske adapterje,
- migracijske adapterje,
- adapterje za varnostno kopiranje podatkov in
- adapterje za spremljanje storitev.

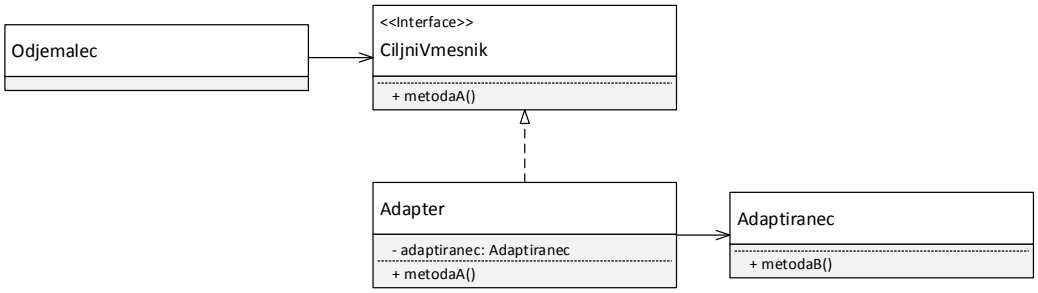
V tem delu se bomo osredotočili na migracijske adapterje.

Migracijski adapter si v tem delu lahko predstavljamo, kot javanski razred oz. skupek javanskih razredov, ki skrbijo za prenos podatkov iz/na aplikacijo tipa SaaS.

[1] Povzeto po SSKJ.

3.1 Načrtovalski vzorec Adapter

Načrtovalski vzorec adapterja (ang. Adapter Design Pattern) je opisan v knjigi *Design Patterns: Elements of Reusable Object-Oriented Software* [5]. Opisuje vzorec, ki se uporablja za prilagoditev komunikacije dveh nekompatibilnih tipov [1]. Je strukturni vzorec, ki opredeljuje način za ustvarjanje odnosov med razredi. Uporablja se za vzpostavitev povezave med obema, sicer nezdružljivima tipoma. To stori tako, da se *adaptiranec* ovije z razredom, ki podpira vmesnik, katerega zahteva odjemalec. Na sliki 3.1 je prikazan diagram UML (Unified Modeling Language), ki prikazuje razrede načrtovalskega vzorca.



Slika 3.1: Diagram UML načrtovalskega vzorca Adapter.

- **Odjemalec** je razred, ki zahteva uporabo nekompatibilnega tipa. Pričakuje interakcijo s tipom, ki razširja *CiljniVmesnik*, vendar mi želimo, da uporabi razred *Adaptiranec*.
- **CiljniVmesnik** je pričakovani vmesnik za razred *Odjemalec*. Ni nujno, da je vmesnik, lahko je tudi razred katerega deduje razred *Adapter*.
- **Adaptiranec** vsebuje funkcionalnosti, ki jih zahteva *Odjemalec*. Ni prilagodljiv z vmesnikom *CiljniVmesnik*, kar je pričakovano.
- **Adapter** je razred, ki zagotavlja povezavo med nekompatibilnima razredoma *Odjemalec* in *Adaptiranec*. *Adapter* razširja vmesnik *CiljniVme-*

snik in vsebuje zasebno instanco razreda *Adaptiranec*. Ko *Odjemalec* pokliče metodo *metodaA* na vmesniku, *Adapter* poskrbi, da se pokliče *metodaB* na instanci razreda *Adaptiranec*.

Primer implementacije omenjenih razredov v programskem jeziku Java, je ponazorjen v izseku kode 3.1.

Izsek kode 3.1: Primer implemetacije omenjenih razredov in vmesnika.

```
1  public class Odjemalec
2  {
3    private CiljniVmesnik cilj;
4    public Client(CiljniVmesnik cilj)
5    {
6      this.cilj = cilj;
7    }
8    public void zahtevaj()
9    {
10     cilj.metodaA();
11   }
12 }
13 public interface CiljniVmesnik
14 {
15   void metodaA();
16 }
17 public class Adaptiranec
18 {
19   public void metodaB()
20   {
21     //izvorna koda metode B
22   }
23 }
24 public class Adapter implements CiljniVmesnik
25 {
26   Adaptiranec adaptiranec = new Adaptiranec();
27   public void metodaA()
28   {
29     adaptiranec.metodaB();
30   }
31 }
```

Namen vzorca adapter je prilagoditev enega razreda drugemu razredu ter na tak način zagotoviti kompatibilnost med razredoma, ki sta bila pred uporabo vzorca nekompatibilna. V našem primeru smo vzorec adapter uporabili

za zagotovitev kompatibilnosti med različnimi aplikacijami tipa SaaS ter njihovimi podatki. To smo storili tako, da smo pretvorili razred, ki zahteve prvega razreda pretvori v ustrezno obliko za drugi razred, in jih pošlje drugemu razredu. Podatke smo predstavili v enotni obliki, katere lahko uporabi poljubna aplikacija tipa SaaS.

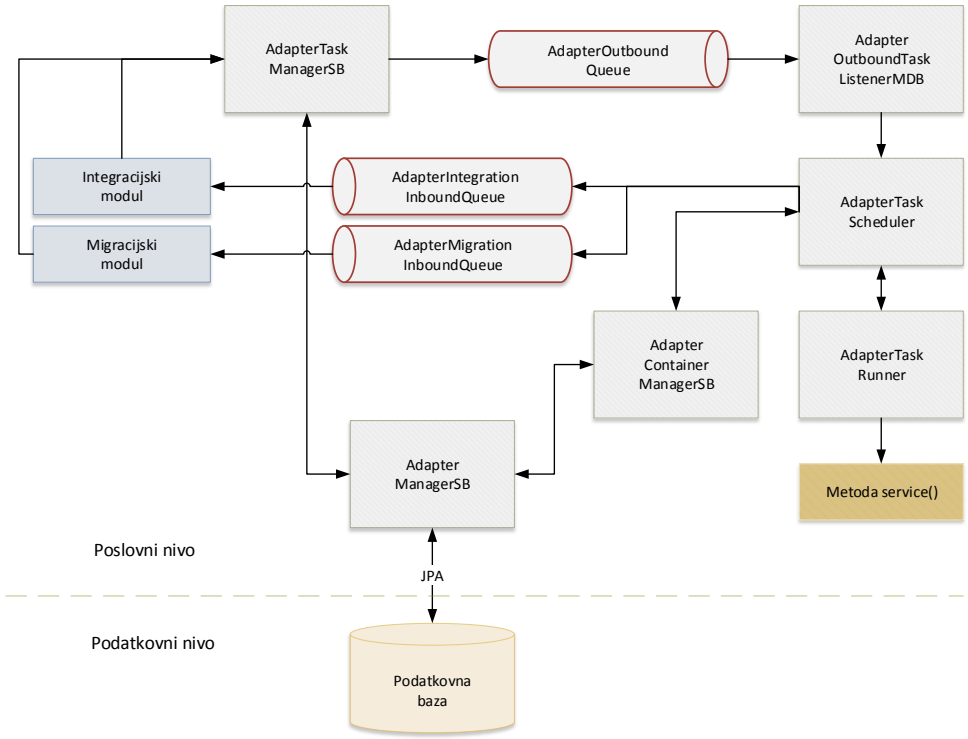
3.2 Ogrodje adapterjev

Izvajanje integracij in migracij omogoča ogrodje adapterjev, ki je sestavljeno iz več javanskih tehnologij. V ogrodju adapterjev so uporabljene naslednje javanske tehnologije:

- javanska strežniška zrna (ang. Enterprise Java Beans – EJB), ki jih lahko razdelimo v dve skupini:
  - sejna zrna (ang. Session Beans – SB),
  - sporočilna zrna (ang. Message-Driven Beans – MDB).
- javanski programski vmesnik za delo s podatki (ang. Java Persistence API – JPA),
- javanski sporočilni sistemi za šibko sklopljenost sistema (ang. Java Message Service – JMS).

Izvajalno okolje adapterjev je prikazano na sliki 3.2. Začetek izvajanja migracije oz. integracije se prične z zahtevo uporabnika (oz. sistema v primeru periodičnega izvajanja), v migracijskem oz. integracijskem modulu. To zahtevo smo poimenovali `Task`, ki si jo lahko predstavljamo kot nalogo, opravilo, ki se mora izvesti. `Task` se iz ustreznega modula pošlje v `AdapterTaskManagerSB`. To sejno zrno lahko razporeja s `Taski`, beleži `Taske`, ki so bili neuspešni, in jih vnaša v bazo preko sejnega zrna `AdapterManagerSB`.

`Task` se nato vstavi v sporočilno vrsto JMS z imenom `AdapterOutboundQueue`, od koder ga prevzame sporočilno zrno `AdapterOutboundTaskListenerMDB`. Naloga tega sporočilnega zrna je le prevzem `Taska` iz sporočilne



Slika 3.2: Izvajalno ogrodje adapterjev.

vrste, ki ga nato dostavi razredu `AdapterTaskScheduler`. Ta razred vrši naslednje naloge.

- Vse `Taske`, uspešno in neuspešno izvedene, pošilja nazaj v ustrezno sporočilno vrsto JMS. Naloge, ki so bile namenjene integraciji, pošlje v `AdapterIntegrationInboundQueue`. Naloge, ki so bile namenjene migraciji, pa pošlje v vrsto JMS z imenom `AdapterMigrationInboundQueue`.
- `Taske`, ki so nared za izvajanje, pošlje razredu `AdapterTaskRunner`.
- Poskrbi, da je ustrezen adapter pripravljen na izvajanje.

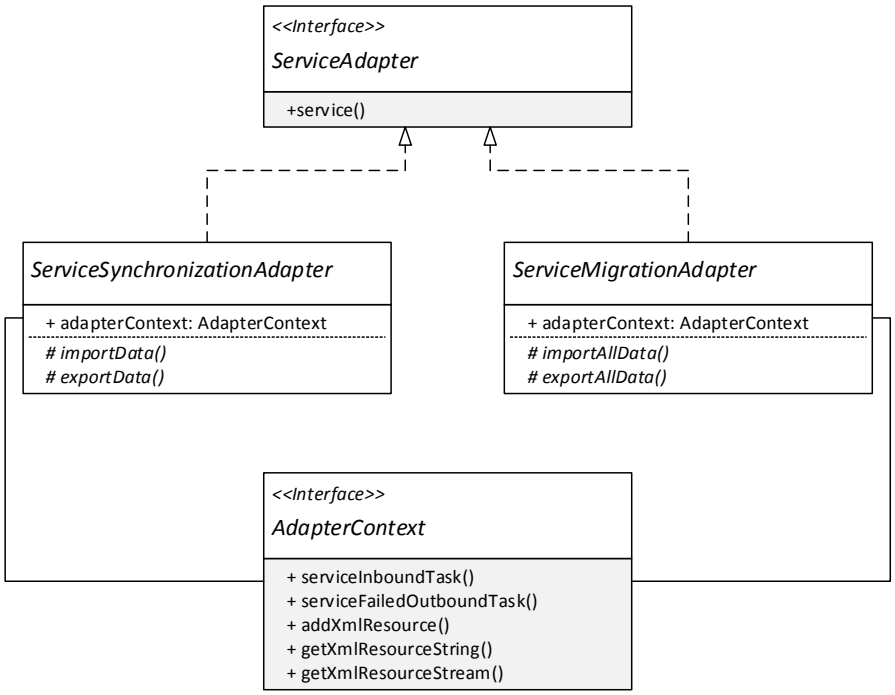
Če ustrezen adapter ni na voljo za izvajanje, ga `AdapterTaskScheduler` zahteva od sejnega zrna `AdapterContainerManagerSB`. Če je instanca ustreznega adapterja že pognana, mu `AdapterContainerManagerSB` vrne to, če pa ni, se adapter naloži iz podatkovne baze in se nato ustvari nova instanca adapterja. Za komunikacijo s podatkovno bazo poskrbi sejno zrno `AdapterManagerSB` preko programskega vmesnika JPA.

Za izvajanje nalog oz. t. i. `Taskov` poskrbi razred `AdapterTaskRunner`. Ta pokliče metodo `service()`, ki jo vsebuje vsak adapter.

3.2.1 Implementacija adapterjev

Vsak adapter mora dedovati ustrezen abstraktni razred. Na sliki 3.3 sta prikazana dva abstraktna razreda, ki implementirata vmesnik `ServiceAdapter`. Abstraktni razred `ServiceMigrationAdapter` dedujejo migracijski adapterji, abstraktni razred `ServiceSynchronizationAdapter` pa integracijski adapterji.

Vmesnik `ServiceAdapter` vsebuje metodo `service()`. To metodo kliče razred `AdapterTaskRunner` (glej sliko 3.2). Ta metoda se povozi (ang. override) v razredih, ki implementirajo vmesnik `ServiceAdapter`. To sta abstraktna razreda `ServiceMigrationAdapter` in `ServiceSynchronizationAdapter`, ki poleg metode `service()` deklarirata tudi metodi za uvoz in



Slika 3.3: Diagram UML razredov in vmesnikov, ki jih uporablja adapter.

izvoz podatkov. Abstraktni razred `ServiceMigrationAdapter` deklarira abstraktni metodi `importAllData()` za uvoz podatkov s storitve in `exportAllData()` za izvoz podatkov na storitev. Abstraktni razred `ServiceSynchronizationAdapter` pa deklarira abstraktni metodi `importData()` za uvoz podatkov s storitve in `exportData()` za izvoz podatkov na storitev.

3.2.1.1 Metodi za uvoz in izvoz podatkov

V primeru migracijskega adapterja je metoda za uvoz podatkov poimenovana `importAllData()`, v primeru integracijskega adapterja pa `importData()`. Metoda za izvoz podatkov je v primeru migracijskega adapterja poimenovana `exportAllData()`, v primeru integracijskega adapterja pa `exportData()`.

Preden začnemo z razvojem adapterja za konkretno aplikacijo tipa SaaS oz. storitev, je potrebno ustvariti nov razred, ki povozi metodi za uvoz in izvoz podatkov. Metoda za uvoz pri migracijskem adapterju kot argument prejme objekt tipa `ImportAllDataOutboundTask`. Ta predstavlja `Task`, ki je prišel v sistem in se mora izvesti. Metoda vrača objekt tipa `ImportedDataInboundTask`. Ta predstavlja opravljen `Task`, ki se vrača v sporočilno vrsto JMS (glej sliko 3.2). Naloga metode za uvoz podatkov je, da pridobi podatke s storitve in jih zabeleži v obliki XML. To v sami metodi storimo tako, da pridobljen XML, zapisan v objektu `String` ali `StreamSource`, dodamo v podatkovno bazo s pomočjo metode `addXmlResource()`, ki jo kličemo na objektu `AdapterContext` (glej sliko 3.3). Ta metoda vrne enolični identifikator (Universally Unique Identifier - UUID) za dodan resurs XML. Objekt `ImportedDataInboundTask`, ki ga vrača metoda za uvoz, dobimo s klicem metode `getImportedDataInboundTask()`. To metodo kličemo nad instanco objekta `ImportAllDataOutboundTask`, ki ga dobimo kot argument v metodo za uvoz. Podamo ji ta enolični identifikator UUID, ki smo ga dobili z dodajanjem resursa XML v podatkovno bazo. Za lažjo predstavo glej izsek kode 3.2.

Izsek kode 3.2: Primer kode za uvoz podatkov.

```
protected ImportedDataInboundTask importAllData(ImportAllDataOutboundTask
    idot) {
  /**
  * Successfully retrieve XML
  */
  UUID uuid = adapterContext.addXmlResource(xml);
  ImportedDataInboundTask idt = idot.getImportedDataInboundTask(uuid);
  return idt;
}
```

Pri metodi za izvoz podatkov je nekoliko drugače. V metodo za izvoz kot argument, prejmemo objekt tipa `ExportAllDataOutboundTask`. Na instanci tega objekta kličemo metodo `getOutboundXmlResourceId()`, ki vrne enolični identifikator resursa XML v podatkovni bazi. Ta identifikator podamo metodi `getXmlResourceString()`, ki jo kličemo nad objektom `AdapterContext`. Ta lahko vrne objekt tipa `String` ali `StreamSource`, v katerem so zapisani podatki za izvoz na aplikacijo tipa SaaS oz. storitev. Na koncu je potrebno še vrniti objekt tipa `ExportCompletedInboundTask`, ki ga dobimo z metodo `getExportCompletedInboundTask()`. Metodo kličemo nad instanco objekta `ExportAllDataOutboundTask`, ki smo prejeli kot argument. Za lažjo predstavo glej izsek kode 3.3.

Izsek kode 3.3: Primer kode za izvoz podatkov.

```
protected ExportCompletedInboundTask exportAllData(ExportAllDataOutboundTask
    edot)
  UUID uuid = edot.getOutboundXmlResourceId();
  String xml = adapterContext.getXmlResourceString(uuid);
  /**
  * Successfully export XML
  */
  ExportCompletedInboundTask ecit = edot.getExportCompletedInboundTask();
  return ecit;
}
```

Prikazani metodi in postopki so opisani za pripravo razreda za migracijski adapter. Pri integracijskem je podobno, le da imamo drugačna imena objektov in metod. V tabeli 3.1 so prikazana imena objektov in metod, ki so različna za migracijo in integracijo, pomenijo pa isto.

	Migracijski adapter	**Integracijski adapter**
1	importAllData()	importData()
2	exportAllData()	exportData()
3	ImportAllDataOutboundTask	ImportDataOutboundTask
4	ExportAllDataOutboundTask	ExportDataOutboundTask

Tabela 3.1: Prikaz različnih imen metod in objektov pri integraciji in migraciji.

Številke v zgornji tabeli 3.1 pomenijo:

1. metodo za uvoz podatkov,
2. metodo za izvoz podatkov,
3. argument, ki ga prejme metoda za uvoz podatkov,
4. argument, ki ga prejme metoda za izvoz podatkov.

Poglavje 4

Razvoj adapterja

V okviru projekta Sintesis smo razvili adapterje za vrsto različnih storitev oz. aplikacij tipa SaaS, kot so: Salesforce, Facebook, Zoho Projects in druge. Adapterji se za vsako posamezno storitev precej razlikujejo.

Prva stvar, ki jo velja omeniti, je vrsta spletne storitve (ang. Web Service), do katere bomo z adapterjem vršili dostop. Ta je lahko Representational State Transfer (REST [10]) spletna storitev ali pa spletna storitev, ki za izmenjavo sporočil uporablja Simple Object Access Protocol (SOAP [12]). Glavna razlika med tema dvema vrstama je, da imamo pri storitvi REST na razpolago spletne naslove (Uniform Resource Locator, krajše URL), ki so enolični za vsak vir posebej. Pri storitvi SOAP pa imamo na razpolago dokument, napisan v jeziku Web Services Description Language (WSDL), s katerim se opiše način klicanja funkcij, ki jih spletna storitev omogoča. Storitve SOAP za izmenjavo sporočil uporabljajo obliko XML, medtem ko storitve REST lahko uporabljajo tudi druge oblike, kot je npr. JSON.

Druga stvar je način preverjanja pristnosti prek programskega vmesnika (ang. Application Programming Interface, krajše API) na spletno storitev. Nekatere aplikacije tipa SaaS oz. storitve uporabljajo standard OAuth, nekatere pa čisto navadne postopke, ki od uporabnika zahtevajo le uporabniško ime in geslo ali ključ API.

Vsi adapterji pa imajo nekaj skupnega in to je, da vsak adapter omogoča

uvoz podatkov iz aplikacije tipa SaaS oz. storitve in pa izvoz podatkov na aplikacijo tipa SaaS oz. storitev.

Za prikaz praktičnega primera razvoja adapterja bomo uporabil migracijski adapter za aplikacijo tipa SaaS oz. storitev Salesforce, ki ponuja programski vmesnik SOAP, in migracijski adapter za aplikacijo tipa SaaS oz. storitev Do, ki ponuja programski vmesnik REST.

4.1 Razvoj adapterja za storitev Salesforce

Salesforce.com je eno izmed vodilnih podjetij s področja računalništva v oblaku [6]. Najbolj znan produkt je Salesforce, ki je produkt za upravljanje odnosov s strankami (ang. Customer Relationship Management, krajše CRM) [11].

Razvili smo adapter, ki zna podatke s storitve Salesforce uvoziti in jih tudi izvoziti nazaj na storitev. Storitev Salesforce ponuja vrsto različnih programskih vmesnikov. Odločili smo se za SOAP API, ker lahko WSDL dokument pretvorimo v Java Archive (JAR) arhive, ki nam služijo kot knjižnice za izvajanje programskih klicev na storitev Salesforce. Na voljo sta nam dve vrsti dokumentov WSDL:

- Partner WSDL in
- Enterprise WSDL.

Enterprise WSDL razvijalcem omogoča razvoj aplikacije le za eno specifično organizacijo na Salesforce storitvi, medtem ko je Partner WSDL namenjen razvoju aplikacij za več vrst organizacij [7]. Ker sta namena naše aplikacije integracija in migracija podatkov med različnimi storitvami oz. aplikacijami tipa SaaS, smo izbrali Partner WSDL.

Partner WSDL definira splošen objekt [1] `sObject`, ki nadomesti vsa specifična v Enterprise WSDL. V slednjem najdemo definicije vseh možnih

[1] V tem razdelku bomo ločevali javanske objekte (pisano navadno - objekt) in objekt, ki predstavlja entiteto na aplikaciji Salesforce (npr. `Account`, `Contact` ipd. Te bomo pisali poševno - *objekt*).

objektov, kot so: `Account`, `Contact` in še veliko drugih[2].

4.1.1 Uvoz podatkov

Preden začnemo z uvozom podatkov s storitve Salesforce, je potrebno preko programskega vmesnika preveriti pristnost uporabnika. V našo aplikacijo bo uporabnik vnesel potrebne podatke, ki se bodo potem posredovali do adapterja. Odločili smo se za navadno avtentikacijo (ang. Basic Authentication), ker je to najpreprosteje in je uporabniku potrebno vnesti samo uporabniško ime in geslo, s katerima je registriran na Salesforce.com, ter varnostni žeton, ki ga uporabnik dobi v nastavitvah svojega uporabniškega računa na storitvi Salesforce. Celoten postopek prijave uporabnika prek programskega vmesnika storimo na začetku izvajanja adapterja. To storimo tako, da uporabimo razred `ConnectorConfig`, definiran v arhivu JAR, ki smo ga dobili s pretvorbo datoteke Partner WSDL. Ustvarimo novo instanco tega razreda in pokličemo metode nad tem objektom: `setUsername`, `setPassword` in `setAuthEndpoint`. Prvi metodi kot argument podamo uporabniško ime uporabnika, drugi metodi podamo geslo, tretji pa končno točko (ang. Endpoint), na katero se bomo povezali. Na voljo imamo dve končni točki - ena se uporablja ob uporabi dokumenta Partner WSDL, druga pa ob uporabi Enterprise WSDL. Mi smo uporabili Partner WSDL, zato moramo za končno točko nastaviti temu primeren spletni naslov (URL). Metodi setAuthEndpoint() torej podamo argument tipa `String` - tj. URL `https://login.salesforce.com/services/Soap/u/24.0/`. Instanco objekta `ConnectorConfig` z nastavljenimi parametri nato podamo kot argument objektu `PartnerConnection`, ki ustvari povezavo do storitve Salesforce. Če preverjanje pristnosti ni uspešno, povezava sproži izjemo `ConnectionException`.

Sedaj, ko smo se uspešno povezali na Salesforce, lahko pričnemo z uvozom podatkov s storitve. Storitev Salesforce ponuja poizvedovalni jezik (Sales-

[2]Opise vseh objektov in njihovih atributov najdemo na straneh uradne dokumentacije programskega vmesnika SOAP storitve Salesforce [8].

force Object Query Language - SOQL), s pomočjo katerega lahko pridobimo podatke s storitve. SOQL je poizvedovalni jezik in ima podobno sintakso kot jezik Structured Query Language (SQL) za poizvedovanje podatkov iz podatkovnih baz.

Za izvrševanje poizvedbe potrebujemo stavek SOQL. Ker želimo uvoziti vse podatke, ki so na voljo o organizaciji na storitvi Salesforce, je potrebno najprej ugotoviti, katere *objekte* in njihove atribute lahko pridobimo. To storimo tako, da pokličemo metodo `describeGlobal()` nad povezavo `PartnerConnection`, ki smo jo vzpostavili na začetku. Ta metoda vrne rezultat tipa `DescribeGlobalResult`[8]. V tem objektu je shranjeno polje vseh *objektov*, ki jih lahko pridobimo za posamezno organizacijo, oz. uporabniški račun, registriran na Salesforce. Polje vseh *objektov* pridobimo s klicem metode `getSobjects()` nad objektom `DescribeGlobalResult`. Vsak objekt v tem polju ima veliko metod, ki ga opisujejo. Na primer metoda `getName()` vrne ime *objekta*. Uporabna je tudi metoda `isQueryable()`, ki vrne vrednost `true`, če lahko poizvedujemo nad tem objektom, ali `false`, če to ni dovoljeno. To metodo uporabljamo, preden se generira stavek SOQL, ker je nesmiselno pošiljati poizvedbe nad *objektom*, za katerega to ni mogoče. To je prikazano v izseku kode 4.1.

Na podlagi imena *objekta*, ki ga dobimo z metodo `getName()`, pridobimo objekt tipa `DescribeSObjectResult`. Nad tem objektom kličemo metodo `getFields()`, ki nam vrne polje objektov tipa `Fields`, ki predstavljajo polja oz. atribute posameznega *objekta* na storitvi Salesforce. Sedaj, ko imamo vse razpoložljive *objekte* in njihove atribute, lahko sestavimo stavek SOQL, s katerim bomo pridobili podatke s storitve Salesforce. S preprostim algoritmom z dvema `for` zankama se sprehodimo čez polje *objektov* in polje atributov ter vse skupaj zapisujemo v en niz.

Prvi del kode je v samem programu, kjer se po generiranem stavku ta tudi uporabi za pridobitev podatkov s klicem metode `query()`.

Izsek kode 4.1: Iteracija čez objekte.

```
for (int i = 0; i < dgsr.length; i++) {
  if (dgsr[i].isQueryable()) {
    String sql = SalesforceUtil.generateSOQL(dgsr[i].getName(), fields);
    xml += SalesforceUtil.query(sql, fields, dgsr[i].getName());
  }
}
```

Izsek kode 4.2 prikazuje metodo, ki zgradi SOQL stavek iz imena objekta in polja atributov.

Izsek kode 4.2: Iteracija čez atribute posameznega objekta in sestava stavka SOQL.

```
public static String generateSOQL(String objectName, Field[] fields) {
  String sql = "SELECT ";
  for (int i = 0; i < fields.length; i++) {
    if (i != fields.length - 1) {
      sql += fields[i].getName() + ", ";
    } else {
      sql += fields[i].getName();
    }
  }
  sql += " FROM " + objectName;
  return sql;
}
```

Metoda `query()`, ki se kliče v izseku kode 4.1, potem uporabi sestavljen stavek SOQL za poizvedbo. Poizvedbo vršimo tako, da kličemo metodo `query()` nad povezavo `PartnerConnection`, ki smo jo vzpostavili na začetku. Tej metodi kot argument, podamo stavek SOQL. Rezultat poizvedbe se shrani v objekt `QueryResult`. Z metodo `getRecords()` nad tem objektom pridobimo zapise, ki jih je vrnila poizvedba. Metoda vrača polje zapisov oz. polje objektov `SObject`. Za vsak objekt `SObject` lahko dobimo imena atributov in njihove vrednosti.

4.1.2 Izvoz podatkov

Izvoz podatkov poteka tako, da adapter najprej prebere vhodni niz v obliki XML. Ta predstavlja podatke, ki so namenjeni izvozu na storitev Salesforce.

Podatki so lahko shranjeni v objektu tipa `String` ali v objektu tipa `StreamSource`. Najprej se je potrebno povezati na aplikacijo prek programskega vmesnika, kot je to storjeno v metodi za uvoz podatkov. Nato lahko pričnemo z razčlenjevanjem niza XML. Iz niza XML lahko ustvarimo nov objekt `Document`, ki predstavlja drevesno strukturo DOM (Document Object Model) niza XML. Po drevesu se sprehajamo tako, da ustvarjamo sezname otrok, ki jih ima določen element, začenši s korenskim elementom. Ko pridemo do elementov, ki so listi (nimajo več otrok), ustvarimo seznam, v katerem so pari ime atributa, vrednost. To storimo za vsak *objekt* posebej.

Preden naložimo določen zapis oz. *objekt* na storitev, moramo preveriti, če zapis že obstaja. S tem se izognemo podvojenim zapisom na storitvi v primeru migracije in integracije. Ker nam programski vmesnik SOAP omogoča izvajanje poizvedb SOQL, lahko to izkoristimo za ugotavljanje, če zapis že obstaja. Preden zapis naložimo, sestavimo stavek SOQL, ki vsebuje vrednosti trenutnega zapisa. Še prej pa smo definirali atribute, po katerih se preverja enakost zapisa. Nesmiselno je preverjati enakost zapisa (npr. `Contact`) po npr. telefonski številki, če lahko preverimo po elektronski pošti, ki je unikaten za vsakega uporabnika oz. organizacijo. Za vsak primer dodamo še ime in priimek. Če poizvedba s stavkom SOQL vrne enega ali več rezultatov, pomeni, da zapis obstaja in ga ne smemo ponovno naložiti. V nasprotnem primeru ga naložimo.

Nov zapis na storitvi Salesforce ustvarimo tako, da ustvarimo nov objekt `SObject`. Najprej mu nastavimo tip z metodo `setType()`, ki jo kličemo nad instanco objekta `SObject`. S to metodo nastavimo tip oz. nastavimo *objekt*, ki pove, za katero entiteto - na storitvi Salesforce - gre (npr. `Account`, `Contact` ali `Lead`). Vsak atribut in njegovo vrednost nastavimo z metodo `setField()`, ki prejme dva argumenta, ime atributa in vrednost atributa. Ime atributa vedno nastavimo kot objekt tipa `String`. Pri nastavljanju vrednosti atributa pa moramo paziti na drugačne tipe, kot so celoštevilski tip (`int`) in datum (`Date`). Metoda `setField()` kot drugi argument prejme objekt tipa `Object`, tako da lahko na to mesto vstavimo poljuben javanski objekt. Ko na-

stavimo vsa imena in vrednosti atributov, ustvarimo objekt. To storimo tako, da kličemo metodo `create()` nad instanco objekta `PartnerConnection`, ki predstavlja vzpostavljeno povezavo na storitev. Metoda za argument prejme polje (ang. array) objektov `SObject`, tako da ustvarjeni SObject z nastavljenimi parametri damo v polje in to polje nato podamo metodi `create()`. Metoda vrne polje objektov `SaveResult`. Vsak objekt `SaveResult` ima metodo `isSuccess()`. V primeru, da vrne `true`, je bil *objekt* uspešno naložen na Salesforce. V nasprotnem primeru pokličemo metodo `getErrors()`, ki vrne polje objektov `Error`, na podlagi katerega lahko z metodama `getStatusCode()` in `getMessage()` ugotovimo, kaj je bilo narobe pri nalaganju *objekta*.

4.2 Razvoj adapterja za storitev Do

Adapter za storitev *Do* (https://www.do.com/) se precej razlikuje od adapterja za Salesforce. Za razvoj adapterja smo uporabili programski vmesnik *Do* (*Do* API [3]). *Do* ponuja storitev REST, dostopno prek protokola HTTPS (HyperText Transfer Protocol Secure). Za uporabo programskega vmesnika *Do* moramo za preverjanje pristnosti uporabiti varnostni standard OAuth verzije 2.

4.2.1 Preverjanje pristnosti z OAuth2

Preverjanje pristnosti z OAuth verzije 2 [2] od uporabnika zahteva vnos več varnostnih ključev oz. žetonov. Prvi korak je pridobivanje žetona za dostop (ang. `access token`). Obstajata dva načina pridobivanja tega žetona [4].

- `Web Flow`: ta način zahteva dva koraka pri preverjanju pristnosti. Prvi korak je pridobitev avtorizacije. Pošlje se zahtevek GET na spletni naslov `https://www.do.com/oauth2/authorize`. Temu zahtevku nastavimo parametre:
  - `response_type=code` – obvezni parameter po standardu OAuth2,

  - `client_id` – obvezni parameter, enolični identifikator, ki nam ga določi storitev *Do*,
  - `redirect_uri` – opcijski parameter, ki pove, kam naj se stran preusmeri po uporabnikovi avtorizaciji.

  Po tem zahtevku mora uporabnik vnesti še uporabniško ime in geslo in s tem aplikaciji odobriti dostop do svojega uporabniškega računa. Po uspešni avtorizaciji pridobimo kodo, ki jo uporabimo v naslednjem koraku. Naslednji in zadnji korak je pridobivanje žetona za dostop. Za pridobitev žetona za dostop, pošljemo zahtevek POST na spletni naslov `https://www.do.com/oauth2/token`. Potrebno je dodati naslednje parametre:

  - `grant_type=authorization_code` – obvezen parameter, ki mora biti vključen v zahtevek po standardu OAuth2,
  - `client_id` – isto kot pri prejšnjem koraku,
  - `client_secret` – obvezen parameter, ki ga prejmemo od storitve Do,
  - `code` – koda, ki smo jo prejeli v prejšnjem koraku.

  V odgovoru na zahtevek dobimo žeton za dostop.

- `Password-Grant Flow`: ta način, za razliko od prejšnjega, potrebuje le en korak. Ta je podoben drugemu koraku pri načinu `Web Flow`. Namesto parametra `code`, ki ga v tem primeru nimamo, dodamo parametra `username` in `password`, ki sta obvezna in predstavljata uporabniško ime in geslo. Spremeniti moramo tudi vrednost parametra `grant_type` in tako `authorization_code` spremeniti v `password`.

Odgovor na zahtevek je v obliki JSON. Vsebuje štiri parametre, in sicer:

- `token_type` – vrednost tega parametra se uporabi v avtorizacijski glavi zahtevka GET oz. POST,

- `access_token` – vrednost tega parametra je žeton za dostop, ki omogoča spreminjanje uporabniškega računa. Tudi ta se uporabi v glavi zahteve HTTP za preverjanje pristnosti,
- `refresh_token` – vrednost tega parametra je osvežilni žeton, ki se uporablja za pridobitev novega žetona za dostop,
- `expires_in` – vrednost tega parametra je veljavnost žetona za dostop in osveževalnega žetona.

Uporabili bomo način `Password-Grant Flow`, ker nam omogoča, da izvedemo preverjanje pristnosti v enem koraku in ne v dveh.

4.2.2 Uvoz podatkov

S storitve *Do* lahko uvozimo več entitet oz. objektov. *Do* ponuja storitev REST, zato je vsak objekt prestavljen z virom, preko katerega lahko dostopamo do objekta.

- `Workspace` – https://www.do.com/workspaces,
- `Project` – https://www.do.com/workspaces/`idW`/projects,
- `User` – https://www.do.com/workspaces/`idW`/users,
- `Task` – https://www.do.com/workspaces/`idW`/tasks in https://www.do.com/workspaces/idW/projects/`idP`/tasks,
- `Note` – https://www.do.com/workspaces/`idW`/notes.

Beseda `idW` nadomešča enolični identifikator, ki ga ima posamezen `Workspace`, beseda `idP` pa nadomešča enolični identifikator, ki ga ima posamezen `Project`.

Pri oblikovanju niza, v katerem je zapisan XML uvoženih podatkov, je potrebno paziti na hierarhijo zgornjih objektov. Kot lahko vidimo zgoraj, so vsi objekti vezani na `Workspace`. To pomeni, da bo `Workspace` najvišje

ležeči, korenski element v strukturi XML. `Task`, ki je poleg na `Workspace` vezan še na `Project`, bo v strukturi XML gnezden pod `Project`.

Prvi korak je, da dobimo vse *objekte* tipa `Workspace`, ker potrebujemo identifikatorje za pridobitev ostalih podatkov. To storimo tako, da pošljemo zahtevek GET na spletni naslov prikazan v zgornji alineji. V odgovoru dobimo vse *objekte* tipa `Workspace`, ki pripadajo avtenticiranemu uporabniku. Podatki so zapisani v obliki JSON. Pretvorimo jih v obliko XML in iz njih preberemo identifikatorje vseh *objektov* tipa `Workspace`.

Sedaj, ko imamo seznam identifikatorjev vseh *objektov* tipa `Workspace`, vstavimo vsakega posebej v spletne naslove drugih *objektov* in tako pridobimo še ostale podatke. Paziti je potrebno pri *objektu* tipa `Task`, gnezden je lahko tudi pod `Project` (poleg tega da je pod `Workspace`). Če je temu tako, moramo paziti da ne pride do podvajanja podatkov, v tem primeru podvajanja *objektov* tipa `Task`. `Project` je že sam gnezden v `Workspace`, zato je `Task`, ki je del *objekta* `Project`, avtomatsko že vezan tudi na `Workspace`. Tisti *objekti* tipa `Task`, ki so gnezdeni pod `Project`, torej ne smejo biti gnezdeni direktno pod `Workspace`. Vsi pridobljeni podatki so v obliki JSON, tako da je potrebna pretvorba v obliko XML.

4.2.3 Izvoz podatkov

Prvi korak pri izvozu podatkov je preverjanje pristnosti. Če je uporabnik vnesel pravilne podatke, lahko pridobimo žeton za dostop in naložimo podatke.

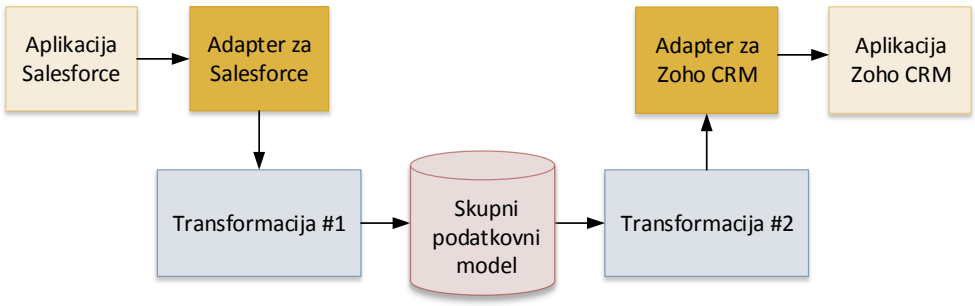
Podatke dobimo iz skupnega podatkovnega modela v obliki XML. Ker so vsi *objekti* vezani na `Workspace`, je iz niza XML najprej potrebno pridobiti vse objekte tipa `Workspace`. V okviru integracije in migracije podatkov je pred nalaganjem potrebno preveriti, če zapis že obstaja. Preverimo, če objekt tipa `Workspace` z nekim imenom na storitvi že obstaja. V primeru, da še ne obstaja, ustvarimo novega s tistim imenom, ki smo ga prebrali iz vhodnih podatkov. V odgovoru nato dobimo podatke o ustvarjenem *objektu*, med katerimi je tudi identifikator. V primeru, da obstaja, je potrebno preko

imena s storitve pridobiti identifikator *objekta* `Workspace`. Edini način, da dobimo identifikator, je ta, da uvozimo vse objekte tipa `Workspace` in nato primerjamo imena s tistim, ki ga želimo naložiti. Potem ko ugotovimo, če `Workspace` obstaja, v vsakem primeru pridobimo identifikator *objekta* `Workspace`. Tega potrebujemo za nalaganje ostalih *objektov*. Podobno je pri *objektih* `Task`, ki so gnezdeni pod `Project`. Tam moramo ravno tako pridobiti identifikator *objekta* `Project`, da lahko naložimo `Task`.

Nalaganje posameznih zapisov poteka preko zahtevkov POST, protokola HTTPS. V glavi zahtevka je potrebno ustrezno nastaviti parameter `Authorization`. V telesu zahtevka pa podamo parametre nekega objekta (npr. `Workspace`, `Project` ...), zapisane v obliki JSON. Med temi parametri morajo biti nujno prisotni obvezni parametri, ker se v primeru, da jih ni, zapis ne naloži. Ker imamo na vhodu v adapter podatke v obliki XML, je potrebno pred nalaganjem podatke pretvoriti v obliko JSON. Pred vsakim nalaganjem določenega *objekta* je potrebno preverjanje že obstoječih zapisov. Ko se zapis uspešno naloži, dobimo v odgovoru celotne podatke o *objektu* v obliki JSON.

4.3 Primer migracije podatkov med dvema aplikacijama

Prikazali bomo konkreten primer migracije podatkov med dvema aplikacijama tipa SaaS oz. storitvama Salesforce in Zoho CRM. Ti dve aplikaciji spadata v skupino aplikacij CRM. Migrirali bomo objekt `Account`, ki bo v tem primeru predstavljal shranjeno organizacijo na uporabniškem računu. Vsi prikazani podatki so zapisani v obliki XML. Slika 4.1 prikazuje tok podatkov, pridobljenih na aplikaciji Salesforce, skozi komponente celotnega sistema.



Slika 4.1: Migracija podatkov iz aplikacije Salesforce na aplikacijo Zoho CRM.

1. S pomočjo adapterja za Salesforce uvozimo podatke iz aplikacije tipa SaaS oz. storitve. Primer pridobljenih podatkov s storitve je prikazan v izseku kode 4.3.

   Izsek kode 4.3: Primer (enega dela) uvoženih podatkov s storitve Salesforce v obliki XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<salesforce>
  <Account>
    <Id>001b000000AQ1I5AAL</Id>
    <Name>Raiden</Name>
    <Type>Customer</Type>
    <BillingStreet>Punjab 14</BillingStreet>
    <BillingCity>Random City</BillingCity>
    <BillingState>Random State</BillingState>
    <BillingPostalCode>90210</BillingPostalCode>
    <BillingCountry>Barbados</BillingCountry>
    <ShippingStreet>Still Street 22</ShippingStreet>
    <ShippingCity>Still City</ShippingCity>
    <ShippingState>Still State</ShippingState>
    <ShippingPostalCode>80221</ShippingPostalCode>
    <ShippingCountry>Belgium</ShippingCountry>
    <Phone>0304321</Phone>
    <Fax>0304123</Fax>
    <Website>www.example.com</Website>
    <NumberOfEmployees>91</NumberOfEmployees>
    <Description>An account description.</Description>
  </Account>
```

```
</salesforce>
```

2. Izvede se transformacija (na sliki 4.1, Transformacija #1) nad uvoženimi podatki. Oblika podatkov iz adapterja za Salesforce se pretvori v obliko podatkov, ki ustrezajo shemi skupnega podatkovnega modela. Primer pretvorjenih podatkov, s pomočjo transformacij XSL, je prikazan v izseku kode 4.4.

3. Podatki se shranijo v skupni podatkovni model.

   Izsek kode 4.4: En del pretvorjenih podatkov, ki se shranijo v podatkovno bazo.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  ...
  <accounts:address>
    <accounts:streetName>Still Street 22</accounts:streetName>
    <accounts:city>Still City</accounts:city>
    <accounts:stateOrRegion>Still State</accounts:stateOrRegion>
    <accounts:postCode>80221</accounts:postCode>
    <accounts:country>
      <accounts:countryName>Belgium</accounts:countryName>
    </accounts:country>
    <accounts:addressLabel>Shipping</accounts:addressLabel>
  </accounts:address>
  <accounts:type>Customer</accounts:type>
  <accounts:description>An account description.</accounts:description>
  <accounts:numberOfEmployees>91</accounts:numberOfEmployees>
  <accounts:website>www.example.com</accounts:website>
  ...
<root>
```

4. Izvede se transformacija (na sliki 4.1, Transformacija #2) nad podatki, ki so v shemi skupnega podatkovnega modela. Pretvorijo se v obliko podatkov, ki ustreza adapterju za storitev Zoho CRM. Primer podatkov, ki so namenjeni na storitev Zoho CRM, je v izseku kode 4.5.

Izsek kode 4.5: Primer podatkov, ki ustrezajo adapterju za Zoho CRM.

```
<?xml version="1.0" encoding="UTF-8"?>
<response uri="/crm/private/xml/Accounts/getRecords">
  <result>
    <Accounts>
      <row no="1">
        <FL val="Billing Street">Punjab 14</FL>
        <FL val="Billing Code">90210</FL>
        <FL val="Billing City">Random City</FL>
        <FL val="Billing State">Random State</FL>
        <FL val="Billing Country">Barbados</FL>
        <FL val="Shipping Street">Still Street 22</FL>
        <FL val="Shipping Code">80221</FL>
        <FL val="Shipping City">Still City</FL>
        <FL val="Shipping State">Still State</FL>
        <FL val="Shipping Country">Belgium</FL>
        <FL val="Account Name">Raiden</FL>
        <FL val="Fax">0304123</FL>
        <FL val="Phone">0304321</FL>
        <FL val="Account Type">Customer</FL>
        <FL val="Description">An account description.</FL>
        <FL val="Website">www.example.com</FL>
        <FL val="Employees">91</FL>
        <FL val="Annual Revenue"></FL>
      </row>
    </Accounts>
  </result>
</response>
```

5. Adapter za storitev Zoho CRM te podatke prebere iz podatkovne baze in jih naloži na storitev Zoho CRM.

Poglavje 5

Sklepne ugotovitve

V diplomskem delu smo predstavili idejo, kako integrirati aplikacije tipa SaaS in migrirati podatke med njimi s pomočjo migracijskih adapterjev. Osredotočili smo se na razvoj migracijskih adapterjev za posamezne aplikacije tipa SaaS. Prikazali smo idejo skupnega podatkovnega modela, ki predstavlja središče vseh zbranih podatkov iz poljubnih aplikacij tipa SaaS oz. storitev. Opisali smo tudi pomembnost transformacij podatkov in jezika XML za predstavitev podatkov. Prikazano je tudi izvajalno okolje adapterjev, ki omogoča, da se implementacija adapterja naloži iz podatkovne baze. To omogoča nalaganje in izvajanje adapterjev, ki jih lahko naložijo uporabniki aplikacije.

Za izvedbo integracije dveh aplikacij tipa SaaS je nujno potrebno, da migracijski adapter pred nalaganjem preveri, če zapisi na aplikaciji oz. storitvi že obstajajo. Možna izboljšava na tem nivoju integracije aplikacij tipa SaaS je dogodkovno vodena integracija (ang. Event-Driven Integration), pri kateri aplikacije tipa SaaS oz. storitve pošiljajo dogodke ob spremembah podatkov na storitvi. Vendar je to odvisno predvsem od aplikacij tipa SaaS in njihovih programskih vmesnikov, ki po večini ne podpirajo pošiljanja obvestil oz. dogodkov ob spremembah podatkov. To bi predstavljajo popolnejšo integracijo aplikacij, saj bi bili podatki različnih aplikacij povsem sinhronizirani in bi se migracija podatkov lahko izvedla ob vsaki spremembi na storitvi.

Računalništvo v oblaku postaja vse bolj popularno, zato je integracija aplikacij tipa SaaS oz. storitev že sedaj nujna. Vzemimo za primer dva različna ponudnika elektronske pošte. Na uporabniškem računu prvega odjemalca imamo več kontaktnih naslovov, katere bi želeli imeti tudi na drugem. Do sedaj je bilo potrebno ročno izvažati in uvažati podatke. To je sedaj možno s pritiskom na gumb in imamo povsem sinhronizirane kontakte pri obeh odjemalcih.

Literatura

[1] Richard Car. Adapter design pattern. http://www.blackwasp.co.uk/Adapter.aspx. (15. 9. 2013).

[2] Microsoft D. Hardt, Ed. The oauth 2.0 authorization framework. http://tools.ietf.org/html/rfc6749. (2. 9. 2013).

[3] Do.com. Do api. https://developers.do.com/. (2. 9. 2013).

[4] Do.com. Do authentication. https://developers.do.com/authentication. (2. 9. 2013).

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[6] Salesforce. About salesforce.com. http://www.salesforce.com/company/. (2. 9. 2013).

[7] Salesforce.com. Salesforce - using the partner wsdl. http://www.salesforce.com/us/developer/docs/api/Content/sforce_-api_partner.htm. (2. 9. 2013).

[8] Salesforce.com. Salesforce soap api developer's guide. http://www.salesforce.com/us/developer/docs/api/. (2. 9. 2013).

[9] Wikipedia. Enterprise application integration. http://en.wikipedia.org/wiki/Enterprise_application_integration. (2. 9. 2013).

[10] Wikipedia. Representational state transfer. http://en.wikipedia.org/wiki/REST. (2. 9. 2013).

[11] Wikipedia. Salesforce. http://en.wikipedia.org/wiki/Salesforce. (2. 9. 2013).

[12] Wikipedia. Simple object access protocol. http://en.wikipedia.org/wiki/SOAP. (2. 9. 2013).

[13] Wikipedia. Software as a service. http://en.wikipedia.org/wiki/Software_as_a_service. (2. 9. 2013).