Designing and implementing Web Services

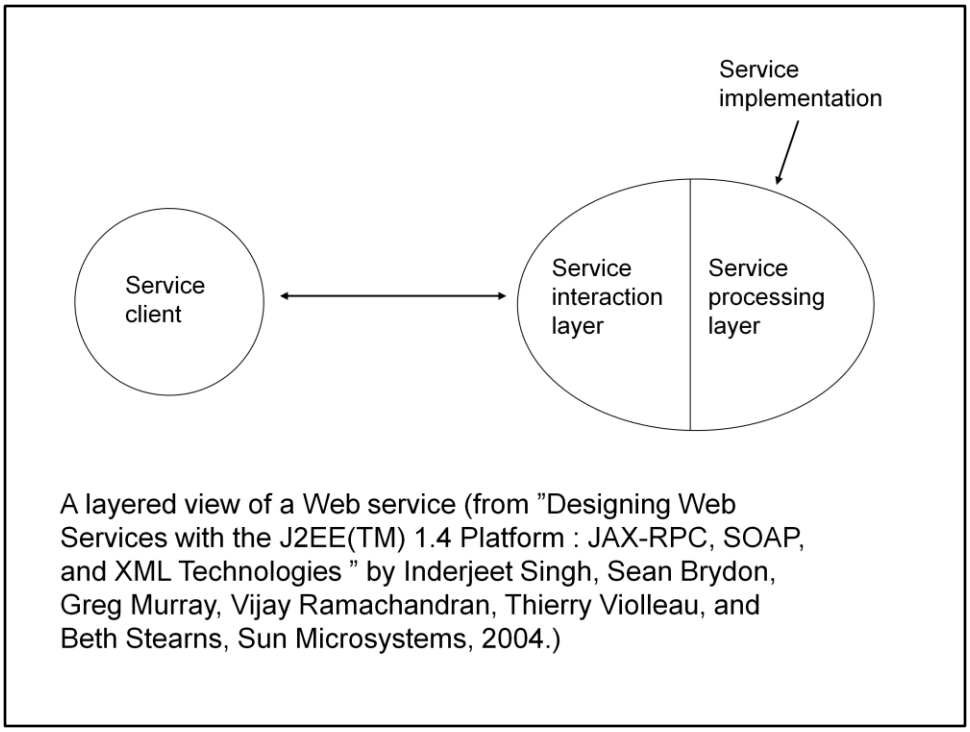
TIE-23600 Palvelupohjaiset järjestelmät

Designing Web Services

- Designing Web service capabilities for an application and designing the business logic of it are two different things
- Business logic design and implementation does not depend on Web service technologies
  - e.g. an existing legacy system
- To design Web service capabilities, one needs to decide on
  - what kind of interface with which service operations the Web service should have?
  - how to receive and process requests?
  - how to delegate the requests to the business logic?
  - how to formulate and send responses?
  - how to manage error situations and how to report about them?

Web-palvelu voidaan ajatella jaettavaksi kahteen erilliseen kokonaisuuteen: itse palvelun toiminnallisuuden toteuttava osa ja osa, joka mahdollistaa ko. toiminnallisuuden hyödyntämisen Web-palveluna. Tämä jako on oleellinen myös Web-palveluita suunniteltaessa. Itse palvelun toiminnallisuuden (puhutaan yleisesti myös liiketoimintalogiikasta) suunnittelun ei tarvitse riippua siitä onko se tarkoitettu käytettäväksi Web-palveluna eikä näin ollen myöskään riipu Web-palveluteknologioista. Se voi esimerkiksi olla olemassa oleva ohjelmisto, jota haluttaisiin hyödyntää Web-palveluna.

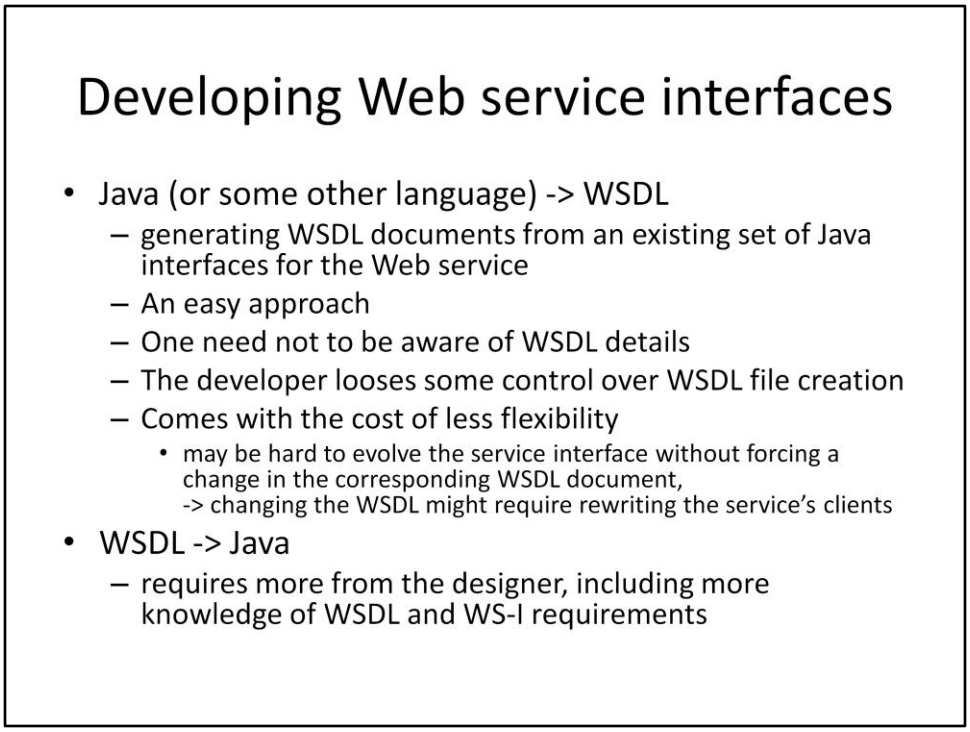
Osaa, joka mahdollistaa ko. sovelluksen hyödyntämisen Web-palveluna, suunniteltaessa tulee päättää millainen on asiakassovelluksille näkyvä rajapinta (operaatiot), millaista interaktioita halutaan tukea, miten tämä interaktio muutetaan interaktioksi toiminnallisuuden toteuttavan osan kanssa, miten virhetilanteet raportoidaan asiakassovellukselle jne. Joissain tapauksissa tässä sovelluslogiikan päälle ajateltavassa ja asiakkaalle näkyvässä kerroksessa halutaan myös esikäsitellä viestejä ennen kuin ne välitetään itse sovelluslogiikan toteuttavalle osalle. Lisäksi palvelun käytön analysointi (esim. viestien monitorointi) voidaan toteuttaa tässä kerroksessa.



A layered view of a Web service (from ”Designing Web Services with the J2EE(TM) 1.4 Platform : JAX-RPC, SOAP, and XML Technologies ” by Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, and Beth Stearns, Sun Microsystems, 2004.)

Palvelun interaktiokerros (interaction layer) koostuu asiakkaille näkyvästä ja niiden käyttämästä palvelun kontaktirajapinnasta (endpoint interface), viestien ja vastausten (SOAP) tulkinnan ja käsittelyn toteuttavasta logiikasta sekä integraatiosta sovelluslogiikan toteuttavan kerroksen kanssa. Vaikka kuvassa palvelun interaktiokerros ja varsinainen sovelluslogiikka on kuvattu yhtenä kokonaisuutena (Web-palvelu), saattavat ne käytännössä olla joko toteutettu tiiviisti integroituina tai selkeästi toisistaan erotettuina kokonaisuuksina, jotka on tavalla tai toisella integroitu toisiinsa.

Interaktiokerroksen tehtävänä on tulkita ja käsitellä (SOAP-)kutsut, delegoida kutsut sovelluslogiikkakerrokselle, muokata vastaukset tarvittavaan muotoon (SOAP) ja lähettää vastaukset asiakassovellukselle. Palvelun sovelluslogiikan toteuttavan kerroksen tehtävänä on toteuttaa varsinainen palvelun toiminnallisuus.

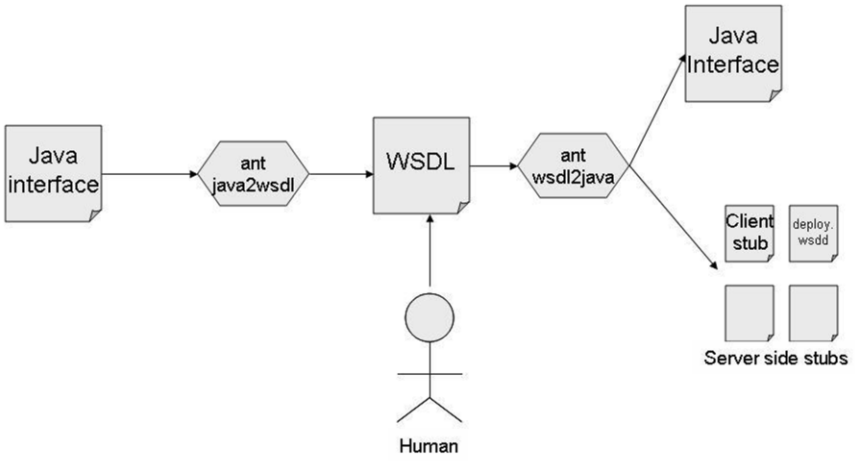


Web-palveluiden toteuttamiseen käytetään yleisesti kahta tapaa: (1) generoidaan WSDL-kuvaukset automaattisesti olemassa olevasta rajapintamäärittelystä (”Java->WSDL”) tai (2) generoidaan suunnitellusta WSDL-kuvauksesta palvelun rajapinnan toteuttavaa koodia (”WSDL->Java”).

”Java->WSDL” –tapa on näistä kahdesta helpompi ja yksinkertaisempi. Lisäksi sitä käytettäessä toteuttajan ei tarvitse välttämättä tuntea WSDL-kieltä. Toisaalta se tarkoittaa myös sitä, että Web-palvelua ajatellaan käytettävän RPC-tyyppiseen kommunikointiin: generoitu WSDL heijastelee suoraan valittua rajapintakoodia. Web-palvelukonseptia tulisi voida käyttää RPC-tyyppisen kommunikoinnin lisäksi dokumenttipohjaiseen kommunikointiin. Lisäksi RPC:n toteuttamiseen on olemassa useita tekniikoita eikä se vastaa Web-palvelukonseptin perustarkoitusta. ”Java->WSDL” –menetelmän huono puoli on myös joustamattomuus ja huono ylläpidettävyys: mikäli palvelun rajapintaan tehdään muutoksia, tulee WSDL generoida uudelleen, mikä puolestaan saattaa aiheuttaa muutoksia asiakaspäähän. Koska palvelulla on potentiaalisesti useita asiakkaista, ovat kaikki muutokset, jotka edellyttävät muutoksia myös asiakaspäähän ei-toivottuja. Versionhallinta ja ylläpidettävyys onkin yksi suurista haasteista Web-palveluille tällä hetkellä.

Toinen tapa toteuttaa Web-palveluita, ”WSDL->Java” vaatii taas enemmän palvelun toteuttajalta ja suunnittelijalta. Se vaatii esimerkiksi WSDL-kielen sekä WS-I suositusten tuntemista. Toisaalta huolellinen suunnittelu voi johtaa parempaan lopputulokseen.

Building Web Services



E.g., Java Web Services Developer Pack (Java WSDP):
Java API for XML Web Services (JAX-WS) + wsimport tool

Designing Web service clients

- Various kinds of clients may use Web services, e.g. heavy and rich client applications or light-weight clients such as wireless devices
- The client application should
  - support the communication method (e.g. SOAP/HTTP) chosen
  - locate the service(s)
  - generate messages (e.g. SOAP) from native calls
  - parse respond messages and transform them to native calls and/or map the data to its object model

Asiakasohjelman näkökulmasta Web-palvelu toimii mustana laatikkona: asiakasohjelma on kiinnostunut vain palvelun rajapinnasta, ei siitä miten se on toteutettu. Asiakasohjelmat voivat olla hyvin erilaisia kevyistä (esim. mobiililaitteet) aina monimutkaisempiin asiakassovelluksiin.

Asiakassovelluksen tulee luonnollisesti tukea palvelun tukemaa kommunikointimenetelmää. Näistä yleisin on SOAP/HTTP. Asiakassovelluksen tulee myös tarvittaessa pystyä paikallistamaan palvelu. Ratkaisuissa, joissa tiedetään mitä palveluita käytetään, ei palveluja luonnollisesti tarvitse erikseen etsiä. Lisäksi asiakassovellus toteuttaa – SOAPia käytettäessä – SOAP-arkkitehtuurin asiakaspään. Tämä tarkoittaa käytännössä ns. proxyn toteuttamista, jonka avulla asiakassovellus kommunikoi palvelun kanssa. Sen tulee esimerkiksi generoida SOAP-viestejä natiivikutsuista tai WSDL-kuvauksista ja toisaalta tulkita palvelun lähettämät SOAP-muotoiset vastaukset natiivikutsuiksi.

Designing Web service clients

- There are three methods for communication with a Web service, each of them relying on a kind of a client-side proxy that represents a Web service and is used for accessing the service's functionality:
  - static stubs (static proxies)
  - dynamic proxies
  - dynamic invocation interface (DII)
    - this is the only approach that does not rely on a (full) WSDL description at development time, but supports locating it at run-time from a service registry

Asiakasohjelmaa suunniteltaessa ja toteutettaessa ensimmäinen tehtävä on selvittää käytettävän Web-palvelun kuvauksen (WSDL) sijainti ja valita osin sen perusteella kommunikoinnin toteutustapa. Näitä tapoja on kolme. Kaikki niistä nojaavat tavalla tai toisella jonkinlaisen asiakaspään proxyn käyttöön. Proxy edustaa Web-palvelua ja sitä käytetään haluttaessa käyttää itse palvelua. Näistä tavoista ensimmäinen, ns. *staattinen tapa*, tarkoittaa sitä, että proxyn koodi tulee kääntää toteutusvaiheessa ja jopa ennen asiakassovelluksen kääntämistä. Tällä tyylillä toteutetut asiakasohjelmat saattavat mennä helpostikin toimintakyvyttömiksi mikäli palvelussa, johon ne ovat yhteydessä, tapahtuu muutoksia. Staattinen tapa on kuitenkin eri tavoista yksinkertaisin ja helpoin. Sekä staattinen että *dynaaminen tapa* edellyttää, että palvelun kuvaus (WSDL) on olemassa, sillä proxyn (dynaaminen tai staattinen) generointi tapahtuu WSDL-dokumentin tietojen perusteella. Dynaaminen tapa on kuitenkin joustavampi kuin staattinen tapa. Se tarjoaa periaatteessa saman toiminnallisuuden, mutta tekee sen dynaamisesti. Kolmas tapa, ns. *dynaaminen kutsurajapinta*, on näistä tavoista ainoa, joka sallii puhtaan SOA:n mukaisen kommunikoinnin, ts. WSDL-dokumentin etsimisen ajonaikana esimerkiksi etsimällä sitä palvelurekisteristä. Seuraavaksi tutustumme näihin tapoihin hieman tarkemmin.

Nämä asiakastyypit voivat esiintyä myös muilla nimillä, riippuen esim. työkalutuesta.

Communicating using static stubs

- Static stub classes that enable the service and the client to communicate are generated during development time
- The platform-specific stub class
  - is generated for a WSDL description prior to the client's deployment and compilation
  - represents the service endpoint interface
  - converts requests to (e.g.) SOAP messages and sends them to the service
  - converts SOAP responses back to a form understandable by the client software
    -> stub acts as a proxy for the service endpoint

Staattisessa tavassa siis myös proxyn koodi kirjoitetaan ja käännetään staattisesti ennen ajamista. Tämä koodi tyypillisesti generoidaan olemassa olevasta WSDL-kuvauksesta. Yleisimmin käytetyt työkalusetit tarjoavat siihen valmiit työkalut. Tässä on huomioitavaa siis se, että käytettävän palvelun WSDL-kuvaus tulee olla tiedossa jo asiakaspäätä toteutettaessa.

Staattinen proxy edustaa asiakaspäälle palvelun rajapintaa. Kuten edellä mainittiin, sen tulee generoida SOAP-viestit ja toisaalta sen tulee purkaa SOAP-muotoiset vastaukset asiakassovelluksen ymmärtämään muotoon. Näin ollen se toimii proxyna Web-palvelulle.

Communicating using static stubs

- Worth using when services and especially their WSDL documents are unlikely to change over time
- Easy to use (working with generated classes)
- Creates dependencies between the client and the service
  -> problems, if the service changes
- Requires that the stub classes are generated prior to the compilation of the client application

Staattista tapaa kannattaa käyttää silloin kun WSDL-kuvaukset on tiedossa ja niihin ei odoteta tulevan muutoksia. Tässä tapauksessa on siis erityisen hankalaa kaikki palvelun evoluutiosta aiheutuvat ja sen kutsurajapintaan heijastuvat muutokset. Tämä tapa on myös suhteellinen helppo toteuttajan kannalta, koska suuri osa koodista voidaan generoida automaattisesti. Toisaalta tämä tapa aiheuttaa riippuvuuksia asiakkaan ja palvelun välille. Web-palvelun perusidea on tehdä asiakas- ja palvelusovellukset mahdollisimman riippumattomiksi ja siten olla todellinen edistysaskel verrattaessa olemassa oleviin hajautusteknologioihin. Tällaiset riippuvuudet ovat aina hankalia palvelua muutettaessa. Lisäksi tämä tapa edellyttää, että proxy luokat generoidaan ennen asiakassovelluksen kääntämistä.

Communicating using dynamic proxies

- Corresponding to the ”static stub” approach, but things are carried out dynamically: instead of requiring early compilation of the stub class (prior to the compilation of the client application), an equivalent dynamic proxy is generated at runtime
- Client-side developers need a (client-side) interface that matches the service endpoint
  -> clients program to that interface
- Like in the static approach, a WSDL description is needed for proxy generation
- Portable, vendor-independent code can be written
- Java classes to serve as value types might also be needed
- Useful approach if portability is desired and if services are exptected to change only occationally
- There might be a performance overhead

Dynaaminen tapa vastaa hyvin pitkälle staattista tapaa, mutta proxyn luominen tehdään dynaamisesti ajonaikana. Tämä menetelmä edellyttää palvelun rajapintaa vastaavan asiakaspään rajapinnan olemassaoloa. Dynaaminen proxy generoidaan sitä hyödyntäen. Lisäksi saatetaan tarvita Java-luokkia, jotka edustavat käytettäviä tyyppejä. Näillä luokilla tulee olla kentille (tyypeille) omat *set*- ja *get*-metodit.

Dynaaminen tapa tuottaa luonnollisesti staattista tapaa siirrettävämpiä toteutuksia. Koska myös tämä tapa edellyttää WSDL-kuvauksen olemassaolon (ja palvelun rajapintaa vastaavan asiakaspään rajapinnan olemassaolon), saattavat palvelun muutokset tuottaa ongelmia. Ongelmat ovat kuitenkin oletettavasti pienempiä kuin staattista tapaa käytettäessä. Dynaaminen tapa tosin voi aiheuttaa hitautta ajonaikana juuri sen dynaamisuudesta johtuen.

Communicating using DIIs

- A client can call a service without knowing the exact service name (operations) and address at compile time
- More difficult for a developer to use, since a more complex interface (compared to static stubs and dynamic proxy approaches) needs to be used
  - This interface is more prone to class cast errors
- Performance overhead
- Useful when a complete WSDL document is not available
- Useful when the service is expected to be changed frequently

Viimeinen asiakaspään toteutustapa, dynaaminen kutsurajapinta, on näistä kolmesta eri tavoista joustavin mutta myös haastavin ja monimutkaisin. Käytettävä rajapinta on huomattavasti monimutkaisempi verrattaessa staattiseen ja dynaamiseen tapaan. Lisäksi ko. rajapinta on käytännössä alttiimpi tyyppimuunnoksista aiheutuville ongelmille.

Dynaaminen kutsurajapinta on ainoa tapa, joka toteuttaa SOA:n puhtaasti: se on tavoista ainoa, joka sallii palvelun kutsumisen ilman, että sen rajapinta (tarjotut operaatiot) ja osoite on tiedossa käännösaikana. Näin ollen se sallii esimerkiksi palvelun etsimisen ajonaikana palvelurekisteristä.

Dynaaminen kutsurajapinta on puhtaasti dynaaminen tapa: se ei edellytä lainkaan käytettävään palveluun liittyvän koodin luomista asiakasohjelmaa luotaessa. Tässäkin tapauksessa – tai erityisesti tässä tapauksessa – tehokkuusongelmia saattaa ilmetä varsinaisia palvelukutsuja tehtäessä. Tämä tapa on erityisen hyödyllinen silloin, kun palvelun rajapinnan oletetaan muuttuvan usein. Lisäksi tämä on ainoa tapa, joka ei edellytä täydellisen WSDL-kuvauksen olemassaoloa.

Some tools and tutorials

- Java Platform, Enterprise Edition (Java EE): Java EE Development Kit, Oracle GlassFish Server 3.1.2.2
  - http://www.oracle.com/technetwork/java/javaee/downloads/index.html
- Eclipse Web Tools Platform (WTP)
  - http://www.eclipse.org/webtools/
- Host a Soap Web Service on Google App Engine with JAX-WS
  - http://www.redheap.com/2014/04/host-soap-web-service-on-google-app-engine.html