Ruby Web-Nuby - Intro til webapplikasjoner i Ruby.

Kent Dahl < `kentda@pvv.org` >

v0.31, Mars 25., 2003

En introduksjon til hvordan man skriver webapplikasjoner i Ruby.

1 Dagens agenda

Planen er å gå igjennom en del ymse emner som er aktuelle med hensyn på programmering av webapplikasjoner. Deretter beveger vi oss til en datasal hvor dere får mulighet til å øve og benytte det vi har gått igjennom.

Forutsetninger:
Det forutsettes kunnskap om programmering, HTML og databaser, samt litt kjennskap til Ruby.

Notasjon:
Eksemplene vises med linjenummer, som *ikke* er en del av kildekoden. Det vil si, når det står
`12| puts "Hello"` er det `puts "Hello"` som er kildekoden.
Kodelinjer er ofte annotert med en kommentar, `#=>`, som viser hva resultatet vil bli eller hva som vil skrives ut. For eksempel: `5 + 2 #=> 7`

2 CGI

Det å skrive CGI-skript (Common Gateway Interface) i Ruby skiller seg ikke nevneverdig fra å skrive CGI-skript i andre språk. Skriv ut HTTP-hodelinjer (headers), hent inn CGI-variabler, generer HTML (eller noe annet interessant innhold) dynamisk og spyl det av gårde til klienten.

[code/cgi_hei.rb]

```
#!/store/bin/ruby
# ^^^ - linje for å angi hvor man finner Ruby-fortolkeren

# Print ut HTTP-hodelinjer for å angi at vi serverer HTML.
print "Content-type: text/html\r\n\r\n"

# Server litt HTML.
print "<html><body><h1>Hei verden!</h1></body></html>"
```

[1] http://opencontent.org/openpub/
[2] http://www.pvv.org/~kentda/ruby/webnuby/

Øverste linje angir hvor Ruby-fortolkeren befinner seg på serveren. `/store/bin/ruby` er bare en vanlig plassering på NTNU maskiner som bruker `store`. Prøv å skriv '`which ruby`' om du lurer på hvor fortolkeren er på den maskinen du er på nå.

Men dette var ikke særlig spennende uten noen mulighet for å påvirke resultatet dynamisk...

2.1 Variabler

... så la oss lage en liten web-basert kalkulator.

[code/cgi_regne.rb]

```
#!/store/bin/ruby
print "Content-type: text/html\r\n\r\n"
# Starter HTML-dokumentet.
print '<html><body>'

# Hent inn CGI-biblioteket.
require 'cgi'
# Lag en instans slik at vi får tak i CGI-variablene.
cgi = CGI.new

# Hent CGI-variablene
x = cgi['x'][0] # Oppslaget returnerer en Array,
y = cgi['y'][0] # så vi plukker ut første element.

# Skriv dem bare ut dersom vi fikk noe.
if (x and x.size.nonzero?) and
   (y and y.size.nonzero?) then
  a = x.to_i # Gjør om til heltall.
  b = y.to_i
  print "#{a} multiplisert med #{b} er ", a*b
end

# Skriv ut et lite skjema.
print '<form>'
print '<input name="x" type="text"> *'
print '<input name="y" type="text"> = '
print '<input type="submit" value="gange">'
print '</form>'

# Avslutter HTML-dokumentet.
print '</body></html>'
```

2.2 HTML

CGI-biblioteket kan også hjelpe deg litt med å skrive HTML-koden.

```
[ code/cgi_pent.rb ]
#!/store/bin/ruby
require 'cgi'

# CGI-biblioteket kan også hjelpe deg med å få skrevet HTML-koden.
cgi = CGI.new('html4')
cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { 'Penere kildekode?' }
    } +
    cgi.body {
      cgi.h1 { 'Gjør dette kildekoden penere?' } +
      cgi.p +
      'Eller får man krøllparentes-overdose?'
    }
  }
}
```

Du slipper å huske hvilke element som skal avsluttes og i hvilken rekkefølge, men får noe som ligner litt for mye på Lisp.

2.3 Cookies

Cookies er en måte å lagre små datamengder på maskinen til brukeren som surfer inn på sidene våre, som vi kan hente ut igjen når de kommer tilbake en annen gang.

```
[ code/cookies.rb ]
#!/store/bin/ruby
require 'cgi'
c = CGI.new('html4')

# Hent ut den gamle kaken.
gammel_cookie = c.cookies['rubywebnuby']

# Kurstekst i første element, antall besøk i andre element.
kurs_tekst, antall_tekst = gammel_cookie
antall_besok = if antall_tekst then
                 antall_tekst.to_i + 1
               else 0 end

# Lag ny kake.
ny_cookie = CGI::Cookie.new('rubywebnuby',    # Kakenavn.
  'Ruby Web Nuby', antall_besok.to_s          # Verdier.
)
```



```

c.out( 'cookie' => [ny_cookie] ) do # Sett kaken via HTTP.
  c.html do
    c.body do
      # Print ut de forrige kakene
      gammel_cookie.join(c.br) +
        "<P>Du har vært her #{antall_besok} ganger før, " +
        "da i forbindelse med #{kurs_tekst} kurs."
    end
  end
end
```

Cookies er et kjekt verktøy når man ikke har noen database å lagre i, dataene er små eller brukerne er overmåte allergisk mot innloggingsskjermer. Ellers er det kanskje en ide å lagre ting på serversiden og/eller i skjulte input-elementer.

2.4 Sesjoner

Sesjoner er et overbygg over cookies, som lagrer en liten bit data i en cookie hos klienten, og (potensielt) et tonn med data på serveren.

```
[ code/sesjon.rb ]
#!/store/bin/ruby
require 'cgi'
require 'cgi/session'
c = CGI.new('html4')
sesjon = CGI::Session.new( c,
  'session_key' => 'rubywebnuby2',
  'prefix' => 'ruby_sesjon.')

# Hent ut antall besøk fra sesjonen
antall_tekst = sesjon['AntallBesok']
antall_besok = (antall_tekst ? antall_tekst.to_i+1 : 0)

# Sett det nye antallet tilbake i sesjonen
sesjon['AntallBesok'] = antall_besok

c.out do
  c.html do
    c.body do
     "Du har vært her #{sesjon['AntallBesok']} ganger før."
    end
  end
end
```

Da har vi iallfall fått lagret en god del mer på serversiden, og høyde, alder, øyenfarge, samt en million "theme"-innstillinger for webapplikasjonen vår må ikke sendes over nettet hver gang en side hentes.

2.5 Ytelse

Til nå har vi bare kjørt disse CGI-skriptene på den trauste, trege måten. For hvert kall til sidene, må Ruby-fortolkeren startes opp og biblioteker lastes. Dette er ufattelig ineffektivt.

Vi trenger å ha minimum Ruby-fortolkeren, og helst også biblioteker, ferdig lastet og klare til dyst når en HTTP-forespørsel kommer inn. Det er flere muligheter, blant annet:

mod_ruby[3], *WEBrick*[4], *Radical*[5], fastcgi, *IOWA*[6], *Borges*[7], httpd, httpserv, wwwd, wwwsrv ... etc

3 Ruby i Apache: mod_ruby

Ruby kan integreres i *Apache-webserveren*[8] ved hjelp av *mod_ruby* . Vi går ikke inn på konfigurering og installering, men fokuserer på den praktiske bruken av mod_ruby.

mod_ruby kan brukes til så mangt, da det egentlig bare:

- lenker Ruby-fortolkeren inn i Apache dynamisk
- slenger gitte HTTP-forespørsler til Ruby-fortolkeren

En vanlig bruk er å sette opp slik at Ruby CGI-skript kjøres i Ruby-fortolkeren som er inne i Apache-prosessen i stedet. I tillegg til at det går fortere, har man da også tilgang til deler av Apaches API.

[code/modruby1.rb]

```
r = Apache.request # Hent den gjeldende forespørsel.
gammel_innholdstype = r.content_type
r.content_type = 'text/html'
r.sync=true # Slå på synkron utskrift.
puts '<HTML><BODY>'
puts '<H1>Grave litt rundt i mod_ruby APIen.</H1>'
puts '<P>Gammel innholdstype: ' + gammel_innholdstype
puts '<P>Server versjon: ' + Apache.server_version
puts '<UL>'
sleep 3 # Bare for å vise at synkron utskrift er påslått.
[ :filename,     :protocol,     :request_method,
  :request_time, :server_name,  :server_port,
  :status,       :uri,
].each do |symbol|
  print '<LI>', symbol.to_s, ' = ', r.send(symbol), '</LI>'
end
puts '</UL></BODY></HTML>'
```

(NB: Det er vanlig å bruke `.rbx`-prefikset når koden skal kjøres direkte i mod_ruby.)

[3] http://www.modruby.net
[4] http://www.webrick.org/
[5] http://idanso.dyndns.org:8081/
[6] http://sourceforge.net/projects/iowa
[7] http://www.segment7.net/ruby-code/borges/borges.html
[8] http://httpd.apache.org/

3.1 Apache-prosesser og mod_ruby

For de som er kjent med Apache er dette kanskje en selvfølge, men det er viktig å innse at Apache-webserveren kjører flere prosesser som mottar og betjener HTTP-forespørslene til en webside.

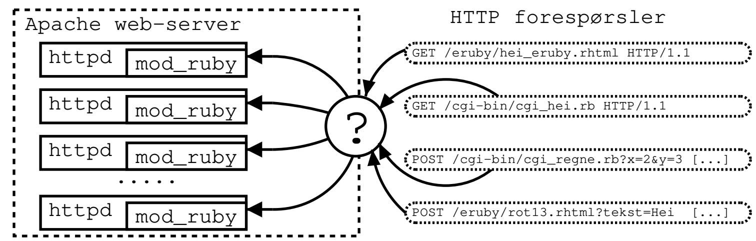

Figure 1: Flere Apache-prosesser.

En påfølgende forespørsel fra en bruker vil ikke nødvendigvis havne hos samme prosess som den forrige forespørselen gjorde. Dermed kan vi ikke stole på at tilstanden vi forlot den interne Ruby-fortolkeren i er den vi møter igjen. (Så tro ikke at globale variable kan brukes for å lagre tilstand mellom forespørsler.)

3.2 Ruby inni HTML: eruby

eruby står for Embedded Ruby og gjør slik at man kan programmere dynamiske websider i Ruby på samme måte som man gjør med f.eks. PHP, JSP eller ASP. "Embedded" betyr her "innbakt" i annen tekst.

[code/hei_eruby.rhtml]

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head><title>Hei, eruby!</title></head>
  <body>
    <!-- utskrift i kode settes inn i HTML-dokumentet. -->
    <% puts "Hei, eruby!" %>

    <!-- Med = tegnet kan man også sette inn variabler direkte. -->
    <% dato = Time.now %>
    <%= dato %>

    <!-- Også har man kommentarer -->
    <%# puts "Denne koden kjøres ikke." %>
  </body>
</html>
```

Det finnes andre alternativer for å generere HTML og lignende fra templater, f.eks. *Amrita* , som tilbyr noe 'renere' separering av HTML og Ruby-kode.

3.3 Anonyme moduler i eruby

Dersom du bare skal kjøre en webapplikasjon på en webserver, trenger du ikke bekymre deg om forsøpling av navnerommene til fortolker-instansene i Apache-prosessene. Du kan bare passe på selv at du ikke roter til og lager to metoder med samme navn som gjør litt forskjellige ting. (Lykke til på sinnssykehuset.)

Skal man derimot ha flere webapplikasjoner, er det ønskelig at man beskyttes litt mot navneforurensing. Alle fortolker-instansene deles jo mellom alle webapplikasjoner som kjøres. Derfor utfører eruby koden som skal kjøres for en dynamisk generert HTML-side innpakket i en anonym modul.

Følgende kode fungerer derfor ikke:

```
[ code/eruby_anonym.rhtml ]
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head><title> ikke-fungerende rot 13 </title></head>
  <body>
<%
class String  # Dette lager en ny String-klasse i
  def rot13   # stedet for å utvide den eksisterende.
    self.tr( "A-Za-z", "N-ZA-Mn-za-m" )
  end
end
require 'cgi'
c = CGI.new
inntekst = c['inntekst'][0]
if inntekst then
  uttekst = inntekst.rot13  # Fungerer ikke.
end
%>
  <b><%=inntekst%></b> rot 13 kryptert blir <b><%=uttekst%></b>.
  <form><input type="text" name="inntekst">
        <input type="submit" value="rot13"></form>
 </body>
</html>
```

Ruby er selvsagt dynamisk nok til å komme seg rundt slike stengsler...

```
[ code/eruby_rot13.rhtml ]
# Med litt stygg eval- og klassemagi går det derimot greit.
String.class_eval {
  def rot13
    self.tr( "A-Za-z", "N-ZA-Mn-za-m" )
  end
}
```

...men dette vil påvirke alle prosesser som noensinne kjører skriptet, og den forrige ikke-fungerende versjonen vil enten feile eller fungere alt etter hvilken tilstand prosessen som blir valgt ut til å håndtere forespørselen er i.

4 Sikkerhet

CGI-skript og webapplikasjoner har sitt grensesnitt ut mot hele verden; en samling som ikke nødvendigvis er like tillitsverdig. Ansvaret er ene og alene vårt for å gjøre våre programmer motstandsdyktige mot angrep, men Ruby tilbyr noen hjelpemidler.

4.1 Tainting - besudlede data

En av de store sikkerhetsfarene er eksterne data: Alt av informasjon som kommer fra brukeren kan potensielt være livsfarlig, skrevet med bare den hensikt å ødelegge dagen din.

Ruby tilbyr en mekanisme som merker alle data fra eksterne kilder: `tainting`. Det antyder hvorvidt data er "besudlet" eller "smittebærende", slik at du ikke kan stole på dem lenger enn du kan kaste dem.

```
[ code/taint1.rb ]
print 'Skriv noe inn og trykk Return: '

# Brukerdata er svart som kull...
skitten = gets
puts 'Skitten brukerdata.' if skitten.tainted?

pur = 'Literal data, hvit som sne.'
puts pur if not pur.tainted?

# Tainting sverter av på andre objekter som lages med
# utgangspunkt i besudlede data.
puts 'Sverter av!' if (pur+skitten).tainted?

# Vi kan gjøre ting eksplisitt tainted...
pur.taint
# ..og renvaske skitten data (dersom vi får lov).
pur.untaint
```

Dette fungerer dog kun som øremerking og har lite innvirkning før det settes i en sammenheng hvor statusen tas hensyn til. Vi skrur opp paranoiaen et hakk og går til DefCon-1...

4.2 $SAFE-nivåer

I Ruby finnes det en spesiell variabel `$SAFE` som angir sikkerhetsnivået. I Ruby versjonene 1.6.* går verdien fra 0 til og med 4. (Se i *Programming Ruby*[9] kapittelet om *nedlåsing av Ruby* for detaljer.)

Vanlige Ruby programmer kjører normalt med `$SAFE==0` og eruby skript med `$SAFE==1` per default, men det kan stilles på i konfigurasjonen. Man kan øke sikkerhetsnivået, men aldri senke det.

[9] http://www.rubycentral.com/book/taint.html

```
[ code/safe1.rb ]
puts "Nåværende $SAFE-nivå: #{$SAFE}."

filnavn = "/etc/hosts"

# Eksempel på potensielt farlig operasjon.
print File.open(filnavn).read.size, " bytes lest.\n"

# La oss simulere en ekstern kilde og øke paranoiaen litt.
filnavn.taint
$SAFE = 1

begin
  print File.open(filnavn).read.size, " bytes lest.\n"
rescue SecurityError => sec_err
  puts sec_err #=> "Insecure operation - open"
end
```

Men hva gjør vi med en besudlet `String` da? For `$SAFE`-nivåer under 3, kan man gjøre litt "vasking" av dataene og så kalle `untaint`. For nivå 3 og oppover går det derimot ikke å kalle `untaint` på objekter.

5 Databaser

Det finnes en bråte av Ruby-bindinger til mange forskjellige databaser. Man kan bruke disse for å koble seg direkte til databasen og ha full tilgang til alle dens særegenheter.

For eksempel kan man benytte *MySQL-bindingene direkte*[10]. (En liten *innføring til Ruby og MySQL*[11])

5.1 Databasegrensesnitt: dbi

Men det gjør også at man knytter seg mer fast til typen database som brukes, så det kan være lurt å generalisere litt. Biblioteket ***dbi***[12] er et overbygg som gir et mer generelt grensesnitt til mange databaser i Ruby. Du må fremdeles installere de databasespesifikke bindingene, men i koden du skriver trenger du ikke knytte deg unødig tett til databasetypen.

```
[ code/dbi_select1.rb ]
require 'dbi'

databasenavn = 'kentda_rubynuby_webapp'
brukernavn   = 'kentda_rubynuby'
passord      = 'w3bRg0y1'
server       = 'mysql.pvv.ntnu.no'

```

[10] http://www.tmtm.org/en/mysql/
[11] http://www.rubywizard.net/ruby-mysql.html
[12] http://ruby-dbi.sourceforge.net/


```
DBI.connect("DBI:Mysql:database=#{databasenavn};host=#{server}",
            brukernavn, passord) do |dbh|

  # Et enkelt select-kall hvor hver resultatrad sendes til en block.
  dbh.select_all('select fornavn, etternavn from person') do |rad|
    puts rad.join(', ')
  end

  # Stedfortreder-argumenter (placeholders) i SQL-uttrykk.
  soek_etternavn = "Nordma%"
  soek_fornavn = "Ol%"
  dbh.select_all('select brukerid, brukernavn, fornavn, etternavn ' +
                 'from person where etternavn like ? and fornavn like ?',
                 soek_etternavn, soek_fornavn ) do |rad|
    puts rad.join(', ')
  end

  # Hent en liste (Array) med alle e-postadressene.
  liste = dbh.select_all('select * from person').collect{|rad| rad['epost'] }
  puts liste.join(', ')

end
```

Legg merke til at vi her angir blokker til metodekallene. Dette gjør at vi slipper å huske på å slippe ressursene løs selv; det skjer automatisk når blokken er ferdig utført.

Når man skal gi argumenter inn i SQL uttrykkene, kan man bruke strenginterpolering eller addere strenger selv. Men stedfortreder-argumenter, som brukt over, er å foretrekke, da det både er ryddigere, bruker mindre minne og lar biblioteket gjøre argument-escaping for deg.

5.1.1 SQL insert

Select-metodene er enkle å bruke, men man har ofte litt større behov, så som å skrive til databasen.

```
[ code/dbi_insert1.rb ]
  # 'do' kjører SQL-uttrykk og returnerer antall rader påvirket.
  n = dbh.do("INSERT INTO person VALUES (NULL, 'rubynuby1', " +
             " 'Nuby', 'Ruby', 'r0bY', 'nuby1@ruby.no')")
  puts "Antall rader påvirket av 'do': #{n}"

  # 'execute' kan brukes omtrent som 'do', men gir et
  # DBI::StatementHandle tilbake.
  sth = dbh.execute('INSERT INTO person VALUES (NULL, ?, ?, ?, ?, ?)',
                    'rubynuby2', 'Nuby', 'Ruby',
                    'r0bY', 'nuby2@ruby.no')
  puts "Antall rader påvirket av 'execute': #{sth.rows}"
  sth.finish # Ikke bruker block? Husk å lukke ressurser.
```

```

  # 'execute' kan også gi oss et "halvferdig" statement.
  dbh.prepare('INSERT INTO person VALUES (NULL, ?, ?, ?, ?, ?)') do |sth|
    # Kjekt når ting skal gjentas endel...
    sth.execute( 'rubynuby3', 'Nuby', 'Ruby', 'r0bY', 'nuby3@ruby.no')
    sth.execute( 'rubynuby4', 'Nuby', 'Ruby', 'r0bY', 'nuby4@ruby.no')
    sth.execute( 'rubynuby5', 'Nuby', 'Ruby', 'r0bY', 'nuby5@ruby.no')
  end # Møkk lei sth.finish nå!
```

6 Annet

Til nå har vi fort tatt for oss det nødvendigste, og man kan forsåvidt løpe ut og skrive noen webapplikasjoner nå. Men HTML og en database utgjør ikke det mest spennende i verden, så la oss titte dypere ned i verktøykassa.

6.1 E-post

Hva er vel en webapplikasjon som ikke kan spamme brukerne sine?

```
[ code/epost1.rb ]
require 'net/smtp'
til = 'kent_dahl@hotmail.com'  # Ikke /dev/null, men nært nok.
fra = 'kentda@pvv.org'
Net::SMTP.start('smtp.pvv.ntnu.no') do |smtp|
  tekst = [ "To: #{til}\n",
    "Subject: En liten test e-post\n", "\n",
    "Hei, hei, alle sammen. Har vi det bra dere?\n"
  ]
  smtp.sendmail( tekst,  fra, [til] )
end
```

Vi angir mottakerne med en `Array`, slik at vi kan sende en e-post til flere om gangen. I dette tilfellet kunne vi sendt en `String` direkte, siden det bare var en.

Dersom man skal skrive en stor e-post og ikke ønsker å bruke så mye minne på å bygge den opp i en `String` eller `Array` før man sender, kan man benytte instansmetoden ***`Net::SMTP#ready`*** etter å ha startet koblingen. Den tar en blokk med et argument, et adapter-objekt som du kan skrive til fortløpende.

6.2 Tråder

Ruby har et lett lite trådsystem internt. Det kjører fullstendig inne i fortolkeren, benytter ikke plattformspesifikke tråder, kan ikke dra nytte av flere CPUer og dersom en tråd kaller en C-metode som blokker, kan det sulte ut alle de andre trådene også. Men dersom mye av prosesseringen du gjør skjer i Ruby, er de veldig nyttige.


```
[ code/traad1.rb ]
puts 'A1'
traad = Thread.new do
  puts 'B1'
  sleep 2
  puts 'B2'
end
puts 'A2'
puts traad.join  # Vent på at tråden avslutter.
# Utput blir: A1 B1 A2 B2
```

Merk at når hovedtråden (main) avslutter, drepes alle andre tråder. Man må kalle `join` eller lignende på trådene om man ønsker å vente på dem.

Ønsker man å returnere en verdi fra en tråd, kan man kalle `value` i stedet for `join`.

```
[ code/traad2.rb ]
pi_traad = Thread.new do
  # Regn ut PI til ørten desimaler ...
  puts 'Regne, regne, regne!'
  sleep 5   # Vanlig student-innsats fra denne tråden...
  Math::PI  # ... og tror du ikke den koker også!
end
puts 'Tråden går i bakgrunnen.'
print 'PI er ', pi_traad.value  # Vent på returverdien fra tråden.
```

NB: Unntak som heves i tråder vil svinne hen i stillhet, med mindre vi eksplisitt venter på at tråden skal avslutte, eller `Thread.abort_on_exception` er satt til `true`.

6.2.1 Tråder som sandkasser

Tråder kan også brukes som sikre(re?) sandkasser. Husker du `$SAFE`-variabelen fra tidligere? Den er ikke en global variabel, men en tråd-lokal variabel. Hver tråd har sitt eget sikkerhetsnivå, og det kan vi benytte oss av for å kjøre "skumle" kodebiter i en noe tryggere "sandkasse".

```
[ code/traad_sandkasse1.rb ]
# Pakker sandkasselekingen i en metode slik at det
# ikke vil være (mange) variabler tilgjengelig i konteksten.
def sandkasse_lek( kode )
  sandkasse = Thread.new do
    # "evil" eval kan være skummel, så la oss være paranoide.
    $SAFE = 4   # Nivå 1-3 lar oss ikke bruke eval på
    eval kode   # besudlet data, men det gjør nivå 4!
  end
  print 'Koden din returnerte: ', sandkasse.value.inspect, "\n"
end

```

```
begin
  print 'Skriv inn vilkårlig Ruby-kode: '
  kildekode = gets.chomp              # Brukerdata er tainted.
  sandkasse_lek( kildekode )
end while kildekode.size.nonzero?
```

6.2.2 Synkronisering av tråder

Med trådene følger det en håndfull verktøy for synkronisering av tråder; `Thread.critical=`, `Mutex` og `ConditionVariable`.

```
[ code/traad_mutex1.rb ]
require 'thread'
$delt_teller = 0
$mutex = Mutex.new

# Lag ti tråder som øker den delte telleren
# gradvis tjuefem ganger.
traader = (1..10).collect do
  Thread.new do
    25.times do |i|
      $mutex.synchronize do                   ### Synkronisert
        gammel_verdi = $delt_teller            # kodebit.
        ny_verdi = gammel_verdi + 1            #
        sleep 0 # Framtvinge trådproblematikk  #
        $delt_teller = ny_verdi                #
      end                                     ###
    end
  end
end

# Vent på alle trådene før vi skriver ut den endelige verdien.
traader.each{|t| t.join}
puts $delt_teller # Skal skrive ut "250"
```

Det er flere synkroniseringsverktøy tilgjengelig, slik som `Queue`, `SizedQueue`, `Synchronizer` og `Monitor` for å nevne noen.

6.3 Distribuert Ruby - druby

Distribuert Ruby[13] (druby a.k.a. DRb) lar oss kommunisere og samhandle enkelt med andre Ruby-programmer over nettet, ikke ulikt RMI, CORBA (*rinn*[14]) eller XML-RPC (*xmlrpc4r*[15]) og lignende.

Et lite eksempel hvor vi deler ut et tjenerobjekt over nettet. Serveren:

[13] http://www2a.biglobe.ne.jp/~seki/ruby/druby.en.html
[14] http://sourceforge.net/projects/rinn/
[15] http://www.fantasy-coders.de/ruby/xmlrpc4r/index.html


```
[ code/drb_server.rb ]
require 'drb'
class Tjener
  def si_hei
    puts 'Hei!'
    return 'Hei fra tjeneren!'
  end
end
tjener = Tjener.new
DRb.start_service( 'druby://localhost:4242', tjener )
DRb.thread.join
```

... hvor vi må vente på at druby sin tråd skal avslutte, slik at serveren ikke stopper med en gang. Dernest kobler vi oss opp med en klient:

```
[ code/drb_klient.rb ]
require 'drb'
DRb.start_service
tjener = DRbObject.new( nil, 'druby://localhost:4242' )
svar = tjener.si_hei
print "Fikk fra tjener: '", svar, "'.\n"
```

Merk at vi angir `nil` som første argument til `DRbObject`-konstruktøren, siden vi ikke har et ordentlig objekt på "vår" side.

6.3.1 Verdioverføring og referanseoverføring

Oversending av parametre og returnverdier i distribuerte systemer deles i to hovedkategorier: verdioverføring (pass-by-value) og referanseoverføring (pass-by-reference).

Standard i druby er at objekter sendes ved verdioverføring. Objektene serialiseres (se ***Marshal*** -modulen), sendes over nettet og en kopi opprettes på mottakerens side. Dersom objektet ikke kan serialiseres/marshalles, fanges unntaket og det sendes over en referanse. Mottakeren ender da opp med en `DRbObject`-instans som videresender metodekall over nettet.

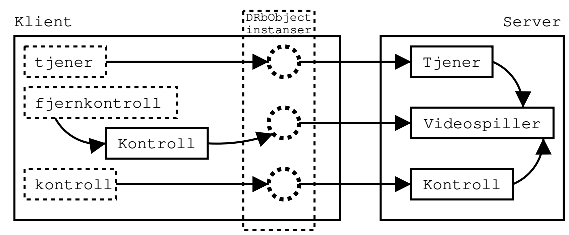

Figure 2: druby og videospiller/tvslave-eksemplet

[code/drb_videospiller.rb]
```
require 'drb'
$hvor = 'i stua'
class Videospiller
  # Gjør slik at Videospiller ikke kan serialiseres.
  include DRbUndumped
  def start
    puts "Videospiller: 'Press play on tape #{$hvor}.'"
  end

  class Kontroll
    def initialize( spiller )
      @spiller = spiller
    end

    def hvor_er_du?
      puts "Kontroll: 'Jeg er #{$hvor}.'"
    end
    def start
      @spiller.start
    end
  end # Kontroll

end # Videospiller

class Tjener
  def initialize
    @spiller = Videospiller.new
  end
  def ny_fjernkontroll     # pass-by-value
    Videospiller::Kontroll.new( DRbObject.new( @spiller ) )
  end
  def ny_kontroll          # pass-by-reference
    Videospiller::Kontroll.new( @spiller )
  end
  def hent_streng   # pass-by-value
    'En streng, en streng, mitt kongerike for en streng.'
  end
  def hent_standard_utput  # pass-by-reference da IO er en av
    $stdout                # klassene som ikke kan serialiseres.
  end
end

if __FILE__ == $0 then # Start tjeneren med våre utvidelser, og
  load 'drb_server.rb' # gjenbruker koden fra forrige eksempel.
end
```


[code/drb_tvslave.rb]
```
require 'drb_videospiller' # Trenger definisjonen til Kontroller
$hvor = 'på soverommet'
DRb.start_service
vcr_tjener = DRbObject.new( nil, 'druby://localhost:4242' )

tjener_utput = vcr_tjener.hent_standard_utput
puts_begge_steder = proc do |tekst|
  puts tekst
  tjener_utput.puts tekst
end

puts_begge_steder.call('--Fjernkontrollen kan vi ta med inn på soverommet')
fjernkontroll = vcr_tjener.ny_fjernkontroll
p fjernkontroll
fjernkontroll.hvor_er_du? #=> Kontroll: 'Jeg er på soverommet.'
fjernkontroll.start       #=> Videospiller: 'Press play on tape i stua.'

puts_begge_steder.call('--Mens kontrollen som er tett knyttet til ' +
                       "videospilleren\n--må forbli i stua.")
# Så vi får bare en referanse til, og ikke en kopi av, kontrollen.
kontroll = vcr_tjener.ny_kontroll
p kontroll
kontroll.hvor_er_du?  #=> Kontroll: 'Jeg er i stua.'
kontroll.start        #=> Videospiller: 'Press play on tape i stua.'
```

6.3.2 Apache-prosesser og druby-server

Dette kan benyttes for å flytte logikk i fra Apache-prosessene og over i en samlet druby-prosess.

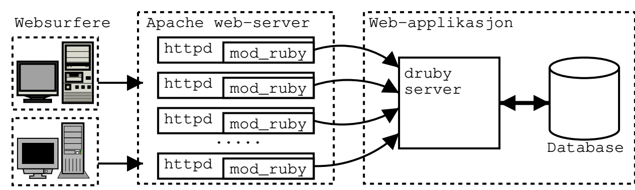

Figure 3: Arkitektur hvor logikken separeres fra mod_ruby-prosessene.

7 Videre

Dette har vært en kjapp gjennomgang av noen verktøy for webprogrammering i Ruby. På RAA (*Ruby Application Archive*[16]) finner du flere biblioteker som kan hjelpe deg. Noen av disse er allerede nevnt i 2.5 (webrammeverklisten), men det er mange, og de har ofte litt sære navn.

Dersom du gjør mye CGI kan det være en ide å titte på *narf*[17], et alternativ til standard CGI-biblioteket.

Går det mye i HTML, kan det være ønskelig å ha en ryddigere separering av HTML-filer og Ruby-kode. *Amrita*[18] har blitt nevnt og er et av de mer populære templatingsverktøyene.

Har du ingen webserver å kjøre på eller vil skrive din egen? Ta da en titt på *WEBrick* .

Ønsker du enkle og ryddige webapplikasjonsrammeverk som lar deg både utvikle og prototype raskt? Titt nærmere på *Radical* , *Borges* eller *IOWA* .

Sist, men ikke minst: Ikke få panikk om du googler og får opp japanske sider. Ruby-koden pleier å være lesbar nok til å gi en løsning på problemet, om ikke annet så iallefall i kombinasjon med det Babelfish spyr ut.

8 Øvingsoppgave

Her følger et forslag til øvingsoppgave for å få prøvd ut det vi har gått igjennom så langt.

Oppgaven går i korte trekk ut på å lage et web-basert system i Ruby hvor studenter kan registrere seg på forskjellige kurs og presentasjoner, slik som dette.

8.1 Oppgave 1)

Implementer følgende krav til systemet i Ruby.

1. Brukere kan logge inn med et brukernavn og passord. (Ikke stud-passordet!)
2. Brukere som logger inn vises en oversikt over:
   (a) Kurs de er påmeldt, som holdes i dag eller i fremtiden.
   (b) Kurs som er tilgjengelig for oppmelding og som holdes innen de neste 30 dagene.
   (c) Nyheter i kurs de er påmeldt.

Dere kan bruke en eksisterende MySQL-database som er ferdig satt opp, da disse kravene ikke trenger skrivetilgang:

```
Server:       mysql.pvv.ntnu.no
Database:     kentda_rubynuby_webapp
Brukernavn:   kentda_rubynuby
Passord:      w3bRg0y1
```

Ta en titt på 9.1 (ER-diagrammet) for en oversikt.

[16] http://raa.ruby-lang.org/
[17] http://narf-lib.sourceforge.net/doc/
[18] http://www.brain-tokyo.jp/research/amrita/

8.2 Oppgave 2)

Utvid systemet med følgende krav. Vel innlogget skal brukere kunne:

1. Registrere seg på kurs som vises i oversikten.
2. Avregistrere seg på kurs de er påmeldt på.
3. Logge ut.

Da disse og senere krav krever skrivetilgang, må dere ha en egen MySQL-database. Dere kan enten sette opp selv, eller få hjelp til dette. Dersom dere setter opp selv, kan dere gjerne endre databaseskjemaet hvis dere syntes designet er uhensiktsmessig, men det kan da bli litt vanskeligere å få hjelp.

Informasjon om *hvordan sette opp MySQL-databaser på stud*[19], som også stemmer for PVV.

8.3 Oppgave 3)

Utvid systemet med følgende krav:

1. Nye brukere kan opprettes forutsatt at brukernavnet ikke allerede er tatt.
2. Ved opprettelse av ny bruker genereres et tilfeldig passord som sendes på e-post til den nye brukeren for å bekrefte identiteten.

Dette krever mulighet for å sende e-post. Titt på dokumentasjonen til *Net::SMTP* og benytt enten smtp.stud.ntnu.no eller smtp.pvv.ntnu.no som utgående mailserver. (Bruk deres egne e-post adresser til testing.)

8.4 Oppgave 4)

Enda mer featuritis. Administratorer skal kunne:

1. Endre passord på andre brukere.
2. Legge til kurs

Administratorer og kursholdere skal kunne:

1. Legge inn nyheter på kurs.
2. Angi en viktighetsgrad på nyheten. (Høy/Lav)
3. Dersom viktighetsgraden er høy, skal systemet sende nyheten på e-post til alle som er påmeldt. (Pass på å bare sende til dere selv når dere tester! Ingen spamming takk!)

Siden det kan være mange som skal ha nyheten på e-post, ønsker ikke administratoren/kursholderen å vente til alle e-poster er sendt før han kan trykke videre i systemet.

[19] http://infoweb.ntnu.no/utvikling+og+programmering/database/mysql.html

8.5 Oppgave 5)

Tenkt scenario: Programvareverkstedet ønsker å ta i bruk det fine systemet ditt, men er ikke like begeistret for databaseavhengigheten. PVV bruker allerede *ADiCT*[20] og ønsker å integrere systemet ditt opp mot ADiCT gradvis.

Første oppgave kan da deles opp slik:

1. Separer databaselogikken fra koden som lager HTML-sidene.
2. Lag et overordnet objekt som gir tilgang til, og innkapsler, databaselogikken.
3. Flytt databaselogikken over i en egen prosess og kommuniser mellom mod_ruby/CGI-skript-prosessene og denne ene prosessen via druby.

9 Vedlegg

9.1 Databaseskjema til oppgave

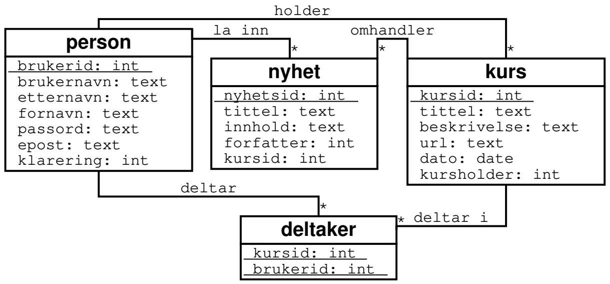

Figure 4: Oversiktsdiagram over den enkle testdatabasen.

SQL koden m.m. er å finne via *Ruby Web-Nuby siden på PVV*[21].

[20] http://www.adict.net/
[21] http://www.pvv.org/~kentda/ruby/webnuby/