



```
WebRequest request =
    WebRequest.Create("http://oiorest.dk/danmark/adresser?postnr=?");
request.Method = "GET";
request.CachePolicy = new RequestCachePolicy(RequestCachePolicy.NoCache);
HttpWebResponse response = (HttpWebResponse)request.GetResponse();

XPathDocument document = new XPathDocument(response.GetResponseStream());
XPathNavigator navigator = document.CreateNavigator();
XmlNamespaceManager manager = new XmlNamespaceManager(navigator);
manager.AddNamespace("n", "http://schemas.danmark.dk/geo");

XPathNodeIterator iterator = navigator.Select("//n:vejnavn");
while (iterator.MoveNext())
{
    class DanmarkXPathClient {
        public final String NS = "http://schemas.danmark.dk/geo";
        public static void main(String[] args) throws Exception {
            InputStream is = new URL("http://oiorest.dk/g...").openStream();
            DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
            domFactory.setNamespaceAware(true);
            DocumentBuilder builder = domFactory.newDocumentBuilder();
            Document document = builder.parse(is);
            XPathNavigator navigator = document.createNavigator();
            XmlNamespaceManager manager = new XmlNamespaceManager(navigator);
            manager.addNamespace("n", "http://schemas.danmark.dk/geo");
            XPathNodeIterator iterator = navigator.select("//n:vejnavn");
            while (iterator.MoveNext())
            {
                String vejnavn = iterator.Current.Value;
                String url = "http://oiorest.dk/danmark/adresser?postnr=2400&vejnavn=" + vejnavn;
                WebRequest request = WebRequest.Create(url);
                request.Method = "GET";
                request.CachePolicy = new RequestCachePolicy(RequestCachePolicy.NoCache);
                HttpWebResponse response = (HttpWebResponse)request.GetResponse();
                XPathDocument document = new XPathDocument(response.GetResponseStream());
                XPathNavigator navigator = document.CreateNavigator();
                XmlNamespaceManager manager = new XmlNamespaceManager(navigator);
                manager.AddNamespace("n", "http://schemas.danmark.dk/geo");
                XPathNodeIterator iterator = navigator.Select("//n:adresse");
                while (iterator.MoveNext())
                {
                    String husnummer = iterator.Current.Value;
                    String postnummer = iterator.Current.Value;
                    String postdistrikt = iterator.Current.Value;
                }
            }
        }
    }
}

require 'net/http'
require 'rexml/document'

page = Net::HTTP.get('oiorest.dk', '/danmark/adresser?postnr=2400&vejnavn=ki...')
doc = REXML::Document.new(page)

doc.elements.each('adresse/adresse') do |adresse|
    vejnavn = adresse.elements['vej'].elements['navn'].text
    husnummer = adresse.elements['husnr'].text
    postnummer = adresse.elements['postdistrikt'].elements['nr'].text
    postdistrikt = adresse.elements['postdistrikt'].elements['navn'].text
end
```

OIOREST i praksis

Guidelines baseret på eksempler



IT- og Telestyrelsen

Ministeriet for Videnskab
Teknologi og Udvikling



OIOREST i praksis

Udgivet af:
IT- & Telestyrelsen

IT- & Telestyrelsen
Holsteinsgade 63
2100 København Ø

Telefon: 3545 0000
Fax: 3545 0010

Publikationen kan også hentes
på IT- & Telestyrelsens
Hjemmeside: <http://www.itst.dk>
ISBN (internet): 87-92311-72-5

>

OIOREST i praksis.

Guidelines baseret på eksempler.

Version 1.00

1. Indhold

>

1.	Indhold	4
2.	Indledning	5
3.	Formål	6
4.	Udstilling af data	7
4.1	Designovervejelser	7
4.2	Udvælgelse af data	7
4.3	Repræsentation	9
4.4	Operationer	12
4.4.1	Pålidelighed med GET	12
4.5	Fejlscenarier	13
4.6	Caching	14
4.7	Queries	14
4.8	Servicespecifikationen	15
4.9	Tjekliste for udstilling af data	15
5.	Opdatering af data.	17
5.1	Pålidelighed	17
5.1.1	Pålidelighed med PUT	18
5.1.2	Pålidelighed med POST	18
5.1.3	Pålidelighed med DELETE	19
6.	Sikkerhed	21
6.1	Sikkerhedsbegreberne	23
6.1.1	Autentifikation	23
6.1.2	Autorisation	23
6.1.3	Fortrolighed	24
6.1.4	Uafviselighed	24
7.	Kø-servicens konkrete implementation	26
7.1	Serverside implementation: Serviceudbyderen	27
7.2	Klientside implementation: Serviceaftageren	31
8.	Referencer	36

2. Indledning

>

Der opleves en stadig større udbredelse i antallet af REST-baserede webservices, og det er oplagt, at meget offentligt data i Danmark, og for så vidt også på europæisk og globalt plan, med fordel kan udstilles vha. denne metode.

De fleste har den opfattelse, at REST-baserede webservices er rettet mod mennesker og web-baserede GUIer, men REST er lige så velegnet til system-system-kommunikation.

For at lette arbejdet med at udstille offentlige data vha. REST-baserede webservices, har IT- og Telestyrelsen udarbejdet en række dokumenter, som dels diskuterer mulighederne og perspektiverne, og dels giver en række retningslinier til støtte for frembringelsen og brugen af REST-baserede webservices.

3. Formål

>

Formålet med nærværende dokument er at yde støtte til frembringelsen af en REST-baseret webservice. Dokumentet vil centrere sig omkring to kørende og realistiske eksempler og benytte disse til diskussion af centrale problemstillinger.

Første eksempel beskriver aspekterne ved udstilling af data, og til dette formål vil vi anvende Danmarks-servicen, som udstiller information om forskellige dele af Danmark så som regioner, kommuner, skole- og valgdistrikter, adresser, veje osv.

Andet eksempel håndterer opdatering af data og aspekterne ved sikkerhedsbelagte data gennem implementering af en kø-service, hvor brugerne (dvs. klienterne eller service-aftagerne) kan udveksle dokumenter, selvom de ikke nødvendigvis er online på samme tid.

Hele dokumentet knytter sig tæt til eksemplerne og er således meget udviklerorienteret. De mere forretningsorienterede facetter ved udstilling af data, især sikkerhedsbelagt data, diskuteres i dokumentet "OIOREST Sikkerhedmodeller" ([Sikkerhedmodeller]).

Undervejs i dokumentet vil der være henvisninger til bogen "RESTful Web Services" ([Referencebogen]). OIOREST er baseret på de tekniske retningslinier, som bogen udstikker og målretter brugen af REST til danske forhold.

4. Udstilling af data

>

Når vi vælger at udstille offentlige data – eller at offentliggøre data, om man vil – melder der sig en række spørgsmål af både teknisk, juridisk og økonomisk karakter. Vi vil i første omgang afgrænse dette ved at beslutte, at vi vil udstille gratis og ikke-sikkerhedsbelagt data og ydermere på en måde, så data ikke kan opdateres vha. REST-interfacet.

4.1 Designovervejelser

I det følgende vil vi lade et konkret eksempel bære opgaven og dermed også være med til at afgøre, hvilke designovervejelser vi gør os undervejs.

Udgangspunktet er, at vi har en database, der indeholder samtlige adresser i Danmark og herunder, knyttet til hver adresse, information om region, kommune, valg- og skoledistrikt, vej og ikke mindst adressens koordinater.

Opgaven består i at udstille data, så man kan udsøge regioner, kommuner, skoledistrikter, sogne, adresser osv. samt deres indbyrdes relationer.

Det komplette eksempel kan ses på: <http://oiorest.dk/danmark>

4.2 Udvalgelse af data

Erfaringerne viser, at når man udstiller gratis data af denne karakter, opstår der flere og andre anvendelsesmuligheder, end man først havde forudset, og jo mere data, der udstilles, jo mere kreativitet ser man på aftagersiden, og jo flere "uforudsete" klienter opstår der ("uforudset" skal i denne sammenhæng ses i ordets allermest positive betydning).

For at være lidt fremsynede vil vi derfor ikke kun udstille data til lige netop de usecases, vi selv kan se på nuværende tidspunkt, men derimod "åbne" datasættet og udstille data som eventuelt kunne blive interessant for andre aftagere eller for andre usecases.

Når man designer REST-servicen, kan det være en god hjælp at se det som udstilling af *punkter* i datasættet (eller objektgrafene som afspejles af datasættet). I vores tilfælde dækker begrebet *punkter* således over eksempelvis de enkelte adresser, de enkelte regioner, kommuner osv.

Typisk vil data eller dele af data udspænde en træstruktur, og man vil være tilbøjelig til at fokusere på bladknuderne, fordi de ofte udgør de mest oplagte entiteter i datasættet. Hvis man i stedet udstiller både bladknuderne og forældrene hertil, giver det mulighed for at tilgå både entiteter og collections heraf - og derudover ligger det lige for at introducere søgefaciliteter.

>

Forældrenes forældre eller relationer til andre punkter i datasættet åbner nye muligheder for at navigere i data, og dette skal sammenholdes med, at det formentlig ikke koster væsentligt mere at udstille praktisk talt alt, når man først har introduceret den underliggende motor.

I adresse-eksemplet vil det således sige, at vi ikke kun vil udstille de enkelte kommuner, men også en ressource, der indeholder samtlige kommuner – altså en collection af kommuner. Og på samme måde vil vi udstille en collection af regioner osv.

Med andre ord så er samlingen af kommuner altså blevet en forældreknude til hver enkelte kommune, men på samme tid er hver enkelte kommune også en forældreknude til eksempelvis sogne i kommunen, og knuden ”sogne i kommunen” bliver til forældreknuden til hvert enkelt sogn. Dette giver en enestående mulighed for at navigere i data.

I vores eksempel, hvor vi råder vi over begreber som regioner, kommuner, sogne, byer, adresser osv., besluttet det at udstille data som følger (her lidt forenklet):

URI	Hvad skal returneres?	Hvad skal være indeholdt i svaret?
/regioner	En liste af regioner	Links til de enkelte regioner
/regioner/<regionsnummer>	Den enkelte region	Regionens navn og nummer Links til regionens kommuner og adresser
/kommuner	En liste af kommuner	Links til de enkelte kommuner
/kommuner/<kommunennummer>	Den enkelte kommune	Kommunens navn og nummer Links til kommunens veje og adresser
/postdistrikter	En liste af postdistrikter	Links til de enkelte

		postdistrikter
/postdistrikter/<postdistriktnummer>	Det enkelte postdistrikt	Postdistriktets navn, Links til postdistriktets veje og adresser
/kommuner/<kommunenummer>/veje	En liste af kommunens veje	Links til de enkelte veje i kommunen
/kommuner/<kommunenummer>/vej/<vejnavn>	Den enkelte vej	Links til vejens kommune
/kommuner/<kommunenummer>/adresser	En liste af kommunens adresser	Links til kommunens adresser
/kommuner/<kommunenummer>/adresse	Den enkelte adresse	Adressens koordinater Links til vejen, postdistriktet, kommunen
OSV...

Læg mærke til, at URLerne (egentlig URIerne) afspejler hierakiet i data, på en let læselig og entydig måde.

For en nærmere diskussion af processen omkring udvælgelsen af data, se [Referencebogen], kapitel 5, specielt afsnittet ”Figure Out the Data Set”.

4.3 Repræsentation

Nu hvor vi har besluttet os for, hvad vi vil udstille, på hvilke URLer og ikke mindst med ord har beskrevet, hvordan de enkelte data skal hænge sammen, er vi klar til at konkretisere det, som skal returneres.

Der er ikke nogen facitliste, der dikterer formen, men ofte vil et simpelt XML-skema give os lige præcis, hvad vi har brug for. XML har flere fordele, men vigtigst er det, at det er læsbart af både maskiner og mennesker, at det er godt til at udtrykke en hierarkisk sammenhæng, og at det er entydigt. Hertil kommer, at det er muligt, at der allerede findes et OIOXML-skema for den nødvendige entitet (det kunne f.eks. være, at man i det Digitale Danmark allerede havde standardiseret begrebet "postdistrikt"). Hvis det ikke er tilfældet,

>

at der allerede findes et passende skema, og man selv vil definere et, er det vigtigt at holde for øje, at det skal være simpelt og dermed umiddelbart forståeligt og indlysende, hvad skemaet indeholder.

XML er ikke den eneste mulighed, og i sagens natur er der mere oplagte kandidater til returnering af f.eks. billeder eller tekst-dokumenter - hvis ressourcen har en veldefineret MIME-type, er det formentlig oplagt at anvende denne.

Da vores Adresse-ressource bl.a. indeholder koordinaterne for en given adresse, er det oplagt, at vi skal kunne returnere forskellige standarder for koordinatsæt, så man udover at få punktet præsenteret med UTM-koordinater også kan få returneret punktet i Google Earth format eller som JSON. Til det formål anvender vi muligheden for at postfixe URL'en med en extension, så hvis vi eksempelvis angiver ".kml" som extension, får vi Google Earth repræsentationen. Denne fremgangsmåde er yderst velegnet til eksempelvis at returnere en given ressource på flere sprog - her kan ".en" eller ".dk" extensions eksempelvis angive, at man ønsker den engelske hhv. den danske repræsentation af ressourcen.

Det står således frit for at anvende andre formater end XML, men hvis man har brug for at udtrykke værdier og samhørighed (attributter og referencer), vil det være oplagt at anvende det, der allerede er konsensus om frem for at opfinde noget nyt og derved introducere fejlmuligheder i fortolkningen heraf - og her er XML et godt valg. Og som sagt så er det vigtigt at holde det simpelt og umiddelbart overskueligt.

En sidste regel vi skal have med er, at sammenhængen/hierarkiet i data afspejles i repræsentationen, eller med andre ord, at repræsentationerne for de enkelte ressourcer indeholder links til andre ressourcer. Helt konkret i vores eksempel kommer det f.eks. til udtryk i, at repræsentationen for "kommune" indeholder links til ressourcerne for "veje" og "adresser".

Vi har derfor valgt at modellere ressourcerne "kommuner" og "kommune" således:

```
<schema targetNamespace="http://itst.dk/schemas/danmarkservice"
elementFormDefault="qualified">
  <include schemaLocation="kommune.xsd"/>
  <element name="kommuner" type="dk:kommunertype"/>
  <complexType name="kommunertype">
    <sequence>
      <element name="kommune" type="dk:kommunetype" minOccurs="0"
maxOccurs="unbounded"/>
    </sequence>
    <attribute name="timestamp" type="time"/>
    <attribute name="ref" type="anyURI"/>
  </complexType>
</schema>
```

```
<schema targetNamespace="http://itst.dk/schemas/danmarkservice"
elementFormDefault="qualified">
```



```
<include schemaLocation="veje.xsd"/>
<include schemaLocation="adresser.xsd"/>
<import namespace="http://rep.oio.dk/cpr.dk/xml/schemas/core/2006/05/23/"
schemaLocation="http://rep.oio.dk/cpr.dk/xml/schemas/core/2006/05/23/CPR_MunicipalityName.xsd"/>
<element name="kommune" type="dk:kommunetype"/>
<simpleType name="KommuneNummerType">
  <restriction base="string">
    <pattern value="[0-9]{3}"/>
  </restriction>
</simpleType>
<complexType name="kommunetype">
  <sequence>
    <element name="nr" type="dk:KommuneNummerType" minOccurs="0"/>
    <element name="navn" type="cpr:MunicipalityNameType" minOccurs="0"/>
    <element name="veje" type="dk:vejetype" minOccurs="0"/>
    <element name="adresser" type="dk:adrestertype" minOccurs="0"/>
  </sequence>
  <attribute name="ref" type="anyURI"/>
</complexType>
</schema>
```

Hvis vi kigger på det konkrete XML, som en ressource returnerer i henhold til ovenstående skemaer, får vi eksempelvis:

```
<kommuner timestamp="20:00:17">
  <kommune ref="http://oioest.dk/danmark/kommuner/101">
    <nr>101</nr>
    <navn>København</navn>
  </kommune>
  <kommune ref="http://oioest.dk/danmark/kommuner/147">
    <nr>147</nr>
    <navn>Frederiksberg</navn>
  </kommune>
  <kommune ref="http://oioest.dk/danmark/kommuner/151">
    <nr>151</nr>
    <navn>Ballerup</navn>
  </kommune>
```

på URLen

<http://oioest.dk/danmark/kommuner>

og:

```
<kommune ref="http://oioest.dk/danmark/kommuner/101">
  <nr>101</nr>
  <navn>København</navn>
  <veje ref="http://oioest.dk/danmark/kommuner/101/veje"/>
  <adresser ref="http://oioest.dk/danmark/kommuner/101/adresser"/>
</kommune>
```

på URLen <http://oioest.dk/danmark/kommuner/101>

Læg mærke til, at ressorens repræsentation i begge tilfælde indeholder de direkte links til det refererede data.

For mere information vedr. design af repræsentationer, se kapitel 5 i referencebogen, afsnittene ”Design Your Representations” og ”Link the Resources to Each Other” samt kapitel 8, afsnittet om ”Why Connectedness Matters”.

4.4 Operationer

Indtil videre har vi beskæftiget os med server-siden, eller implementeringen af selve REST-serviceen. Når vi taler operationer, er det lige så relevant at kigge på klient- eller aftager-siden.

Operationerne er indgangene til oprettelse, læsning, opdatering og nedlæggelse af data (i daglig tale ofte benævnt som CRUD-operationerne for Create, Read, Update og Delete). De tilsvarende http-metoder er således POST for oprettelse, GET for læsning, PUT for opdatering og DELETE for nedlæggelse.

Måske skal servicen tillade både læsning og skrivning, og måske dækker skrivning over både opdatering af eksisterende data og oprettelse af ny data. Man kunne eksempelvis forestille sig, at en administrativ myndighed havde mulighed for at redigere eller oprette adresser. Her gælder præcis de samme overvejelser mht. datarepræsentation, og i de tilfælde hvor en given datatype både kan læses og skrives, er der igen grund til at opfinde forskellige repræsentationer for de to. Tværtimod kan der være en fordel i, at det data, der PUTes er præcis det samme som det, der sidenhen kan GETes igen.

I vores eksempel holder vi os til read only, altså kun GET-metoder. I afsnittet om opdatering vil vi kigge nærmere på de andre HTTP-metoder.

Man kan se eksempler på anvendelse af GET-metoden til læsning af data og navigering i datasættet her:

<http://oiorest.dk/danmark/documentation/samplecode.aspx>

4.4.1 Pålidelighed med GET

Pålidelighed betyder ”garanti for, at requestet er gået igennem”. Denne garanti sikres i bund og grund ved, at den REST-baserede webservice er idempotent, og dermed kan man prøve igen og igen, indtil requestet returnerer succes. GET er den letteste og mest tilgængelige af service-metoderne. Da serveren ikke ændrer state i forbindelse med servicering af GET-requestet, kan vi på klientsiden pakke det ind i en algoritme som angivet nedenfor; herved sikres, at vi enten får et succesfuldt svar, eller at der bliver taget passende aktion, hvis dette ikke er muligt.

```
while (true) {
    Result res = client.get();
    if (res.isSuccess()) {
        stop;
    } else {
        sleep (lidt tid);
        count++;
        if (count > max) {
            // serveren svarer tilsyneladende ikke som forventet
        }
    }
}
```

```
}  
}
```

Bemærk i øvrigt, at et GET-request i langt de fleste tilfælde kan testes med en almindelig browser – det er kun, hvis der i requestet skal sættes specifikke http-headers, eller hvis responset indeholder content, som ikke kan vises eller vises forkert i browseren, at GET ikke med lethed kan udføres fra en almindelig browser.

4.5 Fejlscenarier

Vi mangler stadig at tage stilling til fejlscenarier. Der skal tages stilling til alle tænkelige fejlscenarier: Illegale dataværdier, ugyldig XML (hvis der anvendes XML), "resource not found" osv. Det er vigtigt, at den returnerede fejlstruktur eller kode er informativ - og vel at mærke uanset klient-typen.

Der kan i visse tilfælde være grund til at returnere forskellige typer af fejlbeskrivelser. Eksempelvis kan vi som sagt ikke forudse brugen af vores webservice på længere sigt (læs: hvilke use cases og klienttyper der vil opstå over tid), og derfor kan det måske være ønskværdigt, at "fejl-retur-koden" indeholder information til både mennesker og maskiner - eller med andre ord både en statuskode og en sigende tekst. Man kunne forestille sig, at der var forskellige behov til udvikling/test og produktion – eksempelvis muligheden for at returnere et stacktrace i forbindelse med en fejlsituation. Mere konkret: I denne forbindelse bør man i øvrigt overveje, hvilke detaljer, der ønskes udstillet i et produktionsmiljø.

Alt dette bør dog overvejes nøje; er der behov for yderligere information, end hvad man læser direkte ud af statuskoden? Eksempelvis vil nedenstående XML-struktur ikke tilføje nogen yderlige information, end en statuskode 404 ville gøre i sig selv:

```
<fault>  
  <code>404</code>  
  <parameters>  
    <parameter key="region">23</parameter>  
  </parameters>  
  <description>Ukendt region</description>  
</fault>
```

Når klienten får statuskode 404 tilbage på requestet "giv mig region 23", betyder det lige netop, at denne region ikke blev fundet, og der er således ingen yderligere information at hente i XML-strukturen.

>

Det er også vigtigt at fremhæve, at man bør holde sig strengt til HTTP-statuskodernes betydning og ikke opfinde nye eller lade sig friste til at returnere statuskode 200 sammen med information om, at requestet blev modtaget og behandlet, men den angivne ressource eksisterer ikke.

Hvis man som ovenfor vælger at returnere yderligere information, er det således ikke for at erstatte de eksisterende HTTP-statuskoder - det skal udelukkende ses som en mulighed til at knytte ekstra information på koderne.

HTTP-koderne skal anvendes, og hver enkelt kode skal anvendes til det, den nu en gang er tiltænkt, hvilket igen betyder, at man i mange tilfælde vil kunne klare sig uden at tilføje ekstra information.

[Referencebogen], kapitel 5, afsnittet om "The HTTP Response" diskuterer i øvrigt brugen af HTTP-statuskoderne.

4.6 Caching

Offentligt data vil som hovedregel være konstant over længere tidsrum. Naturligvis er der undtagelser herfra, men eksempelvis store dele af vores adresse-data må formodes at være konstante over ret lange perioder. Omvendt kan man ikke forudsige, hvornår der sker eventuelle ændringer, eksempelvis at en given vej flyttes til et andet valgdistrikt, så en eventuel caching vil skulle afspejle sådanne afvejninger. Man kunne fristes til at hævde, at adresseforespørgsler er relativt sjældne, så caching er ikke nødvendigt, men da vi ikke kan forudsige, hvem der vil anvende vores webservice og til hvad, og da det underliggende datagrundlag er meget stort (der er 2.2 millioner adresser i Danmark), vil det være fornuftigt at tillade caching i eksempelvis 24 timer. Hermed har vi taget stilling til, at data med fordel kan caches, men hvis der skulle ske en ændring, vil den trods alt slå igennem efter senest 1 døgn.

Til det formål anvender vi HTTP headeren `Cache-control: max-age=86400`, som angiver, at klienten *må* cache data i op til 24 timer, men ikke at den nødvendigvis *skal* gøre det.

[Referencebogen], kapitel 8 beskriver caching-principperne i lidt flere detaljer.

4.7 Queries

Vores udstilling af data er ret grovkornet; når man forespørger på en collection, får man den komplette liste tilbage. Ofte vil der være brug for at kunne raffinere forespørgslen lidt mere. Til det formål beslutter vi, at man skal kunne give parametre med og eksempelvis søge efter en specifik kommune:

>

`http://oiorest.dk/danmark/kommuner?q=<del af kommunenavn>`

Det er her vigtigt at holde for øje, at man ofte vil mappe søge-parametrene direkte til søgestrengene i den underliggende sql, og dermed har man åbnet for eventuel sql-injection. Man bør i disse tilfælde altid indskyde et valideringslag, som kan bortfiltrere uønskede søgekriterier.

4.8 Servicespecifikationen

Vi er nu klar til at lave den endelige service-specifikation.

Da servicen er så "rig" og omfattende, og da den først og fremmest skal kunne forstås og fortolkes af andre udviklere (dvs. mennesker i modsætning til maskiner), er det vigtigt, at vi tilpasser servicespecifikationen hertil. En tekstuel beskrivelse på skemaform er meget velegnet - men sørg for at være specifik, så misforståelser undgås.

I vores tilfælde ser servicespecifikationen således ud:

(den komplette specifikation kan findes på

<http://oiorest.dk/danmark/Documentation/apidoc.aspx>)

Resource	URI	Method	Repræsentation	Caching	Status
Regioner	/regioner	GET	regioner.xsd	Max-age=86400	200,405,500
Region	/regioner/<regionsnr>	GET	region.xsd	Max-age=86400	200,404,405,500
Kommuner i region	/regioner/<regionsnr>/kommuner	GET	kommuner.xsd	Max-age=86400	200,405,500
Adresser i region	/regioner/<regionsnr>/adresser?<søgekriterie>	GET	adresser.xsd	Max-age=86400	200,405,500
...

4.9 Tjekliste for udstilling af data

1. Find ud af, hvilke data der skal eksponeres af webservicen.
2. Opdel datasættet i ressourcer. En ressource vil typisk være en konkret entitet eller en collection af entiteter.
3. For hver ressource:
 - Tildel en URI
 - Bind ressourcen sammen med andre relevante ressourcer.
 - Fastlæg datarepræsentationen

>

- Fastlæg eventuelle fejlscenarier
 - Fastlæg caching-strategier
4. Lav servicespecifikationen

Som støtte til ovennævnte punkter og ikke mindst for at sikre at webservicen bliver fornuftigt anvendt, kan det være relevant at gøre sig følgende overvejelser:

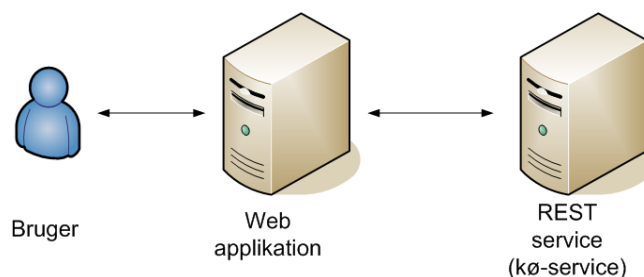
- Udstil alt. Hvis der med minimal indsats kan udstilles mere data, så gør det. Nye anvendelsesmuligheder vil med sikkerhed opstå.
- Anvend simple og entydige repræsentationer. Hvis ressourcen er en kendt type, så anvend den gængse repræsentation (f.eks. OIOXML-skemaer eller kendte MIME types).
- Anvend samme repræsentation for både læse- og skrive-data.
- Anvend HTTP-statuskoderne til det, de er tiltænkt.
- Antag, at servicen på et eller andet tidspunkt bliver anvendt væsentligt anderledes end først antaget. Dette har eksempelvis betydning for caching strategier.
- Antag, at servicen bliver udsat for forsøg på sql-injection og andre angreb.

5. Opdatering af data.

>

Da REST-servicen udstiller et datasæt, dækker begrebet opdatering over flere funktioner. Tilsammen kan vi betegne dette som *ændring af datasættet*, og dermed er opdatering både tilføjelse af nye ressourcer, sletning af eksisterende og ændringer til eksisterende ressourcer.

Til at illustrere dette vil vi fokusere på et andet og mere oplagt eksempel, en kø-service: En web applikation tilgår en REST-baseret kø-service, som tillader, at brugere kan udveksle dokumenter, selvom de ikke er online på samme tid.



Den komplette beskrivelse kan findes her:

<http://oiorest.dk/queueservice>

En af de mest centrale problematikker i forbindelse med opdatering af data knytter sig til begrebet pålidelighed. Hvorledes opnår vi sikkerhed for, at opdateringen er gået igennem? Det er selvfølgelig ret ligetil, hvis der returneres ”succes” som returkode, men hvis requestet timer ud, har vi ikke umiddelbart nogen ide om, hvor i kæden, det gik galt, og om servicen reagerede på requestet. Lad os i det følgende se på, hvorledes dette kan adresseres.

5.1 Pålidelighed

Under Danmarksservice-eksemplet kiggede vi på pålidelighed med GET-metoden. Nu vil vi udvide begrebet og medtage PUT, POST og DELETE metoderne. Som tidligere nævnt sikres pålideligheden i bund og grund ved, at den REST-baserede webservice er idempotent¹, og dermed kan man prøve igen og igen, indtil requestet returnerer succes. Der er dog imidlertid en enkelt krølle på denne regel, for POST er grundlæggende ikke idempotent af natur. Lad os derfor kigge på, hvordan vi sikrer pålideligheden for hhv. PUT, POST og DELETE.

¹ Idempotent betyder i denne forbindelse, at gentagne, ens requests til samme ressource, returnerer samme svar hver gang.

5.1.1 Pålidelighed med PUT

PUT anvendes til opdatering af en ressource, hvis URL er kendt på forhånd. Begrebet opdatering kan evt. dække over, at ressourcen oprettes for første gang, men stadig på en kendt URL. I tilfælde af, at der er tale om en ny ressource, vil URLen for øvrigt ofte være fremkommet som et svar på et POST-request; mere herom senere.

Hvis vi antager, at serveren enten opretter ressourcen eller opdaterer en allerede eksisterende ressource, vil følgende algoritme afspejle en mulig arbejdsgang fra klientens side:

```
while (true) {
    Result res = client.put(newResource);
    if (res.isSuccess() OR res.resourceExists()) {
        stop;
    } else {
        sleep (lidt tid);
        count++;
        if (count > max) {
            // er serveren død?
        }
    }
}
```

5.1.2 Pålidelighed med POST

POST er af natur ikke idempotent og kan dermed ikke anvendes blindt til oprettelse af en ressource, men bør ses i sammenhæng med PUT. POST anvendes, når man skal oprette en ny ressource, som man ikke nødvendigvis kender placeringen af, eller hvis serveren skal foretage sig noget indledende, inden ressourcen kan oprettes. Eller med andre ord hvis man ikke umiddelbart kan PUTte en ressource, fordi dens URL ikke er givet, eller fordi serveren ikke er klar til at modtage den.

Syntaksen er således, at man POSTer for at få angivet en URL til den nye ressource, som man herefter PUTer til den nyoprettede URL/lokation. Til formålet bør Location headeren i øvrigt anvendes. En forsimplet POST-mekanisme kan derfor se således ud:

```
while (true) {
    Result res = client.post(evt. med angivelse af ny ressource);
    if (res.created()) {
        newLocation = res.getLocationHeader();
        put(resource); // efter algoritmen ovenfor
        stop;
    } else {
        if (!res.timeout()){
            // serveren har svaret, men ikke med res=created
            // reager på fejlkoden
            stop;
        } else {
            sleep (lidt tid);
            count++;
        }
    }
}
```

>

```
        if (count > max) {  
            // er serveren nede?  
        }  
    }  
}
```

Bemærk, at succes i forbindelse med oprettelse af en ressource angives med statuskode 201, created.

Det er i ovenstående tilfælde vigtigt at fremhæve, at hvis requestet går godt, men responset ikke når frem, vil klienten opleve, at POSTet fejlede, mens serveren oplevede, at det gik godt, og en ny ressource blev skabt. Afhængig af strategien, risikerer man derfor, at der oprettes ”tomme” ressourcer på serveren, og der bør derfor også være en strategi for oprydningen af disse.

5.1.3 Pålidelighed med DELETE

DELETE kan stort set sidestilles med GET, idet man blot kan prøve igen og igen – og hvis det skulle ske, at man uforvarende kommer til at slette en ressource to gange, bør det ikke have andre sideeffekter, end at statuskoden ændres fra succes (200) til eksempelvis not found (404):

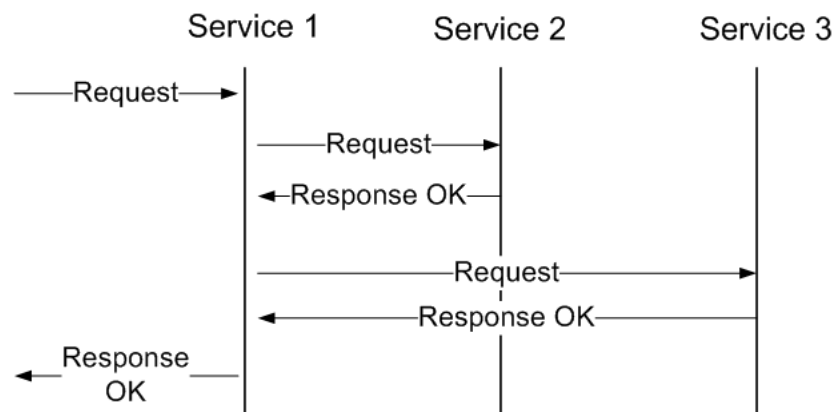
```
while (true) {  
    Result res = client.delete();  
    if (res.isSuccess() or res.notFound()) {  
        break;  
    } else {  
        sleep (lidt tid);  
        count++;  
        if (count > max) {  
            // do something...  
        }  
    }  
}
```

Alle de ovenfor skitserede eksempler er naturligvis en smule (læs: meget) forenkede. Hvis servicen eksempelvis returnerer ”not authorized”, ”moved permanently” eller lign., er der naturligvis ingen grund til at prøve igen og igen. Desuden kan der f.eks. i forbindelse med PUT være situationer, hvor det giver mening at returnere eksempelvis ”not modified” i stedet for ”succes”.

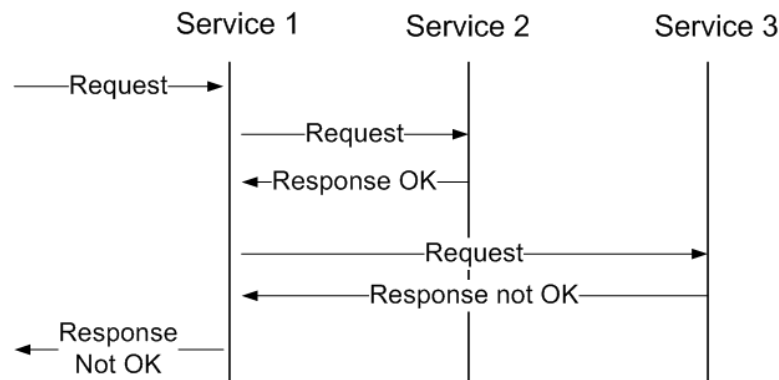
Hvis scenarierne er en lille smule sammensatte af natur, vinder man lidt på, at requestene er synkrone, for derved bliver det eller de bagvedliggende kald til eksempelvis persistenslag eller lign. nastede i det yderste request, hvilket gør det meget veldefineret, hvor i logikken man evaluerer resultaterne fra det bagvedliggende og sender en passende returkode ud mod klienten. Naturligvis opstår der situationer, hvor flere statuskoder skal sammenholdes til en enkelt resulterende statuskode, men dette vil også ligge på et meget naturligt og

>

veldefineret sted i REST-servicen. Dette er meget tydeligt ud fra nedenstående tegninger, hvor den yderste service, 1, invokerer de bagvedliggende services, 2 og 3, inden der returneres en statuskode til aftageren på baggrund af resultaterne fra de enkelte services. I første omgang, går de enkelte requests godt, og der returneres en samlet status kode "ok":



Og naturligvis efter tilsvarende fremgangsmåde, hvis det går galt undervejs:



Om det sidstnævnte eksempel overhovedet er fejlet som konsekvens af, at kaldet til service 3 fejlede, er naturligvis helt og holdent styret af den aktuelle case. Hvis det totale requests succes er styret af, at alle de bagvedliggende kald går godt, ender det meget let i et scenarie, hvor flere requests skal pakkes sammen i en samlet transaktionsbeskyttelse. Det vil dog føre for vidt at gennemgå dette i detaljer i nærværende dokument.

6. Sikkerhed

>

Det har traditionelt været en udbredt og fejlagtig opfattelse, at REST-baserede webservices generelt ikke tilbyder den sikkerhed, som eksempelvis Det Digitale Danmark kræver. Selvom denne misforståelse efterhånden bliver mere sjælden, er der stadig situationer og problemstillinger, hvor det ikke er umiddelbart indlysende, hvordan sikkerheden adresseres med en REST-baseret arkitektur.

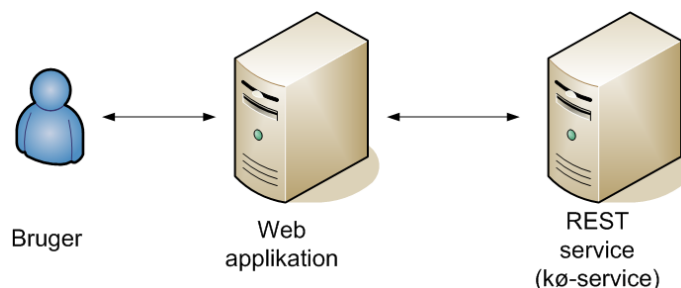
Inden vi kigger på de nærmere detaljer, vil vi lave en opsplitning i ”direkte” sikkerhed, hvor klientens udfører requests på vegne af sig selv, og ”på vegne af” hvor klienten udfører sine requests på vegne af en anden. Det er primært sidstnævnte eksempel, vi vil fokusere på i dette dokument, men for at få en dybere forståelse af begge scenarier henvises til [Sikkerhedsmodeller].

Begrebet sikkerhed er langt fra entydigt, og i daglig tale dækker det over flere principper som f.eks. fortrolighed, netværkssikkerhed, fysisk sikkerhed, applikationssikkerhed, autentifikation og meget andet. Det er klart, at en vis mængde af sikkerhedsbegreberne ligger et stykke fra såvel software som software-arkitektur, og derfor vil vi her undlade at berøre de emner, som enten ikke har berøring med software som sådan eller er uafhængige af den valgte arkitektur.

I det efterfølgende vil vi således fokusere på begreberne (hvoraf nogle, som vi skal se, også ligger i periferien af eller udenfor, hvad arkitekturmodellen adresserer):

- Autentifikation
- Autorisation
- Fortrolighed
- Uafviselighed

Som støtte til at vise, om begreberne adresseres fyldestgørende, vil vi fortsat centrere os omkring kø-service:



>

I eksemplet baserer os på login vha. OCES både overfor web applikationen og overfor kø-servicen. For mere information, se <http://oiorest.dk/queueservice>.

Hvor mange har den opfattelse, at REST nærmest pr. definition betyder, at en web applikation går direkte på en eller flere services og præsenterer data direkte i browseren, er det vigtigt at understrege, at REST "blot" er en arkitektur til realisering af en web service, og at REST er mindst lige så relevant for system-system-kommunikation. Vores eksempel adresserer lige netop denne situation og fokuserer i højere grad på interfacet mellem udbyder og aftager end på snittet ud mod slutbrugers browser.

I eksemplet er der således behov for autentifikation to steder i kæden: Dels skal brugeren autentificeres overfor web applikationen, og dels skal web applikationen, som altså også er aftager, autentificeres overfor service udbyderen.

6.1 Sikkerhedsbegreberne

Lad os i det følgende kigge på, hvordan vi vil adressere de 4 sikkerhedsbegreber.

6.1.1 Autentifikation

Begreberne autorisation og autentifikation bruges tit i sammenhæng, og ofte taler man ikke om det ene uden også at inkludere det andet – og ofte er det ikke helt klart, hvad begreberne præcist dækker over. Lad os derfor starte med at slå fast, at autentifikation betyder brugervalidering og dækker over processen med at logge brugeren ind eller indsamle den information, der skal til for at identificere brugeren entydigt.

HTTP har 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, og det vil derfor formentlig være et krav, at kommunikationen foregår over SSL.

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 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.

OCES certifikater/digital signatur er de facto standarden for login i Det Digitale Danmark, og dermed vil OIOREST i henhold til de ovenfor nævnte autentifikationsmetoder være baseret på 2-vejs SSL med OCES certifikat.

6.1.2 Autorisation

Autorisation har med rettigheder at gøre, dvs. hvem har lov til hvad.

Det er oplagt at ræsonnere sig frem til, at da REST bygger på http og en url-orienteret tilgang til ressourcerne, vil man kunne anvende en almindelig, rollebaseret sikkerhedsmodel for webapplikationer, altså eksempelvis web.xml's <security-constraint> og web.config's <authorisation> for henholdsvis Java- og .Net-plattformene. Det er dog vigtigt at fremhæve, at denne model er statisk

>

af natur og principielt bygger på, at man på deploy-time kan fastsætte roller og grupper af roller, og tildele ressourcer rettigheder på baggrund heraf. I Det Digitale Danmark's mange forskellige systemer er bruger-begrebet meget rigt, og eksempelvis en situation hvor urlen

<http://domain.dk/restservice/minside> vil returnere forskelligt afhængigt af den pågældende bruger, vil dette princip ikke med lethed kunne anvendes.

Der rejser sig i samme forbindelse andre interessante problemstillinger. Lad os antage, at man har et system, som svarer på en url af følgende

<http://domæne.dk/person/følsommeoplysninger/oplysning1>.

Hvis denne url beskyttes med et almindelig rollebaseret princip, vil en uautoriseret bruger få en "not authorized" tilbage, når han tilgår ressourcen.

Dermed er det blevet afsløret, at ressourcen eksisterer, og alene denne information vil i visse tilfælde bryde med hensigten (og måske endda lovgivningen). Hvis der i stedet returneres en "eksisterer ikke", når ikke-autoriserede brugere tilgår ressourcen, vil man ikke kunne uddrage nyttig implicit information alene ud fra returkoden (eller i bedste fald vil man konkludere fejlagtig information, nemlig at ressourcen ikke findes).

En sund og velfungerende autorisationsmodel er derfor ofte "større" end den beskyttelse, som webserveren giver. Eller med andre ord, så ligger det i problem-domænet og ikke i protokollen eller platformen at løse autorisationsproblematikkerne. Styringen af autorisationen ligger i forretningslogikken og er dermed ikke direkte koblet til REST.

6.1.3 Fortrolighed

Begrebet fortrolighed dækker over mange forskellige teknikker og principper, og det vil føre for vidt at berøre det dybere her. Vi vil i eksemplet basere os på 2-vejs-SSL med OCES-/klient-certifikater.

For en mere dybdegående redegørelse herfor inkl. en diskussion af de juridiske aspekter heraf henvises til [Sikkerhedsmodeller].

6.1.4 Uafviselighed

Uafviselighed tjener til at give de involverede parter en gensidig garanti for modpartens handling og eksistens: Afsenderen ønsker sikkerhed for, at beskeden afleveres til den rette modtager, og modtageren ønsker sikkerhed for, at beskeden rent faktisk stammer fra den rette afsender. Begge parter ønsker sikkerhed for, at modparten ikke sidenhen kan påstå, at udvekslingen ikke har fundet sted.

Uafviselighed er tæt koblet til forretningen og juraen omkring denne, og der findes derfor (desværre) ikke noget værktøj eller nogen protokol, som stiller giver denne sikkerhed. Man kan ikke i noget udviklingsværktøj blot krydse af,

>

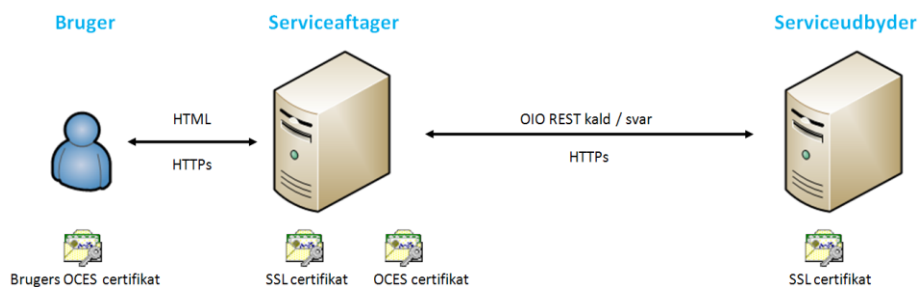
at der ønskes uafviselighed på kommunikationen, uden at udvikleren skal foretage sig yderligere, og man kan ikke tale om, om REST giver uafviselighed eller ej; dette bunder langt dybere i forretningsområdet.

En yderligere diskussion af dette kan findes her: [Sig-bev].

7. Kø-servicens konkrete implementation

>

Inden vi kigger nærmere på, hvordan kø-servicen er implementeret på hhv. aftager- og udbyder-side, skal vi lige have konteksten på plads:



Vores eksempelkode, som kan findes her:

<http://www.oiorest.dk/queueservice/documentation/samplecode.aspx> indeholder implementeringer af både rige klienter og web-eksemplet. Vi vil i det efterfølgende koncentrere os om web-eksemplet.

Da web applikationen skal agere på vegne af brugeren, skal brugerens credentials delegeres videre fra web applikationen til servicen. Som angivet på ovenstående illustration er der to OCES-certifikater i spil, således at brugeren logger ind i browseren med sit personlige certifikat, og web applikationen logger ind på REST-servicen f.eks. med et virksomheds-certifikat, men opererer på REST-interfacet på vegne af brugeren. Dvs. der kræves et trust relationship mellem serviceudbyder og serviceaftager. Hermed kan aftageren nemlig indsætte information om brugeren i "On-Behalf-Of" headeren i det efterfølgende request til udbyderen, og udbyderen kan stole på, at denne information er korrekt.

Lad os kigge på, hvordan det implementeres i praksis.

7.1 Serverside implementation: Serviceudbyderen

Køservice-eksemplet er implementeret i Microsoft .Net 3.5 Framework. Kun de centrale dele bliver gennemgået i det følgende; den komplette implementation kan downloades på:

<http://oiorest.dk/queueservice/documentation/samplecode.aspx>

Logikken i koden er bygget op omkring en række http-handlers, som hver tager sig af en ressource og den funktionalitet, der knytter sig hertil. Der findes således følgende handlers:

- Document
- Message
- MessageId
- Messages
- Owner
- Owners
- Queue
- Queues
- Queuetypes

Inden den specifikke handler rammes, bliver requestet dog først behandlet af en handler, som tager værdien af On-Behalf-Of headeren og hæfter SSN-id'et på requestet. Dette skal sikre, at den efterfølgende servicering af requestet på en let måde kan afgøre, hvilken bruger der repræsenteres af requestet. I AuthenticationModule ser det ud som følger:

```
public class AuthenticationModule : IHttpModule
{
    private void Application_BeginRequest(Object source, EventArgs e)
    {
        HttpApplication application = (HttpApplication)source;
        HttpContext context = application.Context;
        try
        {
            string onBehalfOf = context.Request.Headers["X-On-Behalf-Of"];
            string[] items = HttpUtility.UrlDecode(
                onBehalfOf).Split(new char[] { ',' });
            context.Items["SSN"] =
                Utility.SsnToUrlFriendly(items[0].Trim());
            context.Items["CN"] = items[1].Trim();
        }
    }
}
```

Inden vi kan videre-processere request'et, skal vi dog lige have undersøgt, om klienten – altså ikke slutbrugeren, men service-aftageren – er trust'ed:

```
HttpClientCertificate cer = context.Request.ClientCertificate;
X509Certificate2 x509 = new X509Certificate2(cer.Certificate);
OcesCertificate certificate = new OcesCertificate(x509);
If (!DbUtility.IsTrustedConsumer(certificate.GetSubjectSerialNumber()))
{
    throw new Exception();
}
```

>

Og endelig er vi nødt til at håndtere den situation, at serviceaftageren slet ikke agerer på vegne af en bruger, men selv er slutbrugeren – altså at On-Behalf-Of headeren ikke er sat, men at de reelle bruger-credentials skal tages direkte fra certifikatet, dermed ser metoden ud som følger:

```
    }

    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    if (!context.Request.IsSecureConnection) return;
    try
    {
        HttpClientCertificate cer = context.Request.ClientCertificate;
        X509Certificate2 x509 = new X509Certificate2(cer.Certificate);
        OcesCertificate certificate = new OcesCertificate(x509);
        string onBehalfOf = context.Request.Headers["X-On-Behalf-Of"];
        if (onBehalfOf != null)
        {
            if
            (!DbUtility.IsTrustedConsumer(certificate.GetSubjectSerialNumber()))
            {
                throw new Exception();
            }
            string[] items = HttpUtility.UrlDecode(onBehalfOf).Split(new
            char[] { ',' });
            context.Items["SSN"] =
            Utility.SsnToUrlFriendly(items[0].Trim());
            context.Items["CN"] = items[1].Trim();
        }
        else
        {
            context.Items["SSN"] = certificate.GetSubjectSerialNumber();
            context.Items["CN"] = certificate.GetName();
        }
    }
    catch (Exception ex)
    {
        context.Response.StatusCode = 401;
        context.Response.StatusDescription = "Unauthorized";
        application.CompleteRequest();
    }
}
```

Hermed er de rette credentials lagt tilrette i requestets context, og vi kan behandle requestet. Som eksempel vil vi fokusere på håndteringen af Queue. Allererst delegerer vi til den metode, som skal håndtere det konkrete kald, dvs. den konkrete http-metode:

```
public class Queue : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        switch (context.Request.HttpMethod)
        {
            case "GET":
                GetQueue(context);
                break;
            case "PUT":
                PutQueue(context);
                break;
            case "DELETE":
                DeleteQueue(context);
                break;
            default:
                context.Response.StatusCode = 405;
                break;
        }
    }
}
```

Det første, vi herefter gør – her vist i GET-metoden – er at undersøge, om den pågældende bruger rent faktisk har tilladelse til den pågældende operation på den pågældende ressource; og hvis dette ikke er tilfældet, skal der returneres en passende status kode:

```
private void GetQueue(HttpContext context)
{
    string user = context.Items["SSN"].ToString().Trim();
    string owner = (context.Request.Params["owner"] != null) ?
        context.Request.Params["owner"].Trim() : user;
    if (!user.Equals(owner))
    {
        context.Response.StatusCode = 403;
        context.Response.StatusDescription = "Unauthorized";
        return;
    }
}
```

Her efter er opgaven så at gøre, hvad GetQueue reelt skal gøre:

```
try
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["QueueDBConnectionString"].ToString();
    using (SqlConnection connection = new
        SqlConnection(connectionString))
    {
        SqlCommand command = new SqlCommand("SELECT queues.OwnerId,
        queues.QueueTypeId,
        QueueTypes.name, QueueTypes.mediatype " +
        "FROM queues INNER JOIN QueueTypes ON queues.QueueTypeId =
        QueueTypes.id " +
        "WHERE (queues.OwnerId = @owner) AND (queues.QueueTypeId =
        @queueType)",
        connection);
        command.Parameters.Add("@owner", SqlDbType.NChar, 64).Value =
        owner;
        string queueType = context.Request.Params["queue"];
        command.Parameters.Add("@queueType", SqlDbType.NChar, 10).Value
        = queueType;
        connection.Open();
        SqlDataReader rdr = command.ExecuteReader();
        XmlTextWriter w = null;
        context.Response.ContentType = "text/xml; charset=utf-8";
        w = new XmlTextWriter(context.Response.Output);
        w.Formatting = System.Xml.Formatting.Indented;
        bool found = false;
        while (rdr.Read())
        {
            found = true;
            w.WriteStartDocument();
            w.WriteStartElement(XmlNamespaces.queue, XmlNamespaces.NS);
            string queue = rdr[1].ToString().Trim();

```

>

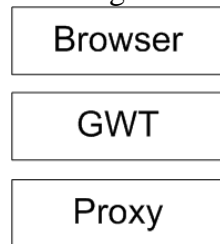
```
string url = Utility.MakeAbsoluteUrl(HttpContext.Current,
"/owners/"
    + owner + "/queues/" + queue);
w.WriteAttributeString("ref", url);
w.WriteElementString("id", XmlNamespaces.NS, queue);
w.WriteElementString("name", XmlNamespaces.NS,
rdr[2].ToString());
w.WriteElementString("mediatype", XmlNamespaces.NS,
rdr[3].ToString());
w.WriteStartElement("messages", XmlNamespaces.NS);
w.WriteAttributeString("ref", url + "/messages");
w.WriteEndElement();
w.WriteStartElement("newdocumentid", XmlNamespaces.NS);
w.WriteAttributeString("ref", url + "/newdocumentid");
w.WriteEndElement();
w.WriteEndElement();
w.WriteEndElement();
}
if (!found)
{
    HttpContext.Current.Response.StatusCode = 404;
    HttpContext.Current.Response.StatusDescription = "Not
found";
}

}
catch (Exception ex)
{
    Exception exp = ex;
    HttpContext.Current.Response.StatusCode = 500;
    HttpContext.Current.Response.StatusDescription = "Internal
Server Error";
}
```

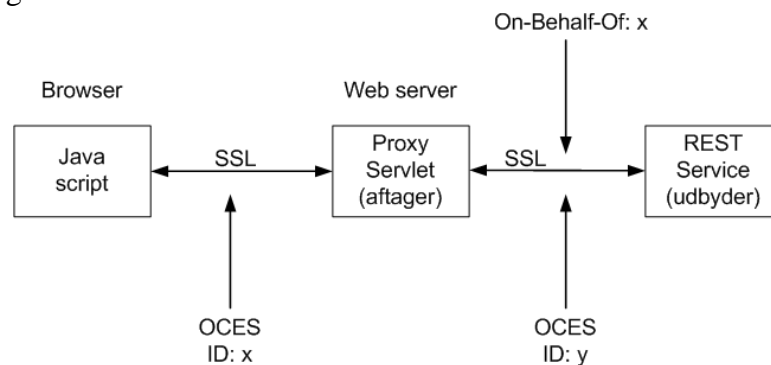
7.2 Klientside implementation: Serviceaftageren

Serviceaftageren, altså køservicens klient, er implementeret i Java. Klienten er lavet som en webapplikation baseret på GWT, altså en ”rig internet applikation”.

Arkitekturmæssigt er den ret simpel:



Proxyen er den centrale komponent. Det er den, som etablerer kommunikationen med REST-servicen, og det er derfor den, som etablerer sikkerheden mellem aftageren og udbyderens. Dette giver dog den udfordring, at SSL-forbindelsen mellem aftager og udbyder bliver etableret vha. et forudbestemt certifikat og ikke vha. brugerens personlige (som også kunne være et virksomhedscertifikat). Endelig er der også den udfordring, at sikkerheden mellem browseren og proxyen ikke må kunne kompromitteres, så der er behov for certifikat-login på et niveau mere, nemlig helt ude hos slutbrugerens:



Forbindelsen mellem slutbrugerens browser og web serveren etableres som en 2-vejs SSL baseret på slutbrugerens OCES certifikat, her simplificeret med ID'et ”x”. Proxyen etablerer en forbindelse mellem web applikationen og REST-servicen baseret på det forudbestemte certifikat – det certifikat, som trusten mellem web applikationen og REST-servicen er baseret på - her simplificeret ved ID ”y”. Det er herefter proxyens opgave at sende ID ”x” med i On-Behalf-Of-header informationen til servicen for hvert request, som den foretager på vegne af slutbrugeren. Det er lige præcis her, trust-begrebet kommer ind i billedet: REST-servicen stoler på, at headerinformationen vedr.

>

slutbrugeren er korrekt, altså at web applikationen/proxyen handler på vegne af slutbrugeren.

Lad os kigge på, hvordan vi i håndterer det i praksis. I første omgang skal vi have etableret en 2-vejs SSL mellem web applikation og REST-service; dette gør vi vha. Apache's HttpClient. Med følgende kode – placeret i servletens init-metode, tilvejebringer vi en instans af httpClient, som herefter bruges til den efterfølgende kommunikation fra servletens service metode(r):

```
try {
    keystore = KeyStore.getInstance("pkcs12");
    keystore.load(config.getServletContext().getResourceAsStream("/keystore/PIDTestBruger2.pfx"), "Test1234".toCharArray());

    truststore = KeyStore.getInstance("jks");
    truststore.load(config.getServletContext().getResourceAsStream("keystore/cacerts"),
        "changeit".toCharArray());

    socketFactory = new SSLSocketFactory(keystore, "Test1234",
    truststore);
} catch (Exception e) {
    e.printStackTrace();
    throw new ServletException("Unable to create SSLSocketFactory", e);
}

SchemeRegistry schReg = new SchemeRegistry();
Scheme sch = new Scheme("https", socketFactory, 443);
schReg.register(sch);

final HttpParams params = new BasicHttpParams();
ThreadSafeClientConnManager connManager = new
ThreadSafeClientConnManager(params,
    schReg);
httpClient = new DefaultHttpClient(connManager, params);
```

Opgaven er nu i servletens service-metoder, dvs. doPost, doGet, doPut og delete, at videresende requestet til REST-servicen, og når responset kommer tilbage herfra at sende dette videre tilbage til den klient, som oprindeligt invokerede proxy-servleten. I praksis gør vi det på følgende måde, her vist med doGet-metoden:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    service(req, res, new HttpGet("n/a"));
}
```

Af ovenstående kan vi se, at doGet ikke gør andet, end at delegere videre til en generel service-metode med et kald indeholdende det oprindelige req/resp par og den pågældende http-metode (get/post/put/delete). Service-metodens opgave er herefter at stykke et nyt request sammen mod en url sammensat af REST-servicens adresse + urien fra det oprindelige request:

```
private void service(HttpServletRequest req, HttpServletResponse res,
    HttpRequestBase httpRequest) throws IOException,
    ClientProtocolException {

    String uri = Configuration.getOne().serviceBaseUrl() +
    req.getPathInfo();
```

>

```
try {
    httprequest.setURI(new URI(uri));
} catch (URISyntaxException e) {...}

try {
    HttpResponse response = httpclient.execute(httprequest);
} finally {...}
```

I ovenstående mangler nu funktionaliteten til at håndtere, at svaret sendes tilbage til den oprindelige kalder, og hermed ser metoden således ud:

```
String uri = Configuration.getOne().serviceBaseUrl() +
req.getPathInfo();
try {
    httprequest.setURI(new URI(uri));
} catch (URISyntaxException e) {
    throw new RuntimeException(e);
}

((ThreadSafeClientConnManager)httpclient.getConnectionManager())
    .getConnectionsInPool();

if (httprequest instanceof HttpEntityEnclosingRequest) {
    HttpEntityEnclosingRequest httpEntityReq =
(HttpEntityEnclosingRequest) httprequest;
    InputStream requestStream = req.getInputStream();
    if (Boolean.parseBoolean(req.getHeader(HEADER_EXPAND_FILE_KEYS))) {
        requestStream = expandFileKeys(req.getInputStream());
    }
    httpEntityReq.setEntity(new InputStreamEntity(requestStream, -1));
}

copyRequestHeaders(req, httprequest);
long startMillis = System.currentTimeMillis();
InputStream source = null;
try {
    HttpResponse response = httpclient.execute(httprequest);
    logger.info(response.getStatusLine() + " after " +
(System.currentTimeMillis()-
startMillis) + " ms");
    HttpEntity entity = response.getEntity();

    if (entity != null) {
        source = entity.getContent();
        ServletOutputStream target = res.getOutputStream();
        res.setStatus(response.getStatusLine().getStatusCode());
        copyResponseHeaders(response.getAllHeaders(), res);
        copyStream(source, target, false);
    }
} finally {
    if (source != null ) {
        source.close();
    }
}
```

Vi mangler nu stadig at håndtere pålideligheden. Her bliver vi ”belønnet” af, at HttpClient-biblioteket allerede tilbyder en retry-mekanisme, så det handler blot om at anvende den:

>

```
        if ("get".equals(httprequest.getMethod().toLowerCase())) {
            httpclient.setHttpRequestRetryHandler(new
DefaultHttpRequestRetryHandler(5, true));
        }
```

Der er stadig en væsentlig ting, vi ikke har håndteret, nemlig den funktionalitet som skal sikre, at brugerens credentials sendes videre til serversiden. Dette håndteres ved at lægge et servlet filter foran proxy-servleten. Filteret kan herefter tage bruger-oplysningerne ud af OCES-certifikatet i requestet og lægge disse i en http-header, som herefter sendes videre til serveren af proxy-servleten. I filterets doFilter-metode foregår det på følgende måde:

```
        public void doFilter(ServletRequest req, ServletResponse res,
FilterChain chain) throws
            IOException, ServletException {
            Cert cert = getFirstCertificateFromRequest(req);
            storeCertInRequest(req, cert);
            chain.doFilter(req, res);
        }

        private Cert getFirstCertificateFromRequest(ServletRequest req) {
            X509Certificate[] certs = (X509Certificate[])
req.getAttribute(CERTIFICATE_REQUEST_KEY);

            if (certs == null || certs.length < 1) {
                return new NoCertAvailable(); // probably running in GWT hosted
mode
            } else {
                return new Cert(certs[0]);
            }
        }

        private void storeCertInRequest(ServletRequest req, Cert cert) {
            req.setAttribute(CERTINFO_REQUEST_KEY, cert);
        }
```

Herefter kan proxy-servleten gøre arbejdet færdigt ved at sætte certifikatets SSN-info på den rette http-header i requestet til REST-servicen:

```
Cert certInfo = ExtractCertificateSSN.getCertFromRequest(req);
certInfo.addOnbehalfOfHeader(httprequest);

public void addOnbehalfOfHeader(HttpRequestBase req) {
    final String headerValue = getSSN() + "," + getCN();
    req.addHeader(ON_BEHALF_OF_HEADER_NAME, headerValue);
}

private String getSSN() {
    return principalProperties.get("serialnumber");
}

private String getCN() {
    return findPropertyValue("cn");
}
```

>

Den komplette klient inkl. den færdige GUI kan downloades her:
<http://oioREST.dk/queueservice/documentation/samplecode.aspx>

8. Referencer

>

- [Referencebogen] Leonard Richardson & Sam Ruby: *RESTful Web Services*
O'Reilly Media.
ISBN-10: 0-596-52926-0
ISBN-13: 978-0-596-52926-0
- [Sikkerhedmodeller
for OIOREST] [*Sikkerhedsmodeller for OIOREST*](#),
IT- og Telestyrelsen.
- [Sig-bev] [*Signatur- og Systembevis – teknisk vejledning i sikring af digitale signaturers bevisværdi*](#),
IT- og Telestyrelsen.

