Systemarkitektur for webapplikasjon for værdata

Vårt system har to hovednoder; webtjeneren og klienten. Under hver av disse nodene er det selvfølgelig flere sub-noder. Disse vil vi prøve å gå gjennom og forklare nærmere i dette dokumentet. For å gjøre oss best forstått, ønsker vi å gå gjennom systemet ved å forklare nodene om en annen. Det vil si at i stedet for å forklare tjeneren og så klienten, går vi fram og tilbake mellom dem. I tillegg har vi en test-node som vi går gjennom på slutten.

Hardware & software og teknologi

Før vi går dypere inn på funksjoner, vil vi skrive litt om maskinvare, programvare og teknologi. På tjenersiden er alle filer lagret lokalt på block device eller non volatile memory (SSD, HDD). OS på server-PC under utvikling var Windows 10, som gir go-serveren et standardisert grensesnitt og tilgang til nettverkskort og andre hardware enheter som CPU og minne. Tjener består av weatherServer.go, samt en rekke HTML filer som gjør det mulig for oss å vise frem data gjennom klientens nettleser.

På klientsiden aksesserer bruker applikasjonen ved hjelp av en nettleser og URL adresse. Ingen data lagres på block device når det kommer til klientens maskin. All data som siden sender og klient mottar skal ligge i minnet på maskinene eller volatile memory. Som vil si at all data lastet inn forsvinner ved restart av enheten.

API

I vår webapplikasjon har vi brukt to forskjellige APIer; en for værdata og en for kart/stedssøk. Værdataen er hentet fra Dark Sky. Dark Sky leverer værdata for hele verden i JSON-format. Et kall til APIen ser slik ut:

`https://api.darksky.net/forecast/[key]/[latitude],[longitude]`

Vi bruker en gratisversjon, og får derfor 1000 API-kall til dagen. APIen har også flere valgmuligheter for værdataen du vil hente. Vi la på to query parameter; *lang=no* og *units=si*, for å endre språk til norsk og få norske måleenheter(mm, celsius og m/s).

Den andre APIen er Google Maps JavaScript API. For å bruke denne APIen måtte vi også her ha en API-nøkkel. Denne fikk vi ved å registrere et prosjekt på Google Cloud Platform. Google Maps APIen ble brukt med flere hensikter. Vi brukte eksempelkoden for Geocoding, funnet i dokumentasjonen, med et par modifikasjoner. Dette passet oss utmerket ettersom ingen av oss hadde noe erfaring med JavaScript. API-kallene her blir gjort når brukeren søker etter sted og får koordinater i retur. På forecast.html blir det også gjort et API-kall hvor koordinatene blir søkt på, og et map med marker blir returnert. Et slikt API-kall blir kalt for Reverse Geocoding. Mer om dette i den mer detaljerte gjennomgangen av systemets funksjoner.

Teknologi

Tjeneren bruker HTTP (HyperText Transfer Protocol) som protokoll. HTTP er til for å snakke mellom klient og tjener. For at en klient skal få innholdet på webtjeneren, må den gjøre en forespørsel. Slike HTTP-forespørsler kan være “GET”, “POST”, “PUT, “HEAD”, og “DELETE”. I vårt tilfelle er det kun “GET” og “POST” som er relevant.

Når klienten skriver inn adressen til tjeneren i URL-feltet på nettleseren, sender klienten en HTTP GET forespørsel til tjeneren. Denne forespørselen inneholder en header og en

body. Headeren inneholder informasjon om klient (hvilken nettleser, språk, tillatte filtyper). Bodyen inneholder eventuelle parametre som klienter sender: for eksempel "www.test.com/**?lang=en**. "?lang=en" er da et parameter som ville vært i bodyen. Når webtjeneren mottar denne "GET"-forespørselen sender den tilbake en respons til klienten. Responsen inneholder blant annet en statuskode. Statuskoden 200 bekrefter at forespørselen til klienten var vellykket. Det finnes mange forskjellige statuskoder, og vi har brukt noen av dem i systemet vårt for å signalere feil.

Beskrivelse av systemets funksjoner

1: Lytter på port 8080

Når server() blir kjørt startes det en "listenAndServe()"-funksjon. Dette er en funksjon fra en pakken 'net/http' i Go-biblioteket. Funksjonen tar adressen 'localhost:8080' som parameter. Det vil si vi har nå en server som lytter etter TCP-koblinger på localhost, port 8080. Port 8080 er et alternativ til port 80, som er den offisielle porten for HTTP. Funksjonen lytter da etter HTTP-forespørsler.

Serveren har to handleFunctions som kaller den rette funksjonen med de rette parametrene basert på forespørselen. Når da en klient gjør en forespørsel, sender listenAndServe-funksjonen forespørselen videre til den rette handleFunctionen basert på hvilken sti forespørselen har.

2: Server index

Når en klient gjør en forespørsel på "localhost:8080/" blir index() kjørt. Index-funksjonen har to roller, den første er å respondere med en html-fil til brukeren, den andre er å sjekke etter input fra brukeren. Da passer det å se på klient-siden.

Klienten blir bedt om å søke etter et sted hvor klienten ønsker å se værdata for. Dette blir gjort mulig av Google Maps Javascript API. Vi bruker søkefunksjonen til APIet for å hente ut koordinater til det ønskede sted. Når brukeren trykker på 'Søk', hentes koordinatene (ved hjelp av Gmaps API) og legges, sammen med stedet som søkes på, i usynlige <input>-felt. Deretter submittes dataen ved hjelp av javascript. Samtidig dukker det opp en knapp som fungerer som en link til 'localhost:8080/forecast'. Når dataen er blitt submittet, blir forespørselen til HTTP POST, og ikke HTTP GET.

Tilbake på tjeneren sjekkes da forespørselsmetoden. Denne er "POST" og uthentingen av brukerinputen begynner. Dette skjer i funksjonen getForm(). Her hentes dataen og legges i variabler.

3: Knapp "vis været" blir available og bruker blir omdirigert til /forecast

Funksjon runForecast() kjøres når brukeren trykker på knappen som tar dem til værdataen. Koordinatene (som ble hentet i forrige punkt) settes sammen med URLen til DarkSky, API-nøkkelen og query parameterne som gjør at vi får værdataen på norsk og i norske måleenheter(mm, celsius og m/s). Her gjøres det API-kall til Dark Sky-APIen. Dark Sky tilbyr, som nevnt tidligere, dataen i JSON-format. Vi gjør en 'GET'-forespørsel til den rette URLen og leser dataen som kommer i responsen. Her bruker vi 'json'-pakken i Go-biblioteket til å dekode dataen og legge dataen inn i det passende structet.

Deretter gjør serveren kalkuleringer på vindhastigheter, nedbørsmengder og temperaturer for å finne en passende melding om været til sluttbrukeren. Når den rette meldingen er blitt valgt, lages det et Go template av forecast.html-filen og den andre passende html-filen (som inneholder tilbakemeldingen). Dette templatet blir kjørt sammen med struct-variabelen som inneholder all værdataen. Denne værdataen blir tatt i bruk ved hjelp av tags i HTML-filene. Disse aksesserer structet, henter de verdiene som er satt til de egenskapene (properties) som blir nevnt i taggene. Vist under:

Rå HTML

```
<div class="rad">
    <div class="col">
        <h4 style="text-align: center">Været for {{.Sted}} akkurat nå!</h4>
        Temperatur:           {{ .Currently.Temperature}}°C <br>
        Føles som:            {{ .Currently.ApparentTemperature}}°C <br>
        Fuktighet:            {{ .Currently.Humidity}} <br>
        Tekstoppsummering:    {{ .Currently.Summary}} <br>
        Nedbør:               {{ .Currently.PrecipIntensity}} mm <br>
        WindSpeed:            {{ .Currently.WindSpeed}} m/s <br>

    </div>
```

Output. Rendret med værdata

Alt over i de to avsnittene skjer når brukeren aksesserer/sender ‘GET’-forespørsel til /forecast. Brukeren får respons med HTML-filen ‘forecast.html’. Her er det et script som bruker taggene *{{ .Latitude }}* og *{{ .Longitude}}* til å hente kart levert av Google Maps API, med en marker på det stedet brukeren får værdataen fra. Dette API-kallet blir kalt for reverse geocoding. Vi ser at det skjer både API-kall på tjenersiden (Dark Sky) og på klientsiden (Google Maps).

Abstraksjon:

Mange av funksjonene i ‘net/http’-pakken gjør at vi slipper å tenke på hvordan tjeneren vår skal få tilgang til tjenermaskinens nettverkskort og drivere. Dette er et bra eksempel på hvordan abstraksjon i GoLang gjør livet lettere for oss som utviklere

Go bruker maskinens nettverkskort/drivere. Standardiserte funksjoner som er bakt inn i OS lar oss bruke Golang til å aksessere nettverkskortet på serveren vår. Vi går da gjennom OS som har tilgang til hardware på maskinene. Dette kalles for abstrastraksjon der det er “laget” standardiserte grensesnitt som applikasjoner gjennom OS kan ta i bruk mange forskjellige typer hardware enheter. Da er vi også innom det som kalles for arbitration som er regelsett i OS for å få disse tilgangen til å styre hardware som NIC gjennom det som kalles for busser (bus).

HTML og CSS er standarder når det kommer til nettsider. Ettersom mange av verdens nettsider bruker disse språkene kan alle moderne nettlesere vise frem eller behandle sider kodet i HTML/CSS. Som følge har vi et standardisert grensesnitt på tvers av OS og maskinvare som i teori skal kunne bruke systemet.

Testing

Testing ble gjort ved hjelp av 'testing'-pakken i Go-biblioteket. Vi skrev positive og negative tester til funksjonene i webtjeneren.

Tester ble gjort på index-funksjonen for å se at korrekt statuskode (200) ble sendt ut.

getAndUnmarshal() ble testet ved å sjekke at variabelen ikke var tom. Den ble og sjekket ved at den korrekt returnerte *false* ved mislykket API-kall.

Vi testet runForecast() ved at den returnerte HTTP Statuskode 200 ved positiv input, og HTTP Statuskode 400 ved negativ input.

latLngFormat() ble testet ved å sjekke at stringen ikke hadde paranteser og mellomrom ved positiv input, og at stringen ble returnet uformatert ved negativ input.

Testen av joinUrlAndCoord() sjekket at den returnerte formaterte URLen innehold det den skulle, ved både positiv og negativ input.

Kjente problemer

Ved testing ble det oppdaget et par problemer som vi har slitt med å håndtere.

Det ene problemet er når det blir gjort API-kall med masse forskjellige symboler istedenfor skikkelige koordinater. Et eksempel er denne URLen:

https://api.darksky.net/forecast/a529911b60d81bab2c791732ad9ddf50/===9((&%#'%60%60#%C2%A4%agsdg?lang=nb&units=si

Det er verdt å merke seg at i vårt system, under normale omstendigheter, ville denne linken aldri ha blitt kalt.

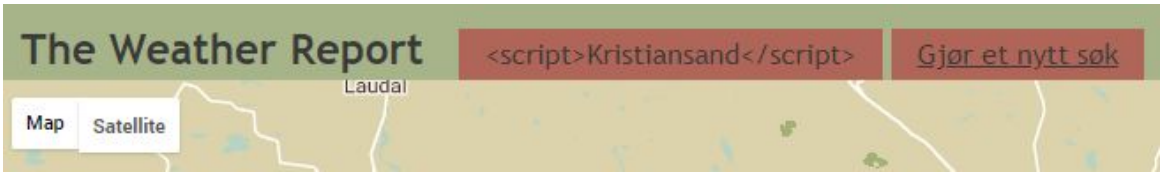
Et argument for å si at mye av testingen er "unødvendig", er at Google Maps APIet har et lag med abstraksjon som gjør at vi egentlig ikke trenger å tenke på feil brukerinput.

Google Maps tester at det vi søker på er et faktisk sted som finnes. Det vil si at under normale omstendigheter, vil vi alltid få inn korrekte koordinater.

Under onde omstendigheter, ved f.eks inspisering av kildekode og endringer i javascript, kunne nok "hackeren" fått gjennom koordinater som ikke er ordentlige koordinater. Det har vi gjort så godt som vi kan for å håndtere. Vi har sjekker som sjekker at koordinatene har i hvert fall nok tegn til å være faktiske koordinater, og at de inneholder de tegn som de skal inneholde.

Vi har også testet HTML-injection manuelt ved å skrive inn f.eks "<script>Kristiansand</script>" i søkefeltet på 'index.html'. Det viser seg at Google Maps ignorerer html-tags og bare søker på det som er inni taggen.

*Vi fikk samme resultat ved å skrive " Kristiansand *



Et kjent problem som vi har for øyeblikket er at brukerinput ligger i globale variabler når de blir hentet inn på tjeneren. La oss si at bruker #1 er inne og ser på værdata for ett sted, og bruker #2 søker på et annet sted. Hvis da bruker #1 laster siden inn på ny, får den værdataen for det bruker #2 søkte på. Vi innser at dette ikke er veldig bra løst, men vi fant ingen god løsning på problemet.

Vi slet også med å lage unit-tester til noen av funksjonene. Funksjonen getForm() trenger brukerinput i <form> tags. Vi prøvde å sende “test.html” til forespørselen i unit-testen, men vi fikk det ikke til. Unit-testen er kommentert ut i weatherServer_test.go.

Funksjonen executeTilbakemelding() var også vanskelig å teste ettersom vi ikke visste hvordan vi skulle sjekke hva som ble sendt ut med responsen. Istedenfor å bruke unit-test, måtte vi ty til manuell testing. Ved manuell testing fant vi noen problemer angående noen temperatur-tilfeller som ikke var tatt hånd om. Dette ble selvfølgelig fikset.

I tillegg til testing har vi prøvd å være grundige med å håndtere feil. For eksempel i runForecast(), sjekker tjeneren koordinatene, og hvis de ikke stemmer så kjører tjeneren en feilmelding til brukeren sammen med å skrive Statuskode 400 Bad Request i respons-headeren.