# 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

- **A static baseline from JSON**
- Dynamic generated page from JSON
- Dynamic generated page from a database
- Template generated page from a database
- Doing data visualisations with Charts.js
- Drawing on the canvas

```
app
└── index.js
public
├── data
│   └── wolympics.json
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# JSON → HTML: index.js

```
'use strict'
const path = require('path')
const util = require('util')
const fs = require('fs')
const express = require('express')
const exphbs = require('express-handlebars')

const app = express()
const port = 3000

const readFile = util.promisify(fs.readFile)
app.engine('.hbs', exphbs({
  defaultLayout: 'main',
  extname: '.hbs',
  layoutsDir: path.join(__dirname, '../views/layouts')
}))
app.set('view engine', '.hbs')
app.set('views', path.join(__dirname, '../views'))

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

app.get('/', (request, response, next) => {
  readFile(path.join(__dirname, '../public/data/wolympics.json'))
    .then((data) => {
      response.render('winners', JSON.parse(data))
    })
    .catch((error) => next(error))
})
…
```

# Wait… *promisify?*

- **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 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 even easier

```
'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) ⇒ { … }

# Writing to a file, old school

```
'use strict'

const fs = require('fs')

const fileName = 'message.txt'
fs.open(fileName, 'a', (err, fd) => {
  if (err) throw err
  fs.appendFile(fd, 'Hello world!\n', (err) => {
    fs.close(fd, (err) => {
      if (err) throw err
    })
    if (err) throw err
  })
})
```

- **Three levels of callback functions!**

# Writing to a file with promises

```
'use strict'
const fs = require('fs')

function openFile (fileName, mode) {
  return new Promise((resolve, reject) =>
{
    fs.open(fileName, mode, (err, fd) => {
      if (err) reject(err)
      else resolve(fd)
    })
  })
}

function appendFile (fd, data) {
  return new Promise((resolve, reject) =>
{
    fs.appendFile(fd, data, 'utf8', (err)
=> {
      if (err) reject(err)
      else resolve(fd)
    })
  })
}
```

```
function closeFile (fd) {
  return new Promise((resolve, reject) =>
{
    fs.close(fd, (err) => {
      if (err) reject(err)
      else resolve()
    })
  })
}

openFile('message.txt', 'a')
  .then(fd => appendFile(fd,
    'Hello world!\n'))
  .then(fd => closeFile(fd))
  .catch(err => {
    if (err) throw err
  })
```

- You can chain promises, if you return a promise

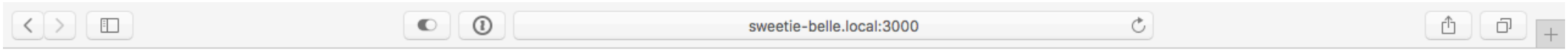
```
app
└── index.js
public
├── data
│   └── wolympics.json
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# JSON → HTML: JSON & HBS

```
{"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}
]}
```

```
<h2>Winter Olympics Winners</h2>
<table>
  <thead>
    <tr><th>Year</th><th>Country</th><th>Gold</th><th>Silver</th><th>Bronze</th></tr>
  </thead>
  <tbody>
  {{#each wolympics}}
    <tr><td>{{year}}</td><td>{{country}}</td><td>{{gold}}</td><td>{{silver}}</td><td>{{bronze}}</td></tr>
  {{/each}}
  </tbody>
</table>
```

# JSON → HTML




## 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 |

# Overviews

- A static baseline from JSON
- **Dynamic generated page from JSON**
- **Dynamic generated page from a database**
- **Template generated page from a database**
- **Doing data visualisations with Charts.js**
- **Drawing on the canvas**

# You have seen XMLHttpRequest…

- When we used it to send a DELETE message

```
const request = new XMLHttpRequest()
request.open('DELETE', requestURL)
request.send()
```

- But there are more modern ways to interact with server, where we get to use promises!

```
fetch(requestURL).then((response) => {
  response.json().then((data) => {
 /* do something with lovely data */
  })
})
```

# Style & anchoring

```
app
└── index.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

```
body {
  font-family: sans-serif;
}

th {
  text-transform: capitalize;
}
```

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

# Generating Web elements

- **We have seen how HTML elements can be created and added to existing elements using** createElement() **and** appendElement()
- **A common approach to make document creation manageable is to create a function for every kind of used element**
- **In the element function, the element is created, its content and attributes set, and lastly, it is returned to be appended to a parent element**

# 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 table from JSON: main.js

```
'use strict'
/* global fetch */

const myTable =document.querySelector('#myTable')
myTable.appendChild(h2('Winter Olympics
Winners'))

fetch('../data/wolympics.json').then((response)
=> {
  response.json().then((data) => {
    myTable.appendChild(table(data.wolympics))
  })
})

function h2 (headline) {
  const h2 = document.createElement('h2')
  h2.textContent = headline
  return h2
}

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 (let line of entries) {
    tbody.appendChild(tr(line))
  }
  table.appendChild(tbody)
  return table
}

function tr (line, type = 'body') {
  const tr = document.createElement('tr')
  switch (type) {
    case 'head':
      for (let name in line) {
        tr.appendChild(th(name))
      }
      break
    case 'body':
    default:
      for (let name in line) {
        tr.appendChild(td(line[name]))
      }
      break
  }
  return tr
}

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
}
```

# A table dynamic and static


## 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 |

# Overviews

- A static baseline from JSON
- Dynamic generated page from JSON
- **Dynamic generated page from a database**
- **Template generated page from a database**
- **Doing data visualisations with Charts.js**
- **Drawing on the canvas**

# Database > JSON

- JSON is a fine format for exchanging information
- A proper database is better for storing data
  - data can be queried, changed, added, and deleted quickly and safely
- However, when we ship data off to the browser, JSON is an excellent choice

```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# Populating the database

```
pool.getConnection((err, connection) => {
  if (err) throw err
  connection.query(
    `CREATE TABLE IF NOT EXISTS wolympics
    (id INTEGER PRIMARY KEY AUTO_INCREMENT, year
INTEGER, country VARCHAR(255) NOT NULL, gold
INTEGER, silver INTEGER, bronze INTEGER,
     UNIQUE (country, year))`, (err) => {
      if (err) throw err
      connection.query('SELECT COUNT(id) FROM
wolympics', (err, results, fields) => {
        if (err) throw err
        if (results[0]['COUNT(id)'] === 0) {
          const fs = require('fs')
          const path = require('path')
          const util = require('util')
          const readFile =
util.promisify(fs.readFile)
          readFile(path.join(__dirname, '../
public/data/wolympics.json'))
            .then((data) => {
              const startData =
JSON.parse(data).wolympics
              for (const entry of startData) {
                WOlympics.insert(entry)
              }
            })
            .catch((err) => {
              console.error(err)
            })
        }
      })
    })
  connection.release()
})
```

```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# Requesting JSON: main.js

```
fetch('/winners', {
  method: 'get',
  headers: {
    'Accept': 'application/json'
  }}).then((response) => {
    response.json().then((data) => {
      myTable.appendChild(table(data))
    })
  })
```

- fetch() **is given extra arguments to specify in the header that we will only accept** 'application/json' **as a response from the server**

```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# Requesting JSON: index.js

```
app.get('/winners', (request, response, next) => {
  if (request.accepts('application/json') && !request.accepts('text/html')) {
    WOlympics.all((err, data) => {
      if (err) return next(err)
      response.contentType('application/json')
      response.end(JSON.stringify(data))
    })
  } else {
    readFile(path.join(__dirname, '../public/data/wolympics.json'))
  .then((data) => {
    response.render('winners', JSON.parse(data))
  })
  .catch((error) => next(error))
  }
})
```

- /winners **is modified to check the** Accept **header. Since an ordinary Web browser request *also* accepts JSON, we have to filter** 'text/html' **to make sure it is really JSON and nothing else that is requested**
- WOlympics **is the class encapsulating the MySQL database**

# Two kinds of /winners here


## 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 |

# Overviews

- A static baseline from JSON
- Dynamic generated page from JSON
- Dynamic generated page from a database
- **Template generated page from a database**
- **Doing data visualisations with Charts.js**
- **Drawing on the canvas**

# Generating dynamic content

- **As you have seen, we can build a Web by assembling calls to element functions**
  - sometimes that is the most elegant solution
- **But we had those neato Handlebars templates…**
  - let's reuse them!

```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   ├── handlebars.runtime.min.js
│   ├── handlebars.winners.js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# Easy-peasy: main.js

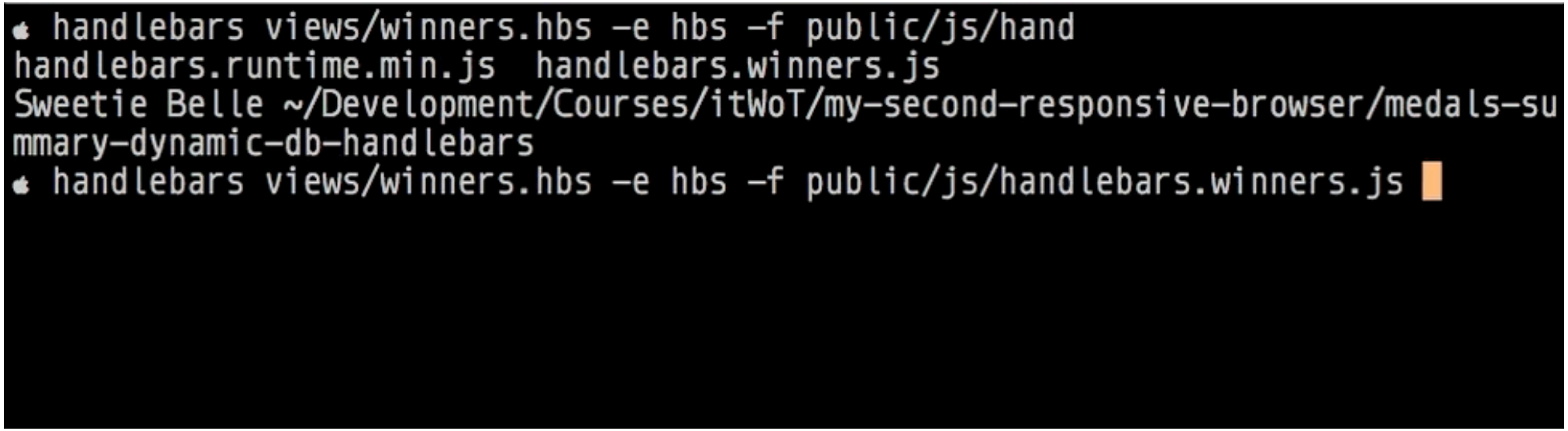
```
'use strict'
/* global fetch Handlebars  */

const myTable = document.querySelector('#myTable')
fetch('/winners', {
  method: 'get',
  headers: {
    'Accept': 'application/json'
  }}).then((response) => {
    response.json().then((data) => {
      myTable.innerHTML = Handlebars.templates.winners({wolympics: data})
    })
  })
```

- **That's the entire file!**

# Handlebars in the browser

- **Handlebars works well in the browser, *especially* if we use the** handlebars **command line tool to precompile the .hbs file into a JavaScript file**
  - npm install -g handlebars
- **We also need to include the handlebars-runtime, which can be found in** node_modules/handlebars/dist/



```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   ├── handlebars.runtime.min.js
│   ├── handlebars.winners.js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

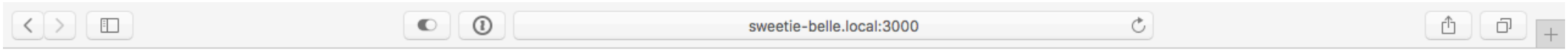
# Handlebars: index.html

## This approach illustrates the advantage of using JavaScript on the server as well as in the browser:

reuse of functionality is (often) fairly simple

```
<!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/handlebars.runtime.min.js"></script>
  <script src="js/handlebars.winners.js"></script>
  <script defer src="js/main.js"></script>
</head>
<body>
  <div id="myTable"></div>
</body>
</html>
```

# The same page, but with Handlebars!




## 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 |

# Overviews

- A static baseline from JSON
- Dynamic generated page from JSON
- Dynamic generated page from a database
- Template generated page from a database
- **Doing data visualisations with Charts.js**
- **Drawing on the canvas**

# 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, 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**

```
app
└── index.js
db
└── db.js
public
├── data
│   └── wolympics.json
├── index.html
├── js
│   ├── Chart.min.js
│   ├── handlebars.runtime.min.js
│   ├── handlebars.winners.js
│   └── main.js
└── style
    └── main.css
views
├── layouts
│   └── main.hbs
└── winners.hbs
```

# 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**

# Setting up a 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/handlebars.runtime.min.js"></script>
  <script src="js/handlebars.winners.js"></script>
  <script src="js/Chart.min.js"></script>
 <script defer src="js/main.js"></script>
</head>
<body>
  <div id="myTable"></div>
  <div style="position: relative; height:40vh; width:80vw">
    <canvas id="myChart"></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**

# Setting up a plot: main.js

```
'use strict'
/* global fetch Handlebars Chart */
const myTable = document.querySelector('#myTable')
const myChartCtx = document.querySelector('#myChart')

fetch('/winners', {
  method: 'get',
  headers: {
    'Accept': 'application/json'
  }}).then((response) => {
    response.json().then((data) => {
      myTable.innerHTML = Handlebars.templates.winners({wolympics: data})
      makeMyChart(data)
    })
  })

function makeMyData (data) {
  const labels = []
  const gold = []
  const silver = []
  const bronze = []
  for (let c of data) {
    labels.push(c.country)
    gold.push(c.gold)
    silver.push(c.silver)
    bronze.push(c.bronze)
  }
  return { labels, gold, silver, bronze }
}
```

# Charting the plot: main.js

```
function makeMyChart (data) {
  const myData = makeMyData(data)
  const myChart = new Chart(myChartCtx,{
    type: 'horizontalBar',
    data: {
      labels: myData.labels,
      datasets: [{
        label: 'Gold',
        data: myData.gold,
        backgroundColor: 'gold'
      }, {
        label: 'Silver',
        data: myData.silver,
        backgroundColor: 'silver'
      }, {
        label: 'Bronze',
        data: myData.bronze,
        backgroundColor: '#E69E23'
      }]
    },
    options: {
      title: {
        display: true,
        text: 'Winter Olympics 2018'
      },
      responsive: true,
      scales: {
        xAxes: [{
          stacked: true
        }],
        yAxes: [{
          stacked: true
        }]
      }
    }
  })
}
```

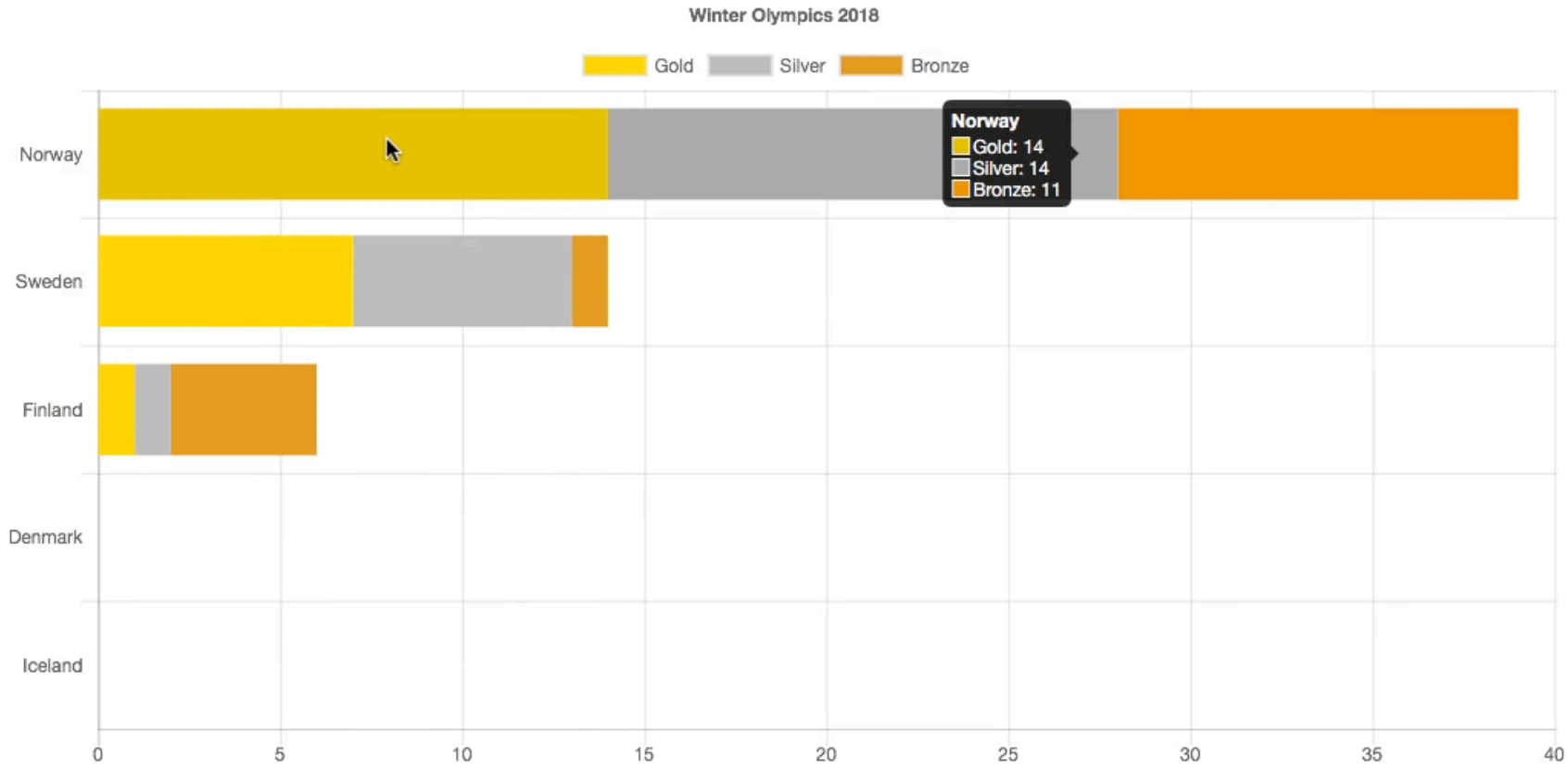
# Chart in action

Safari File Edit View History Bookmarks Develop Window Help 100% Mar 9 Fri 02:54

sweetie-belle.local:3000

## 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 |

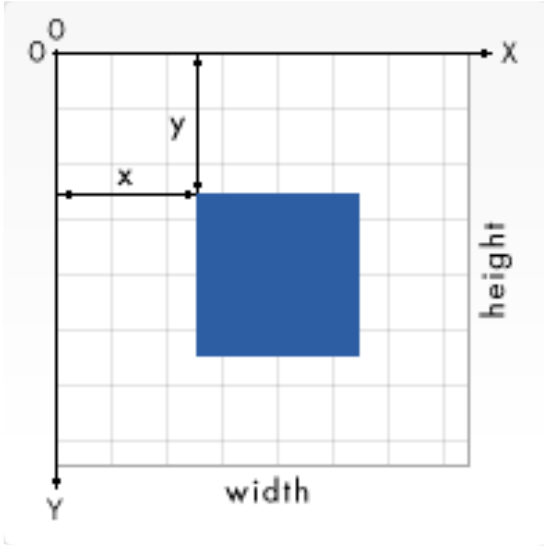


# Overviews

- A static baseline from JSON
- Dynamic generated page from JSON
- Dynamic generated page from a database
- Template generated page from a database
- Doing data visualisations with Charts.js
- **Drawing on the canvas**

# 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()
  - …

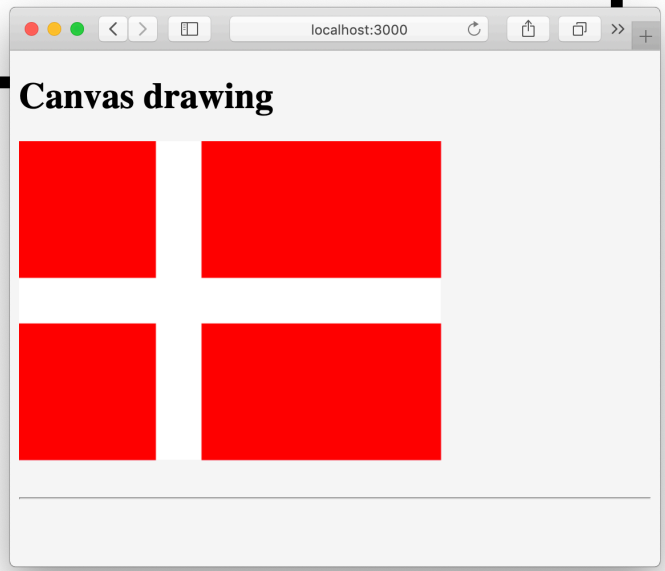


```
app
└── index.js
public/
├── index.html
└── style
    └── main.css
```

# 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)
```



# Data drives the Web

- When we have data in an accessible and malleable format, we can transform it to suit our needs
- JavaScript enables us to use the same frameworks on the server as well as the browser, which reduces cognitive load considerably
- Visualisation libraries are very powerful, and as such can require a bit of work to get *just* right