JavaScript in the Browser

Niels Olof Bouvin

data → Web page

- A recurrent task when building a Web site is making data into Web pages
- We will today look at various techniques to effect this transformation

Overviews

- **Live updates with WebSockets**
- **Getting data from a sensor**
- **Dynamically generated content**

Pushing updates out to Web clients

- Last time, we added DELETE functionality to the Greetings site, so we can delete greetings on the server, and remove the matching greeting on the Web page without having to reload the entire page
- We can do better!
- Additions and deletions should be updated live across all our users' browsers

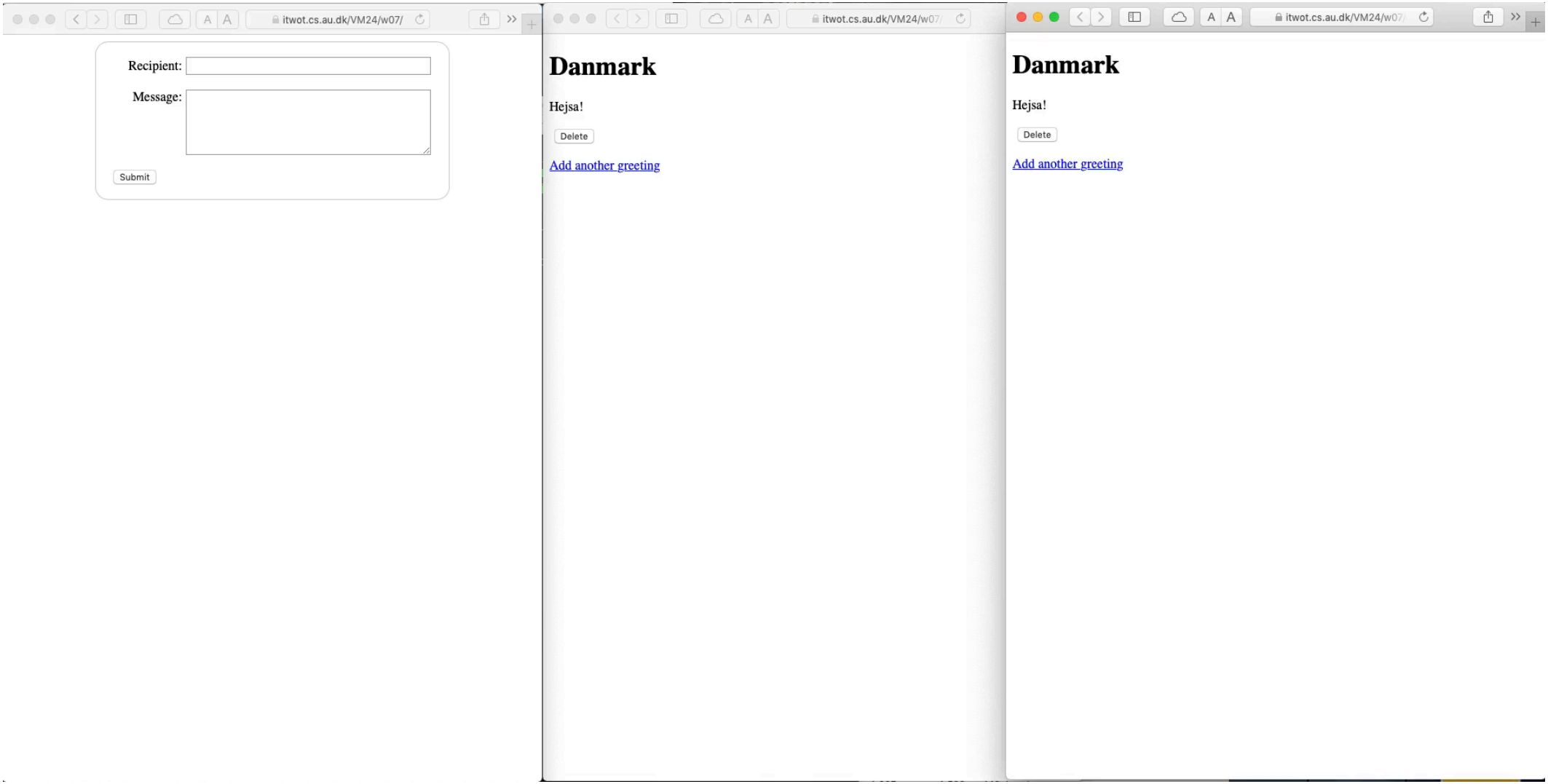
Discovering changes

- **XMLHttpRequest makes a connection, sends/receives some data, and then closes**
  - good for making a change, or for inquiring about data on the server
  - but not well suited for discovering that a change has been made on the server
- **We could, in principle, write a function that would retrieve the greetings once every minute and update the Web page, if there were changes**
  - not very efficient, if there are no changes
  - very laggy, if there are changes—we might have to wait a minutes before the update

WebSockets

- **WebSockets open a connection to the server, and *keeps it open* (until closed)**
- **Messages can be sent back and forth between the client and the server**
  - Without the overhead of making a HTTP connection

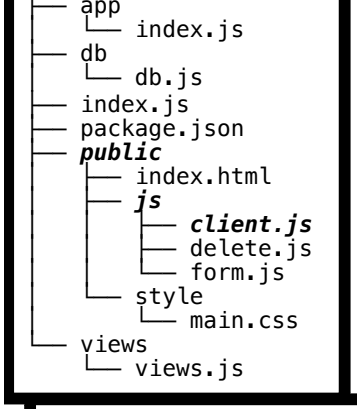
Greetings in three Web browsers



- Additions and deletions are updates immediately

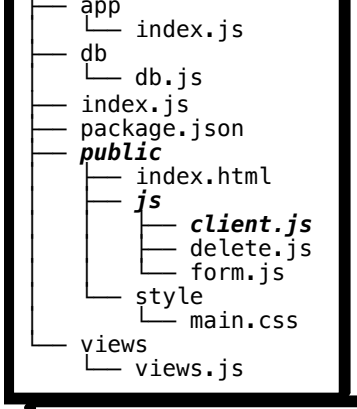
General idea

- We want to be able to keep the existing functionality, but add the following:
- We should be able to receive a ‘delete’ operation from the server (with the ID of the greeting to be deleted)
- We should be able to receive a ‘create’ operation from the server (with the Recipient and Message of the new greeting)



Set up: client.js 1/2

```
const connection = new WebSocket('wss://itwot.cs.au.dk/VM24/wsa');

connection.onopen = event => {
  console.log('WebSocket is open now.');
};

connection.onclose = event => {
  console.log('WebSocket is closed now.');
};

connection.onerror = event => {
  console.error('WebSocket error observed:', event);
};

connection.onmessage = event => {
 console.log('Received over WSS: ', JSON.parse(event.data));
  receivedCommand(JSON.parse(event.data));
};
```

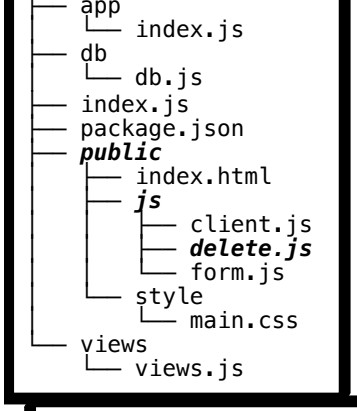


Action: client.js 2/2

```
function receivedCommand (message) {
  switch (message.operation) {
    case 'delete':
      deleteThisGreetingOnPage(message.argument.id);
      break;

    case 'create':
      addThisGreetingToPage(message.argument);
      break;
  }
}
```

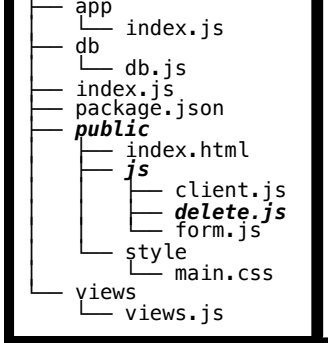
```
{ operation: “create”,
  argument: {
    id: 56,
    recipient: “Norge”,
    message: “Hei!”
  }
}
```

```
{ operation: “delete”,
  argument: {
    id: 54
  }
}
```



Deleting a greeting: delete.js 1/2

```
const deleteButtons = document.querySelectorAll('.delete');
const greetings = document.querySelector('#greetings');

function deleteThisGreetingOnServer (e) {
  const id = e.target.attributes['data-id'].value;
  const requestURL = `${document.URL}/${id}`;
  const request = new XMLHttpRequest();
  request.addEventListener('load', function () {
    deleteThisGreetingOnPage(id);
  });
  request.open('DELETE', requestURL);
  request.send();
}
function deleteThisGreetingOnPage (id) {
  const selector = `#greeting-${id}`;
  const theGreeting = document.querySelector(selector);
  if (theGreeting)
    {greetings.removeChild(theGreeting);}
  else
    {console.log('Trying to delete the deleted')};
}
```

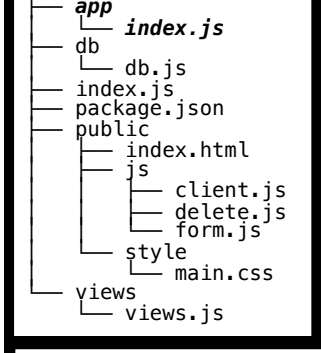


Adding a greeting: delete.js 2/2

```
function addThisGreetingToPage (greeting) {
  const selector = `#greeting-${greeting.id}`;
  const theGreeting = document.querySelector(selector);
  if (theGreeting) {
    console.log('The greeting already exists!');
  } else {
    const div = document.createElement('div');
    div.setAttribute('id', `greeting-${greeting.id}`);
    const h1 = document.createElement('h1');
    h1.textContent = greeting.recipient;
    div.appendChild(h1);
    const p = document.createElement('p');
    p.textContent = greeting.message;
    div.appendChild(p);
    const button = document.createElement('button');
    button.setAttribute('data-id', greeting.id);
    button.setAttribute('class', 'delete');
    button.textContent = 'Delete';
    button.addEventListener('click', deleteThisGreetingOnServer);
    div.appendChild(button);
    greetings.appendChild(div);
  }
}
```

WebSockets on the server

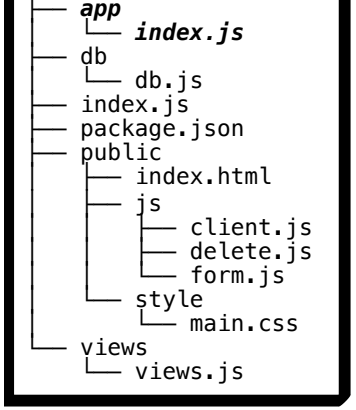
- **Just as with a** http **or Express server, we have to setup a WebSocket server to listen on a specific port**
- **Whereas http(s) connections are closed, when we have sent or received the last data, we want to keep our WebSockets open as long as there is a Web page somewhere with our page open**
- **But, that requires us to do a little bit of work to keep the connection open**
  - this kind of traffic is known as 'ping/pong' traffic, and is supported in the protocol



WS setup and use: index.js 1/3

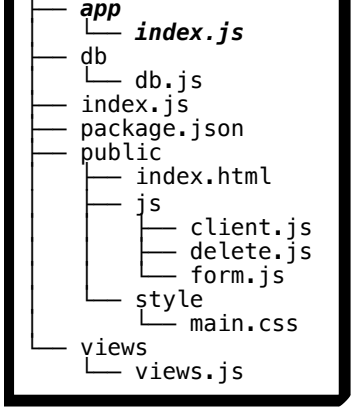
```
const WebSocket = require('ws');

const wsPort = 2000;

const wss = new WebSocket.Server({
  port: wsPort
});

wss.on('connection', ws => {
 ws.isAlive = true;
  ws.on('pong', heartbeat);
});

function heartbeat () {
  this.isAlive = true;
}
```

```
const interval = setInterval(() => {
  wss.clients.forEach(function each (ws) {
    if (ws.isAlive === false) {
      return ws.terminate();
    }
    ws.isAlive = false;
    ws.ping();
  });
}, 30000);

wss.on('close', () => {
  clearInterval(interval);
});

function sendToWSClients (data) {
  wss.clients.forEach(client => {
    if (client.readyState === WebSocket.OPEN){
      client.send(data);
    }
  });
}
```



Creation: index.js 2/3

```
app.post('/greetings', (request, response, next) => {
  const greeting = {
    recipient: request.body.recipient,
    message: request.body.message
  };
  Greetings.create(greeting, (err, results) => {
    if (err) return next(err);
    response.redirect('greetings');
    sendToWSClients(JSON.stringify({
      operation: 'create',
      argument: {
        id: results.insertId,
        recipient: request.body.recipient,
        message: request.body.message
      }
    }));
  });
});
```



Deletion: index.js 3/3

```
app.delete('/greetings/:id', (request, response, next) => {
  const id = request.params.id;
  Greetings.delete(id, err => {
    if (err) return next(err);
    response.status(200);
    response.end();
    sendToWSClients(JSON.stringify({
      operation: 'delete',
      argument: {
        id: id
      }
    }));
  });
});
```

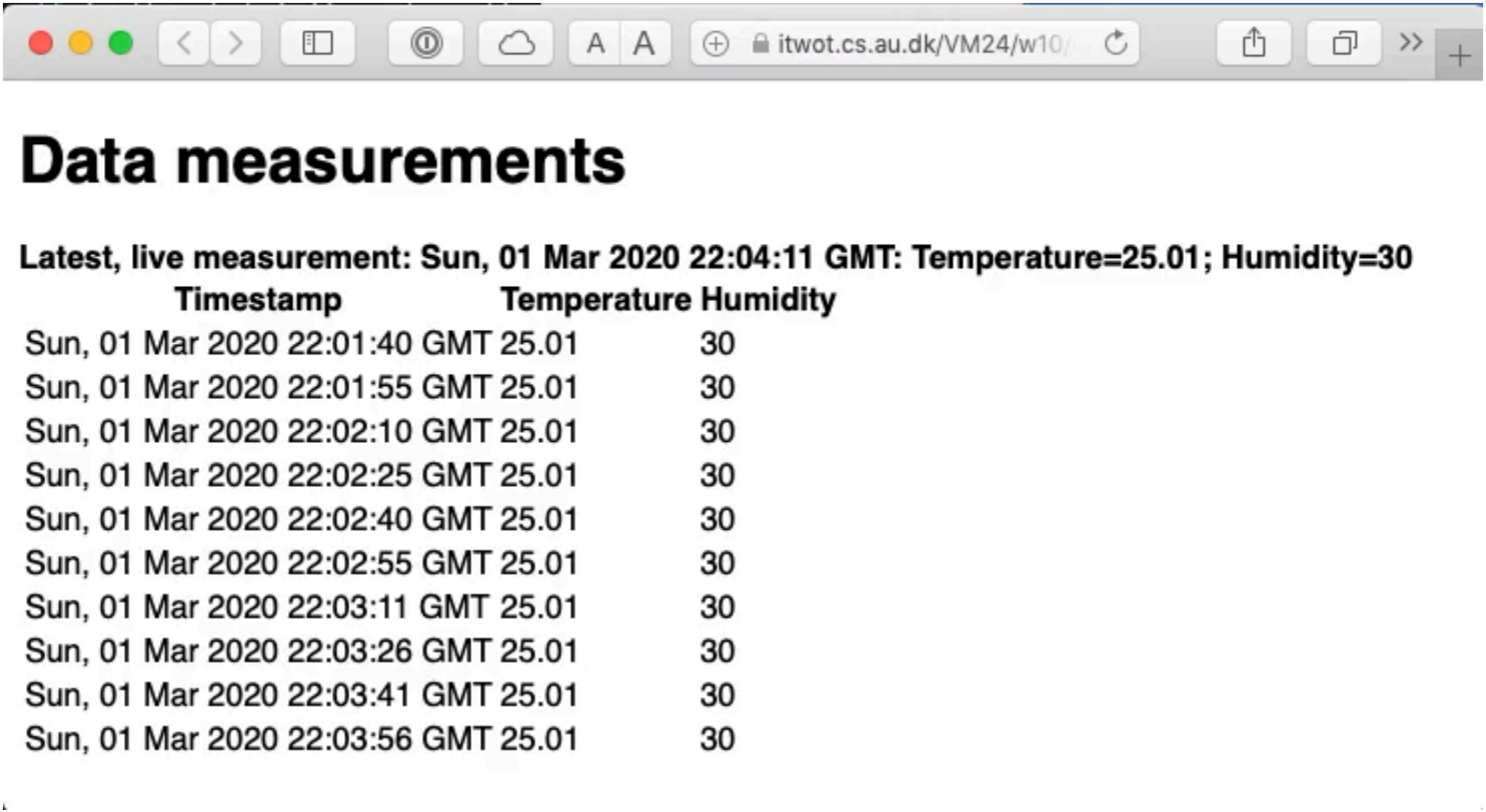
Overviews

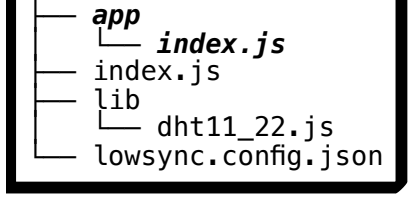
- Live updates with WebSockets
- **Getting data from a sensor**
- **Dynamically generated content**

Getting data off the Neonious One

- The Neonious One can, as we have seen, be hooked up to sensors
- How do we get hold of that data?
- We do not want to connect directly to the NO, because it cannot handle a lot of traffic
- Better to have the NO report to a server, and then talk to the server
  - as the server can be *much* stronger

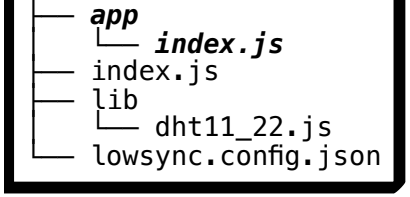
Getting live data off the Neonious One





Measuring temperatur/humidity

```
const axios = require('axios');
const DHT11_22 = require('../lib/dht11_22.js');
const sensor = new DHT11_22(25);

sensor.start((err, temp, humidity) => {
  if (err) console.log(err);
  // you may get some errors due to bad connection or checksums
  else {
    temp = Math.round(temp * 100) / 100;
    humidity = Math.round(humidity * 100) / 100;
    console.log(`Temperature:\t${temp}℃`);
    console.log(`Humidity:\t${humidity}%`);
    const data = {
      timestamp: Date.now(),
      temperature: temp,
      humidity: humidity
    };
    postMeasurements(data);
  }
}, 15000);
```



Sending data with axios

```
function postMeasurements (data) {
  axios
    .post('https://itwot.cs.au.dk/VM24/w10/measurements', data)
    .then(response => {
      console.log('Status:', response.status);
    })
    .catch(error => {
      console.log(error);
    });
}
```

- axios **is a npm module (lightweight enough for the NO), which uses *promises* to handle asynchronous operations**

Promises?

- **You have been using callbacks so far in the course, whenever you needed to deal with the result of a computation**
  - but what if that function also has a callback
    - and that function also has a callback
      - *and that function also has a callback*
        - *and that function also has a callback*
          - …
- **Then you are in what in JavaScript is known as 'callback hell' and that can be a bit of a mess, because it is hard to keep track of**

The pattern of a callback

- **Please Do Thing:**
  - if there was an error, then either throw or handle error
  - if there was no error, process the result

- **Nearly all callbacks look like this**
- **This has led to the development of *Promises***

Reading a file with a callback

```
'use strict’;

const fs = require(‘fs');

fs.readFile('message.txt', 'utf8', (err, data) => {
  if (err) throw (err); // handle any errors
  console.log(data);    // do the thing
});
```

- This should be quite familiar by now

Reading from a file with a Promise

```
'use strict’;

const fs = require(‘fs');

function promiseToReadFile (fileName) {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, 'utf8', (err, data) => {
      if (err) reject(err)
      else resolve(data)
    });
  });
}

promiseToReadFile('message.txt')
.then(text => console.log(text))
.catch(error => console.log('someone erred:', error))
```

- **A Promise take two arguments, both functions**
- **A Promise calls** reject **if there is an error, or** resolve **if things went well**
- **We specify those two using** .then **and** .catch

util.promisify **makes it simpler**

```
'use strict';

const fs = require('fs');
const util = require('util');
const readFile = util.promisify(fs.readFile);

readFile('message.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => {
    throw (err)
  });
```

- util.promisify **assumes a function that takes a callback function of the form (err, result) ⇒ { … }**

```
├── app
│   └── index.js
├── index.js
├── lib
│   └── dht11_22.js
└── lowsync.config.json
```

Sending data with axios

```
function postMeasurements (data) {
  axios
    .post('https://itwot.cs.au.dk/VM24/w10/measurements', data)
    .then(response => {
      console.log('Status:', response.status);
    })
    .catch(error => {
      console.log(error);
    });
}
```

- Axios **sends, by default, data encoded as JSON**
  - so we have keep that in mind, as we process the received data

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── js
│   │   └── client.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

Receiving the data: setup 1/2

```
'use strict';
const path = require('path');
const WebSocket = require('ws');
const express = require('express');
const bodyParser = require('body-parser');

const views = require('../views/views');
const process = require('process');
const app = express();
const port = 4500;
const wsPort = 2000;
const wss = new WebSocket.Server({
  port: wsPort
});

const measurements = [];
app.use(bodyParser.json());

app.use((request, response, next) => {
  console.log(`request.url=${request.url}`);
  next();
});

app.use(express.static(path.join(__dirname, '../public')));
```

```
data-logger
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── js
│   │   └── client.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

Receiving the data: handling 2/2

```
app.post('/measurements', (request, response, next) => {
  const data = request.body;
  console.log(data);
  measurements.push(data);
  response.status(200);
  response.send();
  sendToWSClients(JSON.stringify(data));
  next();
});

app.get('/measurements', (request, response, next) => {
  response.send(views.measurementsView(measurements));
  next();
});

app.listen(port, err => {
  if (err) return console.error(`An error occurred: ${err}`);
  console.log(`Listening on http://localhost:${port}/`);
});
```

The structure of a table

```
<table>
    <thead>
      <tr>
        <th>Heading 1</th>
        <th>Heading 2</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Data</td>
        <td>Data</td>
      </tr>
      <tr>
        <td>Data</td>
        <td>Data</td>
      </tr>
    </tbody>
  </table>
```

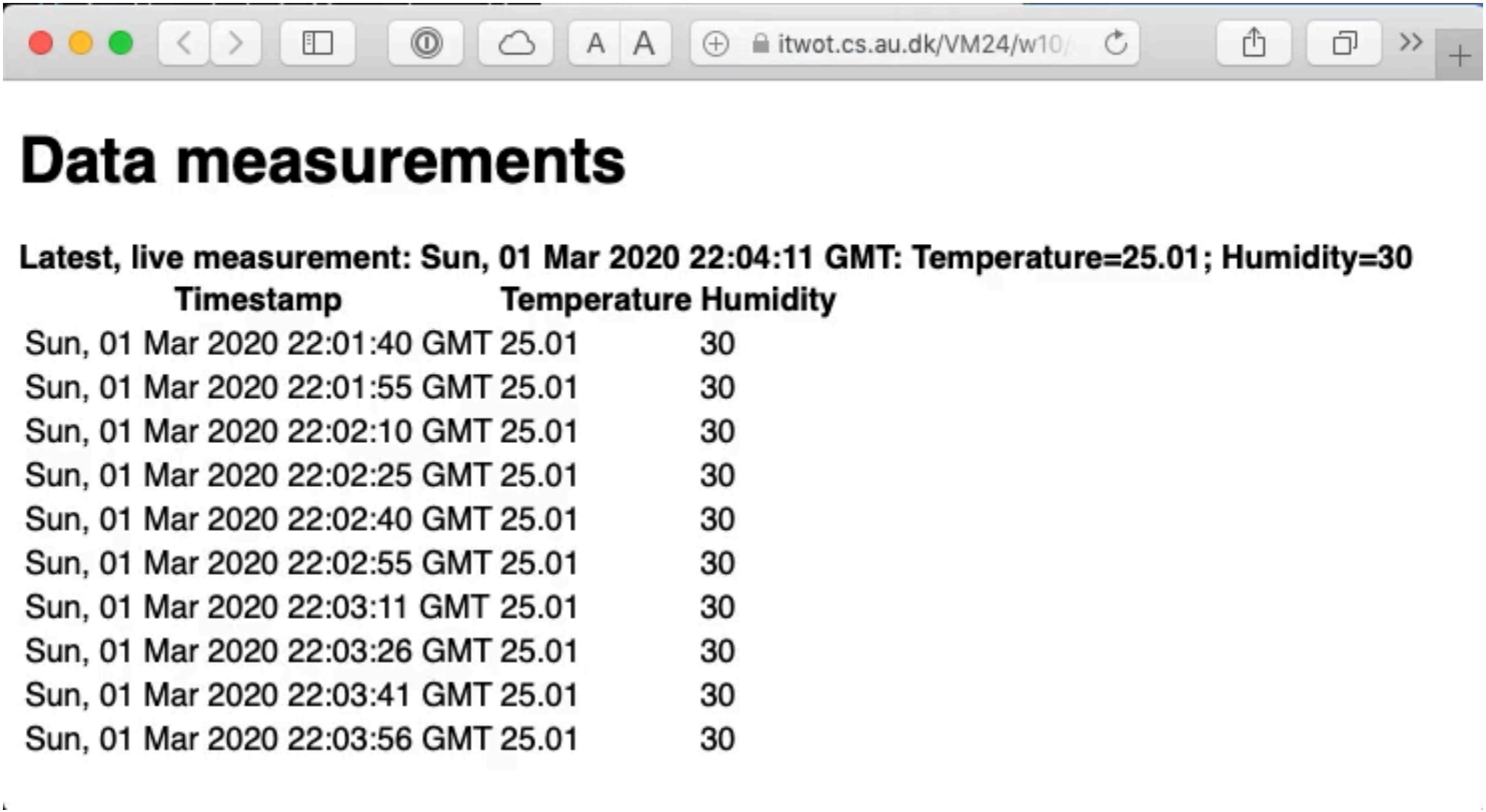
Heading 1	Heading 2
Data	Data
Data	Data

- Rows defined by <tr>, cell contents defined by <td>

Generating a data table

```
exports.measurementsView = function measurements (measurements = []) {
  return page(
    'Data measurements',
    h1('Data measurements') +
    b('Latest, live measurement: ' +
      span('No measurements received yet', {
        id: 'latest'
      })
    ) +
    table(
      thead(
        tr(th('Timestamp') + th('Temperature') + th('Humidity'))
      ) +
      tbody(
        measurements.reduce((acc, measurement) => {
          acc +=
            tr(
              td(new Date(measurement.timestamp).toUTCString()) +
              td(measurement.temperature) +
              td(measurement.humidity)
            );
          return acc;
        }, '')
      )
    )
  );
};
```

Just to remind you



Overviews

- Live updates with WebSockets
- Getting data from a sensor
- **Dynamically generated content**

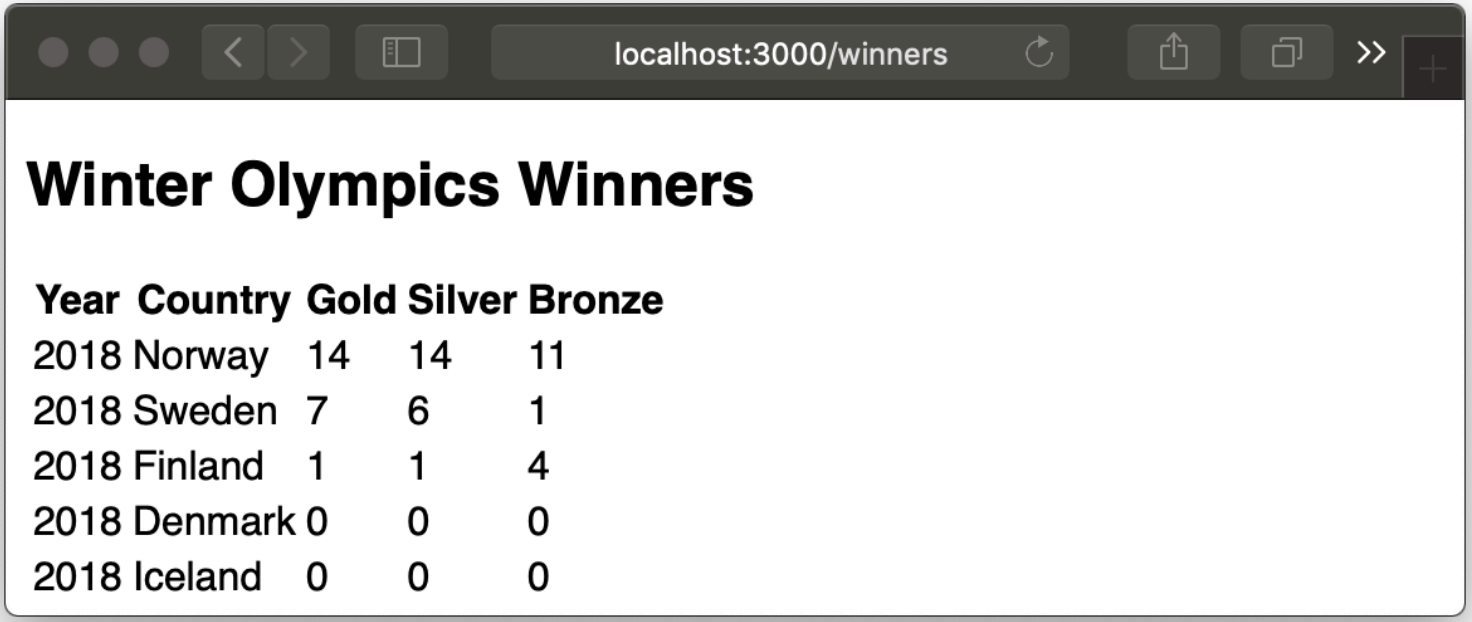
A new use case

- This time, let's create a site that summarises the Winter Olympics 2018 from the perspective of the Nordic countries…

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

The Mighty Dataset

```
{"wolympics":[
  {"year":2018,"country":"Norway","gold":14,"silver":14,"bronze":11},
  {"year":2018,"country":"Sweden","gold":7,"silver":6,"bronze":1},
  {"year":2018,"country":"Finland","gold":1,"silver":1,"bronze":4},
  {"year":2018,"country":"Denmark","gold":0,"silver":0,"bronze":0},
  {"year":2018,"country":"Iceland","gold":0,"silver":0,"bronze":0}
]}
```



Different content generation methods

- **Given data, we can generate suitable representations**
  - On the server
  - In the Web browser
- **We can choose different kinds of representations of the same data**
  - A HTML table
  - Some visualisation of the same data

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

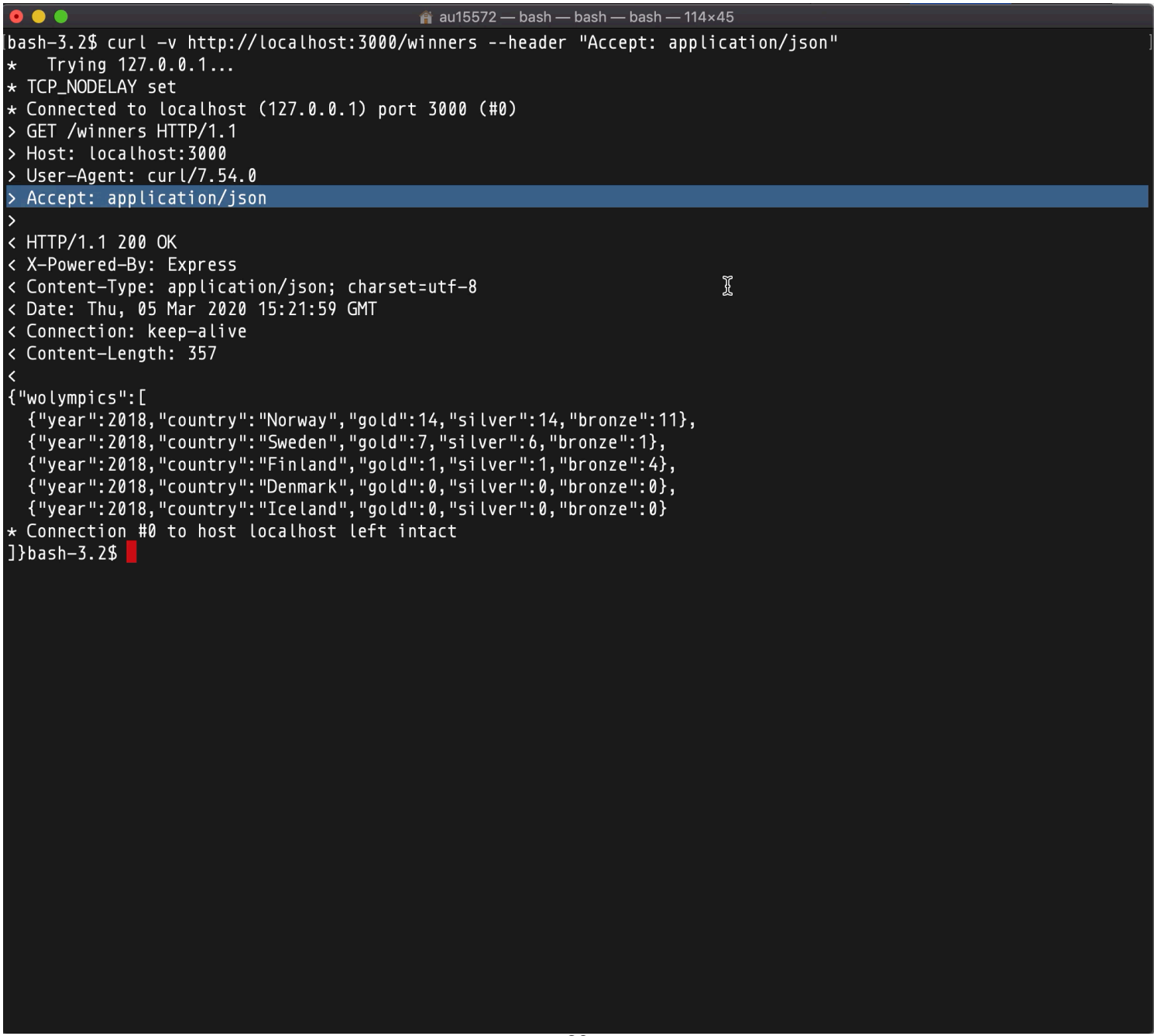
Accepting JSON or HTML: index.js

```
'use strict';
const path = require('path');
const fs = require('fs');
const express = require('express');
const views = require('../views/views');
const app = express();
const port = 3000;
const data = fs.readFileSync(
  path.join(__dirname, '../public/data/wolympics.json')
);

app.use(express.static(path.join(__dirname, '../public')));

app.get('/winners', (request, response, next) => {
  if (request.accepts('application/json') && !request.accepts('text/html'))
{
    response.contentType('application/json');
    response.end(data);
  } else {
    response.end(views.winnerView(JSON.parse(data).wolympics));
  }
});
```

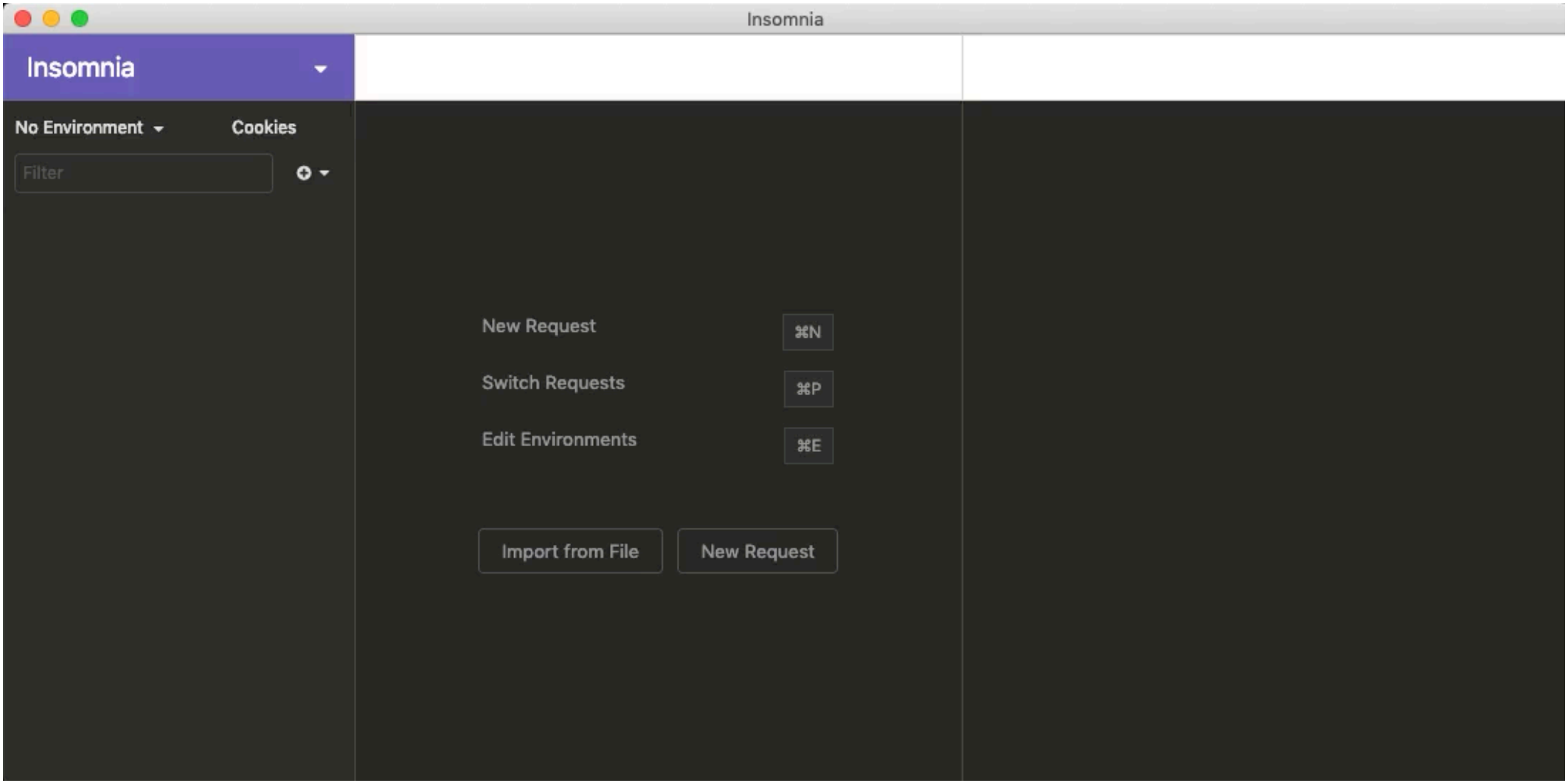
request.accepts()

- **A Web browser/client can signify to the server which formats it would prefer (or can handle) through the `Accept:` header**
  - Certain image/video formats, etc
- **Web browsers typically can accept very many things, so we have to be very specific, when checking for something specific**

The effect of the Accept header



The effect of the Accept header



https://insomnia.rest

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

Two ways to make the Web page

Winter Olympics Winners

Year	Country	Gold	Silver	Bronze
2018	Norway	14	14	11
2018	Sweden	7	6	1
2018	Finland	1	1	4
2018	Denmark	0	0	0
2018	Iceland	0	0	0

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

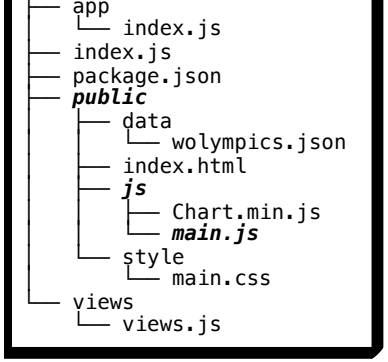
Browser-side content generation

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Olympic Winners</title>
    <link
      rel="stylesheet"
      type="text/css"
      media="screen"
      href="style/main.css"
    />
    <script src="js/Chart.min.js"></script>
    <script defer src="js/main.js"></script>
  </head>

  <body>
    <div id="medalTable"></div>
…
```

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

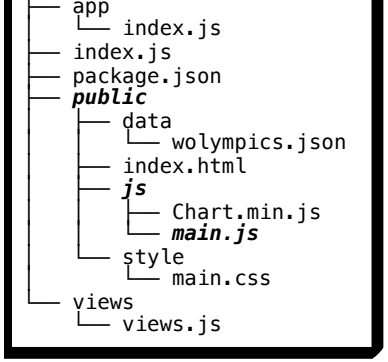
Retrieving JSON: main.js 1/4

```
'use strict';
/* global XMLHttpRequest Chart */
const medalTable = document.querySelector('#medalTable');
const medalChartCtx = document.querySelector('#medalChart');
const dkMedals = document.querySelector('#dkMedals');

if (medalTable) {
  medalTable.appendChild(h2('Winter Olympics Winners'));
  const request = new XMLHttpRequest();
  const requestURL = `${document.URL}winners`;
  request.onload = () => {
    if (request.status === 200) {
      const winners = JSON.parse(request.responseText);
      medalTable.appendChild(table(winners.wolympics));
      const chartData = transformToChartData(winners.wolympics);
      createStackedHBarChart(chartData);
    }
  };
  request.open('GET', requestURL);
  request.setRequestHeader('Accept', 'application/json');
  request.send();
}
```



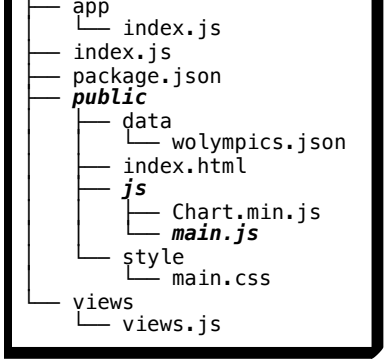
Building HTML: main.js 2/4

```
function h2 (text = '') {
  const h2 = document.createElement('h2');
  h2.textContent = text;
  return h2;
}

function td (text = '') {
  const td = document.createElement('td');
  td.textContent = text;
  return td;
}

function th (text = '') {
  const th = document.createElement('th');
  th.textContent = text;
  return th;
}
```



Building HTML: main.js 3/4

```
function table (entries = []) {
  const table = document.createElement('table');

  const thead = document.createElement('thead');
  thead.appendChild(tr(entries[0], 'head'));
  table.appendChild(thead);

  const tbody = document.createElement('tbody');
  for (const line of entries) {
    tbody.appendChild(tr(line));
  }
  table.appendChild(tbody);

  return table;
}
```



Building HTML: main.js 4/4

```
function tr (line = {}, type = 'body') {
  const tr = document.createElement('tr');
  switch (type) {
    case 'head':
      for (const name of Object.keys(line)) {
        tr.appendChild(th(name));
      }
      break;

    case 'body':
    default:
      for (const name of Object.keys(line)) {
        tr.appendChild(td(line[name]));
      }
      break;
  }
  return tr;
}
```

JSON → different representations

- While transforming JSON to HTML is pretty straightforward, we can do more!
- How about a responsive, graphical representation?

Adding a bit of visualisation

- Data represented in a table can, of course, be highly informative
- But a visualisation will often make the point come clearer across
- Happily, there are *many* different JavaScript frameworks aimed at visualisation
  - D3, plot.ly, Vega (lite), Observable, Stardust, Bokeh, Google Charts, . . .
- We'll take a brief look at Charts.js, which is pretty straightforward to use

Data transformation

- **Visualisation framework invariantly will require data in a specific format**
  - and often not quite the way your data is organised
- **This requires a bit of formatting, such as transposing or extracting elements**

The Dataset transformed


```
{"wolympics":[
  {"year":2018,"country":"Norway","gold":14,"silver":14,"bronze":11},
  {"year":2018,"country":"Sweden","gold":7,"silver":6,"bronze":1},
  {"year":2018,"country":"Finland","gold":1,"silver":1,"bronze":4},
  {"year":2018,"country":"Denmark","gold":0,"silver":0,"bronze":0},
  {"year":2018,"country":"Iceland","gold":0,"silver":0,"bronze":0}
]}
```


transformToChartData(…)


```
{
  "labels": ["Norway", "Sweden", "Finland", "Denmark", "Iceland"],
  "gold": [14, 7, 1, 0, 0],
  "silver": [14, 6, 1, 0, 0],
  "bronze": [11, 1, 4, 0, 0]
}
```

```
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

Transforming the data

```
function transformToChartData (medals = []) {
  const labels = [];
  const gold = [];
  const silver = [];
  const bronze = [];
  for (const c of medals) {
    labels.push(c.country);
    gold.push(c.gold);
    silver.push(c.silver);
    bronze.push(c.bronze);
  }
  return { labels, gold, silver, bronze };
}
```

```
medals-summary-dynamic
├── app
│   └── index.js
├── index.js
├── package.json
├── public
│   ├── data
│   │   └── wolympics.json
│   ├── index.html
│   ├── js
│   │   ├── Chart.min.js
│   │   └── main.js
│   └── style
│       └── main.css
└── views
    └── views.js
```

Installing Chart.js

- **Quite simple:**
  - npm install chart.js --save
- **Then copy** Chart.min.js **from** node_modules/chart.js/dist/ **to** public/js
- **Check http://www.chartjs.org for more information**

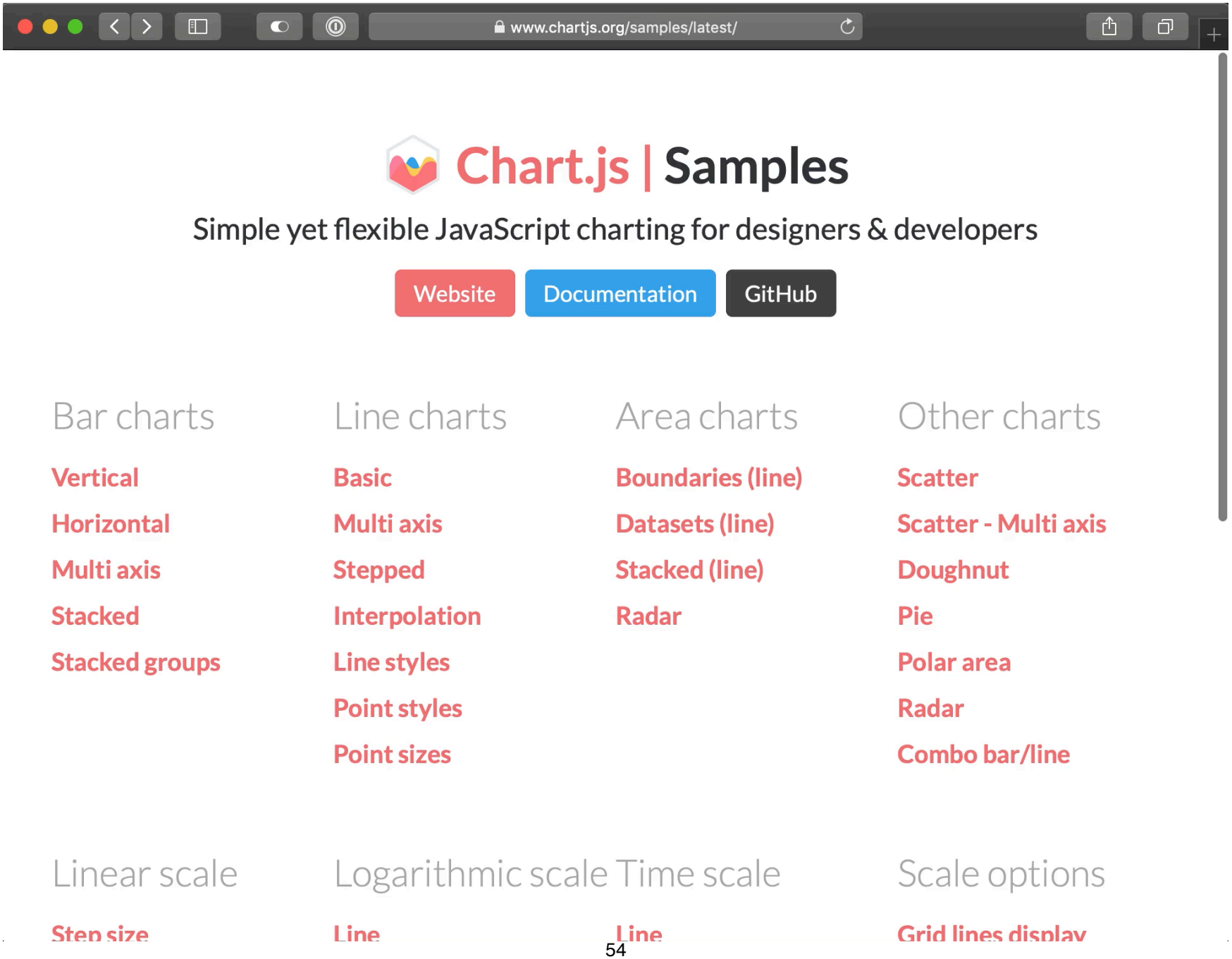
How does Chart.js work?

- Chart.js works by drawing on the *canvas*, which is a (2D) bitmapped surface element
- It creates a chart by calls of the form

```
const stackedBars = new Chart(canvasContext, {
  type: 'horizontalBar',
  data: { … }
  options: { … }
})
```

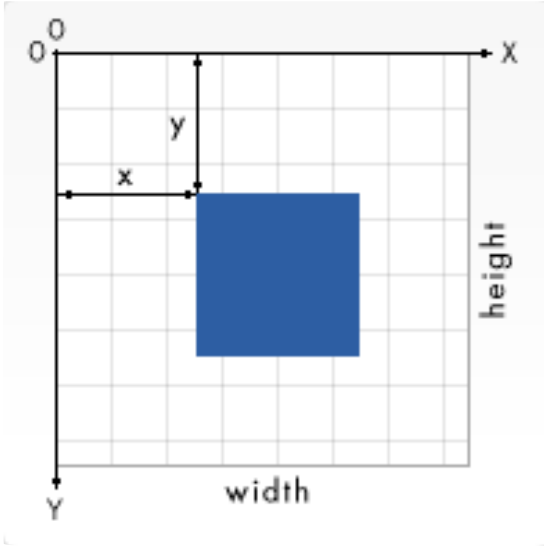
- There are many, *many* options and different kinds of charts
- It can be animated and responsive

Chart.js samples



Introducing the <canvas> element

- Web browsers support several kinds of graphics — notable images through the <img> element, but also JavaScript created graphics drawn on the <canvas>
- A canvas is defined with height and width, and its 'context' can then provide a 2D surface to draw upon
- You are provided with a basic set of operations
  - fillRect()
  - moveTo()
  - lineTo()
  - closePath()
  - stroke()
  - …

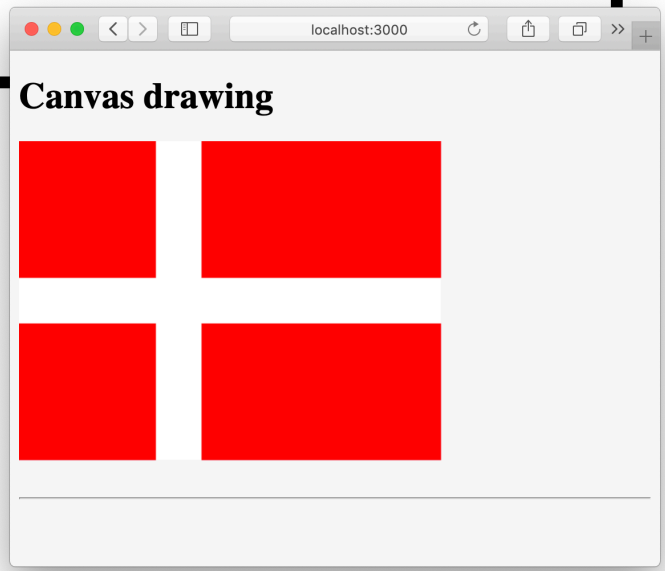


```
app
└── index.js
public/
├── index.html
└── style
    └── main.css
```

A familiar flag

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Basic canvas</title>
  <link rel="stylesheet" type="text/css" media="screen" href="style/main.css" />
  <script defer src="js/main.js"></script>
</head>
<body>
  <h1>Canvas drawing</h1>
  <canvas id="myCanvas" width="400px" height="300px"></canvas>
  <hr>
</body>
</html>
```

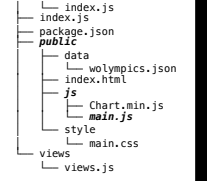
```
const canvas = document.querySelector('#myCanvas')
const ctx = canvas.getContext('2d')
ctx.fillStyle = 'red'
ctx.fillRect(0, 0, 370, 280)
ctx.fillStyle = 'white'
ctx.fillRect(120, 0, 40, 280)
ctx.fillRect(0, 120, 370, 40)
```



Setting up the canvas: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Olympic Winners</title>
    <link rel="stylesheet" type="text/css" href="style/main.css" />
    <script src="js/Chart.min.js"></script>
    <script defer src="js/main.js"></script>
  </head>

  <body>
    <div id="medalTable"></div>
    <button id="dkMedals">Moar medals!</button>
    <div style="position: relative; height:40vh; width:80vw">
      <canvas id="medalChart"></canvas>
    </div>
  </body>
</html>
```

- **Chart.js requires a** canvas **element to function and should be wrapped in an element with** 'position: relative;' **in order to be responsive**



Charting the data

```
function createStackedHBarChart (chartData
= []) {
  if (medalChartCtx) {
    const stackedBars = new
Chart(medalChartCtx, {
      type: 'horizontalBar',
      data: {
        labels: chartData.labels,
        datasets: [
          {
            label: 'Gold',
            data: chartData.gold,
            backgroundColor: 'gold'
          },{
            label: 'Silver',
            data: chartData.silver,
            backgroundColor: 'silver'
          },{
            label: 'Bronze',
            data: chartData.bronze,
            backgroundColor: '#E69E23'
          }
        ]
      },
      options: {
        title: {
          display: true,
          text: 'Winter Olympics 2018'
        },
        responsive: true,
        scales: {
          xAxes: [
            {
              stacked: true
            }
          ],
          yAxes: [
            {
              stacked: true
            }
          ]
        }
      }
    });
    if (dkMedals) {
      dkMedals.onclick = e => {
        chartData.gold[3]++;
        chartData.silver[3]++;
        chartData.bronze[3]++;
        stackedBars.update();
      };
    }
  }
}
```

Chart in action

Data drives the Web

- When we have data in an accessible and malleable format, we can transform it to suit our needs
- If we transmit data as JSON and transform it into HTML (or whatever we want) in the Web browser, we can add/change content seamlessly
- Visualisation libraries are very powerful, and as such can require a bit of work to get *just* right