Debreceni Egyetem
Informatikai Kar

Spring Web szolgáltatások

Témavezető
Dr. Adamkó Attila
Egyetemi tanársegéd

Készítette
Nagy Ignác
Programtervező informatikus

Debrecen
2009

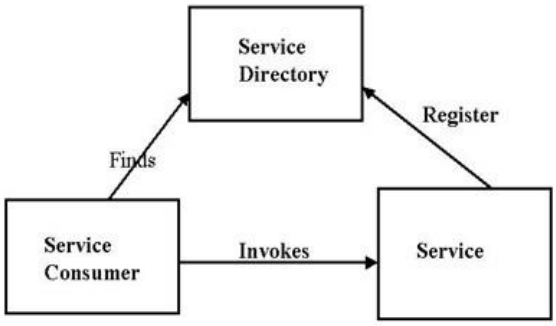
Bevezetés

Az elmúlt időszakot az jellemezte, hogy a vállalatok az egyes üzletágak bizonyos feladatainak ellátására vásároltak rendszerek, amelyek aztán IT-részleg többnyire csak üzemeltetett. Így alakultak ki az elszigetelt rendszerek. Ezek a rendszerek különböző platformokra, különböző technológiával készültek, és adott üzleti területeket csak a "bedrótozott" folyamat szerint támogatnak, nincsenek felkészítve a folyamatok módosítására és más üzleti folyamatokhoz kapcsolására.

A szükségessé váló alkalmazásintegrációt többnyire pont-pont alapon, sokféle egyedi technikával oldották meg, tovább növelve a komplexitást. De nemcsak a funkciók és folyamatok vannak elszigetelve, hanem az adatok is, amelyek heterogén formájúak és inkonzisztensek, ami rendkívül megnehezíti azok egységes vállalati szintű kezelését.

Mivel az üzleti igények általánosak és sürgetők, az IT-iparág kialakította válaszát: a SOA (Service Oriented Architecture), azaz a Szolgáltatás Alapú Architektúra, az a megközelítés, ami a fenti elvárásokat biztosítani képes, és az informatikai ipar fejlődésének hajtóerejévé vált. A SOA nyílt, szabványos komponens technológia, amelynek építőelemei a szolgáltatások. A szolgáltatások önállóan is működőképesek, platform- és eszköz függetlenek (tetszőleges technológiával készülhetnek), szabványos, jól definiált interface-el rendelkeznek, és szabványos adatcsere- és kommunikációs protokollokkal érhetők el az elosztott hálózatokban.

Technikai megközelítésben a SOA egy architektúra és technológiai modell, amely lehetővé teszi, hogy különböző alkalmazásokban (alkalmazáscsomagok, egyedi fejlesztésű alkalmazások stb.) megvalósított különálló üzleti és technikai funkciókból, szolgáltatásokból üzleti folyamatokat megvalósító alkalmazásokat állítsunk össze.



A képen a SOA felépítése látható. 3 részből épül fel:

1. szolgáltatás (Service)
2. szolgáltatás könyvtár (Service Directory)
3. szolgáltatás használó (Service Consumer)

1. A SOA felépítése

A szolgáltatást regisztrálni (Register) kell a szolgáltatás könyvtárban. A szolgáltatás felhasználó megtalálja (Finds) a szolgáltatást a szolgáltatás könyvtárban, majd igénybe veszi (Invokes) a szolgáltatást.

A SOA egyik implementációja a webszolgáltatás. Napjainkban a web szolgáltatások egyre nagyobb szerepet kapnak. Népszerűsége köszönhető a nyílt szabványok használatának, például XML, SOAP, újrafelhasználhatóságának.

Spring WS a Spring közösség készítette. A céljuk rugalmas, könnyen manipulálható webszolgáltatás létrehozása. A Spring WS (WebService) a Spring keretrendszeren alapul. A Spring keretrendszer egy nyílt forráskódú keretrendszer, ami a Java- hoz készült, de már elérhető .Net-hez is.

A szakdolgozat célja a webszolgáltatások bemutatása, és hogy hogyan készíthetünk web szolgáltatást a Spring WS segítségével.

A szakdolgozat további részében megismerhetjük a webszolgáltatások főbb alapelemeit, majd megnézzük, hogyan épül fel ezekből a webszolgáltatás, majd ezek után a Spring WS bemutatása következik, majd végül alkalmazása a gyakorlatban

XML

A web szolgáltatások kommunikációja XML (extensible markup language) alapú, így mindenképp egy rövid XML ismertetéssel kell kezdeni a web szolgáltatások bemutatása előtt. Az XML-ről egy külön szakdolgozatot lehetne írni, ez a bemutatás csak röviden és vázlatosan foglalkozik vele.
Az XML egy általános célú leíró nyelv. Az XML szöveges formátumú, értelmezhető mind a gép mind az ember számára. Egy XML dokumentum az XML fejlécből, elemekből és tulajdonságokból áll.

Példa XML:

```
<?xml version="1.0" ?>
<persons>
<person username="EN">
<name>Ernő</name>
<family-name>Nemecsek</family-name>
</person>
</persons>
```

Az első sor az XML fejéc, melyben az XML verzióját láthatjuk. A 2. sorban a personselem nyitótagja van, a 7. sorban kerül lezárásra. A 3. sorban a persons elem gyerek eleme a person nyitótajga áll, amelynek van username tulajdonsága, melynek erteke „EN”. A person elemnek további gyerek elemi vannak, elsől a name, melyek szöveges értéke az Ernő, az elem lezárása ugyanebben a sorban találéható. Ehhez hasonlóan található z 5. sorban a family-name elem a Nemecsek szöveges értékkel. A 6. sorban kerül lezárásra a person elem.

XML névterek

Az XML dokumentum egyik, fontos tulajdonsága, hogy új dokumentumok építőköveként is felhasználható. Ez az XML újrahasznosításának legegyszerűbb módja. Az XML elem neve előtt elhelyezzük egy egyedi azonosítót, névteret. Az XML névterek azonosítására egységes erőforrás azonosítókat (URI) használunk. A használni kívánt névtér megjelölése az XML dokumentumban 2 lépésből áll:

1. A névtér azonosítójához hozzárendelünk egy előtagot, amely kizárólag az XML elemnevekben használható karakterekből áll.
2. A minősített neveket az előtagból, egy kettőspontból és az elem nevéből állítjuk össze.

A névtereket használat előtt definiálni kell, azon elem nyitó részében ahol használni szeretnénk. Az `xmlns` kulcsszó után adjuk meg a névtér nevét (névtér előtag) valamint a névteret, amely valójában azonosítja a névteret. A névtér előtagot használjuk a dokumentumban, mint névtér azonosító, amelynek megadhatunk tetszés szerinti érvényes nevet. Az egyes névtereket a nyitóelemek, záró elemek illetve tulajdonságnevek előtt egyaránt megadhatjuk kettősponttal elválasztva.

XML helyesség

Egy XML dokumentumnak helyességéhez meg kell felelnie a jól formázottságnak és az érvényességnek. A helyesen formázott XML dokumentumnak szintaktikai szabályoknak kell megfelelnie, következzen néhány alapszabály:

- Egyetlen gyökér elem lehet egy dokumentumban. Azonban az XML deklaráció, feldolgozó utasítások és megjegyzések megelőzhetik a gyökér elemet.
- A nem üres elemeket mind nyitó, mind záró tag-eknek kell határolni.
- Az üres elemek megjelölhetők üres elem (önlezáró) tag-gel
- Minden attribútum érték idézőjelek között van, vagy szimpla(') vagy dupla(") idézőjelek között. Szimpla idézőjel szimpla idézőjelet, dupla idézőjel dupla idézőjelet zár.
- A tag-ek egymásba ágyazhatók, de nem lehetnek átfedők. Mindegyik nem gyökér elemet másik elemnek kell magában foglalnia.
- A dokumentum megfelel a karakterkészlet definíciónak. A karakterkészlet általában az XML deklarációban van meghatározva, de a szállító protokoll (például HTTP) is meghatározhatja. Ha nincs karakterkészlet definiálva, Unicode karakterkészletet feltételez, amit a Unicode bájtsorrend jel határoz meg. Ha a jel nem található, UTF-8 az alapértelmezett.
- Az elem nevek kisbetű-nagybetű érzékenyek.

XML séma

Egy XML séma az XML dokumentum tartalmának és szerkezetének leírása, jellemzően az XML megkötésein túli korlátozásokat tartalmaz. Technikailag a séma meta adatok gyűjteménye, amely séma elemekből épül fel. A XML séma egy igen hatékony, ugyanakkor meglehetősen összetett eszköz. Hatékony, mert kifejező és pontos leírást ad az XML dokumentum tartalmáról. Az XML sémában, XSD-ben (XML Schema Definition) definiálhatunk elemeket, jellemzőket, adat típusokat. Az egyszerű adattípusok között megtalálhatóak a szokásosak, például string, integer, date. Továbbá megadhatunk saját típusokat. Ekkor meg kell mondanunk egy alaptípust, és a saját típusokhoz tartozó megszorításokat. Ilyen megszorítás meg lehet adni például reguláris kifejezésekkel.

Megadhatunk összetett adattípusokat is, amiben olyan elemeket definiálhatunk, amelyek tulajdonságokkal és beágyazott gyerekekkel is rendelkezhetnek.

XML feldolgozás

A programok számára az XML egy szöveges dokumentum, ezért ezt fel kell dolgoznunk, hogy adatokat nyerjünk ki belőle. Az XML-dokumentumok adatközpontú elemzésére két szabványos API terjedt el:

- DOM-alapú (Document Object Modell alapú API) elemzők,
- SAX-alapú (Simple API for XML) elemzők

A DOM felépíti az XML-adatfolyamnak megfelelő objektumokból álló fát, melynek elemeit ezután közvetlenül el lehet érni. Ez azt jelenti, hogy a memóriában létrejött objektumok tagfüggvényei és adattagjai elérhetők.

A SAX nem épít fel DOM szerkezetet, ugyanis itt az elemzés során mindig csak az aktuális elemet látjuk. A SAX emiatt nem teszi lehetővé az XML elemek közvetlen elérését, azokat csak sorosan olvassa fel, aminek következtében események generálódnak. Az elemző alkalmazás ezekre az eseményekre határozza meg az eseménykezelő metódusokat.

XPath

Az XML Path Language (XPath) az XML-dokumentumok csomópontjaiban található információ keresésére és beolvasására szolgáló lekérdezési nyelv. Az XPath alapú lekérdezések kifejezés formátumúak. Ezekkel a kifejezésekkel történik az XML dokumentum különböző részeinek címzése, a karakterláncok, számok és logikai értékek feldolgozása, valamint a dokumentum egyes csomópontkészleteinek egyeztetése. Az XPath az XML-dokumentumot különböző csomópontokból álló faként kezeli. Az XPath kifejezései típusuk, nevük, értékeik, valamint a csomópontoknak a dokumentumon belül egymáshoz viszonyított kapcsolata alapján azonosítják az XML-dokumentumban lévő csomópontokat.

Marshalling

Marhslling: a példányosított osztályokat XML dokumentumba menthetjük le.
UnMarshalling: XML dokumentumból hozhatunk létre objektumokat.
Nem tartozik közvetlenül az XML témakörébe, de a továbbiakban használni fogjuk ezeket az elnevezéseket.

SOAP

Nem beszélhetünk webszolgáltatásokról, amíg nem ismerjük meg a SOAP-ot, ami a webszolgáltatások egyik alapköve. A SOAP eredetileg a Simple Object Access Protocol (egyszerű objektumelérési protokoll) rövidítése volt, de az 1.1-es változattól önállósult. A SOAP egy olyan protokoll, amely alkalmas változatos környezetbeli adatcserék lebonyolítására. A SOAP egy XML alapú, üzenetváltásra kitalált protokoll, amelyben az üzenetek továbbításra kerülnek. A SOAP alatt levő szállítási protokoll az esetek túlnyomó többségében HTTP, de lehetőség van SMTP-t is használni. Ennek megvan az az előnye, hogy nincsenek hatással rá a tűzfalak. A tipikus használati eset az RPC (Remote Procedure Call), amely szerint egy gép üzenetet küld a másiknak, amely azonnal válaszol is rá. A SOAP-nak sajnos megvan az a hátránya, hogy az így formázott üzenetek nagyon hosszúak, az átvitel így sokáig eltarthat, és nagyobb a sávszélesség igénye is.

SOAP boríték

A SOAP felépítésének legfontosabb része a boríték keretrendszer, ezt mindössze néhány XML-ből álló rendszer írja le. A SOAP boríték keretrendszer határozza meg, hogyan lehet azonosítani az üzenetben lévő adatokat, kinek kell feldolgoznia azokat és mely adatok hagyhatóak el, illetve melyek azok, amelyeket kötelező megadni. A SOAP üzenet egy kötelező borítékból (envelop) áll, amely tetszőleges számú elhagyható fejlécet és egy kötelező törzset. A SOAP üzenet XML dokumentumának gyökerében egy *Envelop* elem áll. A SOAP boríték tartalmazhat egy elhagyható *Header* elemet, és minden esetben tartalmaz egy *Body* elemet.

SOAP Fejléc

A fejlécek adják a SOAP bővíthetőségének alapját. Számos területen találkozhatunk fejlécekkel: hitelesítés és engedélyezés, tranzakció kezelés, nyomon követés, stb. A fejlécek segítségével bármilyen, a kérés végrehajtásától független adatokat eljuttathatunk a feldolgozóhoz. A SOAP üzenet tetszőleges számú fejlécet tartalmazhat, minden fejléc a SOAP Header elem gyereke. Egy fejlécben tetszőleges XML szerepelhet.

SOAP Törzs

A SOAP üzenet lényegét a Body elem tartalmazza, ami szintén tetszőleges XML lehet. Itt találhatóak a mindenképp feldolgozandó elemek. Ez az üzenet hasznos része.

Szintaktikai szabályok

- a SOAP üzenetet az XML felhasználásával kell kódolni,
- a SOAP üzenetnek a SOAP Envelope névteret kell használnia,
- a SOAP üzenetnek a SOAP Encoding névteret kell használnia,

- a SOAP üzenetnek tilos DTD hivatkozást tartalmaznia,
- a SOAP üzenetnek tilos XML feldolgozási utasításokat (processing instructions) tartalmaznia.

Példa SOAP üzenet

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

A fenti példa, egy egyszerű értesítő üzenetet ír le. Tartalmaz egy Header blokkot alertcontrol lokális névvel, és tartalmazza a kötelező Body elemet alert lokális névvel. A Fejlécben található egy lejárati dátum és egy prioritás információ. A törzsben található maga az értesítő üzenet szövege.

WSDL

A WSDL (Web Services Description Language), a webszolgáltatások leírására szolgáló dokumentum. A WSDL a webszolgáltatás nyilvános felületét írja le. Ez egy XML-alapú szolgáltatás-leírás a webszolgáltatással történő kommunikációról. A leírásban szerepelnek a protokollkötések, az adatok típusai és az üzenetek, amelyek az adott webszolgáltatások használatához szükségesek. Továbbá az üzenet továbbításához használt protokollok és a szolgáltatás elérhetősége. A leggyakrabban használt protokoll az üzenetek továbbítására a HTTP.

Az XML dokumentum felépítése:

- A WSDL dokumentumon belül, a <types> elem foglalja magában az üzenetküldés során küldött és fogadott adatok típusainak definícióját.
- A <message> elemek foglalják magukban az üzenetek leírásait.
- A WSDL dokumentumon belül, a <portType> elem foglalja magában a szolgáltatás által nyújtott műveleteket, és a műveletek végrehajtásához szükséges üzeneteket.
- A<binding> címke alatti definíciók határozzák meg a protokoll és az adatátvitel tulajdonságát.

- A WSDL dokumentumon belül a <service> címke alatt definiáljuk a szolgáltatás elérhetőségét.

WSDL példa

Egy WSDL dokumentum egyszerűsített részlete:

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
```

Ebben a példában a <portType> elem a port nevét „glossaryTerms”-ként, és a művelet nevét „getTerm”-ként határozza meg.
A „getTerm” műveletnek van egy „getTermRequest” nevű input üzenete és egy „getTermresponse” nevű output üzenete.
A <message> elemek meghatározzák mindenegyes üzenet részeit és a hozzájuk tartozó adattípusokat.
A hagyományos programozással összehasonlítva a glossaryTerms egy függvény könyvtár, a „getTerm” egy függvény, melynek input paramétere a „getTermRequest”és visszatérő értéke a „getTermResponse”.

WSDL portok

A WSDL port leírja a web szolgáltatás által közzétett interfészeket (megengedett műveleteket). A <portType> a legfontosabb WSDL elem.
Ez az elem meghatároz egy web szolgáltatást, a végrehajtható műveleteket, és a befoglalt műveleteket. A port meghatározza a web szolgáltatás kapcsolódási pontját. Ez a hagyományos programozási nyelvek függvény könyvtárához (vagy egy modulhoz, vagy egy osztályhoz) hasonlítható. Minden művelet pedig a hagyományos programozási nyelv egy függvényéhez hasonlítható.

Művelet típusok

A kérés-válasz típus (request-response) a leggyakoribb művelet típus, de a WSDL négy típust határoz meg:

- one-way (egyirányú): a művelet fogadhat üzenetet, de nem fog küldeni választ,
- request-response (kérés-válasz): a művelet fogadhat kérést, és választ fog küldeni,

- solicit-response (kérelem-válasz): a művelet küldhet egy kérést és választ fog várni,
- notification (közlés): a művelet üzenetet küldhet, de nem fog várni válaszra.

WSDL kötések

A WSDL kötések meghatározzák a web szolgáltatáshoz az üzenet formátumát és a protokoll jellemzőit. Kötés a SOAP-hoz, egy request-response művelet példa:

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
<binding type="glossaryTerms" name="b1">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation
     soapAction="http://example.com/getTerm"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

A binding elemnek két attribútuma van, a name és a type attribútum.
A name attribútum (tetszés szerinti név használható) meghatározza a kötés nevét, és a type attribútum a kötés portjára mutat, ebben az esetben a „glossaryTerms” portra. A soap:binding elemnek két attribútuma van, a style és a transport attribútum. A style attribútum „rpc” vagy „document” lehet. Ebben az esetben a document-et használjuk. A transport attribútum meghatározza a használandó SOAP protokollt. Ebben az esetben a HTTP-t használjuk. Az operation elem meghatározza a port által közzétett összes műveletet. Mindenegyes művelethez meg kell határozni a megfelelő SOAP akciót. Ugyancsak ki kell jelölni az input és az output kódolását. Ebben az esetben a „literal”-t használjuk.

UDDI

Az UDDI a Universal Description, Discovery, and Integration, azaz *univerzális leírás, felfedezés és integrálás* rövidítése – egy platform független, XML-alapú nyilvántartó rendszeré. Az UDDI nyílt ipari kezdeményezés (az OASIS szponzorálja), ami lehetővé teszi vállalatok számára, hogy megtalálják egymást és meghatározzák segítségével, hogy miként kommunikáljanak az interneten.

Az UDDI üzleti regisztráció három komponensből áll:

- Fehér oldalak *(White Pages)* – cím, kapcsolat, ismert azonosítók
- Sárga oldalak *(Yellow Pages)* – ipari osztályozás szabványos módszertan alapján
- Zöld oldalak *(Green Pages)* – technikai információk a vállalat által nyújtott szolgáltatásokról

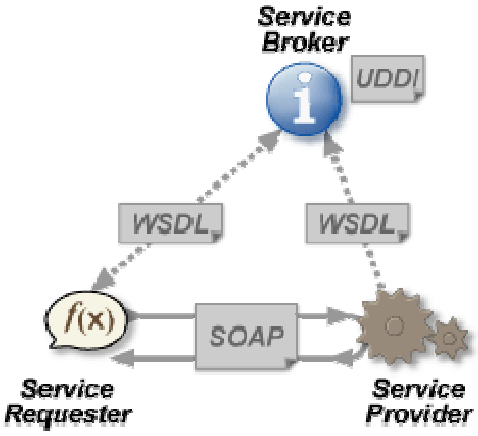
Az UDDI a webszolgáltatások legalapvetőbb komponenseinek egyike. Úgy tervezték, hogy SOAP üzenetekkel legyen lekérdezhető, valamint hogy hozzáférést biztosítson a WSDL dokumentumokhoz, amelyek a könyvtárban felsorolt webszolgáltatások használatához szükséges protokollkötéseket és üzenetformátumokat írják le.

A UDDI regiszter (tárház) a vállalkozások és az általuk támogatott szolgáltatások leírásának program által elérhető leírásait tartalmazza. Tartalmazhat olyan ipar specifikus specifikációkra való hivatkozást is, melyeket a Webszolgáltatások támogathatnak, fogalmi definíciókat (a vállalkozások és szolgáltatások lényegi kategorizálására), és azonosítási rendszereket (a vállalkozások lényegi azonosítására). A UDDI programozási modellt és sémát biztosít, ami meghatározza a tárházzal való kommunikáció szabályait. A UDDI specifikáció összes API-ja XML-ben definiált, SOAP borítékba (envelope) csomagolt és HTTP-vel küldődik.

Webszolgáltatások

Áttekintettük a webszolgáltatások főbb elemeit, ezek felhasználásával, már definiálhatjuk. A web szolgáltatások definíciója a W3C szerint:
„A *Web szolgáltatás olyan szoftver rendszer, melyet arra terveztek, hogy támogassa az együttműködőképes gép-gép közötti egymásra hatást egy hálózaton. Rendelkezik egy géppel feldolgozható formátumban (WSDL) leírt interfésszel. Más rendszerek a Web szolgáltatással SOAP üzenetekkel működnek együtt oly módon, ahogy azt a leírása megszabja, az üzeneteket szokásosan a HTTP felhasználásával, XML szerializációval és egyéb Web-bel kapcsolatos szabványok alapján továbbítja.”*

Webszolgáltatás bármilyen szolgáltatás lehet, amely interneten keresztül, szabványos XML alapú üzenetküldő rendszert használ és egyetlen operációs rendszertől vagy programnyelvtől sem függ. A webszolgáltatás leírható a WSDL segítségével, egy szolgáltatáshoz tartozik egy interface, és létezik egy ember számára is olvasható leírása. A webszolgáltatás felkutatható: a létrehozott szolgáltatás publikálható, ez az UDDI. A webszolgáltatás szerkezeti felépítése:



1. A webszolgáltatások felépítése

- Szolgáltató (Service Provider): az a platform, amely a szolgáltatáshoz való hozzáférést lehetővé teszi.
- Szolgáltatás igénylő (Service Requester): az az alkalmazás amely, keres, meghív vagy inicializál egy kapcsolatot a szolgáltatóhoz.
- Szolgáltatásjegyzék (Service Broker): a szolgáltatás leírások kereshető jegyzéke, a szolgáltató itt teheti közzé az általa biztosított szolgáltatásokat és saját adatait, az igénylők itt kereshetnek számukra szükséges szolgáltatásokat.

Ezzel kapcsolatban 3 műveletet definiálhatunk:

- Közzététel:ez a művelet a szolgáltatás bejegyzésének vagy meghirdetésének folyamata. A közétellel megegyezés jön létre a szolgáltató és a szolgáltatásjegyzék között.
- Keresés: ez a művelet a közzétetel logikai párja. A keresés során megegyezés jön létre a szolgáltatásjegyzék és az igénylő között. Az igénylő keresési feltételei alapján megkapja a megfelelő szolgáltatások leírását.
- Összekapcsolás: z a művelet ügyfél-kiszolgáló kapcsolatot létesít a szolgáltatás igénylője és a szolgáltató között. Ez ehet statikus – fejlesztési időben meghatározott, vagy lehet dinamikus – futásidőben meghatározott.

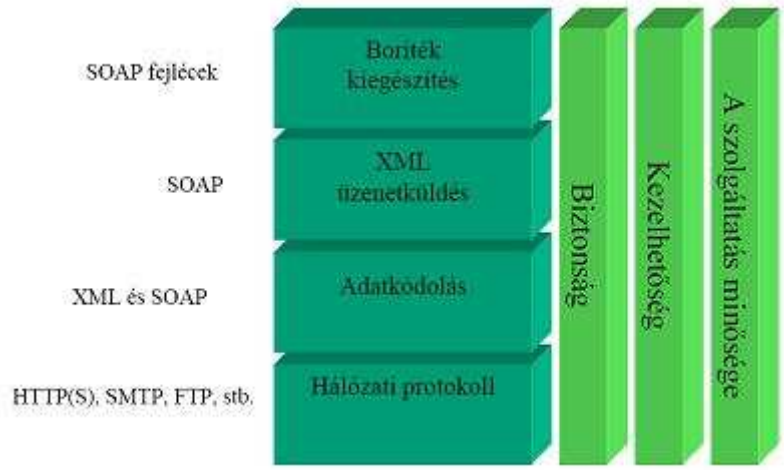
Együttműködési vermek

Ahhoz, hogy pontosan megértsük, hogy hogyan kapcsolódnak egymáshoz a dolgozat első részében ismertetet webszolgáltatás elemek, ahhoz ismernünk kell a webszolgáltatások együttműködési vermeit. A webszolgáltatásokat 3 veremre oszthatjuk fel:

- Vezeték verem
- Leírás verem
- Feltáró verem

Egy webszolgáltatásnak nem kell feltétlenül az összes vermet implementálnia.

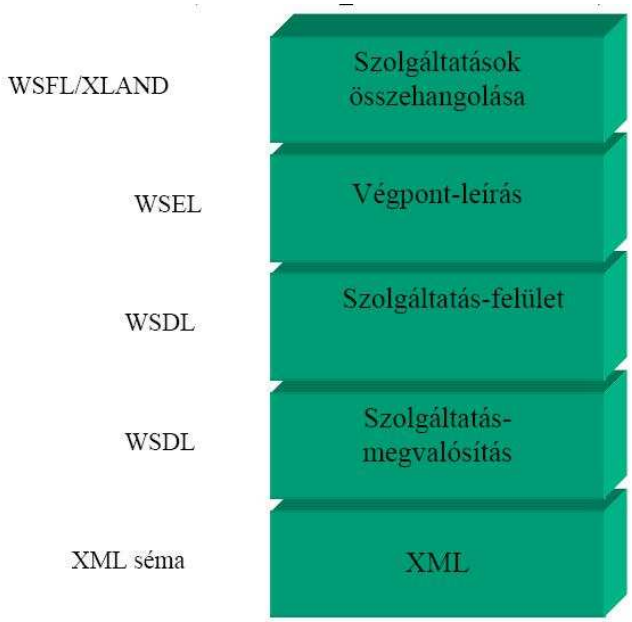
A vezeték verem



1. A vezeték verem

A vezeték verem a az igénylő és a szolgáltató közötti üzenetek továbbításában részt vevő megoldásokat összegzi. A vezeték verem alapja a hálózati protokoll. A webszolgáltatások az adatok kódolására az XML-t használja. Az adatokat XML séma segítségével írják le. A hálózati protokoll és az adatkódoló rétegek felett jelenik meg az XML üzenetküldés. Az XML üzenetküldéshez a webszolgáltatások a SOAP-ot használják. A SOAP borítékoló rétege felett találjuk meg a SOAP fejléceket.

A leírás verem



1. A leírás verem

A szolgáltatásközpontú renszerek legfontosabb eleme amaga a szolgáltatásleírás. A webszolgáltatások esetében a leírásunk alapja az XML. A leírás adattípusai XML sémán

alapulnak, és a veremben lévő megoldások mindegyikét XML írja le. A verem következő kér szintjét, azaz a szolgáltatás felületét és megvalósítását a WSDL segítségével írhatjuk le. A végpont-leírás szerepe, hogy a WSDL felett adatokat szolgáltasson az adott megvalósítás jellemzőiről. A szolgáltatás-leírás verem tetején található a szolgáltatás összehangolás rétege, ami leírhatja, hogy hogyan lehet a különféle webszolgáltatások egyesítésével még kifinomultabb webszolgáltatásokat előállítani.

A feltáró verem

1. A feltáró verem

A verem alján az egyszerű vizsgálódás szintjét látjuk. A vizsgálat, olyan módszer, amellyel felkutathatunk egy szolgáltatás leírást. A feltárás módja még nincs szabványosítva, az IBM az ADS, a Microsoft pedig a DISCO megoldás fejlesztésén dolgozik. A webszolgáltatás jegyzékek (címtárak, könyvtárak) javasolt megvalósítási módja az UDDI szabványt használja fel.)

Spring WS

A Spring WS célja egy dokumentum vezérelt WS létrehozása. Célja egy olyan „contact first” SOAP szolgáltatás kidolgozása, amellyel egy rugalmas, XML leírásokkal könnyen manipulálható webszolgáltatást hozva létre.
A főbb tulajdonságai:

- Erőteljes leképzések A bejövő XML kéréseket bármely object-nek átadhatjuk,az üzenettől, vagy a SOAP fejléctőlvagy egy XPath kifejezéstől függően

- XML API támogatás: A beérkező XML üzeneteket nem csak a standard JAXP API-kal (például: DOM, SAX, SAM StAX) kezelhetjük le, hanem például JDOM, dom4j, XOM vagy marshalling technikákkal is.

- Rugalmas XML marshalling: Az Object/XML Mapping (OXM) modul támogatja a JAXB 1 és 2-t, Castor-t, XMLBeans-ot, JBX-et és XStream-ot. Mivel ez egy különálló modul, ezért nem csak webszolgáltatásoknál használhatjuk fel.

- A Spring szakértelem újrahasznosítása: A Spring WS a Spring alkalmazások felépítését használja az összes konfigurálásban, amellyel az eddigi Spring fejlesztők gyorsan elsajátíthatják a Spring WS használatát. A Spring WS architektúrája sokban hasonlít a Spring-MVC-hez is.

- Támogatja WS Security-t: A WS_Security lehetővé teszi a SOAP üzenetek aláírását, titkosítását, és azok hitelesítését. Beépítve található az Acegi Security, így felhasználhatjuk a meglévő Acegi konfigurációnkat.

- Maven által támogatott: így hatékonyan újrahasználhatjuk a Maven alapú projectjeinkben

- Apache licensz: magabiztosan használhatjuk a Spring WS-t a projectjeinkben.

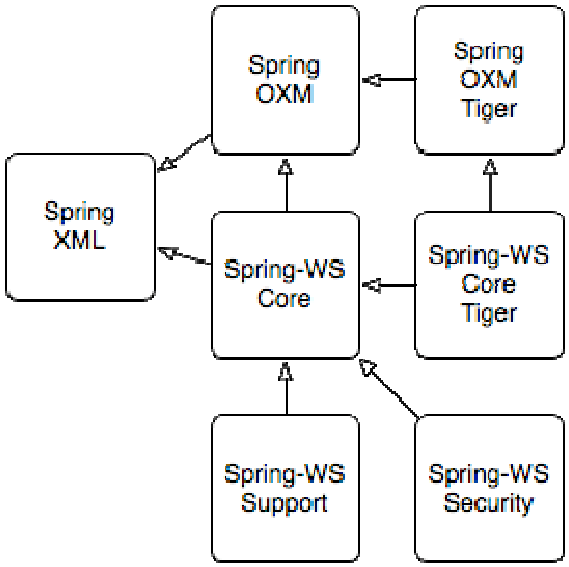
Futtatási környezet

A Spring WS a standard Java 1.3-as Runtime Environment-tel fut, de támogatja az 5.0-asat is, de az ehhez a verzióhoz készült Java típusok, külön csomagban találhatóak. A „tiger" végződésű csomagok tartoznak az 5.0-ás verzióhoz, szintén ezekben a csomagokban a biztonsági modulok, mert Java 5.0 kell a használatukhoz.

A Spring WS több modulból áll:

- Az XML modul (spring-xml.jar) különböző XML-t támogató osztályokból áll. Ez a modul a leginkább független a SpringWS keretrendszertől, ez nem tartozik a webszolgáltatások fejlesztéséhez.
- A Core csomag (spring-ws-core.jar és spring-ws-core-tiger.jar) a központi része a Spring webszolgáltatás funkcióinak. Ebben található a központi WebServiceMessage és SoapMessage interfész, a szerver oldali keretrendszer erős üzenet elosztással, és a különböző támogató osztályok webszolgáltatások végpontjainak implementálásához, és a kliens oldali WebServiceTemplate.
- A Security csomag (spring-ws-security.jar) egy olyan webszolgáltatás Security implementálást tartalmaz, amely magába foglalja a core web szolgáltatás csomagot. Ez lehetővé teszi az aláírásokat, a titkosítását a SOAP üzeneteknek. Szintén ez teszi lehetővé a meglévő Acegi secrity használatát.
- Az OXM csomag (spring-oxm.jar és spring-oxm-tiger.jar) népszerű XML marshalling API-k integrációját tartamlazza, többek között a JAXB 1 és 2-t.Az OXM csomag használatával könnyen felhasználhatjuk a kedvenc marshalling technikánkat.

Az alábbi ábra illusztrálja a Spring WS moduljait, és azok függőségeit. A nyilak jelentik a függőségeket, például: a Spring WS Core függ a Spring-XML és a Spring-OXML-től.



1. A Spring WS moduljai

Contact-first

A webszolgáltatások fejlesztésénél két fejlesztői módszer közül választhatunk: Contact First és a Contact Last. A Spring WS a contact-first megvalósítást támogatja.
A contact-last esetén a Java kód készítésével kezdünk, és ebből generáltatjuk le a WSDL-t. A contact-first metódus esetén a WSDL készítéssel kezdünk, és Java-t használunk a szerződés megvalósításához. A contact-first megvalósításának több előnye is van:

- Ha a Java kódból indulunk ki, akkor nem biztos, hogy az ebből készült XML dokumentumunk kompatibilis lesz más programozási nyelvekkel, míg ha az XML dokumentummal kezdünk, akkor nyelv független adattípusokat hozhatunk létre.
- Az egymásra hivatkozó objektumokból akár egy végtelen körkörös XML dokumentum is létrejöhet, de ha XML-lel kezdünk, akkor egy szimpla referencia tulajdonsággal kezelhetjük ezt.
- Ha változtatunk a Java kódon, akkor változtathatunk a WSDL állományon is, ezzel változhat a szolgáltatásunk leírása és így változtatni kel a klienseken is. Contact-first esetén a Java kód változtatása nincs hatással a WSDL állományra.
- Contact-last esetén létrejöhetnek nagyon nagy XML üzenetek, mert nem tudjuk, hogy a Java osztályok milyen mélyen épülnek be, ha mi addjuk meg az XML dokumentum felépítését, akkor minimalizálhatjuk a dokumentumunkat.

Megosztott komponensek

Ebben a fejezetben azokat a komponenseket ismerhetjük meg, amik a Spring WS kliens és szerver oldali fejlesztésében egyaránt szerepet játszanak. Ezek az osztályok és interface-k az építőelemei a Spring WS-nek.

WebServiceMessage

Az egyik legfontosabb interface-e a Spring WS-nek a WebServiceMessage. Ez az interface olyan metódusokat tartalmaz, amelyek hozzáférnek az üzenetek hasznos részéhez, javax.xml.transform.Source vagy a javax.xml.transform.Result formájában. A Source reprezentálja az XMl inputot, a Result pedig az XML outputot.

SoapMessage

A SoapMessage a WebServiceMessage-ből származik. SOAP specifikus metódusokat tartalmaz, például a SOAP fejlécének vagy a SOAP hibajegyzéknek a lekérését. Általában nem kell ezt használnunk, hiszen a WebService Message tartalmazza a SOAP törzsének elérését (getPayLoadSource() és getPayLoadResult). Ha szeretnénk SOAP mellékletet, vagy új fejléc hozzáadása esetén használjuk.

Message Factories

A üzenet implementációt a WebServiceMessageFactory készíti. Ezzel készíthetünk üres üzenetet, vagy olvashatunk üzenetet egy input stream alapján. Két konkrét implementációja van, az egyik SAAJ (SOAP with Attachment API for Java) alapú, ez a SaajSoamMessageFactory. A másik az Axis 2 AXIOM-án alapul, ez az AxiomSoapMessageFactory. A SAAJ az DOM-on alapul, ezért a SOAP üzenet a memóriában tárolódik, ezért nagyobb SOAP üzenetek esetén érdemesebb az AXIOM-on alapulót választani. Mindkettő rendelkezik soapVersion tulajdonsággal, amellyel megmondhatjuk, hogy melyik SOAP verzióval kompatibilis.

MessageContext

A Spring WS-ben az üzenetváltásért a MessageContext a felelős, amely képes a kérelem üzenetek fogasára és a válasz üzenetek készítésére. A kliens oldalon ez a WebServiceTemplate. A szerver oldalon az üzenetek a transport függő input stream-ből olvassa, például http-ben az olvasást a HttpServletRequest, a válaszolást a HttpServletResponse végzi.

TransportContext

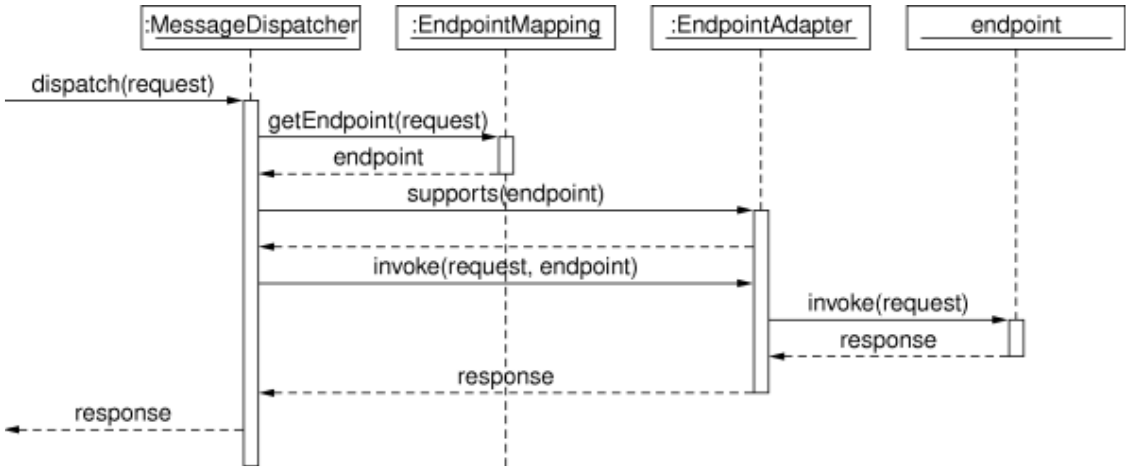
A Spring WS nem támogatja végpontok elérését http kéréseken keresztül, de néha szükség lehet az átviteli réteghez való hozzáféréshez.
A TransportContext lehetővé teszi a WebServiceConnection hozzáférést, ami általában egy HttpServletConnection a szerver oldalon, a kliens oldalon HttpUrlConnection vagy CommonHttpConnection. Például hozzáférhetünk az aktuális kérelem IP címéhez a szerver oldalon. Ritkán van szükség az átviteli protokollhoz való ilyen alacsony hozzáférés.

Szerver oldal

A Spring WS szerver oldalának támogatása a MessageDispatcher-re épül, ami a bejövő üzeneteket elküldi a végpontokba konfigurálható végpont kiosztással és válasz generálással. A legegyzerűbb végpont a PayLoadEndPoint, amiben csak az invoke metódust tartalmazza. Ezt az interface-t is implementálhatjuk, de vannak absztrakt implementációk, például az AbstractDomPayLoadEndPoint, AbstractSaxPayLoadEndpoint és az AbstractMarshallingPayLoadEndpoint.

MessageDispatcher

Rendkívül rugalmas, lehetővé teszi bármilyen osztály használatát végpontként, amíg ezt az osztályt beállíthatjuk a SpringIoC tárolóban. A MessageDispatcher a Spring DispatcherServlet-hez hasonlít.
Az alábbi ábra mutatja be a MessageDispatcher feldolgozását és továbbítását.



1. A MessageDispatcher működése

Amikor egy kérés beérkezik a MessageDispatcher-nek, akkor elkezdi a kérés feldolgozását. Az alábbi lista szemlélteti a teljes feldolgozását a kérelemnek:

1. Megfelelő végpont keresése a beállított EndpointMapping segítségével. Ha talált végpontot, akkor a végponthoz tartozó meghívási lánc végrehajtódik, hogy válasz üzenet generálódjon

2. A végponthoz tarozó megfelelő adaptert keres, majd kiküldi az adapternek, hogy végrehajtsa a végpontot.
3. Ha válasz üzenet készült, akkor azt elküldi. Ha nincs válasz (ami lehet például biztonsági okokból), akkor nem történik küldés.

MessageDispatcherServlet

A MessageDispatcherServlet egy servlet, ami a DispatcherServlet-ből származik és magába foglalja a MessageDispatcher-t. Ugyanazt a folyamatot követi, mint a MessageDispatcher. A MessgaeDispatcherServlet a webalkalmazás web.xml-jében állítható be. Azokat a kéréseket, amiket a MessageDispatcherServlet-tel akarunk lekezelni szintén a web.xml-ben kell beállítani. Web.xml példa:

```
<web-app>
    <servlet>
<servlet-name>spring-ws</servlet-name><servlet-
class>org.springframework.ws.transport.http.MessageDispatcherServlet</ser
vlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

A fenti példában az összes kérelmet a „spring-ws” MessageDispatcherServlet kezeli. Ez csak az első lépés a Spring WS beállításában, mert a Spring WS által használt különböző komponens bean-eket is be kell állítani. A webalkalmazás könyvtárának WEB-INF alkönyvtárában a spring-ws-servlet.xml ([servlet neve]-servlet.xml) állományban kell ezeket beállítani. Ez az állomány tartalmazza az összes Spring WS specifikus bean-t, például végpontokat.
A MessageDispatcherServlet automatikusan felismeri a konfig állományban definiált WsdlDefinition bean-t, és az így felismert bean-eket a WsdlDefinitionHandlerAdapter-en keresztül egyszerűen közzétehetjük a WSDL állományunkat.. Az alábbi példában az orders.wsdl leírása található:

```
<bean id="orders"
class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">
    <constructor-arg value="/WEB-INF/wsdl/Orders.wsdl"/>
</bean>
```

Ezután ez a wsdl állomány egy szokványos GET metódussal elérhető, a http:/[szerver]:[port]/[service neve]/orders.wsdl címről.

A kézzel írt WSDL állomány helyett használhatjuk a Spring WS-t automatikus WSDL állomány generálására XSD sémán keresztül. Az alábbi példában láthatjuk a konfigurációs file beállításait:

```
<bean id="orders"
class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
  <property name="schema" ref="schema"/>
  <property name="portTypeName" value="Orders"/>
  <property name="locationUri"
value="http://localhost:8080/ordersService/"/>
</bean>

<bean id="schema" class="org.springframework.xml.xsd.SimpleXsdSchema">
    <property name="xsd" value="/WEB-INF/xsd/Orders.xsd"/>
</bean>
```

A DefaultWsdl11Definition építi fel a WSDL állományt a megadott (a példában Orders.xsd) sémából. Végigiterál a sémában található összes element elemen és message-eket generál belőlük. Ezután WSDL operation-üket készít az összes olyan elemhez ami a kérés és válasz utótaggal rendelkezik. Ezek az utótagok alapbeállításon Request és Response, de módosíthatjuk is ezeket.
Ha több séma állománnyal is rendelkezünk, akkor megadhatunk több xsd file-t is, ezekből összefűzés után fogja a CommonXsdSchemaCollection feldolgozni és WSDL file-t elkészíteni. Példa:

```
<bean id="schemaCollection"
class="org.springframework.xml.xsd.commons.CommonsXsdSchemaCollection">
    <description>
        This bean wrap the messages.xsd (which imports types.xsd), and
inlines them as a one.
    </description>
    <property name="xsds">
        <list>
            <value>/WEB-INF/xsds/Orders.xsd</value>
            <value>/WEB-INF/xsds/Customers.xsd</value>
        </list>
    </property>
    <property name="inline" value="true"/>
</bean>
```

Szállítás

A Spring WS többféle átviteli protokollt támogat. A leggyakrabban használt a HTTP, de ezen kívül támogatja a JMS (Java Message Service), sőt még az e-mailen keresztüli átvitelt is.

- JMS: A Spring WS a WebServiceMessageListener-t biztosítja, hogy egy MessageListenerContainer-hez csatlakozzon. Ennek a funkciónak a működéséhez WebServiceMessgeFactory-ra és MessageDispatcher-re van szükség. Ennek

alternatívájaként a Spring WS a WebServiceMessageDrivenBean-t addja, egy EJB MessageDrivenBean-t.

- E-mail: Ezt a szolgáltatást a MailMessageReceiver osztály támogatja. Ez POP3-as és IMAP-es könyvtárak figyelésére képes, az e-maileket WebServiceMessage-é konvertálja, és a választ SMTP segítségével küldi el.

Végpontok

A végpontok a Spring WS szerver oldali támogatásának a központja. A végpontok lehetővé teszik az alkalmazás viselkedéséhez való hozzáférését, amit tipikusan egy üzleti szolgáltatás interface-e. A végpont értelmezi az XML kérést, ezt az inputot felhasználva tipikusan az üzleti szolgáltatásnak egy metódusát hívja meg. Ennek a metódusnak az eredménye a válasz üzenet. A Spring WS több különböző végponttal rendelkezik, és sokféleképpen képes az XML üzenet feldolgozására és a válasz előállítására.

A végpontok alapja az org.springframework.ws.server.endpoint.PayLoadEndpoint interface, amely mindössze egy metódusból áll: *Source invoke(Source request) throws Exception.*
A beérkező kérés hatására fut le ez a függvény, ha van Source visszatérő érték, akkor ebből lesz a válasz XML üzenet. Ebben az esetben az XML üzenet hasznos részéhez férhetünk csak hozzá. Ha a teljes XML kéréshez hozzá akarunk férni, akkor használhatjuk a MessageEndpoint-ot.
A Spring WS több végpont implementálást tartalmaz, amelyek közül a leggyakrabban XML üzenetek kezelésére használt a DOM. Ezt az AbstractDomPayloadEndpoint származtatásával érhetjük el. Használhatjuk az org.w3c.dom.Element és az ehhez kapcsolódó osztályokat a kérések kezelésére és a válasz megalkotásához. Ekkor csak az invokeInternal(Elemennt, Document) metódust kell felülírnunk, megírnunk a logikát, és ha szükség van, akkor visszatérni a válasz Element-tel. Ezen kívúl használhatunk más DOM API-t is, például az AbstractJDomPayloadEndpoint segítségével JDOM-t használhatunk, AbstractXomPayloadEndpoint segítségével XOM-ot használhatunk XML üzenetek kezelésére, ezekben az osztályokban is invokeInternal metódus található.

Ha nem szeretnénk közvetlenül az XML dokumentumot használni, akkor marshalling felhasználásával Java objektumokat készíthetünk a beérkező XML-ből. Erre a Spring WS az AbstractMarshallingPayloadEndpoint-ot alkalmazza. Ebben az esetben az invokeInternal(Object) metódust kell felülírnunk. Lehetséges Validator objektumokat használni, a Spring WS két lehetőséget kínál az AbstratValidatingMarshallingPayloadEndpoint-ot és az AbstractFaultCreatingValidatingMarshallingPayloadEndpoint-ot. Az utóbbi a legáltalánosabb módja a SOAP hibák előállítására validálási hibák után. Az onValidationErrors metódust kell felülírnunk a használatukhoz.

Végpont leképzés

Ez felelős a beérkező üzenetek megfelelő végponthoz való hozzárendelésért. A végpont leképzésben található a meghívási lánc, amelyben a végpontok és az esetleges végpont elfogásokat. A MessageDispatcher átadja kérést a végpont leképzésnek, ami megvizsgálja a

kérést és előáll egy megfelelő meghívási lánccal. Ezek után a MessageDispatcher meghívja ezt a láncban található végpontokat és végpont elfogókat.

A legtöbb végponti leképzés az AbstractEndpintMapping-ból származik, amelynek van interceptors tulajdonsága, amely egy lista a használni kívánt elfogókról. Továbbá van defaultEndpoint, ami egy végpont, ha a végpont leképező nem talál megfelelő végpontot, akkor ezt fogja használni.

A PayLoadRootQNameEndpointMapping a kérés gyökér elemének minősített nevét használja fel a megfelelő végpont kiválasztásához. Ez a minősített név a namespace URI és a local részből áll, amelynak egyedinek kell lennie a leképzésekben. Példa:

```
<beans>
    <!-- no 'id' required, EndpointMapping beans are automatically
detected by the MessageDispatcher -->
    <bean id="endpointMapping"
class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEnd
pointMapping">
        <property name="mappings">
            <props>
                <prop
key="{http://samples}orderRequest">getOrderEndpoint</prop>
                <prop
key="{http://samples}order">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>
    <bean id="getOrderEndpoint" class="samples.GetOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
    <bean id="createOrderEndpoint" class="samples.CreateOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
<beans>
```

A fenti pélában a namespace URI a http://samples, a lokális orderRequest névhez a getOrderEndpoint tartozi az order local névhez a createOrderEndpoint tartozik.

Egy másik leképezési mód a SoapActionEndpointMapping. Használhatjuk a SOAPAction http fejlécet is az üzenetek kiosztásához. Minden kliens elküldi ezt a fejlécet a SOAP kérésében, és a fejléc tartalma a WSDL állományból nyerhető ki. Ha ezeket egyedivé tesszük, akkor használhatóak a leképzéshez. Példa:

```
<beans>
    <bean id="endpointMapping"
class="org.springframework.ws.soap.server.endpoint.mapping.SoapActionEndp
ointMapping">
        <property name="mappings">
            <props>
                <prop
key="http://samples/RequestOrder">getOrderEndpoint</prop>
                <prop
key="http://samples/CreateOrder">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>
    <bean id="getOrderEndpoint" class="samples.GetOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
    <bean id="createOrderEndpoint" class="samples.CreateOrderEndpoint">
        <constructor-arg ref="orderService"/>
    </bean>
</beans>
```

A fenti példában, ha a SoapAction http://samples/RequestOrder, akkor a getOrderEndpoint-hoz, ha http://samples/CreateOrder akkor a createOrderEndpoint-hoz lesz irányítva.

Kérések elfogása

A végpont leképzésekben említett végpont elfogók használatához az EndpointIterceptor interface-t kell implementálnunk. Ez az interface 3 metódust definiál

1. HandleRequest, a kérés üzenet feldolgozása mielőtt a végpontjoz kerülne az üzenet. Egy boolean értékkel tér vissza ez a függvény, ettől az értéktől függ, hogy folytatódik-e a feldolgozási lánc, vagy sem.
2. HandleResponse, a normál válasz üzenetek kezelése, miután a végpont feldolgozta a kérést, szintén boolean értékkel tér vissza, és ettől függ, hogy a válasz üznet továbbításra kerül-e.
3. A hiba üzenetek kezelése, miután a végpont feldolgozta az üzenetet, szintén boolean érték és szintén ez az érték dönti el, hogy a válasz üzenet elküldésre kerül-e.

Hasznos lehet a kimenő és bejövő üzenetek naplózása, ezekre használható a PayloadLoggingInterceptor és a SoapEnvelposeLoggingInterceptor. Az első csak az üzenetek hasznos részét, míg az utóbbi a teljes SOAP borítékot (fejlécekkel együt) naplózza. Mindkét elfogónak van logRequest és logResponse tulajdonsága, amelyet ha hamisra állítunk, akkor kikapcsolhatjuk a naplózást. Az alábbi példában a PayloadLoggingIterceptor beállítását láthatjuk:

```
<beans>
    <bean id="endpointMapping"

class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEnd
pointMapping">
        <property name="interceptors">
            <list>
                <ref bean="loggingInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop
key="{http://samples}orderRequest">getOrderEndpoint</prop>
                <prop
key="{http://samples}order">createOrderEndpoint</prop>
            </props>
        </property>
    </bean>
    <bean id="loggingInterceptor"

class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingI
nterceptor"/>
</beans>
```

A PayloadValidatingIterceptor használatával a be- és kimenő üzenetek validálhatjuk egy vagy több séma felhasználásával. A következő példában csak a válasz üzenetet ellenőrizzük le az orders.xsd szerint, de a kérést nem:

```
<bean id="validatingInterceptor"
class="org.springframework.ws.soap.server.endpoint.interceptor.PayloadVal
idatingInterceptor">
    <property name="schema" value="/WEB-INF/orders.xsd"/>
    <property name="validateRequest" value="false"/>
    <property name="validateResponse" value="true"/>
</bean>
```

A kéréseket általában nem validáljuk, mert, ekkor túl kötött lenne a szolgáltatásunk, megelégszünk annyival, hogy a végponthoz elegendő információ jut-e el.

Hasznos lehet, ha át tudjuk alakítani az üzeneteket. Erre használhatjuk a PayloadTransformingIterceptor-t. Ez XSL stíluson alapul, és rendkívül hasznos lehet ha a régebbi verziójú szolgáltatásunk üzeneteit az új-ra tudjuk átalakítani.

```
<bean id="transformingInterceptor"
class="org.springframework.ws.server.endpoint.interceptor.PayloadTransfor
mingInterceptor">
    <property name="requestXslt" value="/WEB-INF/oldRequests.xslt"/>
    <property name="responseXslt" value="/WEB-INF/oldResponses.xslt"/>
</bean>
```

Kivételkezelés

A végpont kivételeket a MessageDispatcher automatikusan elkapja, így csak az EndpopintException Resolver interface-t kall implementálnunk. Ebben csak a resolveException(MessageContext, endpoint, exception) metódus található. A legegyszerűbb implementációja a SimpleSoapException, amely SOAP hibát készít, amelynek a hiba üzenete a kivétel szövege. Ez az alapbeállítás, de ez a beállítás felülírható.

A SoapFaultMappingExceptionResolver lehetővé teszi a kivételek osztály szerinti hozzárendelését.Az alábbi példában a ValidationFailException típusú kivételeket kliens oldali SOAP hibává alakítja, ahol a hibaüzenet az Invalid request lesz. Ha más kivétel történik, akkor szerver oldali hibát generál a kivétel szövegével.

```
<beans>
    <bean id="exceptionResolver"
class="org.springframework.ws.soap.server.endpoint.SoapFaultMappingExcept
ionResolver">
        <property name="defaultFault" value="SERVER"/>
        <property name="exceptionMappings">
            <value>

org.springframework.oxm.ValidationFailureException=CLIENT,Invalid request
            </value>
        </property>
    </bean>
</beans>
```

Kliens oldal

A Spring WS kliens oldali webszolgáltatások API-t is tartalmaz, amely egy XML vezérelt hozzáférést biztosít a webszolgáltatáshoz. Támogatja a marshallingot, így könnyebben kezelhetjük a Java objektumainkat.

Az org.springframework.ws.client.core csomagban találhatóak a kliens oldali API főbb szolgáltatásai. Az ebben található osztályok különféle metódusokat tartalmaznak XML üzenetek küldésére és fogadására, objektumok marshalling-jára, mielőtt elküldenénk azokat.

WebServiceTemplate

A WebServiceTemplate a legfőbb osztály a kliens oldalon. Ez olyan metódusokat tartalmaz, amelyek lehetővé teszik üzenetek küldését és fogadását. Továbbá képes küldés előtt a Java objektumain marshalling-jára, és a bejövő üzenetek unmarshalling-jára.

A WebSericeTemplate egy URI-t használ az üzenetek céljaként. Ezt az URI-t megadhatjuk a defaultUri tulajdonságban, vagy megadhatjuk metódus híváskor is. Ezt az URI-t átalakítja WebServiceMessageSender-ré, amely az XML üzentek küldéséért felel a szállítási rétegen

keresztül. Egy vagy több üzenetküldőt is beállíthatunk a messageSender vagy a messageSenders tulajdonságokkal.

A HTTP protokollon keresztüli küldésre két implementációja van a WebServiceMessageSender-nek:

1. HttpUrlConnectionMessageSender: ez az alap implementáció, amely a Java által nyújtott lehetőségeket használja fel.
2. CommonsHttpMessageSender: amely a Jakarta.Commons.HttpClient.jét használja. Ez fejlettebb és könnyen használható funkciókat rejt magában.

A http átvitel használatához, a defaultUri-nak valami hasonlót kell beállítani: http://example.com/service. Az alábbi példában az alapbeállítást láthatjuk http használatához:

```
<beans>
    <bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
    <bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="defaultUri"
value="http://example.com/WebService"/>
    </bean>
</beans>
```

A következő példa, azt mutatja be, hogy hogyan írhatuk felül az alapbeállítást és a CommonHttp használatát http azonosításhoz.

A kliens oldal is több szállítási módszert támogat:

1. JMS: a Spring WS a JmsMessageSender segítségével a webServiceMessage-t JMS üzenetté alakítja, elküldi, majd megkapja a választ ha van.Ebben az esetben a defaultUri vagy az uri paraméternek JMS URI-nek kell lennie, amely a jms: előtaggal kezdődik, például: jms:RequestQueue?deliveryMode=NON_PERSISTENT. Alapbeállításban JMS ByteMessage-t küld, de ezt felülírhatjuk például TextMessage-re a messageType paraméter módosításával.
2. E-mail: a MailMessageSender-rel az üzeneteket SMTP-n keresztül küldhetjük, és IMAP vagy POP3 segítségével fogadhatjuk. Az defaultUri-t vagy az uri-t mailto-ra kell állítani a használatához., például mailto@john@examples.com.

Message Factories

Két különböző Message Factory van a Spring WS-ben a SaajSoapMessageFactory és az AxiomSoapMessageFactory. Ha nem mondjuk meg külön a messageFactory tulajdonságban, hogy melyik legyen, akkor alapértelmezésként a Spring WS a SaajSoapMessageFactory-t használja.

Üzenetek küldésére és fogadására több metódus is szolgál. Az egyik legegyszerűbb a simpleSendAndReceive(..), melynek segítségével egyszerűen küldhetünk és fogadhatunk

XML üzeneteket. Java objektumok küldéséhez több küldési függvénnyel (send(..) rendelkezik, paraméterként az objektum. A WebServiceTemplate-ben található marshallSendAnrReceive(..) függvény az objektum XML-lé való konvertálását elküldia Marshaller-nek, és a válasz üzenetet az Unmarshaller-nek.

A SOAP üzenetek fejlécéhez a WebServiceCallBack interface-t használhatjuk, amellyel hozzáférhetünk az üzenethez miután elkészült, és a küldés előtt. Erre a célra használhatjuk a SoapActionCallback-et is.

Security

A Spring WS 3 különböző területen támogatja a webszolgáltatások biztonságát:

1. Azonosítás: a felhasználók azonosítása.
2. Digitális aláírások: az aálíró privát kulcsa és a dokumentum alapján képzett hash függvény eredménye.
3. Titkosítás: az üzeneteket átalakíthatjuk titkosított formára, és visszaállíthatjuk ebből az üzenetet.

Az XwsSecurityIterceptor egy végpont elfogó, amely a Sun XML és webszolgáltatások csomagjában (XWSS) található. A szokásos végpont leképezéseknél adhatjuk meg. Az XwsSecurityIteceptor működéséhez szükségünk van egy securityPolicy állományra. Ez is egy XML dokumentum, amelyben konfigurálhatjuk a biztonsági intézkedésinket. Ezen kívül egy vagy több CallBackHandler szükséges még. Ezekkel tudunk például tanúsítványokat, privát kulcsokat elkérni. A Spring WS a leggyakrabban használt estekre már tartalmaz ilyen funkciókat. Egy másik végpont elfogó, ami a biztonsággal kapcsolatos az a Wss4jSecurityInterceptor, amely az Apache WSS4J-n alapul.A WSS4J esetén nincs szükség külön konfigurációs állományra, tulajdonságokon keresztül lehet vezérelni. Az alábbiakban az XwsSecurityInterCeption bemutatása következik.

Azonosítás

A Spring WS-ben kétféle módszer van a felhasználók azonosítására, az egyik felhasználónév-jelszó alapú, vagy lehet X509 tanúsítvány alapú.

A legegyszerűbb azonosítási forma a felhasználónév jelszó páros. Ebben az esetben a SOAP fejléc tartalmaz egy UserName és egy Password elemet. Az UserName elemben található a felhasználónév és a Passowrd elemben a szöveges jelszó. Példa a beállítására:

```
<xwss:SecurityConfiguration
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:RequireUsernameToken passwordDigestRequired="false"
nonceRequired="false"/>
</xwss:SecurityConfiguration>
```

A SimplePasswordValidatonCallBackHandler a legegyszerűbb jelszó validáló eszköz, a jelszavakat a users tulajdonságban lévőkkel hasonlítja össze. A SpringPlainTextPasswordValidationCallbackHandler a Spring Security-t használja, és szüksége van egy AuthenticationManager-re a működéséhez. Ezt a manager-t használja a jelszók érvényesítéséhez. Ha az azonosítás sikeres, akkor a SecurityContextHolder-ben tárolásra kerül. Az alábbi példában láthatjuk a konfigurációját:

```
<beans>
  <bean id="springSecurityHandler"
class="org.springframework.ws.soap.security.xwss.callback.SpringPlainText
PasswordValidationCallbackHandler">
    <property name="authenticationManager" ref="authenticationManager"/>
  </bean>
  <bean id="authenticationManager"
class="org.springframework.security.providers.ProviderManager">
      <property name="providers">
          <bean
class="org.springframework.security.providers.dao.DaoAuthenticationProvid
er">
              <property name="userDetailsService"
ref="userDetailsService"/>
          </bean>
      </property>
  </bean>
  <bean id="userDetailsService"
class="com.mycompany.app.dao.UserDetailService" />
</beans>
```

A jelszavakat nem csak szöveges módon küldhetjük, hanem a jelszó kivonatát is használhatjuk erre a célra. A jelszó kivonat a jelszóból képzett hash függvény eredménye. Ekkor a SOAP fejlécben a Password elemben ez a kivonat fog szerepelni. Használatához a passwordDigestRequired tulajdonságot kell true-ra állítanunk.
A SimplePasswordValidationCallbackHandler a kivonatos jelszavakkal is működik. A SpringDigestPasswordValidationCallbackHandler működéséhez szükség van egy UserDetailService működésére. Ezt a szolgáltatást használja fel a felhasználóhoz tartozó kivonatos jelszó megszerzésére, miután ez megvan, összehasonlítja az üzenetben lévővel. Hamegegyeznek, akkor a SecurityContextHolder-ben tárolásra kerül, a userCache tulajdonság állításával a betöltött felhasználók adatait tárolhatjuk a gyorsabb hozzáférés miatt. Példa:

```
<beans>
    <bean
class="org.springframework.ws.soap.security.xwss.callback.SpringDigestPas
swordValidationCallbackHandler">
        <property name="userDetailsService" ref="userDetailsService"/>
    </bean>
    <bean id="userDetailsService"
class="com.mycompany.app.dao.UserDetailService" />
</beans>
```

A Spring WS-ben lehetőségünk van X509-es tanúsítványon keresztüli azonosításra. Ebben az esetben a SOAP üzenet tartalmaz egy BinarySecurityToken elemet, amely a Base-64-gyel kódolt X509-es tanúsítványunk szerepel. A securitypolcy file-nkban a RequireSignature elemet kell tartalmaznia, ha használni akarjuk ezt az azonosítást.

```
<xwss:SecurityConfiguration
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:RequireSignature requireTimestamp="false">
</xwss:SecurityConfiguration>
```

Ha a SOAP üzenetben nincs tanusítány, akkor SOAP hibát küld a feladónak vissza az XwsSecurityInterceptor. Ha jelen van, akkor egy CertificateValidationCallback-et fog meghívni. 3 különböző kezelő van erre, ezek közül a legfontosabb a KeyStoreCallbackHandler.

A KeyStoreCallbackHandler esetén a tanúsítvány ellenőrzése 3 lépésből áll:

1) A kezelő ellenőrzi, hogy a tanúsítvány a privát keystore-ban van-e, ha benne van akkor érvényes.
2) Ha a tanúsítvány nincs benne, akkor a kezelő ellenőrzi, hogy az aktuális dátum benna van-e a tanúsítvány érvényességi idejében. Ha nincs benne akkor nem érvényes, ha benne van akkor az utolsó lépés következik.
3) Egy tanúsítvány útvonal generálódik a tanúsítványhoz. Ez azt jelenti, hogy a kezelő megnézi, hogy a tanúsítványt a trustStore-ban leírt tanúsítvány állította-e ki. Ha fel tudja építeni ezt az útvonalat, akkor a tanúsítvány érvényes, különben nem.

Ha KeyStoreCallbackHandler-t csak tanúsítvány ellenőrző célokra akarjuk használni, akkor elég csak a trustStore tulajdonságot beállítanunk:

```
<beans>
    <bean id="keyStoreHandler"
class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbac
kHandler">
        <property name="trustStore" ref="trustStore"/>
    </bean>
    <bean id="trustStore"
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:truststore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

Digitális aláírások

Két fő feladat van a digitális aláírásokkal kapcsolatban, az egyik az üzenetek aláírása, a másik az aláírt üzenetek ellenőrzése.

Az aláírt üzenet tartalmaz egy BinarySecurityToken-t ami tartalmazza az aláírt üzenet tanúsítványát, és tartalmaz egy SignedInfo blokkot, amely jelzi, hogy az üzenet melyik része van aláírva. Ahhoz, hogy bizonyosak legyünk abban, hogy a SOAP üzenet tartalmazza a BinarySecuityToken-t, ahhot a konfigurációs file-ban a RequireSignature-t kell beállítanunk. Tartalmazhat egy SignatureTarget elemet is, amely specifikálja, hogy az üzenet melyik részének kell aláírva lennie. Ha nincs jelen az üzenetben az aláírás akkor egy SOAP hiba üzenetet küld a feladónak. A KeyStoreCallbackHandler az aláírások ellenőrzésére a trustStore tulajdonságot használja, példa:

```
<beans>
    <bean id="keyStoreHandler"
class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbac
kHandler">
        <property name="trustStore" ref="trustStore"/>
    </bean>
    <bean id="trustStore"
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location"
value="classpath:org/springframework/ws/soap/security/xwss/test-
truststore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

Üzenetek aláírásakor az XwsSecurityInterceptor addja hozzá a BinarySecurityToken-t és a SignedInfo-t az üzenethez. Ahhoz, hogy ez megtörténjen a securityPolcy file-nak tartalmaznia kell a Sign elemet. Tartalmazhatja a SignatureTarget és különböző más elemet is. Itt definiálhatjuk, hogy milyen kulcsot (privát vagy szimmetrikus) használjon.

```
<xwss:SecurityConfiguration
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
      <xwss:Sign includeTimestamp="false" />
</xwss:SecurityConfiguration>
```

Az üzenetek aláírásához a KeyStoreCallbackHandler a keyStore tulajdonságot használja. Be kell állítanunk a privateKeyPassword tulajdonságot, hogy hozzáférjünk a privát kulcshoz. Példa:

```
<beans>
    <bean id="keyStoreHandler"
class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbac
kHandler">
        <property name="keyStore" ref="keyStore"/>
        <property name="privateKeyPassword" value="changeit"/>
    </bean>
    <bean id="keyStore"
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:keystore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

Titkosítás

Az üzenet titkosításr, olyan formára alakítjuk át az üzenetünket, hogy csak a megfelelő kulccsal lehessen elolvasni, a titkosítás feloldása az a művelet, amikor a titkosított üzenetünket, újra olvasható formára alakítjuk.

Ahhoz, hogy feloldjuk a titkosított SOAP üzenetünket a securityPolcy állománynak tartalmaznia kell a RequireEncryption elemet. Ebben az elemben továbbá lehet EncryptionTarget elem, amely azt jelzi, hogy az üzenet melyik részének kell titkosítva lennie.

```
<xwss:SecurityConfiguration
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:RequireEncryption />
</xwss:SecurityConfiguration>
```

Ha a bejövő üzenet nincs titkosítva, akkor az XwsSecurityInterceptor Soap hibával tér vissza. Ha jelen van, akkor a DecryptionKeyCallback indít el a regisztrált kezelőn. A Spring WS-ben csak egy ilyen kezelő van a KeyStoreCallbackHandler. Be kell állítanunk a privateKeyPassword tulajdonságot, hogy hozzáférjünk a privát kulcshoz, amit a titkosítás feloldására használunk.

```
<beans>
    <bean id="keyStoreHandler"
class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbac
kHandler">
        <property name="keyStore" ref="keyStore"/>
        <property name="privateKeyPassword" value="changeit"/>
    </bean>
    <bean id="keyStore"
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:keystore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

A kimenő SOAP üzenetek titkosításához a konfigurációs file-nak tartalmaznia kell az Encrypt elemet, továbbá tartalmazhatja az EncryptionTarget elemet.

```
<xwss:SecurityConfiguration
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:Encrypt />
</xwss:SecurityConfiguration>
```

A KeyStoreCallbackHandler-t használjuk, ahol a trustStore tulajdonságnak kell lennie beállítva:

```
<beans>
    <bean id="keyStoreHandler"
class="org.springframework.ws.soap.security.xwss.callback.KeyStoreCallbac
kHandler">
        <property name="trustStore" ref="trustStore"/>
    </bean>
    <bean id="trustStore"
class="org.springframework.ws.soap.security.support.KeyStoreFactoryBean">
        <property name="location" value="classpath:truststore.jks"/>
        <property name="password" value="changeit"/>
    </bean>
</beans>
```

Üzenetek naplózása

Lehetőségünk van a Spring WS-ben az összes szerver és kliens oldali üzenetek naplózására, a Common Logging interface-n kersztül.

A szerver oldali üzenetek naplózásához az org.springframework.ws.server.MessageTracing naplózóját DEBUG vagy TRACE szintre kell állítani. A debug szinten csak az üzenet hasznos része mentődik le, trace szinten a teljes üzenet. Külön állíthatjuk a küldött és a fogadott

üzenetek naplózását. A kliens oldalon hasonló naplózó működik: org.springframework.ws.client.MessageTracing

Szavazó program

Ebben a fejezetben egy konkrét webszolgáltatás készítésének a menetét ismerhetjük meg a Spring WS alkalmazásával.

Szerver

A kitűzött cél, egy olyan egyszerű szolgáltatás elkészítése, amely bemutatja a Spring WS lehetőségeit. Ez a szolgáltatás, egy egyszerű szavazó program. A program feladata, hogy a webszolgáltatás segítségével bárki szavazhasson, megtekintse az eredményeket. Adminisztrátori jogokkal rendelkező felhasználók új kérdést és hozzá tartozó válaszlehetőségeket tehetnek fel. A feladat végrehajtásához Tomcat szervert és MySQL adatbázist fogok használni.

Tervezés

A feladatleírásból következően a szervernek a következő szolgáltatásokat kell tudnia:

- Kérdés és a hozzá tartozó válaszok lekérdezése
- Szavazás végrehajtása
- Kérdés szavazati eredményeinek visszaadása
- Új kérdés felvétele

A kérdés eredményének lekérdezése, nem külön szolgáltatás, hanem a kérdés és válaszok visszaadásával együtt valósul majd meg. A szerveralkalmazás középpontja a VotingService osztály lesz, ehhez kapcsolódnak majd a végpontok. Az adatbázis elérését, és az adatbáziskéréseket a DBUtils osztály fogja szolgáltatni.

XML üzenetek

Mivel contact-first a Spring WS lényege, ezért a tervezést nem az osztályok megtervezésével, hanem az XML üzenetek tervezésével kezdődik. Ezek alapján tervezhetjük meg az alkalmazásunk felépítését. Az üzenetekben biztosan fognak szerepelni a következő objektumok:

```
<Question xmlns="http://thesis.com/voting/schemas">
    <ID>1</ID>
    <Text>What is your fvourite fruit?</Text>
    <VoteCount>23</VoteCount>
</Question>

<Answer xmlns="http://thesis.com/voting/schemas">
    <ID>2</ID>
    <Text>Apple</Text>
    <VoteCount>12</VoteCount>
</Answer>

 <User xmlns="http://thesis.com/voting/schemas">
    <Name>admin</Name>
    <Password>asd23</Password>
</User>
```

Ezek alapján lesz kérdés lekérdező üzenet, amely a kérdést, és a hozzá tartozó üzeneteket tartalmazza. A választás üzenet a kérdés ID-ját és a kiválasztott válasz ID-ját tartalmazza. A kérdés beállításhoz tartozik egy felhasználó egy új kérdés és a lehetséges új válaszok. Például a kérdést lekérdező üzenet válaszüzenete:

```
<QuestionResponse xmlns="http://thesis.com/voting/schemas">
    <Question xmlns="http://thesis.com/voting/schemas">
        <ID>1</ID>
        <Text>What is your fvourite fruit?</Text>
        <VoteCount>23</VoteCount>
    </Question>
    <Answer xmlns="http://thesis.com/voting/schemas">
        <ID>2</ID>
        <Text>Apple</Text>
        <VoteCount>12</VoteCount>
    </Answer>
</QuestionResponse>
```

Az üzenetek alapján elkészítjük az adatbázisunkat a következő táblákkal:

- User, név és jelszó tárolása
- Question, kérdés szövegének tárolása
- Answers, válasz szövege, a hozzátartozó kérdés azonosítója
- Votes, a kérdés és a válasz azonosítója

Séma

Az előállított példa XML üzenetekből tudunk XML sémát generálni. Erre a feladatra az Oxygen XML editor próbaverzióját használtam. A generált séma még némi módosításra szorul, meg kell adnunk például, hogy egy kérdés elemhez több válasz is lehetséges, azaz a maximális előfordulását nem kötjük meg. Az előállított sémából tudunk Java osztályokat generálni az xjc segítségével. Az xjc paraméterben kapja meg a sémát, majd a séma névteréből képzett csomagba menti ki az osztályokat.

A project készítése

A project felépítése Maven2-vel történik, a következő paranccsal:

```
mvn archetype:create -DarchetypeGroupId=org.springframework.ws -
DarchetypeArtifactId=spring-ws-archetype -DarchetypeVersion=1.5.6 -
DgroupId=com.thesis.voting -DartifactId=VotingService

mvn eclipse:eclipse
```

Az első paranccsal a Maven előállítja nekünk a Spring WS-hez szükséges XML állományokat, míg a második parancs eclipse projectet készít az állományokból. Ezután importáljuk a projektünkbe a sémából generált osztályokat. Elkészítjük a VotingService osztályt, konstruktorában inicializáljuk a dbutilst. Elsőnek definiálnunk kell, hogy milyen végpontokat szeretnénk felhasználni.

Végpontok

A következő végpontokat kell implementálnunk:

- Kérdés lekérdezése: GetQuestionEndPoint
- Szavazás: VotingEndPoint
- Új kérdés mentése: SetQuestionEndPoint
- Felhasználó jogosult-e kérdés mentésére: IsAdminEndPoint

Az utolsó végpont nem szükséges, hiszen a kérdés mentése üzenetben megtalálhatóak a felhasználó információi, de gyakorlati szempontból hasznos, hiszen a kliens program majd el tudja dönteni, hogy engedje-e a felhasználót a funkció esetleges hiábavaló használatát. A spring-ws-servlet.xml konfigurációs állományba már be is állíthatjuk ezeket a végpontokat.
A vService bean-ben definiáljuk a fő osztályt az AuctionService-t,
A getQuestionEndPoint és a SetQuestionEndpoint marshalling felhasználásával fog történni, míg a másik kettőe Dom Element –ek felhasználásával. Ennek megfelelően állítjuk be a marshalling tulajdonságot. A Jaxb2Marshaller-t használjuk, és a classesToBeBoundban, megadjuk a sémából generált osztályokat, legalábbis azokat, amelyeket használni szeretnénk ezeken a végpontokon.

```
    <bean id="GetQuestionEndPoint"
class="com.thesis.voting.ws.GetQuestionEndPoint">
        <constructor-arg ref="vService"/>
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="SetQuestionEndPoint"
class="com.thesis.voting.ws.SetQuestionEndPoint">
        <constructor-arg ref="vService"/>
        <constructor-arg ref="marshaller"/>
    </bean>

    <bean id="marshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>com.thesis.voting.schemas.QuestionResponse</value>
                <value>com.thesis.voting.schemas.QuestionRequest</value>

<value>com.thesis.voting.schemas.SetQuestionRequest</value>
                <value>com.thesis.voting.schemas.Result</value>
            </list>
        </property>
    </bean>

    <bean id="vService" class="com.thesis.voting.ws.VotingServiceImp"/>

    <bean id="VotingEndPoint"
class="com.thesis.voting.ws.VotingEndPoint">
        <constructor-arg ref="vService"/>
    </bean>

    <bean id="IsAdminEndPoint"
class="com.thesis.voting.ws.IsAdminEndPoint">
        <constructor-arg ref="vService"/>
    </bean>
```

Elkészítjük a megfelelő végpont osztályokat, nézzünk mind a 2 típusra példát. Az első végpont, amit implementálnunk kell az a kérdést visszaadó, a GetQuestionEndPoint.

```
public class GetQuestionEndPoint extends
AbstractMarshallingPayloadEndpoint{
    private final VotingServiceImp votingService;

    public GetQuestionEndPoint(VotingServiceImp votingService, Marshaller
marshaller) {
      super(marshaller);
        this.votingService = votingService;
    }

    protected Object invokeInternal(Object Request) throws Exception {
      return votingService.getActQuestion();
    }
}
```

A végpontunk nagy egyszerű, a konstruktorában megkapja központi osztályt. Csak az invokeInternal metódust kell felülírnunk, ami paraméterben megkapja a kérés üzenetet. A kérés üzenetünk üres, így azzal nem is foglalkozunk. A votingService-ben található getActQuestion() metódus visszatérési értékével tér vissza.

```
	public QuestionResponse getActQuestion()
	{
		dbutils.Connect();
		ObjectFactory of = new ObjectFactory();
		QuestionResponse qr = of.createQuestionResponse();
		Question q = dbutils.getCurrentQuestion();
		if (q == null)
		{
			q = new Question();
			q.setText("Error");
			Answer a = new Answer();
			a.setText("Could not find question");
			qr.setQuestion(q);
			qr.getAnswer().add(a);
			return qr;
		}
		else
		{
		qr.setQuestion(q);
		List<Answer> answers =
	dbutils.getQuestinAnswrs(q.getID().intValue());
		for (int i=0;i<answers.size();i++)
			qr.getAnswer().add(answers.get(i));
		return qr;
		}
	}
```

Csatlakozik az adatbázishoz, majd a sémából generált osztályokból létrehozunk egy QuestionResponse-t. Az adatbázisból inicializáljuk a kérdést, ha van aktuális kérdés, akkor ezt beállítjuk a QuestionResponse –ba, majd lekérdezzük a hozzá tartozó válaszokat, és ezeket is beállítjuk.

ADBUtilsban található egy Connect() metódus, amellyel csatlakozunk az adatbázishoz. Ezen kívül a szolgáltatásainkhoz kapcsolódó metódusok vannak. Az adatbázisból a legfrissebb kérdés lekérdezése:

```
      public Question getCurrentQuestion()
      {
            try
            {
            Statement s = connection.createStatement ();
            ResultSet rs = s.executeQuery("SELECT * FROM question ORDER
By ID Desc LIMIT 1");

            if (rs.first())
            {
                  Question q = new Question();
                  q.setID(BigInteger.valueOf(rs.getInt("ID")));
                  q.setText(rs.getString("Text"));

      q.setVoteCount(BigInteger.valueOf(rs.getInt("ACount")));
                  rs.close();
                  return q;
            }
            rs.close();
            s.close();
            return null;
            } catch (Exception e) {return null;}
      }
```

Egy másik típusú végpontunk a szavazás, a VotingEndPoint. Konstruktorában beállítjuk a VotingService-t és inicializálunk 2 XPathExpression-t:

```
          Namespace namespace = Namespace.getNamespace("voting",
"http://thesis.com/voting/schemas");
        QIDExpression = XPath.newInstance("//voting:QID");
        QIDExpression.addNamespace(namespace);
        AIDExpression = XPath.newInstance("//voting:AID");
        AIDExpression.addNamespace(namespace);
```

Beállítjuk a megfelelő névteret. A QIDExpression-t fogjuk felhasználni a kérdés azonosítójának, az AIDExpressiont a válasz azonosítójának kinyerésére. Az invokeInternal metódust kell deklarálnunk:

```
    protected Element invokeInternal(Element Request) throws Exception {
        int qID = Integer.parseInt(QIDExpression.valueOf(Request));
        int aID = Integer.parseInt(AIDExpression.valueOf(Request));
        String res;
        if (votingService.Vote(qID,aID))
            res = "0";
        else
            res = "1";
        Element e = new Element("VoteResultRespone");

e.setNamespace(Namespace.getNamespace("voting","http://thesis.com/voting/
schemas"));
        Element e2 = new Element("VoteResult");

e2.setNamespace(Namespace.getNamespace("voting","http://thesis.com/voting
/schemas"));
        e.addContent(e2.addContent(res));
        return e;
    }
```

A két XPath kifejezés felhasználásával kinyerjük a megfelelő adatokat, majd ezekkel a votingService-n keresztül az adatbázisba mentjük. Felépítjük a válaszüzenetet, ha sikeresen bekerült az adatbázisba, akkor a VoteResult elem értéke 0, ha nem akkor 1 lesz. Visszatérünk a felépített elemmel.
A másik két végpontot hasonló módszerekkel lettek megvalósítva. A végpontok leképzését meg kell adnunk a konfigurációs file-ban, nézzük például a GetQuestionEndPoint-ét:

```
<bean
class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEnd
pointMapping">
            <property name="mappings">
        <props>
            <prop
key="{http://thesis.com/voting/schemas}QuestionRequest">GetQuestionEndPoi
nt</prop> …
    </property>
</bean>
```

Használhatnánk biztonsági interceptort, de sajnos a Spring WS-ben még egy meg nem oldott probléma az interceptorok végpontonkénti megadása, ezért van szükségünk a felhasználó adatainak az XML üzenetben való küldésére. Ezért biztonsági szempontból mindenképp érdemes a szolgáltatást HTTPS protokollon keresztül használni. A kérdése beállítást akár egy új különálló webszolgáltatásban is el lehetne készíteni.

WSDL

A spring-ws-servlet.xml –ben be kell állítanunk a WSDL állomány automatikus generálását:

```
<bean id="voting"
class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
    <property name="schema" ref="schema"/>
    <property name="portTypeName" value="VotingResource"/>
    <property name="locationUri" value="/VotingService"/>
    <property name="targetNamespace"
value="http://thesis.com/voting/schemas"/>
</bean>
<bean id="schema" class="org.springframework.xml.xsd.SimpleXsdSchema">
    <property name="xsd" value="/WEB-INF/voting.xsd"/>
</bean>
```

Meg kell adnunk a sémánk elérését, és ebből fog generálódni a WSDL állomány. A szerver alkalmazásból az mvn package paranccsal készíthetünk war, állományt, amelyet a Tomcat szerver webapp könyvtárába másolásával telepíthetjüük a szerverre. Letesztelhetjük az automatikus WSDL generálást a http://localhost:8080/VotingService/voting.wsdl címen:

Teszt Kliens

A szolgáltatást tesztelő kliens egy egyszerű konzolos alkalmazás. Fő osztálya a VotingClient. Felhasználjuk a sémából generált osztályokat, ezeken kívül a végpontok elérésre készítünk külön osztályokat. A kliens oldalon is használunk XML konfigurációs állományt, az applicationContext.xml-t. Ebből a file-ból készül el az ApplicationContext osztályunk:

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext(
             "applicationContext.xml");
```

Ebben az állományban definiáljuk a WebServiceTemplate-t, és a hozzá tartozó beállításokat:

```
  <bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
<bean id="messageSender"
class="org.springframework.ws.transport.http.CommonsHttpMessageSender">
</bean>
<bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
        <constructor-arg ref="messageFactory"/>
        <property name="defaultUri"
value="http://localhost:8080/VotingService"/>
        <property name="messageSender" ref="messageSender" />
    </bean>
```

Meg kell adni a WSDL elérhetőségét:

```
    <bean id="abstractClient" abstract="true">
        <constructor-arg ref="messageFactory"/>
        <property name="destinationProvider">
            <bean
class="org.springframework.ws.client.support.destination.Wsdl11Destinatio
nProvider">
                <property name="wsdl"
value="http://localhost:8080/VotingService/voting.wsdl"/>
            </bean>
        </property>
    </bean>
```

Definiáljuk a végpontok eléréséhez szükséges osztályokat, két végpont elérésére itt is a Jaxb2Marshaller-t használjuk, ugyanolyan classesToBeBound beállításokkal.

```
    <bean id="GetQuestion" parent="abstractClient"
class="com.thesis.voting.client.GetQuestion">
        <property name="marshaller" ref="marshaller"/>
        <property name="unmarshaller" ref="marshaller"/>
    </bean>

    <bean id="Vote" parent="abstractClient"
class="com.thesis.voting.client.Vote">
    </bean>
```

A GetQuestionRequest osztály:

```
public class GetQuestion extends WebServiceGatewaySupport {
    public GetQuestion(WebServiceMessageFactory messageFactory) {
        super(messageFactory);
    }
    public QuestionResponse GetQuestionObject()
    {
      ObjectFactory of = new ObjectFactory();
      QuestionRequest qr = of.createQuestionRequest();
      return
(QuestionResponse)getWebServiceTemplate().marshalSendAndReceive(qr);
    }
}
```

A QuesiontRequest üres, mert nem is kell semmi a kérdés lekérdezéséhez. A marshalSendAndReceive visszaadja webszolgáltatástól kapott QuestionReponse elemet.

A VotingClientben inicializáljuk a GetQuestion osztályt az applicationContext-ből, majd meghívjuk a QuestionResponse metódusát:

```
            GetQuestion gq = (GetQuestion)
applicationContext.getBean("GetQuestion", GetQuestion.class);
            qr = gq.GetQuestionObject();
```

A visszakapott QuestionResponset beállítjuk egy publikus változóban, hogy a program további működése során könnyen hozzáférhessünk.

A szavazás végpontot kezelő Vote osztályunk esetén az üzenetet elő kell állítani a paraméterben megkapott értékekkel: String változóban tároljuk a küldendő XML üzenet tartlamát. Ebből készítjük a StringSource-t, amelyet, a sendSourceAndReceiveToResult segítségével továbbítani tudunk a szerver VotingRequest végpontjának. A végpont által visszaadott JDOMResult-ból Xpath segítségével nyerjük ki a VoteResult értékét, ha 0 az értéke true-val, ha 1, akkor false-al térünk vissza

```
String mes = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
                        "<VotingRequest
xmlns=\"http://thesis.com/voting/schemas\">";
            mes += "<QID>"+qid+"</QID>";
            mes += "<AID>"+aid+"</AID>";
            mes += "</VotingRequest>";
            StringSource source = new StringSource(mes);
```

A teszt kliens menüválasztós szerkezetű. Választani az adott lehetőségek előtt látható számmal lehet. A programból kilépni a quit utasítással lehet. A navigáláshoz egyszerűen be kell gépelni a lehetséges választ, például szavazáskor a válasz számát, új kérdésnél a kérdés szövegét. A menüpont kiválasztása, és a kívánt művelet után mindig a főmenü jelentkezik. A program főmenüjében3 lehetőség közül választhatunk:

```
Válasszon a lehetőségek közül. Kilépni a quit paranccsal lehet.
1: Szavazás.
2: Eredmények megtekintése.
3: Új kérdés. (azonosítás szükséges)
```

1. A kliens program indulási képernyője

Az első menüpont alatt, hajtódik végre a GetQuestionRequest, majd eredmények nélküli kiírása következik, és szavazhatunk:

```
1
Kérdés:Melyik a kedvenc gyümölcsöd?
1: Alma
2: Körte
3: Banán
4: Narancs
Kérem válasszon
```

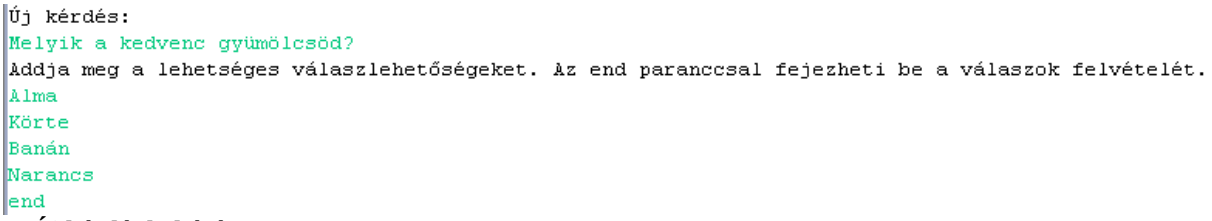
1. Szavazás menüpont

Helyes válaszlehetőség megadásával (jelen példában: 1-4), a szavazás végbemegy, és visszakapjuk a szervertől, hogy a szavazás eredményes volt-e, erről szöveges tájékoztatás kapunk.
Az Eredmények kiírása menüpontban, a kérdés és a válaszok kiírása történik, de most láthatóak a leadott szavazatok száma is. A kérdés kiírás metódus paraméterben kapja meg, hogy kiírja-e az eredményeket is:

```
      public void printQuestionResponse(boolean result)
      {
            if (result)
                  questionGet();
            if (qr!= null)
            {
                  String s;
            System.out.println("Kérdés:"+qr.getQuestion().getText());
            if (result)
                  System.out.println("Eddigi szavazatok:
"+qr.getQuestion().getVoteCount());
            List<Answer> a = qr.getAnswer();
            for (int i=0;i<a.size();i++)
            {
                  s = (i+1)+": "+a.get(i).getText();
                  if (result)
                        s += " Szavazatok: "+a.get(i).getVoteCount();
                  System.out.println(s);
            }

            }
      }
```

A 3. menüpont a felhasználónév és jelszó bekérésével kezdődik, ezzel inicializálva a Votingclientfelhasználóját. Az IsAdminRequest végponthoz kerülnek az adatok. Ha hamis érték érkezik vissza, akkor visszaugrunk a főmenübe, ha igaz érték, akkor bekérjük a kérdés szövegét, a lehetséges válaszok szövegeit.



1. Új kérdés bekérése

Ezekből és a felhasználó adataiból készül el a SetQuestionResponse, amit a szervernek eljuttatva lementi az adatbázisba, és ez lesz az aktuális kérdés.
A programot futtatható jar-ként exportáljuk.

Összefoglalás

A szakdolgozatból megismerhettük a webszolgáltatások építőköveit, nem túl részletesen, de annyira, hogy tudjunk a webszolgtatásokról beszélni. Megismerhettük az XML-t, ami mint láthattuk mindennek az alapja. Megismertük a webszolgáltatások elsődleges üzenettovábbítóját a SOAP-ot, és a szolgáltatás leírót a WSDL-t, és a szolgáltatás közétetelt az UDDI-t. Megnéztük, hogyan is épül fel ezekből a webszolgáltatás.

Áttekintettük a Spring WS alapelveit, működését., és az ahhoz szükséges konfigurációs állományok és osztályok felépítését. A Spring WS bemutatása már részletesebb volt, mint a webszolgáltatások felépítésének leírása, de még így is maradtak ki fontosabb részletek, de ezek a nem túl gyakran használt osztályok és konfigurációk leírásai. Próbáltam csak a lényegesebbekre koncentrálni, és nem elveszni a részletekben. A leírásokban számtalan példa található az osztályok működésének konfigurációihoz.

A Spring WS a gyakorlatban című részben láthattuk, hogy milyen egyszerűen és könnyen építhetünk fel webszolgáltatást a Spring WS segítségével. Ennek a fejezetnek a célja, hogy lássuk, hogyan is kell az elméleti részt átültetni a gyakorlatban, konkrét konfigurációs állományokkal és osztályokkal. A szemléltetéshez használt program egyszerű, így könnyen érthető, és könnyebben átültethető bonyolultabb programokba. A program számtalan módon továbbfejleszthető és bővíthető.

Végül zárszóként, hogy miért is érdemes Spring WS-t használni a webszolgáltatások készítéséhez:

- Contact-first alapú
- XML állományokkal konfigurálható
- Támogatja amarshalling-ot
- A végpontok megadása egyszerű
- A Végpont leképzés könnyen konfigurálható
- Többféle átviteli protokollt támogat

A felsorolt tulajdonságok miatt ma az egyik legnépszerűbb webszolgáltatás készítő eszköz a Spring WS. Használatát ajánlom minden webszolgáltatásokkal foglalkozónak.

Felhasznált irodalom

Könyvek:

Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama:
Java alapú webszolgáltatások
XML, SOAP, WSDL, UDDI

Gottdank Tibor:
Webszolgáltatások
XML alapú kommunikáció az Interneten

McLaughlin, Brett:
JAVA és XML

Nyékyné Gaizler Judit:
J2EE Útikalauz Java programozóknak

Internet:

http://static.springframework.org/spring-ws/sites/1.5/

http://static.springframework.org/spring-ws/sites/1.5/reference/html/what-is-spring-ws.html

http://static.springframework.org/spring-ws/docs/1.0-m3/reference/html/index.html

http://www.roseindia.net/spring/index.shtml

http://www.inf.unideb.hu/~jeszy/download/xml/xpath_10.pdf

http://www.gridshore.nl/blog/index.php?/archives/66-Using-Spring-ws-for-creating-a-webservice.html

http://www.gridshore.nl/blog/index.php?/archives/65-Creating-a-webservice-client-using-Spring-ws-and-maven2.html

http://swik.net/spring-ws

http://www.w3.org/

http://www.javagrund.hu/javasite/dokument/xml/xml_cikk.pdf

http://computerworld.hu/soa-jovo-uzleti-architekturalis-modellje.html

http://www.php-blog.hu/webszolgaltatas-wsdl-es-soap-alapok-i-resz.html

http://www.onjava.com/pub/a/onjava/2004/07/28/XMLBeans.html

http://cse-mjmcl.cse.bris.ac.uk/blog/2005/12/21/1135165498009.html

http://cse-mjmcl.cse.bris.ac.uk/blog/2005/12/21/1135165512463.html

http://gleichmann.wordpress.com/2008/02/24/flexible-marshalling-and-unmarshalling-using-spring-ws/

http://www.ddj.com/web-development/201802027

http://www.java-community.de/archives/48-Playing-with-Spring-WS.html

http://209.85.129.132/search?q=cache:M3f2WmwOZR0J:www.e-source.hu/jenci/jegyzetek/fpt_01-10.doc+SAx+dom+xpath&cd=10&hl=hu&ct=clnk&gl=hu&lr=lang_hu&client=firefox-a

http://www.tudasmorzsak.hu/dev-cikkek/38-dev/45-webszolgaltatasok-i-resz

http://www.linuxvilag.hu/content/files/cikk/10/cikk_10_50_53.pdf

http://www.kitebird.com/articles/jdbc.html

http://www.cafeconleche.org/books/xmljava/chapters/ch14s06.html