

SW3NGK

Besvarelse af opgave

Web.API

Afleveret: 17-12-21]

Afleveret af: Rasmus Baunvig Aagaard

Gruppe: 22

Deltagere i afleveringen

Studienummer	AU-id	Navn	Studieretning
201510642	AU532459	Rasmus Baunvig Aagaard	SW

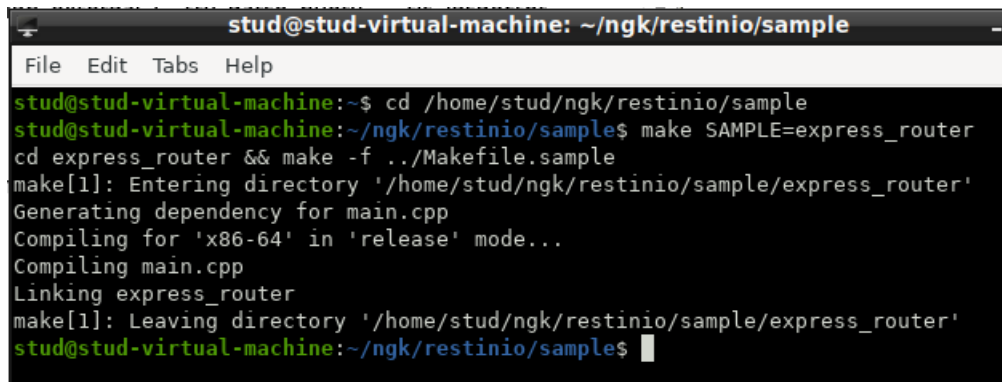
Indhold

Kompilation og opsætning af server	4
Del 1 - Web server	4
Indledning:.....	4
Design og implementering	5
Test og resultater.....	6
Konklusion	7
Del 2 - Web API.....	8
Indledning:.....	8
Delopgave 1 - Design og implementering	8
Delopgave 1 - Test og resultater	9
Delopgave 2 - Design og implementering	10
Delopgave 2 - Test og resultater	11
Delopgave 3 - Design og implementering	15
Delopgave 3 - Test og resultater	16
Del 3 - Web sockets	18
Indledning:.....	18
Delopgave 1 - Design og implementering	18
Delopgave 1 - Test og resultater	19
Delopgave 2 - Design og implementering	21
Delopgave 2 - Test og resultater	21

Figur 1 - make sample.....	4
Figur 2 - express_router	4
Figur 3 - Vejrinformation og værdier.....	4
Figur 4 - Requet handler RESTinio.....	5
Figur 5 - on_weather_data response	6
Figur 6 - Router "/"	6
Figur 7 - Start Webserver.....	6
Figur 8 - Webserveren tilgået i browser	7
Figur 9 - Webserver tilgået i Postman	7
Figur 10 - POST handler "/"	8
Figur 11 - POST 1.....	9
Figur 12 - POST 2.....	9
Figur 13 - POST 3.....	9
Figur 14 - GET singel param	10
Figur 15 - GET /date/:param.....	10
Figur 16 - GET "/id"	10
Figur 17 - GET "/"	11
Figur 18 - GET JSON	11
Figur 19 - GET "/3"	12
Figur 20 - GET "/4"	13
Figur 21 - GET "/5"	14
Figur 22 - GET "/date/04-12-2021"	14
Figur 23 - PUT handler	15
Figur 24 - PUT response.....	15
Figur 25 - PUT example 1.....	16
Figur 26 - GET example 1	16
Figur 27 - Database m. Vejr-information.....	17
Figur 28 - HTML socket	18
Figur 29 - JS Socket script	19
Figur 30 - HTML test event	19
Figur 31 - Event succes	20
Figur 32 - Route "/chat"	21
Figur 33 - registry handler	21

Kompilation og opsætning af server

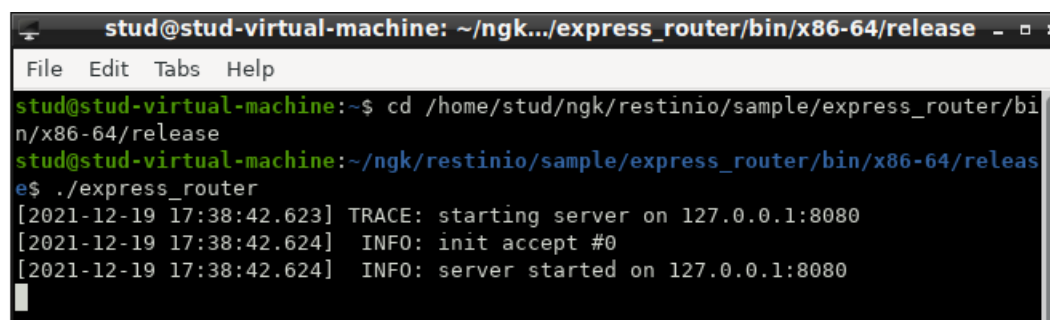
Benytter det opsatte "ngk" mappe li det ubuntu image som er udleveret til kurset: NGK. Der er til denne øvelse taget udgangspunkt i restinio og sample: "express_router". Dertilhørende makefile benyttes til at kompilere programmet.(Figur 1).



```
stud@stud-virtual-machine: ~/ngk/restinio/sample
File Edit Tabs Help
stud@stud-virtual-machine:~$ cd /home/stud/ngk/restinio/sample
stud@stud-virtual-machine:~/ngk/restinio/sample$ make SAMPLE=express_router
cd express_router && make -f ../Makefile.sample
make[1]: Entering directory '/home/stud/ngk/restinio/sample/express_router'
Generating dependency for main.cpp
Compiling for 'x86-64' in 'release' mode...
Compiling main.cpp
Linking express_router
make[1]: Leaving directory '/home/stud/ngk/restinio/sample/express_router'
stud@stud-virtual-machine:~/ngk/restinio/sample$
```

Figur 1 - make sample

Herefter starts serverne ved at køre applikationen express_router(Figur 2).



```
stud@stud-virtual-machine: ~/ngk.../express_router/bin/x86-64/release
File Edit Tabs Help
stud@stud-virtual-machine:~$ cd /home/stud/ngk/restinio/sample/express_router/bin/x86-64/release
stud@stud-virtual-machine:~/ngk/restinio/sample/express_router/bin/x86-64/release$ ./express_router
[2021-12-19 17:38:42.623] TRACE: starting server on 127.0.0.1:8080
[2021-12-19 17:38:42.624] INFO: init accept #0
[2021-12-19 17:38:42.624] INFO: server started on 127.0.0.1:8080
```

Figur 2 - express_router

Del 1 - Web server

Indledning:

I denne del skal der udarbejdes en webserver, som indeholder vejr-informationer som beskrevet nedenfor i Figur 3. Disse vej- informationer er, for nu, hardcoded værdier og Postman* benyttes til request disse data fra webserveren, og verificere at webserveren virker efter hensigten.

ID	1
Tidspunkt (dato og klokkeslæt)	
Dato	20211105
Klokkeslæt	12:15
Sted	
Navn	Aarhus N
Lat	13.692
Lon	19.438
Temperatur	13.1
Luftfugtighed	70%

Figur 3 - Vejrinformation og værdier

* Postman er en platform til at bygge og bruge API's og har indbyggede tools til eksempelvis at simplere et GET request fra webserveren.

Design og implementering

For at lave den ønskede webserver i c++ benyttes RESTinio Library og de tilhørende samples og makefiles udleverede og klargjort i det Ubuntu image, som benyttes til undervisningen.

RESTinio benyttes til at lave Request handlers og dertil hørende responses.

I main() køres `restinio::run`, hvor `weatherCollection` bestående af den ønskede vejr-informationer kaldes som request handler.

```
restinio::run(  
    restinio::on_this_thread< traits_t >()  
        .address( "localhost" )  
        .request_handler( server_handler( weatherCollection ) ) //collection  
        .read_next_http_message_timelimit( 10s )  
        .write_http_response_timelimit( 1s )  
        .handle_request_timeout( 1s ) );
```

Figur 4 - Request handler RESTinio

I request handleren, laves der et response (`resp`) til at håndtere "handle" requested.

Der laves et `req->create_response()` hvor der efterfølgende, i denne implementering, tilføjes teksten - "Collection of weather data: ".

Efterfølgende tilføjes så den hardcoded vejr-information ved brug af `json_dto::to_json()`, som konvertere vejr-information til JSON format.

Opsamling:

- 1) Der laves et request når en Client/Postman ønsker information fra Webservern
- 2) Request handleren laver et automatisk response
- 3) I `resp.set_body()` - tilføjes forklarende tekst
- 4) I `resp.append_body()` - tilføjes vejr-informationen konverteret til JSON format

Nedenfor i Figur 5, vises klassen `weather_handler_t` hvor ovenstående er implementeret.

```

// HANDLER
class weather_handler_t
{
public :
    explicit weather_handler_t( weather_station_t & weather )
        : m_weather( weather )
    {}

    weather_handler_t( const weather_handler_t & ) = delete;
    weather_handler_t( weather_handler_t && ) = delete;

    /*GET WEATHER DATA*/
    auto on_weather_data(
        const restinio::request_handle_t& req, rr::route_params_t ) const
    {
        auto resp = init_resp( req->create_response() );

        resp.set_body(
            "Collection of Weather data: \n" );

        for( std::size_t i = 0; i < m_weather.size(); ++i )
        {
            resp.append_body("\n Converted to json \n");
            resp.append_body(json_dto::to_json(m_weather[i]) + "\n");
        }

        return resp.done();
    }
}

```

Figur 5 - on_weather_data response

Dette er udelukkende et response til at håndtere requests betegnet on_weather_data, hvilket styres af en router. Routeren håndterer de forskellige typer requests til Webservern og vælger hvilken handler der skal besvare disse. I Figur 6 vises hvordan auto on_weather_data response handleren ovenfor er valgt til at håndtere http GET request til webserveren hvor path "/" er valgt.

```

// Handlers for '/' path.
router->http_get( "/", by( &weather_handler_t::on_weather_data ) );

```

Figur 6 - Router "/"

Test og resultater

Filen kompiles og webserveren startes.

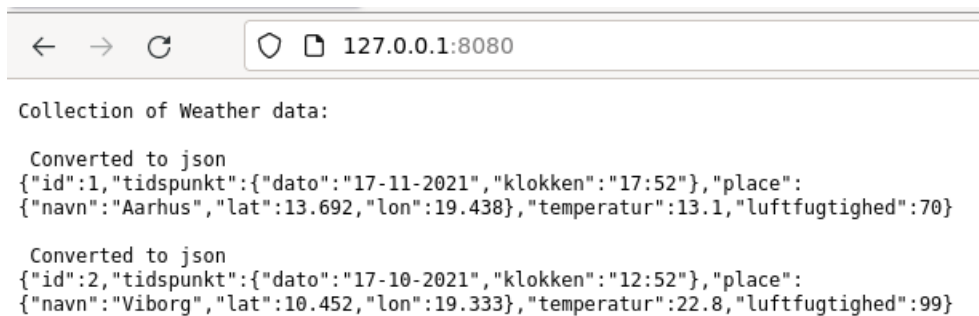
```

stud@stud-virtual-machine:~/ngk/restinio/sample/express_router/bin/x
e$ ./express_router
[2021-12-04 12:44:19.033] TRACE: starting server on 127.0.0.1:8080
[2021-12-04 12:44:19.033] INFO: init accept #0
[2021-12-04 12:44:19.033] INFO: server started on 127.0.0.1:8080

```

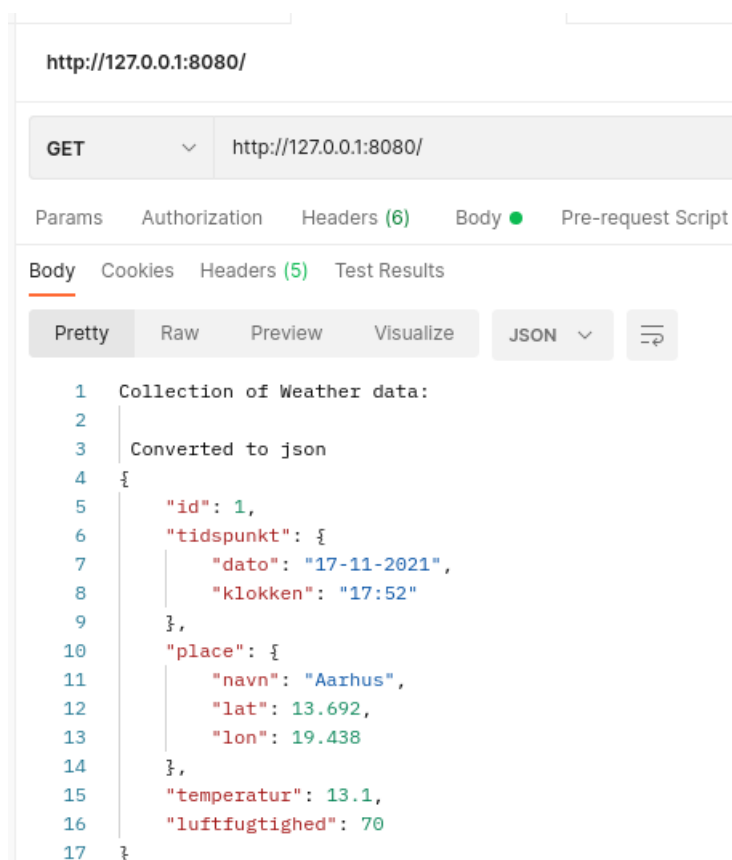
Figur 7 - Start Webserver

Nu tilgås Webserveren først fra en browser og resultatet er vist nedenfor i Figur 8. Der noteres at koden vises i JSON format og ikke visuelt pænt, men indeholder værdierne svarende til Figur 3.



Figur 8 - Webserveren tilgået i browser

Efterfølgende benyttes Postman og resultatet af GET Request, i JSON, format vises nedenfor i Figur 9.



Figur 9 - Webserver tilgået i Postman

Konklusion

Det er lykkedes at lave en Webserver med de hardcoded værdier for den ønskede vejr-information. Disse værdier kan vises til en client både med request fra browser og Postman. Ydermere laver handleren et response hvor vejr-information er konverteret til JSON format.

Del 2 - Web API

Indledning:

Der skal udarbejdes en web API, således at en Client kan oprette (POST) og opdaterer (PUT) vejrdato. Webserveren skal håndtere disse requests og sende det ønskede response af vejr-information til clienten.

Delopgave 1 - Design og implementering

Ved brug af web API skal en Client oprette 3 nye sæt af vejrdato, med samme dato, men forskellige tidspunkter og vejr-information.

Clienten skal kunne lave POST requests til serveren, og Routeren vil håndtere dette med handleren `on_new_weather_data`.

```
// Handlers for '/' path.  
router->http_get( "/", by( &weather_handler_t::on_weather_data ) );  
router->http_post( "/", by( &weather_handler_t::on_new_weather_data ) );
```

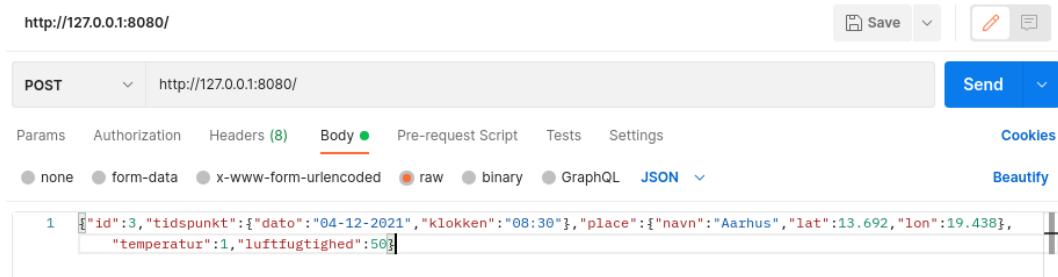
Figur 10 - POST handler "/"

Response handler `on_new_weather_data` laver et response baseret på request body og sender det til Clienten.

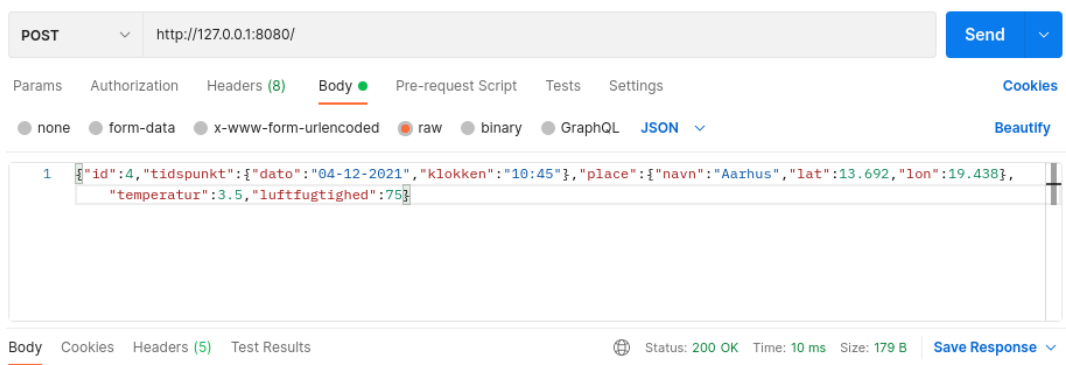
```
auto on_new_weather_data(  
    const restinio::request_handle_t& req, rr::route_params_t )  
{  
    auto resp = init_resp( req->create_response() );  
  
    try  
    {  
        m_weather.emplace_back(  
            json_dto::from_json< weather_t >( req->body() ));  
    }  
    catch( const std::exception & /*ex*/ )  
    {  
        mark_as_bad_request( resp );  
    }  
  
    return resp.done();  
}
```


Delopgave 1 - Test og resultater

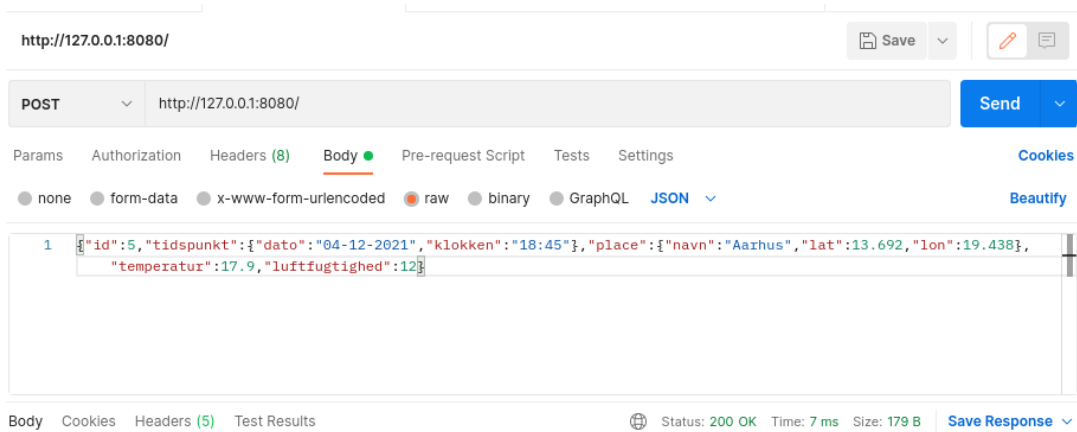
Der laves en POST request i Postman, hvor body data tilføjes svarende til det ønskede vejr data. Datoen fastlægges til 04-12-2021 hvorimod tidspunkt og vejrinformation varieres. Nedenfor vises 3 POST af nye vejr-information til Webserveren.



Figur 11 - POST 1



Figur 12 - POST 2



Figur 13 - POST 3

Delopgave 2 - Design og implementering

Nu verificeres at de nye vejrdato er oprettet og en Client kan request disse vejr-informationer som i øvelsens Del 1, med hardcoded værdier.

Udover handleren til GET, som benyttet tidligere laves en handler til at håndtere en single parameter, her date.

```
// Handler for '/single/:param' path.  
router->http_get( "/date/:param", by( &weather_handler_t::on_date_get ) );
```

Figur 14 - GET singel param

Og response handleren on_date_get laver et response med de vejr-information hvor netop date parameteren matcher. Implementering af dette er vist nedenfor i

```
auto on_date_get(  
    const restinio::request_handle_t& req, rr::route_params_t params )  
{  
    auto resp = init_resp( req->create_response() );  
    try  
    {  
        auto date = restinio::utils::unescape_percent_encoding( params[ "param" ] );  
  
        resp.set_body( "Weather from " + date + ":\n" );  
        for( std::size_t i = 0; i < m_weather.size(); ++i )  
        {  
            const auto & b = m_weather[ i ];  
  
            if( date == b.m_tidspunkt.m_dato )  
            {  
                resp.append_body("\n Converted to json \n");  
                resp.append_body(json_dto::to_json(m_weather[i]) + "\n");  
            }  
        }  
    }  
    catch( const std::exception & )  
    {  
        mark_as_bad_request( resp );  
    }  
  
    return resp.done();  
}
```

Figur 15 - GET /date/:param

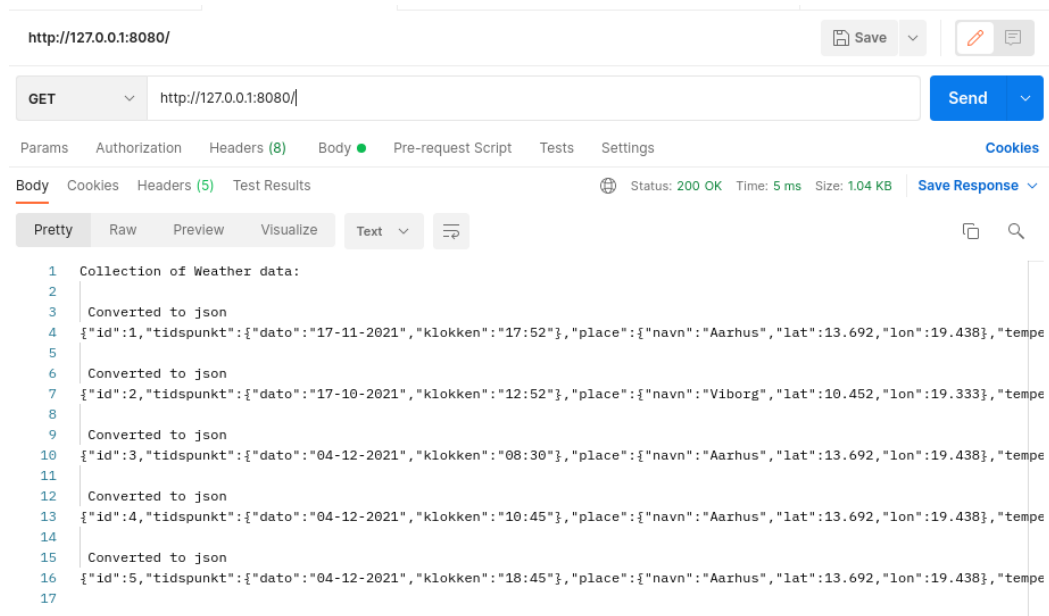
Der er ligeledes en handler til GET af "id"

```
// Handlers for '/:datanum' path.  
router->http_get(  
    R"(/:datanum(\d+))",  
    by( &weather_handler_t::on_weather_get ) );
```

Figur 16 - GET "/id"

Delopgave 2 - Test og resultater

Først laves GET i Postman, uden parameter og de oprettede vejr-information med id= 3, 4 og 5 er nu en del af server response.



Figur 17 - GET "/"

Og vises dette som JSON format(Figur 18). Der noteres at dato er ens, men tidspunkt og vejrdato forskelligt.

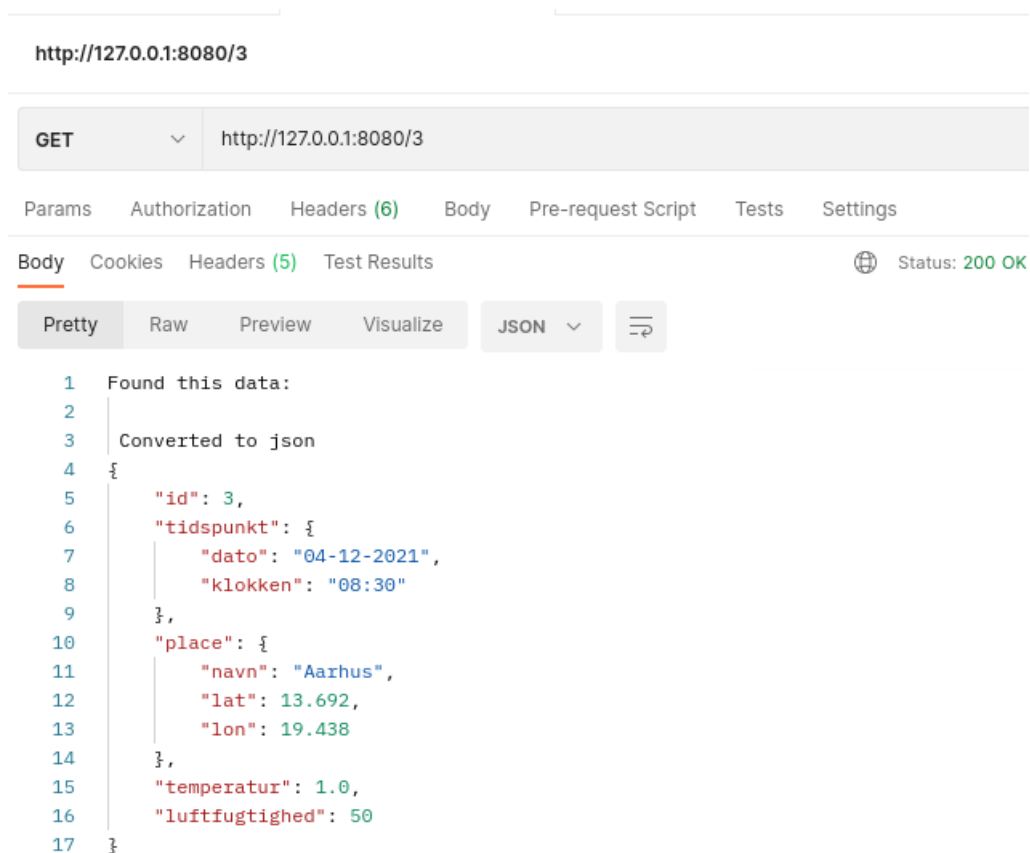
```
35 | Converted to json
36 | {
37 |   "id": 3,
38 |   "tidspunkt": {
39 |     "dato": "04-12-2021",
40 |     "klokken": "08:30"
41 |   },
42 |   "place": {
43 |     "navn": "Aarhus",
44 |     "lat": 13.692,
45 |     "lon": 19.438
46 |   },
47 |   "temperatur": 1.0,
48 |   "luftfugtighed": 50
49 | }

51 | Converted to json
52 | {
53 |   "id": 4,
54 |   "tidspunkt": {
55 |     "dato": "04-12-2021",
56 |     "klokken": "10:45"
57 |   },
58 |   "place": {
59 |     "navn": "Aarhus",
60 |     "lat": 13.692,
61 |     "lon": 19.438
62 |   },
63 |   "temperatur": 3.5,
64 |   "luftfugtighed": 75
65 | }

67 | Converted to json
68 | {
69 |   "id": 5,
70 |   "tidspunkt": {
71 |     "dato": "04-12-2021",
72 |     "klokken": "18:45"
73 |   },
74 |   "place": {
75 |     "navn": "Aarhus",
76 |     "lat": 13.692,
77 |     "lon": 19.438
78 |   },
79 |   "temperatur": 17.9,
80 |   "luftfugtighed": 12
81 | }
```

Figur 18 - GET JSON

I Postman laves GET parametriserede ved id for de forskellige vejr-informationer.



Figur 19 - GET "/3"

http://127.0.0.1:8080/4

GET http://127.0.0.1:8080/4

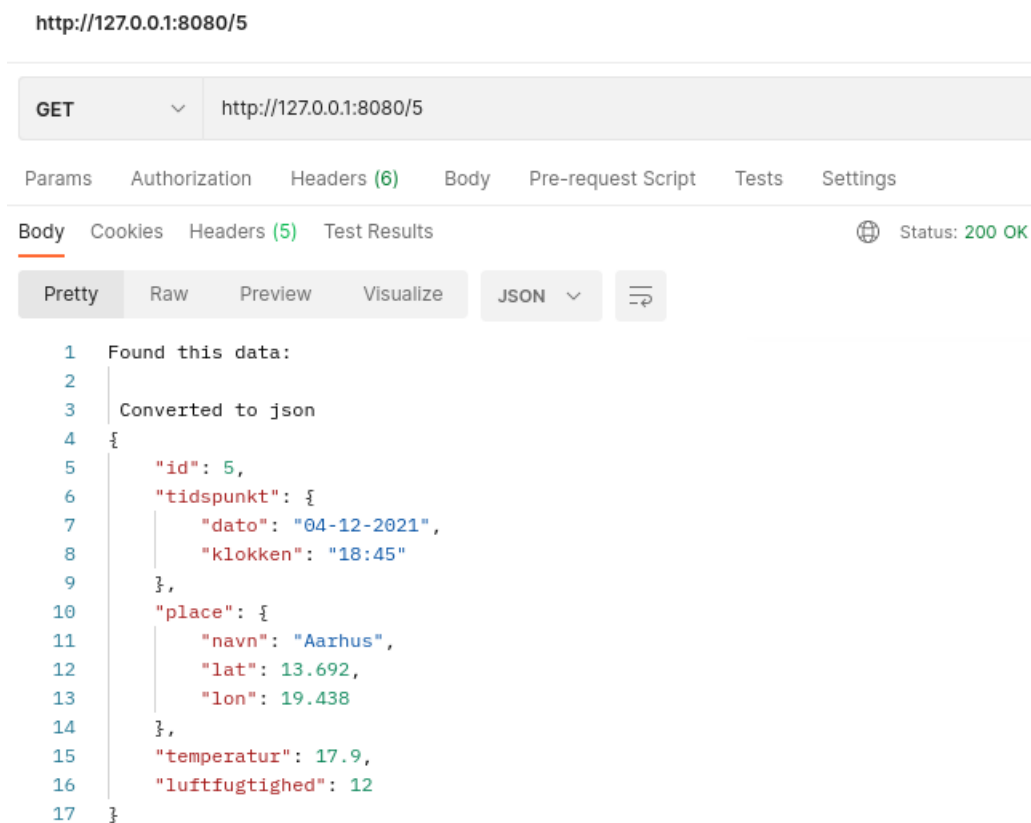
Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

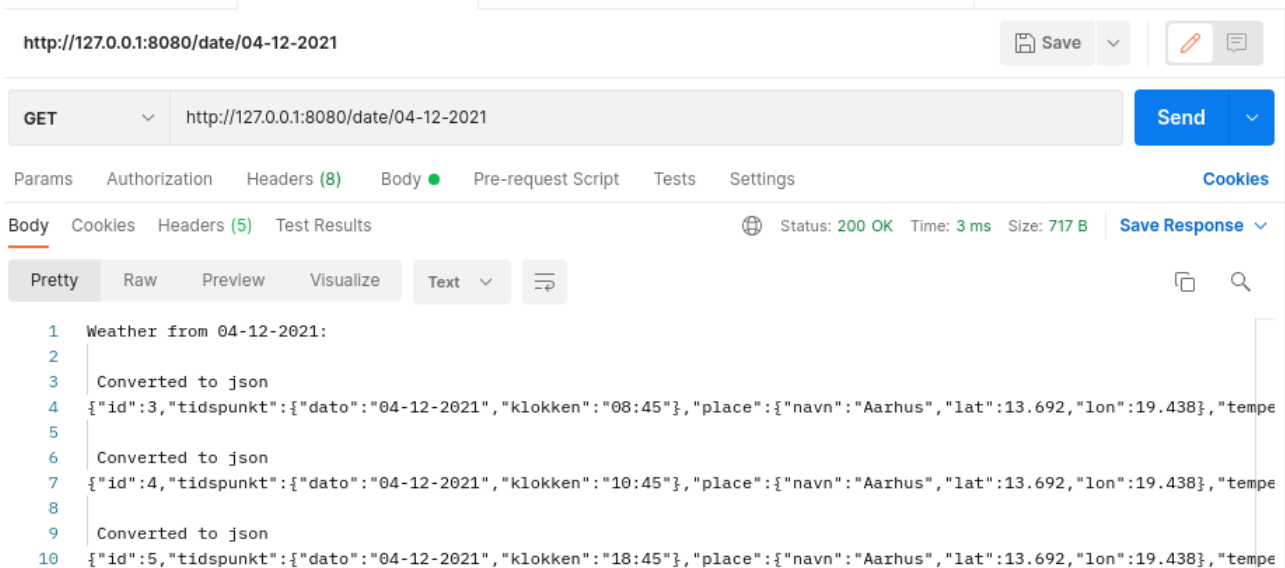
```
1 Found this data:
2
3 Converted to json
4 {
5     "id": 4,
6     "tidspunkt": {
7         "dato": "04-12-2021",
8         "klokken": "10:45"
9     },
10    "place": {
11        "navn": "Aarhus",
12        "lat": 13.692,
13        "lon": 19.438
14    },
15    "temperatur": 3.5,
16    "luftfugtighed": 75
17 }
```

Figur 20 - GET "/4"



Figur 21 - GET "/5"

Ligeledes i Postman laves request på den dato "04-12-2021" som de nye vejr-informationer blev oprettet med. Nedenfor, vist som tekst, ses de oprettede datasæt med den ønskede dato.



Figur 22 - GET "/date/04-12-2021"

Delopgave 3 - Design og implementering

Ligeledes som med POST og GET håndteres PUT af Routeren og her benyttes en handler `on_weather_update`

```
router->http_put(  
    R"(/:datanum(\d+))",  
    by( &weather_handler_t::on_weather_update ) );
```

Figur 23 - PUT handler

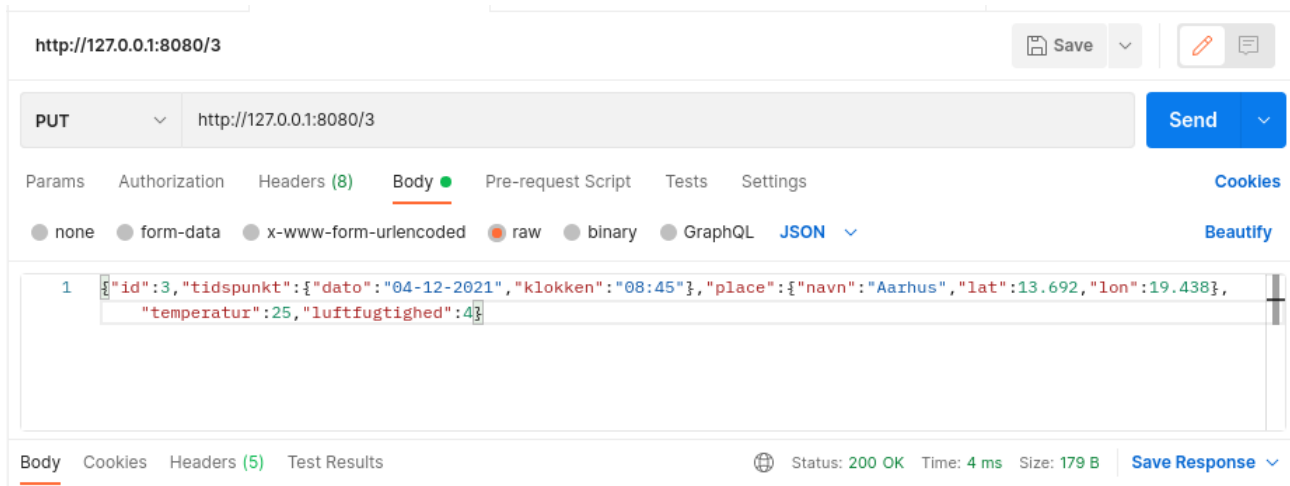
Og lige som med POST laves der et response baseret på `req->body` og implementeringen vises herunder.

```
auto on_weather_update(  
    const restinio::request_handle_t& req, rr::route_params_t params )  
{  
    const auto datanum = restinio::cast_to< std::uint32_t >( params[ "datanum" ] );  
  
    auto resp = init_resp( req->create_response() );  
  
    try  
    {  
        auto b = json_dto::from_json< weather_t >( req->body() );  
  
        if( 0 != datanum && datanum <= m_weather.size() )  
        {  
            m_weather[ datanum - 1 ] = b;  
        }  
        else  
        {  
            mark_as_bad_request( resp );  
            resp.set_body( "No weather data with #" + std::to_string( datanum ) + "\n" );  
        }  
    }  
    catch( const std::exception & /*ex*/ )  
    {  
        mark_as_bad_request( resp );  
    }  
  
    return resp.done();  
}
```

Figur 24 - PUT response

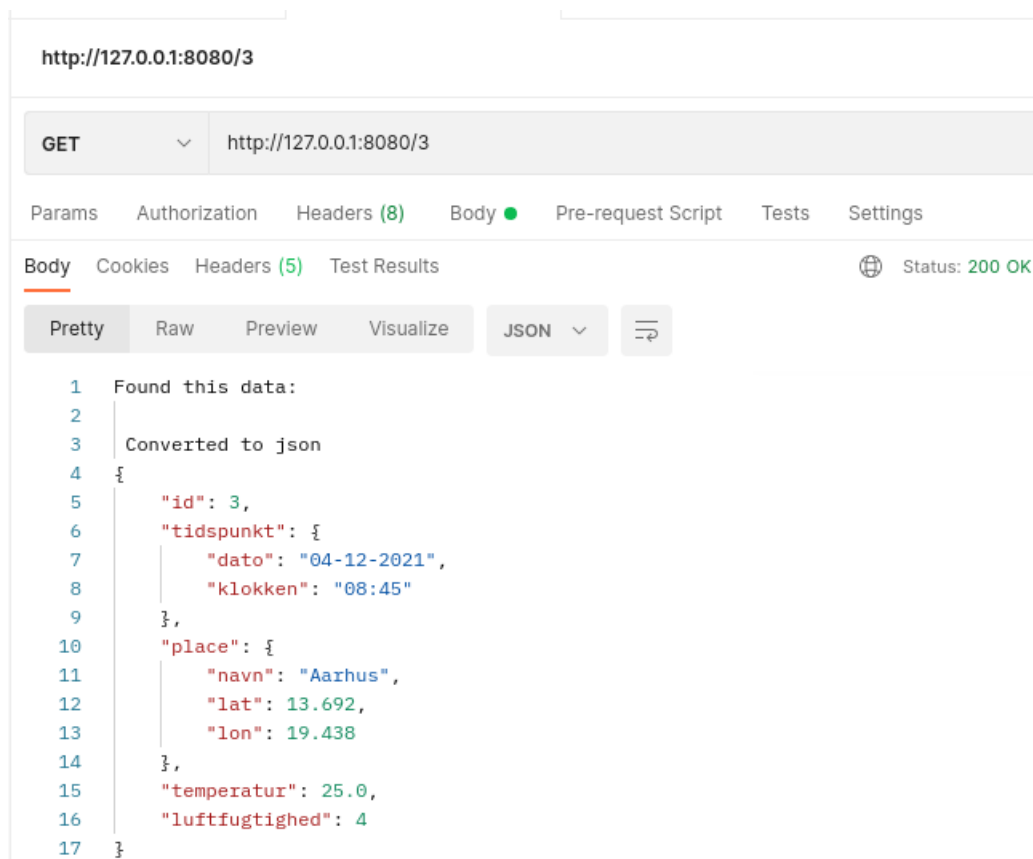
Delopgave 3 - Test og resultater

Postman benyttes igen og vejr-informationen for id =3, opdateres med nye værdier.



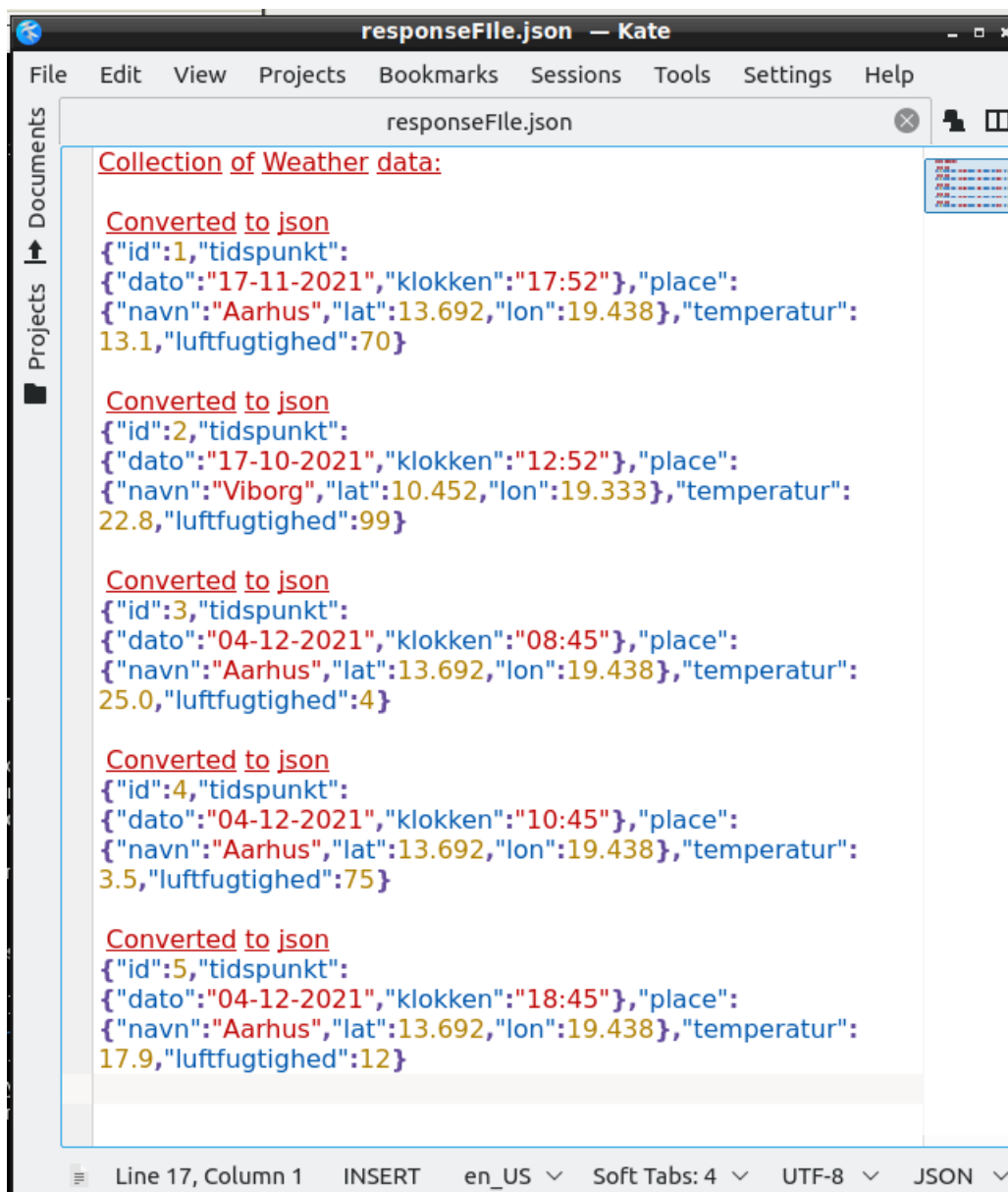
Figur 25 - PUT example 1

Ændringen af klokkeslæt, temperatur og luftfugtighed undersøges med GET og `/3` parameteren for id. Response til Client er som ønsket og med de opdatere værdier som vist nedenfor.



Figur 26 - GET example 1

De nye vejr-informationer og data oprettet af Client gemmes i JSON format.



The screenshot shows a text editor window titled "responseFile.json - Kate". The editor displays a JSON array of five weather data objects. Each object is preceded by a red heading "Converted to json". The JSON is color-coded: strings are in blue, numbers in yellow, and keys in black. The status bar at the bottom indicates "Line 17, Column 1", "INSERT" mode, "en_US" locale, "Soft Tabs: 4", "UTF-8" encoding, and "JSON" format.

```
Collection of Weather data:

Converted to json
{"id":1,"tidspunkt":
{"dato":"17-11-2021","klokken":"17:52"},"place":
{"navn":"Aarhus","lat":13.692,"lon":19.438},"temperatur":
13.1,"luftfugtighed":70}

Converted to json
{"id":2,"tidspunkt":
{"dato":"17-10-2021","klokken":"12:52"},"place":
{"navn":"Viborg","lat":10.452,"lon":19.333},"temperatur":
22.8,"luftfugtighed":99}

Converted to json
{"id":3,"tidspunkt":
{"dato":"04-12-2021","klokken":"08:45"},"place":
{"navn":"Aarhus","lat":13.692,"lon":19.438},"temperatur":
25.0,"luftfugtighed":4}

Converted to json
{"id":4,"tidspunkt":
{"dato":"04-12-2021","klokken":"10:45"},"place":
{"navn":"Aarhus","lat":13.692,"lon":19.438},"temperatur":
3.5,"luftfugtighed":75}

Converted to json
{"id":5,"tidspunkt":
{"dato":"04-12-2021","klokken":"18:45"},"place":
{"navn":"Aarhus","lat":13.692,"lon":19.438},"temperatur":
17.9,"luftfugtighed":12}
```

Figur 27 - Database m. Vejr-information

EN VIDEO DOKUMENTATION AF RESULTATERNE ER VEDLAGT JOURNALEN.

Del 3 - Web sockets

Indledning:

Det skal være muligt for clients at få opdateringer fra serveren. For at lave en simpel applikation for dette vil der i første implementering af dette sendes en "Hello besked" fra serveren til en client, som vil opdatere enHTML side.

Delopgave 1 - Design og implementering

Et simpel design til html filen vises herunder i Figur 28. Der ses et script "clientSocket.js" som beskrives efterfølgende.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Weather station </title>
  <script src="clientSocket.js"></script>
</head>
<body>
  <h1>Weather Station</h1>
  <h2>Weather information</h2>

  <div>
    <p id="test"></p>

  </div>

</body>
</html>
```

Figur 28 - HTML socket

Det inkluderede javascript vises nedenfor i Figur 29. Der ses at der benyttes et 'open', function(event), som vil sende en "Hello Server" besked. Denne besked skal så indsættes i html filen <p> tagget med id="test".

```
const socket = new WebSocket('ws://localhost:8080/chat');

// Connection opened

socket.addEventListener('open', function (event) {

// Sending a message to the web socket server...

socket.send('Hello Server!');

});

// Listen for messages

socket.addEventListener('message', function (message) {
```

```
console.log('Message from server ', message.data);

// DOM

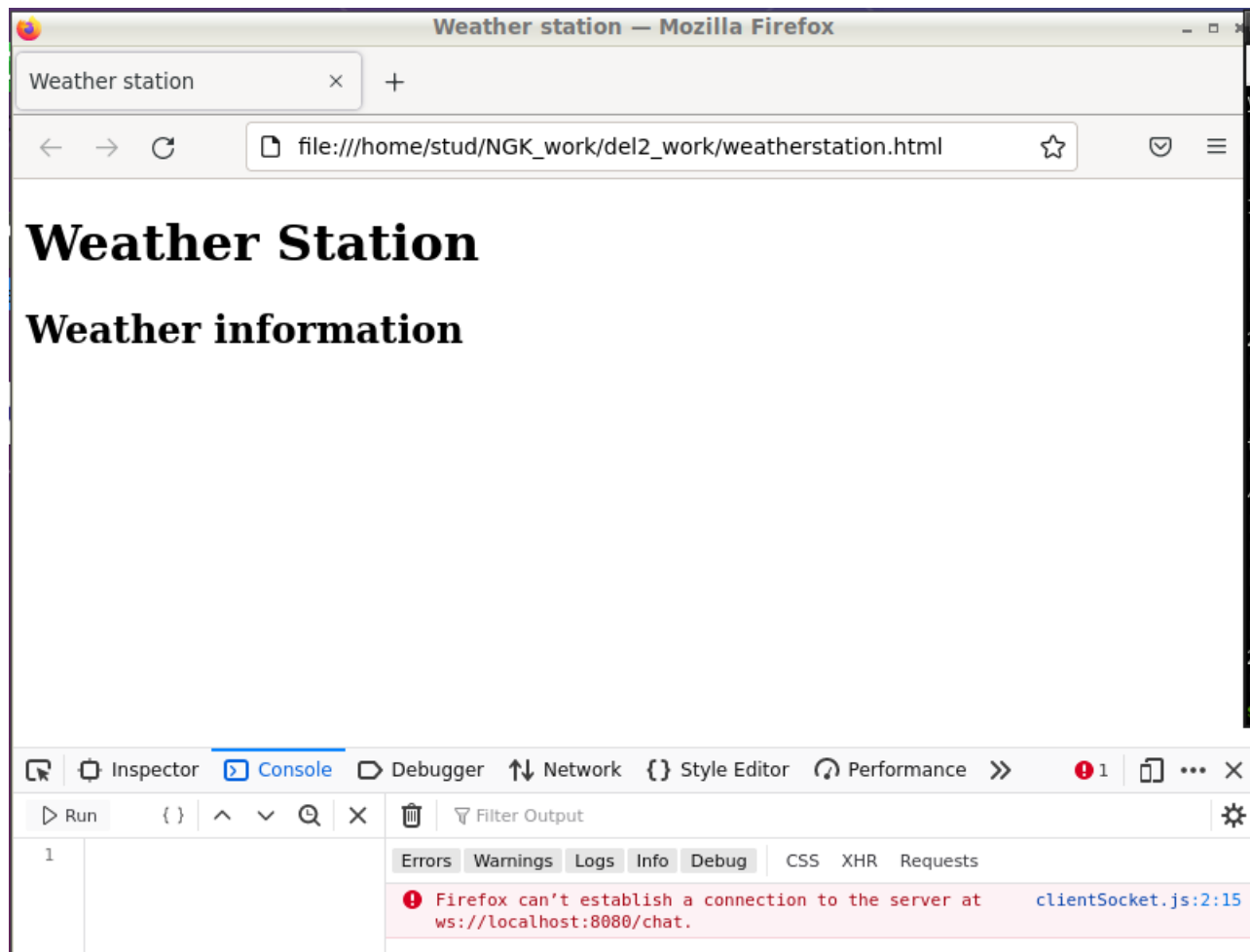
document.getElementById("test").innerHTML = message.data;

});
```

Figur 29 - JS Socket script

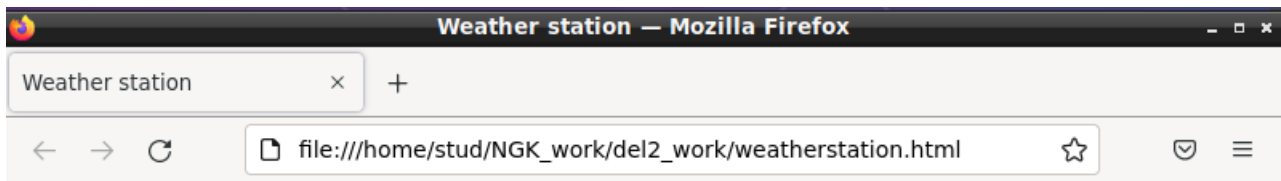
Delopgave 1 - Test og resultater

Html siden åbnes i browser. Her ses (Figur 30) endnu ingen besked, intet event.



Figur 30 - HTML test event

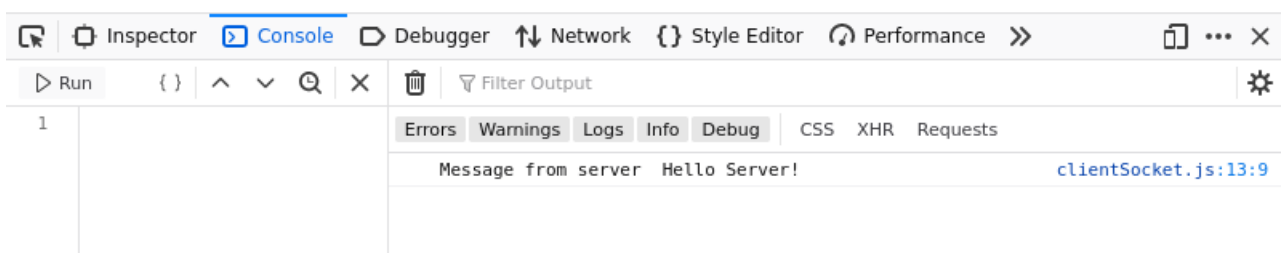
Når server connection så oprettes(Figur 31), event sendes, læses besked til client, og beskeden vises på en opdateret html side. Der ses ved "inspect" at der i console skrives:" Message from server Hello Server!



Weather Station

Weather information

Hello Server!



Figur 31 - Event succes

Det er altså lykkedes at et event opdatere indholdet på siden i realtime.

EN VIDEO AF DETTE ER VEDLAGT JOURNALEN.

Delopgave 2 - Design og implementering

Nu ønskes en integration af implementering i delopgave1 med implementeringen af Webserveren fra tidligere opgave, således det er Vejrinformationer der opdateres ved event som <p> tagget blev i førtes delopgave.

Der oprettes en router "/ chat"(Figur 32).

```
// Router websocket
router->http_get("/chat", by( &weather_handler_t::request_handler ) );
```

Figur 32 - Route "/chat"

Der tilføjes en ny response (Figur 33) til weatherStation.

```
auto request_handler(
    const restinio::request_handle_t& req, rr::route_params_t params )
{
    ws_registry_t registry;
    if( restinio::http_connection_header_t::upgrade == req->header().connection() )
    {
        auto wsh =
            rws::upgrade< traits_t >(
                *req,
                rws::activation_t::immediate,
                [ &registry ]( auto wsh, auto m ){
                    if( rws::opcode_t::text_frame == m->opcode() ||
                        rws::opcode_t::binary_frame == m->opcode() ||
                        rws::opcode_t::continuation_frame == m->opcode() )
                    {
                        wsh->send_message( *m );
                    }
                    else if( rws::opcode_t::ping_frame == m->opcode() )
                    {
                        auto resp = *m;
                        resp.set_opcode( rws::opcode_t::pong_frame );
                        wsh->send_message( resp );
                    }
                    else if( rws::opcode_t::connection_close_frame == m->opcode() )
                    {
                        registry.erase( wsh->connection_id() );
                    }
                } );

        registry.emplace( wsh->connection_id(), wsh );
        init_resp(req->create_response()).done();
        return restinio::request_accepted();
    }

    return restinio::request_rejected();
}
```

Figur 33 - registry handler

Delopgave 2 - Test og resultater

Det er ikke lykkedes at implementerer delopgave 1 med tidligere udviklede WEB api således den ønskede funktionalitet opfyldes.