XML og server klient programmer:

Hvad er en server?

En server er et program som hele tiden lytter og venter på at en klient kalder op! En server er i princippet et meget simpelt program som man kan skrive selv med få linjer kode! En server reagerer på opkald på en bestemt måde. Den producerer en eller anden slags service.

Servere har ikke kun noget at gøre med et lokalnet eller med Internettet. Ethvert program som fungerer som en 'lytter' er en server!

Servere på et netværk som f. eks. Internettet reagerer oftest på den måde at de sender en bestemt HTML side retur hvis de kaldes op! Eller man kan sige at de sender den side tilbage til klienten som klienten har bedt om. Hvis jeg skriver dette i adresse linjen i en browser:

http://www.w3.org

Sender W3C Konsortiets server en velkomst side tilbage til mig.

En server skal kaldes gennem en bestemt protokol – http, ftp eller file – som er en lang række definitioner på hvad der er gyldigt sprog og syntaks! Protokollen svarer til et XML skema! Den definerer hvordan kommunikationen skal foregå. Hvis ikke klienten retter sig efter dette skema kan serveren ikke 'forstå' henvendelsen. I almindelighed kaldes klientens forespørgsel en request og serverens svar et response.

De følgende eksempler forudsætter at Microsoft IIS (Internet Information Server) er installeret på maskinen. I eksemplerne er også oprettet en ny virtuel mappe 'xml' for at gøre det lidt mere overskueligt. De efterfølgende filer skal altså ligge i denne virtuelle mappe og de skal kaldes op gennem en browser som f. eks. Internet Explorer!

Fordelene ved en XML server:

Der er massive fordele ved at bruge XML i kommunikationen server – klient. XML er et struktureret meget fleksibelt data format. XML er ren tekst – en såkaldt streng - som kan modtages på alle maskiner uanset styresystem og uanset software hos klienten. Klienten kan anvende script kode, C++, Java eller hvad som helst til at databehandle XML data. Klienten kan anvende en udgave af Windows, UNIX eller Linux! Klienten behøver overhovedet ikke at have en XML parser og slet ikke den samme parser som serveren har!

I HTTP/1.1 kan XML data komprimeres og XML formatet egner sig godt til at blive sendt komprimeret!

I XML teksten kan sendes binære data som et JPG billede eller et Word dokument! Alle klienter kan modtage disse binære data uanset operativsystem eller platform!

XML bruges nu i stor stil i e-handel og B2B – business to business kommunikation. Microsofts .NET og andre NET udgaver bygger helt på XML.

Eksempel: En telefon XML server:

Vi vil nu oprette en simpel server som kan levere data a la en telefon bog! Vi har skrevet en XML fil i stil med dette:

```
<?xml version="1.0"?>
<telefonliste>
<person id="ej">
<fornavn>Erik</fornavn>
<efternavn>Jensen</efternavn>
<telefon>67677889</telefon>
</person>
<person id="eh">
<fornavn>Eydna</fornavn>
<efternavn>Hansen</efternavn>
```

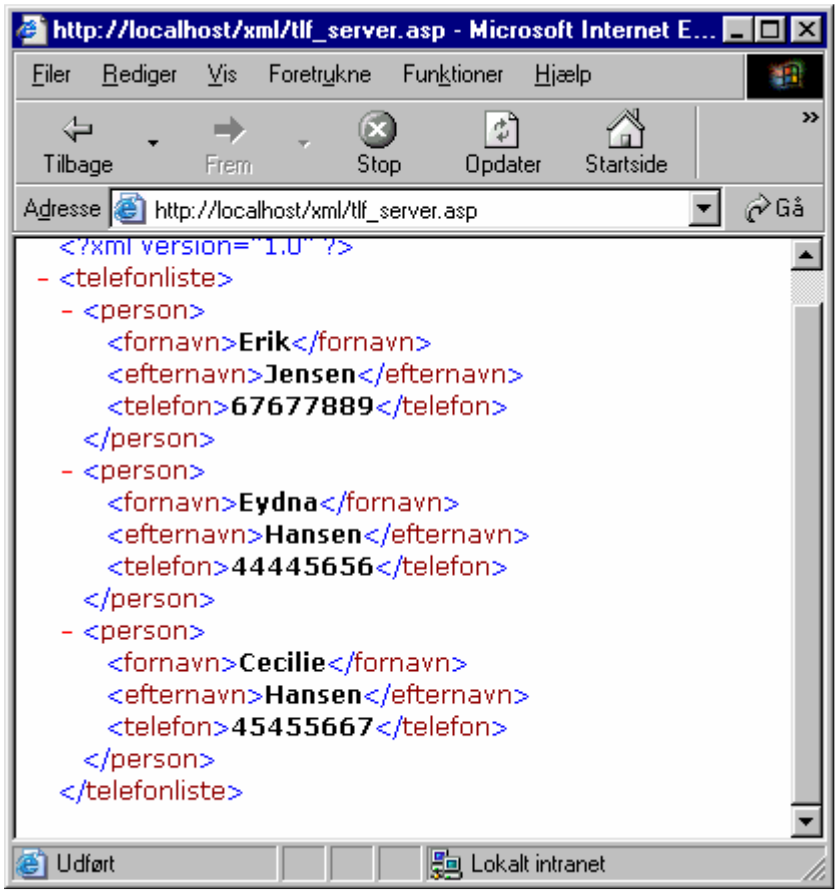
<telefon>44445656</telefon>
</person>
<person id="ch">
<fornavn>Cecilie</fornavn>
<efternavn>Hansen</efternavn>
<telefon>45455667</telefon>
</person>

</telefonliste>
Denne liste er gemt på serveren – i den virtuelle mappe – som telefonliste.xml. Vi kan så svare på opkald på denne måde:

```
<%@ language="VBScript"%>
<%
Option Explicit
Response.ContentType = "text/xml"
dim doc
set doc = server.createobject("Msxml2.DOMDocument.4.0")
doc.load Server.MapPath ("telefonliste.xml")
'Response.Write "<server_svar>" & doc.xml & "</server_svar>"
Response.Write doc.xml
%>
```

Vi har her valgt at skrive serveren som Visual Basic script kode der fungerer sammen med IIS serveren. Serveren defineres i en ASP fil og gemmes som tlf_server.asp. Vi sætter en ContentType til XML hvilket er det samme som vi sender en HTTP header tilbage til klienten! Vores server skal jo kaldes gennem protokollen HTTP! Vi instantierer et DOM dokument og loader det med telefon listen – og sender det hele retur til klienten! Formlen Server.MapPath sikrer at serveren kan finde den rigtige fil i den virtuelle mappe – hvor vi har gemt telefonliste.xml.
Vi kunne også sende andet tilbage til klienten som er antydet i kommentar linjerne!

Når vi kalder serveren i Internet Explorer ses dette:



Systemet kan beskrives således: Serveren henter data i en XML fil som fungerer som en database. Applikationen består altså – selv om den er uhyre enkel – af to lag eller 'tiers'. Vi kan altså forestille os at telefon listen bestod af 500.000 poster – og derfor er det nødvendigt at serveren kan søge og levere bestemte data til en klient!

Vi kan derfor forbedre serveren på denne måde:

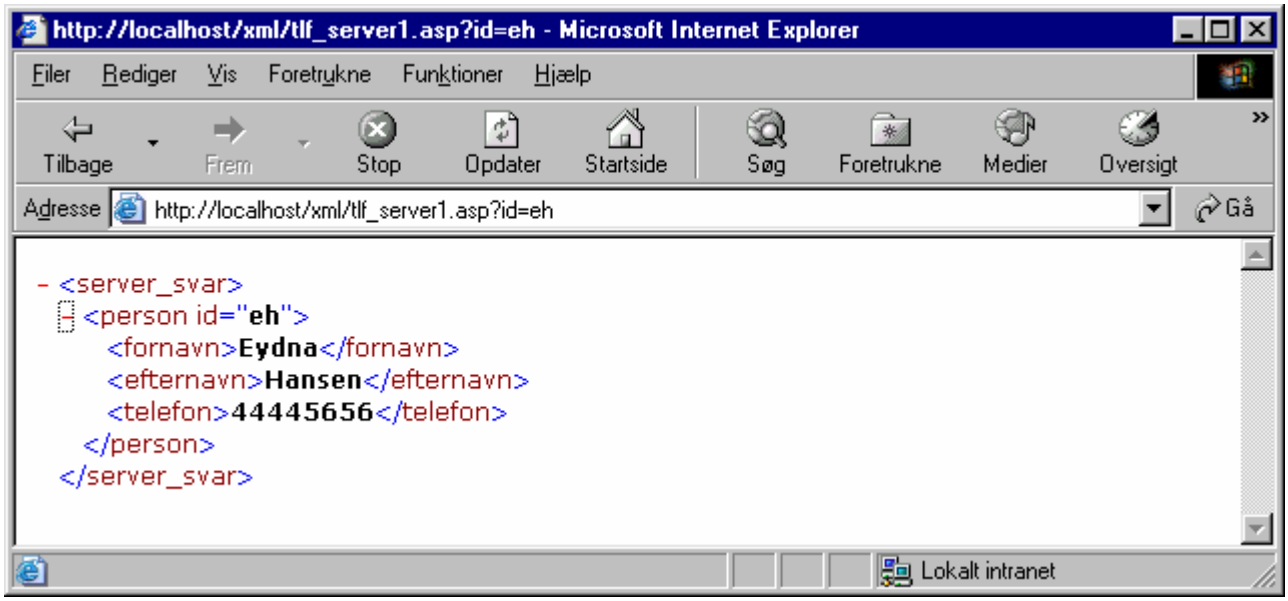
```
<%@ language="VBScript"%>
<%
Option Explicit
dim doc, fragment
dim req_id, str

req_id = Request.QueryString("id")
Response.ContentType = "text/xml"
set doc = server.createobject("Msxml2.DOMDocument.4.0")
doc.load Server.MapPath ("telefonliste.xml")
str = "//person[@id='" & req_id & "']"
set fragment = doc.selectSingleNode (str)
Response.Write "<server_svar>" & fragment.xml & "</server_svar>"
%>
```

I ASP findes et objekt Request som har et under objekt QueryString som indeholder de parametre som klienten har brugt til at kalde op til serveren Normalt stammer disse 'søge ord' fra en HTML

form men her taster vi foreløbigt blot parameteren direkte ind i browserens adresselinje! Dette kan gøres på denne måde:

http://localhost/xml/tlf_server1.asp?id=eh

Serveren finder nu parameteren 'id' og bruger den til at søge i DOM dokumentet – bruger altså helt almindelige DOM metoder. Vi skriver en XPATH sti til den attribut som svarer til id og returnerer attributtens parent til klienten!

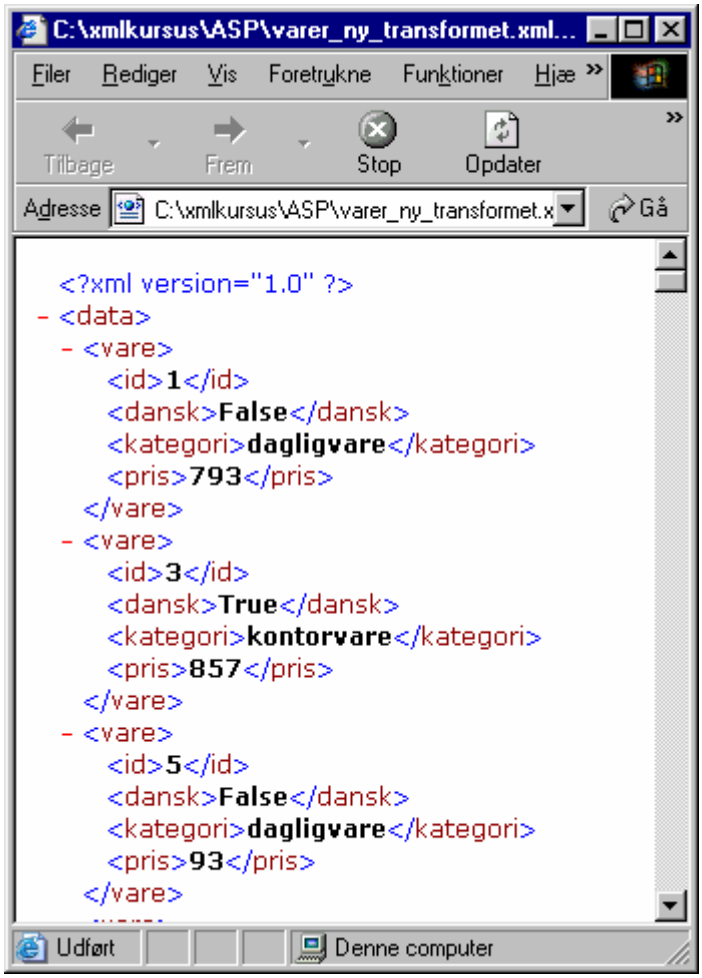


Vi kan også hente en klient parameter på en enklere måde:

req_id = Request("id")

I dette tilfælde søger serveren i begge samlinger – både QueryString og Form samlingerne.

Vi kan lave et tilsvarende eksempel i denne XML fil:

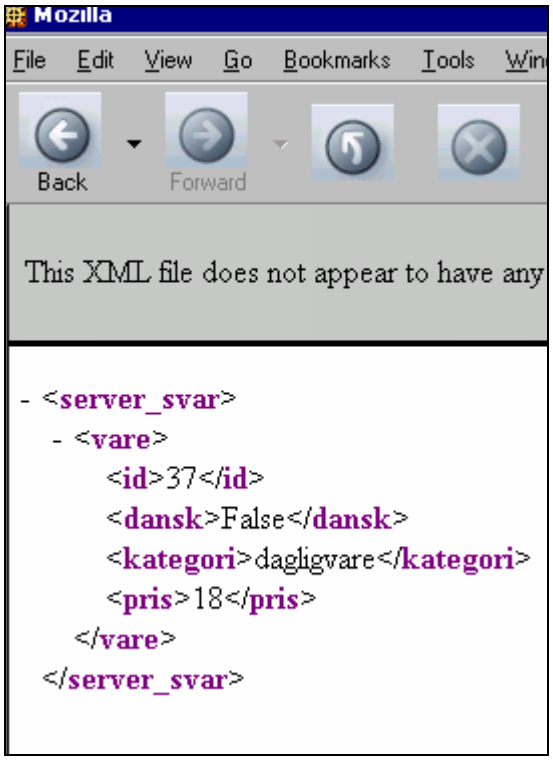


Vi kan nu søge i dokumentet f. eks. efter pris sådan:

```
<%@ language="VBScript"%>
<%
Option Explicit
dim doc, fragment
dim req_pris, req_kategori, str
req_pris = Request.QueryString("pris")
‘req_kategori = Request("kategori")
Response.ContentType = "text/xml"
set doc = server.createobject("Msxml2.DOMDocument.4.0")
doc.load Server.MapPath ("varer_ny_transformet.xml")
str = "//vare/pris[. < " & req_pris & "]/.."
set fragment = doc.selectSingleNode (str)
Response.Write "<server_svar>" & fragment.xml & "</server_svar>"
%>
```

Klienten kan nu søge på en vare som koster mindre end 20 kr!:

http://localhost/xml/varer.asp?pris=20



XML server der trækker på en database:

Et meget stort XML dokument kan fungere som en slags database for en XML server men mere almindeligt er det at serveren trækker på en rigtig **database**. Vi skal se nogle eksempler herpå.

I det følgende eksempel bruger vi en Microsoft **Access** database tabel som er registreret på maskinen som en ODBC datakilde under navnet 'boghandel'. For at eksemplet kan køre skal man altså have en sådan database registreret som en System DNS og en tabel 'bog' med disse poster:

ASP koden kan så skrives sådan:

```
<% @language = VBScript %>
<%
Response.ContentType="text/xml"

' Fra Access database til XML dokument med SAX metoder

dim recordset,query,connection

set writer=Server.CreateObject("Msxml2.MXXMLWriter.4.0")
set reader=Server.CreateObject("Msxml2.SAXXMLReader.4.0")
set attr=Server.CreateObject("Msxml2.SAXAttributes.4.0")
set doc=Server.CreateObject("Msxml2.DOMDocument.4.0")
writer.indent=true
writer.omitXMLDeclaration= true
writer.output = doc
set reader.contentHandler = writer

if isobject(Session("databasesession")) then
 set connection = session("databasesession")
else
set connection = server.createobject("ADODB.Connection")
call connection.open("boghandel","","")
set session("databasesession") = connection
end if

set recordset = server.createobject("ADODB.Recordset")
call recordset.open("select * from bog", connection)

reader.contentHandler.startDocument
reader.contentHandler.startElement "", "", "boghandel", attr

while not recordset.EOF
  reader.contentHandler.startElement "", "", "bog", attr


  reader.contentHandler.startElement "","", "efternavn", attr
    reader.contentHandler.characters recordset("efternavn").value
  reader.contentHandler.endElement "","", "efternavn"

  reader.contentHandler.startElement "","", "fornavn", attr
    reader.contentHandler.characters recordset("fornavn").value
  reader.contentHandler.endElement "","", "fornavn"

  reader.contentHandler.startElement "","", "titel", attr
    reader.contentHandler.characters recordset("titel").value
  reader.contentHandler.endElement "","", "titel"

  reader.contentHandler.endElement "", "", "bog"

  recordset.MoveNext
wend

reader.contentHandler.endElement "", "", "boghandel"
reader.contentHandler.endDocument
recordset.close
connection.close
```

```
response.write doc.xml

%>
```

Vi kalder databasen på denne måde:

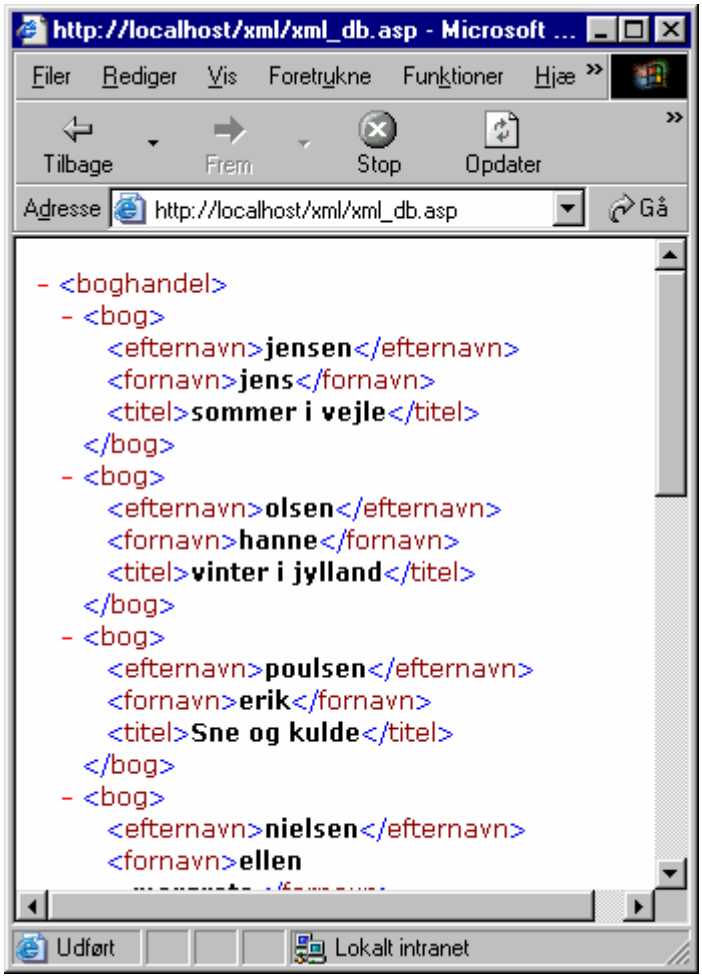
```
set recordset = server.createobject("ADODB.Recordset")
call recordset.open("select * from bog",connection)
```

Vi opretter et record set (et udvalg af poster fra tabellen) – i dette tilfælde vælger vi alle poster fra tabellen bog.

Derefter skriver vi en XML tekst ud fra posterne i tabellen. Vi bruger her en SAX writer. For hver post eller række opretter vi et element bog og for hvert felt i posten opretter vi et nyt sub element som fornavn, efternavn og titel. Fordelen ved denne metode er at vi har fuld **kontrol** over hvordan vi ønsker at vores XML dokument skal se ud.

Vi kunne have undgået at anvende en DOM struktur og have 'streamet' writer'en direkte til klienten! Dette ville have gjort processen hurtigere og nemmere men vi ville så ikke kunne manipulere med XML dokumentet.

Når vi har læst alle poster i tabellen sender vi XML dokumentet til klienten!



Direkte adgang til databasens Recordset:

Vi kan skyde en genvej til adgang til en database tabel. Det er nemligt muligt at få overført et ADO recordset direkte til XML! I .NET og i anden sammenhæng er dette udviklet meget længere således at hele systemet er integreret med XML data formatet.

Vi kan gemme et recordset med en parameter 0 eller 1 idet 1 betyder adPersistXML. Koden hertil er væsentligt enklere end den vi lige har set men ulempen er at vi i første omgang ikke kan kontrollere det XML dokument der kommer ud af processen!:

```
<% @language = VBScript %>

<%
Response.ContentType="text/xml"
dim recordset,query,connection
set doc=Server.CreateObject("Msxml2.DOMDocument.4.0")

if isobject(Session("databasesession")) then
set connection = session("databasesession")
else
set connection = server.createobject("ADODB.Connection")
call connection.open("boghandel","","")
```

```
set session("databasesession") = connection
end if

set recordset = server.createobject("ADODB.Recordset")
recordset.open "select * from bog", connection
recordset.Save doc, 1
Response.Write doc.xml
recordset.close
connection.close

%>
```

Vi får på denne måde dette XML dokument:

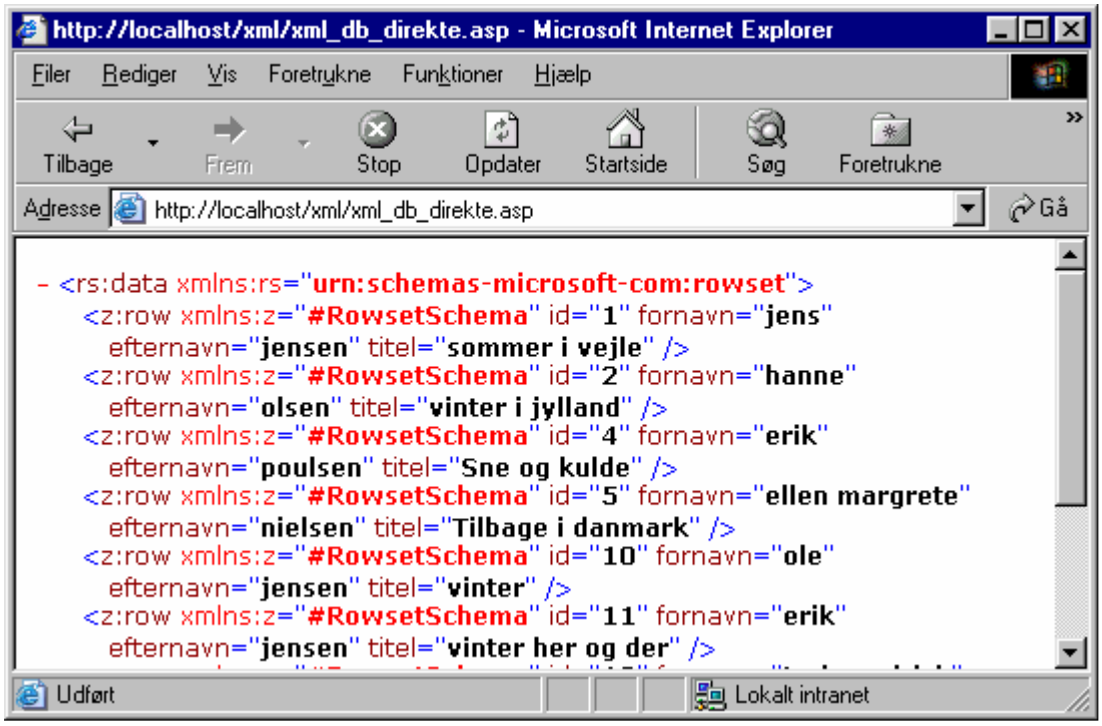
```
- <xml xmlns:s="uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882"
    xmlns:dt="uuid:C2F41010-65B3-11d1-A29F-00AA00C14882"
    xmlns:rs="urn:schemas-microsoft-com:rowset" xmlns:z="#RowsetSchema">
  - <s:Schema id="RowsetSchema">
    - <s:ElementType name="row" content="eltOnly">
      + <s:AttributeType name="id" rs:number="1">
      - <s:AttributeType name="fornavn" rs:number="2" rs:nullable="true"
          rs:write="true">
          <s:datatype dt:type="string" dt:maxLength="50" />
        </s:AttributeType>
      - <s:AttributeType name="efternavn" rs:number="3" rs:nullable="true"
          rs:write="true">
          <s:datatype dt:type="string" dt:maxLength="50" />
        </s:AttributeType>
      - <s:AttributeType name="titel" rs:number="4" rs:nullable="true"
          rs:write="true">
          <s:datatype dt:type="string" dt:maxLength="50" />
        </s:AttributeType>
        <s:extends type="rs:rowbase" />
      </s:ElementType>
    </s:Schema>
  - <rs:data>
      <z:row id="1" fornavn="jens" efternavn="jensen" titel="sommer i vejle"
        />
      <z:row id="2" fornavn="hanne" efternavn="olsen" titel="vinter i jylland"
        />
      <z:row id="4" fornavn="erik" efternavn="poulsen" titel="Sne og kulde"
        />
      <z:row id="5" fornavn="ellen margrete" efternavn="nielsen"
        titel="Tilbage i danmark" />
      <z:row id="10" fornavn="ole" efternavn="jensen" titel="vinter" />
      <z:row id="11" fornavn="erik" efternavn="jensen" titel="vinter her og
        der" />
      <z:row id="12" fornavn="torben ulrich" efternavn="jensen" titel="vinter"
        />
      <z:row id="13" fornavn="marie-louise" efternavn="hansen" titel="jeg er
        aldrig hjemme" />
      <z:row id="14" fornavn="hans" efternavn="johansen" titel="svensk
        vinter" />
```

```
    </rs:data>
</xml>
```

Vi får først XDR skemaet til XML dokumentet og derefter alle poster i tabellen i et bestemt format. Recordset bliver ALTID oversat således til XML. Vi kan derefter bruge DOM metoder på dokumentet. Hvis vi f. eks. i stedet vælger

Response.Write doc.firstChild.lastChild.xml

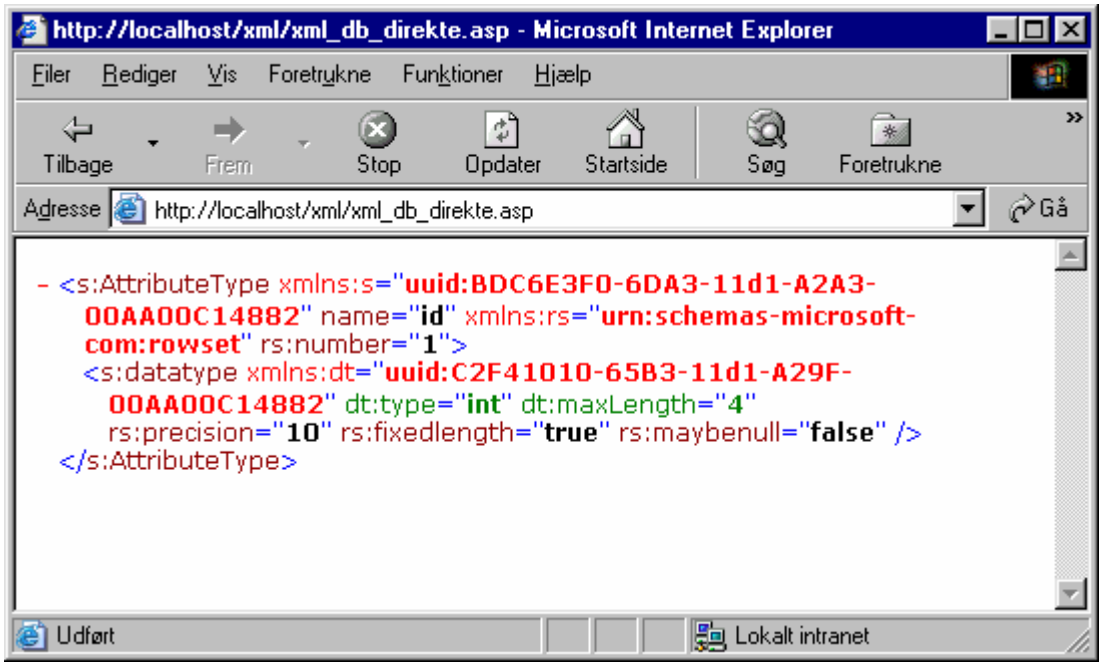
får vi selve data sektionen:



Hvis vi vælger:

Response.Write doc.firstChild.firstChild.firstChild.firstChild.xml

får vi definitionen i XDR skemaet af den 1. attribut i rækkerne:



Vi kan se at XDR er ret forskelligt fra XSD skemaer og er Microsofts interne skema opfindelse. Men vi kan også se at data fra databasen bliver bevaret i skemaet. Dette er også et eksempel på et internt skema. Dette er også muligt med XSD men ikke hensigtsmæssigt og det frarådes generelt!

En ulempe ved denne metode er selvfølgeligt at man ikke kan undgå at få leveret skemaet sammen med XML dokumentet!

Direkte 'streaming' med SAX:

Vi kan undgå den **overhead** der ligger i DOM modellen ved at 'streame' XML dokumentet direkte til klienten således:

```
<%@ language="VBScript"%>
<%

Option Explicit
response.contenttype="text/xml"
dim w, r
set w=server.createobject("Msxml2.MXXMLWriter.4.0")
'w.omitXMLDeclaration = true
w.encoding = "iso-8859-1"
set r=server.createobject("Msxml2.SAXXMLReader.4.0")

set r.contentHandler=w
r.parseURL Server.MapPath("..\telefonliste_basisstyle.xml")
response.write w.output
%>
```

Denne metode sætter en SAXWriter til at være event **handler** for vores SAX reader! På denne måde sendes hele dokumentet blot videre med:

```
response.write w.output
```

Writerens output er XML dokumentet og det kan sendes direkte til klienten!

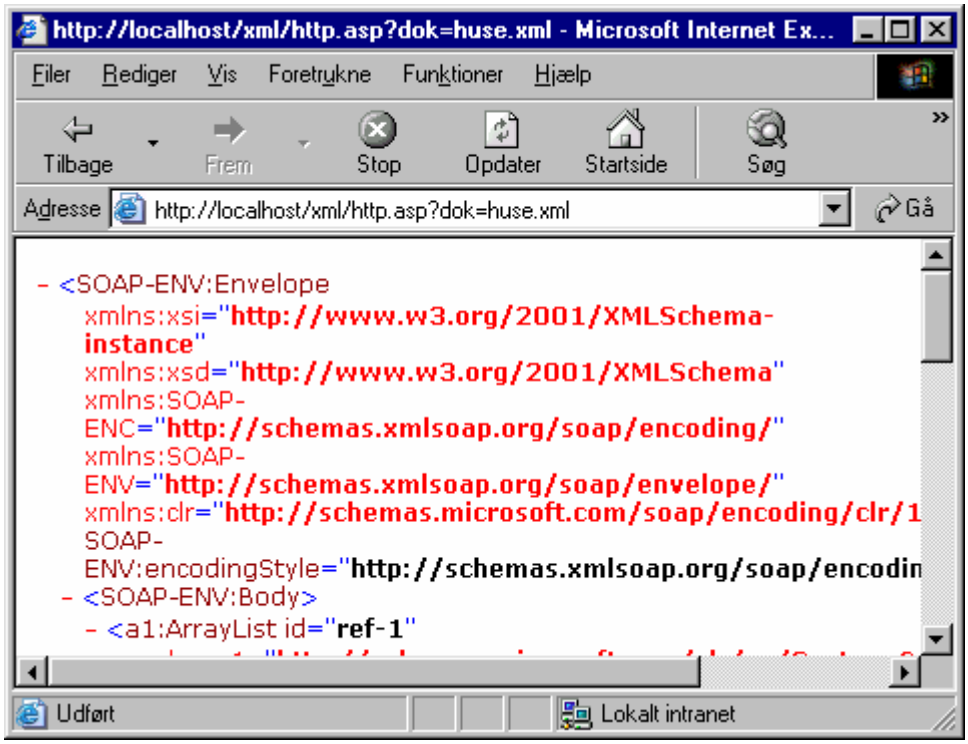
En server, der kalder op til en anden server:

Følgende eksempel illustrerer hvordan en XML server kan hente data på en **anden** server og sende dem videre til en klient som kalder op. Dette system kan ligeså godt anvendes direkte af en **klient** – som det kan ses! Vi har altså også på denne måde defineret en såkaldt **proxy** server altså et gennemgangsled ud til en anden server!

Eksemplet her bygger på at klienten sender ønsket om et bestemt **dokument** til serveren som så henter det på en **anden** server (her har vi brugt to forskellige virtuelle **mapper** som eksempel men systemet kunne lige så godt kalde op til en fjern server på Internettet eller på et intranet/lokalnet).

```
<%@ language="VBScript"%>
<%
Option Explicit

Response.ContentType = "text/xml"
Dim remote_server, dok
dok=Request.QueryString("dok")
set remote_server = Server.CreateObject("Msxml2.XMLHTTP.4.0")
remote_server.open "GET", "http://localhost/" & dok, false
remote_server.send
Response.Write remote_server.responseXML.xml
%>
```

Vi bruger her klassen **XMLHTTP** som etablerer en helt normal **HTTP** forbindelse! HTTP serveren defineres med metoden **open** og requesten bliver sendt med metoden **send**. Vi sender ansøgningen **false** dvs. synkront. Hvis vi ved at responsen er et XML dokument kan vi bruge responseXML ellers kan vi bruge **responseBody** som returnerer en array (tabel) af bytes – som kan bruges ved modtagelsen af binære data. Vi kan også bruge **responseText**.

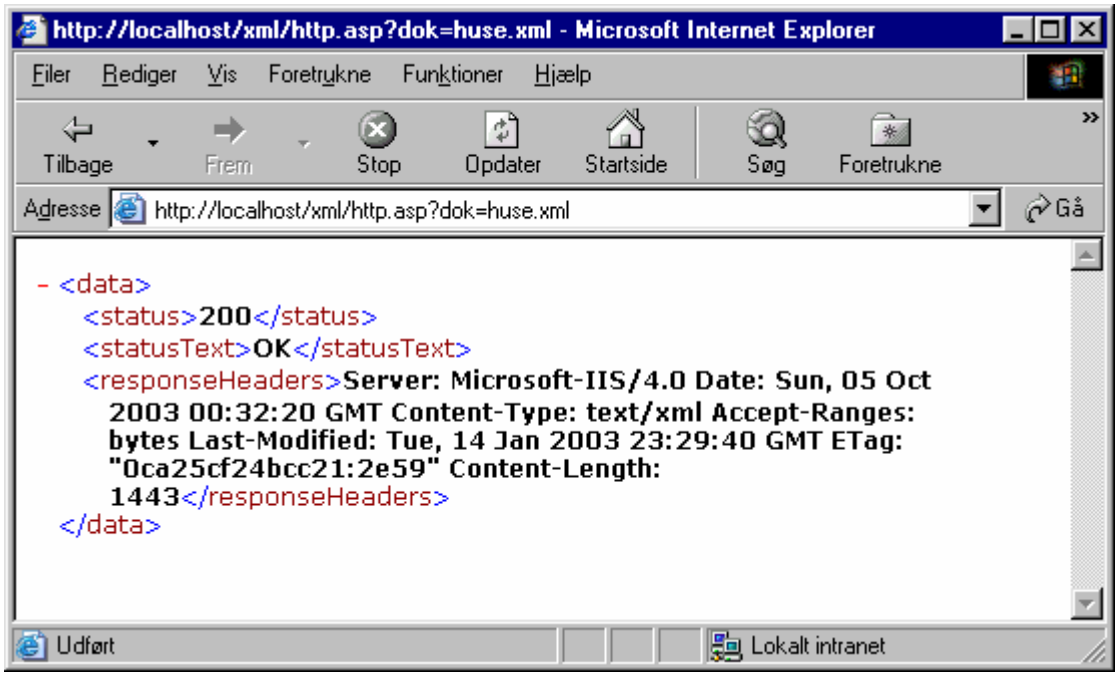


XMLHTTP implementerer **WinInet** COM komponenten som beskriver en række API'er der bruges til at kommunikere på et netværk. Med WinInet kan man skrive sin egen server forholdsvis let.

Vi kan også hente og kontrollere en række data om **HTTP** kommunikationen:

```
<%@ language="VBScript"%>
<%
Option Explicit
Response.ContentType = "text/xml"
Dim remote_server, dok
dok=Request.QueryString("dok")
set remote_server = Server.CreateObject("Msxml2.XMLHTTP.4.0")
remote_server.open "GET", "http://localhost/" & dok, false
remote_server.send
Response.Write "<data><status>" & remote_server.status & "</status><statusText>" & remote_server.statusText &
"</statusText>"
Response.Write "<responseHeaders>" & remote_server.getAllResponseHeaders & "</responseHeaders></data>"
%>
```

Resultatet af at køre dette program kan f. eks. være:



I HTTP betyder en retur **kode** på 200 at dokumentet er returneret OK. Desuden sender serveren som vores server har kaldt op til en række HTTP **headers** tilbage – eksempler kan ses ovenfor. Vi gør det her at vi opbygger et nyt XML dokument og sender det direkte til klienten!

Når vi sender en request af sted med XMLHTTP kan vi også definere en lang række request headers f. eks. om serveren skal sende dokumentet tilbage hvis det ikke er blevet opdateret siden i går!

NB ovenstående kode viser hvordan vi kan **checke** om kommunikationen er gået OK! Inden vi sender noget tilbage til klienten skal vi normalt sikre os at status koden er 200! Ellers skal der sendes en besked om fejlen tilbage til klienten (error handling).

Ovenstående eksempler henter en **statisk** fil fra serveren men systemet kunne lige så godt anvendes til at hente en **dynamisk** service fra en server! Hvis vi kalder op til en server med visse parametre f. eks. kan vi få den anden server til at producere en vis form for **service** og sende resultatet tilbage til os. Dette er grundlaget for de såkaldte **web services** på Internettet (og lokalnet). Dette er også grundlaget for at skabe en distribueret applikation hvor forskellige servere i fællesskab løser en opgave.

Send binære data: JPG billede:

Følgende kan illustrere hvordan vi kan hente et JPG billede på en anden server og sende det retur til klienten – vi bruger stort set koden fra før:

```
<%@ language="VBScript"%>
<%
Option Explicit
Response.ContentType = "image/jpeg"
Dim remote_server, dok
```

```
dok=Request.QueryString("dok")

set remote_server = Server.CreateObject("Msxml2.XMLHTTP.4.0")

remote_server.open "GET", "http://localhost/" & dok, false
remote_server.send

Response.BinaryWrite remote_server.responseBody

%>
```

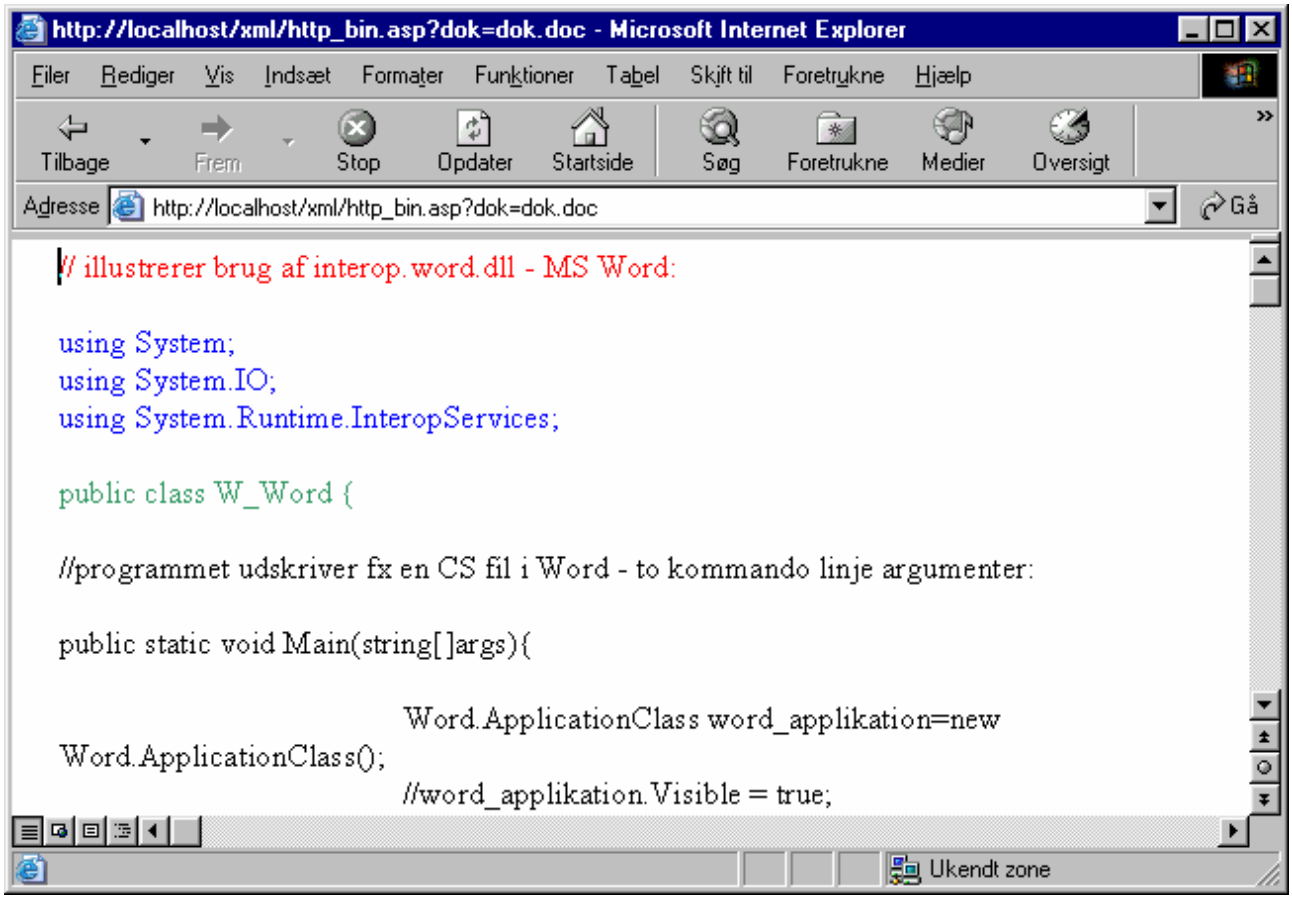
Det eneste nye her at server objektet har en **BinaryWrite** som kan sende binære data direkte til klienten! Hvis klienten her beder om et bestemt billede prøver serveren at hente det på den anden server og hvis det går OK bliver det sendt tilbage til klienten.

Vi kan ikke bruge Vis Kilde i browseren til at se teksten når vi modtager et billede fordi det modtagne ikke er tekst!

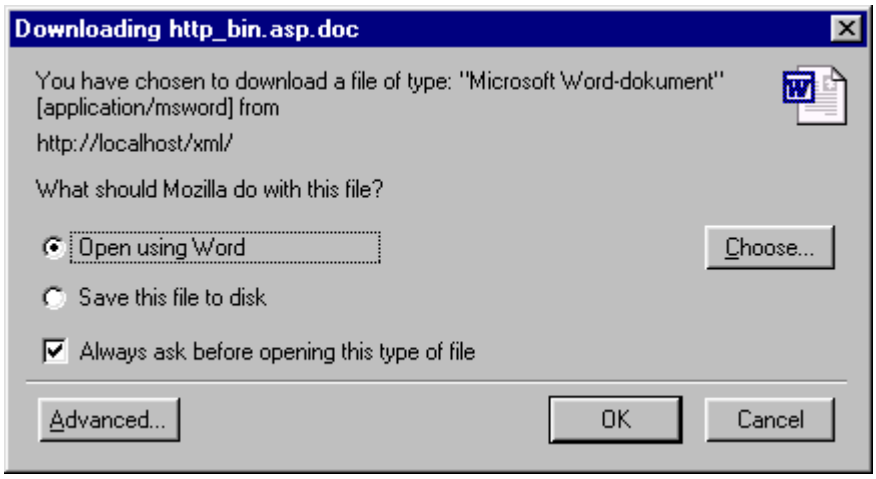
Med BinaryWrite kan vi sende ALLE slags binære filer til en klient! Ovenstående eksempel kunne lige så godt bruges til at hente f. eks. et MS **Word** dokument! Kun en enkelt linje nemlig MIME typen på den binære type skal (eller bør) ændres:

Response.ContentType = "application/msword"

Vi kan du kalde serveren med en dok parameter og få returneret et Word dokument i Internet Explorer:



Hvis ASP siden åbnes i Mozilla browseren kommer denne boks frem:

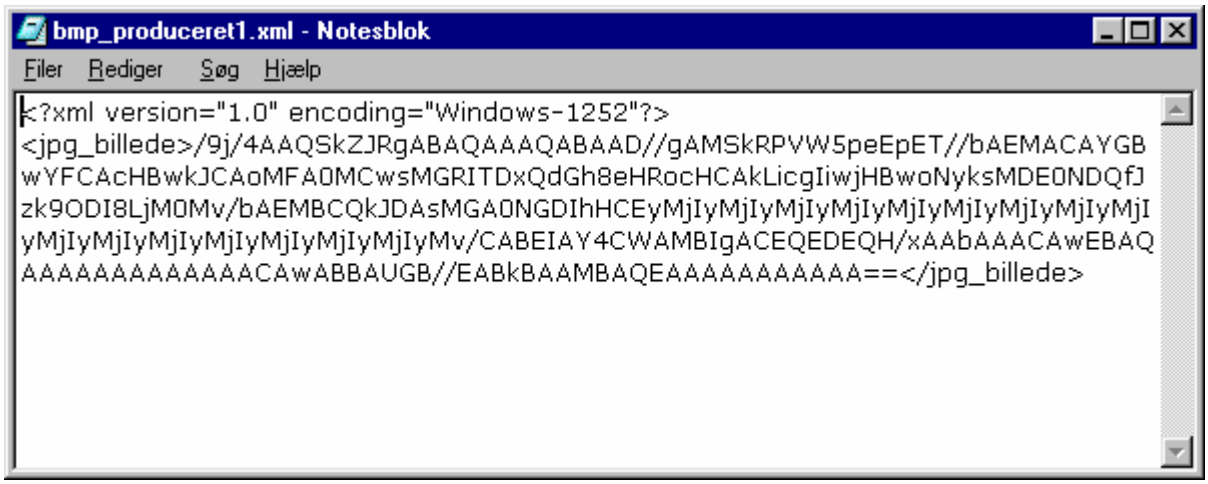


ResponseBody som blev brugt ovenfor er en tabel eller array af **bytes** – en binær kodning af billedet, dokumentet eller objektet.

Binære data i XML dokumenter:

Billeder, Word dokumentet eller andre objekter kan også **kodes** binært og sendes som en **del** af XML dokumentet! I ovenstående eksempel var billedet eller objektet jo **ikke** en del af XML dokumentet.

Vi kan kode disse data på forskellige måder. I afsnittet om **SAX** er vist hvordan man kan kode f. eks. et JPG billede i BinHex eller Base64. Hvis vi forudsætter at vi har kodet et – meget lille – billede, kan det se sådan ud kodet i Base64:



Som man kan se kan sådanne XML noder selvfølgeligt sendes direkte til en klient som en del af en større XML tekst! I **XSD** findes en særlig type som er beregnet til at indeholde binære data – se afsnittet om XSD!

Base64 kodning anvender et **alfabet** af 64 tegn (0-9, +, /, a-z, A-Z). Den koder 4 tegn (32 bits) i det som 3 tegn normalt fylder nemlig i grupper på 24 bits. Denne kodning er mere **effektiv** end BinHex der blot oversætter hver byte i den binære fil til to bytes. Se i øvrigt afsnittet om SAX writer'en.

Styling af serverens dokumenter:

Hvis XML dokumenter sendes retur til klienten vil klienten i nogle tilfælde være interesseret i selve XML teksten i nogle tilfælde være interesseret i en tekst der er stylet – hvor serveren har anvendt et stylesheet!

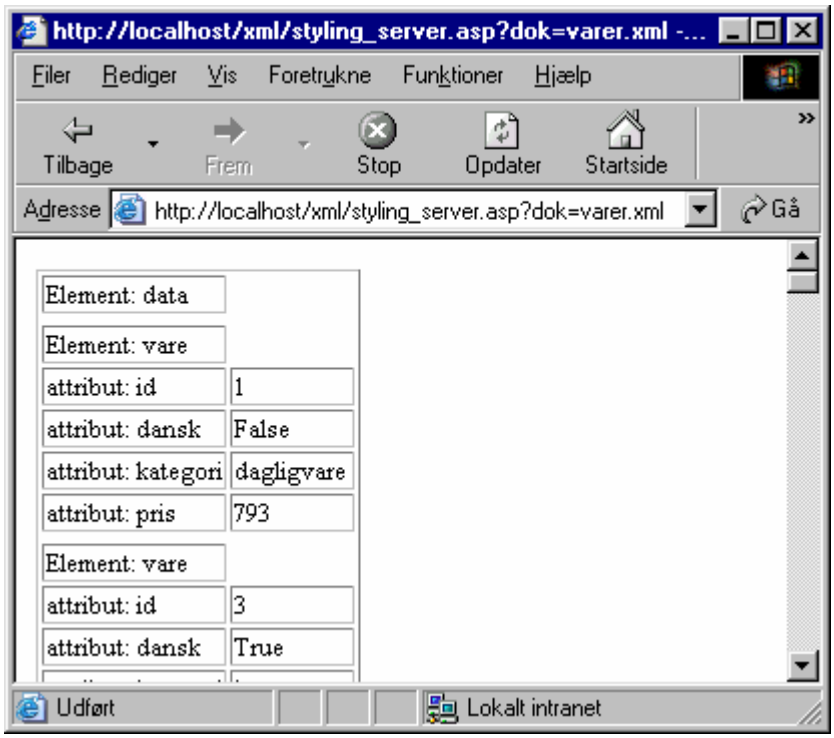
Alle browsere forstår i dag en CSS styling af et XML dokument. Vi kan ikke forudsætte at en klient har en XML parser så derfor er en styling en oplagt mulighed!

Vi kan skrive specielle stylesheets til de enkelte responser men vi kan også anvende mere generelle stylesheets. Som et eksempel på det sidste kan man anvende dette helt **generelle** stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<table border="1">
<xsl:for-each select="//*">
<tr>
```

```
<td><xsl:value-of select="concat('Element: ',name())" /></td>
</tr>
<xsl:for-each select="@*">
<tr>
<td><xsl:value-of select="concat('attribut: ',name())" /></td>
<td><xsl:value-of select="." /></td>
</tr>
</xsl:for-each>
<tr>
<td><xsl:value-of select="." /></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>

</xsl:stylesheet>
```

Vi kan nu bruge eksemplet fra før – at klienten beder om en bestemt side:



Uanset hvilket dokument klienten beder om sætter vi det samme stylesheet på dokumentet og sender det retur til klienten. **Fordelen** er at **alle** klienter kan modtage dette format: HTML med CSS stylesheets!:

```
<%@ language="VBScript"%>

<%
Option Explicit

Response.contentType = "text/html"

Dim docxml, docxsl, docout, dok
```

```
dok = Request("dok")
set docxml = server.createobject("Msxml2.DOMDocument.4.0")
set docxsl = server.createobject("Msxml2.DOMDocument.4.0")

docxml.async = false
docxsl.async = false

docxml.load Server.MapPath(dok)
docxsl.load Server.MapPath("generelt.xsl")

Response.Write docxml.transformNode(docxsl)

%>
```

Vi kunne selvfølgeligt også have gemt det stylede XML dokument – så det evt. kunne genbruges næste gang en klient kalder op og beder om det samme dokument! Vi kunne også bruge transformNodeToObject (doc, doc) der gemmer det stylede dokument. Også subtræer i et XML dokument kan styles på denne måde.

Ved hjælp af XSLT stylesheets kan vi også style det samme XML dokument til helt forskellige modtagere – dels forskellige browsere som Netscape eller Internet Explorer – dels til andre typer som mobil telefoner (WAP) eller PDA'er (Personal Digital Assistents)!

Ideen med XML er netop at dokumentets **informations** indhold jo skal være **uafhængigt** af dets præsentation eller styling! Helt modsat havd der gælder i HTML som primært er et styling sprog!

Kontrol af data fra klienten:

I mange tilfælde er det nødvendigt at en server kontrollerer de data som den modtager fra en klient. Dette kan nemt gøres i XML med et XSD skema (eller et DTD skema). Vi kan forestille os at serveren har en slags log in procedure hvor klienten skal sende en bestemt række af data til serveren. Vi kan f. eks. definere et skema således:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="data">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="bruger" type="xsd:string"></xsd:element>

<xsd:element name="kode">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:enumeration value="abc123" />
<xsd:enumeration value="xyz987" />
<xsd:enumeration value="pqr246" />
</xsd:restriction>
</xsd:simpleType>
</xsd:element>

<xsd:element name="id" type="xsd:string"></xsd:element>
```

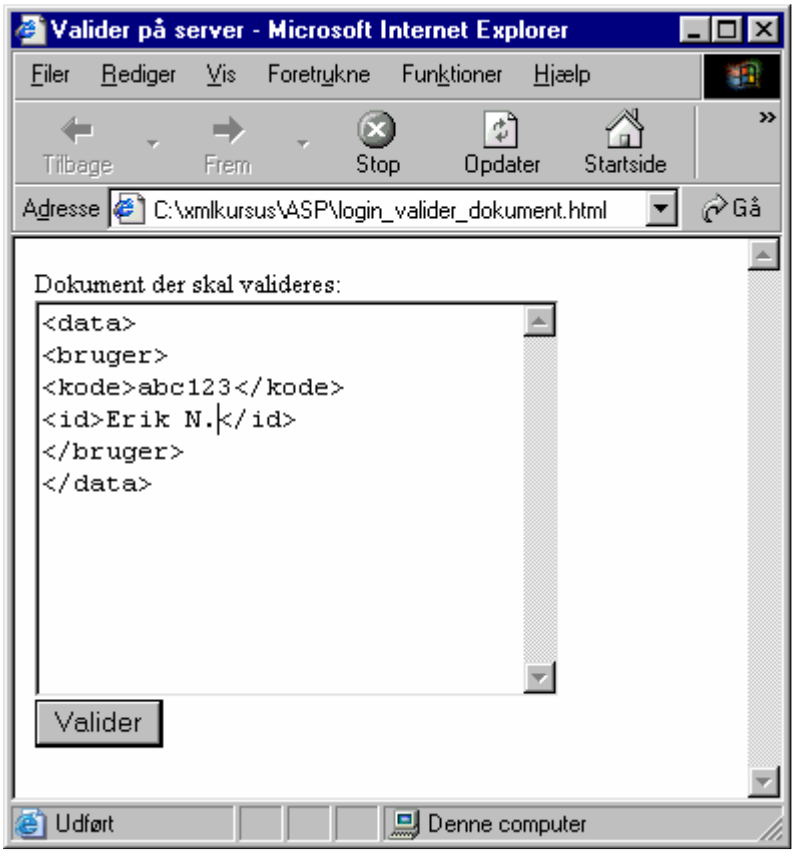
```
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

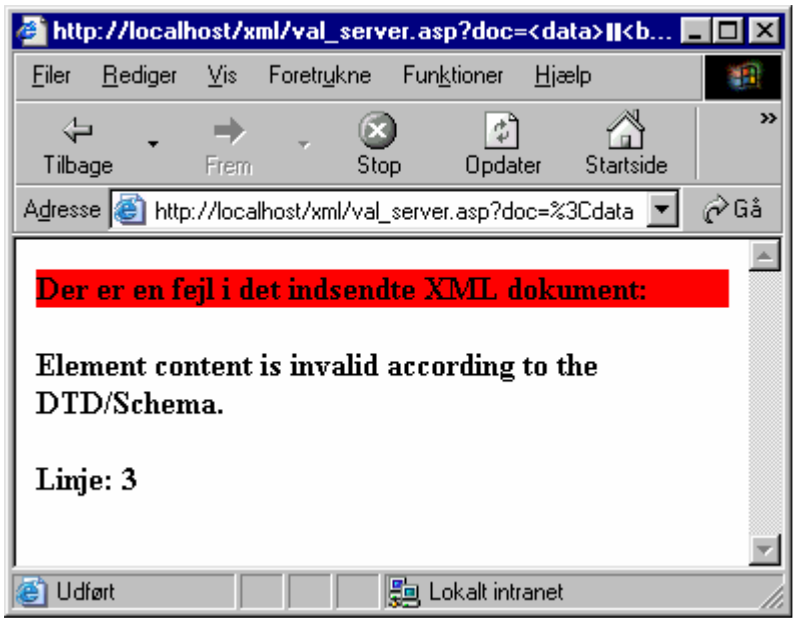
Skemaet definerer altså dels hvad der er en gyldig form i log in proceduren dels - som et eksempel – også gyldige brugere! Dette sidste kan vi gennemføre med en enum som opregner alle gyldige brugere!

Vi kan nu skrive en enkel formular som klienten kan bruge til at sende XML data til serveren:

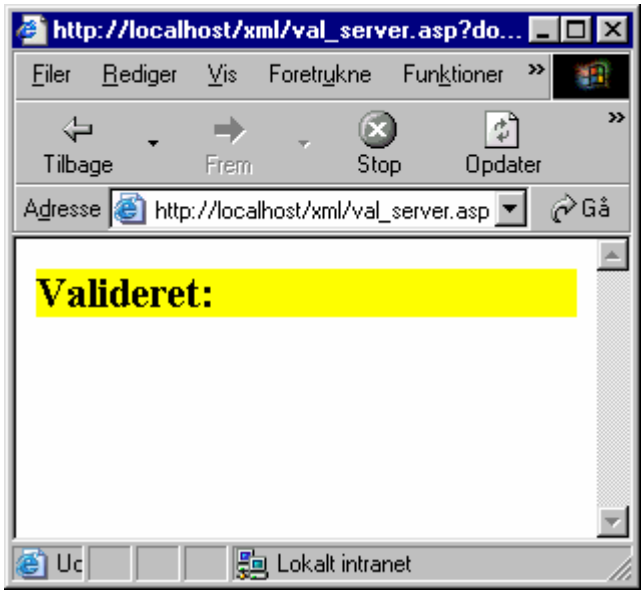
```
<Html>
    <Head>
        <Title>Valider på server</Title>
    </Head>
    <Body>
    <form action="http://localhost/xml/val_server.asp" method="get">
    Dokument der skal valideres:<br>
    <textarea name="doc" id="doc" rows=12 cols=30></textarea>
    <br>
    <input type="submit" value="Valider">
    </form>
    </Body>
</Html>
```

Når der klikkes på submit knappen aktiveres serveren på ASP siden val_server.asp. Vi indtaster først et ugyldigt dokument:





Dette dokument er derimod gyldigt:



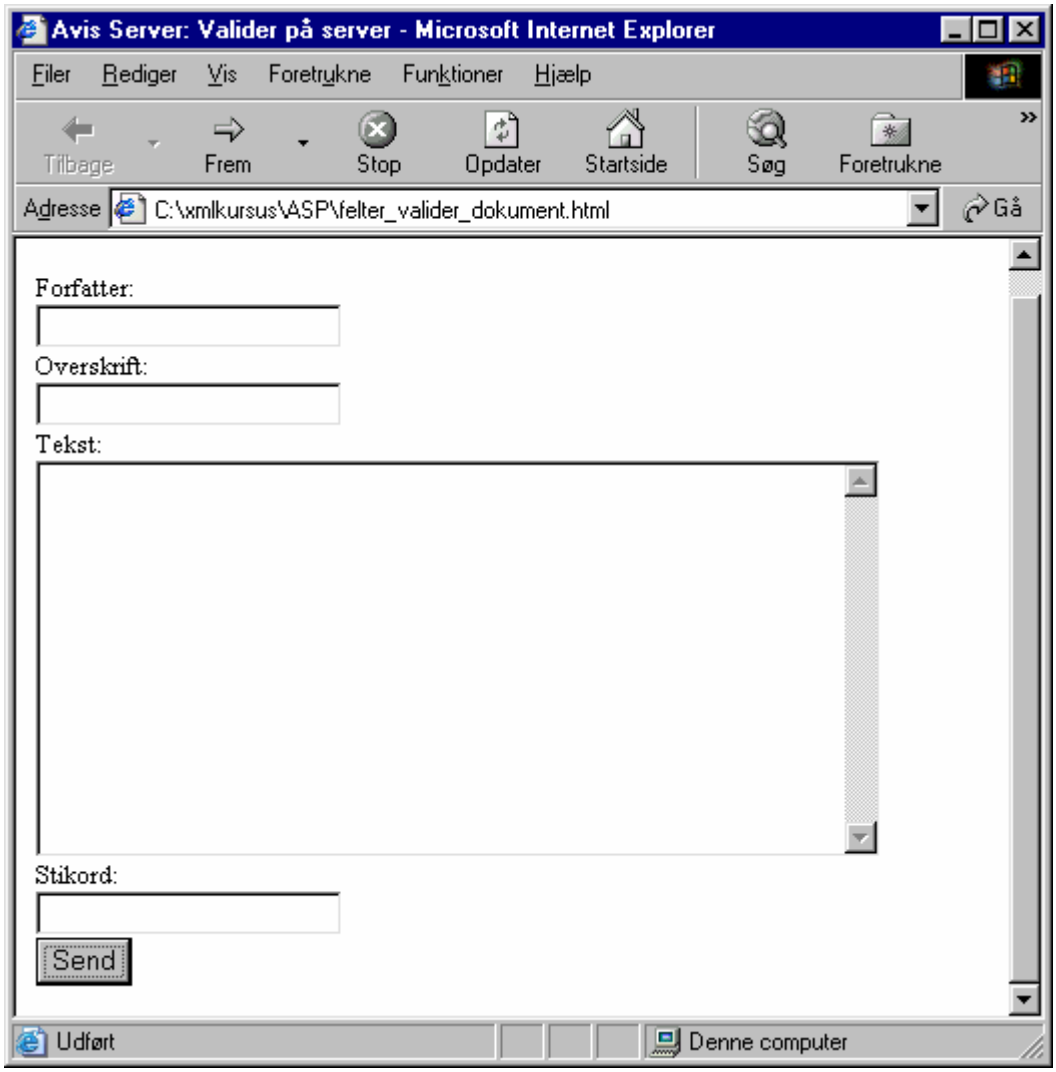
Servere som modtager data fra klienter skal normalt altid en lignende procedure! F. eks. skal en server som kontrollerer **Dankort** kontrollere at dels formen dels indholdet er gyldigt! Eller en **avis** server hvor artikler til en avis kan indtastes skal checke at det indtastede overholder et bestemt format for at kunne bruges siden hen!

Vores ASP dokument som kontrollerer request'en ser sådan ud:

```
<%@ language="VBScript"%>

<%
Option Explicit

Dim docxml, docxsl, docout, str, skema

str = request.querystring("doc")
set docxml = server.createobject("Msxml2.DOMDocument.4.0")
set skema = server.createobject("Msxml2.XMLSchemaCache.4.0")
skema.add "",  "http://localhost/xml/login.xsd"

docxml.validateOnParse = true
docxml.async = false
docxml.schemas = skema
docxml.loadXML str


if docxml.parseError.errorCode <> 0 then
response.write "<h3 style='background-color:red'>Der er en fejl i det indsendte XML dokument: </h3><h3>" &
vbnewline & docxml.parseError.reason & "</h3><h3> Linje: " & docxml.parseError.line

response.end
end if

Response.Write "<h2  style='background-color:yellow'>Valideret:</h2> "

%>
```

Eksempel: En avis server:

Vi kan her **genbruge** koden fra før i høj grad. Vi ønsker en bruger grænse flade som denne:



Vi kan skrive et XSD skema i stil med dette:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="data">
<xsd:complexType>
<xsd:sequence>
   <xsd:element name="forfatter" />
   <xsd:element name="overskrift" />
   <xsd:element name="tekst" />
   <xsd:element name="stikord">
      <xsd:simpleType>
```

```
      <xsd:restriction base="xsd:string">
      <xsd:enumeration value="indenrigs" />
      <xsd:enumeration value="international" />
      <xsd:enumeration value="kultur" />
      </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Nu kan vi validere/kontrollere de indtastede data på samme måde som før! Skemaet kan selvfølgeligt gøres mere eller mindre stramt. Her har vi brugt det eksempel at stikord skal være af en bestemt værdi – en **enumeration**. På samme måde kunne vi kontrollere at kun visse personer er i stand til at sende artikler til avisen!

NB Når vi som her opretter en skema cache **gemmer** vi det kompilerede skema og kan bruge det samme kompilerede skema igen og igen! På den måde opnår vi en stor **performance** gevinst – serveren vil arbejde væsentligt hurtigere! Man kan tilføje en række skemaer til en skema cache INDEN serveren går i gang med at modtage klienter! Disse skemaer ligger så klar til brug. De kan også gemmes i objektet Application på denne måde så de fungerer lige så længe serveren kører!:

```
skema_cache = Application("skema_cache")

if skema_cache = null then
set skema_cache = server.createobject("Msxml2.XMLSchemaCache.4.0")
skema_cache.add "uri:aviser_namespace",  Server.MapPath ("avis.xsd")
skema_cache.add "uri:aviser_namespace1",  Server.MapPath ("avis1.xsd")
skema_cache.add "uri:aviser_namespace2",  Server.MapPath ("avis2.xsd")

Application("skema_cache") = skema_cache
end if

docxml.schemas = skema_cache
```

Metoden **add** fungerer på den måde at den første parameter er det **targetNamespace** som skemaet har og den anden parameter er skemaet **placering** eller **URL**.

En server der cacher en række stylesheets:

På samme måde er det hensigtsmæssigt at de stylesheets som serveren bruger igen og igen bliver cachede altså gemt i RAM hukommelsen så længe serveren kører. De ligger så hele tiden klar. Ellers skulle de for hver gang hentes – enten fra disken eller via HTTP fra en netværksadresse! Det er en indlysende mere effektiv foranstaltning at de ligger klar når de skal bruges til at formattere dokumenter til klienterne!

Det system som anvendes i MSXML er at oprette en XSLT **processor**! Når vi har en sådan processor kan vi anvende en række nye metoder når vi styler!

Dette kan ske på denne måde:

```
<%@ language="JScript"%>

<%
Response.contentType = "text/html"

var xsl = new ActiveXObject("Msxml2.FreeThreadedDOMDocument.4.0");
var doc = new ActiveXObject("Msxml2.DOMDocument.4.0");
doc.load("c:/xmlkursus/asp/tlf.xml");

xsl.async=false;
xsl.load("c:/xmlkursus/asp/generelt.xsl");

var template=Server.CreateObject("Msxml2.XSLTemplate.4.0");
template.stylesheet=xsl;
Session(“style_cache”)=template;

var processor=template.createProcessor();
processor.addParameter("farve","'red'");
processor.input= doc;
processor.output= Response;
processor.transform();

%>
```

Vi lægger her vores templat over i Session dvs. at templaten lever videre som et globalt objekt hen over hele den session som pågår med klienten! På den måde spares meget og processen går hurtigere!

En templat er et objekt som kan rumme et FreeThreadedDOMDocument dvs. et dokument som kan bruges af mange processer samtidigt! Et almindeligt DOMDocument er ’rental’ threaded dvs. det kan kun bruges i een proces og dør derefter!

Vi bruger vores templat objekt til at oprette en XSLT processor. Når vi har en processor kan vi tilføkje parametre til stylingen og tilføje en startMode (hvis vi har defineret XSLT templater med forskellige modes!). En processor gør altså stylingen langt mere fleksibel. Det som kan opnås på denne måde svarer præcist til at anvende msxsl.exe (jvf afsnittet om XSLT stylesheets).

I dette tilfælde transformer vi direkte til en stream nemlig til Response stream’en til klienten. Vi kunne også have valgt en output der var et DOMDocument!

Den viste metode er altså ikke kun gyldig i server programmering men også hvis vi i øvrigt vil gennemføre en styling af et XML dokument. Se i øvrigt afsnittet om XSLT stylesheets.

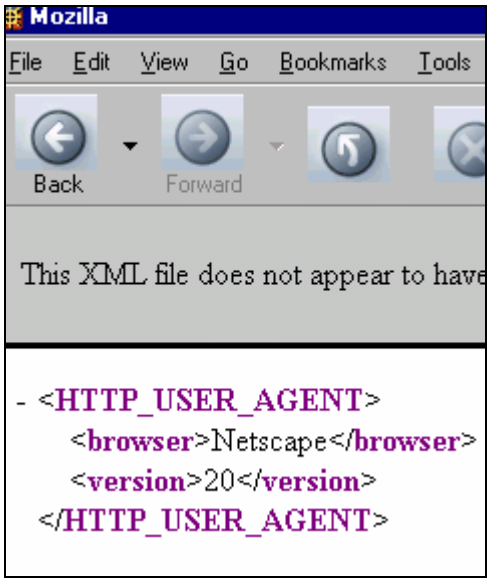
Forskellige styles til forskellige modtagere:

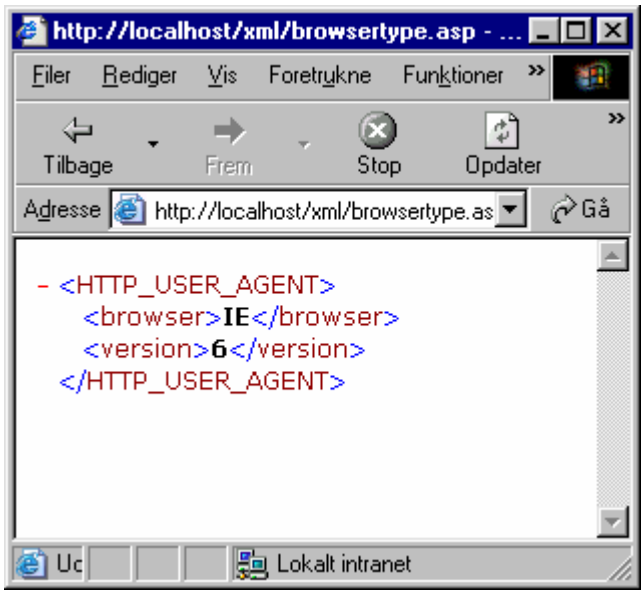
Vi kan på en enkelt måde finde ud af data om den klient der kalder op ved hjælp af en variabel ”HTTP_USER_AGENT” som er en del af de HTTP headers som en klient sender til serveren.

Dette kan f. eks. gøres sådan:

```
<%@ language="VBScript"%>

<%
Option Explicit
Dim str, browser, version
Response.contentType = "text/xml"

str = Request.ServerVariables ("HTTP_USER_AGENT")

if instr(str, "IE") then
browser = "IE"

version=cint(mid(str, instr(str, "MSIE") +5,1))

elseif instr(str, "Gecko") then
browser = "Netscape"
version = cint(mid(str, instrrev(str, "/") +1, 2))

elseif instr(str, "Mozilla") then
browser = "Netscape"
version = cint(mid(str, revinstr(str, "/") +1, 1))

end if

Response.write
"<HTTP_USER_AGENT><browser>"&browser&"</browser><version>"&version&"</version></
HTTP_USER_AGENT>"


%>
```

Når vi kalder op får vi disse beskeder:





Dette kan så udnyttes til at **differentiere** med forskellige **stylesheets** til forskellige browsere. I dette eksempel bruger vi grundlæggende det samme stylesheet men det kaldes med forskellige parametre og forskellig **startMode**!:

```
<%@ language="vbscript"%>

<%
Response.contentType = "text/html"

str = Request.ServerVariables ("HTTP_USER_AGENT")

if instr(str, "IE") then
browser = "IE"

version = cint(mid(str, instr(str, "MSIE") +5,1))
```

```
elseif instr(str, "Gecko") then
browser = "Netscape"
version = cint(mid(str, instrrev(str, "/") +1, 2))

elseif instr(str, "Mozilla") then
browser = "Netscape"
version = cint(mid(str, revinstr(str, "/") +1, 1))

end if


set xsl = Server.CreateObject("Msxml2.FreeThreadedDOMDocument.4.0")
set doc = Server.CreateObject("Msxml2.DOMDocument.4.0")
doc.load "c:/xmlkursus/asp/tlf.xml"

xsl.async = false
xsl.load "c:/xmlkursus/asp/generelt.xsl"

set template = Server.CreateObject("Msxml2.XSLTemplate.4.0")
template.stylesheet = xsl

set processor = template.createProcessor()

if browser = "IE" then
processor.addParameter "farve","'#669966'"
processor.setStartMode ("div")
end if

if browser = "Netscape" then
processor.addParameter "farve","'#eeffee'"
processor.setStartMode ("tabel")

end if

processor.input= doc
processor.output = Response
processor.transform()

%>
```

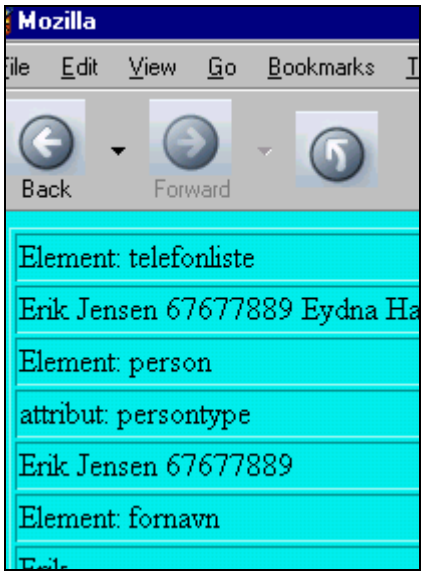
Vores stylesheet **generelt**.xsl ser således ud:

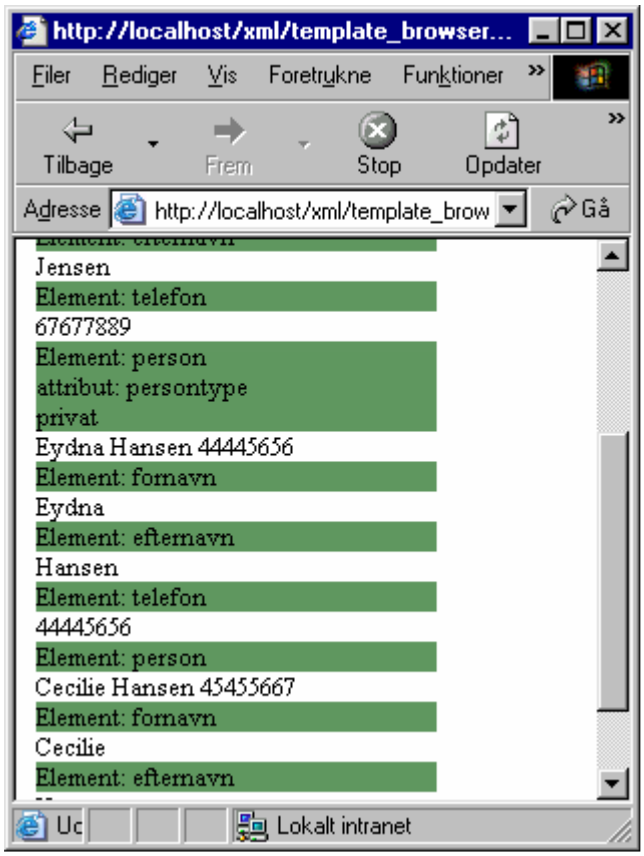
```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="farve"  />

<xsl:template match="/" mode="tabel">
```

```
<body bgcolor="{$farve}">
<table border="1">
<xsl:for-each select="//*">
<tr>
<td><xsl:value-of select="concat('Element: ',name())" /></td>
</tr>
<xsl:for-each select="@*">
<tr>
<td><xsl:value-of select="concat('attribut: ',name())" /></td>
<td><xsl:value-of select="." /></td>
</tr>
</xsl:for-each>
<tr>
<td><xsl:value-of select="." /></td>
</tr>
</xsl:for-each>
</table>
</body>
</xsl:template>

<xsl:template match="/" mode="div">
<xsl:for-each select="//*">
<div style="width:150pt;background-color:{$farve}">
<xsl:value-of select="concat('Element: ',name())" />
</div>

<xsl:for-each select="@*">
<div style="width:150pt;background-color:{$farve}">
<xsl:value-of select="concat('attribut: ',name())" />
</div>
<div style="width:150pt;background-color:{$farve}">
<xsl:value-of select="." />
</div>
</xsl:for-each>
<div>
<xsl:value-of select="." />
</div>
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Vi kan nu se hvordan **forskellige browsere** modtager forskellige transformationer! Vi anvender en mode = ”tabel” til Netscape og en mode = ”div” til Internet Explorer!

Dette system er endnu mere interessant hvis vi f. eks. skelner mellem **WAP** (mobil **telefoner**) på den ene side og **browsere** på den anden side osv! På denne måde kan en XML server betjene meget forskellige klienter – med grundlæggende de **samme** XML dokumenter!





Data Islands eller XML øer:

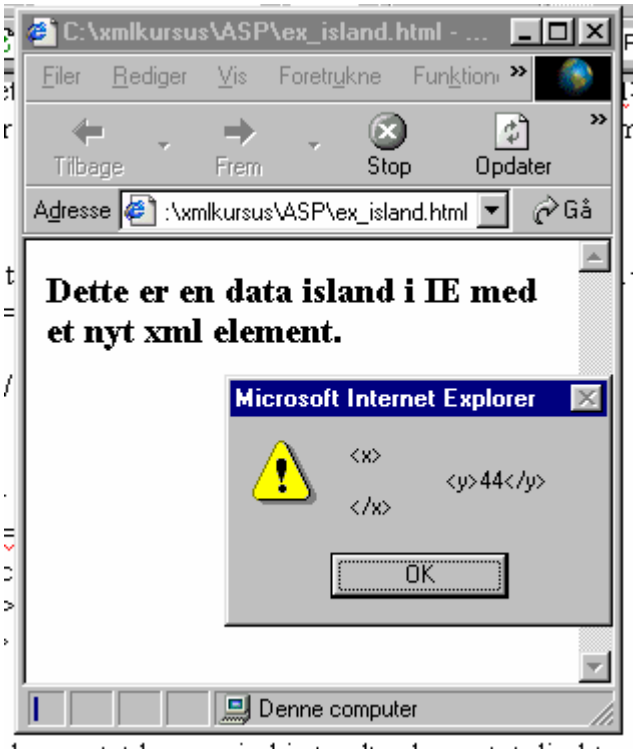
Vi vil i det følgende se på hvordan en server kan **sende** XML dokumenter til en klient. Når vi taler om en 'server' mener vi egentligt en **hvilken** som helst server: På Internettet, på et lokalt net (eller intra net) eller denne ene uafhængige maskine!

Det er meget almindeligt at sende data i form af en – isoleret - data **island** eller 'ø'. Fordelen er at XML dokumentet sendes **sammen** med HTML siden til klienten. Klienten skal altså ikke kalde op til serveren to eller måske mange gange for at hente de relevante XML data! Klienten får det hele på een gang!

I Internet Explorer er en data **island især** en ny tag <xml> som forstås af Internet Explorer – og **kun** af denne browser! Dette kan i strukturen skrives på denne måde:

```
<html>
<body>
<h3>Dette er en data island i IE med et nyt xml element.</h3>
                    <xml id="x">
                    <x>
                    <y>44</y>
                    </x>
                    </xml>
<script>
var doc=x.XMLDocument;
alert(doc.xml);
</script>
</body>
</html>
```

XML dokumentet lægges ind i <xml> elementet direkte og browseren 'forstår' koden xml! Internet Explorer har skabt en short cut idet programmet instantierer et **DOMDocument** med **MSXML** parseren og indlæser teksten i <xml<! Derfor fungerer dette **kun** i Internet Explorer! Hvis vi viser dette HTML dokument vises dette:



Det interessante er mest at IE forstår at den **ikke** skal vise tekst indholdet af den nye tag **<xml>**! Normalt vil en browser **altid** vise tekst indholdet **hvis** den støder på et element som den **ikke** kender! Hvis jeg skriver:

<mintekst>Dette er min tekst.</mintekst>

vil en browser normalt **overspringe** den tag som den ikke forstår og blot vise **teksten** eller tekst noden! Det er derfor at <xml> **kun** fungerer i IE!

Til sammenligning er her **Mozilla** som viser det samme dokument:

I ovenstående eksempel kan vi se noget andet: Det er muligt at sende XML data til en klient og samtidigt medsende et 'bruger **interface'**! Således at klienten **cacher** (gemmer i hukommelsen) XML dokumentet og samtidigt kan **bruge** det, søge i materialet osv. Vi – eller klienten - kan anvende f. eks. alle metoder fra **DOM**!

Serveren kan altså på den måde sende et program med evt. mange data til klienten!

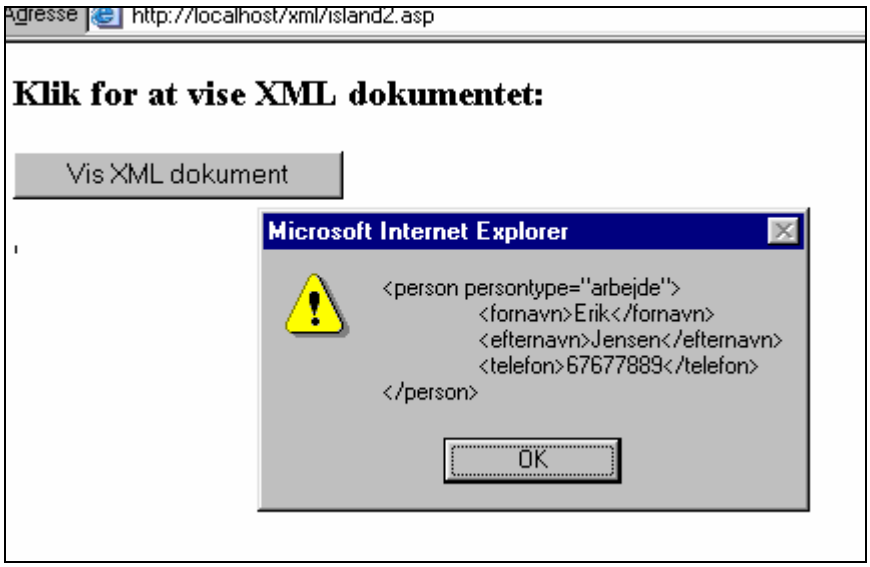
I nedenstående eksempel er dette skrevet i et **ASP** dokument som klienter kan kalde op til:

```
<%@ LANGUAGE="JSCRIPT" %>

<xml id="xml_document">
   <person persontype="arbejde">
   <fornavn>Erik</fornavn>
   <efternavn>Jensen</efternavn>
   <telefon>67677889</telefon>
   </person>
</xml>

<form>
<h3>Klik for at vise XML dokumentet: </h3>
<input type="button" value="Vis XML dokument" onclick="vis()" />
</form>
```

```
<script>
   function vis(){
   var doc=xml_document.XMLDocument;
   alert(doc.xml);
   }
</script>
</html>'
```

Her kan vi se at serveren sender en lille **form** med en knap som kan **aktivere** – kalde - en script funktion. Dette system kan **udbygges** næsten uden grænser! Læg mærke til at selve XML dokumentet er **skjult** – kun tilgængeligt gennem interfaces eller metoder! Det er ikke meningen at XML dokumentet som sådant skal vises (men man kan selvfølgeligt bruge Vis Kilde!). Bemærk at vi kan skrive HTML teksten **direkte** i ASP dokumentet uden ekstra koder!

Fordelen er - som vi så - at klienten **ikke** skal kalde op til serveren hele tiden – men selv sidder med materialet – cachet!

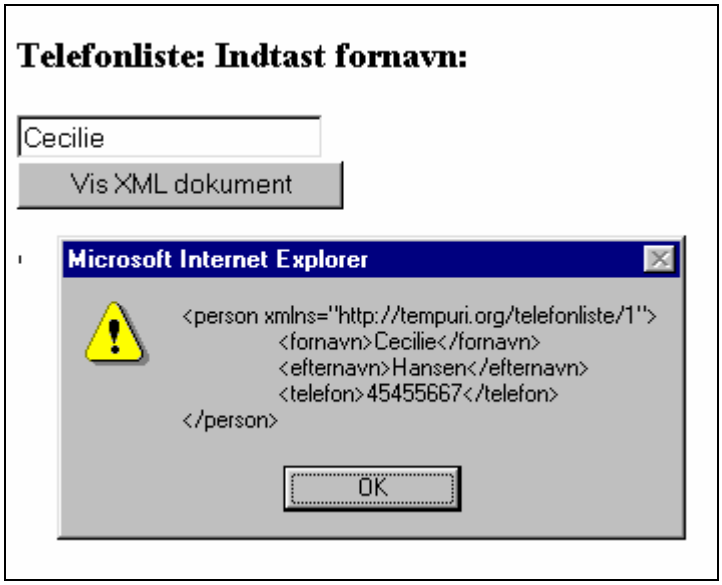


Et andet og mere fleksibelt eksempel på øer i Internet Explorer er følgende hvor et – eventuelt stort – XML dokument indlæses og hvor klienten også modtager en søgeform til at søge i dokumentet:

```
<%@ LANGUAGE="JSCRIPT" %>

<html>
<xml id="xml_document" src="tlf.xml" />
<form>
   <h3>Telefonliste: Indtast fornavn:</h3>
   <input type="text" value="Eydna" id="tekst" name="tekst" />
   <br />
   <input type="button" value="Vis XML dokument" onclick="vis()" />
</form>

<script>
   function vis(){
   var doc=xml_document.XMLDocument;
```

```
    var str=document.all("tekst").value;
    //alert(doc.selectSingleNode("//fornavn[.='Eydna']").xml);
    var nodeset=doc.selectSingleNode("//fornavn[.='"+str+"']");

    if(nodeset==null)alert("Personen er ikke fundet.");
    else {
    alert(nodeset.parentNode.xml);

    }
    }
</script>
</html>'
```

Programmet søger efter et bestemt fornavn og leverer den første person som har dette fornavn – der kunne jo være flere!:



Denne metode er mere fleksibel fordi **<xml>** bruger en **src** attribut hvorved hele dokumentet ikke skal stå i selve ASP dokumentet – hvilket jo ikke altid er lige praktisk! I dette tilfælde skal både HTML dokumentet og src-dokumentet downloades fra serveren - to gange download!

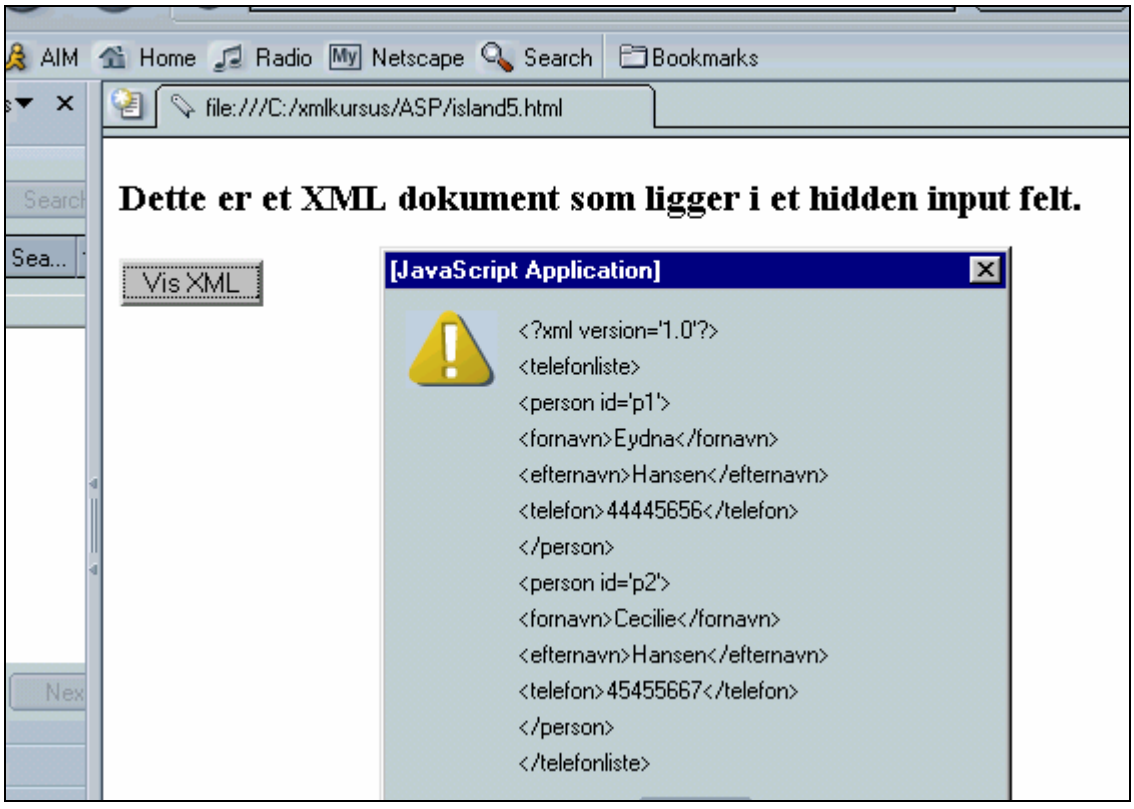
Alternativer til <xml> elementet:

Hvis man ved at alle klienter har installeret Internet Explorer – og **dermed** alle råder over Microsoft XML parseren! - er ovenstående en udmærket løsning! I alle andre tilfælde fungerer den ikke. Vi kan sende XML data på andre måder f. eks. som værdien af et 'hidden' felt i en HTML form. Dette er en noget andet løsning, men den fungerer i praktisk taget alle browsere. Her er vist et eksempel hvor XML også er indlejret skjult i HTML dokumentet. I det nedenstående er kun vist selve HTML dokumentet som serveren sender til klienten:

```
<html>
<body>

<form name="f1" id="f1">
```

```
  <input id="x" name="x" type="hidden" value="
<?xml version='1.0'?>
<telefonliste>
<person id='p1'>
<fornavn>Eydna</fornavn>
<efternavn>Hansen</efternavn>
<telefon>44445656</telefon>
</person>
<person id='p2'>
<fornavn>Cecilie</fornavn>
<efternavn>Hansen</efternavn>
<telefon>45455667</telefon>
</person>
</telefonliste>
" />
  <h3>Dette er et XML dokument som ligger i et hidden input felt.</h3>
  <input type="button" value="Vis XML" onclick="vis()" />
</form>

<script>
function vis(){
   var str=document.f1.x.value;
   alert(str);
}
</script>

</body>
</html>
```

Denne metode fungerer fint i browsere som **Netscape** (vist her) eller **Mozilla**:



Problemet med denne metode er selvfølgeligt at det sendte XML dokument bliver sendt som **ren** tekst og at der først **derefter** skal skrives kode til at **indlæse** denne streng i et **nyt** DOMDocument, som skal instantieres! Og det kan kun gøres ved at man **kender** modtagerens **parser** – **hvis** klienten overhovedet har en XML parser! På et **lokalnet** kender man normalt altid klienternes software og browsere!

Hvis man **ikke** kender klienternes browsere og parsere er det derfor normalt bedst at lave alt arbejdet på serveren og sende klienten resultatet f. eks. som almindeligt HTML som alle browsere forstår!

Anvendelsen af komponenter på siden:

For at give nogle flere funktioner til statiske HTML sider fra en server kan man indsætte komponenter i den side som serveren sender! Der findes bl. a. disse komponenter:

1. **Applets** som er Java klasser som forstås af – stort set – alle browsere og som voldsomt kan forøge mulighederne. En applet kan udgøre en XML **parser**! Applets virker på alle platforme eller operativ systemer!
2. **ActiveX** objekter som er Windows komponenter som f.eks. de forskellige MSXML parser komponenter
3. Kontroller skrevet til platformen .**NET** – som indtil videre stort set kun kører på Windows men også findes til andre operativsystemer.

Ved at lægge en komponent ind i siden bliver serveren helt uafhængig af hvilket software klienten har! Komponenten downloades simpelt hen sammen med HTML siden!

Applets:

Der kan **downloades** mængder af færdige **applets** fra Internettet. En **applet** er et lille program – en Java klasse fil - som stort set kan det samme som et 'rigtigt' program som en C eller Java klasse – **men** er underkastet visse sikkerheds begrænsninger!

En applet downloades fra en fremmed server og kan derfor **ikke** skrive eller læse fra de lokale filer!

En **applet** kan indsættes på en HTML side på denne måde:

```
<html>
<body>
   <applet
     name="xslControl"
     code="com.icl.saxon.XSLTProcessorApplet.class"
     archive="saxon.jar"
     height="0"
     width"0">
     <param name="documentURL" value="telefonliste.xml"/>
     <param name="styleURL" value="flexnumber.xsl"/>
   </applet>
```

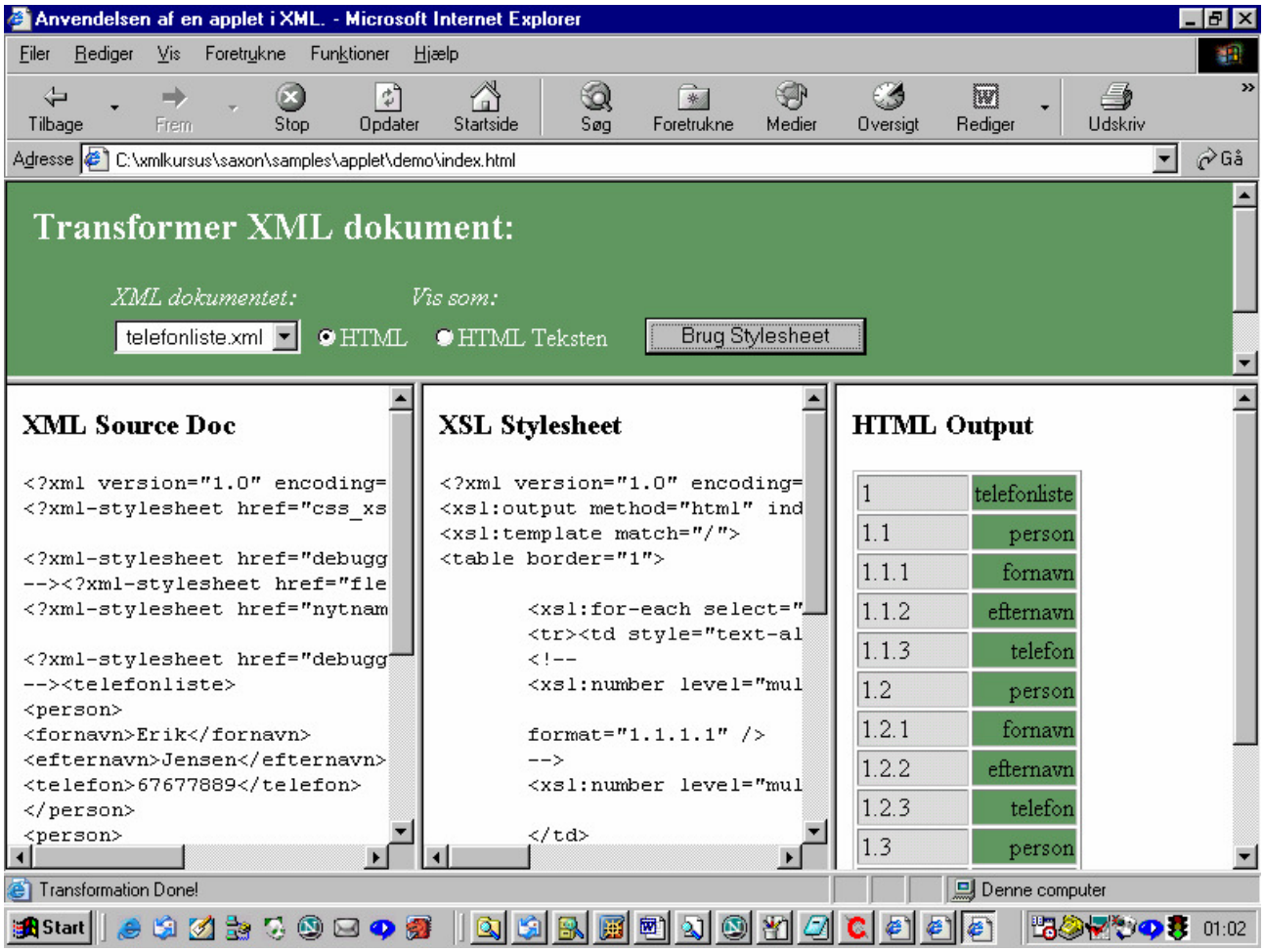
```
  </body>
</html>
```

Vi indsætter altså en komponent på siden. En applet skal som minimum defineres ved en **code** – navnet på klassen her indsætter vi en XSLT processor som komponent! – og evt. en **archive** – der refererer til en Java **jar** fil (en samling af klasser) som definerer klassen. I dette tilfælde henvises som archive til **saxon**.jar som er SAXON processoren – en af de anerkendte XML parsere.

Komponenter kan være synlige eller usynlige! Denne applet er et eksempel på en usynlig komponent – den kan jo ikke ses på siden! Mange applets vil typisk være synlige komponenter f. eks. vise knapper, billeder eller tekstbokse!

Vores applet gives et **navn** som så kan bruges i **script** kode! I dette tilfælde har SAXON processoren kun to **parametre** nemlig et XML dokument og et XSL dokument som bruges til transformation. Disse parametre kan vi så frit sætte i HTML dokumentet! Men denne applet – som bruger saxon.jar - kunne være skrevet helt anderledes!

Et **bruger** interface til en sådan applet kan se således ud:



Vi kan se at man kan **vælge** forskellige XML dokumenter – men i dette tilfælde styles de **altid** med det samme stylesheet – men det kan sagtens ændres. De tre kolonner viser XML, XSL og HTML dokumentet eller eventuelt selve HTML teksten.

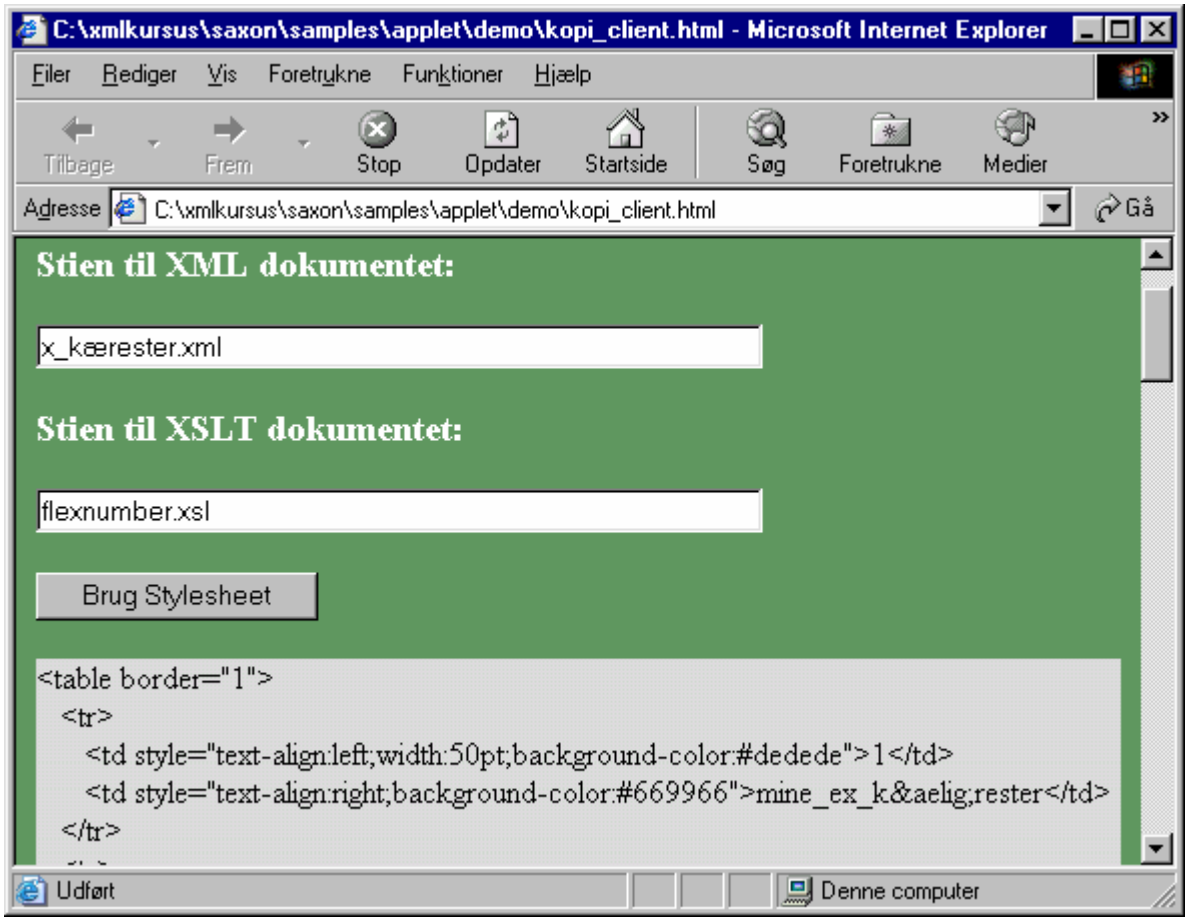
Det er vigtigt at huske at dette skærm billeder IKKE stammer fra komponenten – vores applet – men er HTML kode vi har skrevet **uafhængigt** af komponenten!

Et eksempel på brug af SAXON processoren:

Vi kan skrive **scripts** der anvender denne **applet**. Nedenstående er et eksempel på et simpelt script som gør det. Det er vigtigt at man for hvert kald til processoren (vores applet) kalder metoden **freeCache**() – ellers bryder processoren sammen!:

```
<html>
 <script language="JavaScript">
  function clear()
  {
   document.xsl_komponent.freeCache();
  }

  function transform(){
   clear();
   document.xsl_komponent.setDocumentURL(xmlTransform.xmldoc.value);
   var xml_text=document.xsl_komponent.getSourceTreeAsText();
   var xsl_text=document.xsl_komponent.getStyleTreeAsText();
   var html_text=document.xsl_komponent.getHtmlText();
   vistekst.innerText=html_text;

  }
 </script>
 <body onLoad="clear();" bgcolor="#669966" text="#ffffff">

   <form name="xmlTransform" action="" method="POST">

       <h3> Transformer XML dokument:</h3>
       <h3>Stien til XML dokumentet:</h3>
       <input type="text" id="xmldoc" name="xmldoc" size="50"/>
       <h3>Stien til XSLT dokumentet:</h3>

       <input type="text" id="xsldoc" name="xsldoc" size="50" />

       <br /><br /><input type="button" name="transformButton" value="Brug Stylesheet"
       onClick="transform();">

  </form>
  <div style="color:#030303;background-color:#dedede" id="vistekst"></div>
  <div style="color:#030303;background-color:#fefefe" id="vistekst1"></div>

 <applet
  name="xsl_komponent"
  code="com.icl.saxon.XSLTProcessorApplet.class"
  archive="saxon.jar"
  height="0"
  width"0">
```

```
    <param name="documentURL" value="telefonliste.xml">
    <param name="styleURL" value="flexnumber.xsl">
   </applet>
  </body>
</html>
```



Læg mærke til at processoren **escaper** det danske bogstav 'æ' fordi vi ikke har angivet en **encoding** som f. eks. iso-8859-1 med danske bogstaver! I dette tilfælde transformeres jo til HTML.

Den afgørende **fordel** ved på denne måde at indlejre en applet komponent er at vi er sikker på at klienten har den korrekte XML parser – nemlig den parser som vi sender i vores applet! Dette system virker så altid uanset om klienten sidder ved en Windows, Unix, Linux eller Mac maskine!

ActiveX objekter og COM klasser:

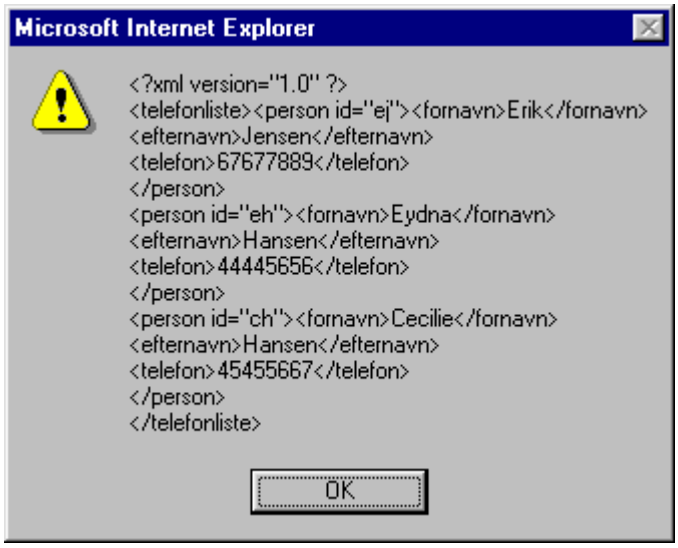
I Windows systemet kan indsættes **ActiveX** komponenter eller Windows klasser på samme måde som **applets**. Disse kan indsættes f. eks. med en <object>. Også applets kan indsættes med en <object> på samme måde - <applet> er blot den traditionelle måde at gøre det på.

Nedenstående eksempel viser hvordan vi kan indsætte en XML parser inden i det HTML dokument som sendes til klienten! På den måde er vi – serveren – sikre på at klienten har den software der skal til for at data behandle XML data!

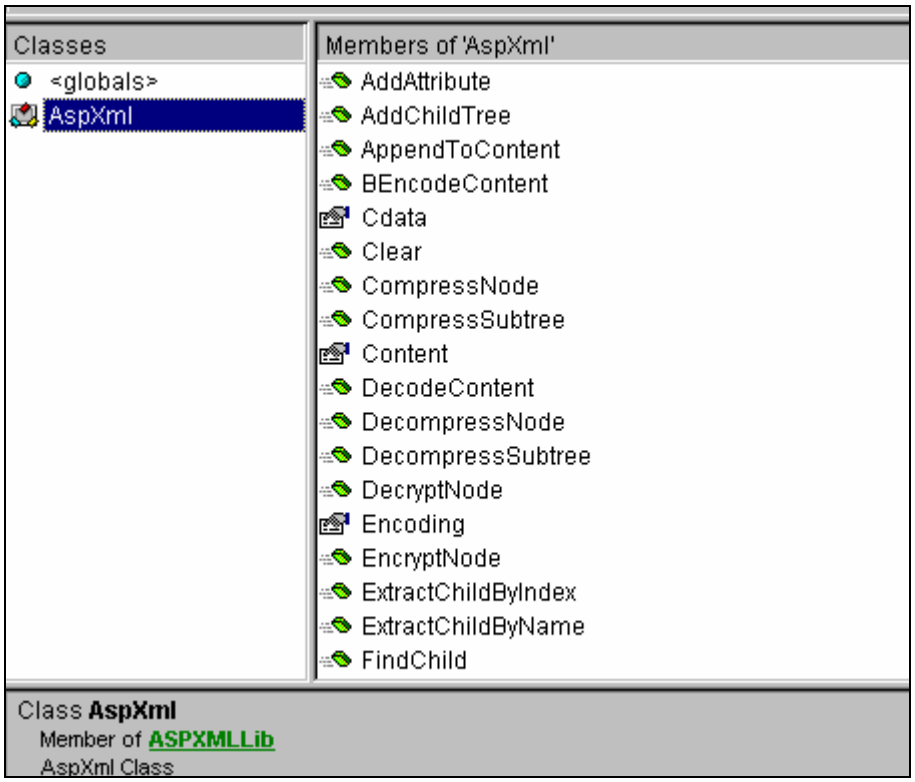
```
<object
id="chilkat"
classid="clsid:10A9E213-DB4B-4A18-93D8-F85EAEEE9B3A"
height="0"
width="0">
Dette er chilkat objektet.
</object>

<script>
chilkat.loadxmlfile("c:/xmlkursus/asp/telefonliste.xml");
alert(chilkat.GetXml());
</script>
```

Også i dette tilfælde indsætter vi en **usynlig** komponent! Som det ses er eksemplet uhyre enkelt. Vi indsætter en ChilkatXml AspXml komponent og derefter kan vi skrive lige så meget **script** som det skal være – for scriptet bruger jo **Chilkat** parseren – **ikke** f. eks. MSXML parseren! Alle klienter – på Windows platformen – vil kunne bruge programmet!

I <object> skal angives et klasse ID for klassen. I Windows findes det i system **registrerings** databasen ved at søge på Chilkat **AspXml**! Alle **registrerede** komponenter får et unikt **hexadecimalt classid** af denne type!



Metoder og properties i komponenten Chilkat AspXml – som her er downloadet fra Internettet – kan findes ved at åbne et Microsoft Office program som f. eks. Access og vælge Funktioner -> Visual Basic. Man kan så sætte en reference til Chilkat komponenten – hvis den ellers er installeret på maskinen. Derefter kan man se alle metoder og properties i Visual Basic editoren i Office programmet!

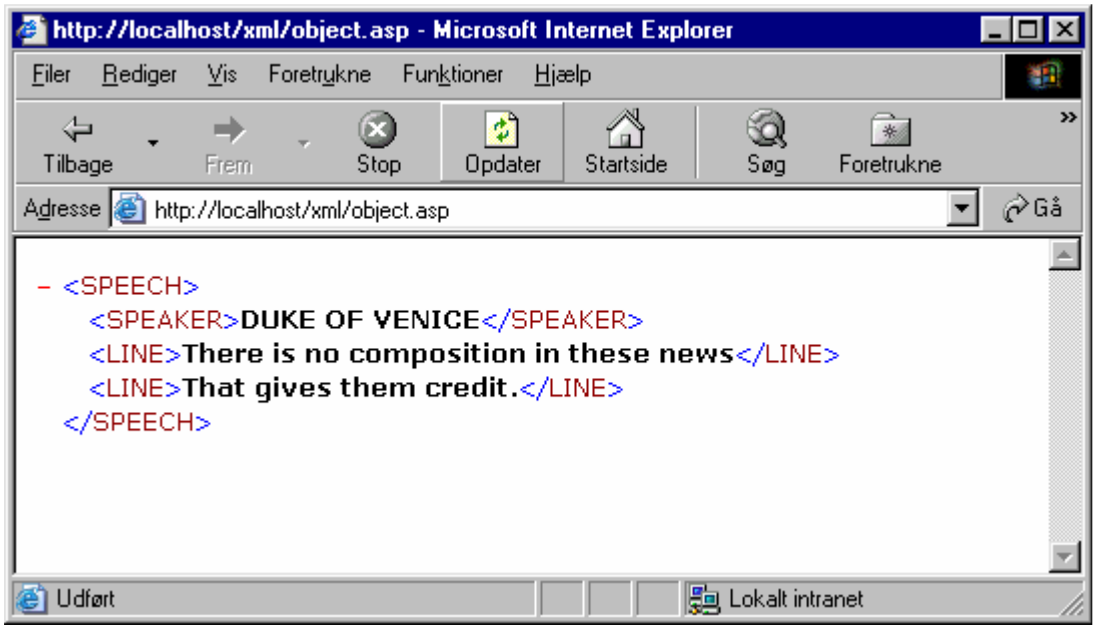


RUNAT Server:

Ovenstående eksempel kan – som regel – skrives lidt mere brugervenligt i det vi bruger en ProgID i stedet for et klasse ID på denne måde:

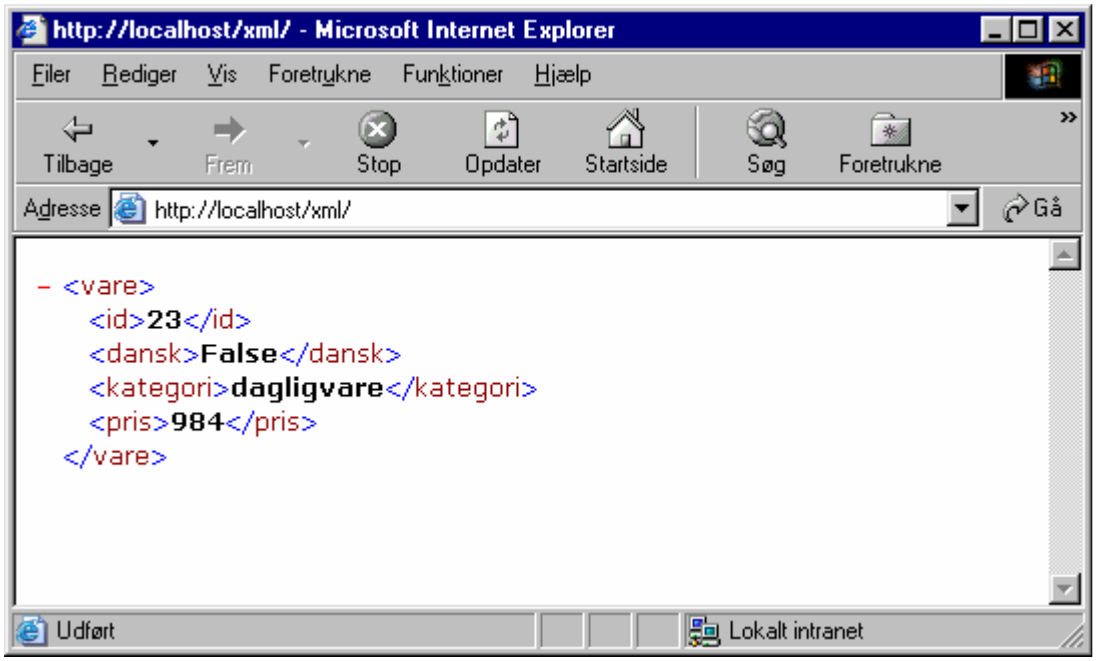
```
<object
ID="doc"
PROGID="AspXml.AspXml"
RUNAT="SERVER"
>
</object>

<%@language="JScript"%>
<%
Response.contentType="text/xml";
doc.loadxmlfile("c:/xmlkursus/asp/scenespeech.xml");
Response.write(doc.GetXml());
%>
```

Dette fungerer på den måde at Chilkat komponenten AspXml.AspXml anvendes af klienten men at den kører på serveren! Fordelen er at serveren på denne måde er sikker på at klienten råder over en XML parser! ASP siden – her http://localhost/xml/object.asp – kan så kaldes af en klient. XML dokumentet er fra Shakespeares Othello:



På samme måde kan vi selvfølgelig indsætte et objekt som refererer til **Microsoft** parseren på denne måde:

```
<object
ID="doc"
PROGID="Msxml2.DOMDocument.4.0"
RUNAT="SERVER"
>
</object>

<%@language="JScript"%>
<%
Response.contentType="text/xml";
doc.load("c:/xmlkursus/asp/varer_ny_transformet.xml");
Response.write(doc.lastChild.childNodes(11).xml);
%>
```



En Web Service eller Internet Tjeneste:

I de sidste år er Web Services blevet det helt store kup på Internettet! En Web Service eller Internet Tjeneste befinder sig på en URL d.v.s. på en **http** adresse. I det følgende eksempel vil vi oprette en Web Service på http://localhost/webservice som et eksempel men den kunne lige så godt ligge på en rigtig server på Internettet eller et lokalt netværk. En Web Service er tilgængelig for alle klienter som befinder sig på det pågældende net – altså i vores eksempel kun på maskinens interne net!

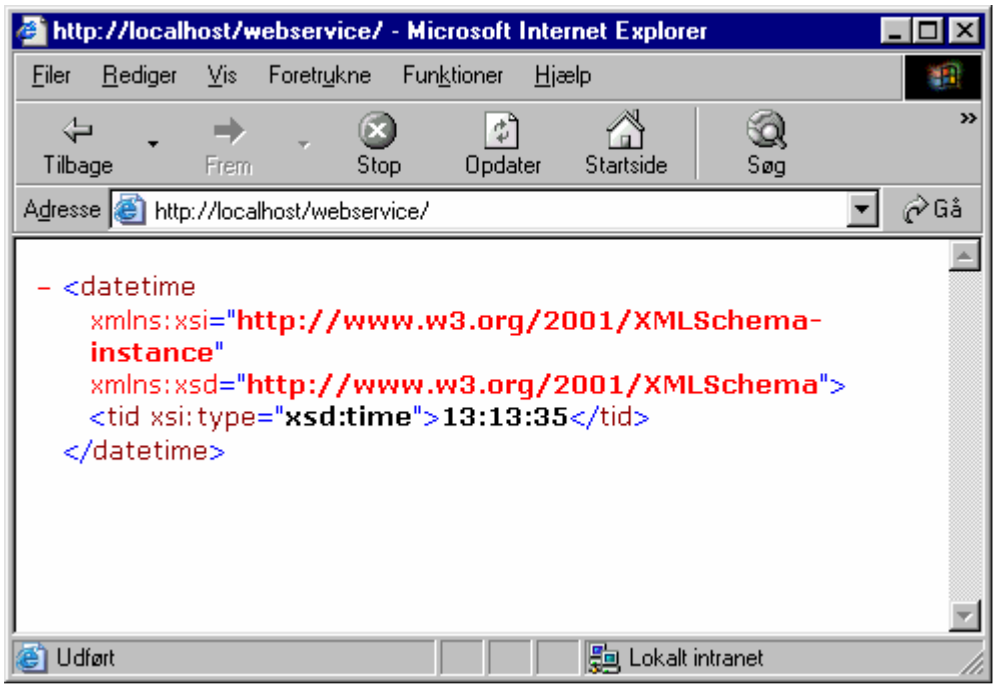
En Web Service svarer ret nøje til **RPC** - Remote Procedure Call - som har en lang glorværdig fortid i UNIX styresystemet! En Web Service bruges til at en komponent eller et program kan kalde op til en anden komponent (et andet program) og få et svar – en tjeneste – tilbage.

Den store forskel i forhold til en almindelig HTML web side er at en Web Service **ikke** har noget bruger interface – den er **blot** en metode som kan kaldes over et netværk. Den er tænkt som en **udvidelse** af de funktioner som allerede findes på min egen maskine. Når jeg (min maskine) kalder en Web Service ser det ud **som om** det er en lokal funktion der kaldes! RPC eller Internet Tjenester skal altså **udviske** skellet mellem forskellige maskiner (eller processer) på et netværk! Dette har altid været ideen i UNIX (og i Linux). Derfor er dette system det centrale i det som i dag kaldes **distribuerede** systemer – distributed processing – altså at en del af databehandlingen simpelt hen foretages på en anden, fjern komputer!

En RPC eller Web Service kan foregå på den samme maskine mellem forskellige processer eller – typisk – mellem forskellige processer på forskellige maskiner.

Vi opretter en ny virtuel mappe ved navn http://localhost/webservice. I denne mappe gemmer vi et default ASP dokument ved navn **default**.asp, som skal definere vores Web Service. Vores Internet

Tjeneste vil kun have een metode, som viser klokkeslettet! Den returnerer altså noget i retning af dette:



En Web Service returnerer **normalt** eller – stort set - altid et **XML** dokument. Ofte bruges **SOAP** (Simple Object Access Protocol) som **formatet** eller protokollen af en Web Service. Den vigtigste grund hertil er at det er ret afgørende at modtager-processen er klar over hvilken **type** der returneres. Man har ikke megen glæde af at alt er af typen tekst eller **string**!

SOAP bruger her **XSD** typerne – som er de **eneste** typer som er bredt anerkendt og som forstås af alle operativ systemer! En offentlig Web Service kaldes jo af alle typer af maskiner gennem en HTTP adresse! Den skal derfor fungere helt generelt!

I vores eksempel har vi ikke valgt at bruge SOAP men et simplere format der dog anvender **XSD** typerne. Vores default.asp ser således ud:

```
<%@language="JScript" %>
<%
var d=new Date();

var h=d.getHours();
if(h<10)h='0'+h;
var m=d.getMinutes();
if(m<10)m='0'+m;
var s=d.getSeconds();
if(s<10)s='0'+s;

var doc=Server.createObject("Msxml2.DOMDocument.4.0");
var str='<datetime xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"';
str+=' xmlns:xsd="http://www.w3.org/2001/XMLSchema"> ';
str+=' <tid xsi:type="xsd:time">'+h+':'+m+':'+s+'</tid></datetime>';
doc.loadXML(str);
Response.ContentType="text/xml";
```

```
doc.save(Response);

%>
```

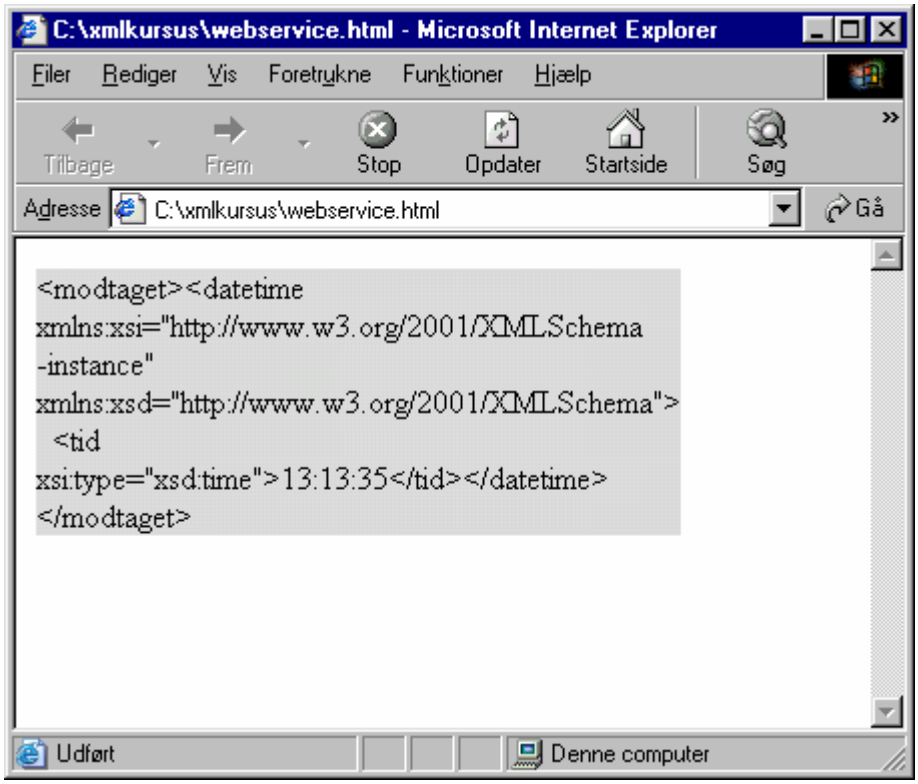
Default.asp opbygger et XML dokument og sender det retur til klienten. Som det kan ses skal XSD typen xsd:time have et **ganske** bestemt format – ellers kan dokumentet ikke valideres – og så bliver det ikke produceret! Hvis timen f. eks. er under 10 - altså klokken 8 om morgenen - er vi nødt til at indsætte et '0' foran time tallet! Fordelen ved at anvende de to **namespaces** – xsi og xsd namespacet - er at vi sikrer os at dokumentet er **gyldigt** – **også** at klokkeslettet har et gyldigt format! Dette er forudsætningen for at det kan bruges på **alle** maskiner og **alle** operativ systemer!

Vi kan kalde denne Web Service f. eks. på følgende måde hvor en klient opretter en **HTTP** forbindelse til tjenesten:

```
<body>
<div id="x" style="width:100pt;background-color:#dedede"></div>
</body>

<script>
try {
var doc=new ActiveXObject("Msxml2.DOMDocument.4.0");
var http=new ActiveXObject("Msxml2.XMLHTTP.4.0");
http.open("GET","http://localhost/webservice",false);
http.send();
doc.loadXML("<modtaget>"+http.responseXML.xml+"</modtaget>");
alert(doc.xml);
document.all("x").innerText=doc.xml;
}
catch(e) {alert(e.description);}
</script>
```

Som det ses kan klienten bruge **retur** dokumentet – http.responseXML - til at indgå i et **nyt** dokument! Web Tjenester bruges ofte på denne måde – en del af databehandlingen **forlægges** til en anden maskine (en slags out-sourcing)!

Vi kunne på denne måde også lade klienten sende visse **parametre** til tjenesten som så returnerede data f. eks. fra en database.



En SOAP Web Service:

På samme måde kan vi implementere en **SOAP** Web Service eller server. SOAP standarden der er defineret i forskellige udgaver af **W3C** er en ret **fri** standard hvor den konkrete form er overladt til de enkelte applikationer. Normalt anvendes dog **XSD** typerne for at garantere at en hvilken som helst platform kan modtage og forstå det returnerede svar.

SOAP defineres på denne måde af W3C Konsortiet:

SOAP Version 1.2 is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.

Et SOAP dokument er grundlæggende blot en 'one way message' fra sender til modtager – men kan udnyttes til at etablere **samtaler** frem og tilbage. Eventuelt kan mange servere virke sammen i et neværk der sender SOAP beskeder frem og tilbage. Budskabet eller beskeden er et XML Infoset.

SOAP beskeder sendes ofte via HTTP protokollen – men kan sendes via mange forskellige slags protokoller. Også som **emails** vis **mailto** protokollen!

Vi kan forestille os en Web Service som kan oplyse om **fly** afgange og hvor der kan **bookes** fly billetter til disse afgange! Grundlæggende kaldes denne Web Service så af en **applikation** som igen f. eks. betjener en **klient**. En fly server kalder op til vores Web Tjeneste når den får en anmodning fra en privat klient.

Vi vil implementere vores Web Service med en simpel **GET** metode. Denne tjeneste kan altså f. eks. kaldes på denne måde:

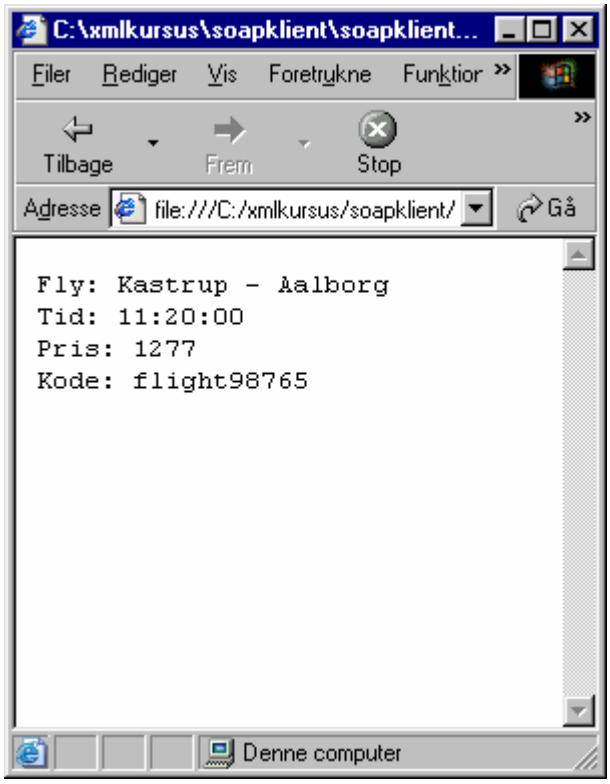
Billund-Kastrup

Vi har lagt en Web Service på en virtuel mappe ved navn **soap** under http://localhost. Denne tjeneste skal kaldes med **nøjagtigt** to parametre nemlig til og fra. Som nævnt tidligere er det imidlertid **ikke** meningen at en privat klient skal kalde op til denne tjeneste. I stedet kan vi skrive en slags bruger fly server på følgende måde. Egentligt skulle denne server sende f. eks. en email tilbage til en klient. For at illudere det lader vi programmet skrive en ny side med visse data:

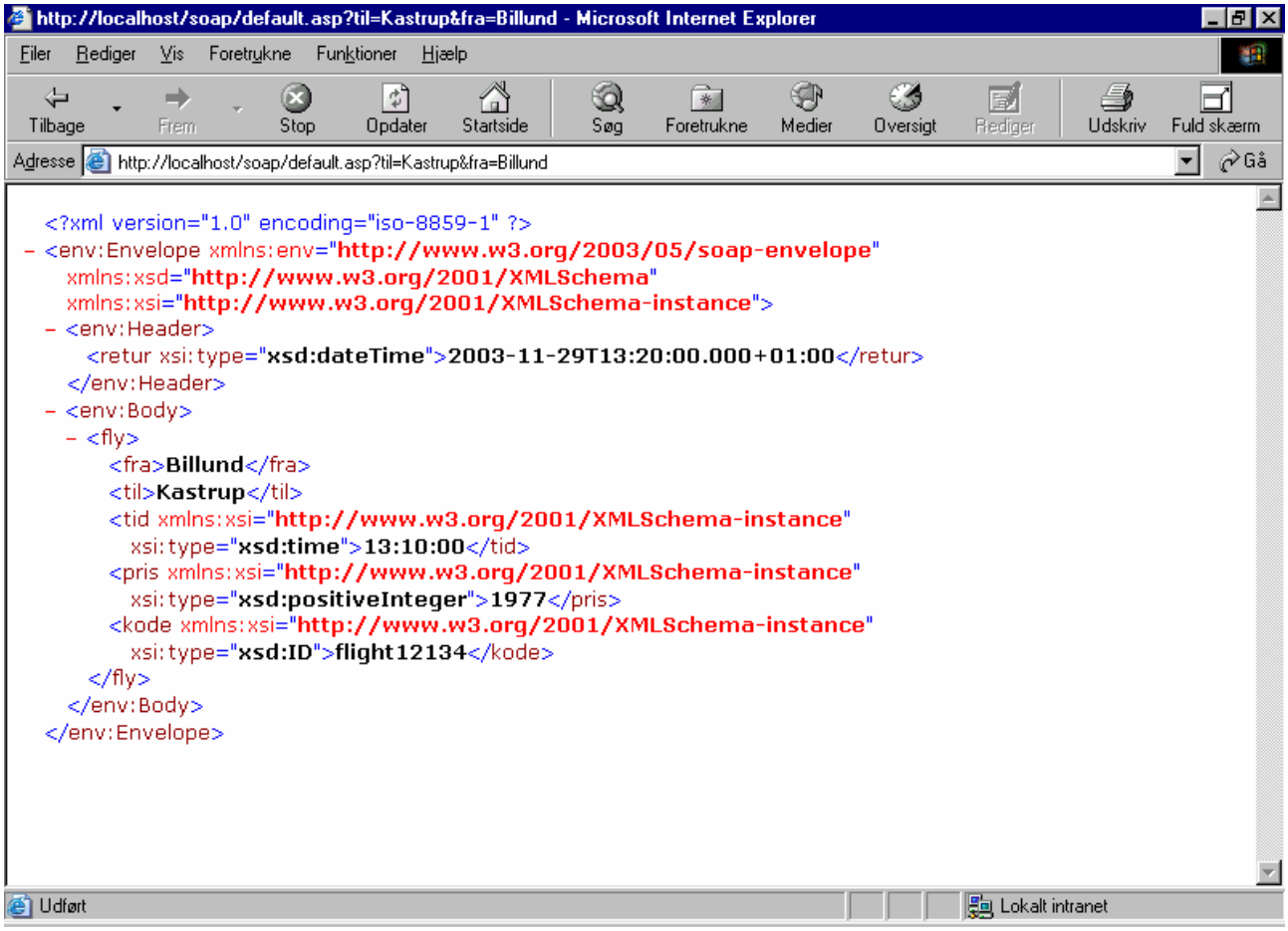
```
<head>
<script>
   function check(){
   try {
      var doc=new ActiveXObject("Msxml2.DOMDocument.4.0");
      var http=new ActiveXObject("Msxml2.XMLHTTP.4.0");
      var fra=f.fra.value;
      var til=f.til.value;
      http.open("GET", "http://localhost/soap/default.asp?til="+til+"&fra="+fra+"", false);
      http.send();
      doc.loadXML(http.responseText);
      var tid=doc.selectSingleNode("//tid").firstChild.nodeValue;
      var pris=doc.selectSingleNode("//pris").firstChild.nodeValue;
      var kode=doc.selectSingleNode("//kode").firstChild.nodeValue;
      var fra=doc.selectSingleNode("//fra").firstChild.nodeValue;
      var til=doc.selectSingleNode("//til").firstChild.nodeValue;

      var str="Fly: "+fra+" - "+til+"\nTid: "+tid+"\nPris: "+pris+"\nKode: "+kode;
      document.write("<pre>"+str+"</pre>");
   }
      catch(e) {alert(e.description);}
   }
</script>
</head>

<body>
<form id="f" name="f">
   <h3>Til:</h3>
   <input name="til" value="Aarhus" />
   <h3>Fra:</h3>
   <input name="fra" value="Kastrup" />
   <br/>
   <input type="submit" onclick="check()" value="Send"/>
</form>
</body>
```

Vi har også – for at kunne teste systemet – lavet et bruger interface på denne måde – selv om dette egentligt ikke er meningen med en Web Service:

Når vi aktiverer vores Web Service skriver dette program så en ny HTML side:



I virkeligheden modtages dette **SOAP** dokument fra serveren i http://localhost/soap:

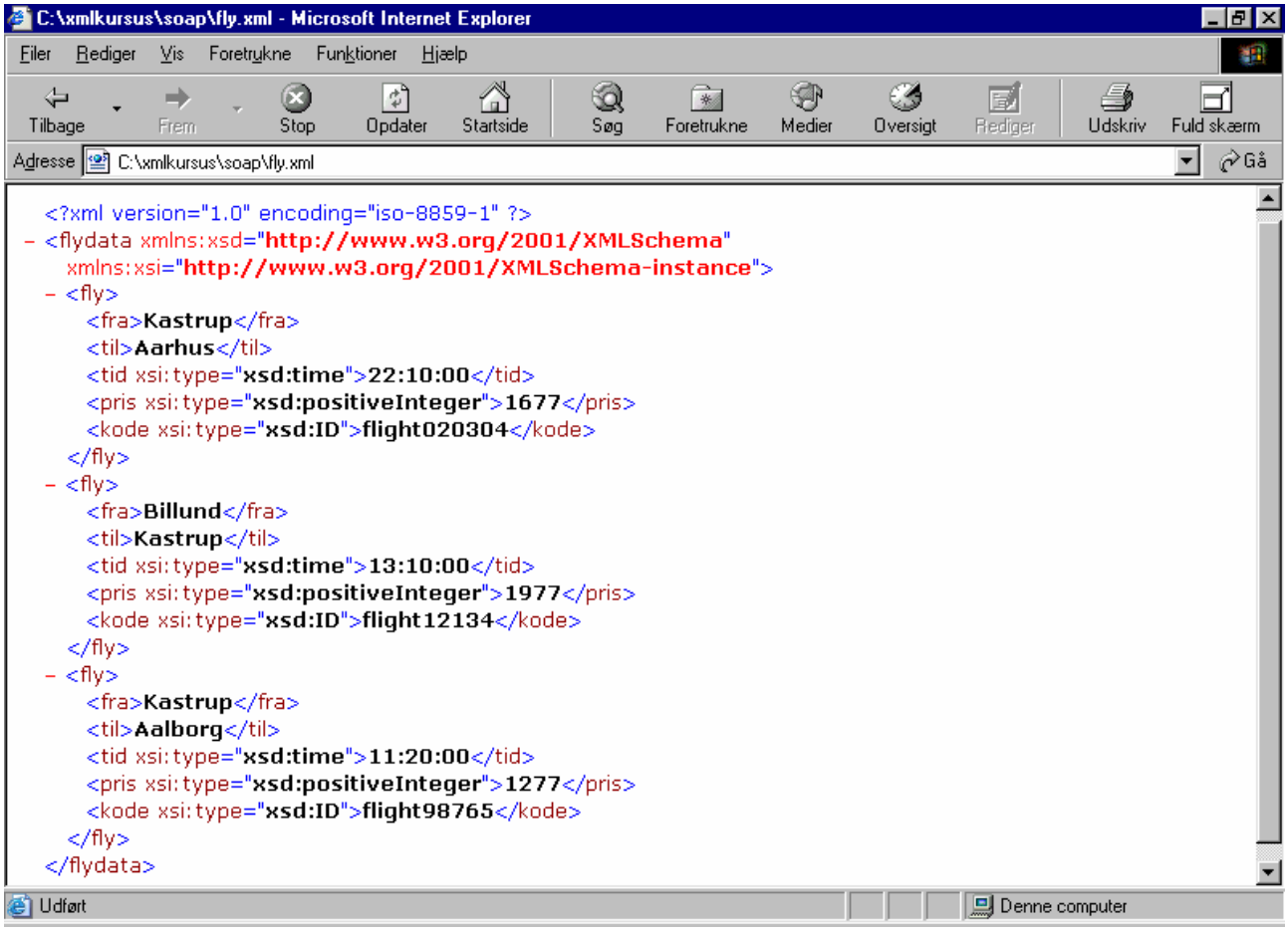


Vi kan se at serveren – vores Web Service – her returnerer et SOAP dokument – en såkaldt SOAP **Envelope**. En Envelope er en SOAP **pakke** eller message. Envelope er altid roden i et SOAP dokument.

Alle de returnerede data er **typedefinerede** med **XSD** – undtaget til og fra data! Et sådant SOAP dokument kan modtages og bruges af **alle** maskiner – Windows, Unix, Linux eller Mac!

I dette tilfælde har vi valgt den simple løsning at SOAP serveren simpelt hen slår op i et XML dokument og finder data ud fra den **request** den modtager. En mere realistisk løsning ville være at Web tjenesten slog op i en **database**!

Vores XML dokument kaldet **fly**.xml – som altså kun er et illustrerende eksempel - ser således ud:



Selve web tjenesten kan så skrives på denne måde som **default**.asp i mappen http://localhost/soap:

```
<%@ language="JScript"%>
<%
til=Request("til");
fra=Request("fra");

var doc = new ActiveXObject("Msxml2.DOMDocument.4.0");
doc.load(Server.MapPath("fly.xml"));

Response.ContentType="text/xml";
var fly="//fra[.='"+fra+"']/../til[.='"+til+"']/..";

Response.Write("<?xml version='1.0' encoding='iso-8859-1'?>");

Response.Write("<env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope' ");
Response.Write("xmlns:xsd='http://www.w3.org/2001/XMLSchema'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>");

  Response.Write("<env:Header><retur xsi:type='xsd:dateTime'>2003-11-
29T13:20:00.000+01:00</retur></env:Header>");
   Response.Write("<env:Body>");
      Response.Write(doc.lastChild.selectSingleNode(fly).xml);
   Response.Write("</env:Body>");
Response.Write("</env:Envelope>");
%>
```

Som det ses skriver programmet en SOAP **Header** og en **Body**. Formatet på disse er i høj grad overladt til den konkrete applikation. **Body** består her simpelt hen af en søgning i dokumentet **fly**.xml ud fra de to parametre.

Header kan f. eks. sende en datering eller meddelelser om evt. fejl eller problemer. Her sender web tjenesten en hard kodet dato retur. I Header sendes meta data eller kontrol data. En Header og en Body kan ændres hen ad vejen hvis SOAP dokumentet sendes gennem flere led (flere SOAP servere) inden det når den egentlige modtager! En Header kan have en attribut mustUnderstand=”true” som bevirker at den skal databehandles af en såkaldt **intermediary** server! Ofte er SOAP dokumentet et produkt af flere servere på række.

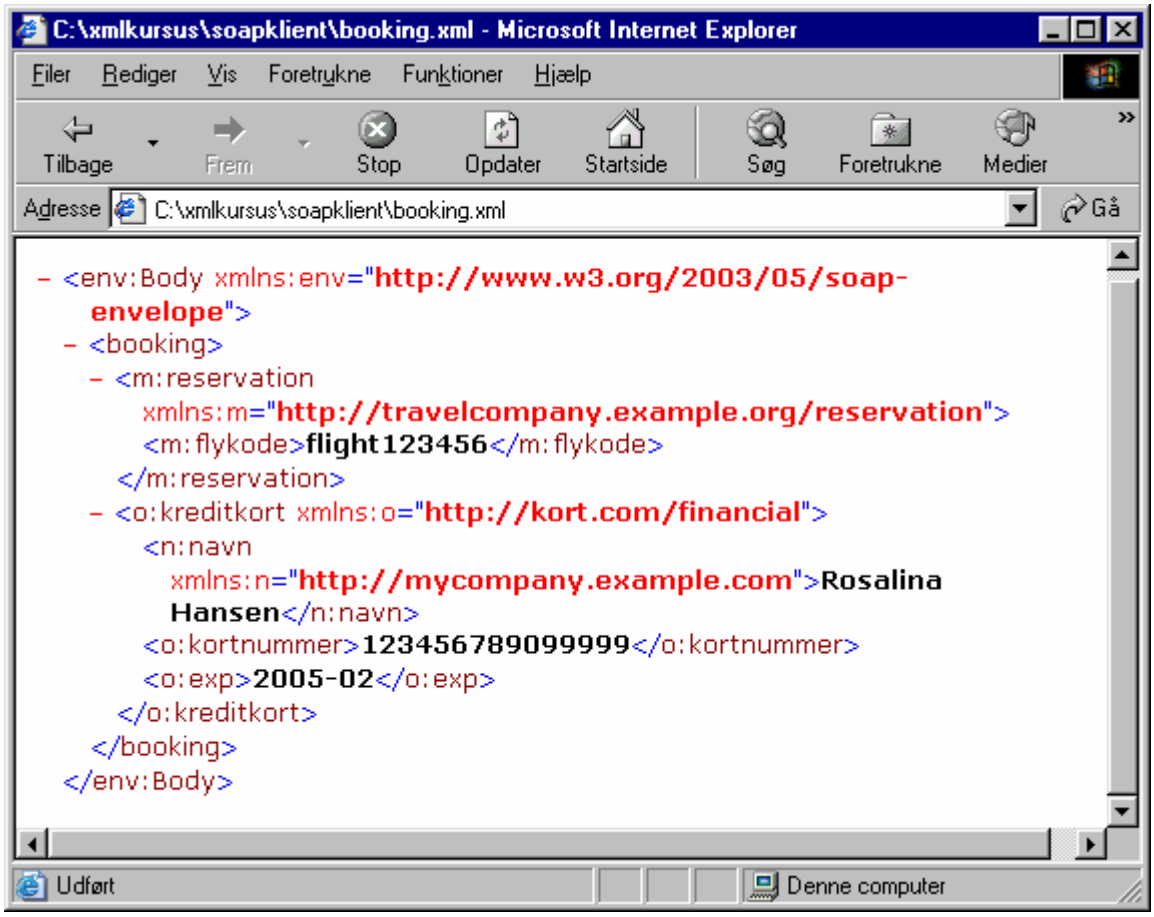
Hvis ovenstående skulle fungere lidt mere realistisk skulle XML dokumentet selvfølgeligt IKKE indlæses for hvert opkald fra en klient! Serveren skulle ligge parat og have indlæst dokumentet på forhånd!

Dette kan gøres ved at indlæse XML dokumentet i en global variabel Application(”doc”) eller i en særlig fil global.asa.

Ovenstående Web tjeneste har **kun** een service - nemlig en informations tjeneste! Dette kan dog let **udbygges** så serveren f. eks. kan modtage bestillinger på fly rejser, reservationer og betalinger!

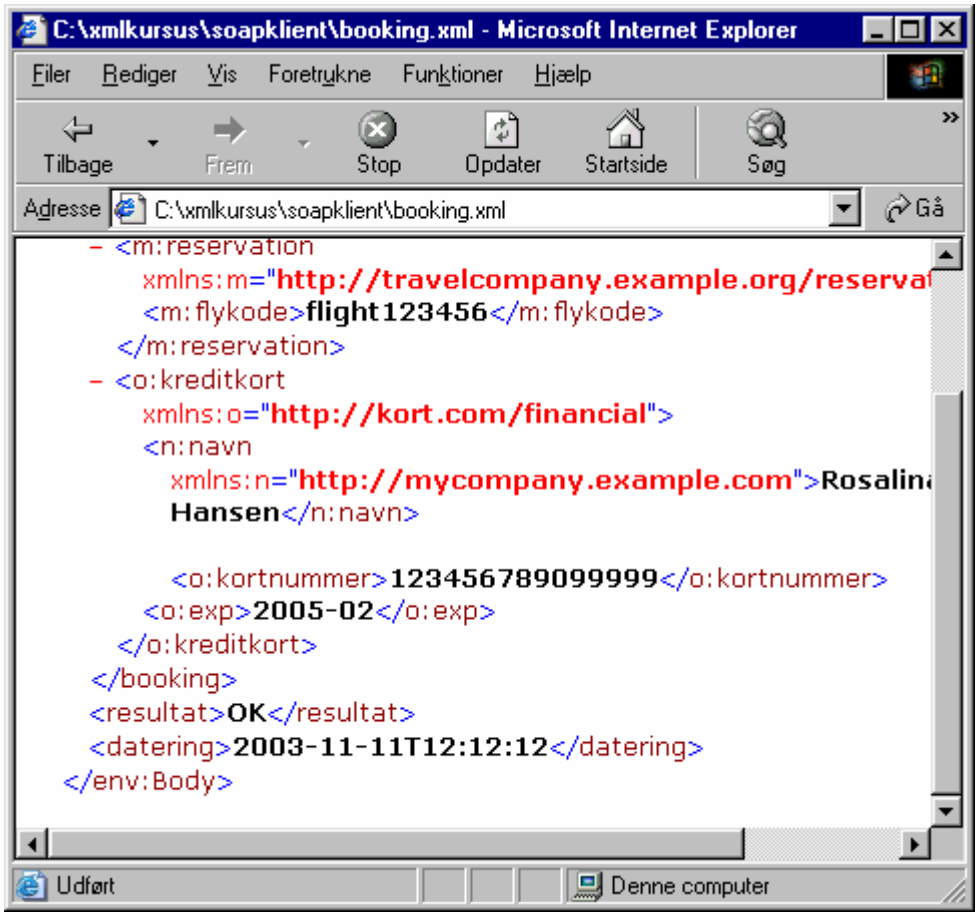
Denne Web Service kaldes op med en HTTP **GET** kommando. Systemet kunne også f. eks. implementeres så at serveren skulle kaldes op med et **XML** dokument – et **SOAP** dokument – således at den **både** modtager **og** returnerer SOAP dokumenter! Ofte er SOAP tjenester dog implementeret ved at de modtager opkald over HTTP protokollen med GET eller POST.

Følgende kan være et eksempel på en besked der indeholder en **betaling** eller **booking** ud fra en tidligere opnået kode for en flyrejse:



Dette eksempel viser hvordan Web Tjenesten modtager et XML eller SOAP dokument som input – ikke blot nogle request parametre via HTTP!

Web Tjenesten kan så siden returnere et SOAP svar – en **bekræftelse** - ud fra det dokument som det har modtaget!:


C:\xmlkursus\soapklient\booking.xml - Microsoft Internet Explorer
Filer Rediger Vis Foretrukne Funktioner Hjælp
Tilbage Frem Stop Opdater Startside Søg
Adresse C:\xmlkursus\soapklient\booking.xml
Gå
- <m:reservation
xmlns:m="http://travelcompany.example.org/reserva
<m:flykode>flight123456</m:flykode>
</m:reservation>
- <o:kreditkort
xmlns:o="http://kort.com/financial">
<n:navn
xmlns:n="http://mycompany.example.com">Rosalin
Hansen</n:navn>
<o:kortnummer>123456789099999</o:kortnummer>
<o:exp>2005-02</o:exp>
</o:kreditkort>
</booking>
<resultat>OK</resultat>
<datering>2003-11-11T12:12:12</datering>
</env:Body>
Udført
Denne computer