

Historikken til dokumentet

2017-03-17: UTKAST

Formål

Blir kjent “concurrency” programmering i Go programmeringsmiljø.

Oppgave

Denne oppgaven er et systemutviklingsprosjekt. ICA05 skal inneholde et systemspesifikasjon inkludert beskrivelse av brukerscenarioer (med forslag til test-scenarier, dvs. hva må man teste på for å garantere en pålitelig tjeneste) og systemarkitektur (systemarkitektur viser alle nodene, som er involvert i systemet og beskriver deres funksjon). ICA05 skal også inkludere en fungerende webserver i UH-laaS skyen og besvare et forespørsel på port 8001. Koden til denne skal ligge i en Git repository.

Siden som skal vises, skal være en kort markedsføring av deres prosjekt (eksempler: finne “alt” om et spesifikt tema, som nyheter, twittermeldinger, facebook profiler; finne spesifikke objekter basert på lokasjon og levere en liste / informasjon til brukeren; følge med på “realtids-data” fra noen databaser på WWW; levere en værmelding fra flere værdatabaser; levere produktinformasjon osv.). Siden trenger ikke å vise “reelle” data fra kilder i denne omgangen (det blir oppgaven i ICA06).

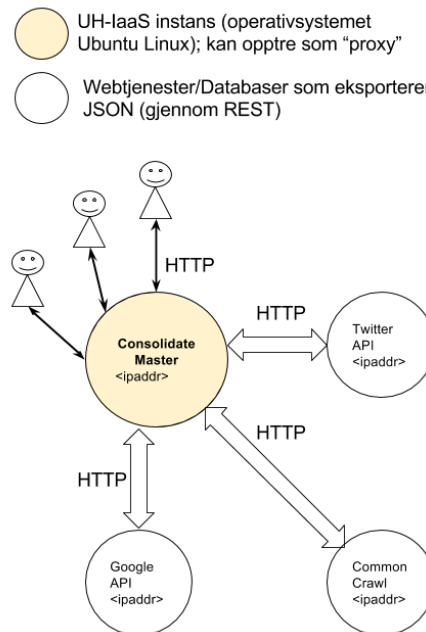
Det som skal evalueres er

- (1) oppsett og konfigurasjon av en (eller flere) webserver i nettskyen (må være i UH-laaS):
 - (a) serveren må respondere på http-request
 - (b) serveren må bruke templates i Go
- (2) dokumentasjon av design til deres data-konsoliderings-side
- (3) oppsett av kildekode i Github (ren kode med kommentarer og implementerte tester der det er mulig)

I tillegg skal dere lage et “vanlig” Go program (ikke en webserver, men en slags webklient), som bruker goroutines (se Pike’s foredrag <https://vimeo.com/49718712> opp til 17:53; se også hele turen om “concurrency” her <https://tour.golang.org/concurrency/1>) og henter inn data fra minst 5 forskjellige kilder (gjennom definerte API-er) og legger data inn i Go strukturer (i RAM i første omgang, dvs. det er ikke nødvendig å bruke databaseapplikasjoner for lagring av data) og lager en utskrift som viser korte utdrag av data fra hver kilde.

Det som skal evalueres er, om deres program viser meningsfulle data til brukeren, dvs. ikke data i JSON-format, men strukturerte data, som “Location: Kristiansand, Temperature high: 5 degrees Celsius, Temperature low: 0 degrees Celsius” osv. Formatet kan dere velge selv, den må være brukbart (se eksempel senere i dokumentet).

Disse aktivitetene for ICA05 skal være forløperen til påfølgende ICA-er. Dere skal fortsette arbeidet med deres prosjekt basert på deres systemspesifikasjon. Derfor blir en mer helhetlig beskrivelse gitt her. Bruke denne beskrivelsen for å definere brukerscenarioer.



Illustrasjonen viser en overordnet forslag (en skisse) for systemarkitektur. Jeg har gitt programmet navn "Consolidate" og viser her en Master-node. Denne noden er sammensatt, dvs. den vil mest sannsynlig bestå av flere programmer / moduler (en modell kan være "proxy"-modellen https://en.wikipedia.org/wiki/Proxy_server). Den skal "kravle rundt" i WWW, samle inn data, bearbeide disse dataene og gjøre de tilgjengelig / brukbare for sluttbrukere.

Et eksempel på konsolidering av data er Google's sin side om finansmarkeder, hvor man, blant annet kan se kursinformasjon i tilnærmet realtid (det er selvsagt alltid en forsinkelse forbundet med bearbeidelse og overføring av data) <https://www.google.com/finance>

Det er ønskelig at dataene skal vises frem i en applikasjon. Det kan være en nettleser, men det kan også være et annet grensesnitt (mobilt, f.eks.).

Det anbefales å starte med planlegging.

Velg et tema, som dere synes er interessant og søk etter mulige datakilder. Jeg kan anbefale å se på to linker:

- <https://www.forbes.com/sites/bernardmarr/2016/02/12/big-data-35-brilliant-and-free-data-sources-for-2016/#515469beb54d>
- <http://commoncrawl.org/>

Utviklingsmiljøet er denne gangen gitt, - Golang, JSON format for utveksling av data. Hvis dere velger en nettleser (kan alternativt være en app på en mobil enhet) for å vise frem de konsoliderte dataene, så er HTML, CSS og Javascript aktuelle utviklings-artefakter.

Bli kjent med aktuell kode / eksempler. Denne linken inneholder det mest nødvendige for å sette opp en webserver og tolke JSON-formatet

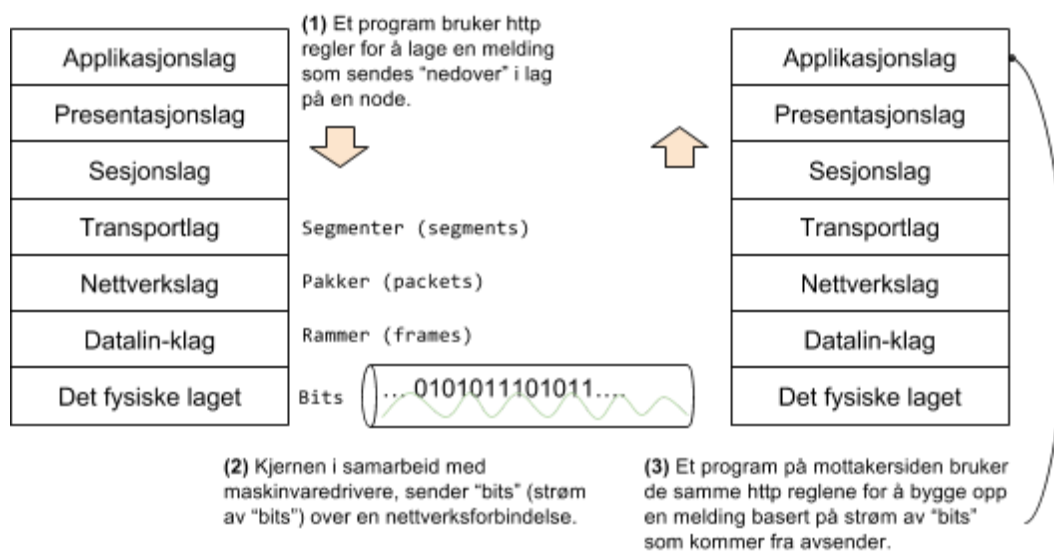
<http://www.alexedwards.net/blog/golang-response-snippets>

Rob Pike's sitt foredrag inneholder introduksjon til hvordan utføre flere Go rutiner "samtidig" og få de til å samarbeide med hverandre.

Beskrivelse med eksempler

Webserver

En oversiktsmodell (OSI-modell) beskriver alle moduler / lag involvert for å sende en melding fra en sender til en mottaker. Til nå har vi kun sett på data representasjoner i de øverste lagene, som er Applikasjonslaget og Presentasjonslaget (forbereder data for Applikasjonslaget, som å formattere med ASCII, f. eks.). Våre go-programmer ligger alle i applikasjonslaget.

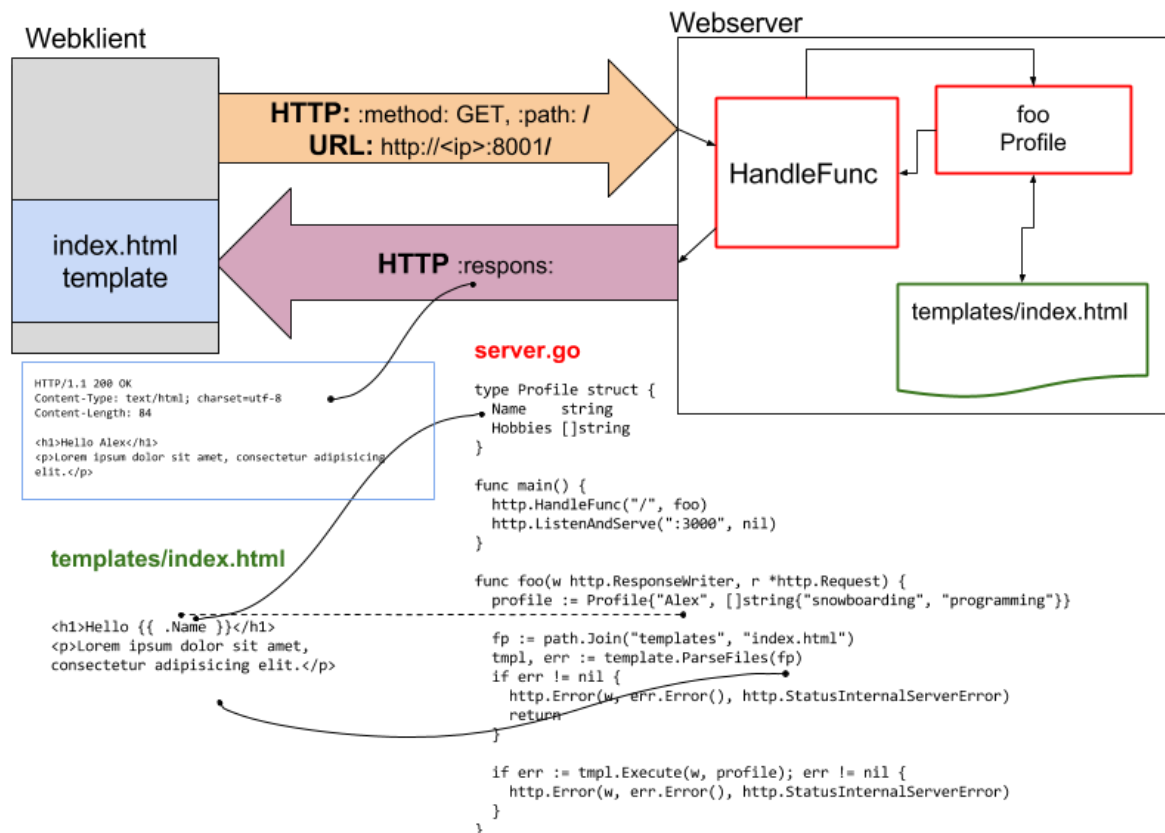


Et program, som bruker HTTP (HyperText Transfer Protocol) spesifikasjonen (regler, <https://www.w3.org/Protocols/>) for å tolke og lage meldinger (request og response), kaller man ofte for en **webserver**. Begrepet kommer fra tiden i begynnelsen av 1990-tallet, når Tim Berners-Lee definerte HTTP spesifikasjonen og HTML formatet for utveksling av data i noe han kalte for World Wide Web. Den første webklient (kalt også nettleser) het nettopp WorldWideWeb (<https://en.wikipedia.org/wiki/WorldWideWeb>) og så slikt ut i et terminalvindu <http://line-mode.cern.ch/www/hypertext/WWW/TheProject.html>

Den første webserveren var CERN httpd, - CERN fordi at Tim Berners-Lee jobbet for CERN, httpd for å påpeke at det er en demon (daemon), som utfører i bakgrunnen og lever vanligvis så lenge systemet er "oppe"-rativt (https://en.wikipedia.org/wiki/CERN_httpd).

En enkel webserver i Go, som kan respondere med en korrekt HTML side, kan bygges basert på denne kilden og eksemplet "Rendering an HTML template"

<http://www.alexedwards.net/blog/golang-response-snippets#html>



Denne illustrasjonen prøver å fange alle aspekter med eksemplet fra <http://www.alexedwards.net/blog/golang-response-snippets#html> (det er litt "rot" med portene her, - webserver hører på porten 3000, mens klienten spør / sender forespørsel på porten 8001; portnummer som webklient bruker, må selvsagt være lik portnummeret til webserver, for at de skal klare å kommunisere).

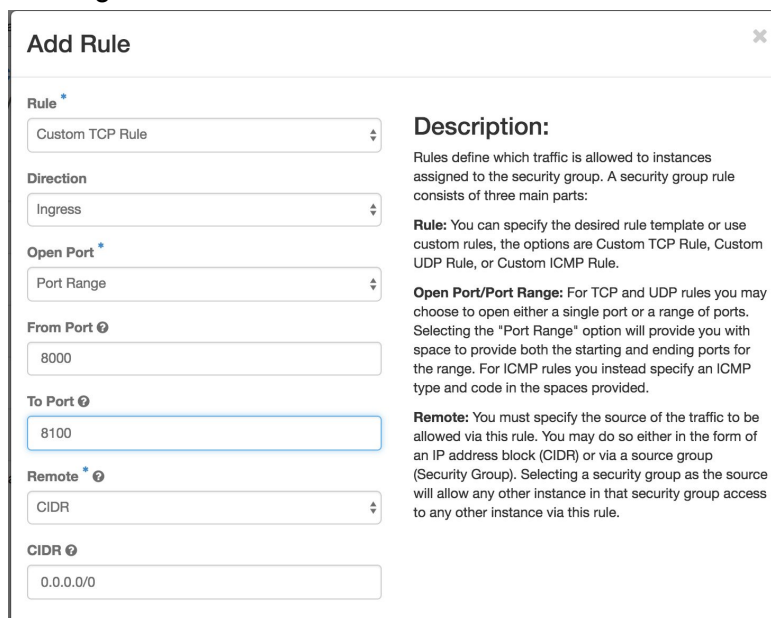
Brukerscenario her er at Webklient (som kan være representert av en nettleser), gjør en HTTP-Request til Webserver (som kjører i deres UH-laaS instansen). server.go er et main-program i golang, som bruker pakker "html/template", "net/http" og "path" (ikke vist i illustrasjonen), for å implementere en HTTP-Response (dvs. konstruere en HTML-fil) og respondere med denne til Webklient. Det er viktig å påpeke at server.go er en fullverdig webserver, som klarer å motta mange asynkrone kall, siden pakken "net/http" er implementert basert på goroutines, slik at funksjonene i denne pakken kan starte mange go-rutiner (se Rob Pike's video og slides).

Dere kan bruke denne template (eller lage deres egen) for å lage markedsføringssiden for deres konsoliderings-applikasjonen.

Konfigurasjon av UH-laaS

For å gjøre webserveren tilgjengelig i deres skyinstans, må dere laste opp koden og åpne de aktuelle portene. Det forutsettes at dere klarer å laste opp filene til UH-laaS instansen deres.

For å åpne portene, gå til “Access & Security” i deres panel og åpne “Manage Rules”. Bruk “+ Add Rule” og spesifiser en “range” av porter (jeg anbefaler 8000 - 8100, men vær oppmerksom på at eksemplene i <http://www.alexedwards.net/blog/golang-response-snippets#html> bruker porten 3000, så dere må eventuelt endre dette til 8001, for eksempel). Se illustrasjon 1 og 2 for hvordan du lager en åpen “port range” i din UH-laaS instans.



Add Rule

Rule *
Custom TCP Rule

Direction
Ingress

Open Port *
Port Range

From Port ?
8000

To Port ?
8100

Remote * ?
CIDR

CIDR ?
0.0.0.0/0

Description:
Rules define which traffic is allowed to instances assigned to the security group. A security group rule consists of three main parts:
Rule: You can specify the desired rule template or use custom rules, the options are Custom TCP Rule, Custom UDP Rule, or Custom ICMP Rule.
Open Port/Port Range: For TCP and UDP rules you may choose to open either a single port or a range of ports. Selecting the "Port Range" option will provide you with space to provide both the starting and ending ports for the range. For ICMP rules you instead specify an ICMP type and code in the spaces provided.
Remote: You must specify the source of the traffic to be allowed via this rule. You may do so either in the form of an IP address block (CIDR) or via a source group (Security Group). Selecting a security group as the source will allow any other instance in that security group access to any other instance via this rule.

Illustrasjon 1. Portene fra 8000 til 8100 vil bli tilgjengelig f. eks. vha. URL `http://<ip>:8000`

<input type="checkbox"/>	Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Actions
<input type="checkbox"/>	Egress	IPv6	Any	Any	::/0	-	Delete Rule
<input type="checkbox"/>	Egress	IPv4	Any	Any	0.0.0.0/0	-	Delete Rule
<input type="checkbox"/>	Ingress	IPv4	ICMP	Any	0.0.0.0/0	-	Delete Rule
<input type="checkbox"/>	Ingress	IPv4	TCP	22 (SSH)	0.0.0.0/0	-	Delete Rule
<input type="checkbox"/>	Ingress	IPv4	TCP	8000 - 8080	0.0.0.0/0	-	Delete Rule

Displaying 5 items

Illustrasjon 2. Etter at man har gjort “Add Rule”, følgende illustrerer statusen. I dette eksemplet er 8000 til 8080, som er “port range” som er åpne.

Webklient

For å demonstrere goroutines, som henter data fra forskjellige kilder, kan dere bruke en kode for en webklient (denne må settes i en main funksjon og startes som en goroutine for hver URL/Data kilde som dere skal hente data fra):

```
func doGet(url string) {
    response, err := http.Get(url)
    if err != nil {
        log.Fatal(err)
    } else {
        defer response.Body.Close()
        contents, err := ioutil.ReadAll(response.Body)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("The calculated length is:", len(string(contents)), "for
the url:", url)
        fmt.Println("    ", response.StatusCode)
        hdr := response.Header
        for key, value := range hdr {
            fmt.Println("    ", key, ":", value)
        }
    }
}
```

I dette eksemplet blir ikke innholdet i selve responsen skrevet ut, kun lengden til innholdet og statuskode for responsen, samt de såkalte “header”-verdiene. Innholdet er tilgjengelig i contents variabelen i dette tilfelle.

Dere kan enkelt skrive ut innholdet med

```
fmt.Printf("%q", contents)
```

I tilfelle linken er fra openweather.org (jeg kaller den OWL - OpenWeatherLink videre):

<http://samples.openweathermap.org/data/2.5/weather?zip=94040,us&appid=b1b15e88fa797225412429c1c50c122a1>

blir contents følgende (sist hentet 2017-03-19, og litt reformatert for være mer brukbar):

```
{
  "coord":{"lon":-122.08,"lat":37.39},
  "weather":[{"id":500,"main":"Rain","description":"light rain","icon":"10n"}],
  "base":"stations",
  "main":{"temp":277.14,"pressure":1025,"humidity":86,"temp_min":275.15,"temp_max":279.15},
  "visibility":16093,
  "wind":{"speed":1.67,"deg":53.0005},
  "clouds":{"all":1},
  "dt":1485788160,
  "sys":{"type":1,"id":471,"message":0.0116,"country":"US","sunrise":1485789140,"sunset":1485826300},
  "id":5375480,
  "name":"Mountain View",
  "cod":200
}
```

```
}
```

Her gjelder det å studere API-en og finne ut hva de forskjellige dataene representerer.

Hvis data er i JSON format, som i dette eksemplet (det kan man sjekke ved å se på verdiene i `hdr := response.Header` fra funksjonen `doGet` ovenfor), kan dere lage passende strukturer og mate contents inn i `json.Decode` funksjonen

(<https://golang.org/pkg/encoding/json/#Decode>), fra pakken “encoding/json” (se kommentarene i kodeeksemplet):

```
func exDecodeMine() {
    // Her brukes det kun et utdrag fra data som var i responsen fra OWL
    // For å bruke strøm fra doGet funksjonen, må hele JSON-strukturen
    // defineres; kun Coordinates og Measurements (main) er definert i
    // dette eksemplet
    var jsonStream = `
        {
            "coord":{"lon":-122.08,"lat":37.39},
            "main":{"temp":277.14,"pressure":1025,"humidity":86,"temp_min":275.15,
            "temp_max":279.15}
        }
    `

    // Definerer en struktur i Golang etter strukturen fra API-en (openweather)
    // Her kan man virkelig se “styrken” av Golangs struct
    // Datafelt i struct må være med en storbokstav og navn må tilsvare
    // de navn som er i jsonStream (de kan begynne med små bokstaver)
    type Coordinates struct {
        Lon float64
        Lat float64
    }
    type Measurements struct {
        Temp      float64
        Pressure float64
        Humidity  float64
        Temp_min float64
        Temp_max float64
    }
    type Weather struct {
        Coord Coordinates
        Main  Measurements
    }

    // Ting er strøm-basert, som vi har snakket om tidligere
    dec := json.NewDecoder(strings.NewReader(jsonStream))
    for {
        // Definerer struktur for en instans av Weather strukturen
        // Dette avhenger selvfølgelig om hva som returneres fra
        // webtjenesten (openweather i dette tilfelle)
        var w Weather
        // Passerer adressen til Weather-strukturen w til funksjonen
        // Decode (som kalles fra en json.NewDecoder med
        // strings.NewReader(jsonStream) som IN-DATA-STRØM
```

```

        // Når det ikke er mer data (EOF) bryter vi utførelsen av
        // denne funksjonen med break
        if err := dec.Decode(&w); err == io.EOF {
            break
        } else if err != nil {
            log.Fatal(err)
        }
        // Her er et par eksempler på hvordan man kan skrive ut
        // data fra denne webtjenesten på en brukbar måte
        // Dette er noe dere skal prøve å imitere med data
        // fra andre webtjenester (med andre API-er, selvsagt)
        fmt.Printf("Coordinates are: longitude %.2f and latitude %.2f\n",
w.Coord.Lon, w.Coord.Lat)
        fmt.Printf("Temperature: %.f\n", w.Main.Temp)
    }

}

```

Denne koden må refaktoreres (<https://no.wikipedia.org/wiki/Refaktoring>), dvs. brytes opp i mindre enheter og den må også testes.

Formater

Alt i datasystemer er kodet og lagbasert. Ved anvendelse av spesifikk kode, kan man også si at man formatterer data på spesifikk måte. Alle strøm av "0" og "1" på lavere lag er formattert. Dere har sett åpne formater, som ASCII og UTF-8, som hører til på høyere lag (Presentasjonslaget i OSI-modellen og blir tolket av tekstbehandlere, nettlesere).

Nå er det også tid å bli kjent med formater for spesifikt strukturerte data (ikke bare tekst for "vanlige" språk, men tekst som datamaskiner kan lett tolke), som f. eks. CSV, TSV, XML, JSON osv. Comma Separated Values og Tab Separated Values har dere sikkert kjennskap til. Det er en meget vanlig måte å lagre data på, spesielt i forskningssammenhenger. XML er eXtensible Markup Language og er et forsøk på å lage et helt generelt tekst-basert format for strukturering av data. JSON er JavaScript Object Notation og er den mest brukte formatet for utveksling av data i WWW. En stor fordel med den er at den er også "human-readable", dvs. kan tydes av mennesker, samtidig som den er egnet for dataprosessering. I denne oppgaven får dere selv erfaringer med dette.

I tillegg finnes det mange binære formater for koding av AV media, komprimering / de-komprimering og arkivering / de-arkivering av filer av alle type formater osv. Disse brukes i diverse transaksjoner mellom systemer, samt av spesifikke applikasjoner som AV media avspillere, pakke- og arkiveringsverktøy med GUI osv.

JSON formatet er beskrevet her <http://www.json.org/>
 JSON støtte er implementert i Golang pakken encoding/json
<https://golang.org/pkg/encoding/json>

Det er mye detaljer her, så det er viktig å ikke gi opp men prøve å legge "puslespillbiter" sammen. Kopier eksempler fra Golang (som denne

<https://golang.org/pkg/encoding/json/#Decoder>) inn i din egen fil lokalt og prøv å eksperimentere med JSON-data uavhengig av nettsider. Du kan bare kopiere dataene inn fra en API-kall som dette
`http://samples.openweathermap.org/data/2.5/weather?zip=94040,us&appid=b1b15e88fa797225412429c1c50c122a1`
og prøve ut Golangs funksjoner i encoding/json pakken inntil du får overført data til en struct i Golang (som vist i eksemplet ovenfor).

Et format, som blir brukt mye i forhold til arkivering av all informasjon på WWW er ARC format:

“The ARC format file has been created in 1996 by Brewster Kahle and Mike Burner from the Internet Archive for managing billions of objects, and is used today by several national libraries.”

<http://bibnum.bnf.fr/WARC/>

Nevner formatet her, siden WebArchive inneholder enorme historiske data, som kan være aktuelle for konsoliderings-applikasjonen.

Spesifikasjon av og eksempler for JSON-LD formatet, som brukes for å prosessere ARC formatet, kan dere finne her <http://json-ld.org/>
JSON-LD støtte er implementert i dette prosjektet <https://github.com/kazarena/json-gold>

Lykke til!

SLUTT.

JG/2017-03