
>

OIOREST

Diskussionsoplæg til workshop vedrørende
anvendelse af REST-baserede webservices i Det
Digitale Danmark.

Version 1.00

1. Indhold

>

1.	Indhold	5
2.	Figurfortegnelse	8
3.	Formål	9
4.	Hvad er REST?	10
4.1	Kort om URI	13
5.	Hvem bruger REST?	14
6.	REST og Web 2.0	15
6.1	Mashup	17
	Mashup i virksomheder	18
6.2	REST velegnet til Dataservices	19
6.3	Mashup på baggrund af virksomhedsdata	20
6.4	Mashup på baggrund af persondata	21
6.5	Stedfæstelser af ustruktureret information	21
7.	REST og serviceorientering	23
7.1	Standarder	23
7.2	Opsummering	25
8.	REST i forhold til Digitaliseringsstrategi 2007- 2010	26
9.	Eksempler på hvor REST kan anvendes	28
9.1	Danmarks Miljøportal	28
9.2	Infostrukturbasen	29
10.	Opsummering	31
11.	Tekniske aspekter	32
11.1	Specifikation	32
	Eksempel på servicespecifikation	33
	Fejlstruktur	35
	11.1.1 Hierarkiske URLer	35
	11.1.2 Læselige URLer	35
	11.1.3 Ressource- orienterede URLer	36
	11.1.4 Giv alle ressourcer en URL	36

>

11.1.5	POST og PUT	36
11.1.6	Undgå så vidt muligt service-specifikke HTTP-headers.	36
11.2	Repræsentation	36
11.3	Sikkerhed	38
	Domæne model og User-begrebet	38
	Autentifikation af klient	39
	End-to-end security	40
11.4	Fejlhåndtering	40
11.5	Stateless opførsel	41
	Pålidelighed	41
	Caching	42
12.	Kodeeksempler	45
12.1	Java	45
12.2	.Net	46
12.3	Ruby	46
12.4	Javascript	47
13.	Frembringelse af en REST-baseret webservice	49

>

Dokument versioner

Version	Dato	Emne
1.0	14-mar-2008	Første version

Bidragydere:

Navn	Firma	E-mail
Henrik Hvid Jensen	Devoteam Consulting	henrik.hvid@devoteam.dk
Preben Thorø	Trifork A/S	pto@trifork.com
Joakim Recht	Trifork A/S	jre@trifork.com
Finn Hartmann Jordal	IT- og Telestyrelsen	fhj@itst.dk

2. Figurfortegnelse

>

Figur 1 Begrænsninger og arkitekturprincipper for REST-baserede services	11
Figur 2 Eksempel på REST i elektronisk tinglysning	12
Figur 3 Eksempler på ting, der kan repræsenteres med en URI.....	13
Figur 4 Web 2.0's hovedområder (Kilde McKinsey)	16
Figur 5 Mashup af data fra CVR, Fødevarestyrelsen, Erhvervs og Selskabsstyrelsen og Politiken.....	21

3. Formål

>

Nærværende dokument er et oplæg til IT- og Telestyrelsens workshop 22. april 2008 vedr. anvendelse af REST-baserede webservices i Det Digitale Danmark. Dokumentets formål er at præsentere REST, som det anvendes i dag samt at diskutere, hvordan udbredelsen af REST tænkes at være fremover – ikke mindst i kontekst af serviceorienteret systemintegration.

Sidste del af dokumentet præsenterer en række tekniske karakteristika ved den REST-baserede arkitektur.

Dokumentet er tænkt som et informerende oplæg til workshoppen, hvis formål er at indsamle information nok til at forme det endelige dokument. Det endelige dokument vil være en værdifuld vejledning under udarbejdelsen af REST-baserede webservices i Det Digitale Danmark.

4. Hvad er REST?

>

REST (Representational State Transfer) er betegnelsen for en arkitektonisk stil, som er beskrevet i Roy Fieldings¹ Ph.D. afhandling. Den arkitektoniske stil REST definerer en række basale regler (constraints) til frembringelse af konkrete arkitekturer. Formålet med at følge disse regler er at frembringe en arkitektur, som besidder en række ønskelige egenskaber: Simplicity, scalability, performance, evolvability, visibility, portability og reliability (Figur 1).

Dokumentet er ikke rettet mod at beskrive denne arkitektoniske stil, men at beskrive hvorledes webservices designs, så de følger den arkitektoniske stil og passer bedst muligt ind i Det Digitale Danmark.

REST anvender HTTP som en tilstandsløs client/server protokol til udveksling af information i et request/response mønster.

Det vil sige, at REST baserer sig på operationerne POST, GET, PUT og DELETE. I REST er en ressource udstillet ved brug af en URI (se afsnit 4.1), som ofte ligner den velkendte URL. Ideen med REST er at ændre tilstanden af en ressource gennem overførsel af repræsentationen af ressourcen. REST er en arkitektonisk stil, som har bragt Web'en succes. Formålet med at anvende REST i webservice sammenhænge er at opnå samme fordele og succes i system-til-system kommunikation, som web'en har opnået i system-til-menneske kommunikation.

RESTful service er uden tilstand. Dette betyder, at servicen behandler hver anmodning som en uafhængig transaktion, der er uden relationer til alle tidligere anmodninger. Derfor må enhver anmodning fra klienten til serveren indeholde al information, der er nødvendig for at forstå anmodningen og kan ikke udnytte tidligere registreret indhold på serveren. Ressourcens tilstand såvel som tilstandsovergange bør derfor være en eksplicit del af repræsentationens informationsmodel. Denne begrænsning tilføjer egenskaber som synlighed, troværdighed og skalerbarhed.

- **Synlighed er forbedret**, fordi et system ikke behøver at kigge videre end en enkelt anmodning for at forstå det fulde formål med anmodningen.
- **Troværdigheden forøges**, fordi det letter opgaven med at genetablere efter en delvis fejl.
- **Skalerbarhed forøges**; når man ikke har behov for at gemme tilstanden mellem service og klienten, kan server-komponenten hurtigere frigive ledige ressourcer. Dertil kommer lettere

¹ Roy Fielding er en af forfatterne til http-specifikationen og en af grundlæggerne til Apaches http server projekt

>

implementation, fordi serveren ikke skal håndtere ressourcebrug på tværs af anmodninger.

Ulemperne er reduceret netværks-performance, fordi samme data sendes i flere anmodninger, da de ikke kan gemmes på serveren i et fælles område.

I forhold til løsninger hvor data og applikation er placeret på samme server og alle anvender denne løsning, vil det at placere applikationens tilstand på klientsiden reducere serverens kontrol over konsistent applikationsopførsel. Da flere klienter skal behandle data, bliver applikationer afhængige af korrekt implementering af semantikken på tværs af flere klient-versioner

RESTful services har et ensartet interface. Denne begrænsning fortolkes typisk til, at de eneste tilladte operationer er HTTP-operationerne: GET, PUT, POST og DELETE

Ensatet identifikation af ressourcer. REST-baserede arkitekturer er opbygget fra ressourcer (stykker af information), der er unikt identificeret ved URIer. For eksempel i et elektronisk patientjournal system vil en patientjournal have en unik URI, i et elektronisk tinglysningssystem, vil hver tinglyst rettighed have en unik URI og i et personregistreringssystem vil hver person have en unik URI.

Manipulerer ressourcer gennem deres repræsentation. REST-komponenter manipulerer ressourcer ved at udveksle repræsentationer af ressourcer. F.eks. kan en patientjournal, en tinglyst rettighed eller en person være repræsenteret ved et XML-dokument. En patientjournal, tinglyst rettighed eller person vil blive opdateret ved at lægge (PUT) et XML-dokument, der indeholder den ændrede patientjournal, tinglyste rettighed eller person til dets URI.

Selvbeskrivende repræsentation. Repræsentationen skal være let at fortolke og forstå.

Figur 1 Begrænsninger og arkitekturprincipper for REST-baserede services

REST er forskellig fra Remote Procedure Call (RPC), som indeholder konceptet om at starte en procedure på et eksternt system. RPC-beskeder indeholder typisk information om proceduren, der skal startes, eller hvad der skal gøres. Denne information refereres typisk til som et verbum i et RPC-kald. I REST-modellen er de eneste tilladte verber GET, PUT, POST og DELETE. I traditionel RPC har man typisk en URI til flere operationer, hvor man i REST i højere grad tilgår de enkelte ressourcer på forskellige URIer.

I en REST-implementation af elektronisk tinglysning kunne man for eksempel kalde en ressource for *ejendom*, og den kunne have mange tusinde udstillede URIer. <http://tinglysning/ejendom/1FA-6940-0//adkomsthaver>, vil referere til adkomsthavere (ejere) af ejendommen på matrikel 1FA-6940-0. For at ændre informationen om adkomsthaveren på den ejendom vil brugeren POSTe eller PUTe et XML-dokument ind i den pågældende URI, og en anden bruger kan

>

efterfølgende hente informationen via et GET-kald til URIen. I stedet for at fortage et funktionskald til hentAdkomsthaver og overføre ejendomsnummer i en ejendomID lægger man i REST noget XML ned i den rigtige ressource. Tilstanden og typen af dataelementet er nøgleaspektet i REST.

Figur 2 Eksempel på REST i elektronisk tinglysning

En del af applikationslogikken er udtrykt i begrebsmodellen og indgår i data, frem for at indgå i koden. Dette sætter nye krav til begrebsmodellen, da det ikke vil være muligt at kompensere i den etablerede kode for uhensigtsmæssigheder i datamodellen. Det har som konsekvens at mange etablerede datamodeller er utilstrækkelige. F.eks. datamodeller der er bygget på baggrund af en automatisering af en papirblanket, hvor papirblanketten ofte bliver basis for datastrukturen. F.eks. vil en udskrivning af en patient fra et hospital ikke være en entitet i sig selv, men er den information, der registreres som del af et tilstandsskift fra, at en person er en patient, til han er udskrevet. Etablerede datamodeller reflekterer ofte artefakter af tidligere procesdefinitioner. Det bemærkes dog, at den eksisterende datamodel sjældent udstilles som serviceinterface.

I REST vil et objekt (f.eks. ejendom) have mange operationer for at danne, slette og ændre posten med mange adskilte instanser af objektet, der hver henviser til forskellige ejendomme. REST er derfor mere at sammenligne med dokument-baseret kommunikation, hvor man kalder services og medsender et XML-dokument. Svaret retur er et andet XML-dokument, og begge dokumenter er dannet i henhold til kontrakten. Som et resultat tilbyder dokument-baseret kommunikation et større niveau af løs kobling end RPC, da enhver ændring i metoden eller operationen har begrænset indflydelse på serviceforbrugeren. I tilgift indholder body'en af en dokument-baseret webservice den faktiske forretningsbesked, det der virkelig tæller. I RPC beskriver XML delen typisk kun metoden og de tilhørende parametre.

Mange REST-baserede løsninger overholder ikke de ovenfor nævnte principper, hvilket bl.a. ses på forskellige e-handel-sites. Her ses det ofte, at man holder indkøbskurv-data på serveren og vedligeholder en session relateret til indkøbsprocessen, som bruger indkøbskurven - selv om dette er en klar overtrædelse af Fieldings princip om tilstandsløs opførsel.

En anden ofte set afvigelse for Fieldings REST-principper er indlejring af verber og parametre i URLerne. Derved fraviges princippet om ensartet interface. I disse situationer begynder REST-baserede løsninger primært at være RPC med brug af XML over HTTP uden SOAP. Dette dokument vil ikke omhandle disse situationer.

4.1 Kort om URI

Den mest kendte form af URI er vores velkendte hjemmesiders Unified Resource Locator (URL), der adresserer de ressourcer, der kan hentes. URI er decentraliseret; der er ingen person eller organisation, der kontrollerer, hvem der laver dem, eller hvordan de bruges. Man behøver ingen tilladelse til at lave en URI for noget. Man kan lave URIer for ting, man ikke ejer, eller for abstrakte begreber, som ikke eksisterer fysisk (se Figur 3).

- **Netværkstilgængelige ting:** Såsom et elektronisk dokument, et billede, en service (Kraks kort), et musikstykke.
- **Ting som ikke er netværkstilgængelige:** Såsom mennesker, firmaer, biler, landområder, historiske steder, begivenheder.
- **Abstrakte koncepter som ikke eksisterer fysisk:** Såsom koncepter omkring en instruktør, at være en dogmefilm, spille golf.

Figur 3 Eksempler på ting der kan repræsenteres med en URI.

5. Hvem bruger REST?

>

Det bedste eksempel på brugen af REST er selve internettet. En almindelig webbrowser kommunikerer med en webserver vha. HTTP-protokollen efter REST metoden.

REST bruges også i forskellige klientprogrammer. F.eks. interagerer man med RSS og ATOM feeds gennem REST, og alle feed-læsere, der bruger disse formater, bruger altså også REST.

Flere forskellige webtjenester udbyder API'er, der følger REST metoden. Eksempler inkluderer Amazons Simple Storage Service (<http://www.amazon.com/gp/browse.html?node=16427261>) og Microsofts Silverlight teknologi (<http://msdn2.microsoft.com/en-us/library/bb851616.aspx>).

Der er en generel tendens til, at store webservice-leverandører tilbyder REST-baserede services. Det skyldes, at ved at overholde principperne i Figur 1, producerer man applikationer, der er skalerbare på tværs af infrastrukturer, lette at integrere med andre applikationer og lettere kan vedligeholdes i forhold til traditionelle designmønstre.

Fra et REST-perspektiv er den underliggende arkitektur uden betydning, man kan udnytte REST for en n-tier applikation såvel som en klient/server løsning, ligeså let som den serviceorienterede tilgang. Enhver applikation kan være RESTful, hvis det udstiller dets muligheder gennem en GET-anmodning via http, og den overholder principperne for REST.

6. REST og Web 2.0

>

SOA er virksomhedens måde at normalisere it-systemer på for at gøre dem lettere at dele, mere dynamiske og lettere at integrere. Web 2.0 er lig dette, men mere rettet mod den enkelte person og det sociale koncept, hvilket også inkluderer måden til at gøre applikationer til platforme, der kan genbruges, deles og sammenstilles.

Både Web 2.0 og SOA adresserer to nøgleområder: Forbindelse og sammensætning. De tidligste implementationer af webservices har fokuseret på at forbinde store egenudviklede systemer, så virksomheden kan få mere værdi ud af de data og applikationer, som de allerede har. SOA har hidtil handlet mest om at forbinde applikationer og databaser ikke så meget om at binde mennesker sammen og hjælpe med at understøtte deres interaktion med hinanden. Web 2.0 lægger mere fokus på at forbinde mennesker og understøtte deres muligheder for at samarbejde. Web 2.0 indeholder også muligheden for at forbinde applikationer og data, men Web 2.0 adskiller sig ved den sociale dimension, som den eksplicit adresserer.

SOA er teknisk mere komplekst og har mere overordnede koncepter såsom orkestrering, mens Web 2.0 har sociale-, præsentationsmæssige-, ad hoc organiserings- og samarbejdende aspekter, som SOA ikke adresserer.

Den kommende bølge af innovation hos virksomhederne vil afhænge af muligheden for at forbinde personer mere effektivt. Det gælder både på alle niveauer hos virksomheden internt mellem medarbejdere, i partnerkanalen samt hos kunderne både horisontalt i niveauerne og vertikalt på tværs af niveauerne. Web 2.0 teknologier som f.eks. wiki'er vil spille nøgleroller i at opbygge og tilføre værdi i virksomheden

Det betyder, at SOA og Web 2.0 i praksis komplementerer hinanden.

Web 2.0 applikationer er bygget af et netværk af samarbejdende services. Det er derfor nødvendigt at opbygge sine services løst koblet og derved gøre det let for andre at bygge videre på og tilføje ekstra værdi, samt selv udnytte data-services fra andre i egne services. REST fokus er netop en simpel måde at opbygge løst koblede services på, og REST er derfor en teknologi, der ofte anvendes i Web 2.0 applikationer.

Web 2.0 og SOA vil afhænge af hinanden. Web 2.0 indeholder i øjeblikket ikke de mekanismer til sikkerhed, troværdighed, forudsigelighed og stabilitet som virksomheder skal bruge for at benytte Web 2.0 som integrerede dele af deres vigtige forretningsprocesser. SOA indeholder netop disse mekanismer og SOA vil derfor udgøre fundamentet, som virksomhederne kan bygge deres personorienterede Web 2.0 løsninger på. SOA værktøjer som Service register, Enterprise Service Bus, Business Process Engine (BPE) vil være værktøjer på linie med Web 2.0 værktøjerne (Figur 4), når virksomheden skal vælge den rigtige løsning til at forfølge dets forretningsmuligheder.

- **Blog.** Er online tidsskrifter eller dagbøger placeret på en vært-hjemmeside og ofte distribueret til andre hjemmesider eller læsere via RSS
- **Kollektiv viden.** Henviser til ethvert system, som forsøger at indhente viden fra en gruppe frem for individuelle for at foretage beslutninger. Den stigende mængde af beslutninger kræver mere business intelligence, men ikke nødvendigvis gennem dataming-tekninger; noget mere umiddelbart er påkrævet for at understøtte beslutninger baseret på nutidig information
- **Mash-ups.** Samlinger af indhold fra flere forskellige online kilder for at danne nye services. Et eksempel kan være et program, der henter servitutter på en ejendom fra en service udstillet af e-TL og viser dem på et satellitkort fra Google Earth (se afsnit 6.1).
- **Peer-to-Peer networking (også kaldet P2P).** Er en teknologi til at dele filer (f.eks. musik, film eller tekst) enten over internettet eller indenfor en lukket brugerkreds. I forhold til den traditionelle metode til at dele en fil på en maskine, som hurtigt kan blive flaskehalsen, hvis mange forsøger at tilgå den, så distribuerer P2P filer på tværs af mange maskiner, ofte hos brugerne selv. Maskinerne modtager så filerne ved at hente og samle småbidder fra mange maskiner.
- **Podcast.** Er audio eller video optagelser, en multimedieform for en blog.
- **RSS (Really Simple Syndication) og APP (Atom Publishing Protocol).** Tillader at man abonnerer på online distribution af blogs, nyheder, podcasts og anden information
- **Sociale netværk.** Henviser til systemer, som tilbyder medlemmer af et specifik site at lære om andre medlemmers evner, talenter, viden eller præferencer. Kommercielle eksempler inkluderer facebook og LinkedIn. Nogle virksomheder bruger disse systemer internt for at kunne identificere eksperter
- **Web Service.** Softwareløsninger som gør det lettere for forskellige systemer at kommunikere med det formål at overføre informationer eller foretage transaktioner
- **Wikis.** Er systemer til offentliggørelse i fællesskab. De tillader mange forfattere at bidrage til et online dokument eller diskussion. Formålet er let og hurtigt samarbejde. Et godt eksempel er Wikipedia.

Figur 4 Web 2.0's hovedområder (Kilde McKinsey)

Både Web 2.0 og SOA betragter software som service, og de ser selve servicen som platformen. Frem for at se services som enkeltstående enheder, designet til at blive forbrugt præcist som beskrevet, indeholder begge teknologier visionen om, at enhver service ultimativt vil blive byggeklods for flere services, som vil bygges på toppen af den oprindelige service. Hvor SOA omhandler sammensatte applikationer bruger Web 2.0 et lignende koncept kaldet Mashups.

6.1 Mashup

Mashupdelen af web 2.0 er den situation, hvor man sammenbinder applikationer fra en samling af eksisterende og sædvanligvis web-baserede aktiver. www.earthalbum.com kombinerer indhold fra fotomapper med kort-hjemmesider og giver mulighed for, at man kombinerer sin egen visuelle tur rundt om jorden.

Begrebet mashup stammer fra musikindustrien og beskriver det fænomen, hvor flere musikspor fra forskellige sange blev sammenblandet (mashed up) for at lave en ny sang. Stemmesporet fra en sang blev lagt sammen med rytmesporet fra en anden og basguitar-sporet fra en tredje sang.

En mashup er et præsentationslag for integration af indhold og applikationer fra flere kilder i typisk en enkelt browserbaseret løsning. De kilder, der integreres, skal ikke ændres, men beholder deres originale formål og struktur.

En mashup-løsning har ikke eget indhold, men sammensætter indhold eller funktionalitet fra eksisterende systemer, præsentationsfunktionaliteten stammer også fra kilder eksternt for mashup'en. Mashups kan indeholde noget forretningslogik i sammensætningen af mashup-komponenterne, men det er sædvanligvis minimalt, og ideelt set bør al understøttende funktionalitet være leveret af mashup-miljøet og er således ikke specifikt til en bestemt mashup.

Mashup er en simplere måde at sammensætte applikationer på end den serviceorienterede orkestrering til sammensætning af applikationer. Hvor sammensatte applikationer i SOA kontekst ofte fokuserer på at binde computerunderstøttede forretningsprocesser sammen internt eller mellem faste forretningspartnere og ofte resulterer i modificeringer af de data, der modtages, bruges begrebet mashup i højere grad om web-baserede applikationer på forbrugerorienterede hjemmesider rettet mod eksterne, der kombinerer information til nye løsninger. Så mashup er i dag rettet mod præsentationslaget, hvor SOA fokuserer på applikation-til-applikation-integration

Mashups udnytter indhold og logik fra andre hjemmesider og web-applikationer, de er simple at implementere og bygges med en begrænset mængde kode. Mashups bruger letvægtsmekanismer såsom REST-baserede API'er og web-udvekslingsformater såsom XML, RSS og Atom. Mashups har sjældent til formål at være strategiske, og er ikke opbygget systematisk med

>

traditionel udvikling internt i virksomheden; de er i højere grad opbygget hurtigt og opportunistisk for at imødegå et taktisk behov.

De kilder som indgår i en mashup, er typisk ikke eksplicit designet til at blive integreret med hinanden, men åbne web-API'er og internetprotokoller muliggør deres kombination - ofte på innovative måder og ofte af ikke professionelle programmører der var involveret i design eller udvikling af kildesystemet. Mange mashups er drevet af webkulturen, dvs. sociale netværks hjemmesider skræddersyet til enkelte interesseområder. Housingmaps er et eksempel, hvor data fra Google Maps kombineres med lejligheder til udlejning fra Craigslist, hvilket samlet giver en applikation, der viser placeringen af tilgængelige lejligheder i en given by. Denne er lavet uden direkte medvirken fra medarbejdere fra hverken Google eller Craigslist. På www.programmableweb.com er der andre eksempler på mashups.

Organisationer kan aktivt opfordre til mashups ved at offentliggøre deres web-API'er f.eks. som REST-services. Mashups er drevet af web 2.0 kulturen, der lægger vægt på deltagelse i grupper for at drive innovationen ved kreativt at benytte og dele web-baseret indhold.

Mashup i virksomheder

Populariteten af mashups på det offentlige internet har medført, at virksomheder er begyndt at eksperimentere med, hvordan mashup-tankegangen kan levere værdi internt i virksomheden og mellem forretningspartnere. Det bemærkes dog, at langt de fleste mashups er på hobby-stadiet og kun tilfører minimal forretningsværdi og typisk afhænger af data og services fra det offentlige internet, såsom Google Maps, eBay og Amazon. Når Web 2.0 teknologier anvendes internt i en virksomhed, bruger nogle begrebet Enterprise 2.0, der vil ikke ske denne skelnen i denne rapport

I de situationer hvor mashup giver forretningsværdi, begynder virksomhederne at stille krav om troværdighed, sikkerhed, governance osv., og det vil formentlig sætte restriktioner på hvor dynamiske, fleksible og letvægtsagtige mashup-komponenterne kan være.

Der er flere værktøjer, der gør det muligt for ikke-programmører at sammensætte indhold fra flere hjemmesider (f.eks. Yahoo Pipes, Microsoft Popfly og Google Mashup), hvorved brugeren kan blive direkte involveret i at sammensætte egne mashups. Denne transformation mod brugerdrevne mashup-applikationer stemmer overens med den forventede udvikling i virksomhederne, hvor medarbejderne selv kan danne mashups, til deres behov. Dette er i tråd med den udvikling, der længe har været indenfor løsninger baseret på regneark, som ikke har været understøttet af it-afdelingen, men har løst et behov hos et individ eller mindre gruppe

Mashups fokuserer typisk på at opfylde et personligt behov eller behovet i mindre grupper frem for generelle krav i en virksomheds længerevarende

forretningssituation. Mashup bruges til hurtigt at integrere indhold og funktionalitet fra flere uafhængige kilder. Afvejningerne sker mellem kortere tid til markedet og lavere udviklingsomkostninger i forhold til applikationens robusthed og levedygtighed. Det vil først være i forbindelse med *virksomhedsmashup*, hvor leverandører af virksomhedssoftware, platforme og middleware adopterer Web 2.0 konceptet og tilpasser det til virksomhedernes behov, at konceptet vil opnå den tilstrækkelige troværdighed, til at virksomheder vil udnytte det i forretningskritiske applikationer.

Indenfor det offentlige, vil det sætte krav om klare retningslinier for offentlige myndigheder, der tilbyder funktionalitet og indhold, der kan indgå i mashups. Det skal være klart, hvilke servicekrav der vil blive opfyldt, for at virksomhederne vil benytte de udbudte services.

Mashups i virksomheder vil ske ved, at virksomheder på forhånd samler og godkender mashup-komponenter fra såvel interne som eksterne kilder. Derved kan mashups forøge it-afdelingens muligheder for hurtigere at levere applikationer og lettere adressere applikationskrav, som ellers ville blive nedprioriteret i it-afdelingen på grund af manglende ressourcer. Denne tendens underbygges også af, at de større leverandører af udviklingsværktøjer er begyndt at understøtte REST-løsninger i deres udviklingsværktøjer

6.2 REST velegnet til Dataservices

Fokus indenfor REST-baserede webservices er på data. Dataservices er brug af SOA-principperne på adgang til data – netop for at opnå en løst koblet tilgang til data. Formålet med dataservices er at give løst koblet adgang for et bredt udvalg af klienter til et bredt udvalg af informationer gemt på mange forskellige enheder lige fra flade filer over spreadsheet til databaser.

Dataservice adskiller sig fra applikationsservices ved, at de følger samme princip, som SOA giver for udvikling af applikationer. Dataservices udstiller data frem for operationer. De fokuserer på at indkapsle et stykke information og udstille det og gøre det tilgængeligt. Det bemærkes, at i dag er hovedparten af WS-* service dataservices, der typisk kaldes Get<datanavn>, som f.eks. GetMedicineChestStructure fra medicinprofilen, GetVisitationCase fra arbejdsmarkedstyrelsens DRV services.

Ved at gøre kvalitetskontrolleret data tilgængelige for forretningsbrugerne undgås det, at afdelinger eller virksomheder laver kopier, for at sikre at de kan få de data, de har behov for. Ved anvendelse af løst koblede dataservice, der er let tilgængelige, opnås konceptet om Netværksdata.

Netværksdata drejer sig om at kunne tilpasse sig dynamisk til netværkets behov. Det kræver, at man holder op med at bekymre sig om at eje eller kontrollere specifikke data og processer og i stedet accepterer, at de er placeret de steder, der er mest optimale for netværket. Virksomheden må derfor

>

acceptere, at det i mange situationer vil være mest optimalt at have en enkelt datakilde tilgængelig for hele netværket, i stedet for at alle deltagere i netværket har deres egne versioner af dataene liggende.

Dataservices kan bruge data fra alle datakilder, så dataservices indkapsler virksomhedens og samarbejdspartnerens forskellige backend datakilder. Dataservices kan altså trække applikationer fra en mængde forskellige databaser. Frem for at alle foretager en integration, hver gang man har behov for at trække dataene, dannes en dataservice, som kan bruges fra mange forskellige applikationer

Mashups er i dag primært forbundet med visualisering, men data-mashup vil blive anvendt, efterhånden som implementationen i virksomhederne modnes. Når dette sker, vil forskellen mellem SOA og Mashup i forbindelse med dataservices, der kombinerer data fra forskellige kilder, bliver udvisket.

6.3 Mashup på baggrund af virksomhedsdata

Langt de fleste mashups, der eksisterer i dag, baserer sig på at vise stedfæstet data med et kort som baggrund. Det skyldes bl.a., at kort er en attraktiv brugergrænseflade, og at en stor del af data kan stedfæstes, hvilket giver en økonomisk drivkraft til at promovere en yderligere anvendelse af dette.

Det vurderes, at 80 % af alle offentlige data kan stedfæstes, størsteparten af de resterende 20 % kan relateres til en person eller virksomhed. Disse person- og virksomhedsdata kan også fungere, som baggrund for data-mashups, men det er endnu ikke sket i væsentlig grad. Det bunder i to ting: Dels har man indenfor kort-information tradition for at lægge lag ovenpå et kort, præcist som formålet er med mashups. Den tradition eksisterer ikke inden for virksomheds- og persondata, blandt andet er disse data ikke standardiseret på samme måde, som stedfæstet data er standardiseret gennem GIS-konsortiet.

For at lette adgangen til data-mashups på baggrund af virksomhedsdata, anbefales det at udstille CVR-data som REST-services. Disse REST-baserede CVR-data kan f.eks. mashup'es med virksomheders regnskab i XBRL, det vil i den forbindelse kræve at Erhvervs- og Selskabsstyrelsen definerer en standard for, hvordan en REST-baseret XBRL service ser ud. Denne XBRL-service kan udstilles fra virksomhedens hjemmeside eller fra en virksomhed, der har specialiseret sig i at tilbyde regnskaber, måske tilføjet yderligere værdi gennem andre mashups. Det vil samtidig kræve, at virksomheder tildeles en unik URI for derved at muliggøre mashup med andre virksomheds-relaterede data. En anden mulighed for CVR-mashup kunne være ved hjælp af statistik fra Danmarks Statistik, hvor de forskellige statistik-dimensioner er udbudt som REST gennem URIs.

På www.findvej.dk/smiley er der lagt smileys ind på et kort, der repræsenterer data fra Fødevarestyrelsen. Data i denne løsning hentes en gang om måneden og lagres i en kommasepareret fil.

Hvis CVR-data, Fødevarestyrelsens smiley-information og regnskabsinformation i XBRL alle blev tilgængelige som RESTbaserede webservices, kunne man lave en data-mashup, hvor CVR-information dynamisk bliver mashup'et med smiley-information og regnskabet for den pågældende virksomhed. Hvis eksempelvis Politiken offentliggjorde sine kokkeheder som REST, kunne denne information også indgå i mashup'en

Figur 5 Mashup af data fra CVR, Fødevarestyrelsen, Erhvervs og Selskabsstyrelsen og Politiken

6.4 Mashup på baggrund af persondata

Mashups på persondata kan foretages på samme måde som med virksomhedsdata, det vil i denne situation kræve, at personer identificeres unikt ved en URI. Persondataloven vil dog sætte grænser for, hvorledes CPR-information kan kombineres og udstilles. En forudsætning kan være, at brugeren identificerer sig via sin OCES-digitale signatur, derved har REST-servicen også viden om personen, og man kan lave mere personaliserede services. F.eks. kan borger.dk lave en mashup fra en kommunes REST-service, der angiver placeringen af vuggestuer og gøre den personlig, da borger.dk ud fra OCES-signaturen umiddelbart ved, hvilket lokalområde borgeren har interesse i.

Borger.dk vil være et naturligt sted at udstille borger-rettede mashup's samt indeholde et katalog over services, som man kan anvende til at bygge borgervendte mashup's

6.5 Stedfæstelser af ustruktureret information

Det faktum, at mashup's passer godt til at vise stedfæstede informationer på en kortbaggrund, bør udnyttes af det offentlige til at lette adgang til ustruktureret information såsom dokumenter, billeder og film. Dette kan gøres, ved at det offentlige definerer retningslinier for, hvorledes ustrukturerede informationer stedfæstes.

En kommunes dokumenter kan i dets ESDH-system generelt være stedfæstet til hele kommunens område, og dokumenter, der f.eks. omhandler en specifik skole, kan stedfæstes til skolens område.

Danmarks Radios og TV2s udsendelser (nye såvel som gamle (www.dr.dk/bonanza)), indslag i TV-avisen og andre relevante udsendelser bør stedfæstes.

Relevante dokumenter hos Statens Arkiver og forskningsrapporter bør også stedfæstes osv.

Med de rette metadata til ustrukturerede informationer vil det gøre information lettere tilgængelig i relevant kontekst for en større del af Danmarks befolkning.

>

Frihedsmuseet under Nationalmuseet er i gang med at oprette et offentligt kilde- og personregister over danske modstandsfolk. Oplysningerne om modstandsfolkene vil blive taget fra bøger og artikler, der findes om besættelsestiden. Hvis disse informationer også blev stedfæstet, vil deres nytteværdi øges. De kan f.eks. bruges til mashup af artiklen på baggrund af et kort fra Miljøportalens areal-information til personer, der gerne vil se kælderen hvor den illegale avis blev trykket, hvor sabotagen fandt sted, eller hvilket hus modstandsmanden boede i; alt sammen let tilgængelige via REST på såvel browsere som bærbare enheder.

Figur 6 Stedfæstelse af artikler om modstandsfolk

7. REST og serviceorientering

>

Det har tidligere været naturligt at opdele services efter, om de skulle være til internt virksomhedsbrug eller globalt brug. Hvor globalt brug betyder, at fokus for anvendelsen af servicen er eksternt rettet og via internettet udvider de computerunderstøttede forretningsprocesser til forbrugere, mobile enheder og forretningspartnere. Globalt brug har traditionelt været kendetegnet ved internet-baseret computerbehandling og udbyder-services, der er mere fleksible og mere simple, end de der er designet til brug i virksomheden.

Med udviklingen i SOA og webservices hvor de computerunderstøttede forretningskritiske forretningsprocesser inkluderer services fra såvel interne som eksterne parter (f.eks. elektronisk tinglysning), og udviklingen i Web 2.0 medfører, at dynamiske letvægtssamarbejder med såvel interne som eksterne systemer og parter bliver en vigtig del af virksomhedens konkurrenceparameter, giver det ikke mening at dele services op på denne måde. Der er i højere grad nødvendigt at opdele services efter den sammenhæng, de skal bruges i.

Er de en del af computerunderstøttede forretningsprocesser eller er fokus på Web 2.0 koncepternes understøttelse af dynamisk anvendelse af services? Services for begge typer stiller krav om modularitet, distribueret, delbart og løst koblet arkitektur og kan indgå på lige fod i en serviceorienteret arkitektur. Men der kan være forskellige behov for troværdighed, sikkerhed, indgåelse i orkestrerede eller koreograferede forretningsprocesser, dynamisk integration og letvægtskoncepter.

Efterhånd som virksomheder begynder at anvende web 2.0 applikationer internt i organisationen, vil de få en afhængighed til eksterne ressourcer. Det kan medføre, at disse applikationer en dag ikke fungerer, fordi et vigtigt feed ikke eksisterer mere. Fokus på governance vil derfor øges, og krav om forudsigelig skalerbarhed, troværdighed, sikkerhed osv., vil være nødvendige for enhver kombination af SOA, REST og web 2.0 applikationer.

7.1 Standarder

Web Service-standarderne er designet til applikation-til-applikations-kommunikation, de indeholder ikke kun standardiserede formater til udveksling af information; de indeholder også en beslutning om principper for opbygningen af løsningen - f.eks. hvordan processer sammensættes, hvordan sikkerheden designes, hvordan beskeder udveksles osv. På den ene side letter det kravspecificeringen, da mange beslutninger dermed allerede er taget. På den anden side begrænser det virksomhedens muligheder for individualitet, og derved begrænser det muligheden for at differentiere sig

REST er ikke som WS-* fokuseret omkring standarder udarbejdet til applikation-til-applikations-kommunikation, hvilket kan give både udfordringer og fordele for både leverandører og it-afdelinger.

>

De fleste webservices i dag er *forbindelsesorienterede*, hvilket tillader, at mange detaljer såsom sikkerhed implementeres på forbindelsesniveauet og kræver en direkte forbindelse mellem serviceleverandøren og forbrugeren. I dag er de fleste webservices punkt til punkt-løsninger. Dette betyder, at mange virksomheder f.eks. beskytter webservices med veletablerede internetbaserede sikkerhedsværktøjer såsom SSL/TLS, og de fungerer derfor i en sikkerhedssammenhæng, der er den samme som på internettet og udnytter ikke de mere avancerede webservice-standarder.

Mere avancerede webservices er *beskedorienterede* uden sikkerhed for, at der er en direkte forbindelse mellem serviceleverandøren og forbrugeren. Mange traditionelle tilgange er derfor ikke velegnede eller er utilstrækkelige for en webservice-arkitektur. SOAP er som udgangspunkt uafhængig af de underliggende kommunikationslag. Mange forskellige kommunikationsteknologier kan bruges i forbindelse med en SOAP-besked, der bevæger sig over mange led.

Det kunne sagtens tænkes, at man bruger HTTP først, derefter SMTP og så videre, f.eks. i næste generations webservices, hvor beskederne bliver sendt og behandlet mellem flere forskellige led.

REST er ikke bygget til kommunikation på tværs af flere led, men egner sig bedre til forbindelsesorienterede løsninger, hvor man kan udnytte teknologier såsom URI, http og XML til at få applikations til applikations-integration til at fungere på samme måde som resten af internettet.

Et af kritikpunkterne ved webservices-standarderne er, at webservice-konceptet er begyndt at bevæge sig væk fra det oprindelige simple koncept, der inkluderede visioner om dynamisk forbindelse til forskellige webservice, med kun begrænset interaktion mellem dem, der udbyder servicen og dem, der forbruger den til et mere komplekst koncept, som kræver en del interaktion for at forstå, hvordan webservice-standarder er implementeret i den pågældende webservice.

REST-baserede services vurderes i højere grad i dag at understøtte den oprindelige webservice-vision om dynamisk forbindelse til forskellige webservices uden menneskelig interaktion, primært fordi de ikke er bundet til brug af de mere avancerede webservice-standarder. Det betyder på den anden side, at i de forretningssituationer hvor kommunikationen mellem forretningspartnerne kræver en avanceret styring af eksempelvis transaktioner, skal de REST-baserede services håndtere dette uden brug af fælles standarder og retningslinier. Det bemærkes at, da REST typisk er XML baseret, kan man anvende mange af de eksisterende XML-standarder såsom XML Digital Signature og XML Encryption.

7.2 Opsummering

Serviceorienteringen er en tilgang til at adressere en virksomheds behov med en samling af services, der repræsenterer de komplekse og dynamiske forretningsprocesser, der selv anvender andre services. Hvordan disse services skal implementeres, er en implementeringsbeslutning, som kan anvende et bredt spektrum af forskellige teknologiske tilgange inklusiv REST og dokument-baserede webservices. Endeligt kan det også være styret af det rette værktøj til opgaven.

REST er defineret til at udstille hypermedia-information på internettet, hvilket betyder, at services til applikations til applikationsintegration, der har ligheder med internettets simple punkt til punkt-metoder, er velegnede til REST.

REST kan bruges til at løse de samme opgaver, som de nuværende relativt simple forbindelsesorienterede webservices, som udgør en væsentlig andel af webservices i dag. REST er ikke umiddelbart designet til at håndtere beskedorienterede webservices, på tværs af flere parter. Det betyder, at i mange tilfælde kan virksomheder udbyde funktionalitet som både SOAP- og REST-baserede services og opnå samme fordele. Begge dele understøtter SOA-tankegangen.

8. REST i forhold til Digitaliseringsstrategi 2007-2010

>

Digitaliseringsstrategien for 2007-2010 bygger videre på det fælles samarbejde mellem de tre forvaltningsniveauer om digitalisering (2001-2004 strategien) og 2004-2006 strategien, der satte skub i den interne digitalisering i den offentlige sektor

Målsætningen i Digitaliseringsstrategien 2007-2010 drejer sig om udvikling af borgerservice og sammenhæng på tværs af det offentlige:

- Indebærer bedre og mere forpligtende samarbejder
- Konkrete digitaliseringsstrategier vil fortsat være forankret i de enkelte offentlige myndigheder
- Muliggør bedre service og kvalitet og frigør ressourcer

Dette afsnit vil med udgangspunkt i krav fra digitaliseringsstrategien forholde sig til, hvordan REST kan indgå i digitaliseringsstrategien.

Digital kommunikation skal foregå, når det er belejligt og på måder, som borgeren og virksomhederne oplever som værdifulde og målrettede.

- REST understøtter dette gennem at være skalerbar på tværs af mange infrastrukturer, let at integrere med andre applikationer og simple at modificere i forhold til traditionelle designmønstre.

Al skriftlig kommunikation skal kunne foregå digitalt.

- RESTs styrke er at lette kombinationen af digitale informationer, og hjælper derfor ikke direkte i forbindelse med digitalisering af skriftlig kommunikation.

Automatisering og forenkling af de bagvedliggende forretningsprocesser.

- REST tilbyder en simpel adgang til data, hvorved automatisering af forretningsprocesser lettes.

Borgerportalen og virksomhedsportalen er rammen om en tilpasset og personificeret indgang til det offentlige.

- Hvis offentlige myndigheder udstiller deres information som REST-baserede services, kan det lette integrationen med borgerportalen og virksomhedsportalen.

Kommunikationen skal være målrettet borgerens dagligdag og kommunikationsmønstre.

- Udstilling af offentlige data som REST letter muligheden for, at en tredje part kan lave mashups, der er målrettet specifikke situationer og emner, som er relevant for mindre grupper af borgere.

Data tilgængelig på tværs af myndigheds- eller forvaltnings-skel.

- REST kan lette adgang til data.
-

Borger og virksomheder skal kun aflevere oplysninger til det offentlige en gang.

- REST kan lette adgang til data, hvorved det bliver lettere at dele data på tværs, og behovet for at indhente data flere gange reduceres.

Borgere og virksomheder skal opleve at få en afklaring eller en afgørelse af deres sag ved første kontakt med den offentlige sektor.

- REST kan lette adgang til data, hvorved det bliver lettere at foretage en hurtig afgørelse eller afklaring på en sag.

Levering af services via forskellige kanaler (fysisk fremmøde, telefonopkald, SMS, e-mail, internettet, system-system ...)

- REST bør også betragtes som en kanal, hvor igennem services kan leveres.

Udarbejde domæne-specifikke strategier for hvordan services bedst og mest effektivt leveres til borgere og virksomheder.

- REST kan indgå i domæne-specifikke strategier.

Brugerdreven serviceudvikling - kun i tæt kontakt med borgere og virksomheder er det muligt at tilrettelægge relevante servicetilbud, så de opfylder de væsentligste behov.

- Det må forventes, at Web 2.0 vil stille krav til de offentlige services, for at de kan opfylde borgeres og virksomheders Web 2.0-behov. REST-services understøtter tankegangen i Web 2.0.

Sikker og tryk håndtering af data i den offentlige sektor.

- REST-baserede services kan beskyttes med de samme sikkerhedsmekanismer, som bruges på internettet i dag.

Flest mulige af de administrative rutiner automatiseres og forenkles.

- REST letter adgangen til data, hvorved det bliver lettere at automatisere og forenkle administrative rutiner.

Største gevinster opnås i sammenhæng med gennemførelse af organisatoriske forandringer og ændrede arbejdsgange.

- REST letter adgangen til data, hvorved det bliver lettere at ændre arbejdsgange.

Der skal sættes klare og målbare mål for digitaliseringen.

- Det bør overvejes, hvorledes der kan sættes mål op for udbud af REST-services.

Gevinster skal kvantificeres og dokumenteres (effektvurdering).

- Det bør overvejes, hvorledes der foretages effektvurdering af REST-services
-

9. Eksempler på hvor REST kan anvendes

>

Til illustration af hvor REST-baserede services kan anvendes, indeholder de følgende afsnit nogle eksempler på, hvordan REST kan bruges i eksisterende løsninger

9.1 Danmarks Miljøportal

Danmarks Miljøportal blev i januar 2007 oprettet som en fælles portal med én indgang til de relevante miljødata for at underbygge visionen for digitalisering på miljøområdet om at opnå en effektiv, digital forvaltning på miljøområdet. I visionen er der opstillet et antal pejlemærker, som den langsigtede it-arkitektur for miljøområdet bør opfylde:

- Relevante miljødata skal være fleksibelt organiseret, så der er åben og ubesværet mulighed for anvendelse i formidling og myndighedsbehandling på tværs af forvaltningsgrænser.
 - REST kan lette anvendelsen af miljødata.
- Data skal være aktuelle og kvalitetssikrede og dermed udgøre et validt og fyldestgørende grundlag for en effektiv sagsbehandling.
 - REST kan lette adgangen til aktuelle data.
- Data skal muliggøre en fyldestgørende afrapportering og sammenstilling af data på tværs og vertikalt mellem de administrative enheder.
 - REST kan lette sammenstilling af data på tværs og vertikalt.
- Data og dataudveksling skal være baseret på vedtagne fælles standarder indenfor de forskellige faglige domæner, således at funktionaliteter (services) og data kan stilles til rådighed for forskellige myndigheder i sagsbehandlingen.
 - RESTs simple arkitekturprincipper kan lette tilgangen til data.
- Digitale services skal være sammensat med borgere og virksomhedernes behov i fokus, uanset at data og informationer bliver født af forskellige myndigheder, og uanset at flere myndigheder indgår i sagsbehandlingen.
 - REST kan lette sammenstilling af data til forskellige og mindre grupper af borgere og virksomheder.

Danmarks Miljøportal har 4 grupper af interessenter, der har forskellige behov og krav til portalen:

- **Myndigheder** der omfatter landets kommuner, regionerne, de statslige miljøcentre og ministerierne med tilhørende styrelser. De har et krav om stabil og hurtig adgang til aktuelle og valide miljødata.
- **Andre professionelle brugere** der omfatter virksomheder og institutioner, hvis produktion og leverancer gør brug af data og funktionalitet på Danmarks Miljøportal. Det er for eksempel konsulenter, rådgivere, laboratorier, forskningsinstitutioner,

leverandører af it-systemer til miljøopgaver, interesseorganisationer og pressen. Denne gruppe har krav om stabil og hurtig adgang til aktuelle og valide miljødata. De har også et ønske om fleksible og robuste muligheder for at kunne integrere egne it-løsninger med services fra Danmarks Miljøportal.

- **Borgere** der søger information og viden om miljøet i Danmark. De har et krav om, at det skal være nemt og entydigt at finde egentlige og berigede miljødata. Borgere vil have glæde af at kunne downloade data om miljøet til håndholdte enheder som mobiltelefon, PDA og GPS for på den måde at have adgang til data om miljøet, mens man færdes i det.
- **Virksomheder** der omfatter virksomheder, hvor produktionen påvirker miljøet, og de virksomheder hvor reguleringen af miljøet påvirker virksomhedens produktionsforhold. Det er for eksempel kommunale driftsorganisationer (vand, spildevand og affald), landbrug, dambrug og produktionsvirksomheder med miljø-emmissioner. De har et krav om, at det skal være nemt og entydigt at finde egentlige og berigede miljødata. Det vil typiske gælde miljødata om virksomheden eller miljødata, der har betydning for virksomhedens produktionsforhold. Et eksempel er arealreguleringer i det åbne land, som har betydning for landbrugserhvervet. Virksomheder vil også have glæde af at kunne downloade data om miljøet til håndholdte enheder som mobiltelefon, PDA og GPS.

Danmarks Miljøportals opgave inkluderer at udstille stedfæstede data via internettet, et område hvor REST bruges i øjeblikket på internettet, nedenstående er eksempler på, hvor Danmarks Miljøportals REST-baserede service kunne indgå i mashups:

- En mashup der henter servitutter på en ejendom fra en service udstillet af Tinglysningsretten samt jordforurening fra en service hos Danmarks Miljøportal og viser begge dele på et satellitkort fra Google Earth.
- En mashup fra Dansk Ornitologisk Forening der henter information om fugleobservationer fra Danmarks Miljøportal og kombinerer det med information om Margerittruten og shelter-placeringer og viser data på baggrund af et kort fra Areal-information

9.2 Infostrukturbasen

Når et element registreres i Infostrukturbasen, kan man vælge også at relatere det til sproglige termer hos DANNET² (<http://www.cst.dk/dannet/index.html>).

² DanNet er et forsknings- og udviklingsprojekt der går ud på at udarbejde et dansk leksikalsk-semantic ordnet; dvs. en sprogsresource hvor ords betydningsstruktur og interne relationer er udtrykt i et formelt sprog og derved gjort anvendelige for IT-systemer, der arbejder med intelligent informationshåndtering

>

Derved registreres hvilke søgeord, som matcher det pågældende element. Til et element kan der defineres en REST-service, som henter information, der viser svaret på den pågældende søgning.

Denne funktionalitet kan tilbydes som en generel service, som alle myndigheder kan implementere i deres søgeløsning, hvorved svaret på en søgning kan gives umiddelbart til brugeren. Brugeren undgår derfor at bladere gennem søgeresultater for selv at finde svaret. Listen over søgeresultatet bør forsat vises, da det skal være muligt for brugeren forsat at finde relevant information i henhold til søgeordet.

10. Opsummering

>

REST er med succes blevet brugt som basis for flere udvidelser til det oprindelige internet. Det er derfor nødvendigt, at offentlige myndigheder i de situationer, hvor offentlige services kan udvide det oprindelige internet, anvender REST, for dermed at sikre, at dets services indgår naturligt på internettet

Der ses i øjeblikket få forretningsmæssige implementationer af REST-baserede services, men da REST-baserede løsninger vil kunne håndtere de fleste typer af forretningsintegration, der anvendes i dag, er det en relevant mulighed at overveje til system-system integration. Eksempelvis kan det være en offentlig strategi, at data fra brugergrænsefladen også udstilles via REST-baserede services, så disse er lige så let tilgængelige for mennesker som for maskiner.

REST kan lette tilgangen til at anvende og kombinere offentlige data og vil være et element til at opfylde Digitaliseringsstrategien 2007-2010.

11. Tekniske aspekter

>

I denne del af dokumentet bliver de tekniske aspekter ved REST gennemgået. De fleste afsnit kommer med anbefalinger til, hvordan REST implementeres, men det er også vigtigt at understrege, at REST i høj grad er en pragmatisk indgangsvinkel til implementation af web-services, og at det i specielle scenarier kan give mening at vælge andre løsninger end dem, der er skitseret her.

11.1 Specifikation

Der kan opstilles mange formål med en service-specifikation, og en del af disse vil være modstridende. Fokus i REST er på at implementere services via nogle få standardiserede principper, og det bør medføre en relativt simpel servicespecifikation. Erfaringer har vist, at selv stramt definerede WSDL-specifikationer ikke nødvendigvis sikrer interoperabilitet. Det betyder, at de fordele, WSDL og dermed en maskinlæsbar specifikation giver, er ret begrænsede. På den baggrund giver det mest mening at holde servicespecifikationerne på et niveau, hvor de er forståelige for de udviklere, der skal anvende de specificerede services.

Uanset formen og formålet vil en eller anden form for specifikation være påkrævet. Generelt set er der tre oplagte muligheder:

- WSDL 2.0
- WADL
- Fritekst

Hver af de ovenstående metoder har fordele og ulemper. Den langt mest udbredte er fritekst-metoden, hvor REST-services beskrives i almindelig tekst, f.eks. på en wiki eller en anden webside. Det er med denne form ikke muligt at lave kodegenerering, da der ikke findes nogen formel specifikation, som værktøjerne i givet fald kan læne sig op ad.

WSDL, formentligt i version 2.0 [WSDL2], er den oplagte mekanisme til at lave en formel specifikation. WSDL er velkendt i forbindelse med SOAP-baserede webservices, men er også kendetegnet ved stor kompleksitet, og på trods af en anden intention, så er interoperabilitet ikke garanteret. WSDL 2.0 er en nødvendighed pga. den nye HTTP Binding, men værktøjsunderstøttelsen for denne version er endnu ukendt. I tilfælde af, at WSDL anvendes, bør man standardisere [OIOWSDL].

WADL [WADL] er Sun's bud på WSDL til REST-services. Ligesom med WSDL giver WADL en formel specifikation af en service, og denne specifikation kan bruges til at generere kode ud fra, evt. automatisk. WADL understøtter både XML Schema og Relax NG som specifikation af datatyper.

>

I OIOREST anbefales en tekstuel servicespecifikation. Det sker ud fra følgende overvejelser:

- Kald til REST-services sker typisk gennem en generisk HTTP-klient via en URL og evt. noget data. Det, der er nødvendigt for at kalde en service, er derfor URL samt datatype, og både WSDL og WADL er to komplicerede måder at udtrykke disse to ting på
- Det giver sjældent mening at autogenerere en klient eller serviceskeletons for en REST-service

Servicespecifikationerne er derfor primært rettet mod udviklere, og her er en tekstuel specifikation mere forståelig end både WSDL og WADL

Bemærk, at dette betyder ikke, at specifikationen er eller skal være ukontrolleret! Det anbefales, at specifikationen udformes som et dokument (eller lignende), der indeholder følgende:

- Generelle designprincipper, f.eks. for URLs
- Ressourcetyper der er i anvendelse
- Hvilke HTTP headers der anvendes
- Generel fejlhåndtering .Requests der resulterer i en 4xx eller 5xx status bør returnere et feildokument af kendt struktur. Se nedenfor for forslag til struktur.
- Dataformater og mime types
- For hver ressource: Mulige operationer (GET, PUT osv.), mulige statuskoder, anvendte datatyper
- Sikkerhedsmekanismer

Et præcist format for denne specifikation er ikke påkrævet, men nedenfor er et eksempel på, hvordan specifikationen kan udformes. Der lægges vægt på, at specifikationen er forståelig, og at de nødvendige informationer er til stede.

Denne form for specifikation falder også godt i tråd med, hvad de større REST-services på nettet gør. En af de eneste udbydere, der selv tilbyder WADL er Yahoo!, men de primære specifikationer er normale websider, der beskriver de ovenstående punkter. Se f.eks. YAHOOEST, AMAREST og SMUGAPI.

Eksempel på servicespecifikation

Eksemplet er baseret på en tænkt "Danmarksportal" og er ikke en komplet specifikation, idet en del ressourcer ikke er beskrevet.

Service URL: <http://oiorest.dk/Danmark>

Under denne URL findes en række ressourcer, der kan hentes via hierarkisk opbyggede adresser:

Ressource	URL	Metode	Repræsentation	Beskrivelse	Svar
-----------	-----	--------	----------------	-------------	------

>

Regioner	/regioner	GET	regioner.xsd	Lister alle regioner	200
	/regioner/{regionnr}	GET	region.xsd	Hent information om en region	200, 404
POI	/poi	GET	Pois.xsd	Lister alle POIs	500
	/poi	POST	poi.xsd	Opret ny POI	201, 500, 400, 401
	/poi/{id}	GET	poi.xsd	Hent POI	200, 404, 500
	/poi/{id}	HEAD	-	Hent header-information	200, 404, 500
	/poi/{id}	PUT	poi.xsd	Opdater POI	200, 400, 401, 404, 500
	/poi/{id}	DELETE	-	Slet POI	200, 401, 404, 500

Alle metoder, der anvender POST, PUT og DELETE, kræver HTTP Basic auth i form af en Authorization-header. Sendes denne ikke med, svares der med en 401-kode (Unauthorized).

Statuskoder

200: Success

201: Ny ressource oprettet. Location-header indeholder URL til den oprettede ressource

400: Request ikke gyldigt. Det sendte XML kunne ikke valideres.

401: Requestet kunne ikke autentificeres

404: Den forespurgte ressource kunne ikke findes

500: Intern serverfejl

Caching

Alle GET-requests returnerer caching-headers i form af Etag og Last-Modified, som kan sendes med i fremtidige requests. Hvis en af disse sendes med i If-None-Match eller If-Modified-Since, og ressourcen ikke er ændret, returneres en 304-kode.

>

Fejlstruktur

I forbindelse med returnering af fejl, er det ofte hensigtsmæssigt at strukturere fejlmeldingen på en standardiseret form. Følgende type foreslås som standardformat til returnering af fejlkoder:

```
<fault>
  <code>404</code>
  <parameters>
    <parameter key="region">23</parameter>
  </parameters>
  <description>Unable to find requested
region</description>
  <stacktrace>
    ...
  </stacktrace>
</fault>
```

Bemærk, at ovenstående blot er tænkt som et eksempel og som inspiration. Der er formentlig tilfælde, hvor der skal returneres et document, således at <parameter>-tag'et ikke er dækkende.

En fejl består altså af en maskinlæsbar kode, der automatisk kan oversættes til tekst. Dette kræver, at en del af servicebeskrivelsen indeholder de fejlkoder, der kan blive sendt tilbage. Til fejlkoden kan der være en række parametre; i eksemplet bliver der f.eks. spurgt efter region 23, som ikke eksisterer. En af fejlparametrene er derfor region=23, så denne værdi kan sættes ind i den genererede fejltekst.

Som optionelle elementer ligger description og stacktrace, der kan indeholde en læselig beskrivelse af fejlen samt et stacktrace fra serveren. Stacktracet vil formentligt kun blive sendt med i udviklingsfasen.

I det følgende opridses en række generelle guidelines for, hvordan REST-baserede webservices implementeres. Flere guidelines kan findes i Leonard Richardsons og Sam Rubys bog "RESTful Web Services".

11.1.1 Hierarkiske URLer

En URL bør være opbygget hierarkisk, så de generelle elementer ligger først, og de specialiserede ligger sidst, f.eks.

<http://oiorest.dk/regioner/23/poi>, som viser en regions interessepunkter.

11.1.2 Læselige URLer

Det bør være muligt ud fra en URL at se, hvad ressourcen består af, dvs.

<http://oiorest.dk/regioner/23>
frem for

>

<http://oiorest.dk/33313k4ks8c8v72>.

11.1.3 Ressouece-orienterede URLer

En URL bør ikke indeholde information omkring en metode eller procedure, da disse ligger i HTTP-metoderne. Dvs.

<http://oiorest.dk/regioner/23>

frem for

<http://oiorest.dk/getRegion/23>

11.1.4 Giv alle ressourcer en URL

Et ofte set eksempel er valg af sprog. F.eks. kan

<http://oiorest.dk/ergioner/23>

give regionsinformationer i default-sproget mens

<http://oiorest.dk/regioner/23.en>

giver det på engelsk.

11.1.5 POST og PUT

Brug POST ved oprettelse af nye ressourcer, hvor klienten ikke selv kender den nye ressources URL, og brug PUT i alle andre tilfælde, hvor der tilføjes eller opdateres data.

Det bør desuden overvejes, om (læs: hvordan) POST giver den ønskede pålidelighed.

11.1.6 Undgå så vidt muligt service-specifikke HTTP-headers.

Ved at undgå disse mindskes kravene til klienter, og servicen bliver lettere at debug'e.

11.2 Repræsentation

Når en ressource hentes eller manipuleres, kræves der en repræsentation af ressourcen. Ofte falder valget på XML, og i OIO-sammenhæng sker det via XML-skema-definitioner. Der er dog også alternativer, og i REST er det så vidt muligt et spørgsmål om at finde det mest simple.

Nogle af de oftest brugte repræsentationer er:

- Skema-specificeret XML, f.eks. OIOXML
- Registrerede og beskrevne mime types [MIME]
- Atom [ATOM]
- RDF [RDF]

>

- XHTML/Microformats [MCF]
- Udefineret XML, typisk objekter serialiseret direkte til XML
- XHTML

Det vigtige i forhold til en repræsentation er, at den udtrykker de centrale elementer i en ressource på en veldefineret måde. I REST lægges der meget vægt på genbrug og simple repræsentationer. Det betyder, at hvis en ressource kan lægges ind i en eksisterende repræsentation, så er der ingen grund til at opfinde en ny. Det gør sig især gældende i forhold til XML-dokumenter, hvor det er meget let at definere nye typer. Her er det især vigtigt at undersøge, om der i forvejen eksisterer typedefinitioner, der kan bruges til det givne formål.

Et vigtigt princip i repræsentationen er desuden 'connectedness', som går ud på, at alle ressourcer eksplicit hænger sammen. Det betyder, at hvis <http://oiorest.dk/regioner/23/poi> giver en liste af interessepunkter for region 23, så indeholder listen ikke bare en række id'er på punkterne; den indeholder rent faktisk en række links til de enkelte punkter. På den måde behøver en klient ikke selv at beregne adresser og bliver dermed simplere.

Input-repræsentation behøver ikke nødvendigvis at være den samme som output, men i de fleste tilfælde giver det god mening. Der er dog enkelte tilfælde, hvor andet giver mening. Det er især i tilfælde, hvor input består af simple key/value-par. I så fald kan input med fordel sendes som application/x-www-form-urlencoded, hvilket er den samme repræsentation, som almindelige web-browsere anvender, når de POSTer til en side.

Anbefalingen til REST-repræsentationer er:

- Brug eksisterende typer så vidt muligt.
- Hvis der anvendes XML, så definer så vidt muligt XML-skemaer i henhold til OIOXML. Dette er dog ikke nødvendigt, hvis skemaet er meget specifikt til den pågældende opgave og derfor med sikkerhed ikke skal anvendes i andre henseender.

Fordelen ved at anvende en af de eksisterende typer er, at der ofte findes biblioteker til at håndtere disse typer, og dermed kan klienterne genbruge kode i stedet for at skrive det selv. Det er f.eks. gældende for RDF, Atom og en lang række andre typer. Ulempen kan til gengæld være, at domænemodellen ikke passer helt ind i den model, der ligger bag typerne, så det er altid nødvendigt med en vurdering af, om typerne er anvendelige.

Hvis der defineres et nyt XML-skema for ressource typerne, giver det en række andre fordele. For det første er typen veldefineret. Det betyder, at det er klart for klienterne, hvilke elementer typen består af. For det andet giver det mulighed for, at klienten kan generere kode udfra XML-skemaet, og dermed kan repræsentationerne blive integreret ind i det sprog, klienten udvikles i. Dog

>

har kodegenerering ofte den ulempe, at de forskellige værktøjer genererer kode på forskellige måder, og disse er sjældent interoperable.

En fremgangsmåde til at finde en repræsentation kan findes på <http://rest.blueoxen.net/cgi-bin/wiki.pl?WhichContentType>

11.3 Sikkerhed

Når vi i daglig tale taler om sikkerhed, dækker det for det meste over sikkerhed på flere niveauer. De tre umiddelbare er:

- *Authentication*: Hvem ejer det igangværende request?
- *Authorization*: Hvad har den givne bruger lov til?
- *Fortrolighed*: Kryptering? Trust-mekanismer?

Domænemodel og User-begrebet

Begrebet "Bruger" eller "User" er oftest en del af domænemodellen. Alene det faktum, at vi her har valgt at skrive "User" i stedet for "Username", argumenterer ganske godt hvorfor: Selvom brugeren i mange situationer identificerer sig ved at indtaste sit navn (username), så er dette ofte blot en af flere attributter på "Bruger", hvor f.eks. "rolle", "gruppe", "password" osv. vil kunne være andre attributter på domæne-typen "Bruger". I mange tilfælde er brugeren endda blot en specialisering af en "Person"-type i domænemodellen. Hvis vi f.eks. forestiller os en elektronisk patientjournal (i daglig tale et EPJ-system), er det oplagt, at der eksisterer flere roller eller persontyper. Lægens rettigheder er formentlig forskellige fra sygeplejerskens, sekretærens og systemadministratorens. Udover, at det handler om autentifikation og autorisation, så har begrebet læge/sygeplejerske også en funktionel betydning for systemet og er således en del af domænemodellen. Hvis man så dertil lægger, at systemet måske er integreret med apotekerne eller sundhed.dk, introducerer det nye persontyper for dele af systemet. GUI'en skal tegnes forskelligt afhængigt af brugeren - hvem er logget ind, og hvad må hun? Domænemodellen og sikkerheds-begreberne har således et overlap, og begrebet "bruger" er blevet til en domænetype på linie med eksempelvis "indlæggelse" og "medicinordination", eller om man vil, "bruger" er både en security-constraint og en ressource i systemet.

Andre overvejelser gør sig gældende vedr. transport-laget og transport-mekanismerne. Der er ingen grund til at opfinde noget, som allerede er lavet og standardiseret. SSL-kryptering og http-authentication vil i langt de fleste tilfælde yde den ønskede sikkerhed og vel at mærke med velafprøvede og velunderstøttede teknikker.

>

Det er vigtigt at holde sig for øje, at sikkerhed og kompleksitet ikke er proportionale størrelser - forstået på den måde, at en mere kompleks sikkerhedsmodel ikke nødvendigvis sikrer et mere sikkert system. Formentlig er det modsatte tilfældet, nemlig at jo større kompleksitet, man introducerer i sikkerhedsmodellen, jo større overflade har man eksponeret ud mod eventuelle ”angribere”.

Anbefalingen er således, at begreberne, der knytter sig til authorization og authentication (bruger, rolle, gruppe, rettighed, osv.) modelleres som ressourcer på linje med andre domænetyper, hvor det giver mening.

Autentifikation af klient

HTTP giver mulighed for tre standardmetoder til at autentificere requests: *HTTP Basic auth*, *HTTP Digest auth* og *SSL med klientcertifikat*.

Med HTTP Basic auth sendes brugernavn og password i en header med hvert request, og serveren checker disse op mod en brugerdatabase. Denne metode er langt den simpleste, men medfører at brugernavn og password transporteres i et læseligt format. Derfor anbefales det altid at køre SSL når der anvendes basic auth, så brugernavn og password ikke kan opsnappes.

HTTP Digest er et alternativ til Basic auth, hvor brugernavn og password ikke sendes i klartekst. I stedet sendes et hash af brugernavn, password og request-url. Serveren kan genskabe dette hash og på den måde sikre sig, at klienten er den rigtige. Ulempen ved Digest er, at serveren skal kende brugerens password for at beregne hash-værdien, og ofte ligger passwordet ikke i klartekst på serveren, hvilket kan vanskeliggøre Digest-metoden.

Den sidste metode er autentifikation med klientcertifikat, også kendt som 2-vejs SSL. Her foregår ingen udveksling af brugernavn og password; i stedet sender klienten sit certifikat, f.eks. et OCES-certifikat, over en SSL-forbindelse til serveren. Serveren checker, om certifikatet er udstedt af en gyldig autoritet, og om det er udløbet. Hvis certifikatet godkendes, kan certifikatets attributter læses på serveren.

I de fleste tilfælde er Basic auth over SSL at foretrække, da det er langt den simpleste metode, hvorimod klientcertifikater kan anvendes når højere sikkerhed er krævet.

For alle metoderne er det også gældende, at det ikke er slutbrugerens identitet, der sendes til servicen. Ofte er det sådan, at en bruger besøger en webside, der kalder en REST-service for at få fat i noget data. Brugeren kan godt sende sin identitet til websiden, men der findes ikke nogen standardmekanisme, hvorved denne identitet kan sendes videre til den bagvedliggende service på en sikker

>

måde. Derfor er det ofte sådan, at bagvedliggende services må stole på, at brugerne er blevet korrekt autentificeret, inden servicen kaldes.

End-to-end security

I nogle applikationer er der behov for *end-to-end security*. Det betyder, at selvom en besked sendes igennem flere forskellige services, er det stadig kun den endelige modtager, der kan se indholdet af beskeden. Dette er ikke umiddelbart muligt i ren HTTP, da kommunikationen her er point-to-point. Det betyder, at services, der ligger mellem afsenderen og modtageren, også har mulighed for at se beskederne, selvom afsenderen har sendt over en SSL-forbindelse.

Hvis der er behov for end-to-end security er det derfor nødvendigt at kryptere indholdet af et request. Nogle standarder, f.eks. Atom Syndication Format, definerer muligheder for at indsætte krypterede elementer for på den måde at sikre beskederne, men det er langt fra alle formater, der er forberedt på den måde. I de tilfælde er REST formentligt et dårligt valg, hvis ikke formatet kan udvides på en passende måde. SOAP-baserede webservices kan her anvende WS-Security, der definerer hvordan signaturer og krypterede beskeder kan placeres i SOAP-headeren. For HTTP eksisterer der ikke en lignende standard, hvorfor det er nødvendigt, at selve dataformatet understøtter signering og kryptering.

11.4 Fejlhåndtering

Fejlhåndtering i applikationslaget er omtalt i afsnittet om specifikation, hvor der er givet et eksempel på, hvordan en fejlmeddelelse kan se ud, og hvordan de forskellige HTTP-statuskoder kan anvendes.

Der mangler dog stadig et andet aspekt af fejlhåndteringen, nemlig hvad der sker, hvis en klient sender et request, men ikke får et svar tilbage. Det manglende svar kan skyldes en række forskellige omstændigheder som f.eks., at klienten brød sammen, netværket blev afbrudt, serveren brød sammen, mens requestet blev processeret osv.

Generelt set er der behov for at håndtere alle de tilfælde, hvor klienten ikke modtager et HTTP-svar fra serveren.

REST-baserede webservices er som tidligere nævnt bygget op omkring GET, POST, PUT og DELETE. På nær POST har de alle den egenskab, at de er idempotente – dvs. der sker ikke noget ved at kalde dem igen og igen. I nogle tilfælde, f.eks. DELETE, vil svaret dog ikke nødvendigvis være det samme ved gentagne kald, men der vil i alle tilfælde komme et gyldigt svar fra serveren, og tilstanden på serveren ødelægges ikke.

Hvor de andre operationer fungerer enten på enkelte elementer eller hele collections, har POST den egenskab/ulempe, at den har sideeffekter, fordi den ofte bruges til at tilføje et element til en collection. Dermed kan POST ikke

>

kaldes igen og igen uden at ødelægge tilstanden på serveren. Der er dog metoder til at om dette, f.eks. POST Exactly Once (POE), som i princippet kan anvendes til helt at undgå POST.

POE fungerer ved, at klienten i stedet for at lave et direkte POST-request først laver et GET- (eller HEAD-)request til ressourcen. Serveren genererer herefter et nyt unikt ressource-id, som returneres som en URL via POE-Links-headeren. Ved dette kald ændrer serveren ikke tilstand, den allokerer blot et id, som kan bruges i fremtidige requests. Denne nye URL kan bruges til at lave et POST-request én gang. Laves der flere POST-requests, fejler kaldet med en 405 (metode ikke tilladt). I stedet for at lave POST, kan den genererede URL også betragtes som ressourcens nye id, og der kan i stedet laves en PUT til adressen – herved undgås POST helt.

En generel algoritme til at lave kald til REST-baserede webservices kan eksempelvis se således ud:

```
Client = new RestClient("http://oiorest.dk/Danmark/regioner/23");
while (true) {
    Result res = client.get();
    if (res.isSuccess()) {
        break;
    } else {
        sleep (2000);
    }
}
```

Her laves et (GET-)request på en ressource, og hvis svaret fejler, ventes der 2 sekunder, og så prøves der igen, indtil kaldet går godt.

Denne simple form for fejlhåndtering betyder også, at selve mekanismen til at gensende requests kan pakkes helt ind, så requests automatisk genseses i tilfælde af fejl. Teknikken betyder også, at ansvaret for at håndtere fejl primært ligger hos klienten og dermed ikke komplicerer de enkelte services. En klient kan den måde selv bestemme, hvilken fejlhåndteringsstrategi, der skal anvendes, ligesom klienten kan anvende et standard-bibliotek, der håndterer fejl på passende vis.

(Bemærk i øvrigt, at ved en permanent fejl vil ovenstående algoritme aldrig terminere. I den slags tilfælde bør der være en retry-count, som sikrer, at algoritmen før eller siden vil afbrydes.)

11.5 Stateless opførsel

Pålidelighed

Serversiden er stateless og idempotent. Dette er en vigtig parameter i pålideligheden: Da en vilkårlig forespørgsel fra klienten, dvs. fra en vilkårlig

>

klient, ikke medfører noget stateskift på serveren, vil samme forespørgsel (fra en vilkårlig klient) altid returnere nyeste version af ressourcen. Dette giver dermed mulighed for at lave passende – og ret simple – retry-mekanismer i klienten.

Det er dog vigtigt at holde for øje, at når vi beskæftiger os med REST-baserede webservices, støder vi naturligt på de sædvanlige problematikker mht. validitet af data. Når data forlader serveren, kan man ikke længere forvente, at de til hver en tid stemmer overens med de data, som ligger på serveren. Problemstillingen kan i bund og grund koges ned til, at når to klienter tilgår og ændrer samme data, må den ene af dem nødvendigvis tabe i kampen om, hvem der har de nyeste data. Det er naturligvis stærkt applikationsafhængigt, om dette er et reelt problem, men når vi taler pålidelighed og stateless opførsel kommer vi ikke udenom, at man i designet af applikationen skal tage stilling til begreber som optimistisk låsning og evt. versionering af data.

Caching

HTTP-caching betyder i al sin enkelhed, at responset på et GET- (eller HEAD-) request kan lagres på klientsiden, så efterfølgende kald af samme request ikke kræver et round-trip over nettet. Dette er således med til at begrænse netværkstrafikken og serverbelastningen, og ikke mindst kan det være stærkt performance-optimerende.

Bemærk her, at en eventuel proxy-cache set fra serversiden også er en klient. Caching er i øvrigt beskrevet i dybden i afsnit 13 i RFC 2616.

Der findes basalt set to måder at implementere caching på: Udløb og validering. Caching via udløb betyder, at serveren giver en tidsramme, hvor en bestemt ressource kan caches, hvorimod validering giver en nøgle til ressourcen, som sendes med i alle requests. Hvis ressourcen ikke har ændret sig, sendes ressourcen ikke tilbage når der kommer et request.

Caching via udløb

Cache-kontrollen ligger på klient-siden, men er dikteret fra serveren ud fra en eller flere HTTP-headers:

`Expires: <dato>`

Dette angiver, hvor længe responset kan cache's. Bemærk, at tidspunktet angivet ved <dato> blot angiver, hvornår pågældende data kan være forældet – ikke at det nødvendigvis vil være tilfældet. Det er således op til klienten, om den vil slette pågældende cache-entry, eller om den vil lave en ny forespørgsel og eventuelt derigennem fremskrive tidspunktet for næste udløb/tjek.

>

Bemærk også, at det eksakte tidspunkt for 'data-udløb' formentlig sjældent kan forudsiges med synderlig god præcision, så det handler mere om performance-optimering af hyppige kald end om egentlig forudsigelse af data-levetid.

En alternativ mulighed for kontrol af levetiden af det cachede response er gennem headeren:

```
Cache-Control: max-age=<antal sekunder>
```

Her er man fritaget for at tage stilling til et eksakt tidspunkt i fremtiden, men kan nøjes med at forholde sig til den faktiske levetid for responset eller til 'cache refresh intervallet'.

Værdien af Cache-control kan desuden sættes til no-cache, hvilket angiver, at data/response ikke skal (læs: må) caches:

```
Cache-Control: no-cache
```

En eventuel proxy mellem klient og server kan reagere på værdien af Cache-Control. For at forhindre dette, kan man sætte værdien til:

```
Cache-Control: private
```

Dette angiver, at svaret må caches af klienten, men ikke af eventuelle mellemlid.

Hvis ingen af overstående direktiver angives, falder klienten tilbage på en default opførsel, som desværre mest beror på en række uskrevne regler og almindelig praksis. Hvis der ikke anvendes andet, anbefales det derfor at anvende

```
Cache-Control: no-cache
```

og

```
Cache-Control: max-age=<antal sekunder>
```

i de tilfælde, hvor man er nødt til at kunne forudsige og stole på cachingen på klient-siden.

Caching via validering

I stedet for at serveren fortæller klienten, hvor længe klienten må opbevare en ressource, giver serveren i stedet et entity tag via ETag-headeren. Dette er et tag til en bestemt ressource-version, f.eks. en md5-sum af ressourcens indhold. Dette tag kan klienten gemme sammen med selve ressourcen, og i fremtidige requests til ressourcen kan værdien af ETag medsendes i If-None-Match-headeren. Hvis ETag-værdien stadig matcher serverens ressource, svarer

>

serveren med en 304 Not Modified, og klienten kan bruge den cachede ressource.

Caching via validering anvendes oftest, når der er ønske om caching af ressourcer, men hvor ressourcernes indhold skal være opdaterede hele tiden. Det betyder, at hver gang ressourcen tilgås, checkes det, om der på serveren findes en nyere version. Dette er normalt meget hurtigt, men kræver stadig et request til serveren. Denne form for caching kræver ekstra funktionalitet på serveren, da det er nødvendigt hurtigt at kunne se, om en ressource er ændret. Er der ikke helt så strenge krav til, om data er opdaterede, kan caching via udløb passende anvendes. Ofte anvendes dette til caching af statiske billeder og stylesheets, som kun ændrer sig meget sjældent. I modsætning til caching via validering kræver caching via udløb ikke nogen ekstra funktionalitet på serveren, da der blot udsendes en header med informationer om, hvor længe en given ressource er gyldig. I begge tilfælde kræves en vis funktionalitet på klienten, men igen er caching via validering mere kompliceret at implementere.

12. Kodeeksempler

>

Som eksempel forestiller vi os, at IT- og Telestyrelsen udstiller en række ressourcer gennem en applikation kaldet Danmarksportalen. Ideen er således, at man kan forespørge eksempelvis efter regionerne, og på de enkelte regioner kan man forespørge på kommuner, postdistrikter, specifikke adresser og lignende. Vi forestiller os således, at man på URL'en <http://oiorest.dk/regioner> kan hente en liste af regioner, og eksempelvis på regionen med kode/nummer 1080 kan tilgås på <http://oiorest.dk/regioner/1080>

Lad os se på, hvordan http-request'et eksempelvis realiseres i Java, .Net, Javascript og Ruby.

12.1 Java

Selvom der over tid er opstået flere tredje-parts-implementationer af forskellige HTTP-biblioteker, vil vi holde os til den implementation, der som standard følger med Java (URLConnection), da den dels er meget simpel at anvende og dels med sikkerhed er til stede i en given Java-installation:

```
URL url = new URL("http://oiorest/Danmark/regioner/1080");
URLConnection con = url.openConnection();

BufferedReader br = new BufferedReader(
    new InputStreamReader(con.getInputStream()));

String line;

While ((line = br.readLine() != null) {
    ...doStuff...
}
```

Bemærk, at for en http-url defaultes ovenstående eksempel til et GET-request. Hvis man ønsker et andet request, skal man benytte sig af, at `url.openConnection()` returnerer en `URLConnection` af typen `HttpURLConnection`, når man åbner en http-url, for på en connection af den type, kan man (bl.a.) sætte `requestMethod`:

```
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("PUT");
con.setDoOutput(true);
OutputStream out = con.getOutputStream();
out.write(...)
```

Bemærk også, at man på en `HttpURLConnection` kan læse og skrive, men pr. default er den kun sat op til læsning; derfor skal man eksplicit kalde `con.setDoOutput(true)`;

>

Der er som sagt andre frameworks som eksempelvis Apache's HttpClient. Disse frameworks tilbyder ofte flere muligheder for at styre connection pooling, timeouts, lifecycle osv. Disse frameworks er tit at foretrække i større systemer, men for den rå http-funktionalitet er Java's "java.net.*"-pakke fuldt tilstrækkelig.

12.2 .Net

.Net har et framework giver en programmatisk tilgang til Http-protokollen i stil med den, vi så i Java-implementationen:

```
HttpRequest request =
(HttpWebRequest)WebRequest.Create("http://oiorest.dk/Danmark/regioner/1081");
request.Method = "GET";
using (WebResponse response = request.GetResponse()) {
    using (StreamReader reader = new
        StreamReader(response.GetResponseStream())) {
        > Console.WriteLine(reader.ReadToEnd());
        > }
        > doStuff...
    > }
}
```

Bemærk, at Http-metoden sættes vha. `request.Method = "GET"`. Hvis der angives et metode-navn, som ikke er en kendt Http-metode, eller hvis det staves forkert (f.eks. "Get" i stedet for "GET"), resulterer det i en runtime-exception.

12.3 Ruby

I Ruby ser det tilsvarende eksempel således ud:

```
require 'net/http'
url = URI.parse('http://oiorest.dk/Danmark/regioner/1081')
request = Net::HTTP::Get.new(url.path)
result = Net::HTTP.start(url.host, url.port) do |http|
    http.request(request)
    doStuff...
end
```

Hvis man i stedet for Get, ønsker at kalde en anden Http-metode, laver man blot en instans heraf og anvender den i stedet:

`Net::HTTP::Put` eller

>

`Net::HTTP::Post` eller
`Net::HTTP::Delete`

12.4 Javascript

I Javascript-verdenen er tingene en anelse mere komplicerede; først og fremmest fordi udviklingen af javascript traditionelt ikke har været underlagt samme strikse styring og standardisering som de ovenfor nævnte programmeringssprog, og dermed er der opstået forskellige dialekter i de forskellige browsere. De seneste år har dette dog ændret sig, dels fordi javascript er blevet mere strømlinet (bl.a. styret af frameworks som AJAX/GWT), og dels fordi browsermarkedet efterhånden har stabiliseret sig omkring Firefox og Internet Explorer.

Eksemplet baserer sig på `XMLHttpRequest` objektet, og det beskrives, hvordan koden ser ud på hhv. Firefox og Internet Explorer. Forskellen består alene i, at `XMLHttpRequest`-objektet er et ActiveX-object kaldet `XMLHTTP` i IE. For at skabe `XMLHttpRequest`-objektet, gør vi følgende:

```
var xmlhttp = null;
if (window.XMLHttpRequest) {
    xmlhttp = new XMLHttpRequest();
    if ( typeof xmlhttp.overrideMimeType != 'undefined' ) {
        xmlhttp.overrideMimeType('text/xml');
    }
} else if (window.ActiveXObject) {
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
} else {
    alert('Din browser understøtter ikke xmlhttprequests?');
}
```

Med dette object i hånden, kan vi foretage http-request'et:

```
xmlhttp.open('GET', "http://oiorest.dk/Danmark/regioner/1081", false);
xmlhttp.send(null);
```

>

open-metoden tager 3 argumenter:

- Http-metoden (GET, POST, HEAD, ...).
- URL'en (bemærk, at URL'en kan også være relativ til siden)
- En Boolean som dikterer, om forbindelsen skal være asynkron. Med synkron (false) connection vil script'et vente, til requestet færdiggøres. Om opførslen skal være synkron eller ej, skal naturligvis indtænkes i det overordnede design. Asynkron kommunikation og alle de muligheder, AJAX giver i den forbindelse, åbner op for meget levende og dynamiske sider.

send-metoden tager et enkelt argument: Indholdet (content), som skal sendes. I dette tilfælde anvender vi GET-metoden, så content er null, men ved eksempelvis POST eller PUT ville content have en værdi forskellig fra null.

Når vi kigger på kodeeksemplet ovenfor ser vi en vis ubalance mellem koden, der løser det egentlige problem og koden, der afgør, hvilken browser det kører i; der bliver brugt uforholdsmæssigt mange kodelinier på sidstnævnte. Dette understreger ret godt behovet for standardiserede frameworks som eksempelvis GWT og Script#, hvor denne stillingtagen er gemt af vejen og fokus er tilbage på programmeringen, som løser det reelle forretningsproblem.

13. Frembringelse af en REST-baseret webservice

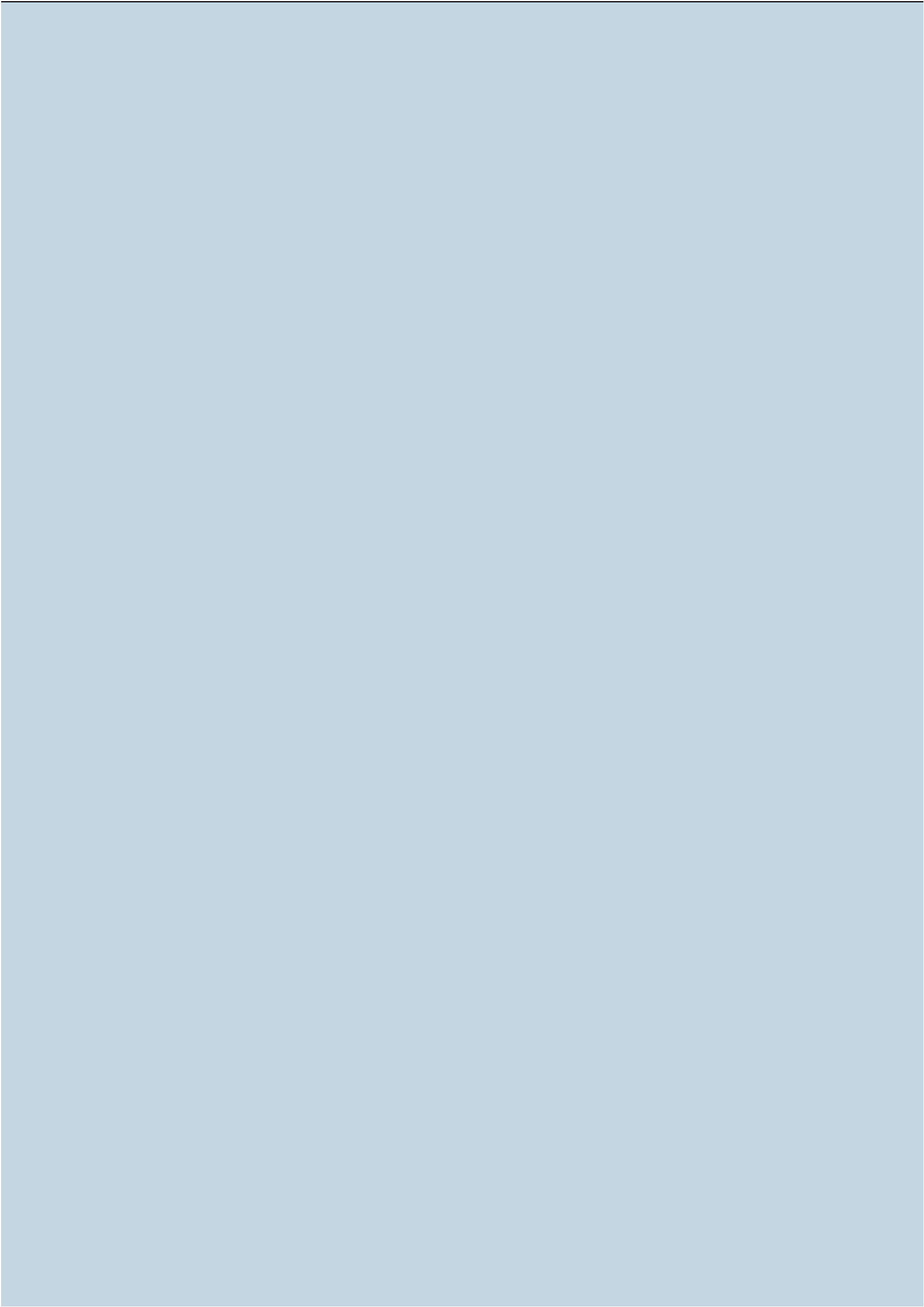
>

Til trods for, at REST-arkitekturen er så bredt anvendelig, er der en række overvejelser, man altid bør gøre sig, og en række punkter, man altid bør tage stilling til under frembringelse af en REST-baseret webservice.

Disse overvejelser er her illustreret gennem følgende 3 (eller 8, om man vil) punkter:

1. Få styr på datasættet: Find ud af, hvilke data, der skal eksponeres af webservice'n. Bemærk, at eksempelvis en transaktionsservice sagtens kan modelleres som data.
2. Opdel dataene i ressourcer: Dette vil ofte være sidestillet med de entiteter, som skal tilgås og kan således også være en transaktionsservice eller lign.
3. For hver ressource:
 - a. Tildel en URI til ressourcen
 - b. Find ud af, hvad der skal tilgås med read og/eller write access.
 - c. Design datarepræsentationen på klientsiden.
 - d. Bind ressourcen sammen med de andre relevante ressourcer (ofte gennem hyperlinks)
 - e. Tag stilling til, hvad der skal/kan udføres på ressourcen.
 - f. Tag stilling til fejl-scenarier

Ovenstående tager kun stilling til den rå datamodellering og de funktionelle aspekter. Dette kan evt. inkludere autentifikation og authorization (jvf. afsnittet om sikkerhed), men begreber som fortrolighed og transportmekanismer er på ingen måde adresseret af ovenstående.



<

Center for Serviceorienteret Infrastruktur

Center for Serviceorienteret Infrastruktur (CSI) blev oprettet 1. december 2006 for en periode på tre år. Centret har til opgave at lede og facilitere opgaven med at producere en åben, national infrastruktur.

Centret har samlet en række af IT- og Telestyrelsens medarbejdere, der hidtil har arbejdet med forskellige aspekter af samme felt, herunder arbejdet med serviceorienteret arkitektur (SOA), brugerstyring og infrastruktur til e-handel.
