Agenda today

Service design,
recap

Service design

TIE-23600 Palvelupohjaiset järjestelmät

Service-oriented architecture (SOA)

- Service-oriented architecture (SOA) is an architectural paradigm to create software based on the interaction of loosely coupled services
- A service is a piece of software that
  - is autonomous
  - performs a distinct business function
  - is defined by an implementation-independent interface, i.e. service contract

3

Tässä kertauksena SOA ja palvelu.

Eri lähteet esittävät erilaisia vaatimuksia SOA-järjestelmän osasille eli palveluille. Yleisimpiä ja tärkeimpiä ovat autonomisuus, löyhä sidonta, toteutusriippumaton rajapinta ja korkea abstraktiotaso. Tässä kalvosetissä tutustutaan tarkemmin näihin ja muihin palveluiden ominaisuuksiin.

Service design principles

- In his SOA book series, Thomas Erl defines eight principles of service design
  - Standardize service contract
  - Service loose coupling
  - Service abstraction
  - Service reusability
  - Service autonomity
  - Service statelessness
  - Service discoverability
  - Service composability
- Other sources have similar service design principles

Thomas Erl on kirjoittanut SOA:sta useita kirjoja, joissa muun muassa määritellään kahdeksan palveluiden suunnitteluperiaatetta.

Suunnitteluperiaatteita ei ehkä kannata noudattaa aina sataprosenttisesti, vaan hieman tilanteen mukaan. Ne ovat myös jonkin verran epämääräisiä; ei ole mitään tarkkaa mittaria esim. sille, milloin palvelu on löyhästi sidottu tahi uudelleenkäytettävä. Lisäksi ne ovat jossain määrin ristiriidassa toistensa kanssa; joudutaan tekemään kompromisseja sen suhteen kumpaa periaatetta painotetaan.

Joka tapauksessa ne ovat hyväksi havaittuja nyrkkisääntöjä suurten hajautettujen järjestelmien suunnittelussa. Käydään niitä tarkemmin läpi seuraavilla kalvoilla.

Standardize service contract

- "Services within the same service inventory are in compliance with the same contract design standards."
  - Service inventory = a collection of complementary services
- Harmonization of how service contracts are defined
  - How to express functionality
  - How to define data types
  - How to deal with policies
  - etc.
- Improves interoperability

Standardoitu palvelusopimus -suunnitteluperiaatteen tavoitteena on parantaa palveluiden yhteentoimivuutta

Tarkempi määritelmä palveluinventoriolle (service inventory):
"An independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise."

Loose coupling

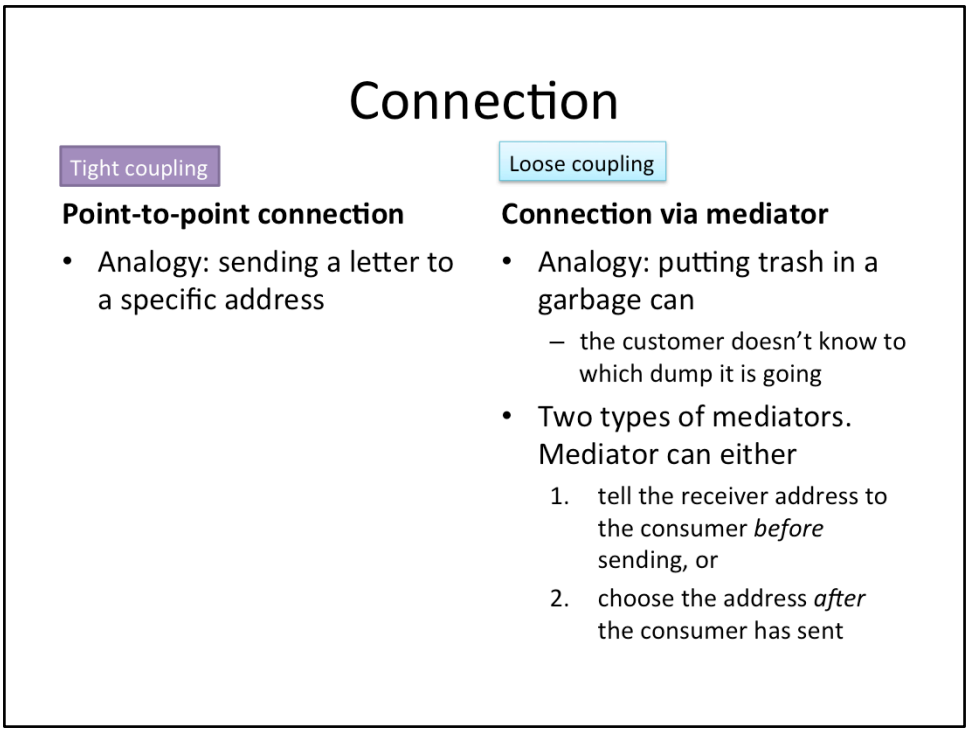
- “Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.”
- The aim of loose coupling is to minimize dependencies
- Loose coupling allows parts of the system to be developed independently of each other
- Loose coupling also has its drawbacks
  - Performance
  - (Complexity)

Some forms of loose coupling

	Tight coupling	Loose coupling
Connection	Point-to-point	Via mediator
Communication style	Synchronous	Asynchronous
Data model	Common complex types	Simple common types only
Interaction pattern	Navigate through complex object trees	Data-centric, self-contained messages
Control of process logic	Central control	Distributed control
Binding	Static	Dynamic
Platform	Strong platform dependencies	Platform independent
Transactionality	2PC (two-phase commit)	Compensation
Deployment	Simultaneous	At different times
Versioning	Explicity upgrades	Implicit upgrades

Nicolai M. Josuttis: SOA in Practice

Tässä joitain osa-alueita, joilla voidaan tehdä valintoja löyhän ja tiukan sidonnan välillä. Taulukko kirjasta Nicolai M. Josuttis: SOA in Practice.

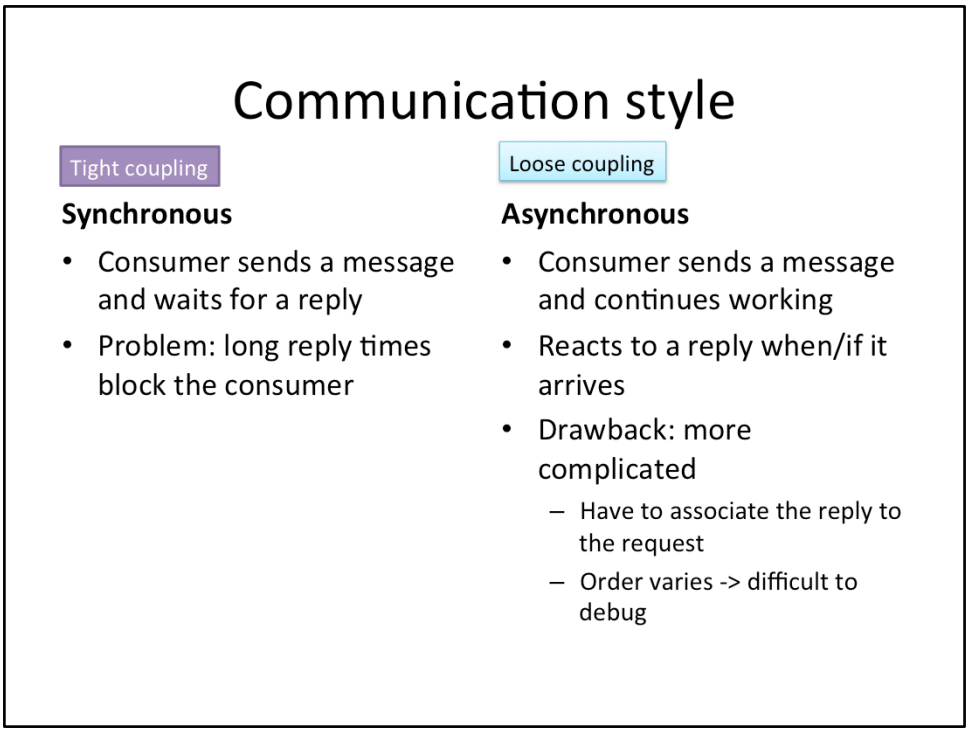


Erilaisilla yhteydenottotavoilla voidaan luoda eriasteista riippuvuutta lähettäjän ja vastaanottajan välillä.

Point-to-point –yhteydenottotavassa lähettäjä tietää etukäteen vastaanottajan osoitteen. Kuin kirjeen lähettäminen: lähettäjän täytyy kirjoittaa vastaanottajan osoite kuoreen.

Löyhempi sidos lähettäjän ja vastaanottajan välillä saadaan aikaan käyttämällä välittäjää (mediator).
Välittäjät voidaan jakaa kahteen perustyyppiin:

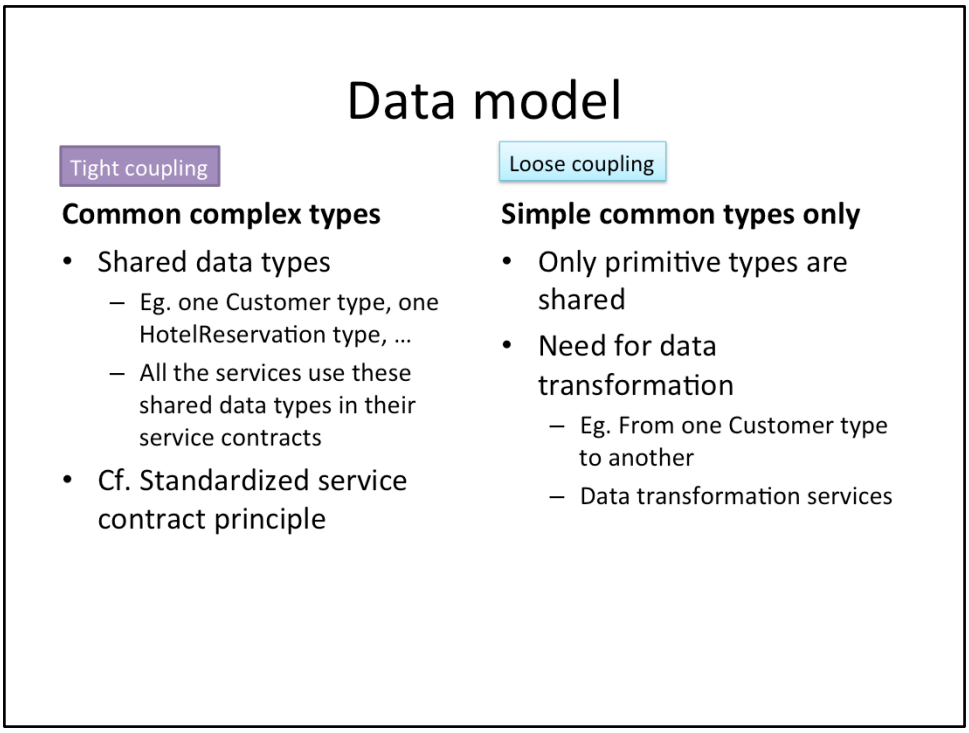
1) Välittäjä määrää osoitteen ennen lähetystä. Tässä tapauksessa lähettäjä kysyy välittäjältä vastaanottajan osoitetta. Välittäjä kertoo osoitteen, minkä jälkeen lähettäjä lähettää viestin. Tässäkin siis lähettäjä kyllä tietää vastaanottajan osoitteen, mutta saa sen selville vasta juuri ennen lähetystä. Esimerkiksi DNS – nimipalvelimet toimivat tähän tapaan.
2) Välittäjä määrää osoitteen lähetyksen jälkeen. Tässä asiakas antaa viestin välittäjälle liitettyään mukaan tarpeeksi tietoa, jotta viesti osataan ohjata perille.



Kommunikointi asiakkaan ja palvelun välillä voi olla synkronista tai asynkronista.

Synkroninen tapa on yksinkertaisempi. Huonona puolena siinä on, että asiakas joutuu odottamaan vastausta tekemättä mitään. Ei sovellu hyvin tilanteisiin, joissa vastauksen saaminen voi kestä kauan.

Asynkronisessa kommunikaatiossa asiakas jatkaa toimintaansa viestin lähettämisen jälkeen ja reagoi sitten sopivalla tavalla saapuneeseen vastaukseen. Huonona puolena tässä on lisääntynyt monimutkaisuus. Asiakkaan täytyy tietää, mikä vastaus liittyy mihinkin pyyntöön. Vastaukset saapuvat mielivaltaisessa järjestyksessä, joten debuggaus on vaikeaa.



Palveluilla täytyy olla jonkinlainen yhteinen käsitys käytettävistä tietotyypeistä, vähintään primitiivistä tyypeistä kuten string ja int. Se kuinka suurelta osin tietotyypit harmonisoidaan eri palveluiden välillä on suunnitteluratkaisu, jossa on tehtävä tradeoffeja.

Käyttävätkö esim. kaikki palvelut samanlaista Asiakas-tietotyyppiä, vai saavatko kaikki määritellä omansa. Jos Asiakas-tietotyyppi on määritelty keskitetysti ja siihen lisätään myöhemmin esim. puhelinnumero-kenttä, on muutoksia tehtävä kaikkiin palveluihin, jotka käyttävät Asiakas:ta rajapinnassaan.

Toisaalta jos Asiakas-tietotyyppiä ei ole harmonisoitu järjestelmän laajuisesti joudutaan tekemään muunnoksia tyypistä toiseen kun käytetään eri palveluita.

Type system

Tight coupling

Strong type checking

- Detect errors early
- Data type correctness can be checked by the SOA infrastructure before reaching the service

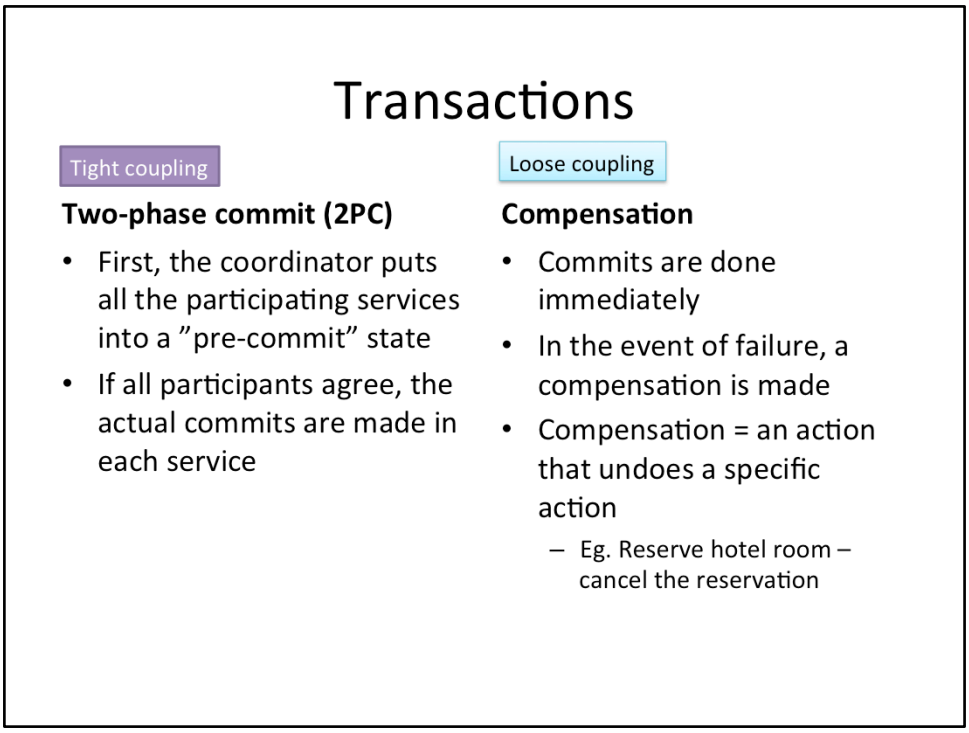
Loose coupling

Weak type checking

- More flexible
- ”Duck typing”
- Errors are detected later
  - Debugging more difficult

Interaction pattern

Control of process logic



Esimerkki: matkatoimisto tilaa lentolipun ja hotellihuoneen eri palveluista. Molempien varausten pitää onnistua tai sitten kumpaakaan ei tehdä.

On ainakin kaksi tapaa toimia: 2PC ja kompensaatio (on muitakin).

2PC: luodaan hajautettu transaktio, jossa ”koordinoija” asettaa kaikki osallistuvat palvelut ensin tilaan, jossa ne ovat valmiita committiin. Jos kaikki palvelut onnistuivat tässä, suoritetaan varsinainen commit kussakin palvelussa.

Kompensaatio: commit tehdään heti. Jos joku palveluista epäonnistui, suoritetaan kaikissa palveluissa ns. kompensoiva operaatio. Esim. hotellihuoneen varausta kompensoiva operaatio on varauksen peruminen. Jokaiselle palvelun operaatiolle on siis oltava myös kompensoiva operaatio.

2PC: http://en.wikipedia.org/wiki/Two-phase_commit_protocol
Compensation: http://www.oracle.com/technetwork/articles/soa/ind-soa-7-service-comp-2005463.html

Binding

Tight coupling

Static (early) binding

- The service to be used is decided at design time

Loose coupling

Dynamic (late) binding

- The service to be used is decided at run time

Service autonomity

- "Services exercise a high level of control over their underlying runtime execution environment."
- Design-time autonomity
  - Services control their execution logic
- Run-time autonomity
  - Services control their execution environment

Service abstraction

- “Service contracts only contain essential information and information about services is limited to what is published in service contracts.”
- Services abstract unessential information about technology, logic and function
- Everything that is not defined in the service contract is hidden

Jotkin muut palveluiden suunnitteluperiaatteet (reusability, composability, …) vaativat, että palvelun rajapinnassa on saatavilla paljon informaatiota. Abstraktio-periaate rajoittaa tätä. Sen mukaan palvelun on julkaistava palvelusopimuksessa vain olennaiset tiedot, mitä se sitten minkin palvelun tapauksessa tarkoittaa. Ja kaikki mitä palvelusopimuksessa ei sanota, piilotetaan ulkopuolisilta.

Palvelusopimuksen abstrahoinnissa joudutaan tekemään kompromisseja.

Roskaesimerkki: asiakkaan kannalta olisi yksinkertaisinta jos jäteyhtiön ”rajapinta” olisi vain yksi laatikko, jonne kaikki roskat voi heittää. Yleensä jäteyhtiöt kuitenkin tarjoavat monimutkaisemman rajapinnan, jossa on erilaisia laatikoita erilaisia roskia varten. Tällöin jäteyhtiön sisäinen jätteidenkäsittelyn toteutus siis hieman ”vuotaa” ulos. Tässä ”epäabstraktimmassa” ratkaisussa on toki hyviä puolia (roskat saadaan näin paremmin kierrätettyä, jätteiden lajittelu yhdestä isosta laatikosta olisi jälkikäteen kallista, …)

Samantapaista tradeoffia voidaan joutua tekemään ohjelmistopalveluidenkin suunnittelussa. Esim. olisi asiakkaan kannalta helppoa ja muutenkin kaunista, että lentolippupalvelulla on yksi rajapintakutsu, jolla saa kaiken tiettyyn tilaukseen liittyvän tiedon. Voi kuitenkin olla esim. sellainen tilanne, että palvelu joutuu hakemaan istumapaikan tiedot jostain hitaasta, kalliista ja huonosta alipalvelusta. Lisäksi asiakkaat tarvitsevat istumapaikkatietoa vain harvoin. Tällöin voi olla järkevää tehdä erillinen operaatio istumapaikan hakua varten, jota asiakkaat kutsuvat vain silloin kun sitä tarvitsevat.

Kuvat varastettu osoitteista
http://www.jokimaki.com/main.site?action=siteupdate/view&id=16

Service reusability

- "Services contain and express agnostic logic and can be positioned as reusable enterprise resources."
- Agnostic functional context
  - Separate multi-purpose logic to separate services (vs. single-purpose logic)
- Generic service logic
- Generic and extensible service contract
- Support for concurrent access

"Agnostic functional context" tarkoittaa, että palvelu ei välitä (eli on agnostinen sen suhteen) missä kontekstissa sitä suoritetaan. Palvelun pitäisi siis olla sellainen, että sitä voi käyttää mahdollisimman monessa tilanteessa. Tätä varten pitäisi pystyä eriyttämään moneen käyttötarkoitukseen soveltuva (multi-purpose logic) logiikka omaksi palvelukseen, ei niin että se olisi osa jotain yhteen tiettyyn tarkoitukseen suunniteltua palvelua.

Palvelun pitäisi olla sopivan geneerinen. Tässäkin joudutaan tekemään tradeoffia sen suhteen, että kuinka helppoa palvelua on käyttää (service discoverability principle) vs kuinka geneerinen se on.

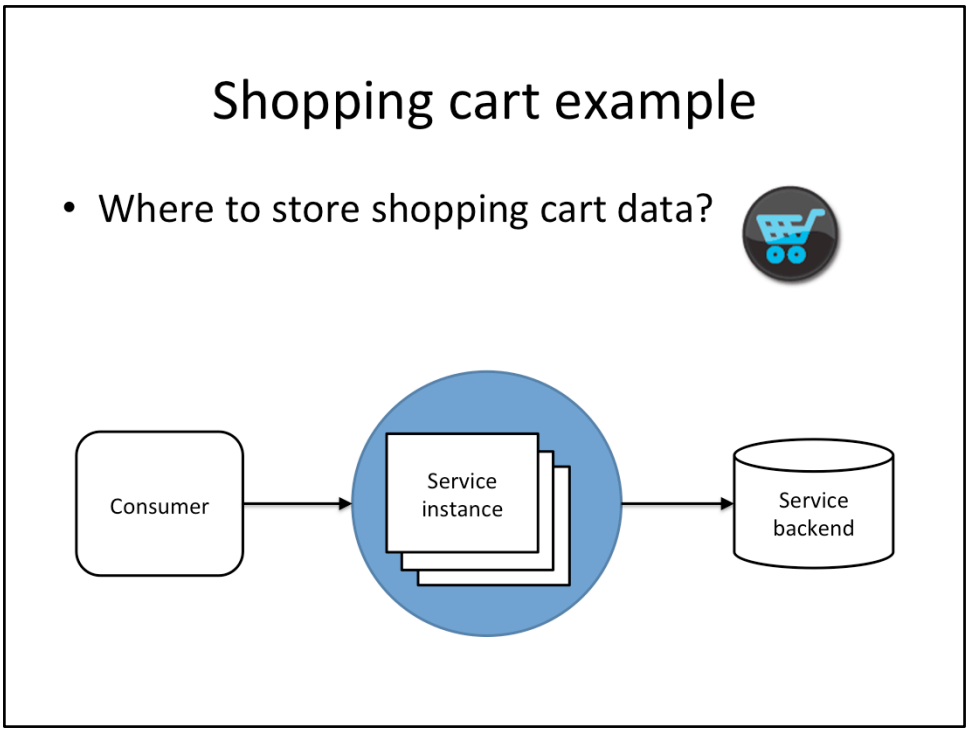
Service discoverability

- "Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."
- Service contracts are accompanied with metadata
  - For humans
  - For computer programs

Service statelessness

- "Services minimize resource consumption by deferring the management of state information when necessary."
- Each call to service is independent
  - Service doesn't store state information between calls
  - State information = information that grows as the number of concurrent customer grows
- Improves scalability

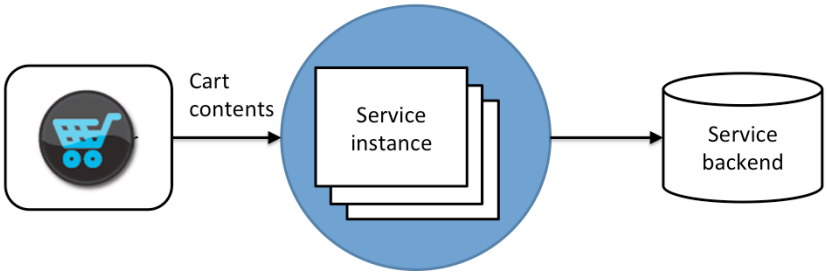
Palvelun tilattomuudella tarkoitetaan sitä, että jokainen sille tuleva kutsu sisältää kaiken kutsun käsittelemiseen vaaditun tiedon. Palvelun ei siis tarvitse tietää mitä sama asiakas on tehnyt aiemmin.



Tässä ostoskori esimerkkinä tilatiedosta, jonka tallentamiseen on useita tapoja.

Shopping cart example

- Shopping cart state is stored in the customer
- Service is stateless

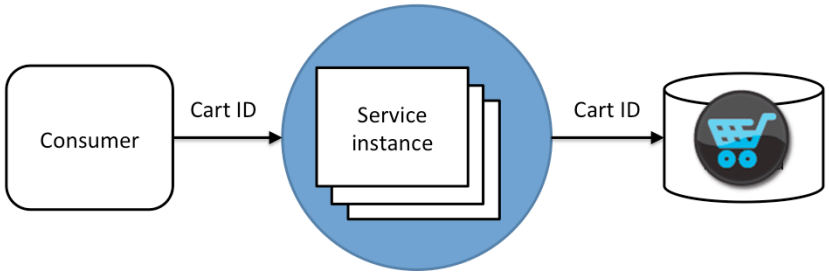


Shopping cart example

- Shopping cart state is stored in the service
- Service is stateful
- Consequent calls have to be addressed to the same service instance (eg. thread)

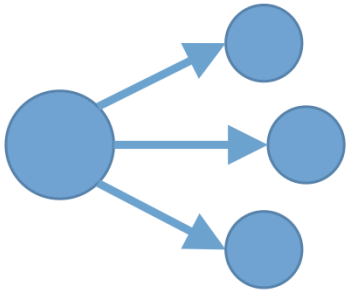
Shopping cart example

- Shopping cart state is stored as data on the backend (database)
- Service is stateless
  - …or is it? (technical state vs logical state)



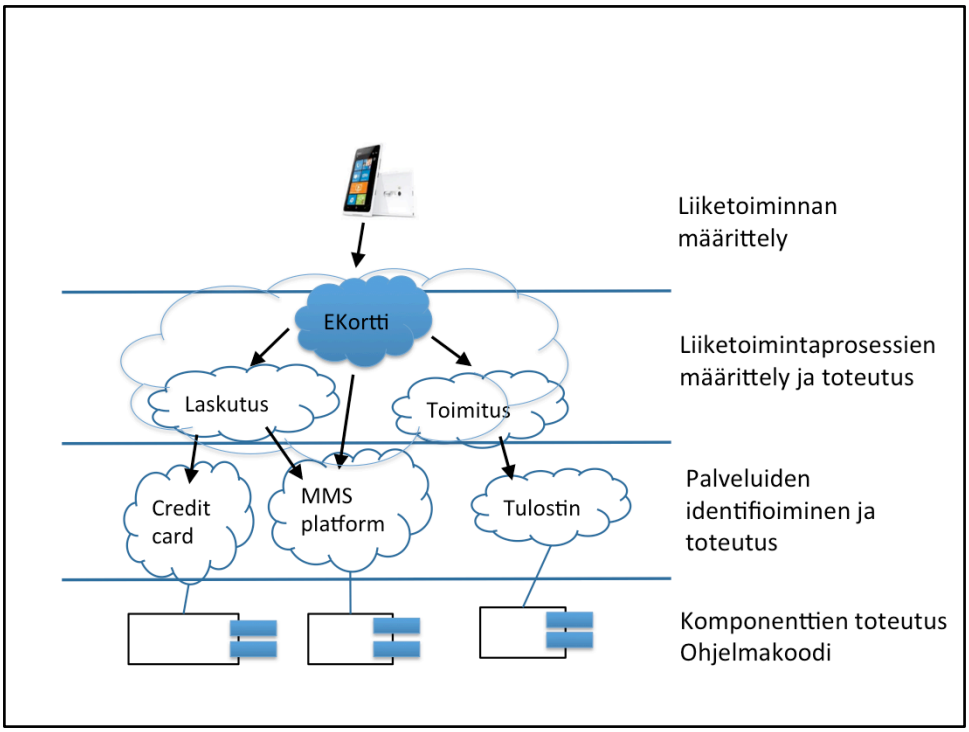
Service composability

- "Services are effective composition participants, regardless of the size and complexity of the composition."
- Closely related to reusability
  - The "reusers" are often other services
- Controller services
  - May remain stateful while the composition members do work



Designing and realizing SOA

TIE-23600 Palvelupohjaiset järjestelmät


EKortti
Laskutus
Toimitus
Credit card
MMS platform
Tulostin
Liiketoiminnan määrittely
Liiketoimintaprosessien määrittely ja toteutus
Palveluiden identifioiminen ja toteutus
Komponenttien toteutus Ohjelmakoodi

Top-down approach

- Design of services and processes
  - Ideally without considering the existing code base
  - i.e. what would be needed
- Implementation of the required services
  - Match with existing code base
    - Modify/use existing services and use wrapping techniques
    - Build new services
- A ”business driven approach”

Edellä esitetty tapa toteuttaa palvelupohjaisia järjestelmiä edustaa nk. ”top-down” – lähestymistapaa. Oleellisesti siinä siis edetään systemaattisesti abstrakteimmalta tasolla tarkentaen yhä yksityiskohtaisemmalle ja tarkemmalle tasolle päätyen lopulta itse toteutukseen. Ideaalisessa tapauksessa suunnittelu alkaa liiketoiminnan tasolta eikä liiketoimintaprosesseja eikä teknisiä realiteetteja vielä edes huomioida tuossa vaiheessa. Kun liiketoimintaprosessit on määritelty, tulee niiden toteuttaminen suunnitella teknisinä prosesseina. Nämä tekniset prosessit voidaan sitten toteuttaa osin hyödyntäen olemassa olevia järjestelmiä ja toisaalta toteuttamalla uusia palveluita tarvittaessa. Tätä lähestymistapaa kutsutaan myös ”liiketoimintaorientoituneeksi” lähestymistavaksi.

Bottom-up approach

- Identification of services and processes from existing code
- Using e.g. wrapping techniques to implement proper service interfaces
- Goals
  - to extend the functionality of a legacy system
  - to migrate a legacy system for a new environment
  - For software integration and/or interoperability reasons
  - Etc.
- An ”IT driven approach”

Edelliselle ”top-down” –lähestymistavalle vastakkainen vaihtoehto on nk. ”bottom-up” –lähestymistapa. Siinä lähdetään nimenomaan olemassa olevista teknisistä valmiuksista. Aluksi pyritään identifioimaan tarvittavat palvelut ja prosessit koodista. Hyödyntämällä erilaisia käärimistekniikoita ja takaisinmallinnustekniikoita (reverse engineering techniques), olemassa oleva koodi muunnetaan palvelupohjaiseksi.

Tätä toiselta nimeltään ”IT-lähtöistä” lähestymistapaa käytetään paljon. Sitä hyödynnetään mm. kun halutaan laajentaa legacy-järjestelmien toiminnallisuutta, integroida ne muiden järjestelmien kanssa, käyttää niitä uudessa ympäristössä jne. Toisaalta tässä lähestymistavassa on myös selkeät heikkoutensa. Suurin niistä lienee se, että muodostettu palvelujoukko on harvoin ideaalinen ja aidosti ”SOA-hengen mukainen”.

Meet-in-the-middle approach

- In practice, approach is rarely neither strictly top-down nor strictly bottom-up
- Strict top-down: may lead to services that are single-purpose, nonreusable and difficult to implement
- Strict bottom-up: may lead to services that are not needed or don't fulfill business requirements

G. Coticchia, Seven Steps to a Successful SOA Implementation. *Business Integration Journal,*10, 5, 2006, http://www.bijonline.com/, pp. 10-13. :

> *"…Business services cannot be developed bottom-up, ad hoc, in other words driven by immediate project needs and implementation details instead of long-time business needs. But one-step top-down approach is not either the only way to go, as it tends to ignore the details and make false assumptions on the cost and scheduling of implementations. Instead, collaborative, iterative top-down mechanism should be used."*

Käytäntö on osoittanut, että ehkä paras edetä olisi yhdistää top-down ja bottom-up –lähestymistapoja. Bottom-up –lähestymistapa saattaa johtaa joustamattomaan palvelujoukkoon eikä huomioi liiketoimintatarpeita. Top-down –lähestymistapa puolestaan saattaa johtaa siihen, ettei teknisiä realiteetteja oteta oikealla tavalla huomioon. Esimerkkeinä voivat olla väärät oletukset toteutuksen kompleksisuudesta, toteutukseen tarvittavasta ajasta jne. Coticchia (2006) on todennut tutkittuaan useaa käytännön projektia, että nk. "meet-in-the-middle" olisi tässäkin tapauksessa paras vaihtoehto. Hän toteaa, että suunnittelun tulisi alkaa aidosti liiketoiminnan näkökulmasta, mutta tekniset realiteetit tulisi huomioida jo melko aikaisessa vaiheessa. Tämä lisäisi myös kommunikaatiota liiketoiminta- ja IT-vastuullisten kesken.

Developing SOA-based systems

- Harry Sneed, ”Migrating to Web Services – A Reseach Framework”, CSMR conf., SOAM workshop, 2007
  - Developing web services is a long term investment [1]. It will take at least two years before enough services of sufficient quality can be made available to the user business processes. It is questionable if the users are willing to wait so long. The benefits of using self-developed web services will only become visible after some years. In the meantime, the IT department must sustain the continuity of the existing systems…

[1] J. Bishop and N. Horspool, “Cross-Platform Development – Software that lasts”, *IEEE Computer*, Oct. 2006, p. 26

Toisaalta bottom-up –lähestymistalle löytyy myös puollustajia, kuten esim. Harry Sneed on esittänyt useissa artikkeleissaan. Kyse lienee kuitenkin lopulta siitä mitä palvelupohjaiselta järjestelmältä halutaan ja mitä tarkoitusta varten se on toteutettu.

Service granularity

- Granularity layers
  - Typically, SOA systems form a hierarchy, such that higher level services make use of lower level services
  - Services at the lowest level are fine grained providing resources and infrastructure services
  - High level services are coarse grained business services used by the business processes and end users
- Service granularity
  - The service size and the scope of functionality provided by a service within one interaction
  - The services should have the right granularity to accomplish a proper unit of work and to enable service reusability and composability

Service Layers

- Utility services
  - Non-business-centric functionality
- Entity services
  - ”Functional context derived from business entities”
    - Eg. Customer, Order, Invoice, …
- Task services
  - ”Functional context based on a specific business process”
  - Not very reusable

Yksi tapa hahmottaa paremmin sitä, millaisia palvelut ovat on jakaa niitä kerroksiin. Tässä yksi jakotapa.

”Apupalvelu” (utility service) tarjoaa toimintoja, joilla ei ole suoraa linkkiä liiketoimintavaatimuksiin. Muut palvelut ovat apupalveluiden asiakkaita.

”Entiteettipalvelu” (entity service) tarjoaa toimintoja, jonkin ”entiteetin” käsittelyyn. Tämä entiteetti on sellainen, että se on käsitteenä olemassa ohjelmistomaailman ulkopuolellakin. Esim. Asiakas, Tilaus, Lasku, …

”Tehtäväpalvelu” (task service) kuvaa jonkin liiketoimintaprosessin tai sen osan. Käyttää hyväkseen apu- ja entiteettipalveluita. Esim. palvelu joka luo laskutusraportin.

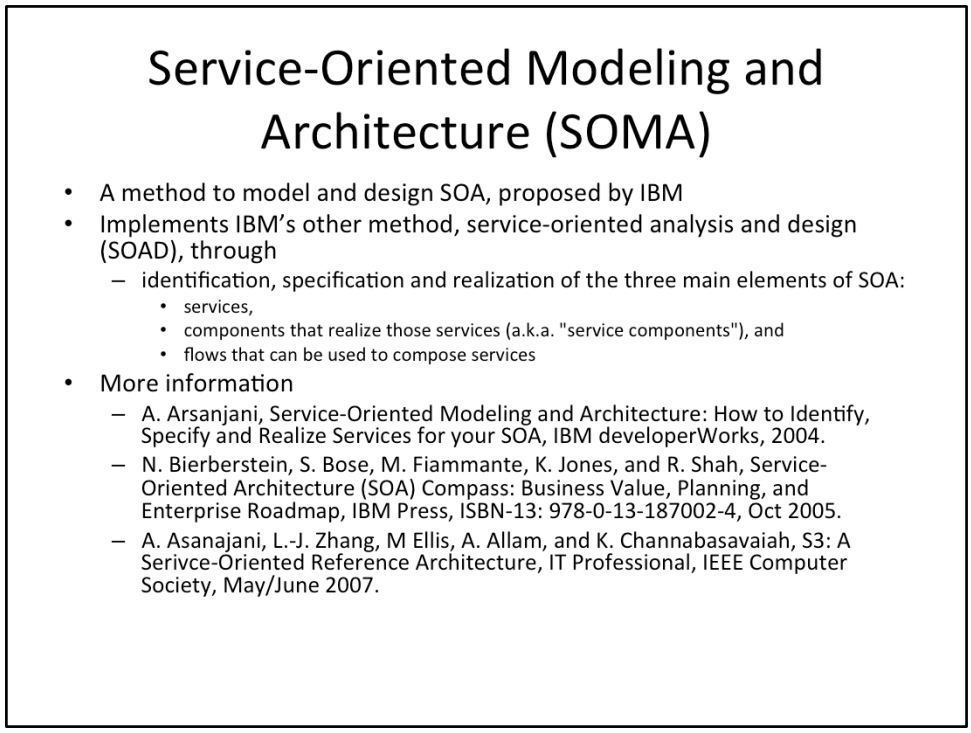
http://soapatterns.org/design_patterns/service_layers

http://serviceorientation.com/soaglossary/entity_service
http://serviceorientation.com/soaglossary/utility_service
http://serviceorientation.com/soaglossary/task_service

Another way to classify services

- Basic services
  - Basic data services
    - Often have ACID properties
  - Basic logic services
- Composed services
  - Use basic services and other composed services
- Process services
  - Present long-term workflows or business processes

Nicolai M. Josuttis: SOA in Practice



IBM:n SOMA on menetelmä, jonka tarkoituksena on tukea SOA-pohjaisten järjestelmien suunnittelua, mallintamista ja toteuttamista. Koska se on yksi ensimmäisistä ja ehkä myös laajimmin tunnetuista suunnittelumenetelmistä ja koska tällaisten menetelmien tarve (ja niiden puute) on laajalti havaittu, esitellään setässä esimerkinomaisesti. On syytä korostaa, että SOMAa ei menetelmänä voida missään nimessä kutsua (edes de facto) standardiksi.

SOMA toteuttaa IBM:n toisen menetelmän SOAD, joka on kehitetty palveluorientoituneiden järjestelmien analyysi- ja suunnittelumenetelmäksi. Tämä menetelmä pohjautuu perinteisiin olio- ja komponenttipohjaisten järjestelmien analyysi- ja suunnittelumenetelmiin ja laajentaa niitä SOAn kannalta oleellisilla näkökulmilla. SOMA koostuu kolmesta päävaiheesta: SOAn peruselementtien *tunnistaminen*, *spesifiointi* ja *realisointi*. Nämä peruselementit ovat *palvelut*, palvelut realisoivat *palvelukomponentit* (service components) sekä palveluiden yhdistämiseen käytettävät *vuot*.

Designing and implementing Web Services

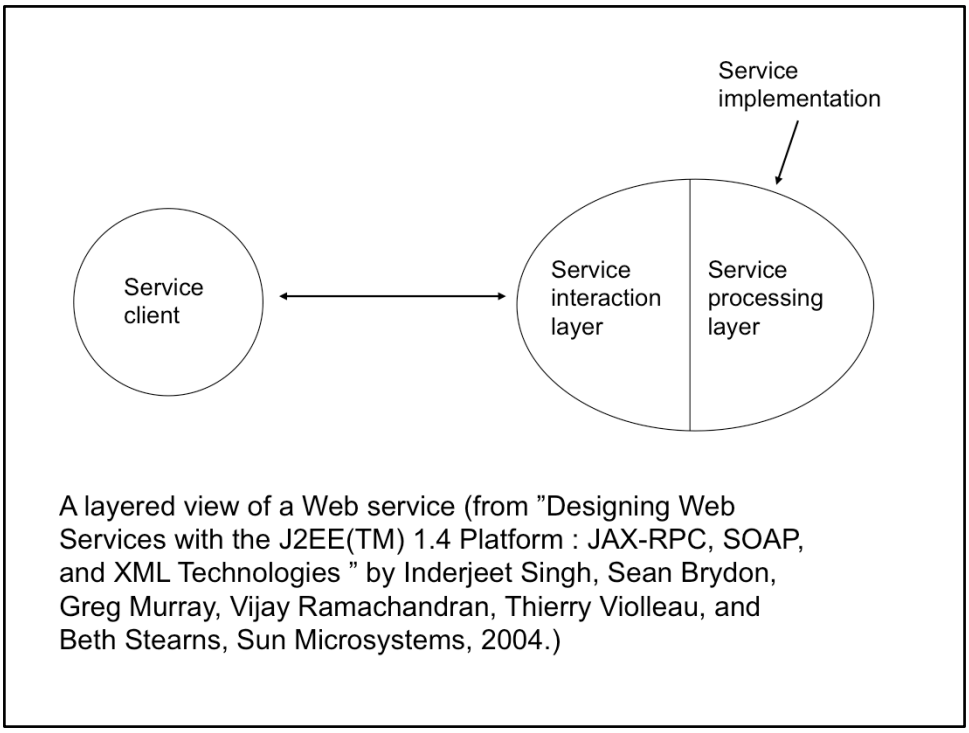
TIE-23600 Palvelupohjaiset järjestelmät

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.

Developing Web service interfaces

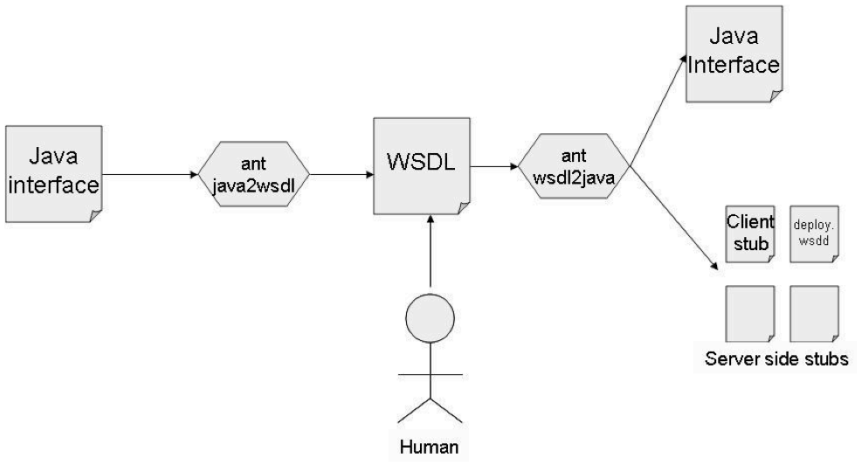
- Java (or some other language) -> WSDL
  - generating WSDL documents from an existing set of Java interfaces for the Web service
  - An easy approach
  - One need not to be aware of WSDL details
  - The developer looses some control over WSDL file creation
  - Comes with the cost of less flexibility
    - may be hard to evolve the service interface without forcing a change in the corresponding WSDL document,
      -> changing the WSDL might require rewriting the service's clients
- WSDL -> Java
  - requires more from the designer, including more knowledge of WSDL and WS-I requirements

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

Legacy systems as services

TIE-23600 Palvelupohjaiset järjestelmät

Legacy systems

- Old, valuable systems that are often designed and implemented using methods and programming languages that are no longer in use
  - ”legacy” means something valuable and something inherited
- There is a growing need to access legacy systems in current net-centric applications and Web services

Legacy-järjestelmällä tarkoitetaan (mahdollisesti) vanhaa, olemassa olevaa ja käyttökelpoista ohjelmistoa, joka on toteutettu käyttäen vanhoja menetelmiä ja/tai ohjelmointikieliä, joiden tuntemus yrityksessä saattaa olla puutteellista. Sana ”legacy” itsessään tarkoittaa jotain perittyä ja arvokasta. Nämä ominaisuudet pätevät myös legacy-järjestelmiin.

Legacy-järjestelmien modifiointi on usein erittäin vaikeaa eikä useinkaan kannattavaa. Syitä tähän on monia:
-järjestelmät on toteutettu vanhoilla ohjelmointikielillä
-suurella todennäköisyydellä ko. järjestelmien arkkitehtejä, suunnittelijoita ja koodaajia on vaikea tavoittaa (sikäli kun käytettyjen ohjelmointikielten osaajia ylipäätään on mahdollista löytää)
-mikäli modifiointeja tarvitaan, niin ne (erityisesti arkkitehtuuritason muutokset) käytännössä vaativat niin paljon työtä, että usein voi olla helpompaa rakentaa uusi järjestelmä
-usein ”legacy-järjestelmän statuksen” saavuttaneet järjestelmät toimivat kohtuullisesti (koska niiden elinikä on osoittautunut odotettua pidemmäksi) eikä suuret rakenteelliset tai toiminnalliset muutokset ole aina välttämättömiä.
-esimerkiksi proseduraalisten ohjelmien konvertoiminen oliopohjaisiksi on osoittautunut usein huonoksi ideaksi: vaikka uusi oliopohjainen ratkaisu toimisikin, (a) se saattaa olla hitaampi tai (b) ohjelmoijat yksinkertaisesti hylkäävät uuden

Legacy systems

- It has proven to be often too costly, too risky, and too time consuming to reimplement the legacy systems from scratch.
- Wrapping the legacy system is often the best solution for enabling its reuse, e.g. transforming it to a Web service
  - minimal changes to the source
  - enables the reuse of current functions in new applications
  - isolates the particular implementation from the user of the function/feature

Edellä mainituista syistä legacy-järjestelmien korvaaminen uusilla tai muilla olemassa olevilla järjestelmillä on usein varsin riskialtista. Lisäksi erityisesti uuden järjestelmän rakentaminen on usein hyvin aikaa vievää ja näin ollen kallista. Tämän vuoksi se ei aina ole edes realistinen vaihtoehto.

Vanhoja legacy-järjestelmiä ei näin ollen käytännössä haluta useinkaan heittää pois vaan ne halutaan tavalla tai toisella käytettäväksi uudessa muuttuneessa ympäristössä. Usein paras tapa on kääriä (wrap) ne. Kääriminen edellyttää vain hyvin suhteellisen pieniä muutoksia ko. ohjelmaan, sallii olemassa olevan toiminnallisuuden ja toteutuksen käytön ja erottaa toteutetut ominaisuudet niiden käyttäjistä. Käärimistekniikoita käsitellään tarkemmin kurssilla “Ohjelmien ylläpito ja evoluutio”.

Requirements and expectations for wrapping

- Legacy systems/applications must include functions with high business value and good technical quality (e.g. robustness)
- The systems and their functionalities must be bounded, and they should have manageable APIs
- The system to be wrapped should have (enough) expected value in the future
- Wrapping should be worth the effort, compared to
  - re-engineering the system
  - building a new system from the scratch
  - replacing the system with another alternative

Jotta kääriminen olisi varteenotettava vaihtoehto, edellytetään myös käärittävältä ohjelmalta tiettyjä ominaisuuksia. Käärittävän ohjelman tulee ensinnäkin olla "käärimisen arvoinen". Sillä tulee toisin sanoen olla tarpeeksi arvokkaita (esimerkiksi luotettavuuden tai tehokkuuden näkökulmasta) ominaisuuksia, joita halutaan käyttää jatkossakin. Käärittävän ohjelmalla tulee myös olla riittävän hyvä ja käytettävä API ja käytettävät ominaisuudet tulee olla selkeästi identifioitavissa ja kutsuttavissa ko. APIa käyttäen. Näiden lisäksi verrattaessa vaihtoehtoisiin ratkaisuihin, kuten ohjelman uudelleen kirjoittaminen tai sen korvaaminen toisella ohjelmalla, käärimisen tulee paras ratkaisu arvioitaessa työmäärää ja hyötyä sekä niiden suhdetta (cost-benefit).

Wrapping considerations

- There might be a need for not corresponding wrapper and application interfaces
  - e.g. when data structures that client uses are not isomorfic with the data structures legacy system uses
  - e.g. when providing a higher level wrapper interface
    - a wrapper's operation might be decomposed to several application calls
- XML wrapping might cause some overhead but is flexible
- Use design patterns, e.g., Wrapper, Adapter, Facade, Bridge

Käärimisen ei välttämättä aina haluta tarjoavan täsmälleen vastaavaa rajapintaa kuin alkuperäinen ohjelma, erona vain käytettävä kieli (esim. SOAP-rajapinta C++ -ohjelmalle). Voidaan esimerkiksi haluta, että uuden rajapinnan kutsuttava operaatio käyttää legacy-ohjelman useampaa operaatiota. Tällöin rajapinnassa – tai tarkemmin ottaen interaktiokerroksessa – voidaan käyttää ns. *mukauttajia* tai *välittäjiä,* jotka tekevät tarvittavat muutoksen. Mukauttajien käyttöä edellyttää myös tilanteet, joissa asiakasohjelman käyttämät tietorakenteet voivat erota legacy-ohjelman käyttämistä tietorakenteista.

Kääreen erottaminen legacy-ohjelmasta - tai yleisemmin interaktiokerroksen erottaminen liiketoimintalogiikasta – kannattaa suunnitella joustavaksi ratkaisuksi. Tähän tarkoitukseen voi käyttää esimerkiksi Façade, Wrapper, Adapter ja Bridge suunnittelumalleja.

XML-kääre aiheuttaa käytännössä hitautta, koska se esimerkiksi edellyttää aina myös XML-jäsentäjän käyttöä. Toisaalta XML-kääreellä on etunsakin. Käärittäessä ohjelma esimerkiksi Internetiä hyödyntäväksi Web-palveluksi, voidaan sitä käyttää laajalti muista ohjelmista käsin.

Kertaus

TIE-23600 Palvelupohjaiset järjestelmät

Sisältöä

- Palvelupohjainen arkkitehtuuri
- Palveluiden toteutustekniikoita
- Pilvilaskentaa

Palvelu

- Itsenäinen ohjelmisto, joka suorittaa jonkin toimintokokonaisuuden
- Esim. hotellihuoneenvarauspalvelu
- Palvelulla on **rajapinta**, jonka kautta kaikki kommunikaatio tehdään
  - Rajapinta on riippumaton palvelun toteutuksesta

Palvelupohjainen arkkitehtuuri

- Service-Oriented Architecture, SOA
- Arkkitehtuurityyli, jossa järjestelmä koostuu palveluista
- Palvelut ovat autonomisia ja mahdollisimman löyhästi toisiinsa sidottuja
  - Pyritään minimoimaan eri palveluiden väliset riippuvuudet

Miksi palvelupohjaisuus?

- Suurten ohjelmistokokonaisuuksien toteuttamiseen
- Järjestelmän osat voivat olla
  - eri tahojen toteuttamia/ylläpitämiä
  - erilaisilla teknologioilla toteutettuja

59

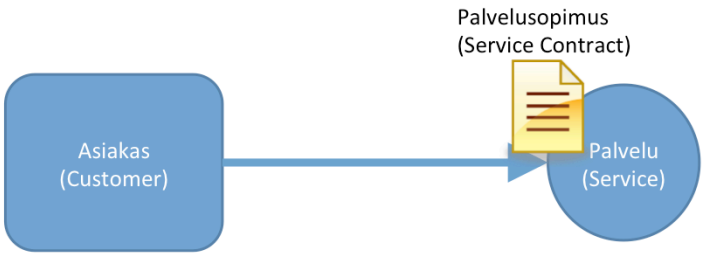
Palvelukeskeisyys ja SOA sopivat hyvin tilanteisiin, joissa kehitetään suurta hajautettua (tai ei-keskitettyä) järjestelmää, jonka eri osat ovat eri tahojen kehittämiä ja ylläpitämiä. Lisäksi eri osat voivat olla eri teknologioilla kehitettyjä.

SOA ei varsinaisesti *pyri* esim. eri järjestelmän osien heterogeenisuuteen, mutta hyväksyy, että se on usein todellisuutta. Samoin kuin vaikkapa ketterät ohjelmistonkehitysmenetelmät eivät *pyri* siihen, että ohjelmiston vaatimukset jatkuvasti muuttuvat, mutta hyväksyvät että näin usein käy ja pyrkivät ottamaan sen huomioon.

Palvelupohjaisuuden hyötyjä

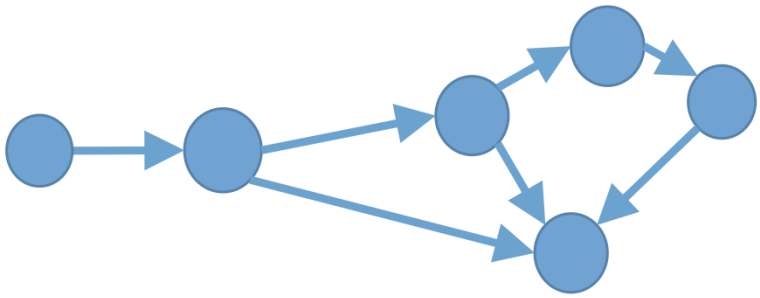
- Olemassaolevan osaamisen hyödyntäminen
  - Palvelut voidaan toteuttaa halutulla teknologialla
- Olemassaolevan ohjelmiston hyödyntäminen
  - Legacy-järjestelmien kääriminen palveluiksi
  - Samaa palvelua voidaan käyttää useassa eri yhteydessä
- Joustavuus
  - Järjestelmän rakenne helposti muutettavissa
  - Palveluita voidaan korvata toisella jne.
- Liiketoiminnan ja tietotekniikan yhdistäminen
  - Palvelut ovat korkean abstraktiotasonsa vuoksi myös muiden kuin ohjelmistoihmisten ymmärrettävissä

Terminologiaa



Palvelut & asiakkaat

- Palvelut voivat toimia myös asiakkaan roolissa
  - Palveluista voidaan koostaa suurempia kokonaisuuksia



Palveluiden toteutus

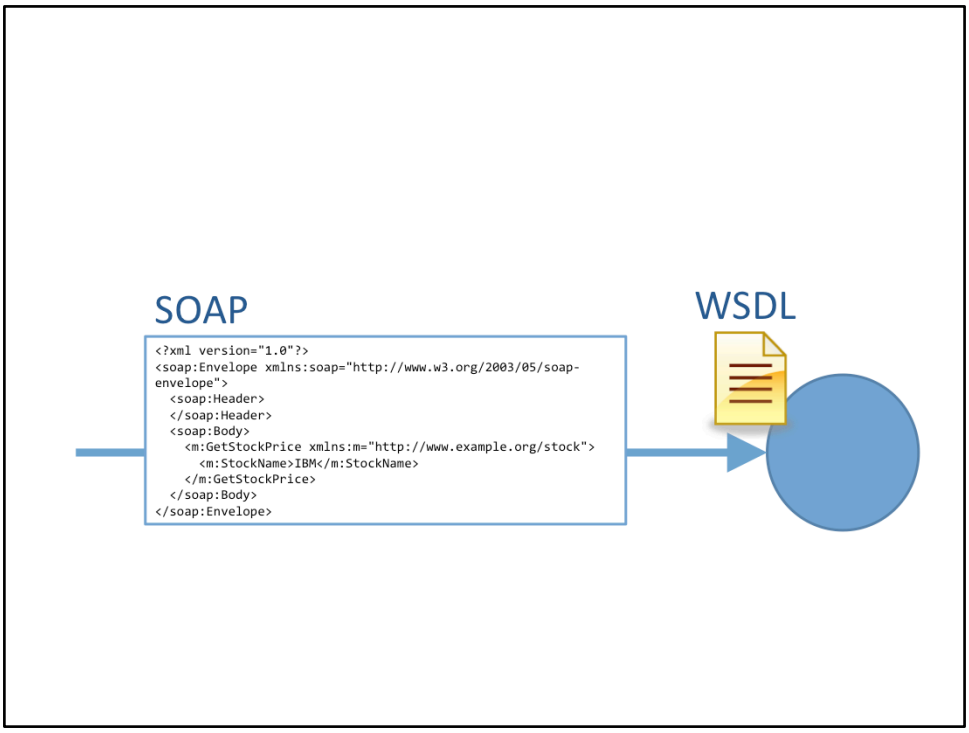
- Kaksi yleisesti käytettyä tapaa palveluiden toteuttamiseen on Web Services –standardit ja REST
- Web Services (WS-*)
  - SOAP, WSDL ja muita XML-pohjaisia standardeja
- REST
  - Arkkitehtuurityyli hajautetuille (hypermedia)järjestelmille

WS-*

- Joukko XML-pohjaisia standardeja palvelukeskeisen arkkitehtuurin toteuttamiseen
- Palveluiden rajapinnat kuvataan WSDL-kielellä
  - Määrittelee palvelun tarjoamat operaatiot, käytetyt tietotyypit, jne.
- Palvelut viestivät käyttäen SOAP-protokollaa
  - Palvelukutsut parametreineen sekä kutsun vastaus
  - Siirtoprotokollana usein HTTP
- Muita WS-* -standardeja
  - WS-Addressing reititykseen
  - WS-Security tietoturvaan
  - …

SOAP

- XML-pohjainen viestinvälitysprotokolla
- *Envelope* sisältää *Header*-osan ja *Body*-osan
  - Header sisältää viestin käsittelyyn liittyvää tietoa
  - Body sisältää viestin varsinaisen sisällön
    - Pyynnön ja vastauksen data
- Siirtoprotokollariippumaton
  - Usein käytössä HTTP
- Viestinvälitysmallit (message exchange patterns)
  - Request-response
  - Response


SOAP
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
<soap:Header>
</soap:Header>
<soap:Body>
<m:GetStockPrice xmlns:m="http://www.example.org/stock">
<m:StockName>IBM</m:StockName>
</m:GetStockPrice>
</soap:Body>
</soap:Envelope>
WSDL

WSDL

- Web Service Description Language
- XML-dokumentti joka kuvaa palvelun palvelusopimuksen
- WSDL-dokumentissa määritellään
  - Tietotyypit, usein XML-skeemojen (XML Schema) avulla
  - Palvelun tarjoamat operaatiot
  - Sitominen (binding) viestinvälitysprotokollaan (esim. SOAP)

REST

- REST (Representational State Transfer) on arkkitehtuurityyli hajautetuille järjestelmille
  - Asettaa arkkitehtuurille tiettyjä rajoitteita
- REST-palvelu sisältää *resursseja* joista voi olla erilaisia *esityksiä*
  - Resurssit sijaitsevat palvelimella
  - Sovelluksen tila on kokonaisuudessaan asiakkaalla
  - Resurssien välillä linkkejä
    - Hypermedia As The Engine Of Application State (HATEOAS)

REST:in rajoitteet

- Asiakas-palvelin
- Tilattomuus
  - Pyyntö sisältää kaiken sen käsittelyyn vaadittavan tiedon
- Välimuistin käyttö
  - Palvelimen vastaukset sisältävät tiedot siitä miten vastaus voidaan säilöä välimuistiin
- Kerroksellisuus
  - Esim. proxy, palomuuri, jne.
  - Eri kerrokset tekemisissä vain viereisten kerrosten kanssa
- Yhtenäinen rajapinta
  - HTTP, URI, MIME-tyypit

Esimerkki REST-palvelusta

- Pohjalla *base URL*, esim. `http://example.com/autot`
- Dataformaatti esim. JSON, XML
- HTTP-metodit (GET, PUT, POST, DELETE)
- Tyypillinen toteutus:

	GET	***PUT***	**POST**	**DELETE**
Resurssikokoelma **`/autot`**	Listaa kokoelman resurssit ja mahdollisesti muuta tietoa	Korvaa koko kokoelman toisella	Luo uuden resurssin. Palauttaa luodun resurssin URI:n.	Tuhoaa koko kokoelman
Yksittäinen resurssi **`/autot/313`**	Palauttaa resurssin esityksen	Korvaa resurssin tai luo uuden	(Luo aliresurssin)	Tuhoaa resurssin

WS-* vs REST

WS-*

- ”RPC-tyylinen”
- Tarkasti määritellyt rajapinnat (WSDL)
- Paremmin koneellisesti käsiteltävissä
  - Työkalutukea koodin generointiin ym.
- Laajennettava
- Tukee monenlaisia kommunikointitapoja
  - Asynkroninen, synkroninen

REST

- ”Resurssipohjainen”
- Käytännössä vapaamuotoisemmin dokumentoitu
  - Helpommin ymmärrettävä
- Luonnostaan skaalautuva
- Vähemmän vaatimuksia asiakkaalle
  - Vain HTTP
  - Voidaan käyttää esim. suoraan JavaScript-web-sovelluksesta ym.

Kahdeksan palveluiden suunnitteluperiaatteita

1. Standardoitu palvelusopimus
   - Yhtenäinen tapa kuvata palveluiden rajapinnat
2. Palveluiden löyhä sidonta
   - Ei tarpeettomia riippuvuuksia palveluiden välillä eikä palvelun ja ympäristön välillä
3. Palveluiden abstraktio
   - Palvelusopimus sisältää vain asiakkaalle välttämättömän tiedon palvelusta
   - Abstrahoidaan tarpeeton tieto palvelun teknologiasta ja toimintalogiikasta
4. Palveluiden uudelleenkäytettävyys
   - Palveluiden tulisi olla käytettävissä useissa eri konteksteissa
   - Monikäyttöinen toiminnallisuus erotetaan omaksi palvelukseen

Kahdeksan palveluiden suunnitteluperiaatteita

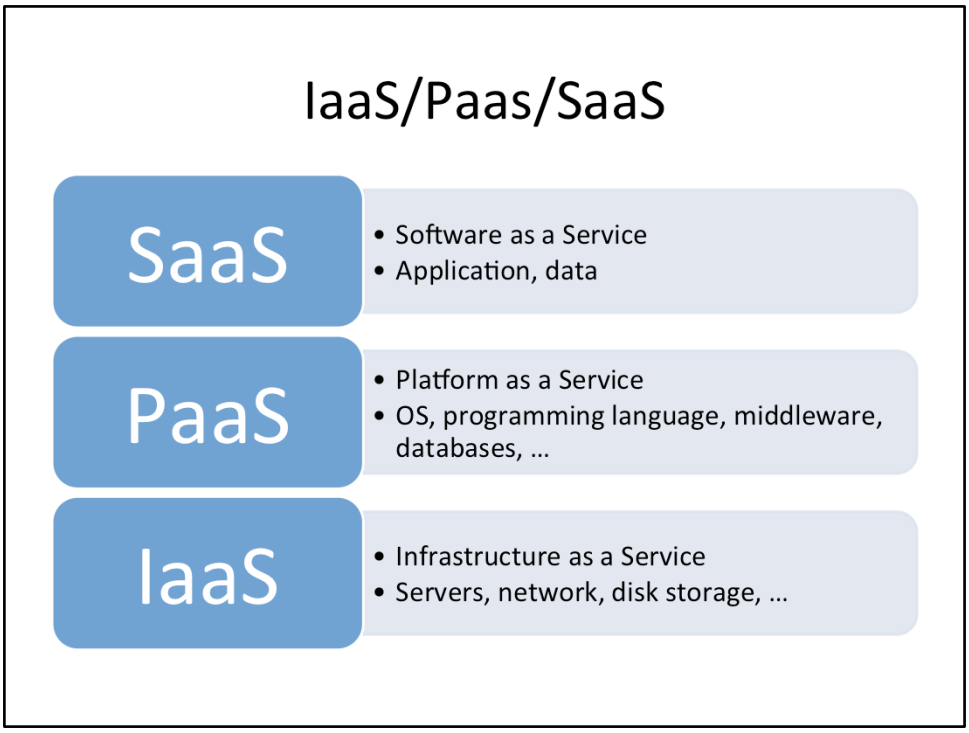
5. Palveluiden autonomisuus
   - Palvelut ovat itse vastuussa toimintalogiikastaan ja ajoympäristöstään
6. Palveluiden tilattomuus
   - Palvelut minimoivat säilyttämänsä tilatiedon
   - Jokainen kutsu palveluun sisältää kaiken kutsun käsittelyyn tarvittavan tiedon
7. Palveluiden löydettävyys
   - Saatavilla tarvittava metatieto sekä ihmisille että koneille
8. Palveluiden yhdisteltävyys
   - Palvelut ovat sellaisia, että niistä voidaan koostaa suurempia kokonaisuuksia

Pilvilaskenta

- Pilvilaskennassa tietoteknisiä resursseja tarjotaan ”palveluina” internetissä
- Resursseja on asiakkaan näkökulmasta saatavilla ~rajattomasti
  - Asiakas maksaa käytön mukaan
- Palvelun käsite on pilvilaskennassa laajempi kuin palvelupohjaisissa arkkitehtuureissa
  - Palvelu voi olla melkein mitä tahansa mitä internetin kautta voidaan tarjota

”as a Service”

- Infrastructure as a Service (IaaS)
  - Tarjotaan resursseja melko ”raa’assa” muodossa
  - Laskentaa (virtuaalikoneita), tallennustilaa, verkkoliikennettä
- Platform as a Service (PaaS)
  - Ajoympäristö sovelluksille
  - PaaS usein tarjoaa kuormantasauksen (load balancing)
- Software as a Service (SaaS)
  - Ohjelmisto palveluna
  - Yleensä toteutettu IaaS:n ja/tai PaaS:n päälle
- Myös muita
  - BaaS, DBaaS, …



Yleinen käytäntö on jakaa pilvipalvelut kolmeen eri tasoon

1)IaaS-taso
Tämä alin taso tarjoaa tietyssä mielessä ”HW-tason”. Se tarjoaa infrastruktuurin, joka yleensä sisältää verkkoyhteydet, tallennustilan, palvelimet sekä niiden ylläpitopalvelun. Infrastruktuuri tarjotaan käytännössä virtuaalikoneinstanssina, jonka avulla voidaan ajaa omia sovelluksia ja palveluita. Infrastruktuurin kaksi päätehtävää ovat tallennustilan ja laskentatehon tarjoaminen asiakkaille.

2)PaaS-taso
PaaS-taso tarjoaa kehitysalustan, joka mahdollistaa ohjelmistokehityksen ja pilvimallin mukaisen teknisen kehityksen antamalla kehittäjille välineet ladata omia sovelluksiaan osaksi kokonaisuutta. Kehittäjien ei tarvitse huolehtia ohjelmiston skaalautuvuudesta tai lisääntyneestä tehotarpeesta käyttäjämäärien kasvaessa, koska alustaa on periaatteessa mahdollista laajentaa tarpeen mukaan joustavasti.

3) SaaS-taso
Tämän tason palvelut muistuttavat – ja usein ovatkin – hyvin lähellä SOA-palveluiden käsitettä, tosin laajennettuna pilvilaskennan liiketoimintamallilla. IaaS- ja PaaS-tasot on tarkoitettu ohjelmisto- /palvelukehittäjien käyttöön, kun taas SaaS-taso on usein tarkoitettu loppukäyttäjän käyttöön. Asiakkaat käyttävät SaaS-palveluja yleensä

Virtualisointi

- Virtualisoinnissa luodaan virtuaalisia tietoteknisiä resursseja
  - Eivät suoraan sidoksissa tiettyyn laitteistoon
- Tärkeässä osassa pilvipalveluntarjoajien (Iaas ja PaaS) toteutuksessa
- Mahdollistaa paremman resurssien käytön kontrolloinnin

Laskennan virtualisointi

- Laitteistotason virtualisointi (hardware-level)
  - Virtuaalikoneita
  - Käytetään IaaS-tarjoajien toteutuksessa
- Ohjelmistotason virtualisointi (programming language-level)
  - Korkean tason virtuaalikoneet (HLVM) suorittavat ohjelmointikielten ”tavukoodia” (bytecode)
  - Käytetään PaaS-tarjoajien toteutuksessa
- Sovellustason virtualisointi (application-level)

Pilvipalveluntarjoajia

	Amazon Web Services	Microsoft Azure	Google App Engine
Virtual machines	EC2	Azure Virtual Machines	Managed VMs
Application runtime	(Any on top of EC2)	.NET, Java, PHP, Python, Node.js, Ruby	Python, Java, Go, PHP
Relational DB	Relational Data Store	SQL Database	Google Cloud SQL
Schemaless DB	SimpleDB, DynamoDB	Table Storage	App Engine Datastore
”File storage”	S3	Blob storage	Cloud Storage
Queues	SQS	Azure Queues, Service Bus Queues	Task Queue
Load balancing	Elastic Load Balancing	Azure Load Balancer	App Engine load balancer