CUPRINS

ABSTRACT

În ultima perioadă asistăm la publicarea unui număr din ce în ce mai mare de date cu ajutorul Internetului. Aceste date sunt disponibile pe diferite servere, la diferite locaţii şi folosesc metode diferite de acces. Dintre metodele diferite de acces, enumerăm următoarele: căutarea pe Internet (web search), servicii web sau baze de date. Informaţiile găsite pe Internet devin încetul cu încetul indispensabile activităţii noastre zilnice. Deciziile noastre sunt luate bazate pe datele găsite online. Fie că dorim să cumpărăm un nou produs, fie că dorim să scriem un articol ştiinţific, web-ul a devenit o sursă de informaţii viabilă şi folositoare.

Totuşi, multitudinea informaţiilor disponibile este copleşitoare şi procesul de căutare al ei consumă mult timp. Din cauza multitudinilor de date disponibile şi a numărului crescând de surse procesul căutării şi folosirii lor se complică. De exemplu, dacă un utilizator doreşte să caute o informaţie pe Internet în mai multe locaţii, trebuie să deschidă mai multe instanţe ale browser-ului şi să introducă în fiecare adresa la care doreşte să execute căutarea împreună cu şirul de caractere dorit. În mod asemănător, dacă dorim să dezvoltăm o aplicaţie care execută o căutare în mai multe locaţii pe web, trebuie să ţinem cont de diferitele politici de acces oferite de fiecare sursă în parte, modul diferit de acces al resurselor, etc. Putem observa deci cu uşurinţă că această sarcină nu este deloc uşoară şi de multe ori, trebuie depus un efort suplimentar pentru a ne putea folosi de multitudinea de informaţii publicate pe web.

Lucrarea de faţă îşi doreşte să propună un framework pentru accesul diferitelor surse de date publicate pe Internet sub forma serviciilor web de tip REST. Vom oferi o modalitate generală de acces al lor. Cu ajutorul acesui framework vom putea dezvolta aplicaţii care să poată procesa cantitatea vastă de informaţii disponibilă într-un mod uniform. Astfel, modul diferit de acces al resurselor nu va mai fi un impediment pentru dezvoltarea unor astfel de aplicaţii. Framework-ul va oferi o flexibilitate crescută în adăugarea de noi locaţii precum şi accesul concurent a mai multor utilizatori. De asemenea, aplicaţia poate fi distribuită pe mai multe noduri, dacă este cazul. În vederea implementării ei ne-am folosit de principiile SOA – Service Oriented Arhictecture. În final, vom demonstra folosirea aplicaţiei prin intermediul unor aplicaţii web.

MULŢUMIRI

Mamei mele – îţi mulţumesc pentru sprijinul şi dragostea ta necondiţionată. Regretul că nu poţi fi prezentă în aceste momente frumoase alături de mine, nu poate fi exprimat în cuvinte.

Tatălui meu, pentru că a insistat ca eu să învăţ matematică şi pentru că mi-a adus primul calculator pe când aveam încă o vârstă fragedă. Acest gest mi-a schimbat traiectoria vieţii.

Cristinei, sora mea geamănă, care m-a motivat prin capacitatea ei de muncă şi dedicarea faţă de reponsabilităţile pe care le are.

Bunicului meu, care a fost un model de perseverenţă şi determinare de-a lungul vieţii, inspirându-mă în mod constant.

Îndrumătorului meu, conf. univ. ing. Cosmina Ivan, care a fost întodeauna deschisă la propunerile mele, ajutându-mă să le analizez şi evaluez, oferindu-mi o direcţie clară în elaborarea lucrării. Fără acest ajutor, demersul meu ar fi fost mult mai dificil.

Colegilor mei, în care am putut să observ mult talent şi dedicare, demonstrându-mi prietenia lor.

Cluj-Napoca,
15 Iunie 2009

1. INTRODUCERE

1.1. Motivare

Accesul la informaţiile disponibile pe web este una dintre activităţile pe care le intreprindem zilnic. Indiferent de natura profesiei noastre sau de interesul pe care îl manifestăm, ele au devenit o necesitate. Informaţiile pe care le accesăm determină deciziile pe care le luăm. Din acest motiv ne interesează agregarea datelor din cât mai multe surse pentru a le putea prelucra cu uşurinţă, după bunul plac. Agregarea mai multor surse de informaţii este un proces anevoios, pe care dorim să îl uşurăm.

1.2. Descrierea problemei

În prezent există metode diverse de accesul la datele publicate pe Internet. În primul rând, trebuie să amintim metoda folosită cel frecvent, metodă care este de altfel accesibilă oricărui utilizator – nu doar specialiştilor. Aceasta este căutarea informaţiilor pe web folosindu-ne de motoare de cătuare (eng. „web search"). În ultima perioadă asistăm la apariţia unui număr din ce în ce mai mare de motoare de căutare. Aceste motoare de căutare se pot accesa la diferite locaţii, fiecare folosind parametri diferiţi pentru a efectua o căutare.

Metoda descrisă mai sus se bazează pe o interfaţă a unui browser web. Recent însă, marile companii care oferă astfel de servicii (Microsoft, Yahoo, Google) permit accesul la bazele lor de date cu ajutorul unor API-uri bazate pe servicii web. Astfel, resursele respective pot fi accesate din interiorul unui program. API-urile au specificaţii exacte şi pot fi consultate de către dezvoltator. Fiecare ofertant de astfel de servicii are o politică proprie de acces, de formare a cererii şi de returnare a rezultatelor.

Aceste API-uri folosesc servicii web REST sau SOAP. În cuprinsul lucrării vom detalia diferenţele şi asemănările dintre aceste două tipuri de servicii. Deocamdată ne mulţumim cu menţionarea faptului că serviciile REST sunt mai uşor de folosit, consumă mai puţină lăţime de bandă, folosesc HTTP şi sunt ideale pentru accesul resurselor de pe Internet.

Mai mult decât atât, serviciile REST permit publicarea conţinutului bazelor de date via HTTP. Pentru a interoga o bază de date, nu mai este necesar decât să accesăm URL-ul bazei de date şi să transmitem nişte parametri prin metode specifice acestui protocol. Observăm deci, că există o multitudine de moduri prin care se pot accesa resursele pe Internet. Pe noi ne va interesa accesul resurselor REST într-un mod uniform.

1.3. Obiectivele lucrării

- **Acces unificat la resursele REST aflate pe Internet**.

  În primul rând, interesul nostru este acela de a oferi o modalitate unificată de acces al serviciilor REST expuse pe Internet. Dorim să oferim un registru de servicii web care să poată fi interogat de către utilizator pentru a obţine informaţii despre sursele de date disponibile, parametri pe care îi acceptă şi tipul de rezultat returnat. Vrem să oferim o modalitate cât mai simplă de a forma cereri şi de a le transmite surselor de date aferente.

- **Acces cât mai simplu la sursele de date**

  În al doilea rând, dorim să oferim un acces cât mai simplu la sursele de date, eliminând necesitatea de cunoaşte constrângerile specifice fiecăreia. De exemplu, două surse de date pot avea politici de securitate diferite. Nu dorim ca utilizatorul să fie nevoit să le cunoască. De asemenea, nu dorim ca utilizatorul să fie conştient de modul în care se formează cererile specifice, locaţia serviciilor pe web, etc.

- **Soluţia simplu de întreţinut şi dezvoltat**

  Dorim ca această soluţie să fie cât mai simplu de întreţinut şi dezvoltat. Dorim să o integrăm pe viitor cu alte aplicaţii. De pildă, putem să dezvoltăm o aplicaţie care efectuează căutări în mai multe surse de date simultan. Acest framework va furniza logica necesară pentru adăugarea surselor de date, efectuarea interogărilor şi primirea rezultatelor.

- **Distribuirea aplicaţiei**.

  Din cauza naturii framework-ului şi a aplicaţiilor care pot fi dezvoltate folosindu-l dorim ca el să poată fi cu uşurinţă distribuit pe mai multe noduri, dacă e cazul. Din acest motiv aplicaţia e concepută pe module care pot interaţiona de pe diferite noduri dacă e cazul.

- **Manipularea cu uşurinţă a surselor de date**

  În egală măsură dorim ca adăugarea sau ştergerea surselor de date să se poată efectua într-un mod uşor. Dorim ca această adăugare să poată fi realizată într-un mod dinamic.

1.4. Rezumatul lucrării

În următorul capitol vom enunţa principiile care stau la baza proiectării aplicaţiei descrise mai sus. De asemenea, vom defini noţiunea de serviciu web şi vom da exemple de servicii web existente care se folosesc pentru a construi aplicaţia. În capitolul 3 vom descrie tehnologiile folosite, urmând ca în următoarele două capitole să prezentăm proiectarea aplicaţiei respectiv implementarea ei. Capitolul 6 va pune la dispoziţie informaţii referitoare la configurarea aplicaţiei şi rezultate ale rulării ei, iar ultimul capitol va sublinia câteva concluzii precum şi câteva direcţii de dezvoltări ulterioare.

2. FUNDAMENTARE TEORETICĂ

În acest capitol vom defini serviciile web şi vom exemplifica modul în care acestea uşurează accesul şi comunicarea datelor pe web. După ce am exemplificat conceptul de „serviciu web”, vom ilustra modul în care acestea sunt folosite în industrie de către companii mari. Odată ce ilustrăm aceste noţiuni vom analiza tipul arhitectural SOA (Service Oriented Architecture) şi îl vom compara cu cel EAI (Enterprise Application Integration), menţionând motivele pentru care l-am ales în implementarea acestui framework.

2.1. Servicii web

Conform [5], termenul de „serviciu web” are un înţeles imprecis supus mereu schimbărilor şi redefinirilor. Cu toate acestea, există câteva trăsături comune serviciilor web. După cum sugerează numele, un „serviciu web” este o aplicaţie disponibilă prin web, adică o aplicaţie care rulează în mod tipic prin HTTP. Un serviciu web este deci o aplicaţie a cărei componente pot fi configurate şi executate pe dispozitive distincte.

Câteva trăsături sunt totuşi specifice serviciilor web. Dintre acestea enumerăm următoarele:

- **Infrastructură deschisă**

  Serviciile web sunt folosite cu ajutorul unor protocoale definite de standarde specifice industriei, independente de dezvoltator, precum HTTP şi XML. Acestea sunt ubicue şi înţelese bine. Serviciile web se pot folosi de protocale existente, formate de date cunoscute, politici de securitate. Această caracteristică uşurează folosirea lor şi promovează interoperabilitatea între sisteme.

- **Transparenţa limbajului**

  Serviciile web şi clienţii lor pot comunica indiferent de limbajul de programare în care au fost scrise. Limbaje precum C, Java, Perl, Python, Ruby şi altele oferă biblioteci, utilităţi şi chiar framework-uri care suportă serviciile web.

- **Design modular**

  Serviciile web sunt modulare în design astfel încât servicii noi pot fi generate prin integrarea şi suprapunerea serviciilor existente.

Suntem nevoiţi să răspundem la următoarea întrebare: care este folosul serviciilor web? Unul din avantaje este independenţa faţă de limbaj. Diferite sisteme sunt scrise în limbaje care diferă şi implicit comunicarea între ele devine dificilă. Ne putem imagina cu uşurinţă cum de exemplu, diferenţele de codificare a structurilor de date generează o multitudine de probleme, deloc uşor de rezolvat. Cu ajutorul serviciilor web, aceste siteme eterogene pot fi interfaţate relativ uşor.

Apoi, trebuie să menţionăm că serviciile web sunt inerent sisteme distribuite. Informaţia transmisă de ele este text codificat, de exemplu, în XML. Această informaţie poate fi procesată cu uşurinţă de către celălalt capăt de comunicare. Cu toate acestea, serviciile web pot transmite conţinut binar dacă e necesar.

Sub reportul protocolului de comunicare, serviciile pot fi împărţite în două categorii principale: REST (REpresentational State Stransfer) şi SOAP (Simple Object Access Protocol). În continuare vom trece în revistă cele două categorii principale de servicii, după care vom realiza o comparare a lor.

2.1.1. Servicii REST (REpresentational State Transfer)

Până nu demult, singura referinţă principală asupra acestui tip de servicii a fost lucrarea de doctorat a lui Roy Fielding din 2000 [14]. Primul aspect de notat în ceea ce priveşte serviciile REST, este că spre deosebire de SOAP care foloseşte WSDL [28] pentru a expune un *algoritm,* acest tip de serviciu îşi propune să expună date [6]. Pe web sunt disponibile date codificate sub formă de XML. REST se foloseşte de HTTP pentru a transmite astfel de date.

După cum rezultă din denumirea lor, aceste servicii expun resurse care au o stare. Resursele reprezintă date care sunt suficient de importante pentru a fi referite. De obicei, resursele pot fi stocate pe un calculator sau într-o bază de date ca un flux de biţi: un document, o înregistrare într-o bază de date, etc [6]. Orice resursă are cel puţin un URI (Uniform Resource Identifier). Acesta constă dintr-un şir de caractere folosit pentru a identifica în mod unic o resursă pe Internet. Dacă o informaţie nu are un URI ataşat, atunci practic ea nu există pe web din moment ce nu poate fi accesată. Să reţinem faptul că o resursă poate avea mai multe URI-uri.

Un client HTTP manipulează o resursă prin faptul că se conectează la serverul care o deţine şi trimite o metodă GET împreună cu o cale către resursă. De exemplu, dacă clientul doreşte să obţină textul aflat la locaţia www.example.com/hello.txt, atunci el trebuie să formeze o cerere GET la host-ul www.example.com şi să ceară resurse aflată la locaţia /hello.txt. Presupunând că resurse conţine textul „Hello, world!”, atunci următorul tabel ilustrează cererea clientului şi răspunsul serverului [6].

Tabel 2.1. Exemplu de cerere şi răspuns HTTP

Cerere Client	Răspuns Server

`GET /hello.txt HTTP/1.1`
`Host: www.example.com`
	`200 OK`
	`Content-Type: text/plain`
	
	`Hello, world!`

Prin definiţie, oricare două resurse sunt distincte. Dacă ar fi identice, atunci ar exista doar una singură. Totuşi, două resurse diferite pot indica aceeaşi data, dar să însemne lucruri diferite. De exemplu, putem avea două URI-uri: URI1 şi URI2. URI1 poate să indice înspre ultima versiune a unui program, iar URI2 să indicce înspre versiunea trei a unui program. În cazul în care versiunea a treia este ultima atunci avem aceeaşi dată referită atât de URI1 cât şi de URI2. O singură resursă poate avea mai multe URI-uri.

În lucrarea lor [6], Richardson şi Ruby propun patru concepte care trebuie să stea la baza unei arhitecturii orientate pe resurse (ROA – Resouce Oriented Architecture).

- **Adresabilitate**

  O aplicaţie este adresabilă dacă expune părţile importante care o compun ca şi resurse. Deci, o aplicaţie va expune un URI pentru fiecare informaţie disponibilă. Aceasta de obicei înseamnă un număr infinit de resurse.

- **Lipsa stării** (eng. „statelessness")

  A doua trăsătură principală este aceea de „lipsa stării". Ea se referă la faptul că fiecare cerere HTTP are loc în izolare completă. În momentul în care clientul execută cererea, ea conţine toţi parametri necesari şi serverul nu se bazează pe parametri cererilor anterioare.

- **Starea aplicaţiei versus starea resursei**

  Ce anume înseamnă „stare"? Starea resursei este aceeaşi pentru fiecare client, adică fiecare client va obţine aceeaşi dată expusă cu ajutorul resurei. Starea aplicaţiei însă poate diferi de la un client la altul. De exemplu, un client doreşte să cumpere un obiect reprezentat de o resursă R, pe când altul a şi efectuat tranzacţia. Starea aplicaţiei diferă pentru cei doi clienţi, dar starea resursei e aceeaşi. Starea aplicaţiei este stocată pe partea clientului, iar starea resursei e stocată pe partea de server.

- **Reprezentări ale resurselor**

  În momentul în care împărţim aplicaţia în mai multe resurse, utilizatorii pot construi URI-uri şi accesa aplicaţia exact în locul în care au nevoie. Dar resursele nu reprezintă data propriuzisă. Un server web trebuie să transmită data într-un format specific, într-o serie de octeţi, într-un limbaj specific. Aceasta este o *reprezentare* a resursei.

Acest tip de servicii se bucură de faptul că are o interfaţă unică, dată de protocolul HTTP:

- Obţinerea informaţiei se face prin metoda HTTP GET.
- Crearea unei noi resurse se face prin HTTP PUT pentru un URI nou, sau HTTP POST pentru un URI existent.
- Modificarea unei resurse existente se face cu ajutorul metodei HTTP PUT.
- Ştergerea unei resurse se face cu ajutorul metodei HTTP DELETE.

Imaginea următoare este preluată din [5] şi are rolul de a evidenţia folosirea metodelor HTTP pentru accesul la resurse:

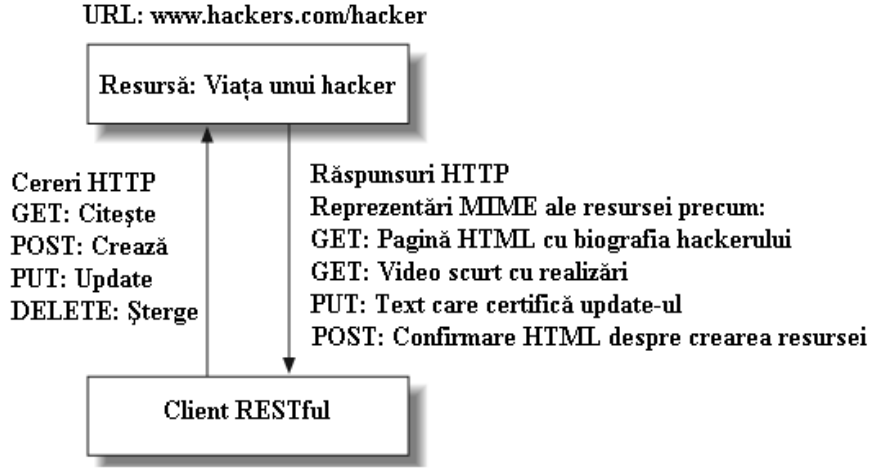


Figura.2.1. O ilustrare a unui sistem REST, comunicând prin HTTP

Următorul tabel ilustrează folosirea acestor verbe (sau metode) HTTP pentru a accesa resursele pe un server [5].

Tabel 2.2. Folosirea verbelor HTTP pentru a accesa resursele pe un server

Verb HTTP/URI	Semnificaţia CRUD
`POST emps`	Crează un nou angajat
`GET emps`	Citeşte o listă cu toţi angajaţii
`GET emps?id=27`	Citeşte o listă singleton a angajatului 27
`PUT emps`	Execută un update asupra angajatului
`DELETE emps`	Şterge lista cu angajaţi
`DELETE emps?id=27`	Şterge angajatul 27

Mai trebuie să amintim noţiunile de siguranţă şi idempotenţă. Siguranţa se referă la faptul că prin accesul unei resurse, starea ei nu a fost modificată astfel încât să defină invalidă. Dintre metodele HTTP, GET şi HEAD sunt sigure. Idempotenţa se referă la faptul că o resursă poate fi accesată de oricâte ori, rezultatul va fi acelaşi. GET, HEAD, PUT şi DELETE sunt idempotente. Aceste două trăsături permit unui client să efectueze cereri sigure pe HTTP. De exemplu, în cazul în care clientul nu a primit un răspsuns la o cerere GET, atunci o mai poate efectua odată – ea este atât sigură cât şi idempotentă!

În concluzie [5], putem afirma că serviciile REST încearcă să simplifice comunicarea folosind HTTP cu tipul său MIME de codificare a datei împreună cu operaţiile CRUD predefinite. Serviciul în schimb poate necesita o putere de calcul ridicată şi algoritmi complicaţi pentru a menţine resursele şi a genera reprezentări adecvate ale lor – de exemplu un document XML poate fi folosit pentru a reprezenta conţinutului unei tabele dintr-o bază de date.

2.1.2. Servicii SOAP (Simple Object Access Protocol)

SOAP [27], este o specificaţie de protocol pentru a schimba informaţia structurată folosită de serviciile web în cadrul reţelelor de calculatoare. Se foloseşte de XML (Extensible Markup Language) pentru a codifica mesajul, iar transportul este cel definit la nivel de aplicaţie dintre care cele mai importante sunt RPC sau HTTP.

De exemplu, dacă dorim să căutăm o informaţie într-o bază de date expusă printr-un serviciu SOAP, atunci creem un mesaj SOAP în care codificăm cererea dorită. Aceasta o trimitem serviciului web care ne returnează un mesaj XML conţinând răspunsul la cererea noastră. În continuare vom prezenta două exemple de mesaje SOAP – unul pentru a efectua o cerere, iar altul reprezentând răspunsul [5].

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
        <req:echo xmlns:req=
             "http://localhost:8080/axis2/services/MyService/">
             <req:category>classifieds</req:category>
        </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Figura. 2.2. Exemplu al unei cereri SOAP

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
    <soapenv:Header>
        <wsa:ReplyTo>
        <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/
                  addressing/role/anonymous
        </wsa:Address>
        </wsa:ReplyTo>
        <wsa:From>
        <wsa:Address>http://localhost:8080/axis2/
               services/MyService</wsa:Address>
        </wsa:From>
        <wsa:MessageID>ECE5B3F187F29D28BC11433905662036
        </wsa:MssageID>
    </soapenv:Header>
    <soapenv:Body>
        <req:echo xmlns:req=
            "http://localhost:8080/axis2/services/MyService/">
             <req:category>classifieds</req:category>
        </req:echo>
    </soapenv:Body>
</soapenv:Envelope>
```

Figura 2.3. Exemplu al unui răspuns SOAP

Acest protocol este independent de platformă şi limbaj, ceea ce constituie un mare avantaj. De asemenea, permite folosirea diferitelor protocoale de nivel de aplicaţie, precum HTTP sau SMTP.

În ciuda acestor avantaje, acest protocol poate fi destul de lent din cauza codificării informaţiei în format XML. Această codificare implică un număr suplimentar de informaţii care trebuie transmise prin fir între cele două capete de comunicare. RMI sau CORBA pot fi mai rapide din acest punct de vedere. Cu toate că SOAP este un standard deschis, nu toate limbajele oferă un suport suficient de bine pus la punct pentru el. De exemplu, Java şi .NET oferă soluţii excelente. În schimb, Python şi PHP au un suport mai slab.

Serviciile SOAP sunt descrise cu ajutorul unor documente WSDL (Web Service Description Language). Ele reprezintă un contract între un serviciu şi consumatorii săi. Pentru o detaliere a structurii unui fişier WSDL se pot consulta [3, 5, 28].

2.1.3. Comparaţie între serviciile REST şi SOAP

După o scurtă trecere în revistă a celor două modalităţi diferite de a codifica serviciile web, [11] enunţă următoarea concluzie pe care o vom ilustra în următorul tabel:

Tabel 2.3. Comparaţie între serviciile de tip REST şi SOAP

	SOAP	REST
Pro	• Independenţă de limbaj, platformă *şi transport* • Folosit pentru medii distribuite de lucru • Standardul cel mai folosit la ora actuală • Mecanisme standard de tratare a erorilor • Extensibil	• Independenţă de limbaj şi platformă • Mai simplu de dezvoltat decât SOAP • Se învaţă mai repede; sunt necesare mai puţine unelte. • Concis: nu e necesar încă un nivel de mesaje • Mai aproape de design-ul şi de

		filosofia Web-ului
Contra	• Mai dificil de implementat • Necesită mai multă informaţie pentru a fi descris • Sunt necesare unelte pentru dezvoltare	• Presupune comunicare punct la punct. Nu e utilizabil în calcul distribuit unde mesajul poate circula prin mai mulţi intermediari • Nu există standarde pentru securitate, mesagerie, politici, etc.

Ca şi o concluzie putem afirma că modul de accesare REST a resurselor este cel optim pentru implementarea temei propuse. Datele sunt privite cel mai natural ca şi resurse pe Internet, deci această abordare este cea mai potrivită.

2.2. API-uri REST existente

În continuare, vom analiza câteva surse de date existente care sunt expuse cu ajutorul serviciilor REST. Vom descrie paşii care sunt necesari pentru a acceasa aceste resurse.

2.2.1. Yahoo

Yahoo îşi expune mai multe funcţii cu ajutorul serviciilor web, precum: căutare web, căutare imagini, căutare ştiri, hărţi, e-mail, etc [29]. Unul dintre API-uri este Yahoo BOSS (Build your Own Search Service). Acesta reprezintă platforma de căutare deschisă publicului. Scopul BOSS este unul simplu: de a stimula inovaţia în industria căutărilor web. Dezvoltatoti software, companii de start-up sau companii mari de Internet pot folosi BOSS pentru a construi produse de căutare web care folosesc index-ul de căutare al Yahoo. BOSS oferă acces la această infrastructură a Yahoo. Combinând diferite idei de business cu datele oferite de această platformă se pot obţine produse originale.

După cum afirmă producătorul [29], politicile prezente limitează inovaţia în acest domeniu, deşi există suficiente API-uri pentru căutarea web. BOSS este diferit, fiind un API deschis cu un număr minimal de limitări şi restricţii. Următorul tabel ilustrează shimbările efectuate de Yahoo între API-ul său anterior de search şi BOSS.

Tabel 2.4. Avantajele folosirii Yahoo BOSS.

	Yahoo Search precedent	**Yahoo BOSS**
Interogări per zi	5.000	Fără limită
Fără restricţii la prezentare	Nu	Da
Posiblitatea re-ordonării	Nu	Da
Îmbinarea conţinutului Yahoo cu cel proprietar	Nu	Da
Termeni cheie/Sugestii de interogare	Nu	Da

Primul pas în vederea utilizătii acestui API îl reprezintă obţinerea unui ID unic pe care îl vom folosi pentru autentificare. Mai multe informaţii se găsesc la pagina web a produslui.

URI-ul folosit de API este următorul:
http://boss.yahooapis.com/ysearch/`{vertical}/v1/{query}?appid=xyz[&param1=val1&param2=val2&etc]`

Următoarea este o listă de argumente generale folosite de acest API, indiferent de tipul căutării (web, image, news).

Tabel 2.5. Argumentele folosite de Yahoo BOSS Web Search

Parametru	**Semnificaţie**
Start	Poziţia de start a primului rezultat. Valoarea minimă e 0. Valoarea implicită e 0.
Count	Numărul total de rezultate returnate. Valoarea maximă e 50. Valoarea implicită e 10.
Lang	Specifică limbajul în care se face căutarea. Valoarea implicită este `"en"`. Trebuie folosit în paralel cu region.
Region	Specifică regiunea căreia aparţine interogarea. Valoarea implicită este "us". Trebuie folosit în paralel cu lang.
strictlang	`Valoarea implicită e 0. Setând valoarea strictlang=1` activează filtrarea strictă după limbaj bazată pe parametrul lang definit în interogare.
Format	Formatul răspunsului. Poate fi setat `"xml"` sau `"json"`. Valoarea implicită e `"json"`.
Callback	Numele funcţiei de callback în care să fie împachetat răspunsul. Parametrul e valid doar dacă formatul e json. Nu există valoare implicită.
Sites	Restricţionează resultatele de cătare la anumite site-uri. Mai multe site-uri terbuie să fie separate prin virgulă.
View	Specificat pentru a obţine informaţie adiţională de la serviciul BOSS respectiv.

În continuare vom prezenta exemple de web search, image search şi news search ale Yahoo BOSS.

- **Yahoo BOSS Web Search**

URL-ul serviciului se găseşte la:

http://boss.yahooapis.com/ysearch/web/v1/{query}?appid={yourBOSSappid}[&param1=val1&param2=val2&etc.

Un exemplu de căutare este următorul:

http://boss.yahooapis.com/ysearch/web/v1/foo?appid={yourBOSSappid}&format=xml

Răspunsul este următorul:

```
<ysearchresponse responsecode="200">
   <nextpage><![CDATA[/ysearch/web/v1/foo?appid={yourBOSSappid}&fo
         rm at=xml&start=10]]>
   </nextpage>
   <resultset_web count="10" start="0" totalhits="29440998"
        deephits="881000000">
        <result>
            <abstract><![CDATA[World <b>soccer</b> coverage
                from ESPN, including Premiership, Serie A, La
                Liga, and Major League <b>Soccer</b>. Get news
                headlines, live scores, stats, and tournament
                information.]]>
            </abstract>
            <date>2008/06/08</date>
            <dispurl><![CDATA[www.<b>soccernet.com</b>]]>
            </dispurl>
            <clickurl>http://us.lrd.yahoo.com/_ylc=X3oDMTFkNXVldG
            yBGFwcGlkA2Jvc3NkZW1vBHBvcwMwBHNlcnZpY2UDWVNlYXJjaA
               RzcmNwdmlkAw--
            </clickurl>
            <size>94650</size>
            <title>ESPN Soccernet</title>
            <url>http://www.soccernet.com/</url>
        </result>
    </resultset_web>
</ysearchresponse>
```

Figura 2.4. Răspuns XML simplu Yahoo BOSS Web Search

- **Yahoo BOSS Image Search**

URL-ul serviciului se găseşte la:

http://boss.yahooapis.com/ysearch/images/v1/{query}?appid=xyz[&param1=val1&param2=val2&etc

Răspuns XML simplu:

```
<ysearchresponse responsecode="200">
   <nextpage>/ysearch/images/v1/soccer?format=xml&start=10&count=
   10<
         /nextpage>
   <resultset_images count="10" start="0" totalhits="4195016"
         deephits="4195016">
         <result>
            <abstract>Now we don't know Crawford from Adam but
               a good friend of WCP-occasional-contributor Beans
               and he somewhat has a point, or at last made us
               laugh while making it.
            </abstract>
            <clickurl>http://us.lrd.yahoo.com/_ylc=X3oDMTFkNXVldG
            JyB
               GFwcGlkA2Jvc3NkZW1vBHBvcwMwBHNlcnZpY2UDWVNlYXJjaAR
               zcmNwdmlkAw-
               G=12f47a1hh/**http%3A//wickedchopspoker.
               blogs.com/my_weblog/images/why_men_love_soccer.jpg
               </clickurl>
```

```
            <filename>why_men_love_soccer.jpg</filename>
            <size>31500</size>
            <format>jpeg</format>
            <height>213</height>
            <date>2006/06/09</date>
            <mimetype>image/jpeg</mimetype>
            <refererclickurl>http://wickedchopspoker.blogs.com/my
            _we
               blog/2006/06/world_cup_musin.html
            </refererclickurl>
            <refererurl>http://wickedchopspoker.blogs.com/my_webl
            og/
               2006/06/world_cup_musin.html
            </refererurl>
            <title>why_men_love_soccer.jpg</title>
            <url>http://wickedchopspoker.blogs.com/my_weblog/imag
            es/
               why_men_love_soccer.jpg</url>
            <width>230</width>
            <thumbnail_height>120</thumbnail_height>
            <thumbnail_url>http://re3.yt-thm-
               a01.yimg.com/image/25/m7/3979077535</thumbnail_url
         >
            <thumbnail_width>130</thumbnail_width>
         </result>
      </resultset_images>
   </ysearchresponse>
```

Figura 2.5. Răspuns XML simplu Yahoo BOSS Image Search

- **Yahoo BOSS News Search**

URL-ul serviciului se găseşte la:

http://boss.yahooapis.com/ysearch/news/v1/{query}?appid=xyz[&param1=val1&param2=val2&etc

Răspuns XML simplu:

```
<ysearchresponse responsecode="200">
   <nextpage>/ysearch/news/v1/soccer?format=xml&start=10&count=10
   </nextpage>
   <resultset_news count="10" start="0" otalhits="8775"
   deephits="8775">
         <result>
            <abstract>June 16 (Bloomberg) -- Adidas AG , the world's
               second - largest sporting-goods maker, will ``clearly
               exceed'' its full-year sales target for soccer-
               related goods and gain share in all major markets,
               Chief Executive Officer Herbert Hainer said.
            </abstract>
            <clickurl>http://www.bloomberg.com/apps/news?pid=2060110
               0&sid=aSSf0jMZtvBU
            </clickurl>
            <title>Adidas Will `Clearly Exceed' Soccer Sales
            </title>
            <language>english</language>
            <date>2008/06/16</date>
            <time>14:21:15</time>
            <source>Bloomberg.com</source>
```

```
                <sourceurl>http://www.bloomberg.com/</sourceurl>
                <url>http://www.bloomberg.com/apps/news?pid=20601100&sid
                   =aSSf0jMZtvBU
                </url>
             </result>
       </resultset_news>
</ysearchresponse>
```

Figura 2.6. Răspuns XML simplu Yahoo BOSS News Search

Mai multe detalii se pot găsi la site-ul de documentaţii Yahoo! BOSS [29, 30].

2.2.2. Microsoft

API-ul folosit de Microsoft este Live Search API v.2.0. Acest API permite incorporarea într-o aplicaţie sau pagină web a funcţionalităţii oferite de motorul de căutare Microsoft. Ultima versiune include: endpoint-uri HTTP care oferă rezultate în JSON sau XML, suport îmbunătăţit pentru SOAP şi posibilitatea de a folosi reclame pentru a obţine un profit.

Endpoint-ul HTTP facilitează folosirea metodei HTTP GET pentru a trimite cereri. Pentru a executa această cerere trebuie suplinit un ID al aplicaţiei (AppID), determinate valorile parametrior necesari şi a parametrilor opţionali, alegerea unui protocol, trimiterea cererii şi procesarea rezultatelor.

Parametrul AppID permite aplicaţiei să valideze cererea. Pentru a obţine o valoare pentru acest parametru, trebuie accesat Live Search Developer Center [22] şi înregistrat folosind de Windows Live ID [23]. Procesul este descris pe site şi este simplu de completat.

În afară de parametrul AppID, trebuie introduşi şi parametrii Query şi Sources. Parametrul Query conţine şirul de caractere care se doreşte căutat. Pentru cererea care o vom forma ca şi exemplu folosim „sushi”. Parametrul Sources indică una sau mai multe surse de tipul SouceType. API-ul include un număr de surse diferite. Pentru a folosi mai multe surse în cătuare, ele trebuie concatenate folosind caracterul „+”. Dacă SourceType este de tip Web, atunci resultatele vor fi acelea ale unei căutări web. Alte valori valide ale SourceType sunt Image, News, Spell şi InstantAnswer. Pentru exemplu vom folosi tipul de sursă Web. În următorul tabel vom prezenta tipurile diferite de surse:

Tabel 2.6. Tipurile diferite de surse oferite de Microsoft

SourceType	**Descriere**	**Exemplu de Query**
Web	Caută conţinut web	Sushi
Image	Caută conţinut de tip imagine	Sushi
News	Caută conţinut de tip ştiri	Sushi
InstantAnswer	Caută Encarta online	What is sushi x*5=7 2 plus 2
Spell	Caută dicţionarul encarta pentru greşeli de scriere	Coffee
Phonebook	Caută într-o carte de telefoane	Sushi in los angeles

În afară de cei trei parametri necesari, fiecare SourceType are parametri opţionali care pot fi folosiţi pentru a rafina cererea. Detalii suplimentare despre toţi parametri şi funcţiile lor pot fi găsite la [21].

O cerere către un endpoint HTTP constă în executarea metodei GET către URI-ul corespunzător. Există două URI-uri, unul pentru rezultate XML şi altul pentru rezultate JSON. Acestea sunt: http://api.search.live.net/xml.aspx, respectiv http://api.search.live.net/json.aspx. Pentru cererea noastră demonstrativă, vom folosi endpoint-ul XML căutând pe web "sushi". URI-ul cererii este deci următorul: http://api.search.live.net/xml.aspx?Appid=[AppId]&Query=sushi&Sources=web.

Formatul rezultatelor diferă de la o cerere la alta şi depinde de parametri care au fost indicaţi, în mod principal de valoarea parametrului Source. Rezultatul include un header şi un body. Elementul documentului este întotdeauna de tipul SearchResponse cu un element copil Query care conţine şirul de caractere care a generat rezultatul. Aceste două elemente formează header-ul. Body-ul urmează header-ul şi conţine rezultatele căutării.

Pentru a lucra cu mai multe tipuri de surse de date simultan, trebuie să indicăm aceasta în metoda GET, modificând valoarea parametrului Sources. De exemplu, dacă dorim să căutăm în surse web şi imagini simultan, valoarea parametrului Sources va fi sources=web+image. API-ul poate fi folosit şi din medii programatice. Până acum, exemplele au fost rulate cu ajutorul unui browser. Un mediu programatic este unul ideal pentru a folosi acest API.

Microsoft limitează accesul la serviciul său, impunând câteva restricţii:

- Trebuie afişate toate rezultatele returnate de căutare.
- Rezultatele trebuie afişate în cadrul unei aplicaţii cu interfaţă utilizator sau web.
- Trebuie menţionat că LiveSearch a oferit rezultatele de căutare.
- Aplicaţia nu poate executa mai mult de 7 interogări per secundă.
- În cazul în care data oferită de Microsoft va fi folosită împreună cu altă sursă de date, ele trebuie diferenţiate în mod clar.
- Nu se pot reordona rezultatele oferite de un SourceType.

2.2.3. Baze de date expuse prin REST

După cum rezultă din secţiunile anterioare, datele pot fi expuse prin servicii REST într-o reţea. Dat fiind URL-ul lor, ele pot fi accesate simplu prin HTTP. Să presupunem că avem o bază de date care conţine abonaţii unei reţele de telecomunicaţii. Fiecare abonat are un identificator unic (codul abonatului), nume, numar telefon, adresă. Să mai presupunem că baza de date este publică şi e accesibilă la adresa www.phonebook.com. Pentru a afla datele despre un abonat care e identificat prin codul 2, am putea executa următoarea cerere HTTP: GET www.phonebook.com/userid=2.

Serverul poate răspunde printr-un document XML care conţine datele despre utilizator. Structura documentului XML poate fi următoarea:

```
<phonebook-response>
    <user>
        <id>2</id>
        <name>John Doe</name>
        <address>Home</address>
        <phone>444222</phone>
    </user>
</phonebook-response>
```

Figura 2.7. Document XML conţinând date despre un utilizator

În cazul în care am dori să obţinem o listă cu toţi utilizatorii, putem trimite următoarea cerere: GET www.phonebook.com/users, iar răspunsul poate fi următorul:

```
<?xml version="1.0" encoding="UTF-8"?>
<phonebook-response>
    <userlist>
        <user>
            <id>1</id>
            <name>Mark Twain</name>
            <address>Address</address>
            <phone>111333</phone>
        </user>
        <user>
            <id>2</id>
            <name>John Doe</name>
            <address>Home</address>
            <phone>444222</phone>
        </user>
    </userlist>
</phonebook-response>
```

Figura 2.8. Document XML conţinând date despre o listă de utilizatori

2.3. Principii de design ale SOA (Service Oriented Architecture)

Odată ce am definit serviciile web şi am analizat modul în care le putem cataloga şi cum acestea pot expune informaţii web, vom defini principiile arhitecturale care stau la baza definirii unei aplicaţii orientate servicii web. Dezvoltarea unei astfel de aplicaţii trebuie să ţină cont de anumite aspecte arhitecturale. Acest tip arhitectural se numeşte SOA – Service Oriented Architecture. În lucrarea lor [7], Ronsen et. al. enumeră principiile care stau la baza dezvoltării unei astfel de aplicaţii. Vom relua aceste principii în materialul care urmează. Ele vor sta la baza dezvoltării aplicaţiei noastre.

2.3.1. Necesitatea SOA

Înainte de a trece la definirea SOA şi la metodele sale de abordare, va trebui să trecem în revistă modul în care sitemele erau proiectate în trecut, problemele inerente şi provocările prezente la care trebuie să adere acest tip arhitectural.

SOA nu este un concept nou. Există sub diferite forme încâ din anii 90. Companiile puteau implementa servicii folosindu-se de CORBA şi DCOM. Cu toate acestea, pe măsură ce complexitatea tehnologiilor creştea, rata de succes a implementării unor astfel de arhitecturi scădea, după cum rezultă din următoarea figură. Motivele cărora se datorează eşecul sunt multiple.

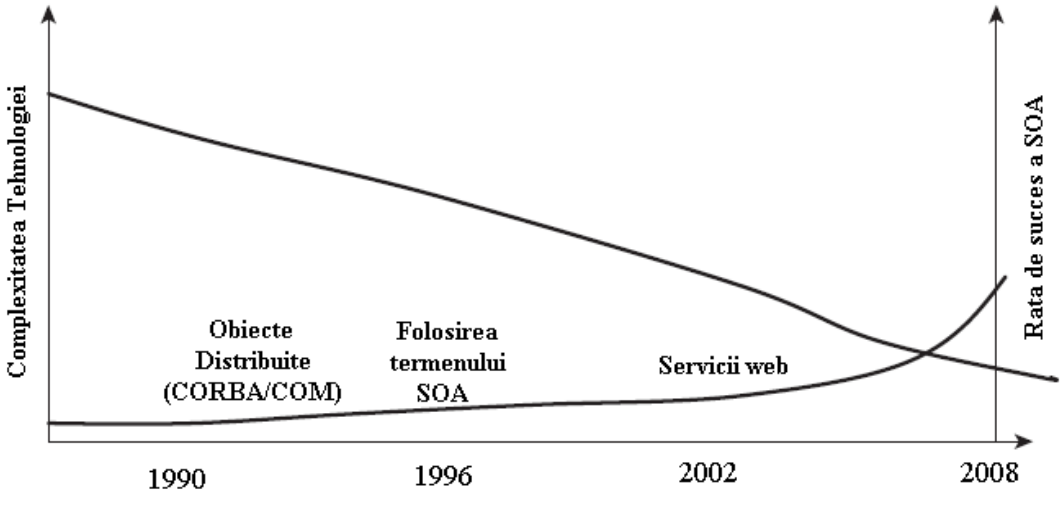


Figura. 2.9. Rata de succes a SOA în timp

În primul rând, tehnologiile menţionate mai sus erau prea dificile pentru a fi stăpânite de un programator obişnuit. Calculul distribuit prin CORBA şi DCOM ridică prea multe dificultăţi pentru a fi folosit în scară largă. O altă problemă a constat în faptul că industria nu a reuşit încă să definească în ce anume constă un serviciu bun. Caracteristicile corecte ale unui serviciu nu au fost încă definite. Odată stabilite aceste detalii, trebuia definită o abstracţie a serviciului, care la rândul ei trbebuia implementată. Resursele necesare creării unui serviciu erau atât de mari, încât multe dintre încercări au eşuat. Astăzi situaţia este mult îmbunătăţită. Serviciile web sunt mult mai uşor de folosit decât tehnologiile precedente. Cunoştinţele implicite necesare sunt acum incorporate în platformă (Java, .NET, etc.).

Obsevaţiile de mai sus implică următoarele cerinţe arhitecturale principale:

- **Arhitectura de referunţă**

  Crearea şi menţinerea unei arhitecturi de referinţă este una dintre cele mai importante şi dificile practici SOA. Următoarele sunt componente esenţiale

  - Definirea modului în care serviciile se integrează în arhitectură.
  - Oferirea unuei separări clare între conceptele de business, aplicaţie şi tehnologie.
  - Arhitectura de referinţă trebuie să fie integrată în procesul de dezvoltare.
  - Specificarea unei ierarhii şi clasificări (taxonomii) a serviciilor.

- **Definirea unei semantici comune**

  Definirea unei semantici comune şi a unui model de date comun este cheia reuşitei obţinerii agilităţii şi flexibilităţii. Semantica comună ar trebui să:

  - Identifice informaţia care trebuie folosită în comun.
  - Definească înţelesul şi contextul informaţiei.
  - Identifice tehnici de mapare a semanticii enterprise la modelele de date existene ale aplicaţiei.

- **Guvernarea** (eng. „governance")

  Guvernarea a fost definită ca arta şi disciplina de a gestiona rezultatele prin relaţii structurate, proceduri şi politici. Constă dintr-un set de politici la care aderă consumatorii şi ofertanţii de servicii. Guvernarea trebuie să includă:

  - Politici pentru definirea şi extinderea serviciilor.
  - Identificarea rolurilor şi a responsabilităţilor.
  - Constrângerea la o politică ar trebui să facă parte din registrul serviciului.
  - Reviziurea interfeţelor serviciilor trebuie să aibă loc după anumite standarde.
  - Revizuirea arhitecturală a serviciilor şi soluţiilor pentru a ne asigura că sunt conforme cu SOA.

- **Modelarea proceselor business** (eng. „Business Process Modeling" – BPM)

  Procesele business sunt supuse schimbărilor frecvent. Acestea ar trebui să:

  - Fie specificate conform unor modele de business şi executate într-un sistem de management al proceselor business.
  - Fie compuse din activităţi care să reprezinte servicii de business.
  - Schimbe informaţia (în interor şi exterior) sub formă de documente care sunt construite pe baza modelului de informaţii comun.

- **Descoperirea serviciilor la faza de design**

  Pentru a folosi serviciile, trebuie să fie găsite şi examinate după criterii de performanţă, calitatea serviciilor, etc. Locaţia serviciilor trebuie realizată la faza de design. Următoarele funcţii sunt necesare:

  - Un catalog cu servicii disponibile.
  - Capabilităţi de căutare sofisticate pentru a identifica serviciile potenţiale.
  - Metrici pentru analiza serviciului.
  - Integrare directă în mediul de dezvoltare.

- **Dezvoltarea bazată pe model** (eng. „Model based development")

  Aceasta este o practică bună în general, dar în mod special în SOA. Modelele oferă o posibilitate de a conceptualiza un sistem fără a intra în detalii şi pentru a descrie părţile sale majore şi relaţiile dintre ele. O astfel de abordare trebuie să dea dovadă de următoarele:

  - Un nivel mai ridicat de abstractizare a dezvoltării software şi abilitatea de a vizualiza design-ul serviciilor şi al software-ului.
  - Suport pentru un limbaj specific domeniului (Domain Specific Language – DSL) pentru implementarea SOA.
  - Separarea preocupărilor de business, servicii şi tehnologie.

2.3.2. Fundament arhitectural al SOA

Arhitectura software înseamnă descrierea unui sistem software în termenii componentelor majore, a relaţiilor şi a informaţiei care circulă între ele. Un obiectiv fundamental al arhitecturii software este acela de a gestiona complexitatea sistemelor şi a modificărilor ce au loc.

Un stil arhitectural se referă la o familie de arhitecturi care au în comun un set de atribute şi principii şi modul în care acestea sunt combinate pentru a atinge nişte obiective de nivel înalt. Exemple de stiluri arhitecturale sunt: client/server, 3-tier, n-tier, EAI (enterprise application integration).

SOA poate fi definit ca un stil arhitectural care promovează conceptul de serviciu enterprise aliniat unui business ca şi unitate fundamentală de design, dezvoltare şi compunere a soluţiilor enterprise. SOA este deci un stil arhitectural pentru a construi soluţii enterprise bazat pe servicii. Mai specific, SOA este interesat de construirea de servicii aliniate unui anumit business care se combină în procese de business de nivel mai ridicat în conextul unei aplicaţii enterprise. Oricine poate crea un serviciu. Nu aceasta e problema. Noi ne propunem să creăm servicii reutilizabile care se pot combina în procese de business.

În termenii SOA, principalele componente sunt:

- Procesele – funcţii business de nivel înalt
- Serviciile – unităţi modulare care oferă funcţionalitate business
- Integrare – conectarea şi expunerea aplicaţiilor ca servicii
- Sisteme existente – aplicaţii care vor fi abstractizate prin servicii
- Documente – unităţi de informaţie business de nivel înalt
- Semantică – înţelesul informaţiei care este schimbată în proces
- Transformări – conversia informaţiei dintr-un format în altul
- Comunicare – capacitatea serviciilor de a comunica între ele

În continuare vom descrie elementele arhitecturale ale SOA

- **Resurse enterprise şi sisteme operaţionale**

  Acest strat constă în aplicaţii existente. Ele oferă tranzacţii care reprezintă unităţi logice de lucru în sistemul operaţional pe care îl reprezintă. Operaţiile au o interfaţă specifică, structurată şi returnează răspunsuri structurate. Data la acest nivel rezidă în aplicaţii existente şi baze de date.

- **Servicii de integrare**

  Oferă integrare şi comunicare între aplicaţiile existente. Separarea între serviciile de integrare şi cele de business este critică pentru a menţine un mediu de lucru flexibil.

- **Servicii de business**

  Acestea oferă o funcţionalitate la nivel înalt de tip business în aplicaţie. Acest nivel interfaţează cu şi abstractizează nivelul inferior, realizând trecerea dintre serciciu şi sistem existent. Ele oferă o implementare a operaţiilor de business aferente.

- **Procese business**

  Constau dintr-o serie de operaţii care sunt executate într-o ordine după un set de reguli de business. Deseori, ele sunt descrise într-un model de business (BPM) şi executate într-un sistem de management specializat al proceselor (BPMS). De regulă implică mai multe invocări de servicii.

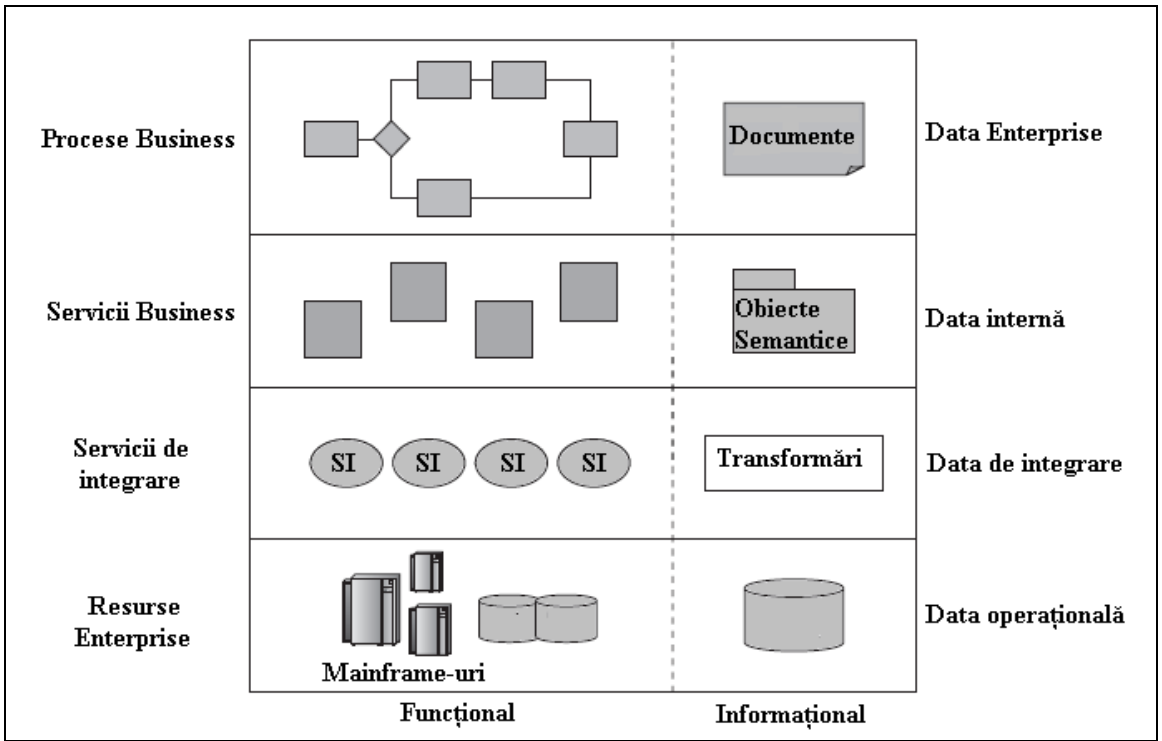


Figura 2.10. Elemente Ahitecturale ale SOA.

2.3.2.1. Comparaţie între SOA şi EA (Enterprise Architecture)

În această secţiune vom trece în revistă o scurtă comparaţie între modelele SOA şi Enterprise Architecture respectiv Software Arhictecture. Enterprise Architecture are la bază un framework conceptual care divide arhitectura în porţiuni gestionabile. Abordările comune realizează această împărţire în arhitectura business, arhitectura informaţiei, arhitectura aplicaţiei şi arhitectura tehnologiei. Arhitectura business defineşte cadrul general al serviciilor. Modelul business este împărţit adesea în grupuri de servicii şi probabil servicii în cadrul acelor grupuri. Serviciile trebuie să deţină o semantică comună. Pentru a putea folosi serviciile în cadrul arhitecturii, ele trebuie să aibă o structură similară şi să îndeplinească roluri relativ similare. Aceasta necesită o arhitectură a aplicaţiei care descrie structura soluţiilor enterprise, elementele arhitecturale care formează structura respectivă, regulile pentru folosirea elementelor şi relaţiilor dintre ele împreună cu rolurile şi responsabilităţile fiecăruia dintre ele. Arhitectura tehnică pune la dispoziţie detaliile tehnice care oferă suport pentru arhitectura noastră.

Atât EA cât şi SOA sunt preocupate de soluţii enterprise în vederea alinierilor soluţiilor IT pentru cerinţe, strategii şi procese comune. Nu este o coincindenţă faptul că cele două abordări sunt similare. În următoarea figură putem observa echivalenţa dintre cele două tipuri arhitecturale.

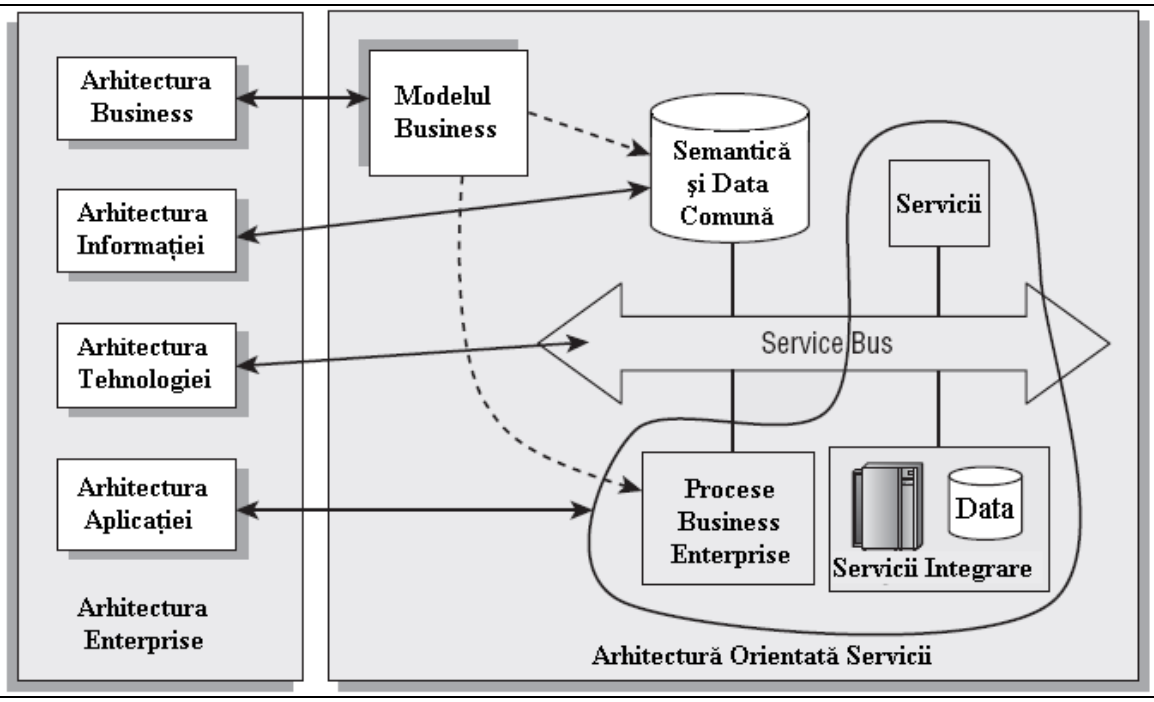


Figura 2.11. Comparaţie între SOA şi EA

2.3.2.2. *Definirea unui serviciu din perspectiva SOA*

Serviciile bine definite prezintă următoarele caracteristici:

- **Modularitate şi granularitate**

  În SOA, procesele business sunt decompuse în servicii modulare. Acestea pot fi la rândul lor compuse din alte servicii modulare şi pot fi combinate pentru a crea noi servicii. Granularitatea este o calitate a bogăţiei funcţionale a unui serviciu. Aceasta presupune că un serviciu înglobează o funcţionalitate într-o operaţie. Astfel se reduce comlexitatea şi folosirea lăţimii de bandă prin faptul că se reduc numărul de paşi necesari pentru a îndeplini o activitate business dată. Deseori ea este obţinută prin combinarea unor sarcini mai mici în altele mai mari.

- **Încapsulare**

  Serviciile impun o separare strictă între interfaţă (*ce* anume îndeplineşte serviciul) şi implementare (*cum* anume îndeplineşte sarcina respectivă). Încapsularea ascunde detaliile interne de implementare şi structurile de date faţă de operaţiile publicate şi modelul semantic.

- **Cuplare joasă**

  Cuplarea descrie numărul de dependinţe între un consumator şi un ofertant de serviciu. Cuplarea joasă presupune un număr mic şi bine gestionat de dependinţe. Serviciile cu un grad ridicat de cuplare au multe dependinţe cunoscute şi necunoscute. Gradul de cuplare afectează în mod direct flexibilitatea şi extensibilitatea unui sistem.

- **Izolarea responsabilităţilor**

  Serviciile sunt responsabile pentru sarcini discrete sau pentru gestionarea resurselor. O caracteristică importantă a design-ului serviciilor este izolarea responsabilităţilor pentru funcţii specifice sau informaţii unui singur serviciu. Acesta oferă un singur loc pentru ca fiecare funcţie să fie îndeplinită, oferind consistenţă şi reducând redundanţa.

- **Autonomia**

  Autonomia este caracteristica ce permite serviciilor să fie rulate, modificate şi menţinute independent unul de celălalt şi de soluţiile care le folosesc. Ciclul de viaţă al unui serviciu autonom este independent de al altor servicii.

- **Refolosirea**

  Împreună modularitatea, încapsularea, cuplarea joasă, anatomia şi izolarea responsabilităţilor permit serviciilor să fie combinate în procese de business multiple sau accesate de mai mulţi consumatori de servicii din locaţii diferite în contexte multiple. În alte cuvinte, serviciile sunt împărţite şi refolosite în construcţia proceselor şi serviciilor compuse.

- **Descoperirea dinamică şi legarea**

  Serviciile pot fi descoperite la design-time prin folosirea unui registru de servicii pentru design-time. Deşi este teoretic posibilă descoperirea serviciilor în mod dinamic la run-time, încă mai sunt multe obstacole de trecut pentru a ajunge la acest deziderat. Cu toate acestea, consumatorii de servicii pot fi legaţi de ofertanţii de servicii la run-time. În acest scenariu, consumatorul interoghează registrul după un serviciu specific fiind rutat şi legat către ofertantul specific de servicii. Legarea dinamică permite cuplarea joasă şi permite capabilităţi suplimentare precum medierea.

- **Lipsa stării** (eng. „statelessness")

  Serviciile nu reţin operaţia precedentă şi nu sunt interesate de operaţia viitoare. Acest tip de servicii oferă un grad ridicat de flexibilitate, scalabilitate şi încredere.

- **Compozabile**

  Serviciile pot fi compuse din alte servicii şi în schimb, pot fi combinate cu alte servicii pentru a forma servicii noi sau procese business.

- **Guvernate de politici**

  Relaţiile între consumatorii de servicii şi ofertanţii de servicii sunt guvernate de politici şi înţelegeri la nivel de servicui (Service Level Agreements – SLA). Politicile descriu cum diferiţi consumatori au permisiunea de a interaţiona cu serviciul.

- **Independenţa de locaţie, limbaj şi protocol**

Serviciile au menirea de a fi transparente la locaţie şi independente faţă de protocol şi platformă. În alte cuvinte, ele sunt accesibile de către orice utilizator autorizat, pe orice platformă, de la orice locaţie.

2.3.3. Design-ul aplicaţiilor SOA

Am definit SOA ca fiind un stil arhitectural care promovează conceptul de servicii aliniate la conceptele de business ca şi fundament al design-ului unităţilor, dezvoltării şi comunerii soluţiilor enterprise. În continuare vom descrie metodologia de dezvoltare a unor aplicaţii SOA şi paşii majori ai metodologiei. Pentru fiecare pas se vor specifica obiectivele sale, cerinţele şi activităţile necesare. Detaliile acestor paşi se vor regăsi în secţiunile următoare.

Principalele activităţi în vederea realizării unei arhitecturi SOA sunt crearea unei arhitecturi de referinţă constând din specificarea formală a serviciilor, a tipului şi a caracteristicilor lor. Următorul pas constă în definirea unui model semantic comun astfel încât serviciile să poată interacţiona. Aceşti doi paşi se află într-o continuă dezvoltare şi adaptare. Identificarea serviciilor foloseşte modelul business şi cel informaţional pentru design-ul serviciilor. În plus, contextul enterprise general, după cum e descris în registrul de servicii, ajută în identificarea serviciilor existente şi în alocarea responsabilităţilor pentru noile servicii. Specificarea serviciilor se referă la ce anume sunt serviciile şi care sunt operaţiile lor. Ele oferă descoperirea serviciilor în timpul identificării serviciilor şi a design-ului proceselor de business şi a specificării detaliilor pentru implementare. După aceasta urmează implementarea serviciilor. Există două metode principale pentru a realiza acest pas: cumpărare sau dezvoltare. Pe parcursul dezvoltării serviciilor, am descris o abodare de mijloc, între top-down şi bottom-up, care s-a dovedit a fi cea mai eficientă.

Acestea sunt aspectele care fac parte din ciclul de dezvoltare a unei SOA.

2.3.3.1. Definirea regulilor de business

Arhitectura Business (eng. „Business Architecture” – BA) este un element cheie în obţinerea convergenţei între cerinţe şi soluţia IT specifică. Aceasta oferă modalităţi pentru a specifica strategia şi obiectivele. În plus, oferă modalităţi de a măsura rezultatele conform acestor obiective şi în final, oferă o modalitate de a specifica procesele business care oferă aceste rezultate. Există două seturi de întrebări care trebuie puse: în primul rând trebuie să ne întrebăm care sunt cerinţele buiness, după care trebuie să ne întrebăm care sunt procesele, activităţile, serviciile şi informaţiile necesare pentru un proiect specific? Acesta este domeniul BA.

Gestionarea proceselor de Business (eng. „Business Process Management” – BPM) presupune alinierea obiectivelor business cu sisteme IT existente. Fiecare proces business este modelat ca fiind compus dint-ro serie de responsabilităţi. Aceste responsabilităţi sunt de regulă implementate ca servicii de business în cadrul soluţiei finale. BPM oferă o abstracţie pentru a defini procesele business împreună cu alte capabilităţi. În schimb, SOA oferă modalităţi prin care aceste servicii pot fi combinate pentru a crea soluţii agile, flexibile. SOA fără BPM este capabilă să creeze servicii consistente şi refolosibile, dar nu are mecanismul pentru a le transforma în soluţii complexe care să interacţioneze.

Un BMP nu este doar o modalitate grafică de a representa procesle business. Modelul este o specificare formală a acelui proces. Aceste modele pot fi programe de sine stătătoare. Când procesle sunt expuse prin limbaje formale de modelare ca şi BPMN atunci acestea pot fi executate prin compilarea modelului BPMN într-un limbaj executabil, cum ar fi BPEL.

Procesele business sunt scrise plecând de la modelul şi cerinţele business. Drept consecinţă, serviciile pot fi organizate plecând de la mai multe criterii. Unul dintre criterii este acela al responsabilităţii pentru serviciu. Serviciile sunt clasificate în funcţie de ofertantul lor. Un alt criteriu este acela al competenţelor şi responsabilităţilor lor. Aceasta ajută la organizarea serviciilor după o coeziune funcţională care poate oferi servicii mai flexibile şi reutilizabile. O altă modalitate (probabil cea mai comună) se referă la organizarea serviciilor în funcţie de domeniile lor de apartenenţă. Domeniile lor se referă la o „lume" reală sau ipotetică în care există un set de obiecte care respectă un set de reguli şi politici proprii. Domeniul presupune o arie de interese comună şi focalizată pe anumite sarcini. Acestea sunt reprezentate în diagramele UML prin stereotipul <<domain>>. Partiţionarea în funcţie de domenii mai este numită şi partiţionarea în funcţie de aspect.

Inventarul (registrul) de servicii (eng. „service inventory") oferă un mecanism pentru a organiza toate serviciile din soluţia enterprise. Salvează serviciile şi întreg setul de relaţii dintre ele. Acesta descrie o „hartă a responsabilităţilor". Ar trebui să descrie setul întreg de servicii împreună cu responsabilităţile grupurilor diferite de servicii în cadrul unui domeniu. Inventarul de servicii oferă un răspuns la două întrebări fundamentale:

— Ce servicii există? În faza de design trebuie să ştim care sunt serviciile existente, ce operaţii au şi cum relaţionează cu alte servicii.
— Ce operaţii paote executa un serviciu? Aceasta este probabil cea mai importantă funcţie. Când o cerinţă a serviciului a fost identificată, trebuie să determinăm cum anume se integrează în setul general de servicii.

În concluzie, putem afirma că un concept fundamental al SOA este alinierea businessului la cerinţele IT. Mai întâi definim care sunt funcţiile business, după care le mapăm pe serviciile business. Aceste servicii sunt compuse din mai multe servicii care interacţionează. Procesele business pot fi descrise cu ajutorul unui limbaj şi rulate într-un mediu de execuţie. Serviciile individuale sunt stocate într-un inventar (registru) de servicii care permite descoperirea lor împreună cu funcţionalitatea ce o oferă.

2.3.3.2. Design-ul interfeţei serviciilor

Interfaţa unui serviciu descrie modul în care sunt expuse operaţiile serviciului în vederea accesului. Un serviciu este o combinaţe între interfaţa şi implementarea sa. Aceasta înseamnă că serviciul ascunde detaliile implementării, interfaţa serviciului exprimă operaţiile şi serviciul manipulează informaţie. Această informaţie este derivată dintr-un model semantic comun. Aceste aspecte sunt evidenţiate în Figura 2.12.

O preocupare improtantă a design-ului interfeţei constă în stilul interacţiei dintre consumator şi producător. Acesta descrie tipul semnăturilor precum şi tipul sincronizării dintre mesajele de cerere şi răspuns. Câteodată, stilul de interacţiune e descris ca şi un tipar de schimb al mesajelor (eng. „Message Exchange Pattern" – MEP). Există două preocupări principale ale MEP: modul în care informaţia este transmisă şi tipul sincronizării mesajului.

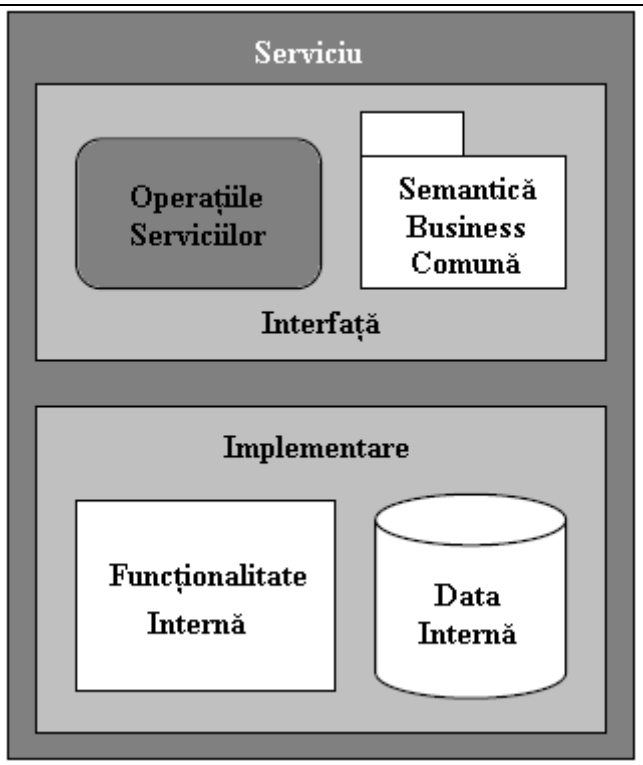


Figura 2.12. Anatomia unui serviciu

O clasificare după tipul schimbării informaţiei este următoarea (după cum rezultă şi din următoarea figură):

- **Schimb de parameteri**

  În acest stil, semnătura operaţiei conţine unul sau mai mulţi parametri individuali. Un exemplu de semnătură poate avea următoarea structură:

  ```
  Response = service_operation(parameter1, parameter2, …);
  ```

  Datele de input sunt transmise ca parametri care au tipuri speciale. Răspunsul poate avea o singură valoarea sau un tip de date complex. Aceasta este cea mai simplă metodă de interacţiune. Este optimă pentru servicii cu nivel mic de granularitate şi e cel mai des înâlnită la serviciile de utilitate (eng. „utility service").

- **Schimb de documente**

  Semnătura operaţiei conţine un document de cerere pentru input şi un document pentru răspuns (output). Un exemplu de astfel de semnătură este următorul:

  ```
  Response_document = service_operation(request_document);
  ```

- **Schimb de date**

  În acest caz, ne interesează să obţinem date despre o entitate. Se specifică identificatorul entităţii şi rezultatul primit constă în informaţii despre entitatea respectivă. O variaţie a acestui tip de acces constă în furnizarea unei interogări în locul identificatorului.

```
Response_data = get_operation(entityID);
```

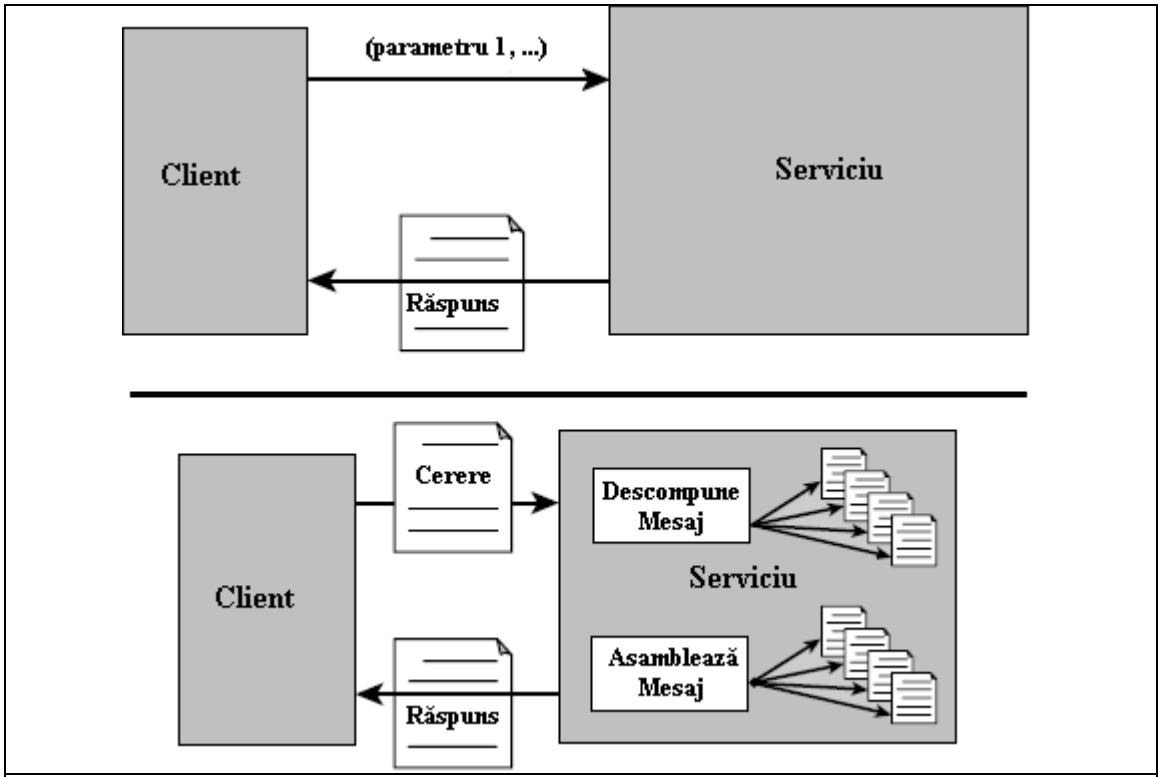


Figura 2.13. Transferul informaţiei între servicii.
Sus: Schimb de parametri. Jos: Schimb de documente.

O clasificare după analiza modul în care mesajele sunt sincronizate este următoarea:

- **Reqest/Reply**

  Acesta e cel mai comun mod de a invoca servicii. În cazul cel mai simpu, se trimite o cerere care primeşte un răspuns într-un mod sincron. În momentul în care serviciul a procesat cererea, trimite răspunsul.

- **Bazat pe evenimente**

  O alternativă la request/reply este abordarea bazată pe evenimente. O modalitate comună este cea publish/subscribe. Acest tip de abordare se bazează pe un intermediar, pe un „event broker", care primeşte notificări de la surse care publică evenimente (eng. „publish") şi informează toate părţile interesate (eng. „subscribers").

În continuare vom menţiona câteva aspecte principale în ceea ce priveşte procesul de design al interfeţelor serviciilor.

- **Izolarea responsabilităţilor**

  Serviciile trebuie să aibă un nivel scăzut de cuplare pentru a putea fi utilizate în scenarii diferite. Serviciile cu funcţionalităţi apropiate trebuie grupate.

- **Identificarea granularităţii**

  O decizie importantă este aceea a granularităţii operaţiilor. O operaţie a unui serviciu ar trebui să corespundă funcţionalităţii aşteptate de consumator. Deci, trebuie să răspundem la următoarele întrebări: care este modul de folosire al serviciului? Care e responsabilitatea serviciului? Care este domeniul de vizibilitate al serviciului?

- **Interfeţe stateless**

  Interfeţele trebuie să fie cât mai statless cu putinţă. Motivul importanţei acestei trăsături este contribuţia ei decisivă în domeniile scalabilităţii, gradului de încredere al serviciului, etc.

- **Excepţiile**

  Toate excepţiile trebuie să fie definite la nivelul interfeţei şi excepţiile comune trebuie să fie definite identic pentru toate serviciile.

2.3.3.3. Design-ul implementării serviciilor

Pînă în acest moment, interesul nostru faţă de servicii a fost unul mai degrabă extern. Nivelul de business al serviciului este responsabil pentru a implementa logica de business. Nivelul pentru acces la resurse conţine logica necesară pentru a accesa resursele enterprise care sunt folosite de implementarea serviciului. Responsabilitatea nivelului de informaţie constă în validarea sintactică a parametrilor şi a transformării datei într-un format care să fie prelucrat de nivelul de business. Responsabilitatea nivelului de business este procesarea necesară pentru a îndeplini cererea împreună cu validarea semantică a datei primite de la layer-ul de interfaţă. Responsabilitatea layer-ului de resurse constă în transformarea datei într-un format procesabil de resursă şi accesul resursei.

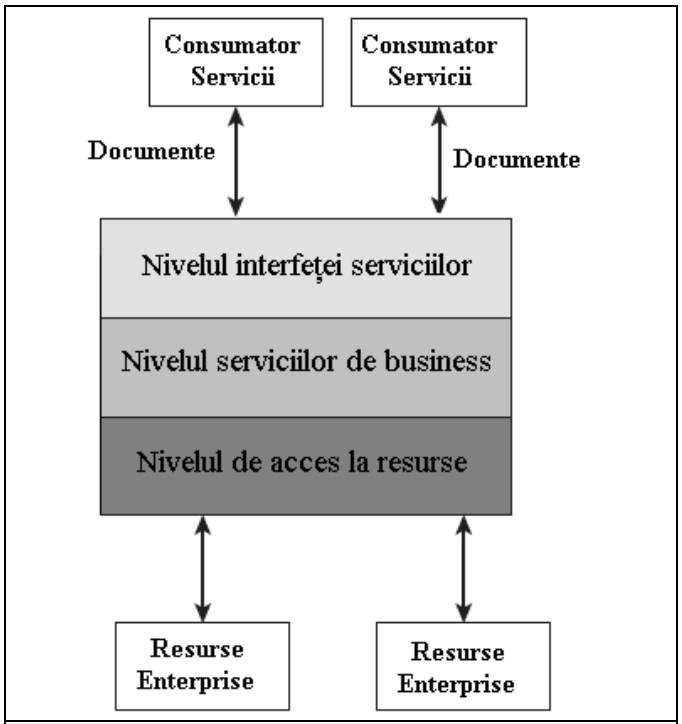


Figura 2.14. Arhitectura de bază a unui serviciu.

2.3.3.4. Compunerea serviciilor

Compunerea performantă a serviciior este o formă de artă construită pe nişte principii arhitecturale şi design pattern-uri solide. Compunerea serviciilor presupune interacţiunea serviciilor cuplate jos rezultând un serviciu compus.

- **Orchestrarea**

  Orchestrarea descrie fluxul de lucru (eng „workflow") al serviciilor, incluzând logica de business şi interacţiunile. Punctul de referinţă al orchestrării este un singur coordonator. WS-BPEL (Web Service Business Process Execution Language) este un limbaj de orchestrare care e folosit pentru a compune serviciile web. Limbajele pentru orchestrare pot fi folosite pentru a scrie „scripturi de orchestrare" care pot fi executate la run-time de un coordonator bazat pe reguli şi secvenţe.

- **Coreografia**

  Descrie secvenţa de mesaje între servicii, gestionând schimbul public de mesaje şi a stării conversaţionale. Spre deosebire de orchestrare, care e privită din punctul de vedere al unui coordonator principal, coreografia se focalizează pe schimbul de mesaje din perspectiva unui observator neutru. Fiecare serviciu implicat în coreografie trebuie să fie conştient de procesul de business, când să execute operaţiile şi modul de interacţionare. WS-CDL (Web Service Choreography Description Language) este un limbaj care descrie astfel de colaborări.

Procesele business sunt orientate pe proces, orchestrând servicii de nivel mai scăzut. Deoarece cerinţele business se schimbă des, e importantă separarea implementării unor astfel de procese de regulile combinărilor serviciilor. Compoziţia se poate baza pe BPM, dar nu este totuna cu BPM!

Există metode multiple pentru a compune servicii. În continuare, vom enumera câteva dintre metodele des folosite:

- **Compunere Programatică**

  La o primă analiză, se pare că folosirea unui limbaj de programare este cea mai simplă variantă pentru a obţine compunerea serviciilor. Această abordare însă crează o implementare rigidă fără un grad ridicat de abstractizare. Orice schimbare a soluţiei implică o reprogramare. Flexibilitatea este scăzută. Un alt dezavantaj îl constituie cuplarea strânsă a serviciilor participante.

- **Service Component Architecture (SCA)**

  SCA este set de specificaţii independente de limbaj şi tehnologie care doreşte să simplifice compunerea serviciilor ascunzând multe dintre elementele de infrastructură. SCA specifică modul de creare, combinare şi expunere a componentelor.

- **Compunere Bazată pe Evenimente**

  Această metodă de compunere foloseşte modelul publish/subscribe introducând un nivel de decuplare între consumatorii şi producătorii de servicii. Această abordare aduce un plus de flexibilitate.

- **Compunere bazată pe Engine-uri de Orchestrare**

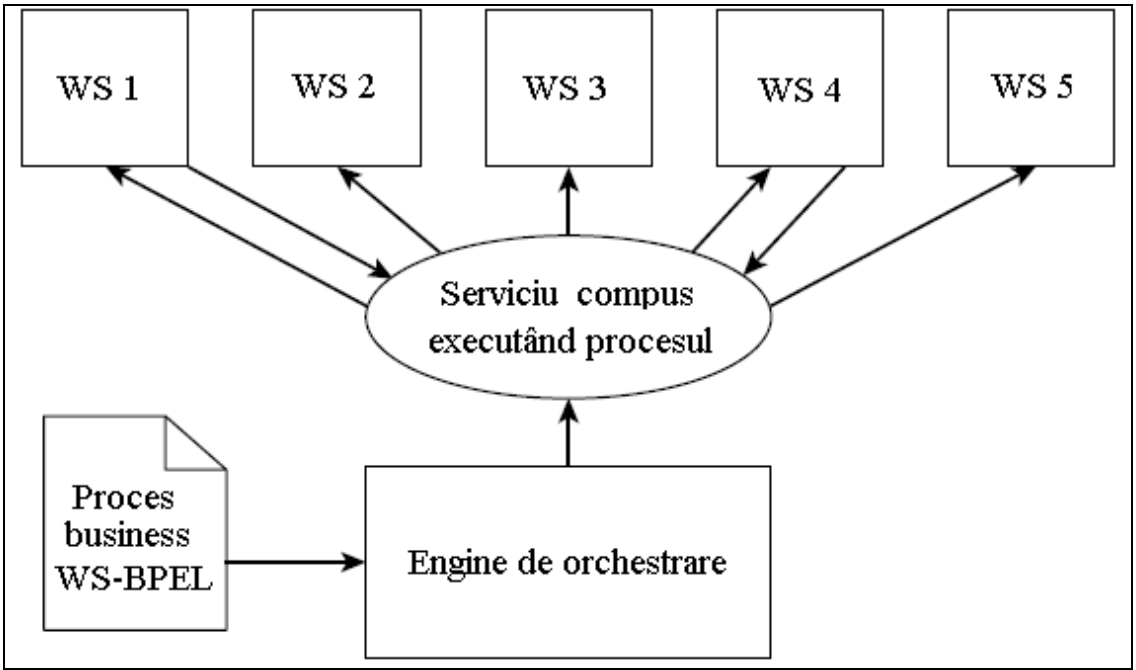


Figura 2.15. Orchestrare a unui proces executabil

O abordare comună este aceea a folosirii unui engine de orchestrare pentru a controla fluxul de execuţie al proceslor exprimate într-un limbaj de orchestrare cum ar fi WS-BPEL. Astfel se pot executa „scripturi de orchestrare", combinând servicii existente. Beneficiul acestei abordări este acela că procesul executabil specifică detaliile şi regulile de compunere, abstractizând detaliile de serviciile implicate.
În continare vom prezenta o schemă a unei astfel de compuneri. Ea prezintă un document WS-BPEL care descrie un proces business rezultat din orchestrarea unor servicii. Acest proces business rulează în cadrul unui engine de orchestrare. Engine-ul crează un serviciu compus care apelează serviciile descrise în documentul BPEL.

2.3.3.5. Folosirea serviciilor pentru construirea aplicaţiilor enterprise

Înţelegerea mecanismelor SOA nu este o sarcină dificilă. Cu toate acestea, implementarea unei aplicaţii cu un design corespunzător poate fi o provocare dificilă. SOA şi aplicaţiile diferă. Fiecare aplicaţie doreşte să rezolve o anumită problemă implementând un subset al funcţionalităţii enterprise. SOA poate oferi o modalitate mai bună de a scrie aplicaţii. Idealul este însă conceperea de servicii care să poată fi folosite în mod dinamic.

Aplicaţiile SOA tipice depind de o mulţime de servicii. Invocarea acestor servicii necesită cunoştinţe despre locaţia lor. În cazul cel mai simplu, este suficientă introducerea locaţiilor direct în cod. Această abordare presupune totuşi o cuplare strânsă între implementarea soluţiei şi localizarea serviciului. Externalizarea locaţiei serviciilor într-un fişier extern prezintă o îmbunătăţire evidentă. Această abordare este mai flexibilă pentru că nu

necesită introducerea locaţiei serviciului în codul aplicaţiei. Folosirea unui intermediar care rezolvă în mod dinamic interogările serviciilor în adrese şi politici de invocare oferă soluţia cea mai flexibilă. Acesta este registrul de servicii. Acest registru de serviciu este definit în standarde ca fiind UDDI (Universal Description, Descovery and Integration). Cu toate că acest registru a fost proiectat iniţial pentru a oferi suport de broker în vederea recomandării unui serviciu în funcţie de operaţiile dorite, el este folosit în prezent doar pentru a salva fişierele WSDL. O folosire practică a registrului constă în determinarea adresei sale pornind de la numele său şi de la politicile sale de acces.

Organizaţiile folosesc infrastructuri de securitate. Acestea protejează resursele de acces nedorit. La baza soluţiei de securitate stă introducerea unui nivel de securitate. Responsabilităţile acestui nivel sunt următoarele: acomodarea eterogenităţii multiplelor platforme, propagarea şi gestionarea informaţiilor de securitate, suport pentru credenţiale multiple de securitate şi suport pentru protocoale de transport multiple. Există două opţiuni arhitecturale în vederea implementării securităţii la nivelul aplicaţiei: security gateway şi interceptor:

- **Security Gateway**

  Este un pachet software sau dispozitiv hardware care filtrează traficul şi blochează traficul neautorizat înainte de a ajunge la un serviciu protejat. În practică, el este folosit ca un mecanism de delimitare al unui permitetru pentru siguranţa aplicaţiei.

- **Interceptor**

  Aceasta este strategia cel mai des folosită pentru a oferi un mecanism de securitate. Implementarea securităţii are loc direct în implementarea serviciului prin folosirea de metode specifice platformei: filtre ISAPI (Internet Server Application Programming Interface), JAX-RPC, JAX-WS, MQ exits şi altele. În figura de mai jos se poate observa arhitectura unui astfel de interceptor. Serviciul de securitate oferă date necesare pentru stabilirea securităţii. În realitate, el poate lipsi. Întreaga logică de securitate poate fi conţinută în interceptor.

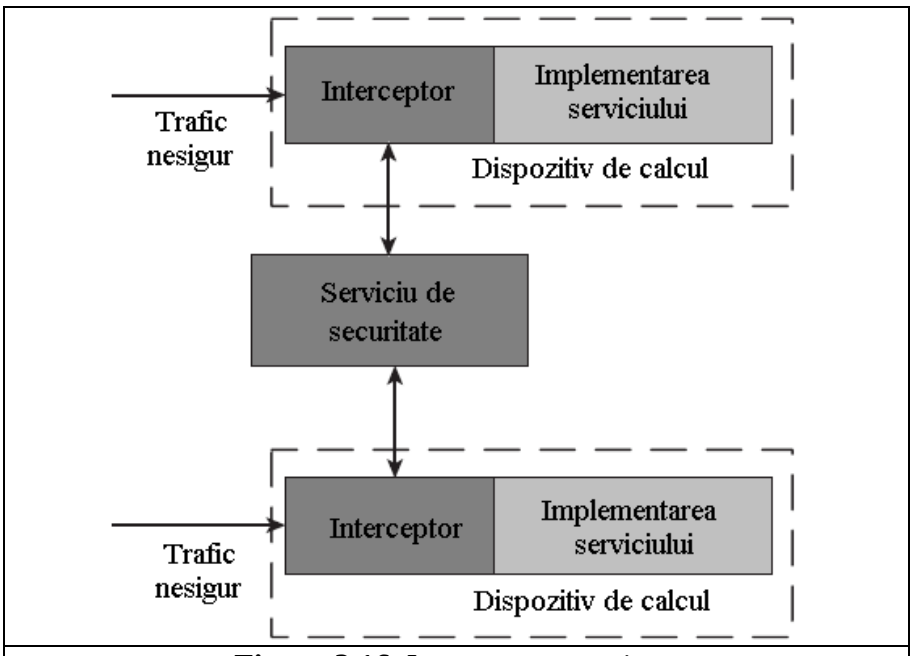


Figura 2.16. Interceptor securitate

O modalitate comună pentru a oferi funcţionalitatea pentru compunere, comunicare, acces al al datelor şi servicii web într-o astfel de arhitectură îl reprezintă un Enterprise Service Bus (ESB). În Figura 2.17 e prezentă arhitectura de bază a unui ESB.

Arhitectura de bază a unui ESB conţine următoarele:

- Motor ESB distribuit responsabil pentru comunicarea între producători şi consumatori
- Servicii distribuite de infrastructură incluzând
  - Registru pentru localizarea şi rutarea serviciilor
  - Suport pentru tranzacţii
  - Mediere (inserţia proceselor intermediare în mesaje)
  - Engine-uri specializate (de ex., pentru orchestrare)
  - Monitorizare
  - Suport pentru securitatea serviciilor
- Configurare run-time consând în metadata care defineşte configurarea şi distribuirea componentelor ESB.
- Administrare şi control centralizate

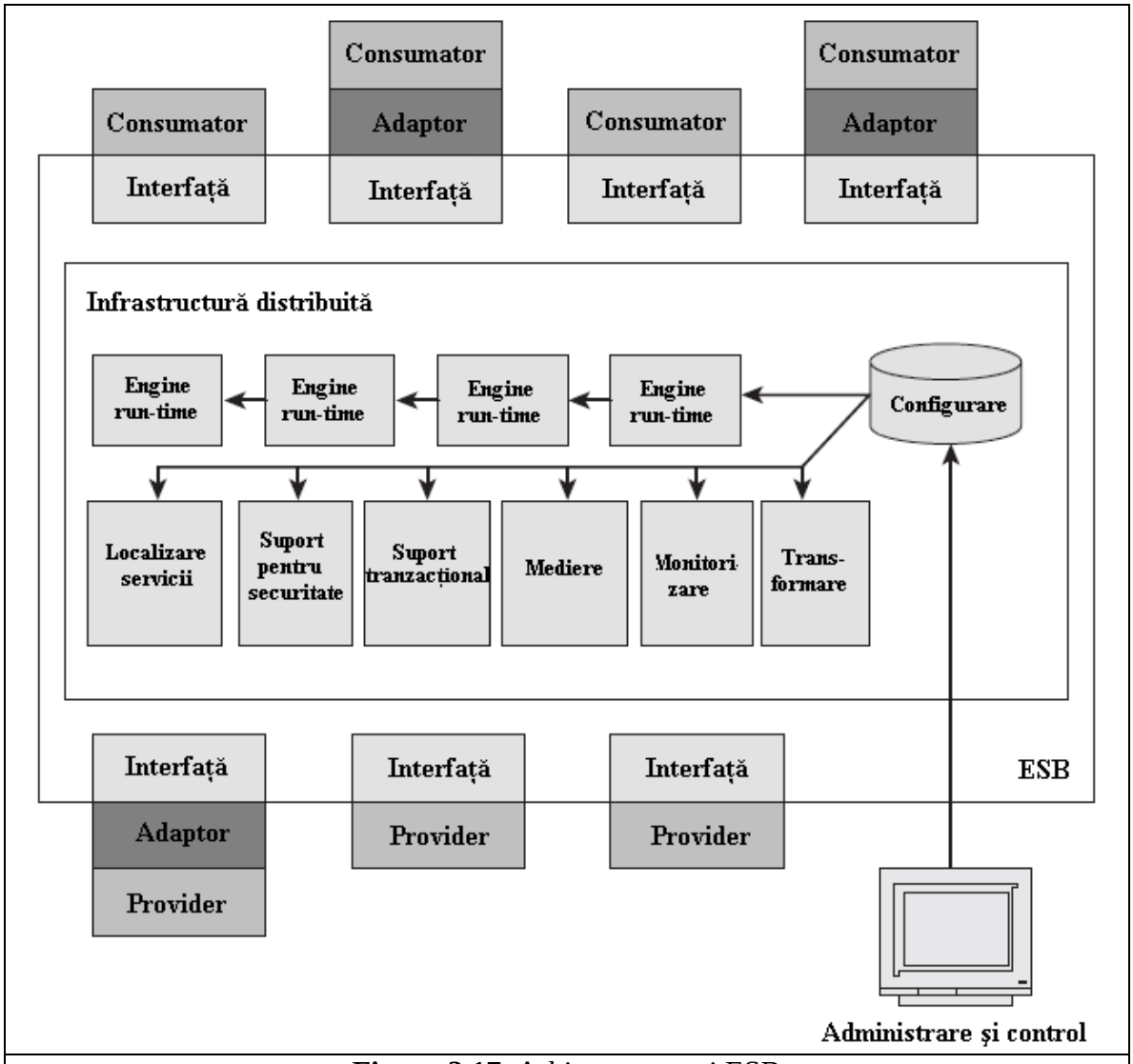


Figura 2.17. Arhitectura unui ESB

2.3.3.6. Folosirea integrării în cadrul SOA

În ciuda eforturilor depuse pentru a standardiza interacţiunea dintre sisteme, majoritatea oepraţiilor de astăzi sunt efectuate de sisteme moştenite (eng. „legacy"). Singura modalitate eficientă de a implementa soluţii SOA constă în folosirea aplicaţiilor deja existente. Realizarea acestui deziderat nu este deloc simplă şi constă în mai mult decât în expunerea vechilor aplicaţii cu ajutorul serviciilor web.

În continuare vom enumera câteva modalităţi prin care se poate obţine integrarea:

- **Folosirea mesageriei**

  Se poate folosi eficient atunci când aplicaţiile enterprise sunt contectate la o infrastructură de mesagerie. Componenta business trimite cererea către o coadă de cereri la care e conectată apliaţia existentă. Odată ce aplicaţia a exeuctat cererea primită, o va trimite cozii de răspuns la care e conectată componenta business. Această modalitate de integrare este prezentată în figura următoare.

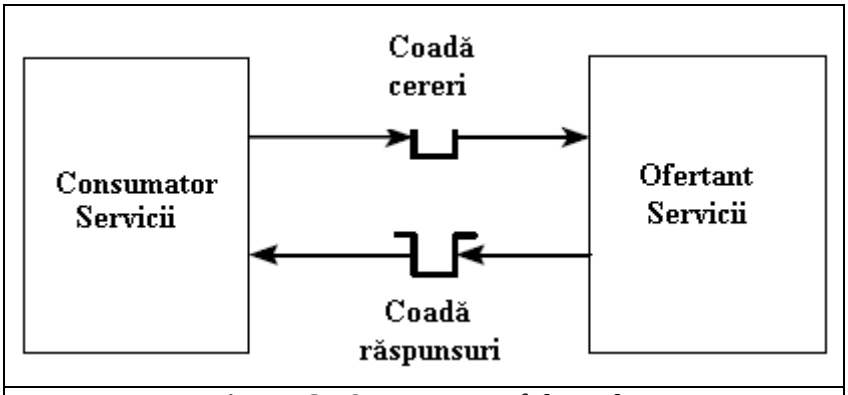


Figura 2.18. Integrarea folosind mesageria

- **Folosirea unui Agent de Mesaje** (eng. „Message Broker")

  Agentul de mesaje introduce un layer suplimentar deasupra sistemelor de mesaje tradiţionale. El este câteodată numit un hub de mesaje. Toate mesajele dintre producători şi consumatori sunt direcţionate indirect prin acest hub. Aceasta permite reformatarea şi rutarea informaţiei de la o sursă la alta în mod programatic. Această abordare reduce impactul asupra sursei şi destinaţiei prin nivelul scăzut de cuplare, după cum rezultă din Figura 2.19.

- **Folosirea serviciilor web**

  Suport pentru servicii web este răspândit în platformele de dezvoltare curente. Integrarea folosind această metodă nu este complicată. În următoarea figură se observă modalitatea prin care consumatorul pur şi simplu invocă un serviciu web oferit de aplicaţia enterprise. Acest procedeu e ilustrat în Figura 2.20.

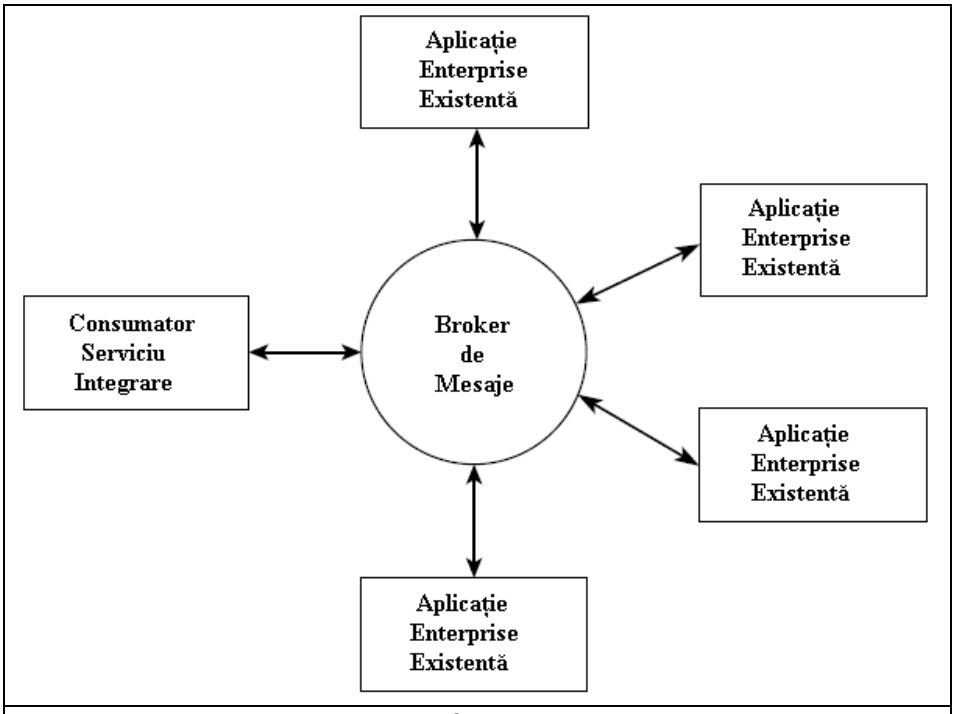


Figura 2.19. Integrarea folosind broker-ul de mesaje

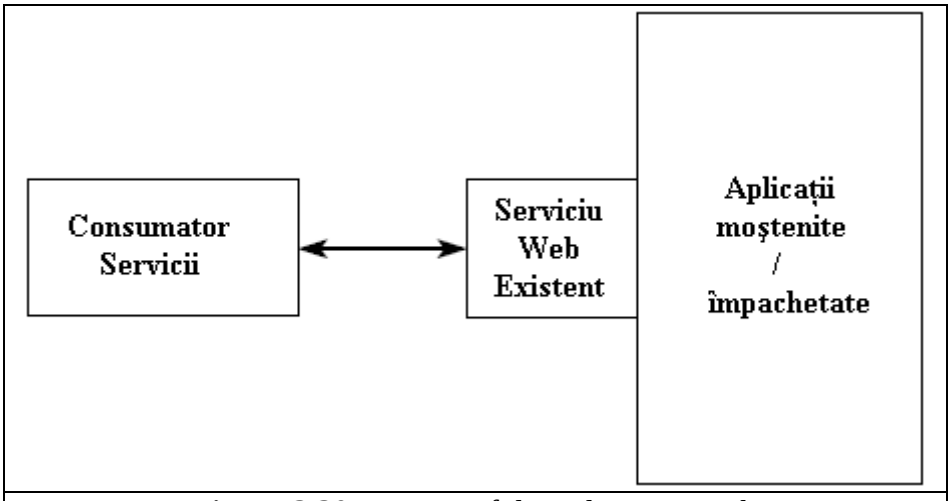


Figura 2.20. Integrare folosind servicii web

2.4. Concluzii

În acest capitol am subliniat rolul important pe care îl joacă serviciile web în descrierea aplicaţiilor curente. Am subliniat diferitele tipuri de servicii web şi modalitatea în care acestea pot expune algoritmi (SOAP) sau date (REST). Mai mult decât atât, am oferit exemple despre modalitatea în care două dintre companiile de top în domeniul IT îşi expun funcţionalitatea cu ajutorul serviciilor web.

În restul capitolului am încercat să trecem în revistă aspectele cruciale care stau la baza dezvoltării unei aplicaţii bazate pe servicii web. Am enumerat caracteristicile la care trebuie să adere un serviciu web pentru face parte dintr-o astfel de aplicaţie, accentul care trebuie pus pe definirea clară a regulilor de business, design-ul interfeţei precum şi implementarea serviciilor. În continuare am ilustrat compunerea serviciilor precum şi modul în care acestea se folosesc pentru a construi aplicaţii enterprise pornind de la soluţii IT existente.

3. TEHNOLOGII FOLOSITE

În acest capitol vom prezenta tehnologiile care au contribuit la implementarea aplicaţiei. Vom descrie acele detalii pe care le considerăm necesare în vederea înţelegrii codului sursă. Descrierea va fi cât se poate de succintă, fără a neglija însă vre-un aspect important. Vom face referiri la literatura de specialitate, care detaliază tehnologiile enumerate de noi.

- **Java EE**

  Java Platform, Enterprise Edition sau Java EE este o platformă folosită pentru programarea server-side. Ea diferă de patforma Java SE prin faptul că oferă biblioteci care contribuie la construirea aplicaţiilor modulare, distribuite care rulează într-un server de aplicaţii. Mai multe detalii despre această plaformă se pot găsi accesând pagina web a dezvoltatorului [16, 17].

- **EJB 3.0**

  Enterprise Java Beans este un standard inclus în API-ul Java EE care oferă peresistenţă folosind API-ul JPA, procesarea tranzacţiilor, controlul concurenţei, evenimente folosind JMS, servicii de numire şi directoare prin JNDI, securitate, rularea aplicaţiilor într-un server de aplicaţii, apelul procedurilor remote şi expunerea metodelor business prin servicii web [13]. În continuare vom descrie succint acele componente ale acestui standard care au contribuit la dezvoltarea soluţiei software propuse de noi.

  - **Stateless Session Beans**

    Acestea sunt obiecte distribuite care nu au asociată o stare conversaţională, permiţând accesul concurent la opreaţii. Conţinutul variabilelor de instanţă nu este garantat între apeluri diferite de metodă. Toate instanţele unui astfel de obiect sunt identice. Obiectul implementează în mod obligatoriu o interfaţă care poate fi locală (adnotată cu `@Local`) sau remote (`@Remote`). Obiectul va fi adnotat cu `@Stateless`. Localizarea unui astfel de obiect se face folosind JNDI. Pentru exemplul de faţă ne vom mulţumi să afirmăm că o instanţă a obiectului poate fi accesată cu ajutorul adnotării `@EJB`. În continuare vom prezenta un exemplu de declarare şi invocare a unui astfel de obiect. Interfaţa se declară în modul următor:

    ```
    import javax.ejb.Remote;

    @Remote
    public interface HelloWorld {
        String getHello();
    }
    ```

    Obiectul care implementează interfaţa este următorul:

    ```
    import javax.ejb.Stateless;

    @Stateless
    public class HelloWorldBean implements HelloWorld {
    ```

```
    public String getHello() {
        return "Hello World !";
    }
}
```

În final, să observăm modul în care referinţa este inserată de către serverul de aplicaţie într-un servlet care o apelează:

```
import java.io.*;
import javax.ejb.EJB;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {
    @EJB
    private HelloWorld helloWorld;

    public void service (HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException, IOException {
      resp.getWriter().println(helloWorld.getHello());
    }
}
```

Pentru mai multe detalii referitor la această tehnologie recomandăm lucrarea [8].

- **Entity Beans**

Un Entity Bean este un tip de Enterprise Bean care reprezintă data persistentă localizată într-o bază de date. O entitate poate gestiona persistenţa sau poate delega această responsabilitate containerului în care rulează. Este identificată în mod unic de o cheie primară. Ea îşi păstrează starea între două instanţe diferite ale containerului. În EJB 3.0 Entity Beans au fost incluse în JPA. La fel ca şi o tabelă într-o bază de date, o entitate poate relaţiona cu altele. Relaţiile posibile sunt 1-N, N-1, M-N. Fiecare dintre aceste tipuri de relaţiile pot fi uni- sau bidirecţionale. În continuare vom ilustra un exemplu de declarare al unui astfel de obiect:

```
@Entity
public class Book implements Serializable {
   @Id
   @GeneratedValue
   Long id;
   public Long getId() { return id; }

   public void setId(Long id) { this.id = id; }
}
```

Mai sus am declarat un obiect adnotat cu Entity care reprezintă un obiect în baza de date. Adnotarea `@Id` indică faptul că `id` este câmpul care reprezintă identificatorul unic, iar `@GeneratedValue` indică faptul că identificatorul este generat automat de către container. Pentru mai multe detalii referitor la această tehnologie recomandăm lucrarea [8].

- **JAX-WS**

Acesta este API-ul Java EE folosit pentru a crea şi utiliza servicii web care folosesc standardul SOAP. Implementarea de referinţă a acestui standard face parte din proiectul GlassFish, un server de aplicaţii open souce Java EE. Această implementare de referinţă face acum parte din proiectul Metro. Declararea unui serviciu web se face cu ajutorul adnotării `@WebService`, iar metodele care sunt expuse sunt adnotate cu `@WebMethod`. Pentru mai multe detalii referitoare la declararea şi apelul serviciilor web, recomandăm lucrările [3, 5, 9].

```
import javax.jws.WebService;

@WebService
public class MyService {

   @WebMethod
   public int myMethod(int x){
      return x;
   }
}
```

- **JAX-RS**

Acesta este specificaţia API-ului care suportă dezvoltarea şi rularea serviciilor RESTful în cadrul plaftormei Java. Jersey este implementarea standard a acestei specificaţii. Serviciile web de tip REST pot fi utilizate cu mare uşurinţă folosind acest API. În continuare vom prezenta codul sursă folosit pentru a descrie un astfel de serviciu.

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.FormParam;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.DELETE;
import java.beans.XMLEncoder;
import java.io.ByteArrayOutputStream;

@Path("/RESTService")
public class MyService {

    private String msg;

    @GET
    @Produces("text/plain")
    @Path("{extra}")
    public String personalized_read(@PathParam("extra") String
         param) {
        return "Hello" + ": " + param + "\n";
    }

    @POST
    @Produces("text/xml")
    public String create(@FormParam("msg") String new_msg ) {
        this.msg = msg;
        ByteArrayOutputStream stream = new
```

```
                                ByteArrayOutputStream();
        XMLEncoder enc = new XMLEncoder(stream);
        enc.writeObject(new_msg);
        enc.close();
        return new String(stream.toByteArray()) + "\n";
    }

    @DELETE
    @Produces("text/plain")
    public String delete() {
        this.msg = null;
        return "Message deleted.\n";
    }
}
```

Calea de acces la acest serviciu este dată de valorea adnotării `@Path`. Adnotările `@GET`, `@POST`, `@DELETE` marchează metodele de clasă care vor fi accesate în cazul invocării metodelor HTTP corespunzătoare. Adnotarea `@Produces` specifică tipul datei returnată de metoda invocată. Formatele valide sunt cele MIME. Adnotarea `@FormParam` specifică faptul că parametrul de intrare al metodei create este un parametru citit într-un form HTML. O atenţie deosebită vom acorda adotării `@Path` din dreptul metodei personalized_read. Aceasta indică faptul că ea va fi apelată atunci când se va executa un GET asupra căii de bază indicată de adnotarea `@Path` a clasei (care este "/RESTService"), urmată de valoarea indicată de adnotarea metodei. Adnotarea `@PathParam` indică faptul că această cale va fi folosită de metodă. Trebuie reţinută posibilitatea de a indica un umăr infinit URL-uri cu ajutorul acestor adnotări (de exemplu, /RESTService/Path1, /RESTService/Path2, etc.).

Înainte de a descrie următoarea tehnologie vom ilustra modul facil în care se pot apela clienţi RESTful cu ajutorul Jersey.

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
……
Client client = Client.create();
WebResource webResource =
           client.resource("http://example.com/base");
String s = webResource.get(String.class);
```

În codul de mai sus se poate observa declararea unui client al unui astfel de serviciu împreună cu resursa care se doreşte accesată. În exemplul de faţă accesul este realizat printr-o metodă simplă GET, rezultatul aşteptat fiind un şir de caractere. Mai multe detalii referitoare la acest API precum şi la implementarea lui se găsesc la adresele web [12, 18, 19].

- **Servlets/JSP**

Acestea sunt obiecte care procesează cereri (de obiecei HTTP) şi returnează un rezultat. Acest rezultat este de regulă afişat într-un browser web. Astfel este permisă adăugarea de conţinut dinamic paginilor web. Pentru mai multe detalii despre modalitatea în care se pot dezvolta interfeţe web folosind aceste tehnologii se poate consulta [1].

- **GlassFish Application Server**

  Acesta este un proiect open source aflat sub tutela Sun Microsystems, pentru platforma Java EE. Versiunea comercială este cunoscută sub numele Sun GlassFish Enterprise Server. GlassFish este bazat pe codul oferit de către Sun Microsystems şi sistemul de persistenţă Oracle TopLink. Serverul expune un API pentru a dezvolta aplicaţii enterprise folosind tehnologiile de mai sus. Modulele Web pot fi servlet-uri sau Java Server Pages, logica de business find construită folosind EJB. Pentru mai multe detalii recomandăm consultarea paginii care găzduieşte proiectul [15], precum şi lucrarea [4].

- **Derby (Java DB)**

  Java Derby este un sistem de gestiune al bazelor de date relaţionale care poate fi inclus în programe Java şi folosit pentru procesare atranzacţiilor online. Mai este cunoscut ca şi Sun Java DB.

4. PROIECTAREA APLICAŢIEI

În cadrul acestui capitol vom analiza deciziile arhitecturale care au stat la baza implementării soluţiei. Vom face referire la principiile enunţate în capitolul 2, „Fundamentare teoretică”, argumentând deciziile noastre. Pentru început vom descrie arhitectura conceptuală a sistemului, care e alcătuită din diferite componente care interacţionează. După aceasta vom trece la descrierea componentelor individuale ilustrând modul în care acestea interacţionează.

4.1. Arhitectura conceptuală

În continuare vom prezenta arhitectura conceptuală a sistemului. Vom trece în revistă componentele constituente. După cum am menţionat în prima parte a lucrării, aceasta este varianta finală a arhitecturii de referinţă ce am folosit-o pe parcursul elaborării framework-ului. Primul pas în realizarea arhitecturii de referinţă l-a constat analiza cerinţelor aplicaţiei. Am plecat de la obiectivele enunţate în introducerea lucrării şi am alcătuit o listă a proceselor de business ce trebuie implementate.

Se pot distinge cu uşurinţă cele trei nivele ale arhiecturii noatre. Primul este cel al interfeţei, al doilea este cel business şi al treilea este nivelul serviciilor. Ultimul nivel nu face parte din aplicaţie, reprezentând doar serviciile externe accesate de aplicaţie.

- **Nivelul interfeţei**

  Acest nivel prezintă două componente. Prima componentă, „Web Application” este un punct de acces către framework şi face parte din framework. Ea prezintă două puncte de acces: un serviciu web SOAP cu funcţii utilizator şi un serviciu web SOAP cu funcţii pentru administrator. Motivul pentru care această componentă expune două servicii web este acela că nu dorim că oricine să beneficieze de acces necondiţionat la business-ul implementat. Nu dorim de exemplu, ca accesul la registrul cu servicii web să fie disponibil oricui. Cea de-a doua componentă prezentă la nivelul interfeţei este o aplicaţie web folosită pentru testarea aplicaţiei. Acesta este un punct de acces la framework cu scop demonstrativ. Mai multe detalii despre folosirea acestei interfeţe vor fi disponibile în capitolul care detaliază rularea aplicaţiei. Acest punct de acces permite executarea unei căutări într-o singură sursă de date sau în mai multe surse de date.

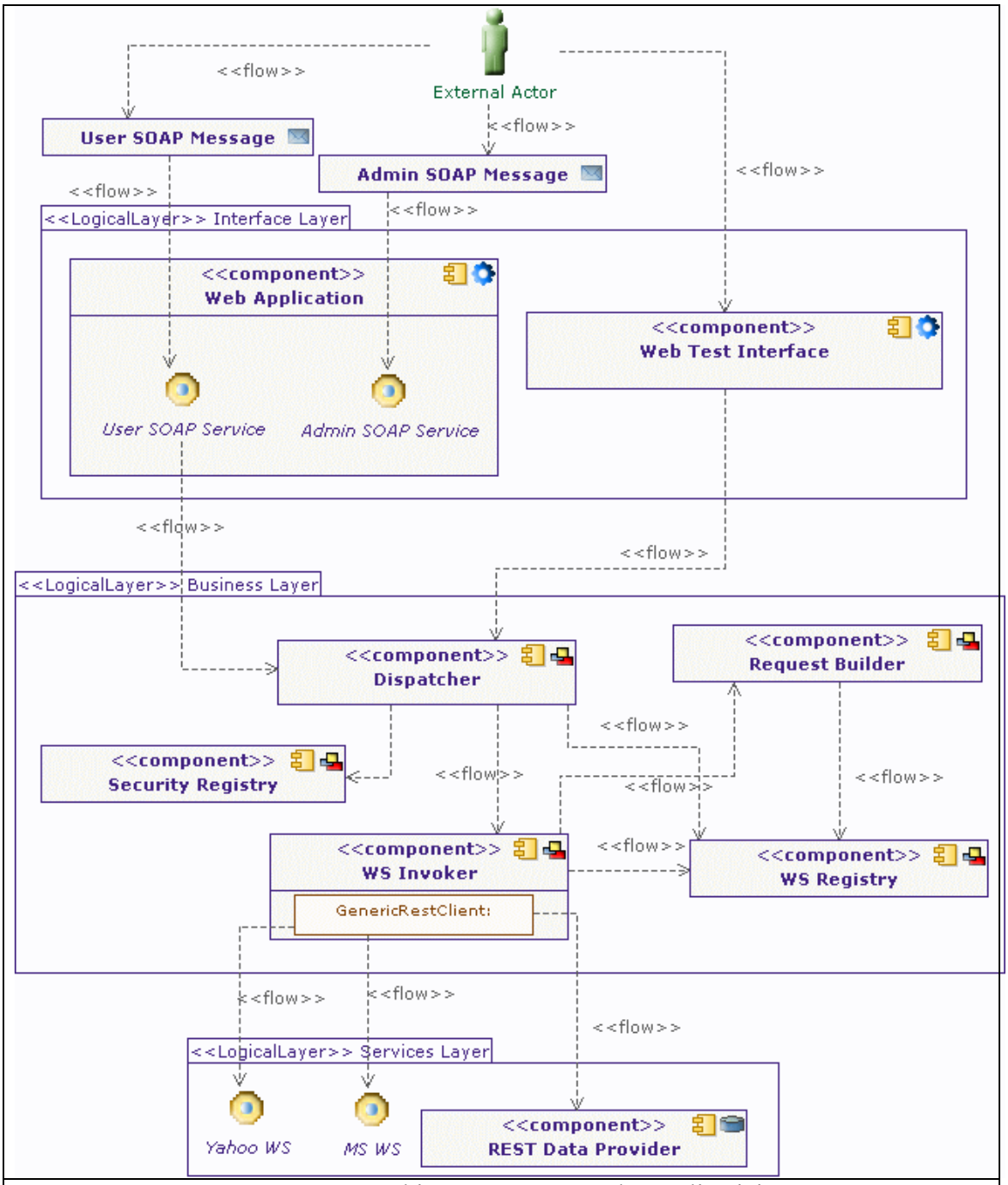


Figura 4.1. Arhitectura conceptuală a aplicaţiei

- **Nivelul proceselor business**

  Acest nivel conţine logica aplicaţiei şi e compus din mai multe module. Componenta dispatcher este componenta care primeşte cererile care se doresc a fi executate. Le analizează şi bazându-se pe politicile de securitate trimite mai departe cererea la componenta care o poate executa. Tipuri posibile de cereri sunt: căutare într-o singură sursă, căutare în mai multe surse, returnează lista serviciilor disponibile, etc. Componenta Request Builder este o componentă a cărei implementare nu a fost realizată în întregime. Vom detalia în cadrul secţiunii „Dezvoltări ulterioare”. Motivul principal pentru care această componentă nu a fost

implementată în întregime îl reprezintă complexitatea sarcinii propuse. Componenta Security Registry reprezintă un registru care conţine date despre utilizatori şi rolurile lor. Această componentă oferă metode prin care se poate verifica dacă un anumit utilizator are dreptul de a accesa o anumită metodă. Componenta WS Registry (Web Service Registry) oferă accesul la un registru de servicii web. Acest registru conţine informaţii despre URL-ul serviciilor, precum şi despre parametrii pe care aceştia îi foloesc. Componenta WS Invoker (Web Service Invoker) este componenta care tratează cererile de invocare a serviciilor web de tip REST. Analizând id-ul serviciului oferit şi parametrii supliniţi, ea are rolul de a forma cererea într-un format inteligibil pentru serviciul web şi de a o trimite pentru a fi procesată.

- **Nivelul serviciilor**

  Nivelul serviciilor nu face parte din framework-ul propus de noi şi constă în serviciile oferite de diferiţi furnizori. Pentru a demonstra conceptul aplicaţiei, am inclus referiri la servicii oferite de companiile Yahoo şi Microsoft. De notat faptul că în acest nivel pot exista servicii REST care expun baze de date.

4.2. Descrierea componentelor

În secţiunea prezentă vom analiza fiecare componentă în parte. Obiectivul nostru e acela de a detalia arhitectura internă a lor, modul de interacţiune cu celelalte componente şi principiile de proiectare care stau la baza lor. Înainte de a trece la descrierea componentelor vom aminti principiile generale de design care au stat la baza proiectării lor. Deciziile de design enunţate aici sunt valabile pentru fiecare componentă în parte.

- **Modularitate, granularitate şi nivel scăzut de cuplare**

  Fiecare componentă este responsabilă cu execuţia unei anumite sarcini sau a unui număr multiplu de sarcini grupate după aceeaşi funcţionalitate. Sarcinile mai complexe sunt efectuate prin compunerea de operaţii mai simple, care nu necesită resurse de calcul atât de ridicate. O componentă nu necesită cunoştinţe avansate despre celelalte sau despre mediul în care rulează. Singurul lucru necesar este adresa componentei cu care relaţionează şi operaţiile pe care le expune. Astfel s-a realizat şi dezideratul cuplării joase.

- **Încapsulare**

  Fiecare modul este proiectat ţinând cont de diferenţa dintre interfaţă (*ce* funţionalitate oferă) şi implementare (*cum* anume se împlineşte funcţia repsectivă). Această abordare decuplează operaţia de proiectare de cea de implementare prin faptul că odată ce s-a definit interfaţa (sau contractul), implementarea se poate schimba după nevoi. Mai mult decât atât, odată ce aplicaţia rulează, se poate recurge la o nouă implementare care poate fi mai eficientă. În cadrul secţiunii „implementare” vom analiza modul în care s-a realizat efectiv această separare folosind tehnologiile prezentate în lucrare.

- **Izolarea responsabilităţilor**

  Fiecare componentă execută o operaţie sau un set de operaţii care aparţin aceluiaşi câmp semantic. Operaţii diferite sunt responsabilitatea componentelor diferite. Din acest motiv a rezultat un număr mai ridicat de componente. Fiecare dintre ele are o anumită responsabilitate şi cererea utilizatorului va fi împlinită ţinând cont de restul.

- **Descoperirea dinamică şi legarea**

  Componentele rulează în cadrul unui container oferit de serverul de aplicaţie. Acest container oferă servicii pentru a descoperi în mod dinamic celelalte componente care sunt referite.

Comunicarea între diferitele componente are la bază tehnica schimbului de parametri. Semnăturile operaţiei sunt disponibile prin interfeţe. Nu am folosit schimbul de documente din necesitatea de a menţine simplitatea aplicaţiei. Încă un strat de abstractizare în plus la nivelul de comunicare (care constă în codificarea documentelor şi în decodificarea lor) nu ar fi fost justificat. Sincronizarea mesajelor este de tipul Request/Reply. Orice operaţie executată de o componentă asupra altei componente aşteaptă un răspuns. În plus, interfeţele sunt stateless. Acesta e un deziderat important al aplicaţiilor SOA.

În ceea ce priveşte compunerea serviciilor, am optat pentru varianta programatică. Folosirea unui engine de orchestrare împreună cu un bus de servicii ar fi fost opţiunea ideală din cauza flexibilităţii pe care o oferă. Această abordare va fi tratată în cadrul secţiunii „dezvoltări ulterioare”. Vom propune folosirea bus-ului de servicii OpenESB, o implementare care e disponibilă cu serverul de aplicaţii GlassFish.

În continuare descriem componentele care alcătuiesc framework-ul.

- **Web Application**

Această componentă, după cum am amintit mai sus, are rolul de a oferi accesul prin intermediul unei interfeţe web către framework. Ea pune la dispoziţie două servcii web SOAP, unul pentru lucrul cu operaţii accesibile administratorului şi altul pentru operaţii accesibile unui utilizator fără drepturi speciale. Serviciile sunt de tipul stateless, urmând recomandările pentru realizarea unei astfel de arhitecturi.

  - **Admin SOAP Endpoint**

    Expune un serviciu web, AdminEndpoint. Acesta este un serviciu SOAP. Operaţiile puse la dispoziţie de acest serviciu web sunt următoarele: adăugare/ştergere serviciu, returnare serviciu, adăugare/ştergere parametru, returnare listă parametri, adăugare/ştergere provider serviciu.

  - **User SOAP Endpoint**

    Acest punct de acces pune la dispoziţie un serviciu web SOAP – UserEndpoint care e disponibil accesului utilizatorilor. Operaţiile disponibile sunt: singleSearch (executarea unei căutări pe o singură sursă de date), multipleSearch (executarea unei căutări pe mai multe surse de date), returnare listă servicii, returnare serviciu, returnare listă parametri.

- **Web Test Interface**

  Aceasdtă componentă pune în evidenţă folosirea framework-ului cu ajutorul unei interfeţe web. Interfaţa web constă din două puncte de acces: unul pentru executarea unei căturări pe o singură sursă de date (SingleWebSearch.jsp) şi un altul pentru executarea unei căutări pe mai multe surse de date (MultipleWebSearch.jsp). Detalii despre folosirea acestei interfeţe web pentru testarea soluţiei se poate găsi în capitolul „Rulare şi rezultate experimentale”.

- **Dispatcher**

  Această componentă este responsabilă cu primirea cererii efectuată de aplicaţiile din interfaţa web, analizarea ei şi trimiterea ei mai departe la modulul care o poate executa. De exemplu, dacă cererea primită este de tipul căutare pe mai multe surse de date, atunci se va invoca o metodă a componentei WS Invoker. În cazul în care cererea doreşte aflarea unor informaţii din registrul web, atunci se va invoca metoda specifică registrului de servicii web. Tipurile de cereri care se pot forma sunt următoarele: Adăutarea/ştergerea parametrului unui serviciu, adăugarea/ştergerea unui serviciu, adăutarea/ştergerea unui provider de servicii, returnarea unei liste de parametri, returnarea unei liste cu serviciile disponibile, returnarea unui serviciu, căutare pe o singură sursă şi căutare pe mai multe surse. În cadrul secţiunii de anexe vom prezenta o diagramă de clase a tututor acestor tipuri de cereri. Componenta este stateless şi urmează design-pattern-ul broker prin funcţionalitatea oferită.

- **WS Registry**

  Această componentă expune o bază de date în care sunt incluse informaţii despre serviciile web. Fiecare serviciu are un provider, un tip de răspuns (xml, json, etc) şi o listă de parametri. Fiecare parametru este caracterizat de un nume şi o descriere. Serviciile mai sunt caracterizate de un URL, nume şi descriere.
  Metodele expuse de această componentă permit returnarea unui serviciu specificat printr-un identificator unic, o listă a serviciilor, adăugarea/ştergerea unui serviciu, adăutarea/ştergerea unui parametru, returnarea listei de parametri, adăutarea/ştergerea unui provider de servicii. Operaţiile oferite sunt stateless.

- **WS Invoker**

  Cererile de căutare într-o sursă de date ajung de la utilizator prin intermediul dispatcher-ului la componenta WS Invoker. Responsabilitatea lui este aceea de a analiza cererea de căutare care poate fi pentru una sau mai multe surse. Ea apelează de clasa SingleWsInvoker pentru a executa cererile şi a returna mesajele. Clasa WsInvoker apelează la rândul ei clasa GenericRestClient. Aceasta este specializată pentru apelul şi returnarea rezultatelor serviciilor de tip REST. Aici este formată cererea în funcţie de specificul fiecărui serviciu împreună cu parametri şi valorile lor.

- **Security Registry**

  În spatele acestei componente se află o bază de date care conţine tabele cu numele utilizatorilor şi rolurile pe care aceştia le au atribuite. Rolurile posibile sunt USER şi ADMINISTRATOR. Componentele descrise mai sus se folosesc de această componentă pentru a permite sau nu accesul la metodele lor.

- **Request Builder**

  Această componentă nu a fost finalizată şi a rămas la stadiul de propunere pentru o dezvoltare ulterioară. Rolul ei este acela de a forma un model de date comun pentru cereri şi răspunsuri între serviciile web. Aceasta ar fi fost o contribuţie la un model de documente comun pentru aplicaţie. Motivul pentru care nu a fost finalizată este gradul ridicat de complexitate al acestei sarcini, precum şi diferenţa ridicată dintre formatul diferitelor surse de date.

5. IMPLEMENTAREA APLICAŢIEI

În capitolul prezent vom descrie modalitatea prin care s-a efectuat implementarea aplicaţiei descrise până în acest punct. Vom descrie fiecare modul în parte evidenţiind operaţiile pe care le oferă, modalitatea de comunicare cu celelalte module prin exemple de program. Fiecărui modul îi corespunde un pachet separat în codul sursă.

5.1. Transmiterea cererilor şi a răspunsurilor

Înainte de a detalia implementarea modulelor, vom descrie tipul de informaţii care se transmite în cadrul aplicaţiei. Am optat pentru codificarea informaţiilor care sunt transmise sub formă de clase. Componentele schimbă între ele instanţe ale claselor care codifică parametri doriţi.

Pentru cereri, am declarat clasa abstractă GenericRequest. Orice cerere care va fi transmisă va implementa această clasă.

```
public abstract class GenericRequest implements Serializable {
    public abstract RequestType getRequestType();
    public abstract Map<String, String> getParams();
    public abstract void setParams(Map<String, String> params);
    public abstract void putParam(String paramName, String
                                  paramValue);
}
```

Metoda getRequestType furnizează informaţii despre tipul cererii. Această metodă este utilă în mod deosebit componentei Dispatcher care analizează conţinutul cererii şi ia o decizie pe baza lui. Metodele getParams şi setParams sunt utile pentru lucrul cu parametrii cererii. Metoda putParam oferă posibilitatea de a seta un singur parametru în cadrul cererii.

RequestType este o enumerare care specifică tipurile posibile de cerere. Declararea acestei enumerări o observăm în secvenţa următoare de cod. Se observă cum fiecare tip de comunicare are asociat o valoare separată.

```
public enum RequestType implements Serializable {
   USER_LOGIN, USER_LOGOUT, USER_SINGLE_SEARCH,
   USER_MULTIPLE_SEARCH, USER_GENERIC_SEARCH,
   GET_SERVICES_LIST, GET_SERVICE, GET_PARAMETER_LIST,
   ADMIN_ADD_SERVICE, ADMIN_ADD_PARAMETER,
   ADMIN_REMOVE_SERVICE, ADMIN_MODIFY_SERVICE,
   ADMIN_REMOVE_SERVICE_PROVIDER, VOID_REQUEST;
   ADMIN_ADD_SERVICE_PROVIDER, ADMIN_REMOVE_PARAMETER,
}
```

Răspunsurile care rezultă în umra cererilor sunt transmise ca instanţe a clasei Result. O instanţă a acestei clase va reţine un mesaj şi eventual data asociată în cazul în care cererea necesită un răspuns mai complex, care să conţină una sau mai multe instanţe ale unor clase.

```
public class Result implements Serializable {

    private ResultType resultType;
    private String message;
    private Object data;

    public ResultType getResultType();
    public void setResultType(ResultType resultType);
```

```
    public String getMessage();
    public void setMessage(String message);
    public Object getData();
    public void setData(Object data);
}
```

Se pot observa câmpurile resultType, message şi data împreună cu metodele asociate care reprezintă respectiv tipul resultatului, conţinutul text al mesajului şi informaţia transmisă de răspuns. Enumerarea ResultType este suplinită doar pentru a generaliza design-ul sistemului. În aplicaţia de faţă, nu am folosit acest câmp.

Pentru a observa o diagramă de clase conţinând toate tipurile de răspunsuri, se poate consulta codul sursă. Clasele sunt descrise în pachetul diploma.business.request.

5.2. Dispatcher

Acest modul reprezintă punctul prin care se primesc cererile efectuate de client prin intermediul endpoint-urilor SOAP, sau al interfeţei web. El este declarat în pachetul diploma.business.dispatcher. Cererile sunt analizate şi în funcţie de tipul lor, sunt trimise componentelor corespunzătoare care conţin logica necesară împlinirii cererii. Componenta este implementată cu ajutorul tehnologiei EJB 3.0. Interfaţa este declarată în modul următor:

```
@Remote
public interface DispatcherRemote {
    public Result dispatch(final GenericRequest request);
}
```

Singura metodă declarată este „dispatch”. Această metodă are ca parametru instanţă a unei cereri şi returnează rezultatul executării acestei cereri. Clasa care implementează această interfaţă defineşte codul pentru executarea cererii după cum urmează:

```
switch (request.getRequestType()) {
      ....................................................
      case USER_SINGLE_SEARCH: {
            result = wsInvokerBean.invoke(request);
            break;
      }
      ....................................................
      case ADMIN_ADD_SERVICE_PROVIDER: {
      //    Apel de metodă care adaugă un serviciu
      ....................................................
      }
      ....................................................
}
```

Se observă cum, în cadrul unei instrucţuini „switch” se ia o decizie bazată pe tipul cererii care a fost primită ca şi parametru. Odată tipul cererii stabilit, se invocă metoda corespunzătoare a componentei care poate îndeplini cererea.

5.3. WS Invoker

Componenta este declarată în pachetul diploma.business.wsinvoker. Interfaţa publicată este următoarea:

```
@Remote
public interface WsInvokerRemote {
    public Result invoke(final GenericRequest request);
}
```

Asemenea componentei Dispatcher, există doar o singură metodă declarată. Aceasta are rolul de a invoca serviciile web corespunzătoare, care codificate în cadrul instanţei „request”. Implemenarea acestei interfeţe defineşte codul pentru tratarea diferitelor tipuri de cereri de căutare:

```
public Result invoke(final GenericRequest request) {
.....................................................
      if (request instanceof SingleSearchRequest) {
           SingleSearchRequest single =
                              (SingleSearchRequest)request;
           return singleSearchRequestInvoke(single);
      }
      if (request instanceof MultipleSearchRequest) {
            MultipleSearchRequest multiple =
                              (MultipleSearchRequest) request;
            return multipleSearchRequestInvoke(multiple);
      }
      if (request instanceof GenericSearchRequest) {
            GenericSearchRequest generic =
                              (GenericSearchRequest) request;
            return genericSearchRequestInvoke(generic);
      }
.....................................................
}
```

Analizându-se tipul cererii de căutare se va returna rezultatul furnizat de o metodă specializată pentru tipul respectiv de cerere. În prezent, sunt implementate metode pentru tipurile de căutare pe o singură sursă de date (SingleSearchRequest) şi pe mai multe tipuri de date (MultipleSearchRequest). Există posibilitatea efectuării unei cereri generice de căutare pe mai multe surse (GenericSearchRequest), însă nu există încă o implementare. În cadrul secţiunii „Dezvoltări ulterioare” se prezintă această propunere.

În continuare vom reda codul sursă al metodei multipleSearchRequestInvoke, detaliind modul în care execută cererea:

```
private Result multipleSearchCommandInvoke(MultipleSearchRequest
      multipleSearchCommand) {

      Result result = new Result();
      result.setMessage("Default response from wsinvoker.invoke: " +
                              "no web service executed.");

      List<SingleSearchRequest> sscl =
                  multipleSearchCommand.getSingleSearchCommandList();
      List<Result> resultList = new LinkedList<Result>();

      //  We invoke the web services and build the result
      //  We declare n threads for the execution
      SingleWsInvoker[] singleWsInvokers = new
                                    SingleWsInvoker[sscl.size()];
      //  We invoke each web service
      for (int i = 0; i < sscl.size(); i++) {
            singleWsInvokers[i] = new
                        SingleWsInvoker(sscl.get(i).getParams());
```

```
                singleWsInvokers[i].start();
            }
            //  We wait for the results to compute (we join the threads)
            String finalResult = "", partialResult = "";
            for (int i = 0; i < singleWsInvokers.length; i++) {
                try {
                    singleWsInvokers[i].join();
                    partialResult = singleWsInvokers[i].getResult();
                    finalResult += partialResult;
                    resultList.add(i, new Result(partialResult));
                    resultList.get(i).setData(sscl.get(i).getParams().
                    get("service-id"));
                    resultList.get(i).setMessage(partialResult);
                } catch (InterruptedException ex) {
                    Logger.getLogger(WsInvokerBean.class.getName()).log(
                    Level.SEVERE, null, ex);
                }
            }

            result.setMessage(finalResult);
            result.setData(resultList);

            return result;
        }
```

În prima fază se declară un răspuns de tipul „Result”. Acest răspuns va conţine un mesaj standard. După care se crează lista „sscl” (*s*ingle *s*earch *c*ommand *l*ist) care va conţine elemente de tipul „SingleSearchRequest”. Trebuie să amintim aici că o instanţă a clasei „MultipleSearchRequest” conţine mai multe cereri simple din care este compusă. În pasul următor, creem instanţe ale clasei „SingleWsInvoker” care e specializată în apelul serviciilor web. Această clasă extinde clasa Thread. Vom lansa în execuţie concurent mai multe fire pentru a apela serviciile web (singleWsInvokers[i].start()). După această secţiune, aşteptăm ca firele de execuţie care au fost lansate să returneze rezultatele parţiale. Următorul pas este acela al compunerii rezultatului final din rezultatele parţiale. Rezultatul returnat (variabila „result”) va conţine în câmpul „message” rezultatul sub formă de şir de caractere primit de la serviciile web apelate, iar câmpul „data” conţine o listă cu id-ul serviciului apelat şi rezultatul returnat.

Clasa SingleWsInvoker se foloseşte de o instanţă a clasei GenericRestClient. Aceata din urmă are sarcina efectivă de a apela serviciul REST, pornind de la parametrii primiţi. Vom afişa codul folosit pentru apelul unui serviciu web, datorită faptului că folosim API-ul JAX-RS cu implementarea Jersey. Acest API constituie o noutate şi va fi inclus în următoarea distibuţie a platformei Java Enterprise Edition.

```
public String invoke() {
        String finalUrl = null;
        try {
            if (params == null) {
                return "no params specified! (params == null)";
            }

            finalUrl = formUrl();

            String str;
            //  Calling the client
            Client client = Client.create();
            WebResource wr = client.resource(finalUrl);
            String response = wr.get(String.class);
```

```
            return response;
        } catch (..................) {
        ..............................................
        }
    }
```

Prima sarcină a clientului este aceea de a forma URL-ul corespunzător. De aceea s-a definit metoda formUrl(). Aceasta, bazându-se pe acei parametri primiţi în momentul apelării constructorului, va forma url-ul corespunzător, indiferent de sursa de date. După ce s-a format url-ul necear, se instanţiază clasa „Client" care e inclusă în JAX-RS. Se poate observa uşurinţa prin cre se crează o resursă, dat fiind url-ul precum şi obţinerea rezultatului invocării serviciului.

5.4. WS Registry

Această componentă oferă metode de acces la registrul în care sunt stocate informaţii despre servicii web (URL, parametrii, etc.). Pachetul în care se află declarată este diploma.wsregistry.

```
@Remote
public interface WsRegistryRemote extends Serializable {
    public String addServiceProvider(ServiceProviderFlat
                                          serviceProvider);
    public Service getService(int id) throws
            RegistryNotFoundException;
    public List<Service> getServicesList();
    public String addService(ServiceFlat service);
    public String addParameter(String paramName, String
                                          paramValue);
    public List<ParameterInfoFlat> getParameterList();
    public String removeParameter(String paramIdStr);
    public String removeService(String serviceIdStr);
    public String removeServiceProvider(String serviceIdStr);
}
```

Metodele declarate sunt:

- `addServiceProvider` – adaugă un nou furnizor de servicii
- `getService` – returnează un serviciu în funcţie de id
- `getServicesList` – returnează lista tuturor serviciilor web
- `addService` – adaugă un serviciu
- `addParameter` – adaugă un parametru
- `getParameterList` – returnează o listă a tuturor parametrilor disponibili
- `removeParameter` – şterge un parametru
- `removeService` – şterge un serviciu
- `removeServiceProvider` – şterge furnizorul de servicii

Clasele Service, ParameterInfo, ServiceProvider şi ServiceResponseType sunt entităţi care mapează clase din baza de date a registrului web. Clasele ServiceFlat, ParameterInfoFlat, ServiceProviderFlat sunt clase suplimentare construite plecând de la clasele Service, ParameterInfo respectiv ServiceProvider. Aceste clase „flat" nu conţin atâtea informaţii ca şi clasele din care se derivă pentru a nu încărca comunicarea. Schema bazei de date se poate găsi la Anexa A5.

5.5. Web Application

Componenta „Web Application” expune două servicii web: unul cu metode de acces pentru administrator şi altul cu metode de acces pentru utilizatorul obişnuit. Pachetul în care se află declarată este diploma.webinterface.soap.

```
@WebService
public class AdminEndpoint {
    @WebMethod
    public String addService(String name, String url, String
      description, int providerId, int responseTypeId,
      List<Integer> paramIdList);
    @WebMethod
    public String removeService(int serviceId);
    @WebMethod
    public List<ServiceFlat> getServices();
    @WebMethod
    public String addParameter(String name, String
                               description);
    @WebMethod
    public String removeParameter(int paramId);
    @WebMethod
    public List<ParameterInfoFlat> getParameters();
    @WebMethod
    public String addServiceProvider(String name, String
                                     description);
    @WebMethod
    public String removeServiceProvider(int serviceProviderId);
}
```

- `addService` – adaugă un nou servicu în registru
- `removeService` – şterge un serviciu
- `getServices` – returnează lista completă de servicii
- `addParameter` – adăugarea unui nou parametru
- `removeParameter` – ştergerea unui parametru
- `getParameters` – returnarea unei liste cu toţi parametrii
- `addServiceProvider` – adăugarea unui furnizor de servicii web
- `removeServiceProvider` – ştergerea unui furnizor de servicii web

```
@WebService
public class UserEndpoint {
    @WebMethod
    public String singleSearch(SingleSearchRequest request);
    @WebMethod
    public String multipleSearch(MultipleSearchRequest
                               request);
    @WebMethod
    public List<ServiceFlat> getServices();
    @WebMethod
    public ServiceFlat getService(int id);
}
```

- `singleSearch` – efectuarea unei căutări într-o singură sursă de date
- `multipleSearch` – efectuarea unei căutări în mai multe surse de date
- `getServices` – returnarea unei liste cu serviciile înregistrate în baza de date

- `getService` – returnarea unui serviciu

5.6. Web Test Interface

Pentru a testa aplicaţia am implementat o interfaţă web care permite efectuarea unei căutări într-o singură sursă de date şi în mai multe surse de date. Pagina web „SingleWebSearch.jsp” pune la dispoziţie un form din care se poate selecta un serviciu din lista de servicii web. După ce s-a selectat serviciul dorit, se vor afişa parametrii ceruţi pentru executare. Utilzatorul introduce valori pentru parametrii şi efectuează căutarea. Cererea va fi redirecţionată înspre „SingleWebSearchResult.jsp”, unde se va afişa o listă cu rezultatul dorit.

Pagina web „MultipleWebSearch.jsp” efectuează o căutare în mai multe surse de date. Mecanismul de funcţionare este similar cu cel al „SingleWebSearc.jsp”. Rezultatul va fi afişat în pagina „MultipleWebSearchResult.jsp”. Pentru a exemplifica căutarea în mai multe surse de date ne-am folosit de serviciile web oferite de Yahoo şi Microsoft care sunt descrise în capitolul „Fundamentare Teoretică”. Din acest motiv, am definit nişte clase care să ajute în reprezentarea conceptuală şi grafică.

- clasa DiplayUtil oferă funcţii pentru afişarea pe ecran a rezultatului cererilor de căutare pe una sau mai multe surse de căutare, afişarea parametrilor serviciului şi afişarea listei serviciilor. Este o clasă care joacă un rol principal în realizarea aplicaţiei demonstrative. Metodele definite de aceasta sunt:
  - executeAndDisplaySingleSearchResult, folosindu-se de datele pentru executarea unei singure cerei, o va executa si va afisa rezultatul
  - displayResults, primind ca parametri un şir de caracetere şi identificatorului unui serviciu, va scrie în fluxul de date lista de rezultate obţinute în urma unei parsări.
  - displayServiceParameters, afişează parametri unui serviciu, dat fiind identificatorul său
  - displayServiceList, va afişa o listă de servicii

- clasa abstractă GenericParser defineşte metoda getParser pentru a parsa rezultatul cererii în funcţie de serviciul care a fost apelat (Yahoo şi Microsoft codifică diferit rezultatul căutării). Clasele YahooParser şi MicrosoftParser sunt implementări ale acestei clase abstracte care parsează răspunsurile primite de la sursele respective de date.

Definiţia clasei este următoarea:

```
abstract class GenericParser {

    public static GenericParser getParser(final String xml, int
                                                    serviceId) {
        if (serviceId >= 1 && serviceId <= 4)
            return new YahooParser(xml, serviceId);
        else if (serviceId == 5)
            return new MicrosoftParser(xml, serviceId);
        else return null;
    }

    abstract public List<SearchResult> getResultList();
}
```

Se observă cum metoda getParser, va returna un parser specific id-ului primit. Nu vom detalia conţinutul claselor YahooParser sau MicrosoftParser, însă vom detalia modul în care se parsează şirul de caractere primit de la un serviciu formându-se listele cu

rezultate sub formă de clase. Impleentarea metodei getResultList a clasei YahooParser este următoarea:

```
public List<SearchResult> getResultList() {
  try {
      switch (serviceId) {
          case 1:
              return getWebSearchResultList();
          case 2:
              return getImageSearchResultList();
          case 3:
              return getNewsSearchResultList();
          case 4:
              return getMapsSearchResultList();
          default:
              return null;
      }
  } catch (XPathExpressionException ex) {
      System.out.println("IO exception");
      ex.printStackTrace();
  }
  return null;
}
```

În funcţie de id-ul specificat al serviciului, se va returna un tip specific de listă. Pentru a exemplifica procedeul de parsare, vom detalia implementarea metodei getWebSearchResultList():

```
private List<SearchResult> getWebSearchResultList() throws
                              XPathExpressionException {
      List<SearchResult> resultList = new LinkedList<SearchResult>();
      YahooWebSearchResult yahooWebSearchResult;
      XPathFactory factory = XPathFactory.newInstance();
      xPath = factory.newXPath();

      //  We replace the xmlns, or else we won't be able to prase,
      //    since the namespaces do not exist
      StringReader strReader = new
            StringReader(xml.replaceAll("xmlns=", "no-xmlns="));
      InputSource inputSource = new InputSource(strReader);

      //Get all search Result nodes
      NodeList nodes = (NodeList) xPath.evaluate(
                "/ysearchresponse/resultset_web/result", inputSource,
                  XPathConstants.NODESET);
      int nodeCount = nodes.getLength();
      System.out.println("Number of nodes: " + nodeCount);

      //iterate over search Result nodes
      for (int i = 0; i < nodeCount; i++) {
            //Get each xpath expression as a string
            String description = (String) xPath.evaluate("abstract",
                  nodes.item(i), XPathConstants.STRING);
            String date = (String) xPath.evaluate("date",
                  nodes.item(i), XPathConstants.STRING);
            String title = (String) xPath.evaluate("title",
                  nodes.item(i), XPathConstants.STRING);
            String url = (String) xPath.evaluate("url", nodes.item(i),
                  XPathConstants.STRING);
            yahooWebSearchResult = new
```

```
                    YahooWebSearchResult(description, date, title, url);
                resultList.add(yahooWebSearchResult);
            }
            return resultList;
    }
```

Mai întâi, declarăm o Listă care va conţine rezultatele parsate (resultList). Rezultatele reţinute de această listă vor fi de tipul YahooWebSearchResult care moşteneşte tipul SearchResult. Folosindu-ne de API-ul Xpath, evaluăm expresiile de la locaţia `"/ysearchresponse/resultset_web/result"` din `răspunsul` xml. Evaluarea XPath ne pune la dispoziţie o listă de noduri (NodeList). Într-o buclă, vom parcurge aceste noduri şi vom evalua atributele “abstract”, “date”, “title” şi “url”. Rezultatul evaluării îl vom stoca în variabile de tip String, după care formăm o nouă instanţă a clasei YahooWebSearchResult şi o adăugăm în lista cu rezultate. În final, returnăm rezultatul.

- clasa SearchResult reprezintă o abstracţie pentru un rezultat de căutare. Implementările lui GenericParser folosesc implementări ale SearchResult pentru a procesa răspunsurile cererilor de căutare. Implementările SearchResult folosite de noi sun YahooMapsSearchResult, YahooNewsSearchResult, YahooWebSearchResult, YahooImageSearchResult şi MicrosoftXmlSearchResult.

Pentru o diagramă a claselor folosite în acest modul, se poate consulta Anexa A9, A10, A11. API-ul XPath a fost utilizat în vederea implementării operaţiei de parsare.

5.7. Security Service

Serviciul de securitate oferă metode pentru a determina rolurile unui utilizator. El este declarat în pachetul diploma.security.

```
List<String> getUserRoles(final String user);
```

În funcţie de rolurile pe care le poate avea, componenta dispatcher îi va oferi accesul la componentele sale.

5.8. Request Builder

Acest modul un a fost implementat. El este folosit pentru a efectua cereri de căutare generice (GenricSearchRequest). Vom detalia o implementare la secţiunea „Dezvoltări ulterioare”. O implementare parţială a lui se găseşte în pachetul diploma.business.requestbuilser.

6. RULARE ŞI REZULTATE EXPERIMENTALE

În prima parte a acestui capitol vom descrie modaliatea de configurare şi rulare a aplicaţiei, iar în a doua vom ilustra modalitatea de testare a ei obţinând rezultate experimentale.

6.1. Configurarea mediului de lucru

Pentru a rula aplicaţia, sunt necesare mediul de dezvoltare NetBeans 6.5 împreună cu serverul de aplicaţii GlassFish v.2. Acestea se pot descărca împreună ca şi un produs unitar, de la adresa [25]. În plus, avem nevoie de un browser pentru a putea accesa interfaţa web. Recomandăm browser-ul Mozilla Firefox, care se poate descărca de la adresa [24].

Pentru a putea rula aplicaţia, primul pas necesar este deployment-ul. Pentru aceasta este necesar ca serverul de aplicaţii GlassFish să fie instalat corespunzător. Detalii despre acest procedeu se găsesc pe site-ul proiectului [15]. Se porneşte NetBeans şi se deschid proiectele Diploma-Final-App-Ejb, Diploma-Final-App-War, Diploma-Final-App-Client şi Diploma-Final-Web-Client. Se efectuează click-dreapta pe fiecare şi se apasă „deploy” în ordinea în care au fost deschise mai sus. Diploma-Final-App-Ejb reprezintă modulul conţinând logica de business a aplicaţiei, Diploma-Final-App-War conţine modulul web al aplicaţiei, Diploma-Final-App-Client conţine un client de tip consolă care accesează serviciul SOAP şi Diploma-Final-Web-Client conţine aplicaţia de tip interfaţă web pentru a testa framework-ul.

Pentru a demonstra funcţionalitatea acestui framework, am scris o aplicaţie web – Diploma-Final-Web-Client care pune la dispoziţie două pagini: „SingleWebSearch.jsp” şi „MultipleWebSearch.jsp” permiţând efectuarea unei căutări într-una respectiv mai multe surse de date aflate pe Internet. Rularea exemplelor de mai jos necesită efectuarea unui deploy după cum s-a indicat mai sus.

6.2. Efectuarea unei cereri pe o singură sursă de date

Se deschide browser-ul web şi se introduce următoarea adresă: http://localhost:8080/Diploma-Final-Web-Client/SingleWebSearch.jsp. Vom vizualiza o listă de servicii din care vom alege „Yahoo BOSS Web Search” şi vom apăsa butonul „Show Parameters”. Ca şi rezultat al acestor paşi, vom vizualiza pagina web din Figura 6.1.

În urma completării câmpurilor după modelul din Figura 6.1. vom acţiona butonul „Search!”. Rezultatul îl putem vizualiza ca şi text xml sau ca şi tabel, după cum e ilustrat în Figura 6.2.

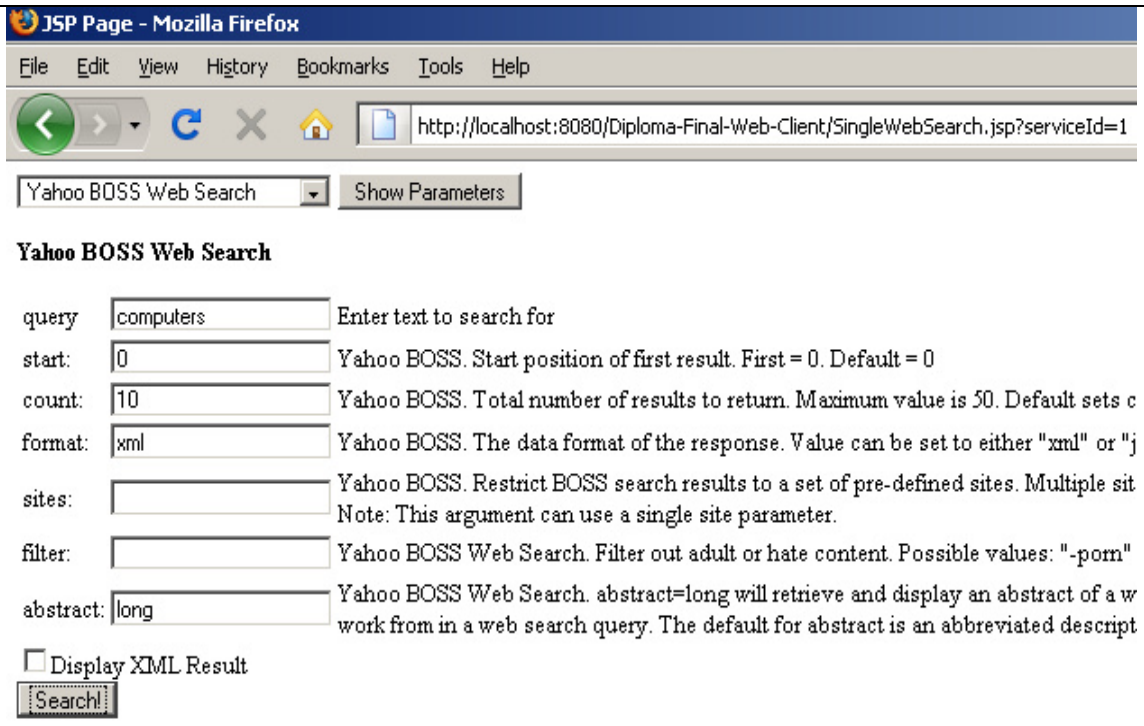


Figura 6.1. Pagina pentru efectuarea unei căutări într-o singură sursă de date.

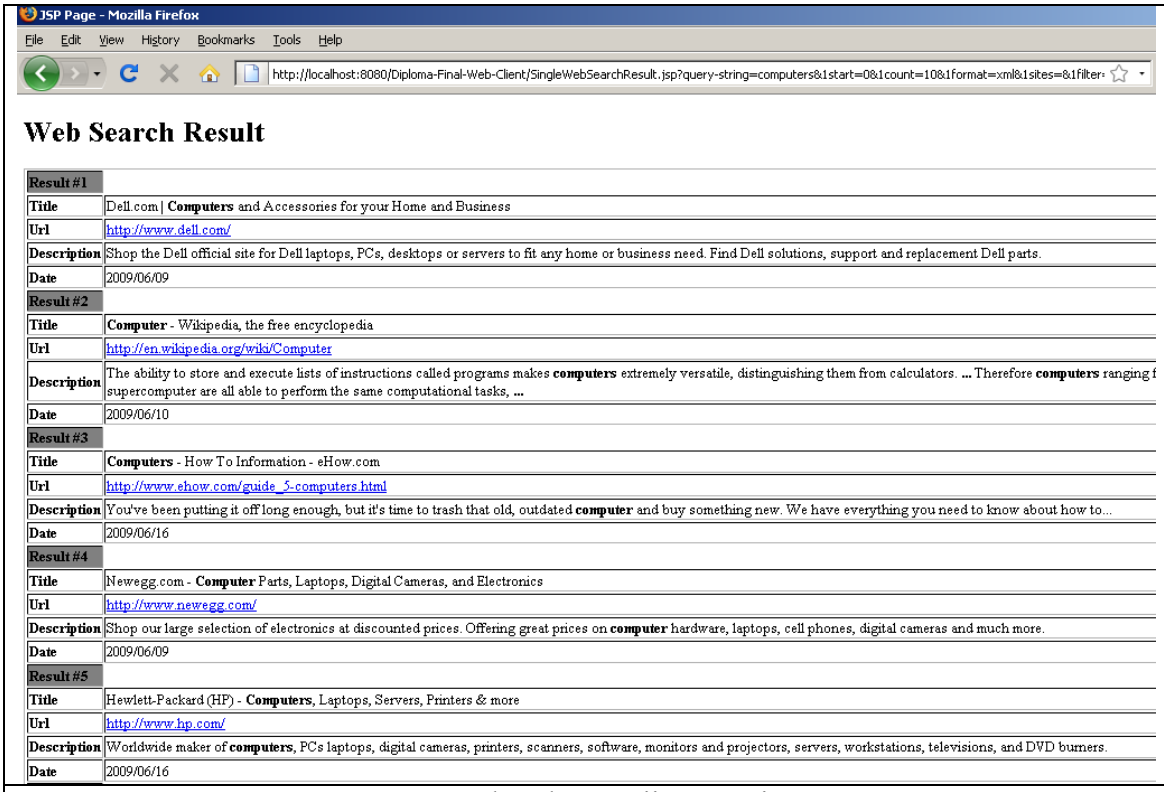


Figura 6.2. Rezultatul căutării într-o singură sursă

6.3. Efectuarea unei cereri pe mai multe surse de date

Se introduce următoarea adresă în browser: http://localhost:8080/Diploma-Final-Web-Client/MultipleWebSearch.jsp. Rezultatul, după ce se completează câmpurile este următorul:

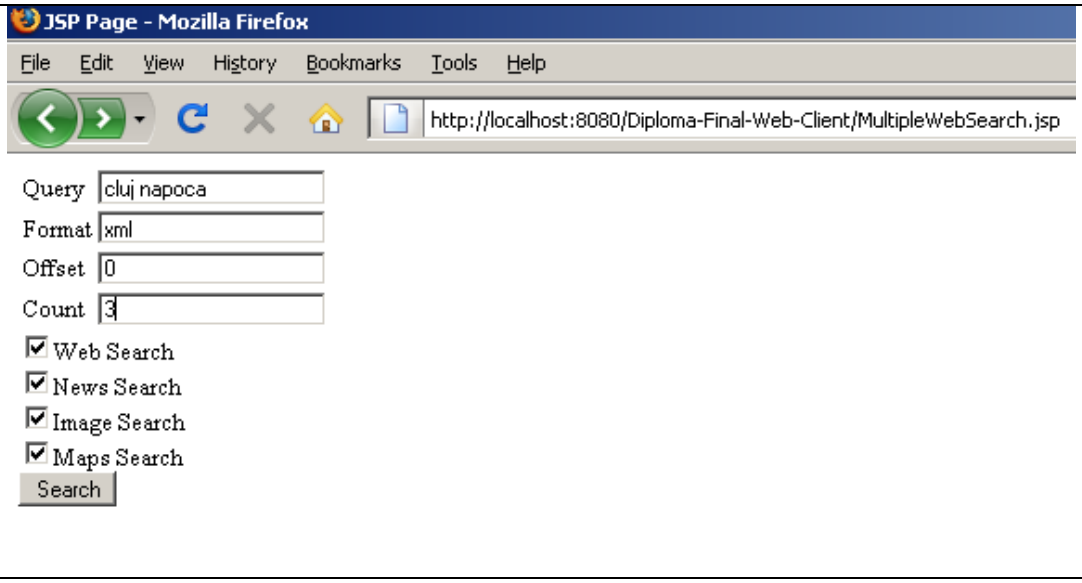


Figura 6.3. Pagina pentru efectuarea unei căutări în mai multe surse de date.

În urma căutării, rezultatul este afişat în următoarele două figuri:

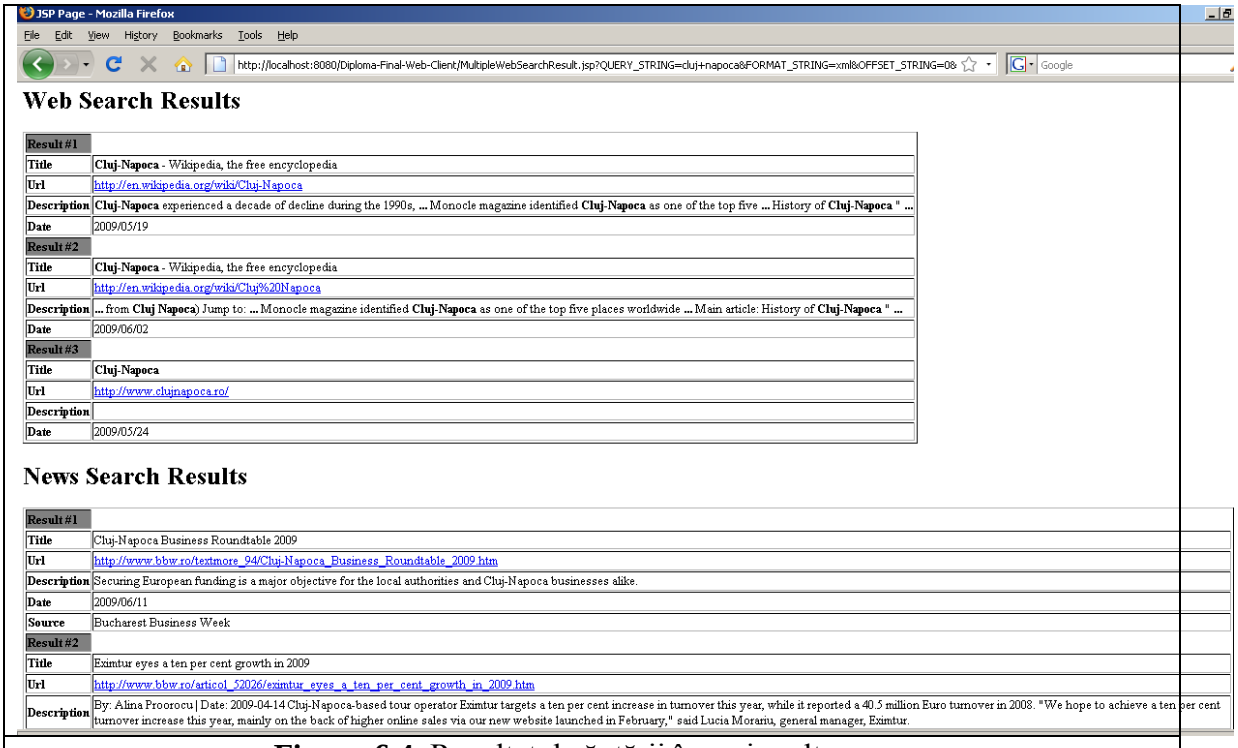


Figura 6.4. Rezultatul căutării în mai multe surse

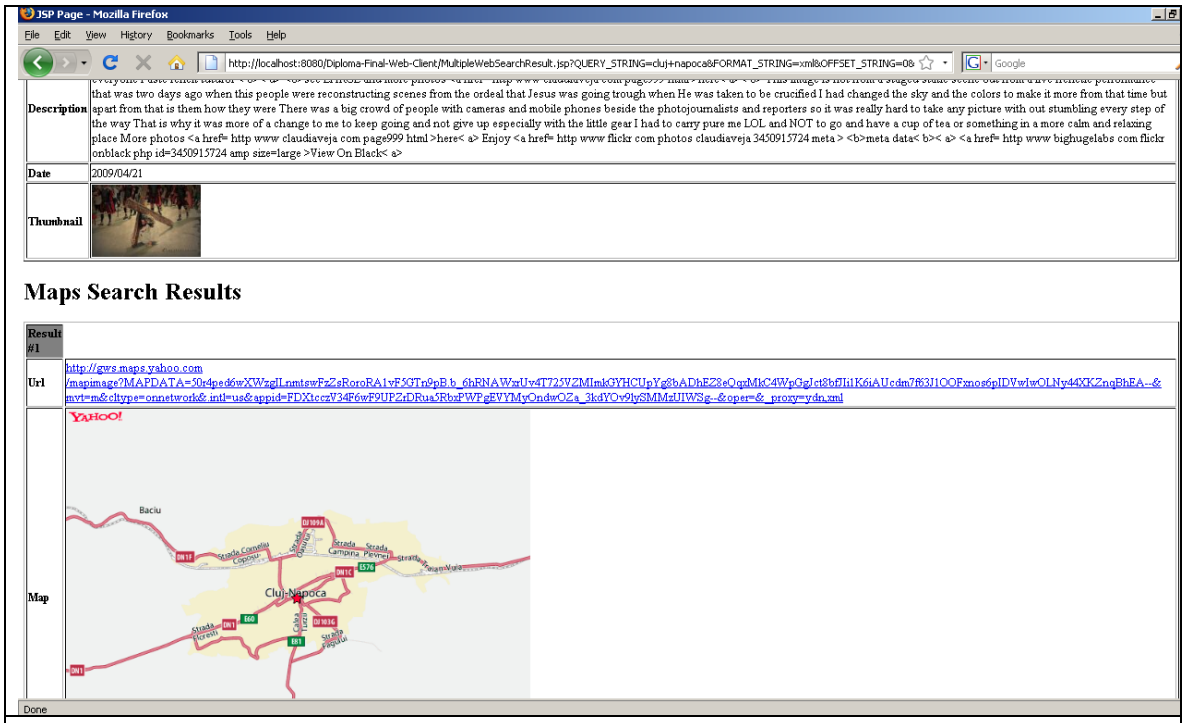


Figura 6.5. Rezultatul căutării în mai multe surse

6.4. Efectuarea unei cereri folosind clientul SOAP

Am mai pus dispoziţie în vederea testării un client de tip aplicaţie consolă (aplicaţia Diploma-Final-App-Client) care apelează serviciu SOAP al utilizatorului (nu cel al administratorului) pentru a efectua o cerere de căutare. Vom prezenta şi explica codul sursă al acestui client.

```
public static void main(String[] args) {
        //  Get the service and its port
        UserEndpointService service = new UserEndpointService();
        UserEndpoint port = service.getUserEndpointPort();

        SingleSearchRequest request = new SingleSearchRequest();

        Params params = new Params();

        Entry entry = new Entry();
        entry.setKey("service-id");
        entry.setValue("1");
        params.getEntry().add(entry);

        entry = new Entry();
        entry.setKey("query-string");
        entry.setValue("computers");
        params.getEntry().add(entry);

        entry = new Entry();
        entry.setKey("format");
        entry.setValue("xml");
        params.getEntry().add(entry);

        request.setParams(params);
```

```
        String result = port.singleSearch(request);

        System.out.println("Result: " + result);
    }
```

Metoda main este singura metodă a acestui client. Se poate obserca cum în primele două linii ale ei, instanţiem serviciul şi portul de acces la el. După aceasta, vom crea o cerere de tip căutare pe o singură sursă de date. Această cerere va trebui să conţină parametri specifici căutării. Pentru a furniza parametri neceari, ne vom folosi de clasa generată automat de către JAX-WS. A se observa că „SingleSearchRequest" din cadrul aplicaţiei client prezente, diferă de clasa „SingleSearchRequest" din cadrul framework-ului. Clientul foloseşte o reprezentare JAXB a clasei originale. De aceea, nu avem acces direct la obicetul „params" de tip Map, ci trebuie să ne creem o instanţă a clasei Params generată automat de JAXB. În cadrul ei vom adăuga noi obiecte de tip „Entry" care conţin numele parametrului şi valoarea sa. Doar după ce am adăugat în acest mod mai multe intrări în listă, vom putea efectua cererea. Odată ce am efectuat-o se va afişa rezultatul.

7. CONCLUZII

7.1. Realizări

Framework-ul propus îmdeplineşte obiectivele enunţate în secţiunea 1.3. Acestea sunt:

- **Acces unificat la resursele REST aflate pe Internet**.

  În baza de date care este gestionată de componenta WS Registry se pot stoca informaţii despre astfel de resurse. Detalii precum URL-ul, parametrii şi valorile aşteptate pot fi obţinute uşor prin intermediul interfeţei expuse de componentă.

- **Acces cât mai simplu la sursele de date**

  Utilizatorul framework-ului poate folosi serviciile REST pe care acesta le intermediază fără a cunoaşte politicile de securitate pe care acestea le cer. În plus, utilizatorul final nu trebuie să cunoască URL-ul serviciului El poate interoga baza de date şi invoca serviciul folosind identificatorul unic al acestuia.

- **Soluţie simplu de întreţinut şi dezvoltat**

  Datorită dezvoltării bazată pe module, soluţia propusă de noi poate fi uşor de întreţinut. Adăugarea serviciilor la baza de date este un procedeu simplu. Dezvoltarea acestui framework poate fi realizată cu uşurinţă pe viitor, datorită principiilor de design care au stat la bază, după cum rezultă din secţiunea 2.3 şi din capitolul 4.

- **Distribuirea aplicaţiei**.

  În cazul în care dorim să rulăm componentele pe noduri diferite, aceasta se poate face cu un minim de efort datorită folosirii unui server de aplicaţie şi a platformei Java EE. Singurul lucru care trebuie modificat este adresa la care se află componentele.

- **Manipularea cu uşurinţă a surselor de date**

  Adăugarea sau ştergerea surselor de date (adică a serviciilor REST) se face cu uşurinţă prin terminalul SOAP al administratorului. Procedeul este unul simplu şi uşor de efectuat.

7.2. Comparaţia cu sisteme existente

În cartea sa, [3], M. Hansen, propune o arhitectură pentru a integra diferite servicii de cumpărare online. Arhitectura integrează atât servicii REST cât şi SOAP pentru a pune la dispoziţie o modalitate de a căuta un produs în mai multe locaţii şi pentru a-l cumpăra găsind cea mai bună ofertă. Arhitectura propusă este totuşi specifică unui anumit domeniu şi nu permite integrarea generală a oricăror servicii web.

Microsoft este pe punctul de a lansa un nou motor de căutare, numit Bing (www.bing.com), folosit pentru a căuta informaţii de diferite tipuri într-un mod unitar(web,

imagini, etc.). Acest proiect este doar în stadiul beta. Trebuie să menţionăm că proiectul nu îşi propune să agrege date furnizate de diferiţi furnizori, ci va folosi doar datele oferite de propriile sale motoare de căutare.

LeapFish [20] este un portal care permite efectuarea unor căutări în mai multe locaţii (Google, Yahoo, MSN) şi de mai multe tipuri (web, imagini, ştiri, blog, video, shopping). La fel ca şi soluţiile de mai sus, aceasta nu prezintă o modalitate generică de a accesa surse de date prin servicii web. Spre exemplu, site-ul foloseşte API-ul REST AJAX al Google. Acest API se poate folosi doar prin intermediul unui browser.

7.3. Dezvoltări ulterioare

În secţiunea curentă vom trasa patru direcţii ale unor posibile dezvoltări ulterioare. Am ales acele propuneri de dezvoltare ulterioară pe care le-am considerat a fi cele mai importante.

7.3.1. Alegerea unui format comun de date

Soluţia propusă oferă posibilitatea de a efectua căutări într-una sau mai multe surse de date folosind parametrii specifici acestor surse de date. Se ridică problema găsirii unui format comun de date pentru cerere şi răspuns. De exemplu, atunci când formăm o cerere, nu dorim să încărcăm utilizatorul cu necesitatea de a cunoaşte parametrii specifici surselor de date invocate. În cazul surselor de date Yahoo şi Microsoft, deşi oferă aceleaşi servicii, numele parametrilor diferă. Nu dorim ca utilizatorul să fie conştient de diferenţa de nume a parametrilor sau de valorile diferite pe care aceşti parametri le pot avea în cazul a două surse de date diferite. Deci, pentru aceeaşi funcţionalitate, surse de date diferite pot oferi parametri cu nume diferit şi care iau valori diferite.

Ne dorim găsirea unui format comun de date între mai multe surse. Utilizatorul va trebui să fie familiarizat doar cu acel format comun de date şi nu cu formatul de date necesar fiecărei surse (ne referim aici şi la parametri unei cereri precum şi la formatul răspunsului primit de la sursa de date). Astfel, va trebui oferit un mecanism care să permită specificarea acestui format comun, precum şi a unor mapări între formatul comun şi formatul specific fiecărei surse.

În concluzie, clientul va defini o cerere de căutare pentru mai multe surse de date specificând o listă de parametri generici. Dispatcher-ul va trimite mai departe cererea către componenta de invocare a serviciilor web. Această componentă, folosindu-se de cererea generică pe care a primit-o va forma cererile specifice fiecărei surse cu ajutorul mapărilor definite. După ce sursele de date primesc cererea specifică şi o procesează, vor returna un rezultat în format propriu. Pornind de la formatul propriu se va forma un răspuns generic cu ajutorul mapărilor. Clientul va primi răspunsul generic şi îl va procesa în mod uniform independent de sursa de date care a oferit acest răspuns.

7.3.2. Folosirea web-ului semantic

Stadiul curent al aplicaţiei nu permite decât folosirea surselor de date REST. Cele SOAP nu se pot folosi. În plus, adăugarea unui nou serviciu este mai complicată în sensul în care administratorul care introduce serviciul este obligat să introducă şi informaţii care să descrie serviciul respectiv, precum şi parametrii săi împreună cu descrierile lor. Acest procedeu este necesar din cauză că utilizatorul formează cererile bazându-se pe informaţiile din registrul web. Utilizatorul trebuie să cunoască de exemplu numele fiecăui parametru în parte, valorile pe care le poate lua precum şi rolul parametrului în formarea cererii.

Pentru a oferi o flexibilitate mai mare în formarea cererilor, sursele de date ar trebui să ofere metadata cu caracter semantic despre serviciile web pe care le oferă. Astfel, indiferent dacă ele sunt de tip REST sau SOAP, s-ar putea include în framework un modul care să interpreteze descrierile serviciilor respective şi să fromeze automat, la cerere, un client pentru ele. În momentul de faţă, aplicaţia nu poate efectua acest lucru tocmai din cauza faptului că serviciile nu oferă informaţii semantice cu privire la operaţiile şi parametri pe care îi oferă.

Scopul serviciilor web semantice este acela de a permite descoperirea, medierea, selecţia, compunerea şi invocarea automată a lor permiţând aplicaţii flexibile. WSMO (Web Service Modeling Ontology) identifică elementele conceptuale care sunt necesare unei astfel de descrieri, oferind modalităţi de descriere a serviciilor din punctul de vedere al utilizatorului. RDF şi OWL sunt limbaje pentru a descrie ontologii ale web-ului semantic. WSML este un limbaj care permite descrierea funcţionalităţii şi a comportamentului serviciilor web [2].

7.3.3. Folosirea BPEL şi OpenESB

Procesele business descrise au fost implementate programatic, folosind un limbaj de programare obiectual. Comunicarea între diferitele componente se face prin apel de metodă şi transmitere de parametri. O îmbunătăţire a acestei abordări este folosirea unui limbaj dinamic de execuţie împreună cu un bus de servicii. Varianta propusă de noi estea aceea a folosirii BPEL împreună cu OpenESB – bus-ul enterprise pentru servicii oferit de Sun care implementează BPEL. OpenESB este inclus în ultima distribuţie de GlassFish, sau se poate descărca separat de la adresa [26].

JBI este o specificaţie dezvoltată pentru a implementa SOA. E bazată pe modelul serviciilor web şi oferă o arhitectură "pluggable" pentru un container care găzduieşte componente care pot fi consumatori sau provideri de servicii. Serviciile se conectează la container cu ajutorul unor "binding components" (BC) sau pot fi găzduite de container cu ajutorul unui "service engine" (SE). Mecanismul folosit pentru comunicare este Normalized Message Router (NMR). JBI e integrat în GlassFish. O aplicaţie Java EE poate fi împachetată ca şi modul JBI şi rulată. "Service engine"-urile pot fi de tipul: BPEL, Java EE, SQL, XSLT şi altele. "Binding component"-urile pot fi de tipul: File, SMTP, FTP, SOAP, JDBC, JMS şi altele.

Mediul de dezvoltare NetBeans oferă unelte pentru asistarea în ceea ce priveşte dezvoltarea componentelor JBI şi a integrării folosind OpenESB. Acesta conţine un Designer BPEL care permite descrierea grafică a secvneţelor BPEL, un editor WSDL şi un designer pentru schemele XML. Pentru o detaliere a modului de dezvoltare a unei astfel de aplicaţii, recomandăm lucrarea [10].

7.3.4. Suport complex pentru servicii REST

Framework-ul propus de noi oferă suport pentru servicii REST definite printr-un singur parametru. Cu toate acestea, am observat în cadrul secţiunii „Fundamentare teoretică" faptul că un astfel de serviciu poate răspunde la un număr infinit de URI-uri. De exemplu, dacă numele utilizatorilor se găseşte la adresa service.com/users, atunci putem construi interogări de tipul service.com/users/John pentru a returna informaţiile referitoare la utilizatorul cu numele „John". Procedeul este analog cu introducerea unei variabile de parametru în cazul API-ului JAX-RS. În acest moment, nu oferim suport pentru astfel de cereri.

Un alt aspect care poate fi îmbunătăţit este acela al oferirii suport pentru metodele HTTP POST, PUT şi DELETE. În prezent nu oferim suport decât pentru GET. Astfel, dacă una dintre sursele de date accesate oferă suport pentru o altă metodă în afară de GET, noi nu putem îndeplini în stadiul actual cererea respectivă.

Bibliografie

[1] **Basham Brian**, **Sierra Kathy**, **Bates Bert**, *Head First Servlets and JSP*, O'Reilly, 2008
[2] **de Bruijn Jos**, **Fensel Dieter**, **Kerrigan Mick**, **Keller Uwe**, **Lausen Holger**, **Scicluna James**, *Modeling Semantic Web Services. The Web Service Modeling Language*, Springer, 2008
[3] **Hansen D. Mark**, *SOA Using Java Web Services*, Prentice Hall, 2007
[4] **Heffelfinger R. David**, *Java EE 5 Development using GlassFish Application Server*, Packt Publishing 2007.
[5] **Kalin Martin**, *Java Web Services: Up and Running*, O'Reilly, 2009
[6] **Richardson Leonard**, **Ruby Sam**, *RESTful Web Services*, O'Reilly, 2007
[7] **Ronsen Michael**, **Lublinsky Boris**, **Smith T. Kevin**, **Balcer J. Mark**, *Applied SOA. Service-Oriented Architecture and Design Strategies*, Wiley, 2008
[8] **Sriganesh Rima Patel**, **Brose Gerald**, **Silverman Micah**, *Mastering Enterprise JavaBeans 3.0*, Wiley, 2006
[9] **Salomie Ioan**, **Cioară Tudor**, **Anghel Ionuţ**, **Salomie Tudor**, *Distributed Computing and Systems. A Practical Approach*, Editura Albastră, 2008
[10] **Salter David**, **Jennings Frank**, *SOA Based Composite Applications Using NetBeans IDE 6 (XML, BPEL, Java WS)*, Packt Publishing, 2008
[11] **Spies Brennan**, *Web Services, Part 1: SOAP vs. REST*, http://www.ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest
[12] Consuming RESTful Web Services With the Jersey Client API, http://blogs.sun.com/enterprisetechtips/entry/consuming_restful_web?services_with
[13] Enterprise JavaBean, http://en.wikipedia.org/wiki/EJB
[14] **Fielding Roy**, *Architectural Styles and the Design of Network-based Software Architectures*, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, Teză doctorat, 2000
[15] GlassFish Project, https://glassfish.dev.java.net/
[16] Java EE Sun Developer Network, http://java.sun.com/javaee/.
[17] Java Platform, Enterprise Edition, http://en.wikipedia.org/wiki/Java_ee
[18] JSR 311: JAX-RS: The Java™ API for RESTful Web Services, http://jcp.org/en/jsr/detail?id=311
[19] Jersey Home Page, https://jersey.dev.java.net/
[20] LeapFish, https://glassfish.dev.java.net/
[21] Microsoft Live Search API Reference, http://msdn.microsoft.com/en-us/library/bb266180.aspx
[22] Microsoft Live Search Developer Center, http://dev.live.com/
[23] Microsoft Windows Live ID, http://login.live.com/
[24] Mozilla Firefox Download, http://www.mozilla-europe.org/ro/firefox/
[25] NetBeans 6.5. Download, http://www.netbeans.org/downloads/start.html?platform=windows=en&option=java&version=6.5.1
[26] OpenESB Project, https://open-esb.dev.java.net/
[27] SOAP, http://en.wikipedia.org/wiki/SOAP
[28] Web Service Description Language, http://en.wikipedia.org/wiki/Web_Services_Description_Language
[29] Yahoo! BOSS API Guide, http://developer.yahoo.com/search/boss/boss_guide/
[30] Yahoo! Developer Page, http://developer.yahoo.com/

Acronime

API – Application Programming Interface
BA – Business Architecture
BPEL – Business Process Execution Language
BPM – Business Process Management
BPMN – Business Process Modeling Notation
BPMS – Business Process Management System
CORBA – Common Object Request Broker Architecture
CRUD – Create, Read, Update, Delete
DCOM – Distributed Common Object Model
DSL – Domain Specific Language
EA – Enterprise Architecture
EJB – Enterprise Java Beans
ESB – Enterprise Service Bus
HTTP – HyperText Transfer Protocol
HTML – HyperText Markup Language
ISAPI – Internet Server Application Programming Interface
JAX-RPC – Java API for XML based RPC
JAX-WS – Java API for XML Web Services
JAX-RS – Java API for XML-RESTful Web Services
JBI – Java Business Integration
JNDI – Java Naming and Directory Interface
JMX – Java Messaging System
JPA – Java Persistence API
MIME – Multipurpose Internet Mail Extensions
MOM – Message Oriented Middleware
MEP – Message Exchange Pattern
MQ – Message Queue
NMR – Normalized Message Router
OWL – Web Ontology Language
RDF – Resource Description Framework
REST – REpresentational State Transfer
ROA – Resouce Oriented Architecture
SA – Software Architecture
SCA – Service Component Architecture
SLA – Service Level Agreements
SOA – Service Oriented Architecture
SOAP – Simple Object Access Protocol
UDDI – Universal Description, Discovery and Integration
WS-BPEL – Web Service Business Process Execution Language
WS-CDL – Web Service Choreography Description Language
WSDL – Web Service Description Language
WSMO – Web Service Modeling Ontology
WSML – Web Service Modeling Language
XML – Extensible Markup Language

A1 – Articol selectat pentru publicare în revista ASCS 2009

A framework for accessing data resources on the Internet exposed as RESTful web services

Lect. Eng. C. Ivan, Stud. G.D. Borza
cosmina.ivan@cs.utcluj.ro, d_borza@yahoo.com
Technical University in Cluj-Napoca

Abstract - **Over the last decade, Internet users have been using web search intensively. However, the ever-increasing number of web search providers makes it difficult to search for data in different data sources. More than that, developers face the problem of accessing different such resources exposed as web services by having to support various sets of constraints and rules. This framework aims at identifying a common way to accessing these different data sources. We also provide a common and simple data model for some of these various sources, thus helping the developer and ultimately the end-user finding and combining results from more than one web search provider. Thus, applications that search for a resource simultaneously at different web search providers will be developed more easily.**

Keywords
web services, REST, Internet search, SOA

I. INTRODUCTION

Internet search has become a common aspect of our lives. We use it on daily basis in order to find important information. We are also witnessing an increase of Internet search providers. The existence of multiple data sources on the Internet is benefic to the community. More providers means greater competition and inherently an increase in the quality of their services. Despite all these advantages we can think of, this ever-increasing number of data sources is becoming more and more difficult to manage by the end-user and the software developer alike.

For instance, the end user has to open multiple browser windows and enter the search query in each of these. Wouldn't it be more simple if s/he had the chance to access a portal, enter the query string only once and select the web search provider(s) that are most appropriate? The response results from various sources will be displayed in the browser in a common interface. The developer also faces a challenge in having to design and implement software clients: one for every data source. Each of the web search providers or data sources on the Internet offers a specific way of accessing their data, a different URL, different query parameters, different authentication mechanism, etc. Wouldn't it be easier if s/he could write a client in the same manner for each of these sources?

Still, the greatest difficulty of all may lie in the fact that each provider offers its data in a different format and structure. Some use XML, the others JSON, etc. But, even if the data format is the same (e.g. XML), the structure differs. There is no common way of specifying a unified model of data structure and semantics, although research is being made [5].

Web Services have emerged in the last decade as an important way to leverage interoperability and integration between heterogeneous software systems as [3] suggests. More complex applications can be built starting from existing ones with the use of web services. They offer a common interface between the systems. Two major types of web services are being used at the moment: SOAP-based and REST-based [9]. SOAP (Simple Object Access Protocol) is a suited protocol since it is platform and language independent. More than that, it can be mapped on various transport protocols, HTTP being the most often used. However, because of it's XML format it induces a great transport overhead [7]. As an alternative, RESTful (REpresentational State Transfer) web services have emerged lately as an alternative to SOAP services [8]. They use HTTP as transport protocol with it's set of standard, well-known operations (GET, POST, PUT, DELETE, etc.). Therefore, RESTful web services are suited to accessing data over the Internet, since they use HTTP directly with it's simple, built-in operations.

Recently, the major web search providers published a set of REST APIs. These API's can be used by developers to build software that uses their search functions. Microsoft [6], Yahoo [10], Google [4], Amazon and eBay are well-known examples. Let's consider the following scenario: we want to search for the query "computers" in the section news and retrieve the results in xml format. Both Microsoft and Yahoo offer a way to accomplish the task. At Yahoo!, we would have to submit the following URL: `http://boss.yahooapis.com/ysearch/news/v1/computers?appid={appID}&format=xml`. Microsoft would provide the same service at the following URL: `http://api.search.live.net/xml.aspx?Appid={appID}&query=computers&sources=news`. The appID is a parameter specific to each search provider. We observe from this simple example, that each web search provider has a specific way of building the request URL, a different set of parameters, etc. In a similar fashion, databases might be queried using this approach too. Suppose we have an database with users stored in it. Each user has an id, a name and an address. This database might be exposed with the ease of web services at the URL http://database.company.com. If we would like to search for the user with id=5, the following HTTP request would suffice: GET http://database.company.com/users/id=5. We could imagine other scenarios such as listing all the users by using the following request: GET http://database.company.com/users. Alternatively, we could query for all the users that start with "A" using GET http://database.company.com/users/name="A*". Therefore, we conclude that accessing a database over the Internet is as simple as issuing a corresponding HTTP GET method. Still, there is no unified way to accessing all these data sources (web search results or databases). We propose a solution that aims in facilitating the access to various RESTful data sources over the Internet, offering a common registry for storing information about these data sources such as their URL, expected parameters and description. We will provide a generic way to form requests for all of the web services. The requests will contain the service id, the query string and the needed parameters. Search queries will be executed against multiple data sources with ease. Only one authentication mechanism will be needed. More than that, we propose a simple yet intuitive solution in defining a generic data model for more than one source. In this way, one could query Yahoo! and Microsoft using the same set of generic parameters, instead of forming two different requests, each using specific parameters. Therefore, given N different data sources we will provide the mechanism to define one generic model and N mappings between the generic data model and the model used by the specific data sources.

As a proof of concept we will build an application on top of the framework proving the way in which one can form a request on a single data source, a request on multiple data sources and a generic request using a common data model between two data sources.

II. METHOD

In order to build the framework for accessing different such web search providers in a common way, we used the principles of SOA – Service Oriented Architecture. We use the existing web services as portals to data sources and also in order to retrieve the results. Access to our framework is available through a SOAP client. With the use of this client we build a request a send it to the business logic. This request is decoded and the appropriate action takes place.

A. Architecture and Design

We will present the framework logical architecture and briefly describe it's logical modules as shown in "Fig. 1".

SOAP Service Module – enables the client to execute a search request towards one or more search sources. This module exposes a web services which can be called to perform queries on the database alongside with other operations (logging, building custom requests). The SOAP Client exposes a method named submitRequest which takes as parameter an instance of an implementation of the abstract class GenericRequest. Our framework provides three implementations of GenericRequest: SingleSearchRequest, MultipleSearchRequest and GenericSearchRequest. SingleSearchRequest is used in order to form a request on a single data source. MultipleSearchRequest is composed of at least one SingleSearchRequest. It is used for executing queries on more the one data source.

GenericSearchRequest is a special type of request. We use it in order to form generic search requests. These requests use a common data model for different data sources. We pass them a generic set of parameters, and the business logic maps this generic set of parameters on a specific set of parameters that are understandable by each of the web service providers.

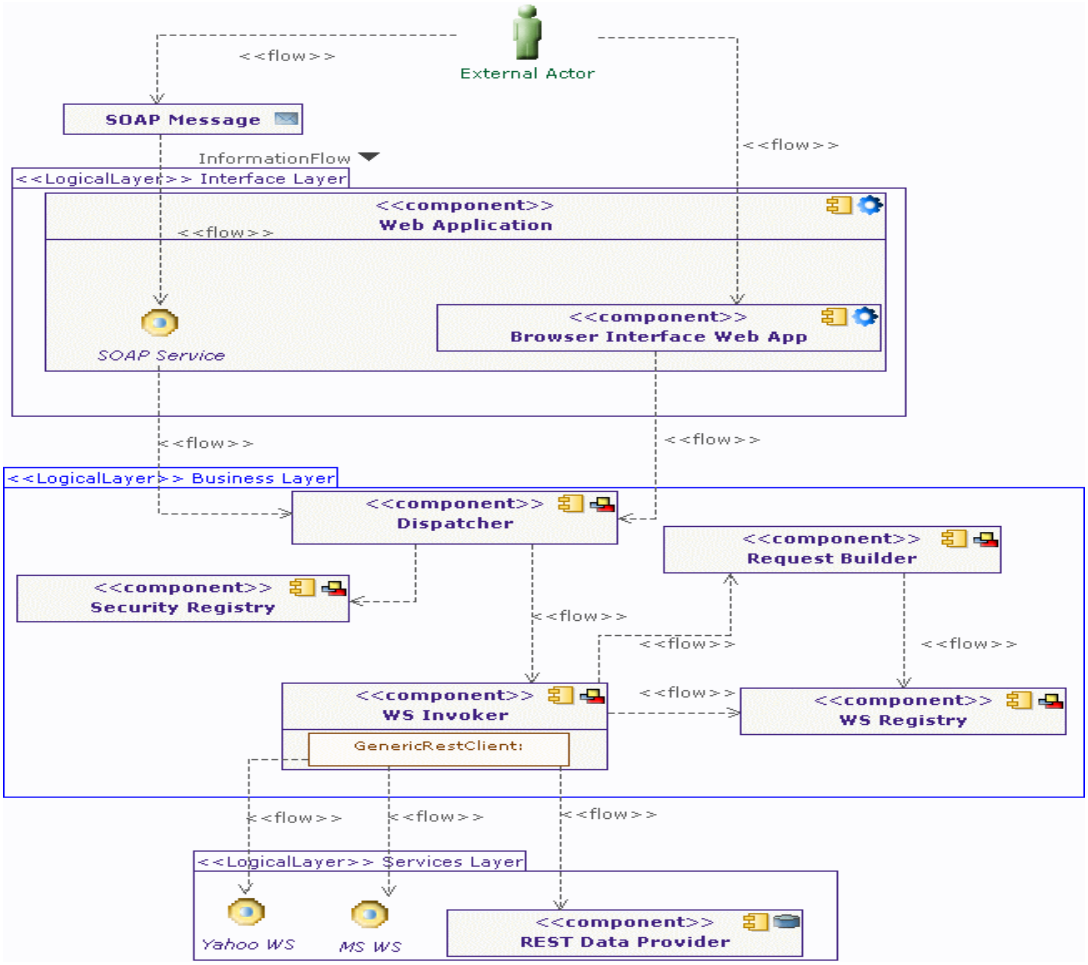


Figure 1. Conceptual Architecture

Browser Web App Module – offers a web-browser based approach to the framework. This module is still in work and will enable the end-user to build requests in a graphical way. He will be able to select various data sources and execute a simultaneous search against these data sources. The results will be displayed in a uniform style.

Dispatcher module – this module analyzes the request and routes it to the specific handler. The types of requests are: user login, user logout, user single search, user multiple search, user generic search. If the request is of type user login/logout then the security registry will be invoked to handle this specific request. If the request is of type user single search, user multiple search or user generic search, the WS Invoker module will be used to handle the search request.

Security Registry module – this module uses a database in order to authenticate and authorize the users. It receives the requests and based on the user credentials it grants or not a specific privilege.

WSInvoker Module – this module lies at the core of our framework. It receives the search request and analyzes it. This module makes extensive use of the SingleWsInvoker class which is used to start a thread that executes the search on a single data source with parameters specific to that data source. SingleWsInvoker uses an instance of the GenericRestClient class. This class uses the Jersey API to facilitate the communication with a REST resource on the Internet. If the request received is of type SingleSearchRequest, only one instance of the SingleWsInvoker class is used. However, if the request is ot type MultipleSarchRequest, the module launches into execution concurrently more SingleWsInvoker threads. The most important part is that of GenericSearchRequest. If the requestparameters to the specific parameters forming a SingleSearchRequest or a MultipleSearchRequest which are executed as described above. In order to achieve the mapping, we use the RequestBuilder module.

WS Registry Module – contains a database with information regarding the web services. The developer has access to this registry, collecting the information needed to form a request. Available information to the developer is the following: the name of the services, their unique ids, and the parameters they used alongside their description.

Request Builder module – this is the document that enables the use of generic requests and generic responses. It uses a database that defines the generic request, the generic responses and the mappings between these and the service specific parameters. This module has methods that receive a GenericSearchRequest and, using the specific mapping return a SingleSearchRequest or a MultipleSearchRequest that can be easily handled by the WSInvoker module. More than that, given a specific response retrieved from the SingleWsInvoker they form a GenericResponse composed of generic response fields.

RESTful Data Providers – we execute the search queries against them. These are external APIs provided by web search providers (such as Yahoo! and Microsoft) and by RESTful databases. These are the "data sources" available on the Internet used to form the results.

B. Implementation Details

We use the Java platform in order to build the framework. The reasons we chose this platform is because if offers a complete stack of tools that integrate well and are easy to use, yet powerful enough to model such a complex task.
We use the JAX-WS specification with its Metro implementation to implement and access a the SOAP module. In order to implement the access to the REST resources on the Internet, we used the JAX-WS specification with the Jersey implementation. Of course, we could use simple Http communication, but this is more cumbersome as [2] illustrates. A better option that is also recommended by the author is the one that we adopted. Jersey is a new API and will be shipped as part of the Java EE 6 specification. To illustrate the ease of use with Jersey, let's consider the following REST web service invocation:

```
Client client = Client.create();
WebResource wr =
       client.resource(finalUrl);
String response =
       wr.get(String.class);
```

EJB3.0 technology is used to implement the business logic. The various modules described earlier are designed as beans in order to achieve SOA goals as: separating service implementations from service interfaces with the use of remote interfaces, modularity, reusability of code, service granularity and integration of other existing systems such as databases. EJB3.0 also offers container management for multithreading on the server-side. Therefore, server-side programming is relatively easy. Persistence is also provided by means of the entity beans and annotations. The container can easily handle references to database resources that are stored on the server. The application server used is GlassFish.

III. RESULTS

Next, we will offer an example of accessing a web service. The service used will be yahoo web search. The yahoo web search is stored in the registry at id=1. The query string is "computers".

First, we build the request by using a parameter map. "service-id" and "query-string" are general parameters. Using them, we are able to identify the service we want to access and the string we want to search for. If, for instance, we want to obtain information about the available services, their parameters and expected values, we can use the SOAP endpoint. But, for this example we assume that we know that service-id=1 is the id assigned to yahoo web search. "start", "format" and "count" yahoo web search specific parameters. Information about these is available by using the SOAP endpoint or accessing the Yahoo site [10].

```
Map params = new HashMap();

params.put("service-id", "1");
params.put("query-string",
"computers");
params.put("start", "0");
params.put("format", "xml");
params.put("count", "10");

SingleSearchRequest request =
   new SingleSearchRquest(params);
```

After building the request, we invoke the web service through the SOAP endpoint.

```
SOAPServicePort service =
                    Utils.getSOAPServicePort();
String result = service.submitRequest(request);
```

After retrieving the result, we can parse the result as we wish. As seen, accessing a resource is very easy. We are able to identify the corresponding service's id and it's parameters in a straightforward manner (using the SOAP endpoint). Building the parameter list and submitting the request is also task that involves no difficulty.

This is a process particular to each request, since it is not a generic request. It uses specific parameters, used by the web service invoked. Unfortunately, at this stage we aren't able to provide a generic request example since the module is in the design and implementation phase.

IV. RELATED WORK

In his book [1], Mark D. Hansen uses SOA for integrating eBay, Amazon and Yahoo! Shopping. He builds a SOA application for searching and buying items from different locations. This was the

starting point of our approach. In this work, the author offers access to various resources on the Internet (http://search.creativecommons.org/) mechanism, but without trying to integrate the results. Their browser-based search portal has the option of searching google, yahoo, flickr, blip.tv, owl and spinxexpress. However, they merely offer a portal to these services by displaying those pages in theirs.

Leapfish (http://www.leapfish.com/) offers a similar functionality with that of my framework, but also without trying offer a unified view on the different data sources. I think it is somewhat better than Creative Commons, in the sense that it offers the possibility to simultaneously search different sources (web, news, blogs) and display the results in a unified form.

As shown by [5], the task of achieving a common data representation over the Internet is a current issue that is being researched. This is one of the reasons we weren't able to offer a generic approach towards web search. Besides that, we could provide a mechanism for defining and accessing generic data sources by means of specific mappings. The different formats and structures of the data provided by the data sources is also a great issue in finding such an approach.

V. Future Work

More web search providers should be integrated using this pattern. Besides that, refinements could be made in the way we map generic data sources to specific data sources.

Still, research has to be made in the fields of semantic web, ontology and semantic web services to overcome the issues mentioned before. A more comprehensive security mechanism could also be integrated in the current framework. The Request Builder module is yet to be implemented.

References

[1] Hansen D. Mark, *SOA Using Java Web Services*, Prentice Hall, 2007

[2] Kali Martin, *Java Web Services Up and Running* 1st ed, O'Reilly, 2009

[3] Rosen Michael, Lublinsky Boris, Smith T. Kevin, Balcer J. Marc, *Applied SOA. Service Oriented Architecture and Design*, Wiley Publishing 2009

[4] Google REST API, http://code.google.com/apis/ajaxsearch

[5] Huynh D.F., Karger D.R., "Adopting a Common Data Model for End-User Web Programming Tools", htp://davidhuynh.net/media/papers/2009/chi2009-eup.pdf, 2009, unpublished

[6] Microsoft Live Search, http://dev.live.com/livesearch

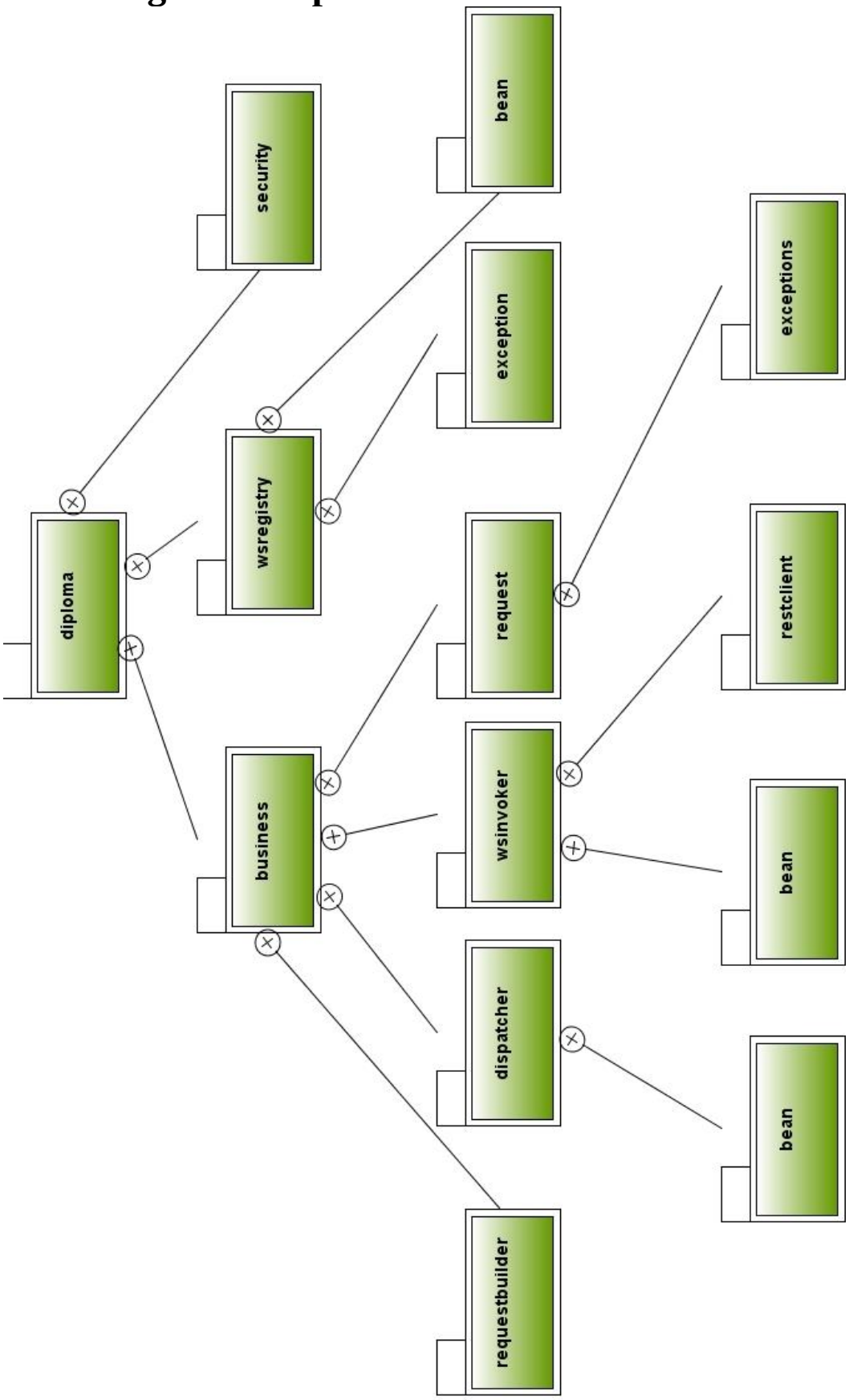
[7] REST, http://en.wikipedia.org/wiki/Representational_State_Transf

[8] SOAP Protocol, http://en.wikipedia.org/wiki/SOAP

[9] Web Service, http://en.wikipedia.org/wiki/Web_service

[10] Yahoo Web Services, http://developer.yahoo.com

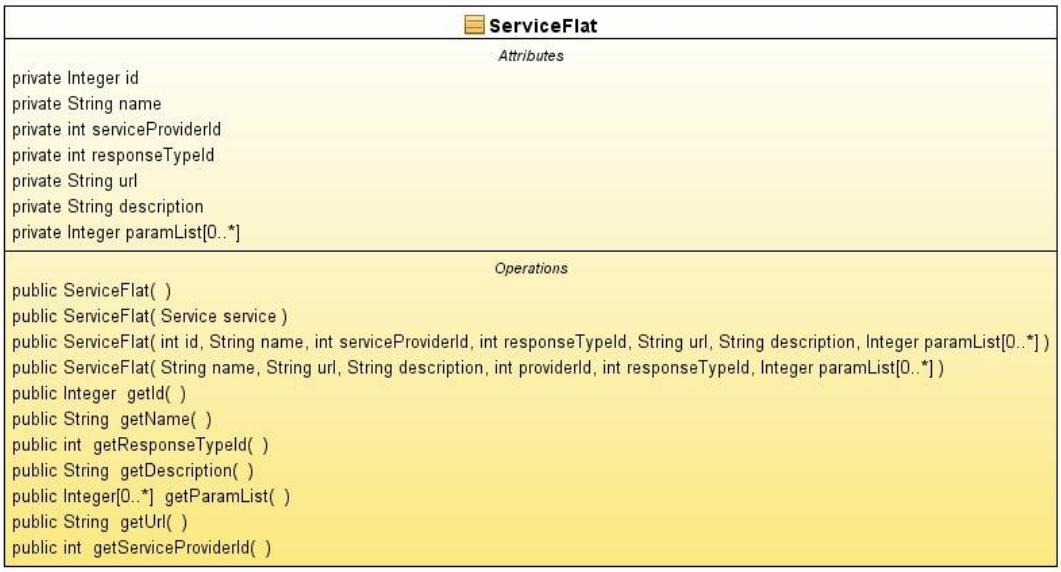
A2 – Diagrama de pachete a nivelului business

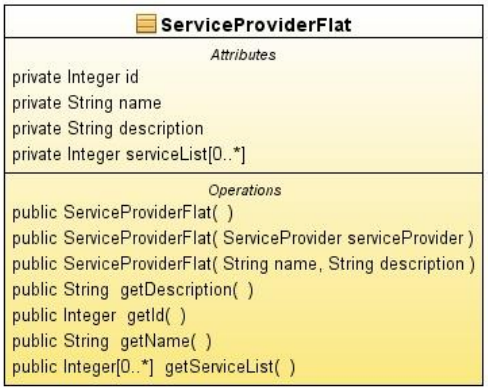


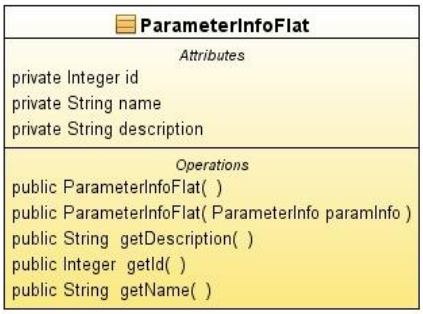
A3 – Diagrama de clasă a tipurilor de cereri care pot fi efectuate (diploma.business.request)

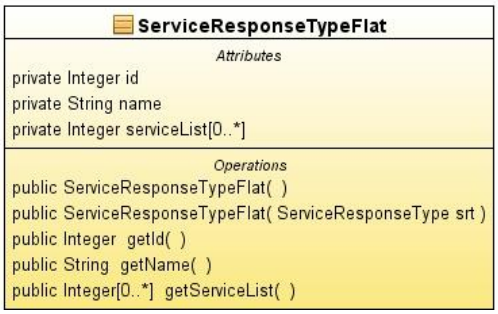
A4 – Diagrama de clase a entităţilor ce reprezintă registrul de servicii (incluse în pachetul diploma.wsregistry.bean.data)

A5 – Diagrama de clase a claselor derivate din entităţile registrului de servicii (incluse în pachetul diploma.wsregistry.bean.data)

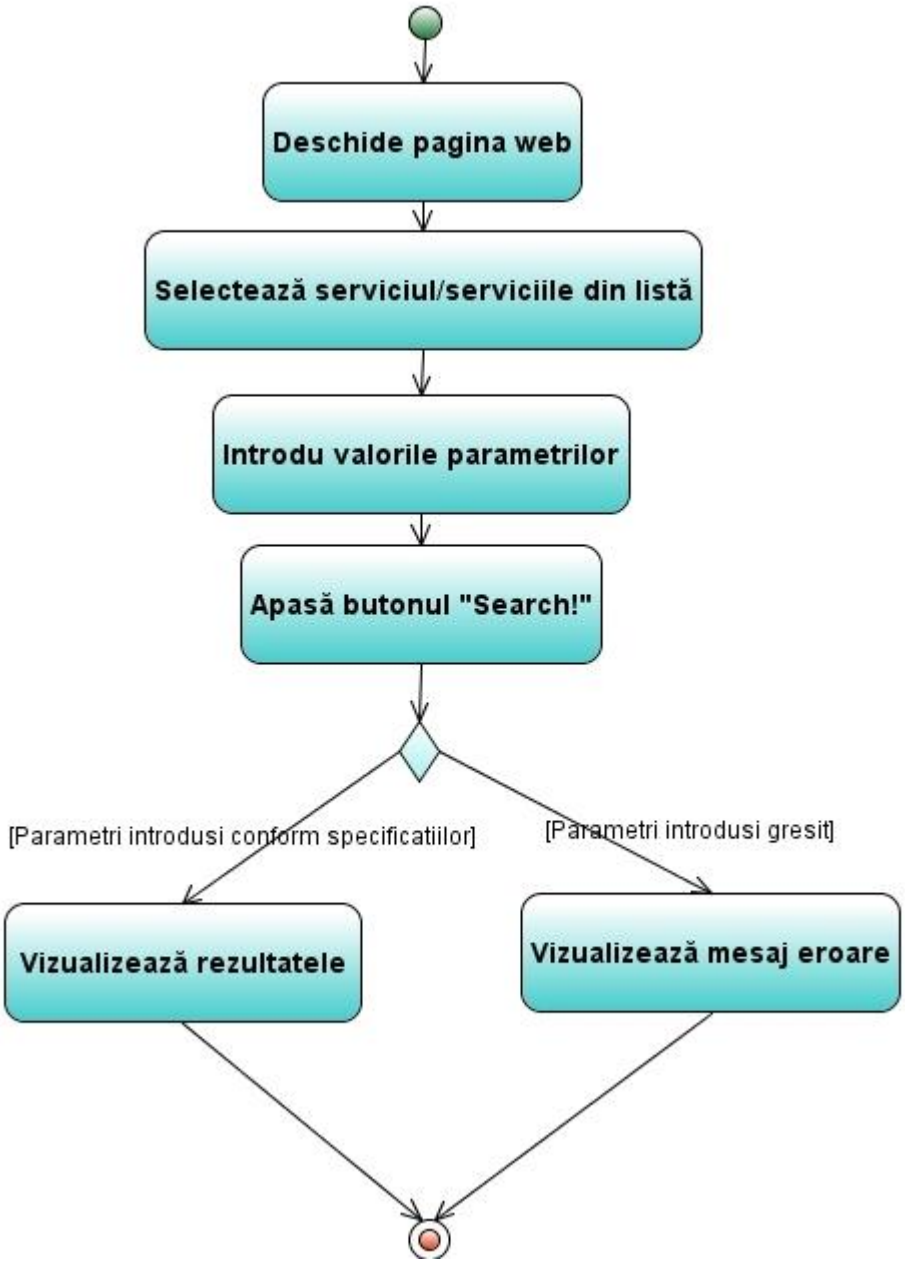




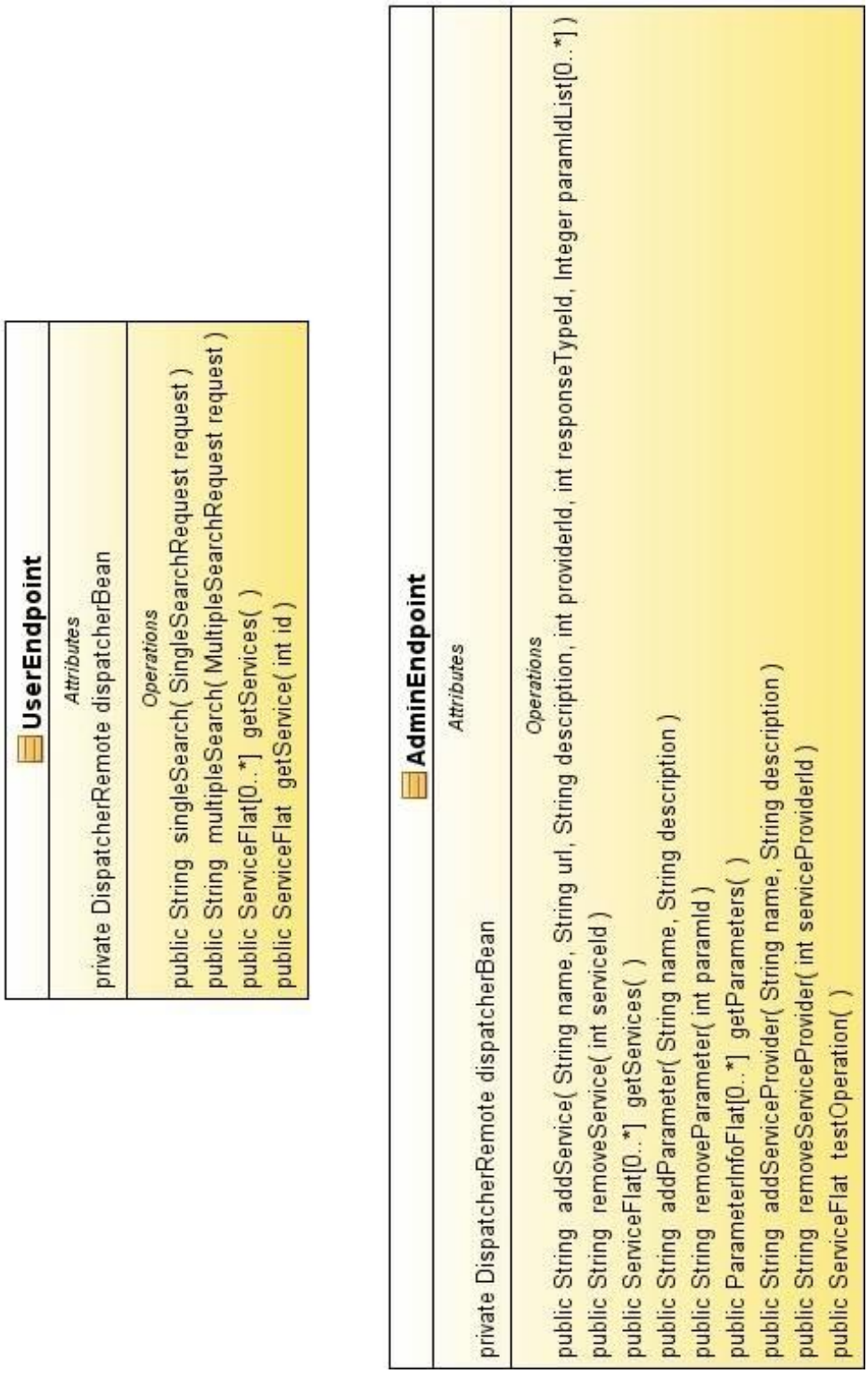




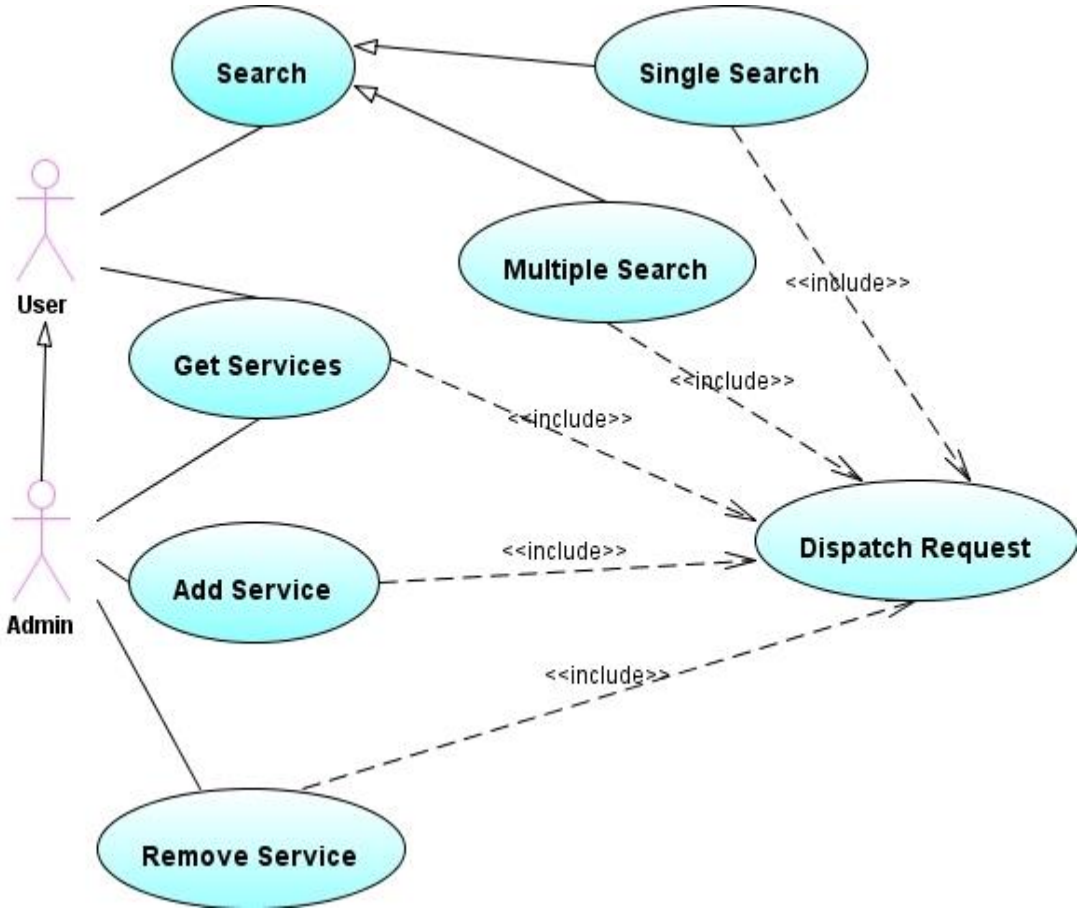
A6 – Diagramă de activităţi a componentei Dispatcher



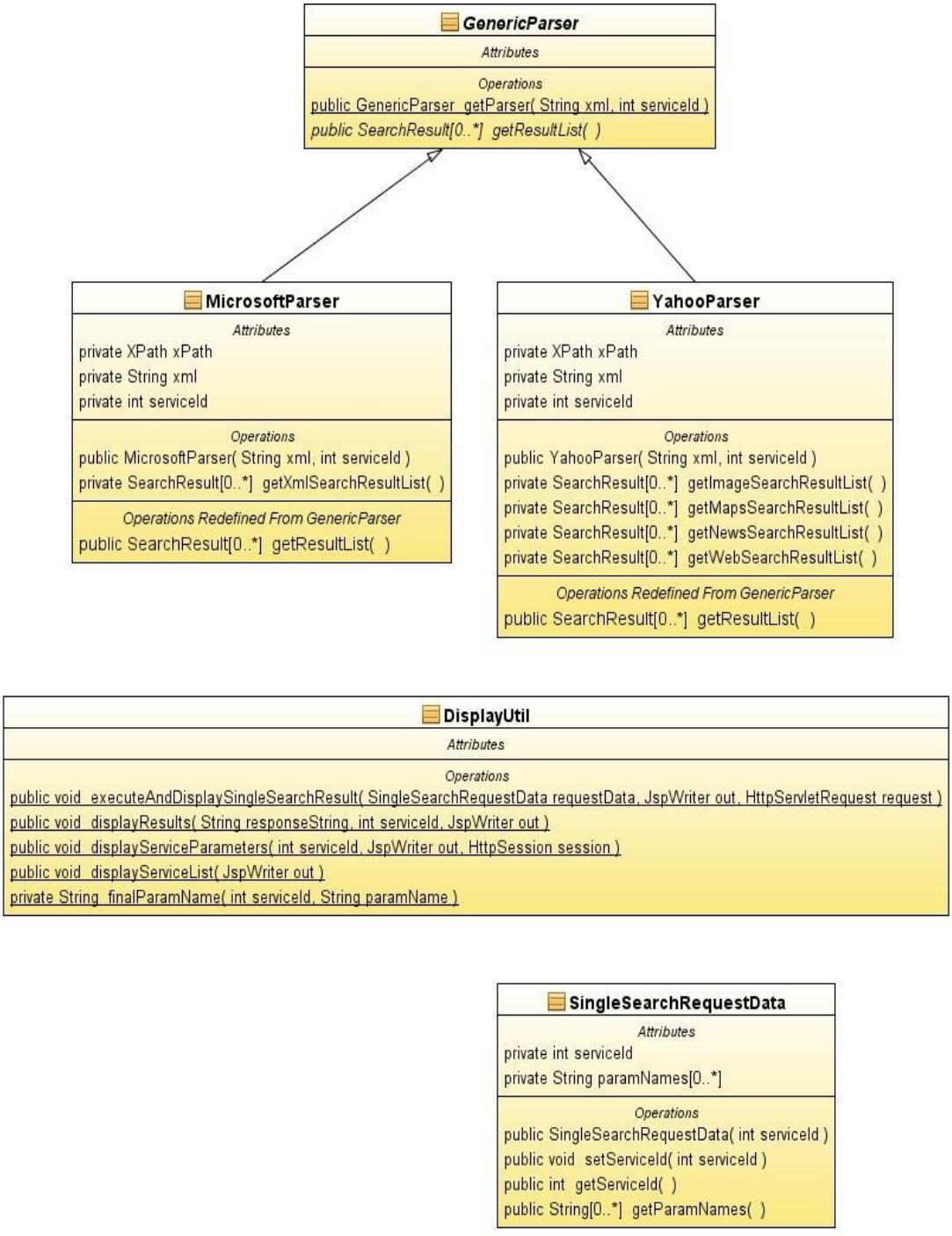
A7 - Diagrama de clasă ale punctelor de acces SOAP la aplicaţie



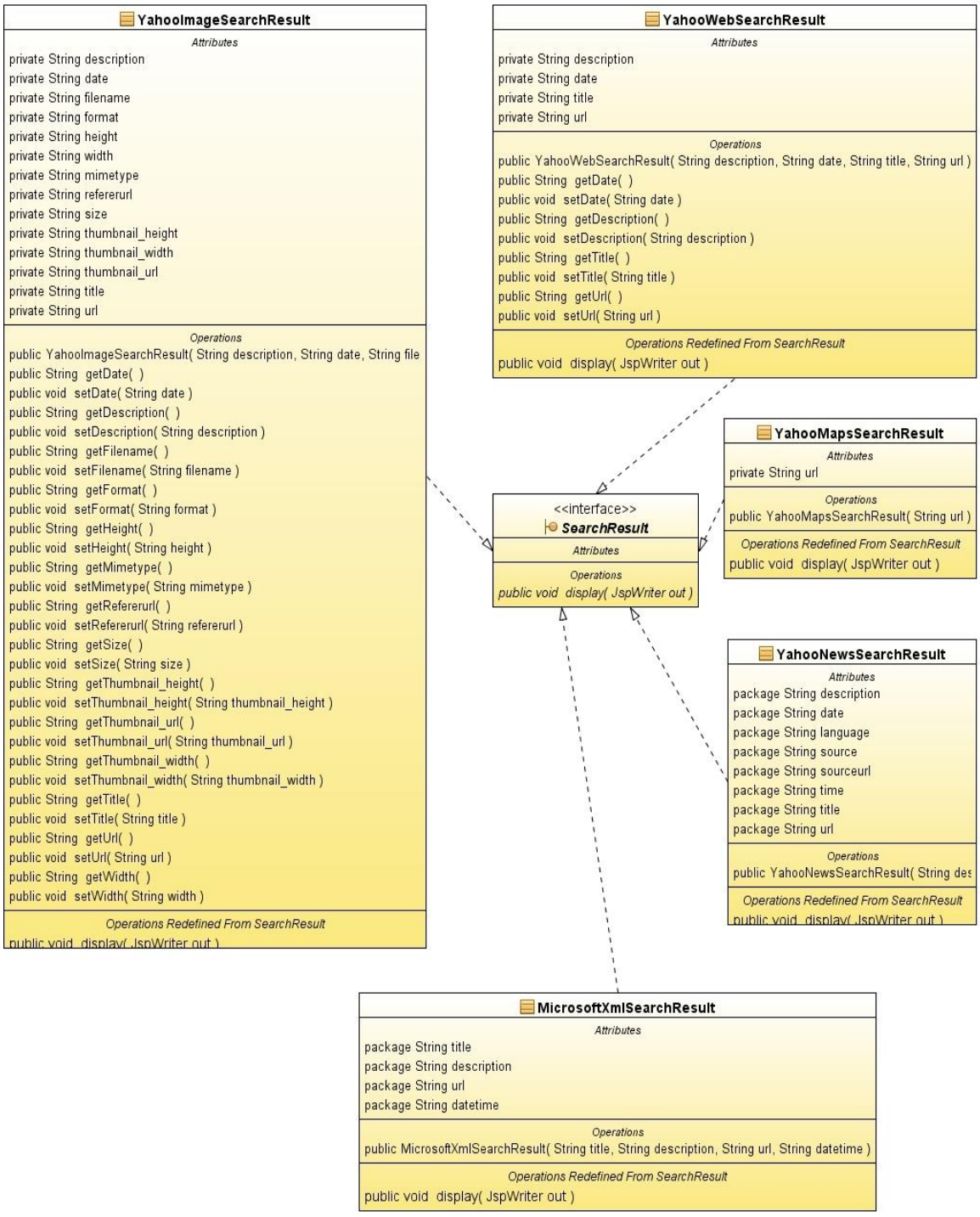
A8 - Diagrama de use case a interfeţei web a framework-ului



A9 – Diagrama de clase a parserului de rezultate din cadrul aplicaţiei web demonstrative



A10 – Diagrama de clase a rezultatelor web folosite de aplicaţia demonstrativă



A11 – Diagramă de activităţi a aplicaţiei web demonstrative