

CREATE WITH DATA

# D3 START TO FINISH

Learn how to make a custom  
data visualisation using D3.js

PETER COOK

# D3 Start to Finish

Learn how to make a custom data visualisation using D3.js.

Peter Cook

This book is for sale at <http://leanpub.com/d3-start-to-finish>

This version was published on 2022-10-17



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 Peter Cook

# Tweet This Book!

Please help Peter Cook by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#d3starttofinish](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#d3starttofinish](#)

# Contents

|   |           |
|---|-----------|
| <b>1. Introduction</b>                    | <b>1</b>  |
| 1.1 Prerequisites                         | 1         |
| 1.2 What you'll learn                     | 2         |
| 1.3 What you'll build                     | 3         |
| 1.4 Code download                         | 6         |
| 1.5 Where to get help                     | 6         |
| 1.6 Stay in touch                         | 6         |
| 1.7 Translators                           | 6         |
| <b>2. Energy Explorer</b>                 | <b>7</b>  |
| 2.1 Technical overview                    | 7         |
| 2.2 Overview of the Energy Explorer build | 8         |
| <b>3. Practical: Setting Up</b>           | <b>10</b> |
| 3.1 What you need                         | 10        |
| 3.2 Get set up                            | 10        |
| 3.3 Summary                               | 15        |
| <b>4. Introduction to D3</b>              | <b>16</b> |
| 4.1 Joining data with D3                  | 16        |
| 4.2 D3 modules                            | 18        |
| 4.3 D3 versions                           | 23        |
| <b>5. Requesting Data with D3</b>         | <b>25</b> |
| 5.1 d3.csv                                | 25        |
| 5.2 d3.tsv                                | 28        |
| 5.3 d3.dsv                                | 28        |
| 5.4 d3.json                               | 29        |
| 5.5 CSV, TSV or JSON?                     | 30        |
| 5.6 Static resources and APIs             | 30        |
| <b>6. Practical: Load the Data</b>        | <b>31</b> |

## CONTENTS

|            |  |           |
|------------|--|-----------|
| 6.1        | Overview . . . . .   | 31        |
| 6.2        | Inspect the data . . . . .                                     | 31        |
| 6.3        | Include JavaScript files in index.html . . . . .               | 32        |
| 6.4        | Request data . . . . .   | 33        |
| 6.5        | Summary . . . . .  | 36        |
| <b>7.</b>  | <b>D3 Selections . . . . .</b>                                 | <b>37</b> |
| 7.1        | Creating a selection . . . . .                                 | 37        |
| 7.2        | Updating a selection's elements . . . . .                      | 38        |
| 7.3        | Multiple updates . . . . .                                     | 43        |
| 7.4        | Chained selections . . . . .                                   | 44        |
| <b>8.</b>  | <b>Data Joins . . . . .</b>                                    | <b>46</b> |
| 8.1        | Creating a data join . . . . .                                 | 46        |
| 8.2        | Updating the joined elements . . . . .                         | 49        |
| 8.3        | Joining arrays of objects . . . . .                            | 56        |
| <b>9.</b>  | <b>Practical: Draw the Data . . . . .</b>                      | <b>61</b> |
| 9.1        | Overview . . . . .   | 61        |
| 9.2        | Add a container for the circles . . . . .                      | 61        |
| 9.3        | Join the data array to circle elements . . . . .               | 62        |
| 9.4        | Save and refresh . . . . .                                     | 63        |
| <b>10.</b> | <b>Scale Functions . . . . .</b>                               | <b>65</b> |
| 10.1       | scaleLinear . . . . .  | 65        |
| 10.2       | scaleSqrt . . . . .  | 68        |
| 10.3       | Putting scale functions to use . . . . .                       | 68        |
| <b>11.</b> | <b>Practical: Size the Circles . . . . .</b>                   | <b>70</b> |
| 11.1       | Overview . . . . .   | 70        |
| 11.2       | Convert indicator values from strings into numbers . . . . .   | 70        |
| 11.3       | Create a sqrtScale function and set the circle radii . . . . . | 72        |
| <b>12.</b> | <b>Architecture for Web-based Data Visualisation . . . . .</b> | <b>75</b> |
| 12.1       | Layout functions . . . . .                                     | 75        |
| 12.2       | Modules . . . . .  | 77        |
| <b>13.</b> | <b>Practical: Add Modules . . . . .</b>                        | <b>80</b> |
| 13.1       | Overview . . . . .   | 80        |
| 13.2       | Add new modules . . . . .                                      | 80        |
| 13.3       | Add a layout function . . . . .                                | 81        |
| 13.4       | Move update code . . . . .                                     | 83        |

|            |  |            |
|------------|--|------------|
| 13.5       | Use layout function . . . . .  | 84         |
| 13.6       | Save and refresh . . . . .   | 84         |
| <b>14.</b> | <b>Arranging Items in a Grid . . . . .</b>                           | <b>86</b>  |
| <b>15.</b> | <b>Practical: Arrange the Data . . . . .</b>                         | <b>91</b>  |
| 15.1       | Overview . . . . .   | 91         |
| 15.2       | Add configuration object . . . . .                                   | 92         |
| 15.3       | Modify layout function . . . . .                                     | 93         |
| 15.4       | Save and refresh . . . . .   | 94         |
| 15.5       | Summary . . . . .  | 95         |
| <b>16.</b> | <b>The Build So Far . . . . .</b>                                    | <b>96</b>  |
| <b>17.</b> | <b>More on D3 Selections . . . . .</b>                               | <b>101</b> |
| 17.1       | More selection methods . . . . .                                     | 101        |
| 17.2       | Update functions . . . . .   | 104        |
| 17.3       | Summary . . . . .  | 107        |
| <b>18.</b> | <b>More on D3 Joins . . . . .</b>                                    | <b>108</b> |
| 18.1       | Joining an array to groups of elements . . . . .                     | 108        |
| 18.2       | Nested joins . . . . .   | 113        |
| <b>19.</b> | <b>Practical: Add Labels . . . . .</b>                               | <b>121</b> |
| 19.1       | Overview . . . . .   | 121        |
| 19.2       | Add label information to layout.js . . . . .                         | 122        |
| 19.3       | Modify update function to join data to <g> elements . . . . .        | 123        |
| 19.4       | Center the labels using CSS . . . . .                                | 125        |
| 19.5       | Summary . . . . .  | 127        |
| <b>20.</b> | <b>Practical: Add More Circles . . . . .</b>                         | <b>128</b> |
| 20.1       | Overview . . . . .   | 128        |
| 20.2       | Add properties for each energy type in the layout function . . . . . | 129        |
| 20.3       | Add four circles in the update function . . . . .                    | 130        |
| 20.4       | Style the circles . . . . .  | 132        |
| 20.5       | Save and refresh . . . . .   | 133        |
| 20.6       | Summary . . . . .  | 133        |
| <b>21.</b> | <b>Practical: Style the Circles . . . . .</b>                        | <b>134</b> |
| 21.1       | Overview . . . . .   | 134        |
| 21.2       | Design a colour scheme . . . . .                                     | 135        |
| 21.3       | Style the circles . . . . .  | 137        |
| 21.4       | Center the visualisation . . . . .                                   | 138        |

|            |   |            |
|------------|---|------------|
| 21.5       | Change the background colour . . . . .                  | 139        |
| 21.6       | Save and refresh . . . . .                              | 140        |
| 21.7       | Summary . . . . .                                       | 140        |
| <b>22.</b> | <b>D3 Event Handling . . . . .</b>                      | <b>141</b> |
| 22.1       | Event types . . . . .                                   | 144        |
| 22.2       | index and this . . . . .                                | 145        |
| <b>23.</b> | <b>Flourish Popup Library . . . . .</b>                 | <b>146</b> |
| 23.1       | Installing the popup . . . . .                          | 146        |
| 23.2       | Initialising the popup . . . . .                        | 146        |
| 23.3       | Popup methods . . . . .                                 | 147        |
| 23.4       | Popup styling . . . . .                                 | 147        |
| 23.5       | Example . . . . .                                       | 147        |
| <b>24.</b> | <b>Practical: Add a Popup . . . . .</b>                 | <b>150</b> |
| 24.1       | Overview . . . . .                                      | 150        |
| 24.2       | Link to the popup library . . . . .                     | 151        |
| 24.3       | Add new module for popup . . . . .                      | 151        |
| 24.4       | Add event handlers . . . . .                            | 153        |
| 24.5       | Populate popup with energy data . . . . .               | 155        |
| 24.6       | Offset the popup . . . . .                              | 157        |
| 24.7       | Summary . . . . .                                       | 162        |
| <b>25.</b> | <b>State Management . . . . .</b>                       | <b>163</b> |
| 25.1       | State management example . . . . .                      | 164        |
| 25.2       | Summary . . . . .                                       | 168        |
| <b>26.</b> | <b>Practical: Add State Management . . . . .</b>        | <b>169</b> |
| 26.1       | Overview . . . . .                                      | 169        |
| 26.2       | Add a new module . . . . .                              | 169        |
| 26.3       | Add an empty state object and action function . . . . . | 170        |
| <b>27.</b> | <b>Practical: Add a Menu . . . . .</b>                  | <b>172</b> |
| 27.1       | Overview . . . . .                                      | 172        |
| 27.2       | Add a container for the menu . . . . .                  | 174        |
| 27.3       | Add selectedIndicator state property . . . . .          | 175        |
| 27.4       | Add a new module for the menu . . . . .                 | 175        |
| 27.5       | Add code to construct and manage the menu . . . . .     | 176        |
| 27.6       | Basic styling of the menu . . . . .                     | 178        |
| 27.7       | Call updateMenu from update function . . . . .          | 179        |
| 27.8       | Summary . . . . .                                       | 180        |

|   |            |
|---|------------|
| <b>28. Data Manipulation</b>                                  | <b>181</b> |
| 28.1 Installing Lodash  | 181        |
| 28.2 Lodash syntax  | 181        |
| 28.3 <code>_uniq</code>                                       | 182        |
| 28.4 <code>_includes</code>                                   | 182        |
| 28.5 <code>_without</code>                                    | 182        |
| 28.6 <code>_filter</code>                                     | 183        |
| 28.7 <code>_sortBy</code>                                     | 184        |
| 28.8 <code>_orderBy</code>                                    | 186        |
| 28.9 Summary  | 187        |
| <b>29. Practical: Sort the Countries</b>                      | <b>189</b> |
| 29.1 Overview   | 189        |
| 29.2 Link to Lodash   | 190        |
| 29.3 Sort the data  | 191        |
| 29.4 Hide countries with a zero or missing value              | 194        |
| 29.5 Summary  | 195        |
| <b>30. D3 Transitions</b>                                     | <b>197</b> |
| 30.1 Creating D3 transitions                                  | 197        |
| 30.2 Transition duration                                      | 200        |
| 30.3 Transition delay   | 201        |
| 30.4 Key functions  | 202        |
| 30.5 Summary  | 204        |
| <b>31. Practical: Add Transitions</b>                         | <b>206</b> |
| 31.1 Overview   | 206        |
| 31.2 Add a key function to the data join                      | 207        |
| 31.3 Add transition to the country groups                     | 208        |
| 31.4 Add transition duration and delay                        | 209        |
| 31.5 Initialise the country group positions                   | 211        |
| 31.6 Add a transition to the circle radii                     | 212        |
| 31.7 Summary  | 214        |
| <b>32. Practical: Add a Legend</b>                            | <b>215</b> |
| 32.1 Overview   | 215        |
| 32.2 Add the legend to index.html                             | 216        |
| 32.3 Add a function to update the radius of the legend circle | 217        |
| 32.4 Style the legend   | 218        |
| 32.5 Summary  | 220        |
| <b>33. Practical: Finishing Touches</b>                       | <b>221</b> |

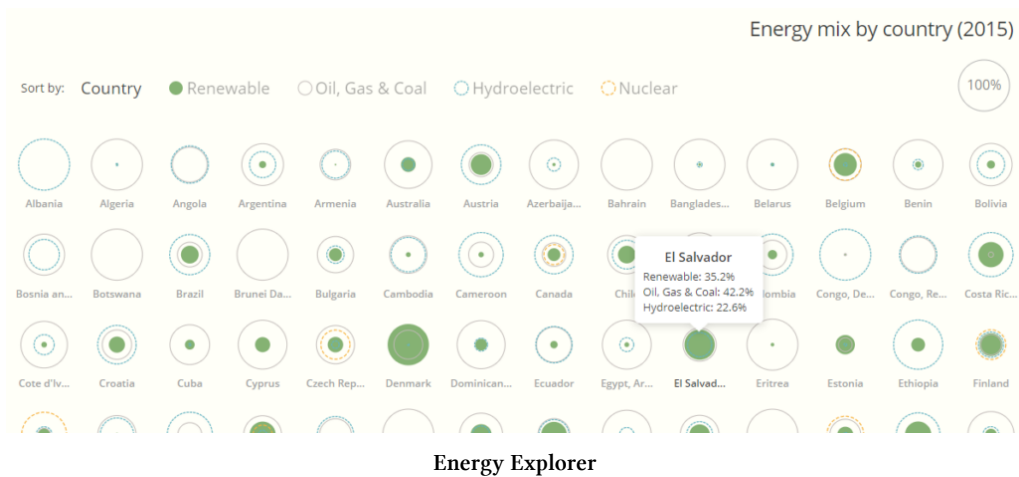


## CONTENTS

|                    |   |            |
|--------------------|---|------------|
| 33.1               | Overview . . . . .                      | 222        |
| 33.2               | Add and apply Open Sans font . . . . .  | 222        |
| 33.3               | Style the menu . . . . .                | 224        |
| 33.4               | Add header and footer . . . . .         | 229        |
| 33.5               | Other CSS tweaks . . . . .              | 232        |
| 33.6               | Summary . . . . .                       | 235        |
| <b>34.</b>         | <b>Wrapping Up . . . . .</b>            | <b>237</b> |
| 34.1               | Where next? . . . . .                   | 238        |
| 34.2               | Wrapping up . . . . .                   | 239        |
| <b>Appendix A:</b> | <b>Tools and Set-up . . . . .</b>       | <b>241</b> |
|                    | Web development tools . . . . .         | 241        |
| <b>Appendix B:</b> | <b>Example Set-Up . . . . .</b>         | <b>246</b> |
|                    | Create a project directory . . . . .    | 246        |
|                    | Install a code editor . . . . .         | 246        |
|                    | Create a minimal HTML file . . . . .    | 246        |
|                    | Install Live Server extension . . . . . | 247        |
|                    | Start Server . . . . .                  | 247        |
|                    | Add CSS and JavaScript files . . . . .  | 248        |
|                    | Summary . . . . .                       | 250        |

# 1. Introduction

Welcome! First of all, I'd like to thank you for buying this book. The aim is to teach you about D3.js and show you how to build a real-world interactive data visualisation called **Energy Explorer**.



This book uses a mix of **theory** and **practice**. You'll learn an aspect of D3 then put it into practice during the build of Energy Explorer. The book is divided into several chapters. Some contain theoretical information and others contain practical steps to build Energy Explorer. Each practical chapter has the name 'Practical' in its title, for example 'Practical: What You'll Need'.

## 1.1 Prerequisites

To get the most out of this book I recommend that you're familiar with:

- HTML
- SVG
- CSS
- JavaScript

You should also be comfortable opening and editing code in a code editor (such as VS Code, Atom or Brackets) and setting up a local web server.

If you need to get up to speed on the above languages and tools I suggest reading my [Fundamentals of HTML, SVG, CSS & JavaScript for Data Visualisation<sup>1</sup>](#) book.

## 1.2 What you'll learn

This book aims to teach you enough D3 to build a **custom data visualisation** from the ground up. Broadly speaking we'll cover:

- **D3 selections and joins** (these are used to update HTML and SVG elements in a data-driven fashion).
- **D3 modules** (such as scale functions and transitions).
- **Third party libraries** such as Lodash that are useful when building data visualisations.
- General **web development patterns** that are helpful when building interactive data visualisations.

### 1.2.1 D3 selections and joins

D3 has a powerful mechanism for adding, removing and updating HTML/SVG elements in a data-driven fashion known as selections and joins. You'll learn how to use **selections** to select and update HTML/SVG elements. You'll also learn how to update the elements in a data-driven fashion using D3 **joins**. For example, given an array of data, you'll learn how to keep a group of SVG circles synchronised with the data.

### 1.2.2 D3 modules

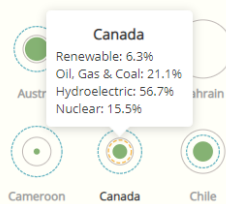
D3 has a large number of modules that help you build data visualisations. We cover the following D3 modules (and concepts) in this book:

| Name                   | Description  |
|------------------------|--|
| <b>Scale functions</b> | Scale functions transform data values (e.g. percentages) into visual values (e.g. position, size or colours) |
| <b>Data requests</b>   | Data requests let you load CSV and JSON files  |
| <b>Event handling</b>  | Event handling lets you respond to clicks and hovers from the user   |
| <b>Transitions</b>     | Transitions let you animate the position, size and colour of HTML/SVG elements                               |

### 1.2.3 Third party libraries

We look at a couple of JavaScript libraries in this book: Flourish's [popup library](#)<sup>2</sup> and [Lodash](#)<sup>3</sup>.

Flourish's popup library helps you add an information popup to a web page and Lodash provides functions for processing data (such as sorting and filtering).



Flourish's information popup (appears when an item is hovered)

### 1.2.4 Web development patterns

We also cover web development techniques that are useful when building interactive data visualisations:

| Name                     | Description  |
|--------------------------|--|
| <b>Modularisation</b>    | Split your code up into separate modules rather than having all your code in a single file.                            |
| <b>State management</b>  | Implement a state management pattern so that user interaction can be handled in a clean and easy to understand manner. |
| <b>Data manipulation</b> | Use the lodash library to process (sort, filter etc.) data.  |

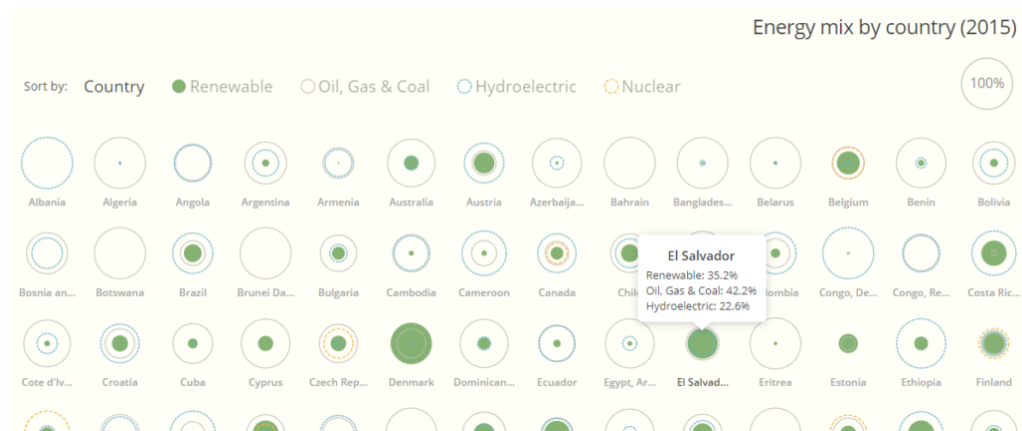
The above techniques will be used during the build of Energy Explorer.

## 1.3 What you'll build

In this book we'll build an interactive data visualisation called **Energy Explorer** which shows the energy mix of 141 countries. You can visit a live version at <https://d3-start-to-finish-energy-explorer.surge.sh/><sup>4</sup>.

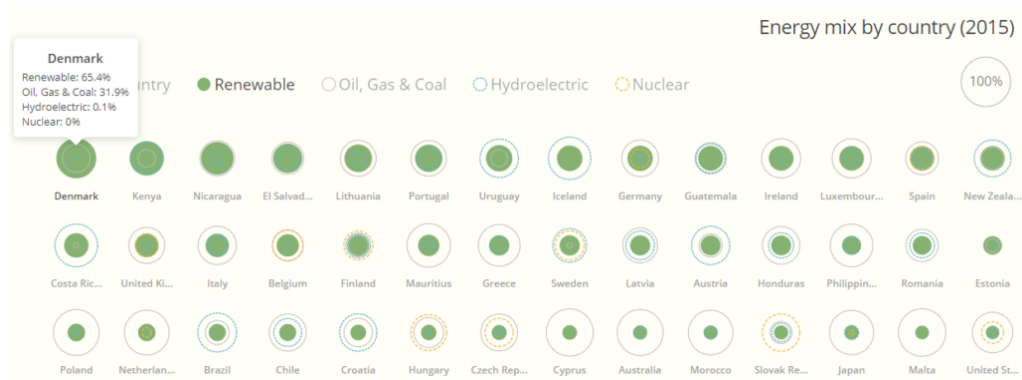
Each country is represented by 4 circles and each circle shows the amount of **renewable**, **fossil fuel**, **hydroelectric** and **nuclear** energy produced by a country.





Information popup when a country is hovered

The countries can be sorted by name or one of the energy types. When the circles sort they animate into their new positions. The following image shows the view when Renewable has been selected. The countries are sorted by the percentage of renewable energy in their energy mix:



Countries sorted by renewable energy

The visualisation is styled to look clean, modern and fresh. Colours, fonts, line widths, spacing etc. have been carefully selected to make the chart attractive and easy to digest.

Energy Explorer will be built using **HTML**, **SVG**, **CSS**, **JavaScript** and **D3**. HTML, SVG, CSS and JavaScript are standard technologies used to build websites and web applications. Practically every website (or web application) you visit is built from these four technologies. D3 is a JavaScript library that helps you build interactive data visualisations. It offers advanced functionality such as adding, removing and updating DOM elements, transforming data into shapes and animating between application states.

This book requires that you're reasonably proficient in HTML, SVG, CSS and JavaScript.

If you think your knowledge of these technologies needs a boost I recommend reading my [Fundamentals of HTML, SVG, CSS & JavaScript for Data Visualisation](#)<sup>5</sup> book.

## 1.4 Code download

You can download the code for this book from [here](#)<sup>6</sup>.

## 1.5 Where to get help

If you need help on anything in this book please tweet me at [@createwithdata](#)<sup>7</sup> or email me at [help@createwithdata.com](mailto:help@createwithdata.com).

## 1.6 Stay in touch

I love to stay in touch with my readers. One of the best ways to do this is via my mailing list. I send occasional messages containing useful information related to implementing data visualisations (e.g. using JavaScript or other tools). There'll also be discount codes for my other books. You can sign up on my website [www.createwithdata.com](http://www.createwithdata.com)<sup>8</sup>.

## 1.7 Translators

If you're interested in translating D3 Start to Finish to another language contact me at [info@createwithdata.com](mailto:info@createwithdata.com) and I'll be happy to discuss further.

## Notes

1 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

2 <https://github.com/kiln/flourish-popup>

3 <https://lodash.com/>

4 <https://d3-start-to-finish-energy-explorer.surge.sh/>

5 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

6 <https://cwg-resources.netlify.app/d3starttofinish/2-6-22-3jrisb/d3-start-to-finish-code.zip>

7 <https://twitter.com/createwithdata>

8 <https://www.createwithdata.com>

## 2. Energy Explorer

This chapter gives a technical overview of Energy Explorer and an overview of the build steps.

### 2.1 Technical overview

This section gives a broad overview of Energy Explorer from a technical perspective. There's no need to understand this section fully, it's just here to give you a flavour of what you'll be building.

Energy Explorer is built using HTML, SVG, CSS and JavaScript. It uses a few JavaScript libraries namely D3, Lodash and Flourish's popup library.

The data originates from the [World Bank's World Development Indicators](#)<sup>9</sup> database and has been transformed into a comma separated value (CSV) file using a Node script. The CSV file looks like:

**data.csv**

---

```
id,name,oil,gas,coal,nuclear,hydroelectric,renewable
AGO,Angola,46.8,,53.2,0.0
ALB,Albania,0.0,,100.0,0.0
ARE,United Arab Emirates,99.8,,0.0,0.2
ARG,Argentina,66.9,,26.2,1.9
ARM,Armenia,35.9,,28.3,0.1
...
```

---

This CSV file is loaded into Energy Explorer using D3. We use D3 selections and joins to create a group of four circles that represent each country. A text label is also created for each country.

A mouse over event handler is added to each country. This triggers a popup containing additional information.

At the top of the visualisation is a menu which allows the user to choose how the countries are sorted.

Sort by: Country ● Renewable ○ Oil, Gas & Coal ○ Hydroelectric ○ Nuclear

Energy Explorer's sort menu



The menu is created using D3. When a menu item is clicked the data is sorted and D3's transitions are used to animate the countries into their new positions.

The application is split into a few modules. One for the main application and further modules for computing the circle positions, for adding a popup and for adding a menu.

The final file structure looks like:

```
energy-explorer
├─ css
│   └─ style.css
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    └─ lib
        ├─ d3.min.js
        ├─ lodash.min.js
        └─ popup-v1.1.1.min.js
    ├─ main.js
    ├─ menu.js
    ├─ popup.js
    ├─ store.js
    └─ update.js
```

This is a typical web application structure. It's a static application, meaning there's no server-side code. All files are loaded into the browser when the visualisation first loads. (In order to stay focused, this book doesn't use any build tools such as Webpack.)

The application consists of a single HTML file (`index.html`). This contains a handful of HTML elements that act as containers for the chart. The main JavaScript file is `main.js` and this loads the data from `data/data.csv`. It transforms the data into position and radius information using a function in `layout.js`. The chart is drawn in `update.js`.

`menu.js` creates the sort menu and `popup.js` manages the popup. `store.js` helps with state management.

D3, Lodash and Flourish's popup library live in the `lib` directory.

## 2.2 Overview of the Energy Explorer build

The build of the Energy Explorer is divided into **fifteen steps**. Each step focuses on a single aspect. For example, loading the data or adding transitions.

The code download includes the **before** and **after** code for each of the fifteen steps. Therefore if you don't complete a particular step, you can still work on the next one.

Typically in each practical session you'll **load** the step's code into your editor, **edit** the code and **view** the result in your web browser. The steps are:

| Step | Description   |
|------|---|
| 1    | Getting Started   |
| 2    | Load the Data   |
| 3    | Draw the Data (you'll represent each country with a circle) |
| 4    | Size the Circles  |
| 5    | Add Modules   |
| 6    | Arrange the Data (in a grid formation)                      |
| 7    | Add Labels  |
| 8    | Add More Circles (for representing each energy type)        |
| 9    | Style the Circles   |
| 10   | Add Popup   |
| 11   | Add State Management and a Menu                             |
| 12   | Sort the Countries  |
| 13   | Add Transitions   |
| 14   | Add a Legend  |
| 15   | Finishing Touches   |

## Notes

<sup>9</sup> <http://datatopics.worldbank.org/world-development-indicators/>

# 3. Practical: Setting Up

## 3.1 What you need

This section covers the tools you need to build Energy Explorer. You need three things:

- A **modern web browser** such as Chrome or Firefox.
- A **code editor** such as [Brackets<sup>10</sup>](#) or [VS Code<sup>11</sup>](#).
- A **web server** running on your computer.

You probably already have a web browser. I recommend using Chrome or Firefox. You're probably fine using other browsers so long as they're modern and up to date.

If you're a developer and already using a preferred code editor then please stick with that. If you don't have a preferred code editor I recommend [Brackets<sup>12</sup>](#) or [VS Code<sup>13</sup>](#). These are covered in Appendix A.

Finally you need to have a web server running on your machine. This is so that the data can be loaded without running into browser security restrictions. If you're a web developer there's a good chance you're familiar with setting up a server using Node, Python or similar. If not I recommend using Brackets as your code editor as this has a built-in server. For further information refer to Appendix A and Appendix B.

## 3.2 Get set up

The aim of this section is to:

- **download the code** (in the form of a zip file)
- **expand** the zip file
- **open the code** in your code editor
- **run a web server** to serve the code
- **view the page** in your web browser

(If at any point you get stuck, refer to Appendix A and Appendix B for further information.)

### 3.2.1 Download the code

Start by downloading the code from [here](#)<sup>14</sup> (it's in the form of a zip file).

Expand the zip and make sure you have a directory structure like:

```
d3-start-to-finish-code
├─ step1
│   └─ index.html
├─ step10
│   └─ css
│       └─ style.css
│   └─ data
│       └─ data.csv
│   └─ index.html
│   └─ js
│       ├── config.js
│       ├── layout.js
│       ├── lib
│       │   ├── d3.min.js
│       │   └─ popup-v1.1.1.min.js
│       ├── main.js
│       └─ update.js
...
```

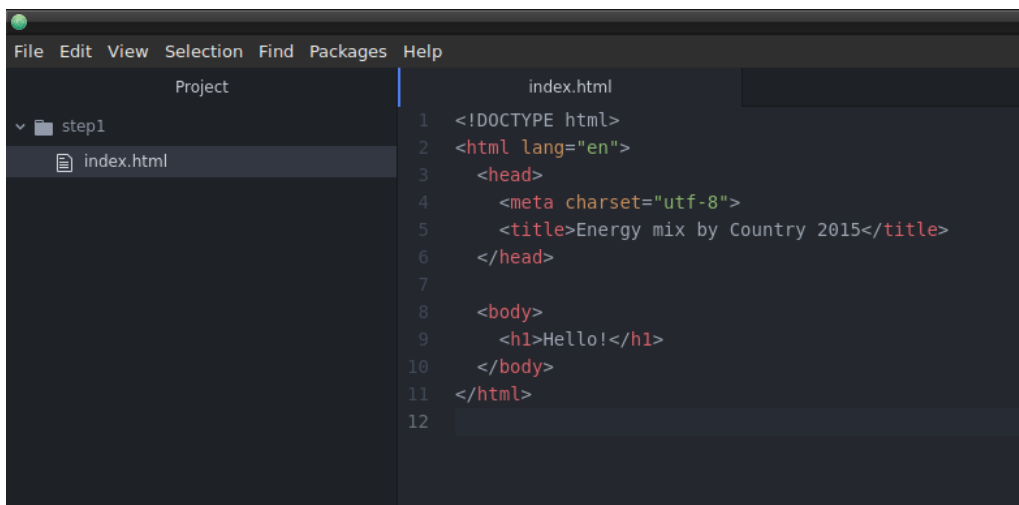
Move the `d3-start-to-finish-code` directory somewhere convenient. Perhaps you already have a directory where you store your projects. (I have a `development` directory within my home directory and have placed `d3-start-to-finish-code` there.) You should aim for something like:

```
peter-home
├─ development
│   ├── d3-start-to-finish-code
│   │   ├── step1
│   │   │   └─ index.html
│   │   ├── step10
│   │   │   ├── css
│   │   │   │   └─ style.css
│   │   │   ├── data
│   │   │   │   └─ data.json
│   │   │   └─ index.html
│   │   └─ ...
│   ├── documents
│   ├── ...
│   └─ music
```

```
| ...  
├─ photos  
| ...
```

### 3.2.2 Open step1 in code editor

Open your text editor and open `d3-start-to-finish-code/step1`. In VS Code or Brackets you select ‘Open Folder..’ from the ‘File’ menu and select `d3-start-to-finish-code/step1`. Within the code editor’s file browser, expand `step1`, click on `index.html` and you should see something like:

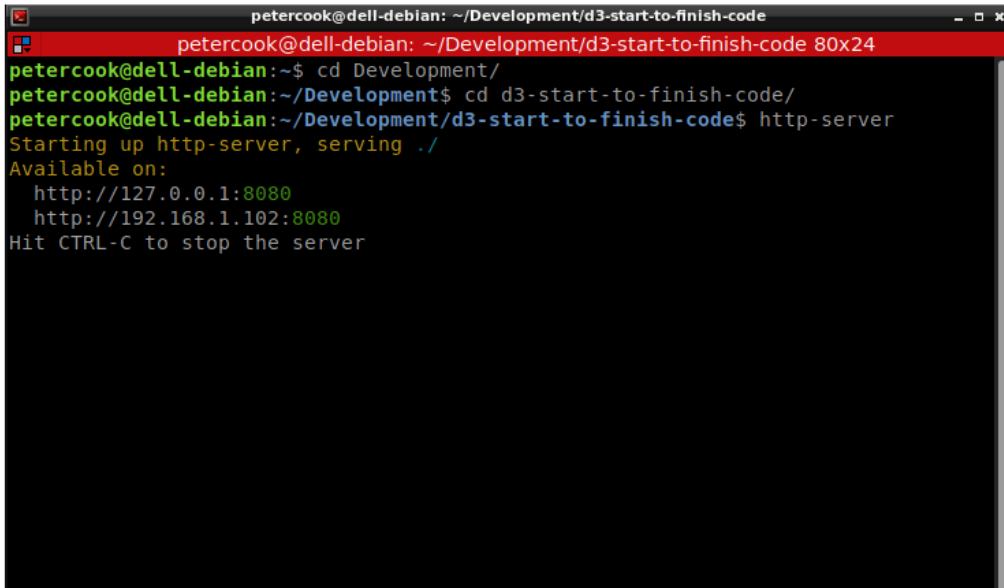
A screenshot of a code editor interface. On the left, a file explorer shows a folder named 'step1' containing a file named 'index.html'. The main editor area displays the content of 'index.html' with the following HTML code:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3   <head>  
4     <meta charset="utf-8">  
5     <title>Energy mix by Country 2015</title>  
6   </head>  
7  
8   <body>  
9     <h1>Hello!</h1>  
10  </body>  
11 </html>  
12
```

step1/index.html in the code download

### 3.2.3 Start web server

If you’re comfortable setting up a local web server (for example using Node or Python) please do so by navigating to `d3-start-to-finish-code` within a command line interface and starting the web server there. Here’s a screenshot of my command line after starting `http-server` (a node web server):

A terminal window titled 'petercook@dell-debian: ~/Development/d3-start-to-finish-code' with a red header bar. The terminal shows the following commands and output:

```
petercook@dell-debian:~$ cd Development/  
petercook@dell-debian:~/Development$ cd d3-start-to-finish-code/  
petercook@dell-debian:~/Development/d3-start-to-finish-code$ http-server  
Starting up http-server, serving ./  
Available on:  
  http://127.0.0.1:8080  
  http://192.168.1.102:8080  
Hit CTRL-C to stop the server
```

Starting node's http-server

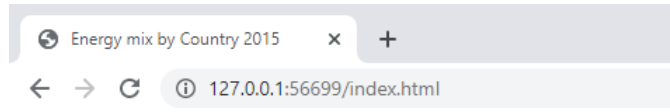
Otherwise I recommend using Brackets as your code editor and using its Live Preview (see next section), or using VS Code's Live Server extension (see next section).

### 3.2.4 View step1 in your browser

If you're using a Node or Python server (or similar), navigate to the URL specified by the server and open step1. The URL should be something like `localhost:8080/step1`.

If you're using Brackets, select `step1/index.html` in the sidebar and click the lightning symbol (or select Live Preview from the File menu). If you're using VS Code's Live Server extension, select the `step1` directory in the sidebar and click the 'Go Live' button (usually it's in the footer menu). See Appendix A and Appendix B for more information.

You should now see a friendly greeting in your web browser:



**Hello!**

step1 of the code download (viewed using Brackets' server)

### 3.2.5 Edit the code

Just to check everything's set up properly try editing the greeting in `index.html` and saving.

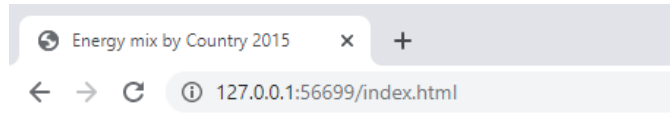
`index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
  </head>
  <body>
    <h1>Let's learn D3!</h1>
  </body>
</html>
```

---

Refresh your browser and verify that the greeting has updated.



## Let's learn D3!

Greeting after editing index.html

### 3.3 Summary

You've completed all the necessary set up steps and are ready to start learning and building with D3! The next chapter introduces D3 and gives a broad overview of what it is and what it can be used for.

### Notes

10 <http://brackets.io/>

11 <https://code.visualstudio.com/>

12 <http://brackets.io/>

13 <https://code.visualstudio.com/>

14 <https://cwd-resources.netlify.app/d3starttofinish/2-6-22-3jrisb/d3-start-to-finish-code.zip>



## 4. Introduction to D3

[D3<sup>15</sup>](#) is a JavaScript library that helps you build **custom**, **interactive** and **beautiful** data visualisations. Rather than providing a library of chart types (which is what libraries like [Highcharts<sup>16</sup>](#) and [Chart.js<sup>17</sup>](#) do) it provides **building blocks** for creating custom (and standard) charts.

D3 primarily consists of:

- a mechanism for joining (or binding) arrays of data to HTML or SVG elements
- a vast number of modules that help build data visualisations

This section gives an overview of D3 joins and some of D3's modules. Don't worry too much if you don't follow everything because we'll cover this material in depth later on.

### 4.1 Joining data with D3

One of the cornerstones of D3 is its **data joining**. A data join creates a correspondence between an **array of data** and a **group of HTML or SVG elements**. Specifically a data join means that:

- each array element has a **corresponding** HTML/SVG element
- each HTML/SVG element may be **positioned**, **sized** and **styled** according to the value(s) of its corresponding array element

Here's an example where an **array of numbers** has been joined to **SVG circles**:

Array: [ 10, 40, 30 ]



An array of numbers joined to circle elements

Each array element **corresponds** to a circle. Furthermore each circle is **sized** according to its corresponding array value.

**Adding** a new array element causes a new circle to be created:

Array: [ 10, 40, 30, 21 ]



Adding a new array element results in a new circle

**Removing** array element(s) causes the corresponding circle(s) to be removed:

Array: [ 10, 40 ]



Removing array elements results in circle removal

**Changing** array value(s) causes the circles to change size accordingly:

Array: [ 34, 26 ]



Modifying array values results in resized circles

You can view an interactive version of this example on Codepen at:

<https://codepen.io/createwithdata/pen/NWxjKVo><sup>18</sup>

The data join ensures that the circles ‘mirror’ the array. This is the basis for creating data visualisations using D3. In effect array elements are represented by shapes (whether circles, lines, rectangles etc.). The shapes may be sized, positioned and styled according to the corresponding data.

We cover D3 joins in detail later in this book. For now just remember that one of D3’s cornerstones is its ability to join arrays to HTML/SVG elements. This results in the HTML/SVG elements mirroring the array elements.

## 4.2 D3 modules

D3 has a large number of modules that help you build interactive data visualisations. Let's look at some of the more commonly used ones.

### 4.2.1 Data requests

D3's **fetch module** makes loading comma-separated value (CSV) and JSON files straightforward. Comma-separated value (CSV) files are text files that represent data in a tabular format. For example:

```
name,indicator-1,indicator-2
Paris,4030,13.45
Tokyo,3912,45.41
New York,19481,32.53
```

JSON files are text files with a format closely related to JavaScript objects and arrays. For example:

```
[
  {
    "name": "Paris",
    "indicator-1": 4030,
    "indicator-2": 13.45
  },
  {
    "name": "Tokyo",
    "indicator-1": 3912,
    "indicator-2": 45.41
  },
  {
    "name": "New York",
    "indicator-1": 19481,
    "indicator-2": 32.53
  }
]
```



CSV files are often produced by exporting data from spreadsheets and databases while the JSON format is often used by APIs.

You can use a D3 function `d3.csv` to request a CSV file from a given URL. When the CSV file arrives, D3 converts it into an array of objects.

For example, suppose there's a CSV file located at `data/cities.csv`:

`'data/cities.csv'`

---

| name     | indicator-1 | indicator-2 |
|----------|-------------|-------------|
| Paris    | 4030        | 13.45       |
| Tokyo    | 3912        | 45.41       |
| New York | 19481       | 32.53       |

---

You can request this data using:

```
d3.csv('data/cities.csv')
```

and D3 converts the CSV file into an array of objects:

```
[
  {
    "name": "Paris",
    "indicator-1": "4030",
    "indicator-2": "13.45"
  },
  {
    "name": "Tokyo",
    "indicator-1": "3912",
    "indicator-2": "45.41"
  },
  {
    "name": "New York",
    "indicator-1": "19481",
    "indicator-2": "32.53"
  }
]
```

Each **object** corresponds to a **row** from the CSV file. For example, the first object corresponds to the first row of data (Paris). This makes working with CSV (and similar) data very simple. We'll take a closer look at data requests in the next section.

## 4.2.2 Scale functions

We've seen that a common pattern in D3 is the data join. For example, given the data:

```
var myData = [ 1, 4, 3 ];
```

we could create a rectangle for each element and set the length of the rectangle proportionally to the value. Rarely will the data correspond exactly to screen coordinates so we usually need to apply some scaling.

We could just multiply each value by a factor but this can make the code harder to understand. Instead the usual approach is to create a scale function. For example you could do this:

```
function barLengthScale(value) {  
  return value * 50;  
}  
  
barLengthScale(1); // returns 50  
barLengthScale(4); // returns 200
```

However D3 provides a system for creating scale functions that saves you having to figure out the underlying mathematics. It's also expressive (it communicates to anyone reading the code what the intention is).

For example, suppose you have some data that represents scores out of 10. This means the data range is  $[0, 10]$ . And suppose you want the length of the bars to vary from 0 to 500, this means the output range is  $[0, 500]$ .

You can create such a scale function using:

```
var barLengthScale = d3.scaleLinear()  
  .domain([0, 10])  
  .range([0, 500]);  
  
barLengthScale(1); // returns 50  
barLengthScale(4); // returns 200
```

The above makes it clear what the scale function's intent is: its input range is 0 to 10 and its output range is 0 to 500. This makes reading and maintaining the code easier. Furthermore you can do things like clamp the scale so that if the input exceeds the input domain, the function clamps the output to the output range. For example:

```
barLengthScale.clamp(true);  
  
barLengthScale(10); // returns 500  
barLengthScale(100); // returns 500
```

You'll learn more about scale functions later in this book and you'll use them to size the circles in Energy Explorer.

### 4.2.3 Axes

Have you ever looked closely at a chart axis and wondered how to create one?

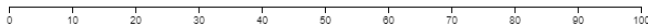


Chart axis generated by D3

It's probably not as easy as you might think! The axis will be different if it appears to the left, right, top or bottom of the chart. How many tick marks should there be? Do you want it to animate if the chart zooms in?

To save you figuring this out D3 provides an axis module. You pass a scale function and an HTML/SVG element into the axis module and D3 figures out how to draw the axis.

### 4.2.4 Shapes

Imagine you were making a doughnut chart consisting of a few arc segments.



Doughnut chart

Each segment is represented by an SVG path element but figuring out the path string is non trivial. For example the SVG code for the large segment on the left is:

```
<path d="M12.53915742027986,95.05140537899537A4,4,0,0,1,8.936653576956331,99.599880636702\
96A100,100,0,1,1,-5.208315972309041,-99.86427511744422A4,4,0,0,1,-0.9999833334166643,-95.\
86970411274646L-0.9999833334166648,-53.76802178493789A4,4,0,0,1,-4.6296141976080225,-49.7\
8520535642397A50,50,0,0,0,2.4424052633738556,49.94031093745256A4,4,0,0,1,6.59776767084554\
9,53.3710557802093Z"></path>
```

Fortunately D3 has a range of helper functions that create SVG path strings. Suppose you want to create an arc segment with start angle 90, end angle 180, inner radius 50 and outer radius 100 you can use:

```
var arcGenerator = d3.arc();

arcGenerator({
  startAngle: 0.5 * Math.PI,
  endAngle: Math.PI,
  innerRadius: 50,
  outerRadius: 100
});
// returns "M100,0A100,100,0,0,1,6.123233995736766e-15,
// 100L3.061616997868383e-15,50A50,50,0,0,0,50,0Z"
```



The angles are specified in radians, so 90 degrees corresponds to  $0.5 * \pi$  radians.

This results in this path:



An arc segment generated by D3

D3 provides generators for other shapes commonly found in data visualisations. You can find out more at [D3 in Depth](#)<sup>19</sup>.

## 4.2.5 Transitions

Animations between different chart states add flair to a data visualisation but they also achieve **object constancy**. Object constancy is a technique that helps someone viewing a chart keep track of points of interest, even if they move to new positions.

For example look at these two charts containing a number of circles:



Before and after circles change position

The left chart is before the circles change position and the right chart is after the circles have changed position. It's impossible to tell which circle has moved where. However if the circles animate to their new positions you stand a chance of following a circle as it moves to its new position.

You can visit <https://codepen.io/createwithdata/pen/pogPwOy><sup>20</sup> to play with this example.

D3 has a powerful system for managing transitions. You can alter things such as the animation duration and the easing function (which lets you control how quickly things accelerate and decelerate). You can also add delays to each element in order to achieve staggered transitions.

Transitions are very well implemented in D3 and its one of the features that stands D3 apart. We cover transitions in more detail later and use them in Energy Explorer.

## 4.2.6 Summary

Hopefully this chapter has given you a flavour of what D3 is and what it can do. One of the key takeaways is that D3 provides modules to help you build data visualisations. It doesn't provide ready built charts – use a library such as Chart.js for that.

This book covers a cross section of D3 functionality. It doesn't cover everything, but it covers enough to get you started building your own custom visualisations.

## 4.3 D3 versions

Since its inception in 2011, D3 has undergone a number of changes and at the time of writing is at version 7. This book uses **version 7** which was launched in June 2021, but the content also applies to version 6.



## Notes

15 <https://d3js.org/>

16 <https://www.highcharts.com/>

17 <https://www.chartjs.org/>

18 <https://codepen.io/createwithdata/pen/NWxjKVo>

19 <https://www.d3indepth.com/shapes/>

20 <https://codepen.io/createwithdata/pen/pogPwOy>

## 5. Requesting Data with D3

This chapter covers data requests using D3. This is a nice topic to start your D3 journey as it's not too hard to understand and it allows you to load some data into the browser (in order to visualise it).

First let's look at HTTP requests. **HTTP** stands for **Hypertext Transfer Protocol** and is a protocol (or 'agreement') used for communication between a web browser and web server. Any time a web browser needs to request a resource, be it an HTML file, a JPEG image or a CSV file, it uses an HTTP request. Typically the data (or resource) will have a URL (Uniform Resource Locator) such as `https://assets.codepen.io/2814973/my-csv.csv`. (The URL can also be relative to the `index.html` of your web application, so it'll often look like `data/my-csv.csv`.)



Beware that things get more complicated if the resource has a different domain (i.e. the `assets.codepen.io` part) to your web application due to [CORS](#)<sup>21</sup> restrictions. In this book, the data will be hosted on the same server so this isn't an issue.

D3's request methods make the HTTP request for you and parse the incoming data so that it's ready to use. (Parsing CSV files is non trivial so this saves a lot of work.)

In this chapter we look at four request methods: `d3.csv`, `d3.tsv`, `d3.dsv` and `d3.json`. These are the most common methods used when visualising data. (In fact, `d3.csv` and `d3.json` are by far the most commonly used.)

### 5.1 d3.csv

`d3.csv` makes an HTTP request for a comma-separated value (CSV) file and converts it into an array of objects. The syntax of `d3.csv` is:

```
d3.csv(url [, row])
```

`url` is a string containing the URL and `row` is an optional function that transforms each row of the CSV file. (The square brackets denote an optional argument.)

For example given a CSV file with URL `https://assets.codepen.io/2814973/my-csv.csv`:

<https://assets.codepen.io/2814973/my-csv.csv>

---

```
name,indicator1,indicator2
Paris,4030,13.45
Tokyo,3912,45.41
New York,19481,32.53
```

---

use the following to load it:

```
function dataIsReady(data) {
  console.log(data);
}

d3.csv('https://assets.codepen.io/2814973/my-csv.csv').then(dataIsReady);
```

`d3.csv` requests the CSV file. When the CSV loads, the function `dataIsReady` gets called. (This is known as a callback function.) The first parameter (`data`) of `dataIsReady` is an array of objects.



A more detailed explanation is: `d3.csv` evaluates to a ‘promise’. A [promise](#)<sup>22</sup> is an object that represents an ‘asynchronous’ operation. This means `d3.csv` makes the request **but doesn’t wait for the resource to arrive**. Instead the program continues to execute while the resource loads. The promise object has a method `.then` which lets you specify a callback function that’s called when the CSV file loads.

D3 converts the CSV file into an **array of objects** and passes it into the callback function. Each object corresponds to a row in the CSV file (excluding the header) and has properties corresponding to each name in the CSV header:

```
[
  {
    name: "Paris",
    indicator1: "4030",
    indicator2: "13.45"
  },
  {
    name: "Tokyo",
    indicator1: "3912",
    indicator2: "45.41"
  },
  {
```

```
    name: "New York",
    indicator1: "19481",
    indicator2: "32.53"
  }
]
```

Notice that the `indicator1` and `indicator2` property values are strings instead of numbers. D3 assumes everything in the CSV file is a string and doesn't convert anything automatically. It's a good idea to convert strings to numbers as soon as you can and this can be done by providing a function as the second argument to `d3.csv`:

```
function dataIsReady(data) {
  console.log(data);
}
```

```
function transformRow(row) {
  var ret = {
    name: row.name,
    indicator1: parseFloat(row.indicator1),
    indicator2: parseFloat(row.indicator2)
  };
}
```

```
  return ret;
}
```

```
d3.csv('https://assets.codepen.io/2814973/my-csv.csv', transformRow)
  .then(dataIsReady);
```

The `transformRow` function gets called for each row. Its parameter is an element from the array of objects that D3 generated from the CSV file. For example the first time `transformRow` is called, `row` is:

```
{
  name: "Paris",
  indicator1: "4030",
  indicator2: "13.45"
}
```

You then create a new object with any transformations you wish to apply and return it. In our example, we convert `indicator1` and `indicator2` into numbers using `parseFloat`. The array of objects that's passed into `dataIsReady` now consists of the transformed objects:

```
[
  {
    name: "Paris",
    indicator1: 4030,
    indicator2: 13.45
  },
  {
    name: "Tokyo",
    indicator1: 3912,
    indicator2: 45.41
  },
  {
    name: "New York",
    indicator1: 19481,
    indicator2: 32.53
  }
]
```

Each indicator is now a **number** instead of a string.

To recap:

- `d3.csv('my-url', transformRow)` requests a CSV file located at 'my-url' and returns a promise
- when the request fulfils, D3 converts the incoming CSV file into an array of objects and transforms each row using the function `transformRow`
- D3 then calls `dataIsReady`, passing in the array of transformed objects

## 5.2 d3.tsv

`d3.tsv` is the same as `d3.csv` except it processes the incoming file as a tab-separated value (TSV) file.

## 5.3 d3.dsv

`d3.dsv` is a generalised version of `d3.csv` and `d3.tsv` which processes the incoming file using a specified separator character. The first argument of `d3.dsv` is the separator character, so:

```
d3.dsv(',', 'my-url', transformRow);
```

is the equivalent of

```
d3.csv('my-url', transformRow);
```

## 5.4 d3.json

`d3.json` makes a request for a JSON file and parses it into a JavaScript array or object. Given a JSON file at <https://assets.codepen.io/2814973/my-json.json>:

<https://assets.codepen.io/2814973/my-json.json>

---

```
[
  {
    "name": "Paris",
    "indicator1": 4030,
    "indicator2": 13.45
  },
  {
    "name": "Tokyo",
    "indicator1": 3912,
    "indicator2": 45.41
  },
  {
    "name": "New York",
    "indicator1": 19481,
    "indicator2": 32.53
  }
]
```

---

You can request it using:

```
function dataIsReady(data) {
  console.log(data);
}

d3.json('https://assets.codepen.io/2814973/my-json.json')
  .then(dataIsReady);
```

Similarly to `d3.csv`, `d3.json` makes the request and evaluates to a promise. `.then(dataIsReady)` specifies that `dataIsReady` is called when the JSON file arrives. D3 converts the JSON file into a JavaScript array or object. JSON is closely related to JavaScript's arrays and objects so when you load a JSON file using `d3.json` you're creating an array or object that looks just like the JSON file.

## 5.5 CSV, TSV or JSON?

What's best: CSV, TSV or JSON? It depends. In my experience, CSV is most often used when the data has been **exported from a spreadsheet**. JSON is usually used to return results **from an API call** (see the next section). JSON also has the ability to represent more complex structures than CSV files (which are basically tables). (We use CSV to represent the data for Energy Explorer.)

## 5.6 Static resources and APIs

If the data you're visualising doesn't change very often then hosting it **statically** should suffice. Static hosting is where files (be it HTML, JavaScript, CSV or JSON files) live on a server's hard drive and are sent to a web browser when requested. For example the CSV file (at <https://assets.codepen.io/2814973/my-csv.csv>) is statically hosted.

If the data is frequently updated (or particularly complex) then it's often served via an API. An API (Application Programming Interface) is a program on a server that receives an HTTP request, makes a database query and returns the result (usually as a JSON file).

Whether your data is statically hosted or comes from an API it doesn't matter to D3. So long as the incoming data is in the agreed format (i.e. CSV if you're using `d3.csv`) D3 will load data from a static resource or an API. For simplicity the Energy Explorer will load data from a statically hosted CSV file.

## Notes

<sup>21</sup> <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

<sup>22</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

# 6. Practical: Load the Data

You learned about D3 requests in the previous section. In this practical we introduce the data that Energy Explorer uses and we show how to use D3 to load this data.

## 6.1 Overview

Open `d3-start-to-finish-code/step2` in the code download in your text editor. The directory structure is:

```
step2
├── data
│   └── data.csv
├── index.html
├── js
│   ├── lib
│   │   └── d3.min.js
│   └── main.js
```

New files and directories are shown in red. There are three new directories: `data`, `js` and `js/lib`. `data` is where the data is located and `js` is where JavaScript files live. `js/lib` contains libraries such as D3.

There are also three new files: `data.csv`, `d3.min.js` and `main.js`. The first is the CSV file containing data for Energy Explorer, `d3.min.js` is D3 and `main.js` is an empty file that'll contain the Energy Explorer JavaScript code.

In this practical we:

1. Inspect the CSV file `data/data.csv`.
2. Link to `js/main.js` and `js/lib/d3.min.js` in `index.html`.
3. Load `data/data.csv` using D3.

## 6.2 Inspect the data

Open `data/data.csv` in your text editor. It should look like:



**data/data.csv**


---

```
id,name,oilgascoal,nuclear,hydroelectric,renewable
AGO,Angola,46.8,,53.2,0.0
ALB,Albania,0.0,,100.0,0.0
ARE,United Arab Emirates,99.8,,0.0,0.2
ARG,Argentina,66.9,,26.2,1.9
ARM,Armenia,35.9,,28.3,0.1
...
```

---

This is the data that'll be visualised by Energy Explorer. Each row represents a country and has an id, a name and four indicators (hydroelectric, nuclear, oilgascoal and renewable). The indicators describe the energy mix (as a percentage) of a country. For example, 53.2% of Angola's electricity is provided by hydroelectrics (from water, such as wave power or a dam).

The data originates from the [World Bank's World Development Indicators](#)<sup>23</sup> database.

## 6.3 Include JavaScript files in index.html

Now open `index.html`. Add the two JavaScript files `d3.min.js` and `main.js` to the page by using the `<script>` element. This uses an attribute `src` to specify the path of the JavaScript file:

**index.html**


---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
  </head>

  <body>
    <script src="js/lib/d3.min.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

Now save `index.html`.

`d3.min.js` is the minified version of D3 and is located in the `js/lib` directory. (Minified means that the size of the library has been reduced by, for example, shortening the names

of variables.) We use version 7 of D3 in this book. `main.js` is where Energy Explorer's code will live.



There's a number of ways to include D3 in your project. We've opted for one of the simpler approaches which is to download the library from the [D3 homepage](#)<sup>24</sup> and link to it in `index.html`. When the page loads, the browser will open each JavaScript file in turn and execute it. For larger projects you can use 'bundlers' such as Webpack that can obtain library code for you and package it all together. Such tools add a layer of complexity and to minimise distraction I don't use them in this book.

## 6.4 Request data

We now request `data/data.csv` using `d3.csv`. Open `js/main.js` and add the following:

`js/main.js`

---

```
function dataIsReady(csv) {  
    console.log(csv);  
}  
  
d3.csv('data/data.csv')  
    .then(dataIsReady);
```

---

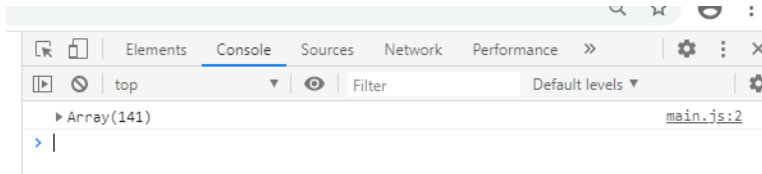
This code is very similar to the code in the D3 Requests chapter. `d3.csv` requests a CSV file at `data/data.csv`. (This URL is relative to `index.html` and it points at the `data.csv` file.) D3 converts the CSV file into an array of objects then calls `dataIsReady` which outputs the array to the console.

Now save `index.html` and `js/main.js` and load `step2` in your browser.



Make sure your browser is showing `step2`. If you're running a Node, Python, PHP or similar server, you might need to browse back to the directory list and open `step2`. If you're using Brackets, select Live Preview from the File menu.

You'll see a blank page in your browser. However if you open the Console within the Developer Tools of the browser you should see something like:



Developer Tools console showing the output of `dataIsReady`



In Google Chrome you can open the Developer Tools by either pressing F12 or pressing `ctrl-shift-j` (Windows) or `cmd-option-j` (Mac). (See [here](#)<sup>25</sup> for further information on opening the console in Chrome's developer tools. Similar instructions for Firefox are [here](#)<sup>26</sup>.)

The Console allows you to inspect useful information while you develop your application. You can write to the console using `console.log`. In our case, `dataIsReady` calls `console.log(csv)` which outputs the `csv` variable.

Within the console, expand the array and the first few elements. You should see something like:



Inspecting the data array

Can you see that this array corresponds to the `data.csv` file? (Each array object corresponds to a row of data.)

For example the first object in the array represents Angola:

```
[
  {
    "name": "Angola",
    "id": "AGO",
    "hydroelectric": "53.2",
    "nuclear": "",
    "oilgascoal": "46.8",
    "renewable": "0.0"
  },
  ...
]
```

and corresponds to the first row of data in data/data.csv:

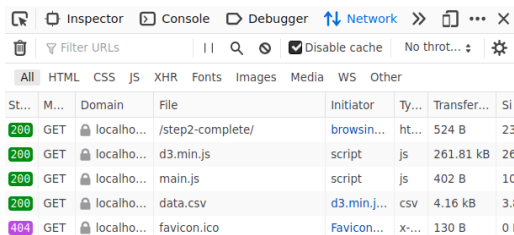
data/data.csv

```
id,name,oilgascoal,nuclear,hydroelectric,renewable
AGO,Angola,46.8,,53.2,0.0
ALB,Albania,0.0,,100.0,0.0
ARE,United Arab Emirates,99.8,,0.0,0.2
ARG,Argentina,66.9,,26.2,1.9
ARM,Armenia,35.9,,28.3,0.1
...
```

If you got this far congratulations – you’ve loaded the data into your application!

If not, check the following:

- are you definitely running a local webserver or the live server within Brackets?
- are you getting any errors on the developer tools console?
- try opening the Network tab of the developer tools and refreshing. You should see a list of files that the browser has loaded (see below). Make sure that data.csv is one of them
- try opening step2-complete which is the completed version of this section



| St... | M... | Domain       | File             | Initiator   | Ty... | Transfer... | Si  |
|-------|------|--------------|------------------|-------------|-------|-------------|-----|
| 200   | GET  | localhost... | /step2-complete/ | browsin...  | ht... | 524 B       | 23  |
| 200   | GET  | localhost... | d3.min.js        | script      | js    | 261.81 kB   | 26  |
| 200   | GET  | localhost... | main.js          | script      | js    | 402 B       | 10  |
| 200   | GET  | localhost... | data.csv         | d3.min.j... | csv   | 4.16 kB     | 3.1 |
| 404   | GET  | localhost... | favicon.ico      | Favicon...  | x-... | 130 B       | 0.1 |

### Inspecting the Network tab in Chrome's developer tools

This is one of the trickier steps to get working, so don't worry if you're having difficulty. Once this has been solved the remaining practicals shouldn't present similar issues.

## 6.5 Summary

In this section we used D3's `d3.csv` method to load the energy data. We added a function `dataIsReady` that's called when the data arrives. `dataIsReady` outputs the data to the console.

In the next chapter you learn how D3 adds, removes and manipulates HTML and SVG elements.

## Notes

23 <http://datatopics.worldbank.org/world-development-indicators/>

24 <https://d3js.org/>

25 <https://developers.google.com/web/tools/chrome-devtools/open>

26 [https://developer.mozilla.org/en-US/docs/Tools/Web\\_Console/Opening\\_the\\_Web\\_Console](https://developer.mozilla.org/en-US/docs/Tools/Web_Console/Opening_the_Web_Console)

# 7. D3 Selections

D3 selections are basically **arrays of HTML or SVG elements**. They let you **modify the style and attributes** of the HTML or SVG elements you've selected. For instance you can use D3 to select some SVG circles and change their colour to red. Selections are also used when **joining an array** to HTML or SVG elements (see the next chapter).

## 7.1 Creating a selection

D3 has two functions for making selections: `d3.select` and `d3.selectAll`.

`d3.select` selects a single element while `d3.selectAll` selects multiple elements.

### 7.1.1 `d3.select`

The syntax of `d3.select` is:

```
d3.select(selector);
```

where `selector` is a string containing a CSS selector string (for example `'#chart'`).



If you're not familiar with CSS selectors take a look at the CSS section in the [Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation book<sup>27</sup>](#).

`d3.select` selects the **first** element on the page that matches the selector string. It returns an object which has a number of methods (such as `.attr` and `.style`) that modify the selected HTML/SVG element. (A method is a function that's been assigned to a property of an object.)

For example, if you have some SVG circles on the page:

```
<circle></circle>
<circle></circle>
<circle></circle>
```

the following statement makes a selection containing just the first circle:

```
d3.select('circle');
```

You can assign a D3 selection to a variable. For example:

```
let s = d3.select('circle');
```

## 7.1.2 d3.selectAll

`d3.selectAll` is similar to `d3.select` but it selects **all** matching elements on the page. For example, if you have some SVG circles on the page:

```
<circle></circle>
<circle></circle>
<circle></circle>
```

the following statement makes a selection containing all three circles:

```
d3.selectAll('circle');
```

`d3.selectAll` returns an object which has a number of methods (such as `.attr` and `.style`) that modify all the selected HTML/SVG elements. The next section covers these methods in detail.

## 7.2 Updating a selection's elements

Once you've used `d3.select` or `d3.selectAll` to create a selection of HTML/SVG elements you can **modify the elements** using a number of methods. Four of the most commonly used selection methods are `.style`, `.attr`, `.classed` and `.text`.

| Method name           | Description  |
|-----------------------|--|
| <code>.style</code>   | Adds CSS rules to the selection's elements         |
| <code>.attr</code>    | Adds attributes to the selection's elements        |
| <code>.classed</code> | Adds a class attribute to the selection's elements |
| <code>.text</code>    | Sets the text content of the selection's elements  |

(If you're not familiar with CSS, HTML or SVG you can read up on them in the [Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation book](#)<sup>28</sup>.)

## 7.2.1 .style

The `.style` method **sets a CSS style property** on each element in a selection. The syntax is:

```
s.style(property, value);
```

where `s` is a D3 selection.

`property` is a string representing the CSS property (such as `color`, `fill` or `font-size`) and `value` is the value you're setting the property to and can be any valid CSS value such as `red`, `#ddd` or `12px`.

For example suppose you have the following SVG elements:

```
<circle r="10"></circle>
<circle r="20"></circle>
<circle r="30"></circle>
```

You can update the `fill` style of each circle using:

```
let s = d3.selectAll('circle');
s.style('fill', 'red');
```

This will change the fill colour of each circle to red. An alternative format that omits the variable declaration is:



```
d3.selectAll('circle')  
  .style('fill', 'red');
```

Each method call is on a separate line and indented. This is the usual way of formatting D3 code.

Under the hood, D3 adds inline CSS to the element(s). Inline CSS is where an attribute named `style` is added to the element:

```
<circle r="10" style="fill: red;"></circle>  
<circle r="20" style="fill: red;"></circle>  
<circle r="30" style="fill: red;"></circle>
```

If you need to add more than one style rule, you can **chain** `.style` calls:

```
d3.selectAll('circle')  
  .style('fill', 'red')  
  .style('stroke', '#333');
```

Method chaining is a technique where you can make multiple method calls by writing them one after the other. So instead of writing:

```
var s = d3.selectAll('circle');  
s.style('fill', 'red');  
s.style('stroke', '#333');
```

you write:

```
d3.selectAll('circle')  
  .style('fill', 'red')  
  .style('stroke', '#333');
```

Here's a CodePen example that demonstrates `.style`:

<https://codepen.io/createwithdata/pen/abvYowg><sup>29</sup>.

As an exercise, try adding another `.style` call to change the stroke of the circles to `#333`.

## 7.2.2 .attr

`.attr` sets an attribute on each element in a selection. The syntax is:

```
s.attr(name, value);
```

where `s` is a D3 selection.

`name` is a string representing the attribute name (for example `id`, `cx` or `width`) and `value` is the value you're setting the attribute to (for example `main-menu`, `10` or `100`).

For example suppose you have the following SVG elements:

```
<circle></circle>
<circle></circle>
<circle></circle>
```

You can update the `r` attribute of every circle using:

```
d3.selectAll('circle')
  .attr('r', 50);
```

This will change the radius of each circle to 50:

```
<circle r="50"></circle>
<circle r="50"></circle>
<circle r="50"></circle>
```

As with `.style` (and all other selection methods) you can chain method calls:

```
d3.selectAll('circle')
  .attr('cx', 200)
  .attr('cy', 100)
  .attr('r', 50);
```

Here's a CodePen example that uses `.attr` to set all the circle radii to 30:

<https://codepen.io/createwithdata/pen/GRZWPjM><sup>30</sup>

As an exercise, try using `.attr` to change the circles' vertical position (use the attribute `cy`) to 30.

## 7.2.3 .classed

You can **add and remove class attributes** to/from a selection's elements using `.classed`. The syntax is:

```
s.classed(className, value);
```

where `s` is a D3 selection.

`className` is the name of the class you're adding or removing and `value` indicates whether you're adding or removing the class attribute. If `value` is `true` the class is added to the element. If `value` is `false` the class is removed from the element.

For example to add a class attribute named `highlighted` to each element in a selection use:

```
d3.selectAll('circle')
  .classed('highlighted', true);
```

To remove a class attribute named `highlighted` use:

```
d3.selectAll('circle');
  .classed('highlighted', false);
```

You can add more than one class attribute by separating the class names with a space:

```
d3.selectAll('circle');
  .classed('item highlighted', true);
```

Under the hood `.classed` adds a class attribute to a selection's element(s):

```
<circle class="item highlighted"></circle>
```

Here's a CodePen example that uses `.classed` to add a class attribute named `highlighted` to each circle:

<https://codepen.io/createwithdata/pen/WNwpLog><sup>31</sup>

The CSS contains a rule that sets the fill colour for elements with class `highlighted`.

## 7.2.4 .text

The `.text` method lets you **set the text content** of each element in a selection. It's generally used on selections of HTML elements (such as `div`, `p` and `span`) and SVG text elements. The syntax is:

```
s.text(value);
```

where `s` is a D3 selection and `value` is the string you're setting the content to.

Suppose you have the following HTML:

```
<div></div>
<div></div>
```

You can set the text content of each `div` element using:

```
d3.selectAll('div')
  .text('Some content');
```

This results in the following change to the HTML:

```
<div>Some content</div>
<div>Some content</div>
```

Here's an example in CodePen: <https://codepen.io/createwithdata/pen/ExKWGWj><sup>32</sup>

## 7.3 Multiple updates

The `.style`, `.attr`, `.classed` and `.text` methods may be chained together which allows you to set multiple style, attribute and classes in a single statement. For example:

```
d3.selectAll('circle')
  .attr('cx', 100)
  .attr('cy', 100)
  .attr('r', 50)
  .style('fill', 'red')
  .classed('item', true);
```

### 7.3.1 .select and .selectAll

The `.style`, `.attr`, `.classed` and `.text` methods can be used on selections created using `.select` and `.selectAll`. All the above examples use `.selectAll`. If `.select` is used, only the first matching element will be selected. Try changing `.selectAll` to `.select` in the CodePen examples and you should see that only the first element on the page is modified.

## 7.4 Chained selections

You can make a selection of elements **within another selection** by **chaining** calls to `.select` and `.selectAll`. This concept is best explained by an example. Suppose you have the HTML:

```
<div class="first">
  <p></p>
  <p></p>
</div>
<div class="second">
  <p></p>
  <p></p>
</div>
```

You can select the `p` elements inside the first `div` by selecting the `div` and chaining a call to `selectAll`:

```
d3.select('div.first')
  .selectAll('p');
```

Similarly you can select the `p` elements from within the second `div` using:

```
d3.select('div.second')
  .selectAll('p');
```

Here's a Codepen containing these examples:

<https://codepen.io/createwithdata/pen/xxZYvBZ><sup>33</sup>

You might be wondering whether you can achieve something similar using a nested CSS selector. For example, you might think:

```
d3.selectAll('div.second p');
```

is equivalent to:

```
d3.select('div.second')
  .selectAll('p');
```

They're similar but there is a difference: when selections are chained **the selection keeps a record of the parent element**. In the previous code snippet the selection keeps a record of the parent element (`div.second`) of the selected `p` elements.

This will come in useful when joining data. For now, don't worry too much about this, but just remember that selections can be chained.

## Notes

27 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

28 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

29 <https://codepen.io/createwithdata/pen/abvYowg>

30 <https://codepen.io/createwithdata/pen/GRZWPjM>

31 <https://codepen.io/createwithdata/pen/WNwpLog>

32 <https://codepen.io/createwithdata/pen/ExKWWGj>

33 <https://codepen.io/createwithdata/pen/xxZYvBZ>

# 8. Data Joins

A fundamental concept in D3 is the **data join**. A data join creates a correspondence between an **array of data** and a **selection** of HTML or SVG elements. Joining an array to HTML/SVG elements means that:

- Each array element has a **corresponding** HTML/SVG element.
- Each HTML/SVG element may be **positioned**, **sized** and **styled** according to the value(s) of its corresponding array element.

## 8.1 Creating a data join

The general pattern for creating a data join is:

```
d3.select(container)
  .selectAll(element-type)
  .data(array)
  .join(element-type);
```

where:

- `container` is a CSS selector string that specifies an element that will **contain** the joined HTML/SVG elements
- `element-type` is a string describing the **type** of element you're joining (e.g. `'div'` or `'circle'`)
- `array` is the name of the **array** you're joining.

The container is a single element that'll act as the parent of the HTML/SVG elements that you're joining. Let's look at a specific example. Suppose you've an array of values:

```
let myData = [10, 40, 30];
```

and the following SVG:

```
<svg>
  <g class="chart">
  </g>
</svg>
```

We'll use the `g` element as the container and will join the array to `circle` elements. The end result will look like:

```
<svg>
  <g class="chart">
    <circle></circle>
    <circle></circle>
    <circle></circle>
  </g>
</svg>
```

The code for this is:

```
d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle');
```

The first line:

```
d3.select('g.chart')
```

makes a selection of a single HTML or SVG element that will be container (or 'parent') of the joined HTML or SVG elements.

The second line:

```
.selectAll('circle')
```

selects the HTML/SVG elements you're joining. In our example we're joining `circle` elements to the array so we select all `circle` elements. You always use `.selectAll` to make this selection.



Even though there are no circles on the page you can still make a selection of circles. The resulting selection is empty but it's still a valid selection. This is an aspect of D3 joins that can be hard to grasp so it can be confusing. Rest assured that once you get used to it it'll make more sense.

The third line:



```
.data(myData)
```

specifies the array you're wanting to join. In this example, we're joining the `myData` array. The final line:

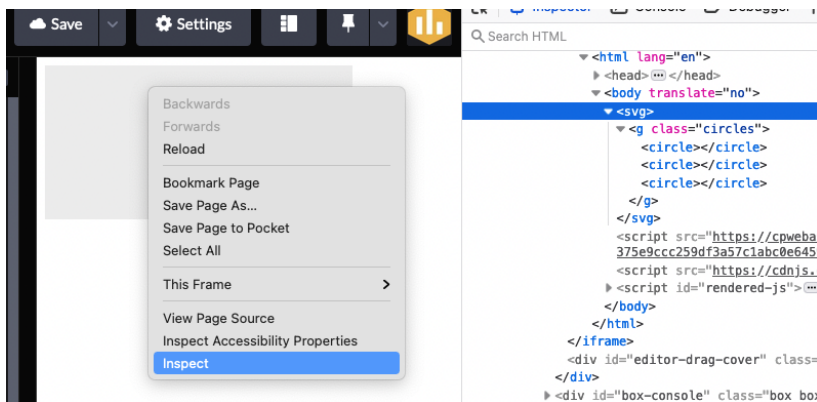
```
.join('circle');
```

creates the join. You pass in the type of HTML or SVG element you're wanting D3 to create. (In general the same element type will be passed into `.selectAll` and `.join`.)

When these four lines of code are executed `circle` elements are added or removed such that each array element has a corresponding `circle`.

Here's an example in Codepen: <https://codepen.io/createwithdata/pen/ZEQxyyJ><sup>34</sup>

You won't see any output because each circle has radius zero so will be invisible. However, if you right-click on the SVG element (which has been coloured light grey) and choose Inspect (or similar) you'll be able to see the page's elements. Expand the `svg` and `g` elements and you should see the three circles that have been added by D3:



Inspecting the `<svg>` element to check the circles have been added

### 8.1.1 Recap

A D3 data join creates a correspondence between an **array of values** and a **selection of HTML/SVG elements**. The general pattern for creating a join is:

```
d3.select(container)
  .selectAll(element-type)
  .data(array)
  .join(element-type);
```

The first line selects the container element. The second line selects the HTML/SVG elements you're joining. The third line specifies the array you're joining. The last line specifies the type of HTML/SVG element you're joining.

Each time these four lines are executed D3 adds (or removes) HTML/SVG elements such that **each array value has a corresponding HTML/SVG element**.

In the next section we'll look at how the joined HTML/SVG elements can be modified using `.style`, `.attr`, `.classed` and `.text`.

## 8.2 Updating the joined elements

Once you've joined an array to a selection of HTML/SVG elements you can update the position, colour, size etc. of each element using `.style`, `.attr`, `.classed` and `.text`. (These are the same four methods introduced in the 'Updating a selection's elements' section in the previous chapter.)

Broadly speaking there are three approaches to updating each HTML/SVG element. You can:

- set the style, attribute or content to a **constant value**
- use the **value of the corresponding array element** to set the style, attribute or content
- use the **index** (i.e. the position within the array) to set the style, attribute or content

We covered the first approach in the D3 Selections chapter.

The second approach is probably the most interesting. It allows you to set an element's style, attribute or text content based on the **value of the corresponding array element** (or the 'joined value'). This allows you to modify the selection's elements in a **data-driven** fashion.

The third approach allows you to set an element's style, attribute or text content based on the **array index** of the corresponding array element. This is typically used to evenly space HTML/SVG elements. For instance, you might use the array index to evenly space bars in a bar chart.

## 8.2.1 Constant value update

The **constant value** approach was covered in the D3 Selections chapter. As a reminder, suppose you've an array of values:

```
let myData = [10, 40, 30];
```

and the following SVG:

```
<g class="chart"></g>
```

You can join the array to `circle` elements using:

```
d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle');
```

You can set the radius of each circle to a value of 50 by chaining a call to `.attr`:

```
d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', 50);
```

Let's also set the x and y position of each circle to 100 and the fill of each circle to #af90ca:

```
d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', 50)
  .attr('cx', 100)
  .attr('cy', 100)
  .style('fill', '#af90ca');
```

Here's the example on CodePen:

<https://codepen.io/createwithdata/pen/LYGBOGx><sup>35</sup>



Joined (overlapping) circles updated with constant value

Although there are three circles, they are on top of one another because they have the same center point. As an exercise, try changing the `r`, `cx`, `cy` and `fill` values.

## 8.2.2 Joined value (or ‘data-driven’) update

The **joined value** of an HTML/SVG element is the value of its corresponding array element.

In order to use each circle’s joined value to update its style and attributes you **pass a function** into `.style`, `.attr`, `.classed` or `.text`. For each of the selection’s elements D3 passes the **joined value into the function as the first parameter**. By convention, the first parameter is named `d`. The function’s **return value** is used to set the style, attribute or content.

For example you can set the radius of each circle so that it’s equal to the joined value using:

```
let myData = [10, 40, 30];
```

```
d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  });
```

This results in the first circle getting a radius of 10, the second circle 40 and so on. This is known as a **data-driven update** and is a cornerstone of D3.

Let’s look at another example. You might want to set the circle radii to twice the joined value:

```
let myData = [10, 40, 30];

d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return 2 * d;
  });
```

For each circle in the selection D3 calls the function that's been passed into `.attr` and sets the circle's radius to the return value of the function.

Thus for the first circle, `d` is 10 so the function returns 20 and the circle's radius is set to 20. Then for the second circle `d` is 40 so the function returns 80 and the circle's radius is set to 80 and so on.

You can pass a function into any of `.style`, `.attr`, `.classed` or `.text`. For example let's use the joined value to set the circle fills:

```
let myData = [10, 40, 30];

d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return 2 * d;
  })
  .style('fill', function(d) {
    if(d >= 40) {
      return 'red';
    }
    return '#aaa';
  });
```

Now the circles will be coloured red if the joined value is greater or equal to 40 otherwise they'll be grey. Here's this example on CodePen:

<https://codepen.io/createwithdata/pen/pogZdYj><sup>36</sup>



### Joined circles with data-driven update

The circles are on top of one other so you can't see all the circles in this example. Ideally we'd space the circles evenly and this is when array index updates are used.

## 8.2.3 Array index update

The third approach to updating joined HTML/SVG elements is to use the **index of the corresponding array element** to update the style, attribute or content. When a function is passed into `.style`, `.attr`, `.classed` or `.text` the **second parameter** of the function is the array index of the joined value. By convention the second parameter is named `i`. The first circle has an index of 0, the second circle 1 and so on.

A common use case is to evenly space the HTML/SVG elements. (For example bar charts consist of evenly spaced rectangles.) Let's space our circles horizontally using `i`:

```
let myData = [10, 40, 30];

d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  })
  .attr('cx', function(d, i) {
    let circleSpacing = 100;
    return i * circleSpacing;
  });
```

This spaces the circles 100 pixels apart. (The first circle will have an x coordinate of 0, the second 100 and so on.) Here's the example on CodePen:

<https://codepen.io/createwithdata/pen/oNbMoOq><sup>37</sup>



### Joined circles with array index update

(The circles are cut off because the `cy` attribute hasn't been set and defaults to zero.) As an exercise, try changing the value of `circleSpacing`. Can you guess what will happen to the circle positions?

## 8.2.4 Examples

Let's put all three approaches together. We'll set:

- each circle's y position to 100
- the radius to 1.25 times the joined value
- each circle's x position to 100 times the index `i`
- each circle's fill to `#aaa`

The code for this is:

```
let myData = [10, 40, 30];

d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('cy', 100)
  .attr('r', function(d) {
    return 1.25 * d;
  })
  .attr('cx', function(d, i) {
    let circleSpacing = 100;
    return i * circleSpacing;
  })
  .style('fill', '#aaa');
```

The above code is a very typical pattern in D3. Here it is on CodePen:

<https://codepen.io/createwithdata/pen/ZEQxXwO><sup>38</sup>



Joined circles with constant, data-driven and index based updates

(The first circle is clipped because it's x coordinate is 0.)

Here's another example where a simple horizontal bar chart is created. The same array `myData` is used but `rect` elements are joined to the array rather than circles. Here's the SVG:

```
<g class="barchart"></g>
```

Each `rect` will have a width that's 4 times the joined value, a height of 45 and they'll be vertically spaced 50 pixels apart:

```
let myData = [10, 40, 30];

d3.select('g.barchart')
  .selectAll('rect')
  .data(myData)
  .join('rect')
  .attr('width', function(d) {
    return 4 * d;
  })
  .attr('height', 45)
  .attr('y', function(d, i) {
    let barSpacing = 50;
    return i * circleSpacing;
  })
  .style('fill', '#aaa');
```

You can view this example at: <https://codepen.io/createwithdata/pen/ZEQxXNp><sup>39</sup>



Joined rectangles with constant, data-driven and index based updates

Try changing the array values and you'll see the bar chart update accordingly. (Bear in mind that each time you modify code in CodePen, it automatically re-runs which is why you see the bars update.)

Hopefully you can begin to understand D3's data-driven philosophy. Unlike libraries such as Chart.js where you choose a chart type (bar chart, line chart etc.) D3 lets you choose HTML or SVG elements and manipulate them in a data-driven fashion. You can still create standard charts using this approach but the real power is that it allows you to create **custom charts of your own design**. The only limit is your imagination!



In the next section we delve deeper and learn how to join **arrays of objects**.

## 8.3 Joining arrays of objects

In the previous two sections we showed how to join an array of numbers to HTML/SVG elements. However data is usually **multi-variable** so we're usually dealing with arrays of objects such as:

```
[
  {
    name: "Paris",
    indicator1: 9030,
    indicator2: 13.45
  },
  {
    name: "Tokyo",
    indicator1: 3912,
    indicator2: 45.41
  },
  {
    name: "New York",
    indicator1: 19481,
    indicator2: 32.53
  }
]
```



It's worth noting that `d3.csv` (which we met in the Requesting data with D3 section) converts CSV files into this format.

An array of objects can be joined to HTML/SVG elements in the same manner as an array of values. The difference is that the **joined value is an object** rather than a number. Let's look at an example.

Suppose our SVG looks like:

```
<g class="cities"></g>
```

and our data array is:

```
let myData = [
  {
    name: "Paris",
    indicator1: 9030,
    indicator2: 13.45
  },
  {
    name: "Tokyo",
    indicator1: 3912,
    indicator2: 45.41
  },
  {
    name: "New York",
    indicator1: 19481,
    indicator2: 32.53
  }
];
```

We can create the join using a similar approach to before:

```
d3.select('g.cities')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    // return something here
  });
```

Each array element is an object so when we pass a function into `.attr` the first parameter of the function (`d`) is an **object**. For example `d` might look like:

```
{
  name: "Paris",
  indicator1: 9030,
  indicator2: 13.45
}
```

This means we need to access the **object's properties** (for example `d.name` or `d.indicator1`) instead of `d` alone. Let's use `indicator2` to set the radius of the circles:

```
d3.select('g.cities')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d.indicator2;
  });
```

Here's the example on CodePen: <https://codepen.io/createwithdata/pen/MWKVNMx><sup>40</sup>



Joining array of objects to circle elements



In the above CodePen example we're also updating `cx` and `cy`. Note also there's a transform applied to the `g` element to stop the circles getting cropped (look inside the HTML tab).

We can begin to see the power of D3 as we now have multiple variables which can drive the position, size and style of HTML/SVG elements. For example we can create a simple scatter plot by setting the x coordinate of each circle relative to `indicator1` and the y coordinate relative to `indicator2`:

```
d3.select('g.cities')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('cx', function(d) {
    return 0.01 * d.indicator1;
  })
  .attr('cy', function(d) {
    return 2 * d.indicator2;
  })
  .attr('r', function(d) {
    return 5;
  });
```

You can view this example on Codepen at: <https://codepen.io/createwithdata/pen/vYLjBBx><sup>41</sup>



### Joining array of objects to create (upside down) scatter plot



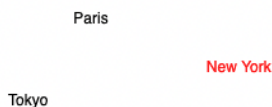
Note the above example creates an ‘upside down’ scatter plot because an SVG’s y-axis points downwards.

Here’s a final example where `myArray` is joined to `text` elements:

```
d3.select('g.cities')
  .selectAll('text')
  .data(myData)
  .join('text')
  .attr('x', function(d) {
    return 0.01 * d.indicator1;
  })
  .attr('y', function(d) {
    return 2 * d.indicator2;
  })
  .classed('high', function(d) {
    return d.indicator1 > 10000;
  })
  .text(function(d) {
    return d.name;
  });
```

In this example we’re joining `myArray` to `text` elements. The `x` and `y` attributes are set in the same manner as the previous scatter plot.

We’re using `.classed` to set add a `high` class attribute if `indicator1` is greater than 10,000. We’re also using `.text` to set the content of the `text` element to the city name.



### <text> elements joined to an array of objects

Here’s the example on CodePen: <https://codepen.io/createwithdata/pen/VwexZvP42>

### 8.3.1 Summary

This section has shown how to join an **array of objects** to HTML/SVG elements. We saw the only difference is that the **joined value is now an object** rather than a number.

This means that when a function is passed into `.style`, `.attr`, `.classed` or `.text` you have to use the object properties (such as `d.indicator1` and `d.name`) rather than `d` itself.

You've also seen that joining an array of objects gives you a wide range of possibilities for setting the style, attributes and content of HTML/SVG elements as you have multiple variables to play with.

## Notes

34 <https://codepen.io/createwithdata/pen/ZEQxyyJ>

35 <https://codepen.io/createwithdata/pen/LYGBOGx>

36 <https://codepen.io/createwithdata/pen/pogZdYj>

37 <https://codepen.io/createwithdata/pen/oNbMoOq>

38 <https://codepen.io/createwithdata/pen/ZEQxXwO>

39 <https://codepen.io/createwithdata/pen/ZEQxXNp>

40 <https://codepen.io/createwithdata/pen/MWKVNMx>

41 <https://codepen.io/createwithdata/pen/vYLjBBx>

42 <https://codepen.io/createwithdata/pen/VwexZvP>

# 9. Practical: Draw the Data

In this chapter we modify Energy Explorer to draw a **single circle for each country**.

The end result will look like:

.....

Circles joined to the energy data

It's not the most exciting of data visualisations, but each circle represents a country. Later on we show how to set the radius of each circle according to a data variable.

## 9.1 Overview

Open `d3-start-to-finish-code/step3` in the code download. The file structure is:

```
step3
├── data
│   └── data.csv
├── index.html
└── js
    ├── lib
    │   └── d3.min.js
    └── main.js
```

Recall that in the previous practical we used D3's `.csv` function to load `data/data.csv`.

In this practical we:

1. Add a container for the circles in `index.html`.
2. Join the energy mix data to `<circle>` elements.

## 9.2 Add a container for the circles

In `index.html` add a container for the SVG `<circle>` elements and set its width and height to 1200.

Now add a `<g>` element inside the `<svg>` element and give it an `id` attribute of value `chart`. This element will contain the circles:

**index.html**


---

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
  </head>
  <body>
    <svg width="1200" height="1200">
      <g id="chart"></g>
    </svg>
    <script src="js/lib/d3.min.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>

```

---



At this point we're not sure what the size of the chart will be, but setting the width and height to 1200 pixels is a good starting point.

## 9.3 Join the data array to circle elements

Make the following changes in `js/main.js`:

`js/main.js`

---

```

let data;

function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 10;
    })
    .attr('cy', 100)
    .attr('r', 2);
}

```

```

function dataIsReady(csv) {
  —console.log(csv);
}

```

```
data = csv;
update();
}

d3.csv('data/data.csv')
  .then(dataIsReady);
```

---

We've added a global variable `data` at the top of the file. In the `dataIsReady` function we assign the array of data (`csv`) to this new variable.



The `data` variable is a global variable because it sits outside of any function and is accessible everywhere in the code.

We also create a new function `update` that uses D3 to join `data` to `<circle>` elements. This new function is called from `dataIsReady`. The code inside `update` is similar to the code we covered in the previous sections on data joins. Let's go through it step by step.

First we select the container element using `d3.select('#chart')` and then call `.selectAll` to select the SVG elements we're joining.

We then specify the array that we're joining using `.data(data)` method. (We're calling a method called `.data` and passing the global variable `data` into it.)

Next we call `.join('circle')` to perform the join. (This will cause a `<circle>` element to be created for each array element.)

Finally we call `.attr` to update the `cx`, `cy` and `r` attributes of the circles. The `cx` attribute is set according to the array index `i`. This results in evenly spaced circles. The `cy` and `r` attributes are set to constants:

To summarise, when Energy Explorer loads, it'll load the CSV file using `d3.csv` and assign the resulting data array to the global variable `data`. It'll then call `update` which joins `data` to circles resulting in a row of circles appearing in the browser.

## 9.4 Save and refresh

Save `index.html` and `main.js` and make sure your browser is loading `step3`.



Make sure that your browser is showing `step3`. If you're running a Node, Python, PHP or similar server, you might need to browse back to the directory list and open `step3`. If you're using Brackets, select Live Preview from the File menu.



You should now see the following in your browser:

.....

#### Circles joined to the energy data

Each circle represents a country. There should be 141 circles on the page but some of them will be invisible because they go beyond the dimensions of the SVG element.

The complete code for this section can be found in the `step3-complete` directory.

The next step in building Energy Explorer will be to size each circle according to one of the underlying data variables. But before you do this you'll learn about **scale functions** in the next chapter.

# 10. Scale Functions

Scale functions help you transform your **data values** into **visual values**.

**Data values** are the values of your data. For example, in Energy Explorer the data values are the percentage values for each of the four energy types.

**Visual values** are visual properties such as position, size and colour. For example an x coordinate of 50px, a height of 100px or a colour of #aaa.

Suppose you have an array of numbers representing percentage values:

```
let data = [15, 76, 41, 67, 97];
```

and you want to create a bar chart where the maximum length of the bars is 500px.

A scale function which takes a number between 0 and 100 and returns a value between 0 and 500 would help here and D3 can create it for you. This saves you having to figure out the mathematics and code yourself.

D3 has around twelve scale types. Some accept **numbers** as input, others accept **strings**. Some output **numbers**, others output **colours**. In this book we cover two scale types `scaleLinear` and `scaleSqrt`.

`scaleLinear` is by far the most common D3 scale type and has several use cases. `scaleSqrt` is particularly useful when working with circles. We'll use it to size circles in Energy Explorer.

## 10.1 scaleLinear

`d3.scaleLinear` creates linear scale functions. A linear scale takes a **number** as input and returns a **number**. Use the following to create a very simple linear scale function:

```
let myScale = d3.scaleLinear();
```

The return value of `d3.scaleLinear` is a function (which we've assigned to `myScale`). By default this function takes an input value and returns the same value:

```
myScale(0);    // returns 0
myScale(1);    // returns 1
myScale(50);   // returns 50
myScale(100);  // returns 100
```

Not very useful, but you can define the input and output ranges using `.domain` and `.range`, respectively. D3's terminology uses **domain** to represent the input minimum and maximum and **range** to represent the output minimum and maximum. Let's set the domain to between 0 and 100 and the range to between 0 and 1000:

```
myScale.domain([0, 100]).range([0, 1000]);
```

(You pass an array with two values into `.domain` and `.range`. The first value is the minimum and the second value the maximum.)

Now look at the output values:

```
myScale(0);    // returns 0
myScale(1);    // returns 10
myScale(50);   // returns 500
myScale(100);  // returns 1000
```

Our linear scale function `myScale` takes an input between 0 and 100 (the domain) and linearly maps it to an output between 0 and 1000 (the range).



If you plot the output against the input of a **linear** scale you get a straight line.

You can also pass colours into the range:

```
myScale.range(['white', 'red']);
```

This results in:

```
myScale(0);    // returns "rgb(255, 255, 255)"
myScale(1);    // returns "rgb(255, 252, 252)"
myScale(50);   // returns "rgb(255, 128, 128)"
myScale(100);  // returns "rgb(255, 0, 0)"
```

Now the scale is returning colours ranging from white (`rgb(255, 255, 255)`) to red (`rgb(255, 0, 0)`). By default D3 scale functions extrapolate the output if you input values outside of the domain:

```
myScale.domain([0, 100]).range([0, 1000]);  
myScale(200);    // returns 2000  
myScale(-100);   // returns -1000
```

However you can ‘clamp’ the scale function so that the output stays within the range:

```
myScale.clamp(true);  
myScale(200);    // returns 1000  
myScale(-100);   // returns 0
```

You can try out `scaleLinear` at `jsconsole.com` by visiting <https://jsconsole.com><sup>43</sup>. First include D3 using `jsconsole.com`’s `:load` command:

```
:load https://cdnjs.cloudflare.com/ajax/libs/d3/7.4.4/d3.min.js
```

Then type (or copy and paste) the following:

```
let myScale = d3.scaleLinear();  
myScale.domain([0, 100]).range([0, 1000]);
```

Now if you enter `myScale(0)` you should see `0` output. If you enter `myScale(100)` you should see `1000` output.

```
Use :help to show jsconsole commands  
version: 2.1.2  
  
Loading script...  
  
Loaded https://cdnjs.cloudflare.com/ajax/libs/d3/7.4.4/d3.min.js  
  
> var myScale = d3.scaleLinear();  
   myScale.domain([0, 100]).range([0, 1000]);  
  
< f l(n) { ... }  
  
> myScale(0)  
  
< 0  
  
> myScale(100)  
  
< 1000  
  
>
```

## 10.2 scaleSqrt

`d3.scaleSqrt` is similar to `d3.scaleLinear` except the **square root** of the input value is used. This is particularly useful when using circles to represent a quantity. It's widely accepted when visualising data that the **area** of a circle (rather than the radius of a circle) should be proportional to a value. This potentially tricky calculation is taken care of by D3's `scaleSqrt` scale. Suppose we have a value that varies between 0 and 1 and we wish to represent it with a circle with maximum radius 100 we create a square root scale using:

```
let myScale = d3.scaleSqrt().domain([0, 1]).range([0, 100]);
myScale(0.5); // returns 70.71067811865476
myScale(1); // returns 100
```



Recalling circle mathematics, the area of a circle is  $\pi * \text{radius} * \text{radius}$ . Thus the area of a circle representing a value of 0.5 is  $\pi * 70.7106 * 70.7106 = 15707.96$  and the area of the circle representing 1 is  $\pi * 100 * 100 = 31415.93$ . The second circle is twice the area of the first circle.

The main thing to remember is that if you're using circle area to represent a value, use `scaleSqrt`.

## 10.3 Putting scale functions to use

Consider the following code which performs a data join and sets the circle radius according to the joined value:

```
let myData = [10, 40, 30];

d3.select('g.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  });
```

This scales the circle **radius** proportionally to the joined value `d`. As we've just discussed this isn't recommended – you should scale **area** proportionally to the joined value.

This is achieved by applying a `scaleSqrt` scale:

```
let myData = [10, 40, 30];  
let radiusScale = d3.scaleSqrt().domain([0, 50]).range([0, 50]);  
  
d3.select('g.chart')  
  .selectAll('circle')  
  .data(myData)  
  .join('circle')  
  .attr('r', function(d) {  
    return radiusScale(d);  
  });
```

Notice that the function passed into `.attr` is now using `radiusScale`.

Here's a similar example in CodePen: <https://codepen.io/createwithdata/pen/yLeqRLp><sup>44</sup>  
The second data value (40) is twice the first data value (20). This results in the second circle having twice the area of the first circle.

Now change `scaleSqrt` to `scaleLinear` in the CodePen example. When `scaleLinear` is used it's fairly clear that the second circle's area is much more than twice the first circle's, even though its value is only 2 times bigger. We'll use `scaleSqrt` to set the circle sizes in the Energy Explorer in the following practical.

## Notes

<sup>43</sup> <https://jsconsole.com>

<sup>44</sup> <https://codepen.io/createwithdata/pen/yLeqRLp>

# 11. Practical: Size the Circles

In this chapter we modify Energy Explorer to size each circle according to its **renewable energy** percentage. (This is the percentage of a country's energy mix that comes from renewable energy.)

By the end of this practical Energy Explorer will look like:



The energy data represented by circles, sized according to each country's renewable energy percentage

## 11.1 Overview

Open `d3-start-to-finish-code/step4`. The file structure is:

```
step4
├── data
│   └── data.csv
├── index.html
└── js
    ├── lib
    │   └── d3.min.js
    └── main.js
```

Recall that in the previous practical we joined the energy mix array to circle elements.

In this practical we:

1. Convert the indicator values from strings into **numbers**.
2. Create a `scaleSqrt` **scale function** and use it to **set the radius** of each circle.

## 11.2 Convert indicator values from strings into numbers

When `d3.csv` loads a CSV file it treats all values (including numbers) as strings. Therefore we need to convert each of the indicator values (`hydroelectric`, `nuclear`, `oilgascoal`

and renewable) into numbers using `parseFloat`. We use a transformation function (as explained in the Requesting Data with D3 chapter). Make the following changes in `js/main.js`:

`js/main.js`

---

```
let data;

...

function dataIsReady(csv) {
  data = csv;
  update();
}

function transformRow(d) {
  return {
    name: d.name,
    id: d.id,
    hydroelectric: parseFloat(d.hydroelectric),
    nuclear: parseFloat(d.nuclear),
    oilgascoal: parseFloat(d.oilgascoal),
    renewable: parseFloat(d.renewable)
  };
}

d3.csv('data/data.csv', transformRow)
  .then(dataIsReady);
```

---

`transformRow` is a new function that converts the four indicator values into numbers using `parseFloat`. (If the value is missing, `parseFloat` returns `NaN` which stands for ‘not a number’.)

For example, if the following is passed into `transformRow`:



```
{
  hydroelectric: "53.2",
  id: "AGO",
  name: "Angola",
  nuclear: "",
  oilgascoal: "46.8",
  renewable: "0.0"
}
```

the following is returned:

```
{
  hydroelectric: 53.2,
  id: "AGO",
  name: "Angola",
  nuclear: NaN,
  oilgascoal: 46.8,
  renewable: 0.0
}
```

## 11.3 Create a sqrtScale function and set the circle radii

In `js/main.js` make the following changes:

`js/main.js`

---

```
let data;
```

```
let radiusScale = d3.scaleSqrt()
  .domain([0, 100])
  .range([0, 20]);
```

```
function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 10;
    })
    .attr('cy', 100)
    .attr('r', 2);
    .attr('r', function(d) {
```

```

        return radiusScale(d.renewable);
    });
}

function dataIsReady(csv) {
    data = csv;
    update();
}

function transformRow(d) {
    return {
        name: d.name,
        id: d.id,
        hydroelectric: parseFloat(d.hydroelectric),
        nuclear: parseFloat(d.nuclear),
        oilgascoal: parseFloat(d.oilgascoal),
        renewable: parseFloat(d.renewable)
    };
}

d3.csv('data/data.csv', transformRow)
    .then(dataIsReady);

```

---

A `scaleSqrt` scale function is created with domain `[0, 100]` and range `[0, 20]` and assigned to `radiusScale`. The domain is set to `[0, 100]` because the data values vary between 0 and 100 and the range is set to `[0, 20]` so that the maximum circle radius is 20. We'll set this to a more suitable value later in the book. We saw in the previous chapter that `scaleSqrt` is the correct scale to use when using circle area to represent a quantity.

We now pass a function into `.attr('r', ...)` so that the radius is set according to the joined value (instead of a constant value). For now we use the `renewable` property and apply the `radiusScale` scale function to it.

Therefore each circle will be sized such that its area is proportional to the `renewable` property.

Save `main.js` and load `step4` in your browser. The circles now look like:



Circles sized according to energy indicator

Each circle represents a county and is sized according to its renewable energy percentage. We can already see that there are a few countries that appear to have quite a high percentage and plenty others with a low percentage.

Later on in this book we'll arrange the circles more effectively and add labels so we can see the individual countries more clearly.

# 12. Architecture for Web-based Data Visualisation

This chapter introduces two techniques for architecting interactive data visualisations for code clarity and maintainability. These techniques become increasingly important as an application's size increases and they are of significant benefit to Energy Explorer.

The first technique is to use a layout function that takes an array of data and outputs a new array containing information (such as position and size) for rendering the visualisation. The benefit is that the logic for calculating the visual variables (such as position, size and colour) is separate to the rendering code. This makes maintaining and understanding the code easier and also makes porting the code from D3 to a different rendering engine (such as React) simpler.

The second technique is to modularise the code. This means that the code is split over several files, with each file responsible for a particular aspect of the application. This makes the application easier to understand and maintain.

## 12.1 Layout functions

A layout function is a function that **takes an array of data** and **returns a new array** containing position, size and other visual variables for each item. For example, given the data:

```
let myData = [30, 10, 20, 40];
```

a layout function could return an array containing position and radius data:

```
[
  { x: 100, y: 100, r: 30 },
  { x: 200, y: 100, r: 10 },
  { x: 300, y: 100, r: 20 },
  { x: 400, y: 100, r: 40 }
]
```

Each object in the above array corresponds to an element in `myData`. The layout function that produces the above is:

```
function layout(data) {  
  let layoutData = data.map(function(d, i) {  
    let item = {};  
    item.x = 100 + i * 100;  
    item.y = 100;  
    item.r = d;  
    return item;  
  });  
  
  return layoutData;  
}
```

This function generates a new array `layoutData` by calling `map` on `data`. The `map` function iterates through `data` and computes `x`, `y` and `r` for each array element. The result is a new array containing the position and radius data.



See the JavaScript Iteration chapter of the [Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation book<sup>45</sup>](#) if you're not familiar with `map`.

Typically you assign the output of `layout(myData)` to a new variable:

```
let layoutData = layout(myData);
```

and join `layoutData` to HTML or SVG elements:

```
d3.select('#chart')  
  .selectAll('circle')  
  .data(layoutData)  
  .join('circle')  
  .attr('cx', function(d) { return d.x; })  
  .attr('cy', function(d) { return d.y; })  
  .attr('r', function(d) { return d.r; });
```

There are a few benefits of this approach:

- the join code is much **cleaner and easier to read** because it doesn't contain any code for computing size, positions and other visual variables
- the **layout and rendering code is decoupled**, so swapping D3 for a different rendering library (such as React or Vue) will have lower impact
- **debugging is easier** because you can easily inspect the output of the layout function

- it's relatively easy to run **automated tests** on the layout function

Here's a similar example on CodePen: <https://codepen.io/createwithdata/pen/gOaezWJ><sup>46</sup>

You don't have to use layout functions but for more complex visualisations they can make your code easier to understand and maintain. D3 also provides a number of layout functions but it adds geometric information directly on each array element. The approach in this section creates a new array and doesn't change (or mutate) the original array. This is more in keeping with modern JavaScript practice.

## 12.2 Modules

A **module** is a chunk of code that has a **single purpose** and typically occupies a single file. It's common practice to use modules when developing applications (especially for medium to large applications) as it makes your code easier to understand and maintain. Modules are often re-usable so they allow you to use other people's code in your own application. (D3 itself is modular. It has modules for creating selections, for scale functions, for drawing axes and for many other purposes.)

### 12.2.1 JavaScript modules

JavaScript modules have a long and complicated history. (To see for yourself, have a look at [Exploring JS's section on modules](#)<sup>47</sup>.) In this section we look at two approaches:

- script loading
- ES6 modules

### 12.2.2 Script loading

This is a simple way to modularise your code. It's straightforward and reliable but not suitable for large applications. (Once you have more than 5 or 6 modules you ought to look at the other approaches.) You can split your code up into separate files and load them individually using `<script>` tags in `index.html`. The load order is important: each file is **loaded** and **executed** in the same order as the `<script>` tags.

A simple example consists of two JavaScript files `add.js` and `main.js`. Here's `add.js`:

**add.js**

---

```
function add(a, b) {  
  return a + b;  
}
```

---

and here's `main.js`:

**main.js**

---

```
let n1 = 10, n2 = 20;  
let sum = add(n1, n2);  
console.log(sum);
```

---

They can be included in `index.html` using:

**index.html**

---

```
<html>  
...  
<body>  
...  
  <script src="add.js"></script>  
  <script src="main.js"></script>  
</body>  
</html>
```

---

When the page loads in the browser `add.js` loads and executes. This results in the function `add` being defined. Next `main.js` loads and executes. This defines three variables `n1`, `n2` and `sum`. The latter is set using the `add` function. The `sum` variable is output to the browser's debug console.

We use this approach in Energy Explorer as it's simple, it'll work in all browsers and doesn't require any additional tooling.

## 12.2.3 ES6 modules

JavaScript is an evolving language and one of the most significant releases was ES6 (also known as ECMAScript 2015) which has a built-in module system. Currently it's fairly well supported in modern browsers but not in Microsoft's Internet Explorer 11. Therefore to use ES6 modules you either drop support of older browsers or use tooling (such as [Webpack](#)<sup>48</sup>) to transform your code into an older JavaScript version named ES5 which is more widely supported. With ES6 modules you explicitly export and import JavaScript primitives (such as numbers, strings arrays, objects and functions).

For example you can export a function from a file `add.js` using:

add.js

---

```
function add(a, b) {  
  return a + b;  
}  
  
export { add };
```

---

You can then import the function using import:

```
import { add } from './add.js';  
  
let n1 = 10, n2 = 20;  
console.log(add(n1, n2));
```

Currently browser support isn't complete so you'd typically use a bundler such as Webpack that convert your files to ES5. Module support is a rapidly evolving area and it can be hard to keep up. It looks like the end game is built-in support for modules in all browsers but we aren't quite there yet.

The focus of this book is to learn D3 and in order to minimise distractions we use the simple script loading approach in Energy Explorer.

## Notes

45 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

46 <https://codepen.io/createwithdata/pen/gOaezWJ>

47 [https://exploringjs.com/es6/ch\\_modules.html](https://exploringjs.com/es6/ch_modules.html)

48 <https://webpack.js.org/>



# 13. Practical: Add Modules

This practical modularises Energy Explorer by splitting it into separate JavaScript files. It also adds a layout function that'll be used by the update function. This is very much a refactoring exercise and there won't be any difference in the output. However it sets the groundwork for subsequent additions.

## 13.1 Overview

Open `d3-start-to-finish-code/step5`. The file structure is:

```
step5
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ lib
    │   └─ d3.min.js
    └─ main.js
```

In this practical we:

1. Add two new empty modules: `layout.js` and `update.js`.
2. Add a layout function to `layout.js`.
3. Move the update code from `main.js` to `update.js`
4. Modify the update function to use the layout function.

## 13.2 Add new modules

Create two new files `layout.js` and `update.js` in the `js` directory. The file structure should now look like:

step5-complete

```
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ layout.js
    └─ lib
        └─ d3.min.js
    └─ main.js
    └─ update.js
```

Now link to these modules in index.html:

**index.html**

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
  </head>

  <body>
    <svg width="1200" height="1200">
      <g id="chart"></g>
    </svg>

    <script src="js/lib/d3.min.js"></script>

    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

## 13.3 Add a layout function

Add a layout function to js/layout.js:

js/layout.js

---

```
function layout(data) {
  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, 20]);

  let layoutData = data.map(function(d, i) {
    let item = {};

    item.x = i * 10;
    item.y = 100;
    item.radius = radiusScale(d.renewable);

    return item;
  });

  return layoutData;
}
```

---

This takes a similar form to the layout functions in the previous chapter. The scale function `radiusScale` and the geometric calculations for `x`, `y` and `radius` are lifted from the update function (`js/main.js`).

The layout function accepts an array where each element looks something like:

```
{
  "name": "United Arab Emirates",
  "id": "ARE",
  "hydroelectric": 0,
  "nuclear": null,
  "oilgascoal": 99.8,
  "renewable": 0.2
}
```

and outputs a new array where each object represents the position and radius of the country's circle and looks something like:

```
{
  "x": 20,
  "y": 100,
  "radius": 0.89442719099999159
}
```

The output of `layout` will be used in the new update function (see later).

## 13.4 Move update code

Now delete `radiusScale` from `js/main.js` (because it's now defined in `js/layout.js`) and **cut** (so you can paste it elsewhere) the update function from `js/main.js` :

`js/main.js`

---

```
let data;

let radiusScale = d3.scaleSqrt()
  .domain([0, 100])
  .range([0, 20]);

function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 10;
    })
    .attr('cy', 100)
    .attr('r', function(d) {
      return radiusScale(d.renewable);
    });
}

function dataIsReady(csv) {
  data = csv;
  update();
}

function transformRow(d) {
  return {
    name: d.name,
    id: d.id,
    hydroelectric: parseFloat(d.hydroelectric),
    nuclear: parseFloat(d.nuclear),
    oilgascoal: parseFloat(d.oilgascoal),
    renewable: parseFloat(d.renewable)
  };
}

d3.csv('data/data.csv', transformRow)
  .then(dataIsReady);
```

---

Paste the update function you've just cut into `js/update.js`:

`js/update.js`

---

```
function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 10;
    })
    .attr('cy', 100)
    .attr('r', function(d) {
      return radiusScale(d.renewable);
    });
}
```

---

## 13.5 Use layout function

Modify the update function `update` (in `js/update.js`) so that it uses the layout function:

`js/update.js`

---

```
function update() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('circle')
    .data(layoutData)
    .join('circle')
    .attr('cx', function(d) {
      return d.x;
    })
    .attr('cy', function(d) {
      return d.y;
    })
    .attr('r', function(d) {
      return d.radius;
    });
}
```

---

## 13.6 Save and refresh

Save all the modified files: `index.html`, `js/main.js`, `js/update.js` and `js/layout.js`.

Refresh your browser (making sure it's loading `step5`) and you should see the same circles as before:



Output after modularising Energy Explorer

If this hasn't worked for you, see if you can spot the problem. Perhaps you can add `console.log(layoutData)` after calling `layout(data)` in the update function. Check that it outputs an array of objects containing position information. Failing that, compare your code with `step5-complete`.

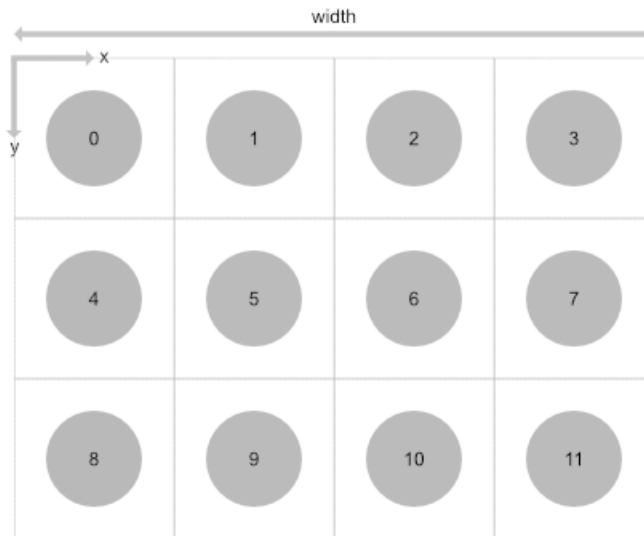
# 14. Arranging Items in a Grid

Arranging a list of items in a grid is a fairly common exercise but it isn't always obvious how to do this so this chapter covers the underlying mathematics and code.

Given an array of numbers such as:

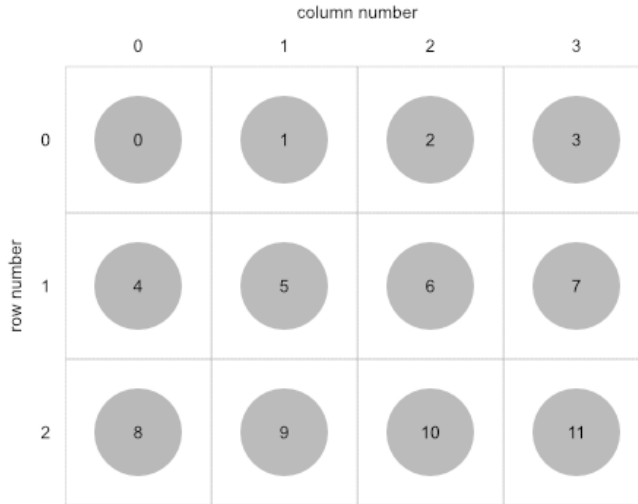
```
[10, 60, 40, 30, 70, 40, 30, 30, 10, 60, 50, 10]
```

we'd like to represent the array using a grid of circles:



Circles arranged in a grid (with 4 columns)

The number within each circle indicates the array index. The first circle is top left and represents the first element of the array. The grid has 4 columns and we'd like to calculate the centers of each circle relative to the origin (where the x and y axes meet). Let's start by figuring out the row number and column number of each cell. The first row and first column are numbered zero:



Grid of circles with row and column numbers

We can calculate the column number using JavaScript's modulo operator `%`. This returns the **remainder** of a division.

For example, for the 9th circle,  $9 \% 4$  returns 1 because the remainder of  $9 \div 4$  is 1. Thus the 9th circle is in column 1. (The number after the `%` is the **divisor**.)

If we evaluate the modulo operator with divisor 4 on numbers up to 10 we get:

| Modulo operation | Result |
|------------------|--------|
| $0 \% 4$         | 0      |
| $1 \% 4$         | 1      |
| $2 \% 4$         | 2      |
| $3 \% 4$         | 3      |
| $4 \% 4$         | 0      |
| $5 \% 4$         | 1      |
| $6 \% 4$         | 2      |
| $7 \% 4$         | 3      |
| $8 \% 4$         | 0      |
| $9 \% 4$         | 1      |



| Modulo operation | Result |
|------------------|--------|
| $10 \% 4$        | 2      |

The result of the modulo operation wraps around every  $n$  times (where  $n$  is the divisor). Thus we can use `%` to compute the column number from the array index  $i$ :

```
let column = i % numColumns;
```

where `numColumns` is the number of columns.

We can calculate the row number using division together with JavaScript's `Math.floor` function. `Math.floor` returns the nearest whole number below a given number. For example:

```
Math.floor(7.65); // returns 7
```

If we divide (using a divisor of 4) each number up to 10 and apply `Math.floor` we get:

| Operation   | Result of $i \div n$ | Result of <code>Math.floor(<math>i \div n</math>)</code> |
|-------------|----------------------|--|
| $0 \div 4$  | 0                    | 0  |
| $1 \div 4$  | 0.25                 | 0  |
| $2 \div 4$  | 0.5                  | 0  |
| $3 \div 4$  | 0.75                 | 0  |
| $4 \div 4$  | 1                    | 1  |
| $5 \div 4$  | 1.25                 | 1  |
| $6 \div 4$  | 1.5                  | 1  |
| $7 \div 4$  | 1.75                 | 1  |
| $8 \div 4$  | 2                    | 2  |
| $9 \div 4$  | 2.25                 | 2  |
| $10 \div 4$ | 2.5                  | 2  |

The right hand column starts at 0 and increments every 4 rows. In general, the result of

`Math.floor(i / n)` increments every `n` times. (The JavaScript symbol for division is `/`.) Thus we can use the following to calculate the row number from the array index `i`:

```
let row = Math.floor(i / numColumns);
```

Now we've figured out how to compute the row and column numbers let's compute the coordinates of the top left corner of each square cell.

The width (`cellWidth`) of each square cell is `width ÷ numColumns` where `width` is the overall width and `numColumns` is the number of columns:

```
let width = 1000;  
let numColumns = 4;  
let cellWidth = width / numColumns;
```

We can compute the x coordinate of the top left of a cell by multiplying the cell's column number `column` by `cellWidth`:

```
let x = column * cellWidth;
```

Each circle is located in the center of its square cell, so we can compute the the x coordinate of a circle's center by adding `0.5 * cellWidth` to `x`:

```
let x = column * cellWidth + 0.5 * cellWidth;
```

The y coordinate is computed in a similar manner:

```
let cellHeight = cellWidth; // the height is the same as the width  
let row = Math.floor(i / numColumns);  
let y = row * cellHeight + 0.5 * cellHeight;
```

Therefore the complete code for computing the circle centers is:

```
let data = [10, 60, 40, 30, 70, 40, 30, 30, 10, 60, 50, 10];
let width = 1000;
let numColumns = 4;

// Compute cell width and height
let cellWidth = width / numColumns;
let cellHeight = cellWidth;

// Iterate through array
data.forEach(function(d, i) {
  // Compute column and row number
  let column = i % numColumns;
  let row = Math.floor(i / numColumns);

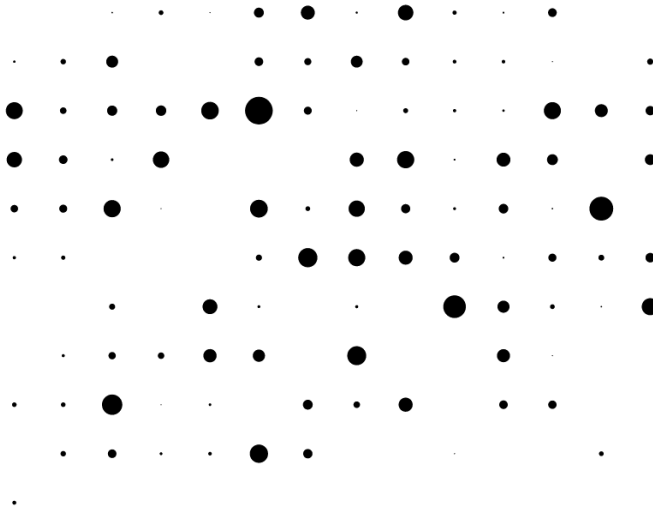
  // Compute circle center
  let x = column * cellWidth + 0.5 * cellWidth;
  let y = row * cellHeight + 0.5 * cellHeight;
});
```

To recap, we iterate through the array of data. The column number is calculated using the modulo operator `%` and the row number using division and `Math.floor`. The x coordinate of each circle is then calculated by multiplying the column number by `cellWidth` and adding another `0.5 * cellWidth` (to locate the circle in the center of the cell). Likewise for the y coordinate.

In the next practical you'll apply a similar technique to Energy Explorer.

# 15. Practical: Arrange the Data

In this practical you'll arrange the countries in a grid format:



Grid of circles

The previous chapter explained the mathematics and code to arrange an array of data in a grid and we use a similar approach in Energy Explorer.

## 15.1 Overview

Open `d3-start-to-finish-code/step6`. The file structure is:

```
step6
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    ├─ main.js
    └─ update.js
```

In this practical we:

1. Add a new module `config.js` for configuring Energy Explorer.
2. Modify the layout function to arrange the circles in a grid.

## 15.2 Add configuration object

Create a new file within the `js` directory and name it `config.js`.

Your directory structure should now look like:

```
step6
├─ data
│   └─ data.csv
├─ index.html
└─ js
    └─ config.js
        ├─ layout.js
        └─ lib
            └─ d3.min.js
                ├─ main.js
                └─ update.js
```

Include the new file `config.js` in the application by adding a new `<script>` tag to `index.html`:

**index.html**

---

```
<!DOCTYPE html>
<html lang="en">
...
  <body>
...
    <script src="js/lib/d3.min.js"></script>
    <script src="js/config.js"></script>
    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

Now add an object named `config` to the new file which stores properties for the chart width and the number of columns:

js/config.js

---

```
var config = {  
  width: 1200,  
  numColumns: 14  
};
```

---

## 15.3 Modify layout function

Make the following changes to js/layout.js:

js/layout.js

---

```
function layout(data) {  
  let cellWidth = config.width / config.numColumns;  
  let cellHeight = cellWidth;  
  
  let maxRadius = 0.35 * cellWidth;  
  
  let radiusScale = d3.scaleSqrt()  
    .domain([0, 100])  
    .range([0, 20]);  
  .range([0, maxRadius]);  
  
  let layoutData = data.map(function(d, i) {  
    let item = {};  
  
    let column = i % config.numColumns;  
    let row = Math.floor(i / config.numColumns);  
    item.x = i * 10;  
    item.y = 100;  
    item.x = column * cellWidth + 0.5 * cellWidth;  
    item.y = row * cellHeight + 0.5 * cellHeight;  
    item.radius = radiusScale(d.renewable);  
  
    return item;  
  });  
  
  return layoutData;  
}
```

---

Three new variables have been added to the layout function:

- cellWidth (the width of each grid cell)

- `cellHeight` (the height of each grid cell)
- `maxRadius` (the maximum circle radius)

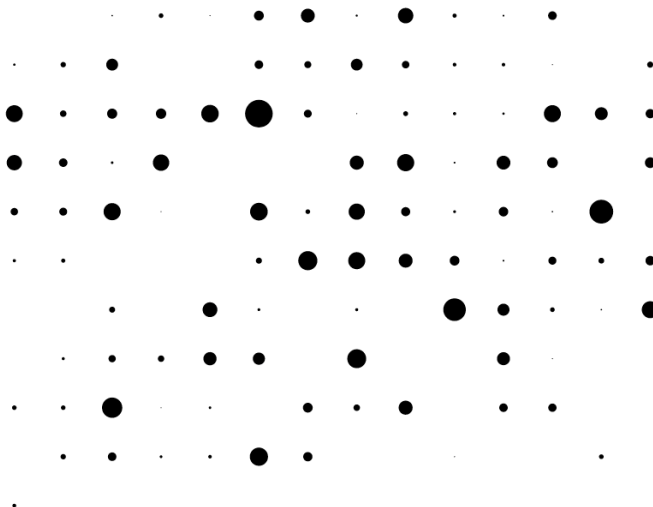
`cellWidth` and `cellHeight` are derived from the chart width (`config.width`) and number of columns (`config.numColumns`). `maxRadius` defines the maximum circle size. It's set to 0.35 times `cellWidth` which should give a nice amount of space between each circle. (If it's set to 0.5, the circles will just touch one another.)

In the function that's passed into `data.map`, we compute the column and row number based on the index `i` and the number of columns (see previous chapter). The `x` and `y` properties are set so that each circle occupies the center of its grid cell (see previous chapter).

## 15.4 Save and refresh

Now save `index.html`, `config.js` and `layout.js` and refresh your browser (making sure it's pointing at `step6`).

You should see:



Country circles arranged in a grid

By modifying the layout function you've changed the way in which the circles are arranged. Notice that you didn't have to change any of the data join code in `update.js`!

Try changing the value of `numColumns` in `config.js`. For example, set it to 30 and you should see:



Grid of circles with 30 columns

Notice that the circles adapt their size to the cell size. (This is because `maxRadius` is derived from the cell size.)

## 15.5 Summary

Hopefully you can also see the benefit of the architecture we've chosen. You're able to change a single number in a configuration object and the visualisation layout adapts in a sensible manner. You were also able to change the arrangement of the circles without having to touch the data join code.



# 16. The Build So Far

This chapter gives an overview of the Energy Explorer build so far. The directory structure of step6-complete is:

```
step6-complete
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    ├─ main.js
    └─ update.js
```

The data directory contains a single file `data.csv` which contains the data for Energy Explorer. It's a CSV file where each row represents a country:

**data/data.csv**

---

```
id,name,oil,gas,coal,nuclear,hydroelectric,renewable
AGO,Angola,46.8,,53.2,0.0
ALB,Albania,0.0,,100.0,0.0
ARE,United Arab Emirates,99.8,,0.0,0.2
ARG,Argentina,66.9,,26.2,1.9
ARM,Armenia,35.9,,28.3,0.1
...
```

---

Not including D3, there are four JavaScript files in the `js` directory: `main.js`, `update.js`, `layout.js` and `config.js`.

The first statement that executes is the call to `d3.csv` in `main.js` which loads `data/data.csv`:

**js/main.js**

---

```
let data;

function dataIsReady(csv) {
  data = csv;
  update();
}

function transformRow(d) {
  return {
    name: d.name,
    id: d.id,
    hydroelectric: parseFloat(d.hydroelectric),
    nuclear: parseFloat(d.nuclear),
    oilgascoal: parseFloat(d.oilgascoal),
    renewable: parseFloat(d.renewable)
  };
}

d3.csv('data/data.csv', transformRow)
  .then(dataIsReady);
```

---

When the data loads, D3 converts the CSV into an array of objects and calls `dataIsReady`. This assigns the array to the global variable `data`. `dataIsReady` also calls `update` which is responsible for updating the circles.

`update` (`js/update.js`) starts by calling the `layout` function and assigning the output to `layoutData`:

**js/update.js**

---

```
function update() {
  let layoutData = layout(data);
  ...
}
```

---

The `layout` function (`js/layout.js`) iterates through `data` and returns an array of circle positions and radii:

## js/layout.js

---

```
function layout(data) {
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth;

  let maxRadius = 0.35 * cellWidth;

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let layoutData = data.map(function(d, i) {
    let item = {};

    let column = i % config.numColumns;
    let row = Math.floor(i / config.numColumns);

    item.x = column * cellWidth + 0.5 * cellWidth;
    item.y = row * cellHeight + 0.5 * cellHeight;
    item.radius = radiusScale(d.renewable);

    return item;
  });

  return layoutData;
}
```

---

The input of layout looks like:

```
[
  {
    "name": "Angola",
    "id": "AGO",
    "hydroelectric": 53.2,
    "nuclear": null,
    "oilgascoal": 46.8,
    "renewable": 0
  },
  {
    "name": "Albania",
    "id": "ALB",
    "hydroelectric": 100,
    "nuclear": null,
    "oilgascoal": 0,
    "renewable": 0
  }
]
```

```

    },
    {
      "name": "United Arab Emirates",
      "id": "ARE",
      "hydroelectric": 0,
      "nuclear": null,
      "oilgascoal": 99.8,
      "renewable": 0.2
    },
    ...
  ]

```

and the output of `layout` looks like:

```

[
  {
    "x": 42.857142857142854,
    "y": 42.857142857142854,
    "radius": 0
  },
  {
    "x": 128.57142857142856,
    "y": 42.857142857142854,
    "radius": 0
  },
  {
    "x": 214.28571428571428,
    "y": 42.857142857142854,
    "radius": 1.3416407864998736
  },
  ...
]

```

In function `update` the output of `layout` is assigned to `layoutData`. `update` then joins `layoutData` to a selection of circles and updates the circle attributes using `layoutData`'s `x`, `y` and `radius` properties:

**js/update.js**

---

```
function update() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('circle')
    .data(layoutData)
    .join('circle')
    .attr('cx', function(d) {
      return d.x;
    })
    .attr('cy', function(d) {
      return d.y;
    })
    .attr('r', function(d) {
      return d.radius;
    });
}
```

---

This results in a grid of circles representing each country (and sized according to the renewable indicator) being drawn on the screen.

Make sure you're fairly comfortable with this overview before moving on. Perhaps look through `step6-complete` in your code editor and see if you can understand most of the code.

# 17. More on D3 Selections

This chapter covers additional methods you can call on a D3 selection. It also covers **update functions** which are handy for keeping your HTML/SVG elements synchronised with your data.

## 17.1 More selection methods

In this section we'll look at some additional methods you can call on a D3 selection. The main ones to take note of are `.append` and `.each` as you'll use them later in Energy Explorer.

### 17.1.1 `.size`, `.empty`, `.node` & `.nodes`

The following code examples assume the SVG is:

```
<circle r="10"></circle>
<circle r="20"></circle>
<circle r="30"></circle>
```

and that `s` is a D3 selection of all the circles:

```
let s = d3.selectAll('circle');
```

You can query the number of elements in a selection using `.size`:

```
s.size(); // 3
```

You can check whether a selection is empty using `.empty`:

```
s.empty(); // false
```

You can get the first HTML/SVG element in a selection using `.node`:

```
s.node(); // <circle r="10"></circle>
```



The `.node` method returns a DOM element. This is a JavaScript object that represents an HTML or SVG element in the document.

Finally you can get an array containing all the elements using `.nodes`:

```
s.nodes(); // [<circle r="10" />, <circle r="20" />, <circle r="30" />]
```

You can see these four methods in action at <https://codepen.io/createwithdata/pen/NWxaZyM><sup>49</sup>. (Open your browser's Developer Tools console to see the output.)

|   |             |
|---|-------------|
| 3   | pen.js:57:9 |
| false                                       | pen.js:58:9 |
| ▶ <circle r="10"> ⚙                         | pen.js:59:9 |
| ▶ Array(3) [ circle ⚙, circle ⚙, circle ⚙ ] | pen.js:60:9 |

Example output of `.size`, `.empty`, `.node` and `.nodes` selection methods

## 17.1.2 .append & .remove

You can add a child element to each element of a selection using `.append`. The syntax of `.append` is:

```
s.append(elementType)
```

where `s` is a selection and `elementType` is the type of element you wish to append. For example suppose the HTML and SVG looks like:

```
<g></g>
<g></g>
```

You can append an SVG circle to each `g` element using:

```
let s = d3.selectAll('g');
s.append('circle');
```

This will result in a circle inside both `g` elements:

```
<g>
  <circle></circle>
</g>
<g>
  <circle></circle>
</g>
```

You can also remove elements using `.remove`. For example:

```
let s = d3.selectAll('g');
s.remove();
```

which removes both `g` elements.

## 17.1.3 .each

`.each` lets you call a function on each element in a selection. It's analogous to JavaScript's `.forEach` method. The syntax of `.each` is:

```
s.each(fn)
```

where `s` is a D3 selection and `fn` is a function. Within the function, the current HTML or SVG element is assigned to the `this` keyword. Suppose you have:

```
<circle r="10"></circle>
<circle r="20"></circle>
<circle r="30"></circle>
```

You can iterate over each circle using:

```
let s = d3.selectAll('circle');

s.each(function() {
  console.log(this);
});
```

In the above example `this` is being output to the console. As previously explained, the HTML or SVG element of the current iteration step is assigned to `this` so the following (or similar) will be output in the console:



```
<circle r="10"></circle>
<circle r="20"></circle>
<circle r="30"></circle>
```

A common pattern within the callback function is to use D3 to select the element assigned to this:

```
d3.select(this);
```

The above allows you to modify the element. For example:

```
let s = d3.selectAll('circle');
s.each(function() {
  d3.select(this)
    .style('fill', 'red');
});
```

changes the fill of each circle to red. This is equivalent to:

```
let s = d3.selectAll('circle')
  .style('fill', 'red');
```

## 17.1.4 .on

Another selection method of interest is `.on` which lets you attach event handlers to a selection's elements. We'll cover this in more detail in the D3 Event Handling chapter.

## 17.2 Update functions

You've learned how to join an array of values to HTML/SVG elements using code similar to:

```
let myData = [10, 40, 30];

d3.select('g.circles')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr(...)
  .style(...)
  etc.
```

If the array changes length or its values change you might expect the joined HTML/SVG elements to update accordingly. However **D3 doesn't automatically update the HTML/SVG elements** each time the array changes. Instead you have to re-join the array and update the style and attributes each time the array changes. Therefore a common pattern is to put the join and update code in its own function (known as an **update function**):

```
function update() {
  d3.select('g.circles')
    .selectAll('circle')
    .data(myData)
    .join('circle')
    .attr(...)
    .style(...)
    etc.
}
```

Each time the update function executes, HTML/SVG elements will be added, removed and updated to match the array. Generally this function should be called each time the **data changes** or some **user interaction** has occurred.

Let's look at an example. Suppose the SVG is:

```
<svg>
  <g class="circles" transform="translate(50, 0)">
    </g>
</svg>
```

Now let's add an array `myData` and a function that updates `myData` with random data:

```

let myData = [];

function updateData() {
  let maxItems = 5, maxValue = 25;
  myData = [];
  let numItems = Math.ceil(Math.random() * maxItems);
  for(let i = 0; i < numItems; i++) {
    myData.push(Math.random() * maxValue);
  }
}

```



updateData updates myData with an array of random length. Each array element is a random number between 0 and maxValue.

Now we'll define an update function that joins myData to circle elements. Each circle has a radius that's 2 times it's joined value and will be equally spaced in a horizontal direction:

```

function update() {
  d3.select('g.circles')
    .selectAll('circle')
    .data(myData)
    .join('circle')
    .attr('cy', 100)
    .attr('r', function(d) {
      return 2 * d;
    })
    .attr('cx', function(d, i) {
      let circleSpacing = 100;
      return i * circleSpacing;
    })
    .style('fill', '#aaa');
}

```

Now let's set up a timer so that updateData and update are called every second (1000 milliseconds):

```
window.setInterval(function() {  
  updateData();  
  update();  
}, 1000);
```

When this example runs, `myData` changes every second and the circles are added, removed and updated accordingly.



`window.setInterval` sets up a timer that calls the supplied function at regular intervals. The first argument is the function and the second argument the interval in milliseconds.

Each time the timer fires (once every second) `updateData` and `update` are called. `updateData` clears `myData` and then adds a random number of random values to `myData`. `update` performs the data join which adds or removes circles so that there are `n` circles where `n` is the length of `myData`. The `cx`, `cy` and `r` attributes of each circle are also updated.

Navigate to <https://codepen.io/createwithdata/pen/Yzwaarg><sup>50</sup> to view the example in CodePen. You'll see that every time the timer fires, circles appear, disappear or change size.

## 17.3 Summary

This chapter has introduced some additional methods on D3 selections and has shown how you can create an **update function** which updates HTML/SVG elements in a data-driven fashion. Typically the update function is called whenever the **data changes** or when some **user interaction** has occurred.

This pattern is very common when building data visualisations with D3 and will be used in Energy Explorer.

## Notes

<sup>49</sup> <https://codepen.io/createwithdata/pen/NWxaZyM>

<sup>50</sup> <https://codepen.io/createwithdata/pen/Yzwaarg>

# 18. More on D3 Joins

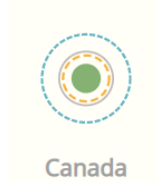
This chapter covers a couple of advanced techniques when joining data.

We first look at joining arrays to groups of elements, so that each array element corresponds to a group of elements. This is a common scenario because you often need to represent data points with more than one HTML/SVG element (for example, a labelled circle requires two elements). Energy Explorer uses this technique so it's worth taking some time to understand. (This is one of the most advanced topics in this book.)

The second technique is for joining nested arrays to nested elements. This is quite an advanced technique and isn't used by Energy Explorer so feel free to skip.

## 18.1 Joining an array to groups of elements

The Energy Explorer currently displays a **single circle** for each country. However we would like each country to be represented by **4 circles** (one for each of the energy types) and a **text label**. For example, Canada which uses all 4 energy types should look like:



Each country is represented by 4 circles and a text element

A sensible approach is to:

- create a single SVG `<g>` element for each country
- add four `<circle>` elements and a single `<text>` element to the `<g>` element

so that each country looks something like:

```

<g class="country">
  <circle class="renewable"></circle>
  <circle class="oilgascoal"></circle>
  <circle class="hydroelectric"></circle>
  <circle class="nuclear"></circle>
  <text class="label"></text>
</g>

```

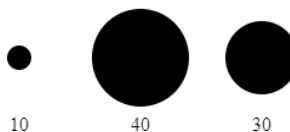


Remember a `<g>` element represents a group of SVG elements.

There's more than one way of achieving this using D3 and we'll look at a simple approach that doesn't require any additional D3 knowledge. Let's start with an easier example. Suppose you have some data:

```
let myData = [10, 40, 30];
```

and you'd like to join the array to `<g>` elements containing a `<circle>` and `<text>` element. And suppose you'd also like to size the circle using the data value and populate the text element with the data value (so that it acts as a label):



Array of numbers joined to a `<g>` element containing a `<circle>` and `<text>` element

We'd like the HTML/SVG to look like:

```

<g class="chart">
  <g transform="translate(0, 0)">
    <circle r="10"></circle>
    <text y="60">10</text>
  </g>
  <g transform="translate(100, 0)">
    <circle r="40"></circle>
    <text y="60">40</text>
  </g>
  <g transform="translate(200, 0)">
    <circle r="30"></circle>
    <text y="60">30</text>
  </g>
</g>

```

```

    </g>
  </g>

```

Let's start by joining `myData` to `<g>` elements:

```
let myData = [10, 40, 30];
```

```

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g');

```

The above is similar to the join code seen in previous sections but the array is joined to `<g>` elements instead of `<circle>` elements. This will create the following elements:

```

<g></g>
<g></g>
<g></g>

```

We now need to add a `<circle>` and `<text>` element to each of the `<g>` elements.

We can do this by adding a new function `updateGroup` which gets called on each `<g>` element using D3's `.each` function:

```
let myData = [10, 40, 30];
```

```
function updateGroup() {
  var g = d3.select(this);
```

```

    if(g.selectAll('*').empty()) {
      g.append('circle');
      g.append('text')
        .attr('y', 60);
    }
  }
}

```

```

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g')
  .each(updateGroup);

```



We covered the `.each` method and `d3.select(this)` in the previous chapter.

`updateGroup` gets called for each `<g>` element. It begins by selecting `this` (which represents the `<g>` element `updateGroup` has been called on) and assigns the resulting selection to the variable `g`.

`updateGroup` then checks whether the group `<g>` is empty. If it is, it appends a `<circle>` and `<text>` element to the group. It also sets the `y` attribute of the `<text>` element. In effect we're **initialising** each of the `<g>` elements.



`g.selectAll('*')` makes a selection containing all the child elements of the `<g>` element.

`g.selectAll('*').empty()` returns true if `<g>` has no child elements.

This code creates the following SVG:

```
<g>
  <circle></circle>
  <text y="60"></text>
</g>
<g>
  <circle></circle>
  <text y="60"></text>
</g>
<g>
  <circle></circle>
  <text y="60"></text>
</g>
```

Now we'll modify `updateGroup` to update:

- the `transform` attribute on the `<g>` element (so that each `<g>` element is translated proportionally to the index `i`)
- the radius of the `<circle>` element to the joined value `d`
- the content of the `<text>` element to the joined value `d`:



```

let myData = [10, 40, 30];

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) {
    g.append('circle');
    g.append('text')
      .attr('y', 60);
  }

  let x = i * 100;
  g.attr('transform', 'translate(' + x + ', 0)');

  g.select('circle')
    .attr('r', d);

  g.select('text')
    .text(d);
}

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g')
  .each(updateGroup);

```



Remember that D3 passes in the joined data *d* and index *i* to *updateGroup*. The first time *updateGroup* is called, *d* will be 10 and *i* will be 0.

This results in the following SVG:

```

<g class="chart">
  <g transform="translate(0, 0)">
    <circle r="10"></circle>
    <text y="60">10</text>
  </g>
  <g transform="translate(100, 0)">
    <circle r="40"></circle>
    <text y="60">40</text>
  </g>
  <g transform="translate(200, 0)">
    <circle r="30"></circle>

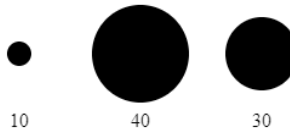
```

```

    <text y="60">30</text>
  </g>
</g>

```

and looks like:



An array of numbers joined to a `<g>` element containing `<circle>` and `<text>` elements



Notice we **transform** the `<g>` element rather than setting the x coordinate of the `<circle>` and `<text>` elements individually. This is a common technique when working with a group of elements. It's usually easier (and more expressive) to **transform the group as a whole** and position the constituent elements relative to the group.

Navigate to <https://codepen.io/createwithdata/pen/RwWmwel><sup>51</sup> to view the example in CodePen. (There's a bit of additional CSS and the `g.chart` element has been transformed in the order to prevent clipping.)

The same example but using an update function (so that the data join is called each time the data changes) is at <https://codepen.io/createwithdata/pen/abdXJYv><sup>52</sup>.

## 18.2 Nested joins

This section shows how to join a nested array (in other words, an array of arrays) to HTML or SVG elements.



Although Energy Explorer doesn't use this technique, it's a useful technique that isn't widely known. However feel free to skip ahead if it's not of interest!

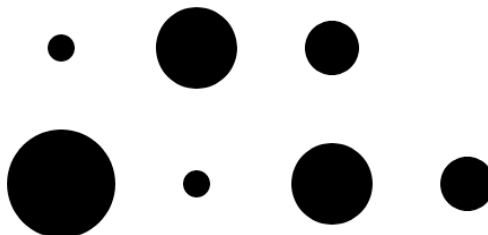
Suppose you have this nested array:

```
[
  [10, 30, 20],
  [40, 10, 30, 20]
]
```

The aim is to join this to nested HTML/SVG elements. For example, you might want to join the outer array to `<g>` elements and the inner arrays to `<circle>` elements. This would result in something like:

```
<g transform="translate(0,0)">
  <circle cx="0" r="10"></circle>
  <circle cx="100" r="30"></circle>
  <circle cx="200" r="20"></circle>
</g>
<g transform="translate(0,100)">
  <circle cx="0" r="40"></circle>
  <circle cx="100" r="10"></circle>
  <circle cx="200" r="30"></circle>
  <circle cx="300" r="20"></circle>
</g>
```

This would look like:



Nested array joined to `g` and `circle` elements

Each `<g>` element represents a row of circles. Each `<circle>` element is spaced horizontally by 100 pixels and sized according to the inner array values.

## 18.2.1 Overview

The approach is to join the outer array to `<g>` elements. This means that each `<g>`'s joined value is one of the inner arrays. We then iterate through the `<g>` elements and join the inner arrays to `<circle>` elements.

## 18.2.2 Join outer array to `<g>` elements

Assume there's an `<svg>` and `<g>` element on the page that'll act as the container:

```
<svg width="1200" height="1200">
  <g class="chart" transform="translate(50, 50)"></g>
</svg>
```

Let's join an array of arrays named `myData` to `<g>` elements:

```
let myData = [
  [10, 30, 20],
  [40, 10, 30, 20]
];

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g');
```

We'll name this join the **outer** join. This will create the following elements:

```
<g class="chart" transform="translate(50, 50)">
  <g></g>
  <g></g>
</g>
```



The first `<g>` element's joined value is the array `[10, 30, 20]`. The second `<g>` element's joined value is the array `[40, 10, 30, 20]`.

## 18.2.3 Add `updateGroup` function

Now we add a new function `updateGroup` and call it on each `<g>` element using D3's `.each` method. This is similar to what we did when joining arrays to groups of elements earlier in this chapter.

```

let myData = [
  [10, 30, 20],
  [40, 10, 30, 20]
];

function updateGroup(d, i) {
  let g = d3.select(this);
  g.attr('transform', 'translate(0,' + i * 100 + ')');
}

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g')
  .each(updateGroup);

```

`updateGroup` gets called for each element in `myData`. Each time it's called, the joined value is passed in as the first parameter and the index as the second parameter. Therefore the first time `updateGroup` is called, `d` is the array `[10, 30, 20]` and `i` is `0`. The second time it's called, `d` is the array `[40, 10, 30, 20]` and `i` is `1`.

A selection containing the `<g>` element is assigned to the variable `g` (using `d3.select(this)`). The `<g>` element is then translated by `(0, i * 100)` where `i` is the index of the `<g>` element. Therefore the first `<g>` element is translated by `(0,0)` and the second one by `(0,100)`.

The resulting SVG looks like:

```

<g class="chart" transform="translate(50, 50)">
  <g transform="translate(0,0)"></g>
  <g transform="translate(0,100)"></g>
</g>

```

## 18.2.4 Join `d` to `circle` elements

We now join `d` to `<circle>` elements using standard join code. We'll name this the **inner join**:

```
var myData = [
  [10, 30, 20],
  [40, 10, 30, 20]
];

function updateGroup(d, i) {
  var g = d3.select(this);
  g.attr('transform', 'translate(0,' + i * 100 + ')');

  g.selectAll('circle')
    .data(d)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 100;
    })
    .attr('r', function(d) {
      return d;
    });
}

d3.select('g.chart')
  .selectAll('g')
  .data(myData)
  .join('g')
  .each(updateGroup);
```

The important thing to note is we're now joining one of the **inner arrays** to `<circle>` elements.

`g.selectAll('circle')` selects all `circle` elements within the `<g>` element. (The first time this is called, this will be an empty selection. Return to the D3 Selections chapter to read more on this.)

The next line `.data(d)` specifies that we're joining `d` (which is one of the inner arrays such as `[10, 30, 20]`.)

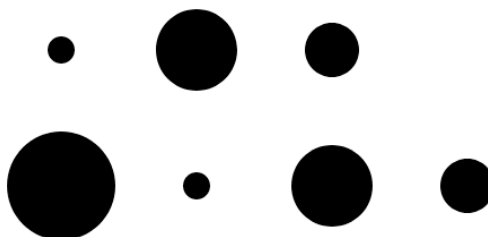
The next line `.join('circle')` specifies the type of element we're joining. The `cx` attribute is then updated using the element index `i` (so that they're evenly spaced). The `r` attribute is updated using the joined value.

This produces the output:

```

<g class="chart" transform="translate(50, 50)">
  <g transform="translate(0,0)">
    <circle cx="0" r="10"></circle>
    <circle cx="100" r="30"></circle>
    <circle cx="200" r="20"></circle>
  </g>
  <g transform="translate(0,100)">
    <circle cx="0" r="40"></circle>
    <circle cx="100" r="10"></circle>
    <circle cx="200" r="30"></circle>
    <circle cx="300" r="20"></circle>
  </g>
</g>

```



Array of arrays joined to `<g>` and `<circle>` elements

Navigate to <https://codepen.io/createwithdata/pen/dyNgzqJ><sup>53</sup> to view this example in CodePen.

To summarise, there are two data joins: the outer one joins `myData` to `<g>` elements and the inner one joins the inner arrays (e.g. `[10, 30, 20]`) to `<circle>` elements. This can be quite tricky to understand, so don't worry too much if you don't get it at first!

## 18.2.5 Update function with nested join

The nested join can be also put inside an update function (see the More on D3 selections chapter). For example:

```
function getData() {
  // Return a randomised array of arrays
}

function updateGroup(d, i) {
  var g = d3.select(this);
  g.attr('transform', 'translate(0,' + i * 100 + ')');

  g.selectAll('circle')
    .data(d)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 100;
    })
    .attr('r', function(d) {
      return d;
    });
}

function update() {
  d3.select('g.chart')
    .selectAll('g')
    .data(data)
    .join('g')
    .each(updateGroup);
}

window.setInterval(function() {
  data = getData();
  update();
}, 1000);
```

A function named `getData` is added which creates a randomised array of arrays. The outer join is moved to a function named `update` and we use `window.setInterval` to call `getData` and `update` at regular intervals:

Navigate to <https://codepen.io/createwithdata/pen/gOLmMrg><sup>54</sup> to view the example in CodePen.

Every second, `getData` randomises the nested array. The `update` function is then called and this performs the nested join, resulting in the `<circle>` elements being added, removed and resized.

## Notes



51 <https://codepen.io/createwithdata/pen/RwWmweL>

52 <https://codepen.io/createwithdata/pen/abdXJYv>

53 <https://codepen.io/createwithdata/pen/dyNgzqJ>

54 <https://codepen.io/createwithdata/pen/gOLmMrg>

# 19. Practical: Add Labels

Currently in Energy Explorer we join the `layoutData` array to `<circle>` elements (see `js/update.js`). We would now like to add a label to the circle indicating the country name. In this practical we'll do this by joining `layoutData` to `<g>` elements instead of `<circle>` elements. Each `<g>` element will be populated with a `<circle>` and `<text>` element. This technique was covered in the More on D3 Joins chapter.

By the end of this practical, each country will be represented by a `<g>` element containing a `<circle>` and `<text>` element:

```
<g class="country" transform="translate(471,264)">
  <circle r="24.26"></circle>
  <text class="label" y="50">Denmark</text>
</g>
```

## 19.1 Overview

Open `d3-start-to-finish-code/step7`. The file structure is:

```
step7
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    ├─ main.js
    └─ update.js
```

In this practical we:

1. Add label information to the layout function (`js/layout.js`).
2. Modify the update function (`js/update.js`) so that it joins `layoutData` to `<g>` elements instead of `<circle>` elements.
3. Add a CSS file containing a rule to center the labels.

## 19.2 Add label information to layout.js

Make the following changes to `js/layout.js`:

`js/layout.js`

---

```
function getTruncatedLabel(text) {  
  return text.length <= 10 ? text : text.slice(0, 10) + '...';  
}  
  
function layout(data) {  
  let labelHeight = 20;  
  let cellWidth = config.width / config.numColumns;  
  let cellHeight = cellWidth;  
  let cellHeight = cellWidth + labelHeight;  
  
  let maxRadius = 0.35 * cellWidth;  
  
  let radiusScale = d3.scaleSqrt()  
    .domain([0, 100])  
    .range([0, maxRadius]);  
  
  let layoutData = data.map(function(d, i) {  
    let item = {};  
  
    let column = i % config.numColumns;  
    let row = Math.floor(i / config.numColumns);  
  
    item.x = column * cellWidth + 0.5 * cellWidth;  
    item.y = row * cellHeight + 0.5 * cellHeight;  
    item.radius = radiusScale(d.renewable);  
  
    item.labelText = getTruncatedLabel(d.name);  
    item.labelOffset = maxRadius + labelHeight;  
  
    return item;  
  });  
  
  return layoutData;  
}
```

---

Two new properties `labelText` and `labelOffset` are added to each item. `labelText` represents the label text. Some of labels will overlap due to their length so we add a function `getTruncatedLabel` which truncates a label to a maximum of 10 characters. `labelOffset` is the vertical position of each label (with respect to the circle center) and is

computed from `maxRadius` and `labelHeight`. We also increase `cellHeight` to accomodate the label.

Now `layoutData` has label information and looks like:

```
[
  {
    "x": 42.857142857142854,
    "y": 52.857142857142854,
    "radius": 0,
    "labelText": "Angola",
    "labelOffset": 50
  },
  {
    "x": 128.57142857142856,
    "y": 52.857142857142854,
    "radius": 0,
    "labelText": "Albania",
    "labelOffset": 50
  },
  {
    "x": 214.28571428571428,
    "y": 52.857142857142854,
    "radius": 1.3416407864998736,
    "labelText": "United Ara...",
    "labelOffset": 50
  },
  ...
]
```

## 19.3 Modify update function to join data to <g> elements

The following changes are made to `js/update.js`:

js/update.js

---

```

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) {
    g.append('circle');

    g.append('text')
      .classed('label', true);
  }

  g.classed('country', true)
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')');

  g.select('circle')
    .attr('r', d.radius);

  g.select('.label')
    .attr('y', d.labelOffset)
    .text(d.labelText);
}

function update() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('circle')
    .selectAll('g')
    .data(layoutData)
    .join('circle')
    .join('g')
    .attr('cx', function(d) {
      return d.x;
    })
    .attr('cy', function(d) {
      return d.y;
    })
    .attr('r', function(d) {
      return d.radius;
    })
    .each(updateGroup);
}

```

---

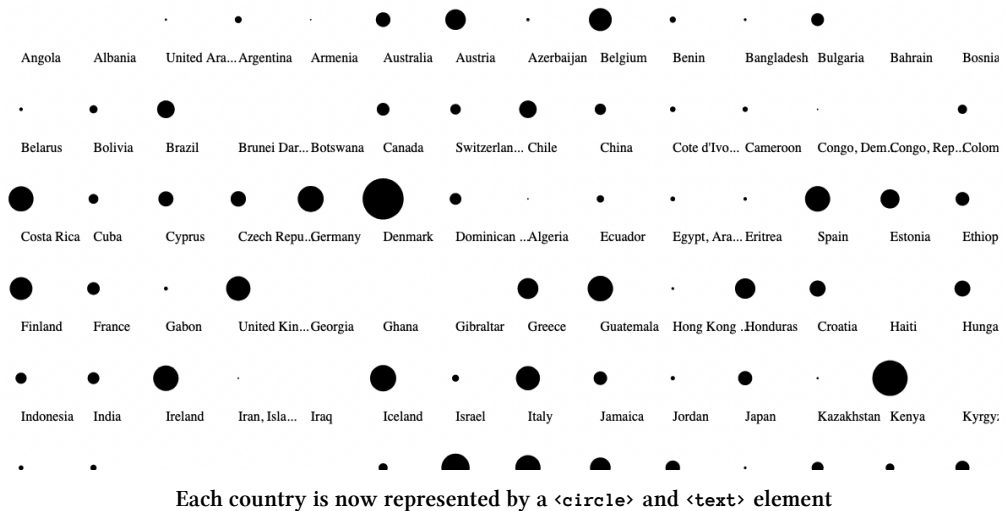
The changes follow the same pattern as in the ‘Joining an array to groups of elements’ section of the More on D3 Joins chapter.

The join code in function `update` now joins `<g>` elements instead of `<circle>` elements. A new function `updateGroup` is called on each joined element.

`updateGroup` appends a `<circle>` and `<text>` element to the current `<g>` element if it's empty. The `<g>` element is given a class attribute of value `country` (which helps us later on when we style each country).

Instead of setting the center of each circle we now apply a `translate` transform to each `<g>` element. Next we set the circle radius. Finally we set the `y` attribute of the label and its text content using the two properties we added in the previous section.

Now save `js/layout.js` and `js/update.js` and load Step 7 in your browser. You should see:



## 19.4 Center the labels using CSS

Finally we horizontally center the labels so that they're aligned with the circles.

Add a new directory to `step7` named `css` and add a new file `style.css`. Your directory structure should now look like:

step7

```
|— css
|   └─ style.css
|— data
|   └─ data.csv
|— index.html
└─ js
    ├── config.js
    ├── layout.js
    ├── lib
    │   └─ d3.min.js
    ├── main.js
    └─ update.js
```

Include the CSS file in `index.html` using a `<link>` element:

**index.html**

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    ...
  </body>
</html>
```

---

Now add a rule to `style.css` to center the text labels:

**css/style.css**

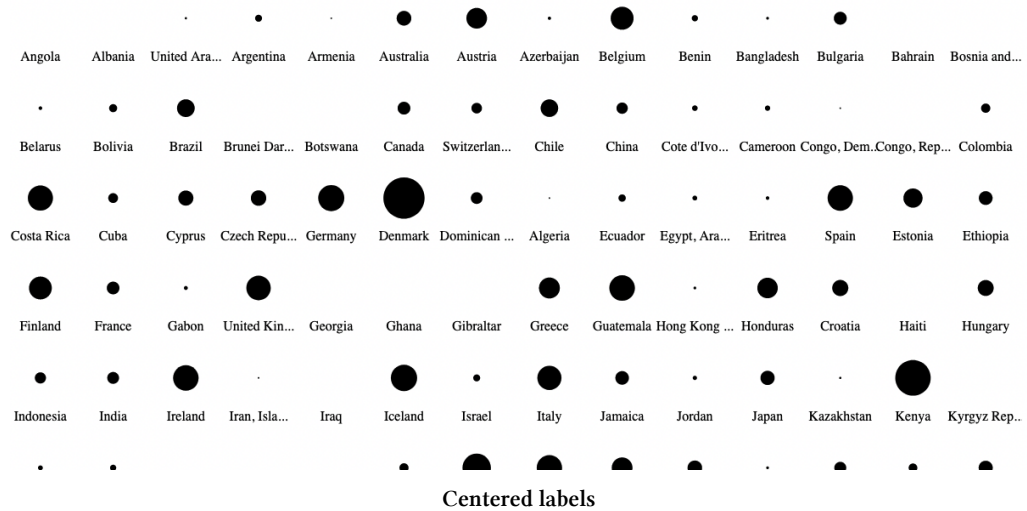
---

```
.country .label {
  text-anchor: middle;
}
```

---

The CSS classes `.country` and `.label` were added to each `<g>` and `<text>` element in `js/update.js`. The `text-anchor` property applies to SVG `<text>` elements and sets the horizontal alignment of the text.

Save `index.html` and `css/style.css` and refresh your browser. The labels should now be centered:



The completed code for this practical can be found in `step7-complete`.

## 19.5 Summary

If you've got to this point you've done well! You've completed a major part of the Energy Explorer build. Converting the data join to a nested join was quite tricky but it creates a good foundation for the remaining practicals. In the next section you'll add the remaining circles so that each country has four circles.



## 20. Practical: Add More Circles

In this practical we add **three more circles** so that each country is represented by four circles (one for each energy type).



Each country represented by four circles

Currently each country group looks like:

```
<g class="country" transform="translate(471,264)">
  <circle r="24"></circle>
  <text class="label" y="50">Denmark</text>
</g>
```

At the end of this practical each country group will look like:

```
<g class="country" transform="translate(471,264)">
  <circle class="renewable" r="24"></circle>
  <circle class="oilgascoal" r="17"></circle>
  <circle class="hydroelectric" r="1"></circle>
  <circle class="nuclear" r="0"></circle>
  <text class="label" y="50">Denmark</text>
</g>
```

There are three additional circles. Each circle also has a class attribute indicating which energy type it represents.

### 20.1 Overview

Open `d3-start-to-finish-code/step8`. The file structure is:

```

step8
├─ css
│   └─ style.css
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    ├─ main.js
    └─ update.js

```

In this practical we:

1. Add properties for each energy type in the layout function (`js/layout.js`).
2. Modify the update function (`js/update.js`) to append and update four circles.
3. Style the circles in `css/style.css`.

## 20.2 Add properties for each energy type in the layout function

In `js/layout.js` add new properties for each of the energy type values:

`js/layout.js`

---

```

function getTruncatedLabel(text) {
  return text.length <= 10 ? text : text.slice(0, 10) + '...';
}

```

```

function layout(data) {
  let labelHeight = 20;
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth + labelHeight;

  let maxRadius = 0.35 * cellWidth;

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let layoutData = data.map(function(d, i) {
    let item = {};

```

```

    let column = i % config.numColumns;
    let row = Math.floor(i / config.numColumns);

    item.x = column * cellWidth + 0.5 * cellWidth;
    item.y = row * cellHeight + 0.5 * cellHeight;
    item.radius = radiusScale(d.renewable);
    item.renewableRadius = radiusScale(d.renewable);
    item.oilGasCoalRadius = radiusScale(d.oilgascoal);
    item.hydroelectricRadius = radiusScale(d.hydroelectric);
    item.nuclearRadius = radiusScale(d.nuclear);

    item.labelText = getTruncatedLabel(d.name);
    item.labelOffset = maxRadius + labelHeight;

    return item;
  });

  return layoutData;
}

```

---

As a reminder, each object in data looks similar to:

```

{
  "name": "Angola",
  "id": "AGO",
  "hydroelectric": 53.2,
  "nuclear": null,
  "oilgascoal": 46.8,
  "renewable": 0
}

```

In the above changes the layout function takes each of the four energy indicators (hydroelectric, nuclear, oilgascoal and renewable), applies the radius scale and assigns the result to new properties `renewableRadius`, `oilGasCoalRadius`, `hydroelectricRadius` and `nuclearRadius`.

## 20.3 Add four circles in the update function

Currently the update function (`js/update.js`) adds a single circle. Make the following changes which add circles for each of the four energy types:

## js/update.js

---

```

function initialiseGroup(g) {
    g.classed('country', true);

    g.append('circle')
      .classed('renewable', true);

    g.append('circle')
      .classed('oilgascoal', true);

    g.append('circle')
      .classed('hydroelectric', true);

    g.append('circle')
      .classed('nuclear', true);

    g.append('text')
      .classed('label', true);
}

function updateGroup(d, i) {
    let g = d3.select(this);

    if(g.selectAll('*').empty()) {
        g.append('circle');

        g.append('text')
        .classed('label', true);
    }
    if(g.selectAll('*').empty()) initialiseGroup(g);

    g.classed('country', true)
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')');
    g.attr('transform', 'translate(' + d.x + ', ' + d.y + ')');

    g.select('circle')
    .attr('r', d.radius);
    g.select('.renewable')
      .attr('r', d.renewableRadius);

    g.select('.oilgascoal')
      .attr('r', d.oilGasCoalRadius);

    g.select('.hydroelectric')
      .attr('r', d.hydroelectricRadius);

```

```
g.select('.nuclear')
  .attr('r', d.nuclearRadius);

g.select('.label')
  .attr('y', d.labelOffset)
  .text(d.labelText);
}

function update() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('g')
    .data(layoutData)
    .join('g')
    .each(updateGroup);
}
```

---

We add a new function `initialiseGroup` to take care of initialising each `<g>` element. This function sets the class attribute of each `<g>` element to country, adds a circle for each energy type and adds the `<text>` element for the label. Each of the circles also gets a class attribute that describes the energy type it refers to.

`updateGroup` now selects each of the four circles and updates its radius using the new properties we added in the previous section.

## 20.4 Style the circles

By default SVG circles have a fill colour of black meaning we can't distinguish the circles, so let's set the fill to none and the stroke to #aaa:

css/style.css

---

```
.country .label {
  text-anchor: middle;
}
```

```
circle {
  fill: none;
  stroke: #aaa;
}
```

---

## 20.5 Save and refresh

Save `css/style.css`, `js/layout.js` and `js/update.js` and load `step8` in your browser.

You should see:



Energy Explorer with a circle for each energy type

The completed code is in `step8-complete` of the code download.

## 20.6 Summary

Adding the nested join in the previous section was fairly complicated it made adding further elements to the `g` element fairly straightforward. Each country now has 4 circles (each of which represents one of the four energy types) and a text label. It's beginning to take shape!

It's not possible to see which circle relates to which energy type so in the next section you'll style each circle in accordance with its energy type.

# 21. Practical: Style the Circles

This practical is a fun one. We **style the circles** so that they represent each of the 4 energy types. The chart will look like:



Energy Explorer with each circle styled according to the energy type

Renewable energy is represented by a solid green circle, Oil, Gas & Coal by a grey outline, Hydroelectric as a dotted blue outline and Nuclear as a dotted orange outline.

## 21.1 Overview

Open `d3-start-to-finish-code/step9`. The file structure is:

```
step9
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   └── d3.min.js
    ├── main.js
    └── update.js
```

In this practical we:

1. Design a suitable colour scheme for the circles.
2. Style the circles in `css/style.css`.
3. Center the visualisation.
4. Change the background colour.

## 21.2 Design a colour scheme

Each country is represent by four overlapping circles so we need a colour scheme that makes it easy to identify a circle's energy type. Also, when designing data visualisations it's recommended that a colour-blind friendly scheme is chosen. There are a few visual variables we can use such as:

- fill colour
- fill pattern
- stroke colour
- stroke width
- stroke style (dotted, dashed etc.)

**Fill** refers to the colour of the inside of the circle and **stroke** refers to the circle outline.

Let's start with the renewable energy type. An obvious choice for this circle is **green**:



Solid green circle for renewable energy

Similarly, an obvious choice for Oil, Gas and Coal is **black** or **grey**. Black is probably too strong a colour, so let's use grey. If we use a filled grey circle we have to make sure smaller circles are drawn in front of larger circles. An easier approach is to use an empty circle with a grey stroke:



Grey outlined circle for oil, gas and coal



Hydroelectric refers to energy produced from water pressure (think of water behind a dam). **Blue**'s an obvious choice for anything related to water so let's choose blue for hydroelectric power:



Blue outlined circle for hydroelectric

Finally we have nuclear. The nuclear power symbol has a yellow background so yellow might be a good choice. A yellow outline won't show up very well against a light background so perhaps **orange** works well:



Orange outlined circle for nuclear

We're using **three colours** (green, blue and orange) which might not be colour blind safe so let's also change the stroke style of the circles.

Let's use a **short dashes** for the hydroelectric circles and **long dashes** for the nuclear circles:



Use dashed outlines to further distinguish the circles

It should now be possible to identify the energy type of each circle even if colour blind.

I don't have a design background (I have an engineering background) so choosing colours isn't my strongest skill. I often visit [coolors.co](https://coolors.co)<sup>55</sup> which lets you cycle through different colour schemes. I like the look of the colours that coolers generates and I usually keep cycling through until I spot a colour I like the look of. This is how I found the particular hues of green, blue and orange seen above.



If you'd like to learn more about designing data visualisation colour schemes a good place to start is Lisa Charlotte Muth's [A detailed guide to colors in data vis style guides](#)<sup>56</sup>.

Here's a table with the colours and styles we'll use:

| Energy type   | fill    | stroke  | stroke-dasharray |
|---------------|---------|---------|------------------|
| renewable     | #7FB069 | none    | none             |
| oilgascoal    | none    | #BCB5B2 | none             |
| hydroelectric | none    | #5EB1BF | 3,1              |
| nuclear       | none    | #F6AE2D | 4,2              |

stroke-dasharray is a CSS property that defines a stroke's dash style. The first number defines how long the dash is and the second number defines the size of the gap between the dashes.

## 21.3 Style the circles

In `css/style.css` add CSS rules for each of the four energy types:

`css/style.css`

```
eircle {  
  fill: none;  
  stroke: #aaa;  
}
```

```
circle.renewable {  
  fill: #7FB069;  
}
```

```
circle.oilgascoal {  
  fill: none;  
  stroke: #BCB5B2;  
}
```

```
circle.hydroelectric {  
  fill: none;  
  stroke: #5EB1BF;  
  stroke-dasharray: 3,1;  
}
```

```

}

circle.nuclear {
  fill: none;
  stroke: #F6AE2D;
  stroke-dasharray: 4,2;
}

.country .label {
  text-anchor: middle;
}

```

---

(Make sure the old rule for all `circle` elements is removed.)

In the previous practical we assigned a class attribute on each circle indicating which energy type it's representing. This allows us to select each circle type in `style.css` and style it accordingly.

## 21.4 Center the visualisation

In `index.html` wrap the `svg` element in a `div` element with class attribute `wrapper`:

**index.html**

---

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <div id="wrapper">
      <svg width="1200" height="1200">
        <g id="chart"></g>
      </svg>
    </div>

    <script src="js/lib/d3.min.js"></script>

    <script src="js/config.js"></script>
    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/main.js"></script>

```

```
</body>
</html>
```

---

The main reason for adding the `div` element is that further interface elements (such as a menu and a legend) will be added later on and it's good practice to enclose them in a single wrapper element.

In `css/style.css` add a rule for the new `div` element:

`style.css`

---

```
#wrapper {
    width: 1200px;
    margin: 0 auto;
}
```

```
circle.renewable {
    fill: #7FB069;
}
```

...

---

To center a `div` element you can set its width and set its margin to `0 auto`. This isn't at all obvious if you don't have much CSS experience but it's common practice.

## 21.5 Change the background colour

Finally in `style.css` set the background colour to off-white:

`style.css`

---

```
body {
    background-color: #FFFFFF;
}
```

```
#wrapper {
    width: 1200px;
    margin: 0 auto;
}
```

```
circle.renewable {
    fill: #7FB069;
}
```

...

---

## 21.6 Save and refresh

Save `index.html` and `style.css` and load `step9` in your browser. You should see that the circles are styled according to energy type, the visualisation is centered and the background is off-white:



Energy Explorer with styled circles and off white background

## 21.7 Summary

A major part of the visualisation is finished now. Each country is represented by four circles and you can now begin to compare the energy mixes of different countries.

I'm personally drawn to the solid green circles. The larger they are, the more renewables contribute to that country's energy mix. Denmark's a good example. Albania's an interesting case as it looks like the vast majority of its energy comes from hydroelectric (blue outline). And not surprisingly for many countries, the biggest contributor is oil, gas and coal (grey outline).

In the next few chapters we look at adding an information popup that appears when a country is hovered over.

## Notes

55 <https://coolors.co/>

56 <https://blog.datawrapper.de/colors-for-data-vis-style-guides/>

## 22. D3 Event Handling

In this chapter you'll learn how to add **event handlers** to HTML and SVG elements. An event handler is a function that gets called when the **user interacts** with an HTML or SVG element. The most common interactions are:

- **click** (the user clicks an element)
- **mouseover** (the user moves the pointer into an element)
- **mouseout** (the user moves the pointer out of an element)
- **mousemove** (the user moves the pointer within an element)

You can add an event handler to the HTML/SVG elements of a D3 selection using the `.on` method:

```
function handleClick() {  
  alert('A circle was clicked');  
}  
  
d3.selectAll('circle')  
  .on('click', handleClick);
```

In the above code D3 selects all `circle` elements on the page. It then attaches an event listener to each circle so that `handleClick` is called when a circle is clicked.

Navigate to <https://codepen.io/createwithdata/pen/PoZvrmy><sup>57</sup> to view this example in CodePen. Click on a circle to see an alert appear.

Things get more interesting if the selection has been joined to an array of data. For example, suppose your HTML is:

```
<svg>  
  <g class="circles" transform="translate(50, 50)">  
    </g>  
</svg>
```

and you join an array to circle elements:

```
let myData = [10, 40, 30];

d3.select('g.circles')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  })
  .attr('cx', function(d, i) {
    let circleSpacing = 100;
    return i * circleSpacing;
  });
```

You can add an event handler to the selection using `.on`:

```
let myData = [10, 40, 30];
```

```
function handleClick(e, d, i) {
  alert("A circle was clicked and it's value is " + d);
}
```

```
d3.select('g.circles')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  })
  .attr('cx', function(d, i) {
    let circleSpacing = 100;
    return i * circleSpacing;
  })
  .on('click', handleClick);
```

When a circle is clicked, the event handler `handleClick` is called and the **joined value is passed in as the second parameter** (usually named `d`). The index of the element is passed in as the third parameter (usually named `i`).



Event handlers in D3 versions up to including 5 look like `function(d, i) { ... }` where the **first** parameter is the joined value and the second parameter is the index.

In version 6 of D3 the browser `MouseEvent` object is passed in as the first parameter to an event handler function. (This is a breaking change.)

Navigate to <https://codepen.io/createwithdata/pen/jOWogEr><sup>58</sup> to view this example in CodePen

This approach lets you create **data-driven interactions**. In other words you can create interactivity that uses the joined data. Here's another example where a simple information panel is updated when an element is clicked. Here's the HTML:

```
<div id="info"></div>
<div>
  <svg width="400" height="100">
    <g class="circles" transform="translate(50, 50)">
      </g>
    </svg>
  </div>
```

Notice there's a div element (with id info) that acts as an information panel. The JavaScript is:

```
let myData = [10, 40, 30];
```

```
function handleClick(e, d) {
  d3.select('#info')
    .text("Value is " + d);
}
```

```
d3.select('g.circles')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  })
  .attr('cx', function(d, i) {
    let circleSpacing = 100;
    return i * circleSpacing;
  })
  .on('click', handleClick);
```

When a circle is clicked, handleClick is called. handleClick selects the information panel and updates its text content with the joined value of the clicked circle.

Navigate to <https://codepen.io/createwithdata/pen/zYBoxpy><sup>59</sup> to view this example in CodePen.



## 22.1 Event types

The most common event types you'll use when building data visualisations are `click`, `mouseover` and `mouseout`.

This table shows when each event type occurs:

| Event type             | When it occurs                                       |
|------------------------|--|
| <code>click</code>     | When an element is clicked                           |
| <code>mouseover</code> | When the mouse pointer first moves over an element   |
| <code>mouseout</code>  | When the mouse pointer first moves out of an element |

You can call the `.on` method more than once if you want to react to more than one event type. For example, let's modify the previous example so that all three event types are handled:

```
let myData = [10, 40, 30];
```

```
function handleMouseover(e, d) {  
  d3.select('#info')  
    .text("Value is " + d);  
}
```

```
function handleMouseout(e, d) {  
  d3.select('#info')  
    .text(null);  
}
```

```
function handleClick(e, d) {  
  d3.select('#info')  
    .text("Clicked element with value " + d);  
}
```

```
d3.select('g.circles')  
  .selectAll('circle')  
  .data(myData)  
  .join('circle')  
  .attr('r', function(d) {  
    return d;  
  })  
  .attr('cx', function(d, i) {  
    let circleSpacing = 100;
```

```
    return i * circleSpacing;
  })
  .on('mouseover', handleMouseover)
  .on('mouseout', handleMouseout)
  .on('click', handleClick);
```

When the mouse pointer first moves over a circle, `handleMouseover` is called and the information panel updates with the circle's joined value. When the mouse pointer moves off a circle `handleMouseout` is called which clears the information panel. When a circle is clicked, `handleClick` is called and the information panel updates with a message saying an element has been clicked.

Navigate to <https://codepen.io/createwithdata/pen/oNbRKEV><sup>60</sup> to view this example in CodePen.

## 22.2 index and this

The element index (usually named `i`) is passed in as the third parameter to the event handler. The HTML/SVG element which triggered the event is assigned to the `this` keyword within the event handler function. They can be used in this manner:

```
function handleClick(e, d, i) {
  console.log('the HTML/SVG element', this, 'was clicked');
  console.log('the joined value is', d);
  console.log('the element index is', i);
}
```

## Notes

<sup>57</sup> <https://codepen.io/createwithdata/pen/PoZvrmy>

<sup>58</sup> <https://codepen.io/createwithdata/pen/jOWogEr>

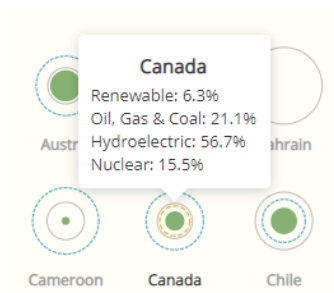
<sup>59</sup> <https://codepen.io/createwithdata/pen/zYBoxpy>

<sup>60</sup> <https://codepen.io/createwithdata/pen/oNbRKEV>

## 23. Flourish Popup Library

One of the most useful features you can add to a data visualisation is an **information popup**. This is a box that displays information about a selected item.

For example the final version of Energy Explorer has a popup that shows the energy mix as numbers:



Information popup in Energy Explorer

It's possible to implement your own popup using D3 but there are libraries that do this for you.

A good popup library is Flourish's [popup component](#)<sup>61</sup>. This is a popup that's used in many of Flourish's templates.

### 23.1 Installing the popup

The Flourish popup can be installed by downloading it from <https://cdn.flourish.rocks/popup-v1.1.js><sup>62</sup>, moving it to your JavaScript directory in your project and including it using `<script>` tags. You'll follow this approach in the next practical.

See the [popup's documentation](#)<sup>63</sup> for other (more modern) ways of installing it.

### 23.2 Initialising the popup

Once the popup component has been installed it can be initialised using:

```
let popup = Popup();
```

## 23.3 Popup methods

The Flourish popup's most interesting methods are: `.point`, `.html`, `.draw` and `.hide`.

The `.point` method sets the popup's location. You can pass in either:

- x and y co-ordinates or
- an HTML/SVG element

I personally find it easier to **pass in an HTML/SVG element** because this saves you having to figure out x and y (which can sometimes get tricky).

The `.html` method sets the content of the popup. You pass in a string containing HTML code. The `.draw` method draws (or redraws) the popup. The `.hide` method hides the popup.

## 23.4 Popup styling

In general you need to set `pointer-events` to `none` on the popup, otherwise the popup might flicker when the mouse pointer is close to the popup. The popup has a class attribute named `flourish-popup` so you can apply the rule using:

```
.flourish-popup {  
  pointer-events: none;  
}
```

You can style the content of the popup using the `flourish-popup-content` class attribute:

```
.flourish-popup-content {  
  font-family: sans-serif;  
  color: #555;  
}
```

## 23.5 Example

Let's add the popup to the circles example from the D3 Event Handling chapter. The HTML looks like:

```
<svg width="400" height="200">
  <g class="circles" transform="translate(50, 100)">
    </g>
  </svg>
```

and the JavaScript is changed to:

```
let myData = [10, 40, 30];
```

```
let popup = Popup();
```

```
function handleMouseover(e, d) {
  popup.point(this)
    .html('The value is ' + d)
    .draw();
}
```

```
function handleMouseout() {
  popup.hide();
}
```

```
d3.select('.circles')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('r', function(d) {
    return d;
  })
  .attr('cx', function(d, i) {
    return i * 100;
  })
  .on("mouseover", handleMouseover)
  .on("mouseout", handleMouseout);
```

The code is mostly the same as the code in the D3 Event Handling chapter. The main differences are:

- a Flourish popup is initialised and assigned to the variable `popup`
- `handleMouseover` sets the popup position by passing `this` into `.point()`. (this is the HTML/SVG element that triggered the event.)
- the content is set using `.html` and `.draw` is called to make the popup visible
- `handleMouseout` hides the popup using `.hide`

Navigate to <https://codepen.io/createwithdata/pen/MWKgJLX><sup>64</sup> to view this example in CodePen.

A popup appears when a circle is hovered:



**Flourish popup activated when a circle is hovered**

One of the benefits of using the Flourish popup library is it handles cases where the popup might be cropped. For example if the first circle is hovered the popup automatically aligns to the right:



**Flourish popup when circle near edge of container is hovered**

In the next practical you'll add a popup to Energy Explorer.

## Notes

<sup>61</sup> <https://github.com/kiln/flourish-popup>

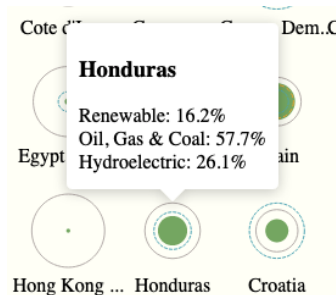
<sup>62</sup> <https://cdn.flourish.rocks/popup-v1.1.js>

<sup>63</sup> <https://github.com/kiln/flourish-popup>

<sup>64</sup> <https://codepen.io/createwithdata/pen/MWKgJLX>

## 24. Practical: Add a Popup

In this practical we add a **popup** to Energy Explorer using the Flourish popup component:



Energy Explorer with Flourish popup

When a country is hovered a popup will appear containing the country name together with the country's energy mix.

### 24.1 Overview

Open `d3-start-to-finish-code/step10`. The directory structure is:

```
step10
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
├── js
│   ├── config.js
│   ├── layout.js
│   ├── lib
│   │   ├── d3.min.js
│   │   └── popup-v1.1.1.min.js
│   ├── main.js
│   └── update.js
```

The Flourish popup library has been added so you don't need to download it yourself. (It was downloaded from <https://cdn.flourish.rocks/popup-v1.full.min.js>.)

In this practical we:

1. Link to the popup library in `index.html`.
2. Add a new module `js/popup.js` to manage the popup.
3. Add event handlers to the country groups that show/hide the popup.
4. Populate the popup with the energy data.
5. Offset the popup (so that it's above the circle center).

## 24.2 Link to the popup library

In `index.html` add script tags to load the popup library:

**index.html**

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <div id="wrapper">
      <svg width="1200" height="1200">
        <g id="chart"></g>
      </svg>
    </div>

    <script src="js/lib/d3.min.js"></script>
    <script src="js/lib/popup-v1.1.1.min.js"></script>

    <script src="js/config.js"></script>
      <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

Save `index.html`.

## 24.3 Add new module for popup

Create a new file `popup.js` in the `js` directory that'll contain the popup related code:



```
step10
├─ css
│   └─ style.css
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    │   └─ popup-v1.1.1.min.js
    ├─ main.js
    └─ popup.js
    └─ update.js
```

Link to this new file in index.html:

#### index.html

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <div id="wrapper">
      <svg width="1200" height="1200">
        <g id="chart"></g>
      </svg>
    </div>

    <script src="js/lib/d3.min.js"></script>
    <script src="js/lib/popup-v1.1.1.min.js"></script>

    <script src="js/config.js"></script>
    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/popup.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

Save index.html.

## 24.4 Add event handlers

In `js/popup.js` create a popup object and add mouseover and mouseout event handlers:

`js/popup.js`

---

```
let popup = Popup();

function handleMouseover(e, d) {
  popup
    .point(this)
    .html(d.labelText)
    .draw();
}

function handleMouseout() {
  popup.hide();
}
```

---

This code is similar to what we covered in the ‘Flourish Popup Library’ chapter.

In `handleMouseover`, this represents the `<g>` element that triggered the mouse event. We pass the `labelText` property into the popup, just so we can get something up and running quickly.

Now register the event handlers in the update function using D3’s `.on` method:

`js/update.js`

---

```
function initialiseGroup(g) {
  g.classed('country', true)
    .on('mouseover', handleMouseover)
    .on('mouseout', handleMouseout);

  g.append('circle')
    .classed('popup-center', true)
    .attr('r', 1);

  g.append('circle')
    .classed('renewable', true);

  g.append('circle')
    .classed('oilgascoal', true);

  g.append('circle')
    .classed('hydroelectric', true);
}
```

```

    g.append('circle')
      .classed('nuclear', true);

    g.append('text')
      .classed('label', true);
  }

  ...

```

---

There's also a couple of CSS rules we need to add. By default, circles with a fill colour of none don't trigger events. We can change this by setting `pointer-events` to `all` on each country group. We also set `pointer-events` to `none` on the popup itself so that mouseover events aren't triggered if the mouse pointer moves over the popup itself. (These aren't very obvious things to add, and knowing to do these comes with experience.)

#### css/style.css

---

```

body {
  background-color: #FFFFFF7;
}

...

circle.nuclear {
  fill: none;
  stroke: #F6AE2D;
  stroke-dasharray: 4,2;
}

.country {
  pointer-events: all;
}

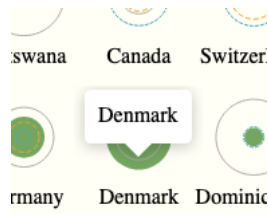
.country .label {
  text-anchor: middle;
}

.flourish-popup {
  pointer-events: none;
}

```

---

Save `index.html`, `js/popup.js`, `js/update.js` and `css/style.css` and load `step10` in your browser. Now when you hover over a country you should see a popup displaying the country name.



Popup showing country name

## 24.5 Populate popup with energy data

We now add the four energy indicators to the popup. Start by adding a new property `popupData` to each item in `js/layout.js`:

`js/layout.js`

---

```
function getTruncatedLabel(text) {
  return text.length <= 10 ? text : text.slice(0, 10) + '...';
}

function layout(data) {
  let labelHeight = 20;
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth + labelHeight;

  let maxRadius = 0.35 * cellWidth;

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let layoutData = data.map(function(d, i) {
    let item = {};

    let column = i % config.numColumns;
    let row = Math.floor(i / config.numColumns);

    item.x = column * cellWidth + 0.5 * cellWidth;
    item.y = row * cellHeight + 0.5 * cellHeight;

    item.renewableRadius = radiusScale(d.renewable);
    item.oilGasCoalRadius = radiusScale(d.oilgascoal);
    item.hydroelectricRadius = radiusScale(d.hydroelectric);
    item.nuclearRadius = radiusScale(d.nuclear);
  });
}
```

```

        item.labelText = getTruncatedLabel(d.name);
        item.labelOffset = maxRadius + labelHeight;

        item.popupData = {
            name: d.name,
            renewable: d.renewable,
            oilgascoal: d.oilgascoal,
            hydroelectric: d.hydroelectric,
            nuclear: d.nuclear
        };

        return item;
    });

    return layoutData;
}

```

---

The new property `.popupData` is an object that contains the information that'll be displayed in the popup. Now modify `js/popup.js` so that each energy indicator is displayed:

#### js/popup.js

---

```

let popup = Popup();

function getPopupEntry(d, type, label) {
    if (!isNaN(d.popupData[type])) {
        return '<div>' + label + ': ' + d.popupData[type] + '%</div>';
    }
}

return '';
}

function popupTemplate(d) {
    let html = '';
    html += '<h3>' + d.popupData.name + '</h3>';

    html += getPopupEntry(d, 'renewable', 'Renewable');
    html += getPopupEntry(d, 'oilgascoal', 'Oil, Gas & Coal');
    html += getPopupEntry(d, 'hydroelectric', 'Hydroelectric');
    html += getPopupEntry(d, 'nuclear', 'Nuclear');

    return html;
}

function handleMouseover(e, d) {

```

```

    popup
      .point(this)
      .html(popupTemplate(d))
      .draw();
  }

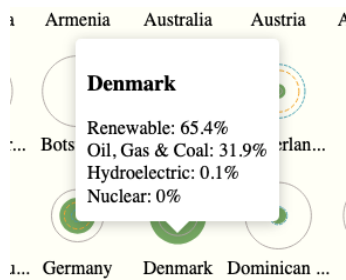
  function handleMouseout() {
    popup.hide();
  }

```

---

In `handleMouseover` we now pass `popupTemplate(d)` into the `.html` method. `popupTemplate` is a function that constructs a string containing HTML code that represents the popup content. We build up the string line by line starting with a heading containing the country name. (`.popupData` was added to the layout object items in the previous step.) A helper function `getPopupEntry` constructs a single line that displays an indicator name and associated value. If the value is `NaN` (which means it was missing from the CSV file) the indicator is omitted.

Save `js/layout.js` and `js/popup.js` and refresh your browser. You should now see the energy indicators when you hover over a country:



Popup with all four energy indicators

## 24.6 Offset the popup

Currently the popup is positioned in the center of the `<g>` element (which contains the circles and the label). Most of the country's circles are obscured so we'd like the popup to appear **towards the top** of the circles. We'll do this by adding a hidden element, placing it where we'd like the popup to appear. This element will get passed into the popup's `.position` method.

Start by adding a new property `popupOffset` to the layout items. This specifies where the popup pointer (the triangle underneath the popup) should appear in relation to the circle centers:

**js/layout.js**

---

```

function getTruncatedLabel(text) {
  return text.length <= 10 ? text : text.slice(0, 10) + '...';
}

function layout(data) {
  let labelHeight = 20;
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth + labelHeight;

  let maxRadius = 0.35 * cellWidth;

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let layoutData = data.map(function(d, i) {
    let item = {};

    ...

    item.labelText = getTruncatedLabel(d.name);
    item.labelOffset = maxRadius + labelHeight;

    item.popupOffset = -0.8 * maxRadius;
    item.popupData = {
      name: d.name,
      renewable: d.renewable,
      oilgascoal: d.oilgascoal,
      hydroelectric: d.hydroelectric,
      nuclear: d.nuclear
    };

    return item;
  });

  return layoutData;
}

```

---

We position the popup pointer  $0.8 * \text{maxRadius}$  above the circle centers. This value was arrived at by experimentation and strikes a balance between the popup being high enough to reveal the underlying circles but not so high it disconnects from smaller circles. Ideally we'd position the popup according to the largest circle, but we're keeping things simple and focused for the benefit of learning.

Now in `js/update.js` add a new circle to each group:

**js/update.js**


---

```

function initialiseGroup(g) {
    g.classed('country', true)
      .on('mouseover', handleMouseover)
      .on('mouseout', handleMouseout);

    g.append('circle')
      .classed('popup-center', true)
      .attr('r', 1);

    g.append('circle')
      .classed('renewable', true);

    g.append('circle')
      .classed('oilgascoal', true);

    g.append('circle')
      .classed('hydroelectric', true);

    g.append('circle')
      .classed('nuclear', true);

    g.append('text')
      .classed('label', true);
}

function updateGroup(d, i) {
    let g = d3.select(this);

    if(g.selectAll('*').empty()) initialiseGroup(g);

    g.attr('transform', 'translate(' + d.x + ', ' + d.y + ')');

    g.select('.popup-center')
      .attr('cy', d.popupOffset);

    g.select('.renewable')
      .attr('r', d.renewableRadius);

    ...
}

```

---

The new circle is added in `initialiseGroup` and given a class attribute of `popup-center`. It doesn't really matter what its radius is, so long as it isn't zero. In `updateGroup` the circle's `cy` attribute is updated using the new layout item property `.popupOffset`.



Now use the new circle to position the popup:

#### js/popup.js

---

```
let popup = Popup();

function getPopupEntry(d, type, label) {
  if (!isNaN(d.popupData[type])) {
    return '<div>' + label + ': ' + d.popupData[type] + '%</div>';
  }

  return '';
}

function popupTemplate(d) {
  let html = '';
  html += '<h3>' + d.popupData.name + '</h3>';

  html += getPopupEntry(d, 'renewable', 'Renewable');
  html += getPopupEntry(d, 'oilgascoal', 'Oil, Gas & Coal');
  html += getPopupEntry(d, 'hydroelectric', 'Hydroelectric');
  html += getPopupEntry(d, 'nuclear', 'Nuclear');

  return html;
}

function handleMouseover(e, d) {
  let popupCenter = d3.select(this)
    .select('.popup-center')
    .node();

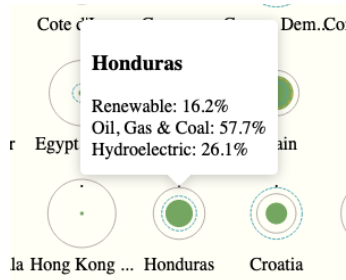
  popup
    .point(popupCenter)
    .html(popupTemplate(d))
    .draw();
}

function handleMouseout() {
  popup.hide();
}
```

---

In `handleMouseover` we select the new circle and call `.node()` to get the actual `<circle>` element (see the D3 Selections chapter for a reminder). We then pass the element into the popup's `.point` method.

Save `js/layout.js`, `js/update.js` and `js/popup.js` and refresh your browser. The popup should now appear a bit higher up:



Popup positioned with the 'popupCenter' circle (which is visible)

We also need to hide the new circles, so add the following to `css/style.css`:

`css/style.css`

---

```
body {
    background-color: #FFFFFF7;
}
```

...

```
.country {
    pointer-events: all;
}
```

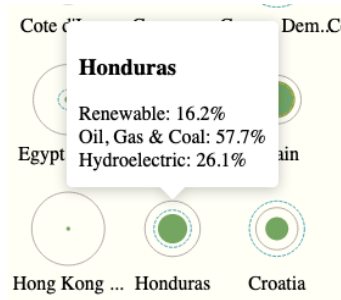
```
.country .label {
    text-anchor: middle;
}
```

```
.popup-center {
    opacity: 0;
}
```

```
.flourish-popup {
    pointer-events: none;
}
```

---

Now the Energy Explorer with popup looks like:



Energy Explorer with Flourish popup

## 24.7 Summary

In this practical we added an information popup to Energy Explorer. This allows users to get precise information for each country.

Although there was quite a lot of work to get the popup up and running, using an existing library has saved a lot of work. For instance, the popup library automatically positions the popup so that it doesn't disappear off the edges of the screen.

## 25. State Management

The state of an application describes what the application is currently doing. For example the state of a data visualisation might include things like:

- the **type of chart** (bar, line, pie etc.) that the user has chosen
- the **colour palette** the user has chosen
- **zoom level** and **pan position**
- **sort** and **filter** settings

Typically each time there's a user interaction (e.g. clicking an item, choosing from a menu) the state changes. And each time the state changes, the display updates accordingly.

There are several approaches to managing state in an application and we'll look at an approach that is straightforward, effective and based on a very common pattern in web development known as [Flux](#)<sup>65</sup>. The basic idea is that the entire application state is stored in a JavaScript object. For example:

```
let state = {  
  selectedIndicator: 'GDP',  
  filterType: 'region',  
  filterValue: 'East Asia',  
  colorScheme: 'category10'  
}
```

This goes hand in hand with a function named `action` that's responsible for updating the state object. It accepts two arguments:

- a string that describes the state change that should take place (such as `'setSelectedIndicator'` or `'setFilterType'` )
- any further information that is required for the state change. Typically this is a string or object

For example:

```
sets `state.selectedIndicator` to `CO2_emissions`.
```

Typically `action` is called when there's an event such as a button click. The `action` function consists of a `switch` statement containing `case` blocks for each action type:

```
```js
function action(type, param) {
  switch(type) {
    case 'setSelectedIndicator':
      state.selectedIndicator = param;
      break;
    case 'setFilterType':
      state.filterType = param;
      break;
  }

  update();
}
```

Each action has a block of code which updates the state accordingly. At the end of action the update function is called which triggers a redraw of the whole application.

Let's look at a full example.

## 25.1 State management example

Suppose you've an update function that joins an array of data to SVG circle elements:

```
let data = [30, 10, 20, 40];

function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('r', function(d) {
      return d;
    })
    .attr('cx', function(d, i) {
      return 100 + i * 100;
    })
    .attr('cy', 100);
}

update();
```

Let's add a feature whereby a circle is selected by clicking it. Only one circle may be selected at a time and we'll set the colour of the selected circle to red.

We start by adding a state object containing a property `selectedCircle` which indicates the circle that's been selected:

```
let data = [30, 10, 20, 40];
```

```
let state = {  
  selectedCircle: null  
};
```

```
function update() { ... }
```

`state.selectedCircle` is initialised to `null` to indicate that nothing has been selected. Now let's add a function named `action` which handles a single action type named `setSelectedCircle`:

```
let data = [30, 10, 20, 40];
```

```
let state = {  
  selectedCircle: null  
};
```

```
function action(type, param) {  
  switch(type) {  
    case 'setSelectedCircle':  
      state.selectedCircle = param;  
      break;  
  }  
}
```

```
  update();  
}
```

```
function update() { ... }
```

When `action` is called with action type `'setSelectedCircle'` it'll set `state.selectedCircle` to `param`. We'll see in the next step that `param` represents the **index** of the selected circle.

Now let's add a click event handler named `handleClick` and attach it to each circle. `handleClick` calls `action`, passing in `'setSelectedCircle'` as the action type and the circle index `i` as the action parameter:

```
let data = [30, 10, 20, 40];

let state = {
  selectedCircle: null
};

function action(type, param) {
  switch(type) {
    case 'setSelectedCircle':
      state.selectedCircle = param;
      break;
  }

  update();
}

function handleClick(d, i) {
  action('setSelectedCircle', i);
}

function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    ...
    .attr('cy', 100)
    .on('click', handleClick);
}
```

Finally let's set the fill of each circle according to whether its index is the same as `state.selectedCircle`:

```
let data = [30, 10, 20, 40];

let state = {
  selectedCircle: null
};

function action(type, param) {
  switch(type) {
    case 'setSelectedCircle':
      state.selectedCircle = param;
      break;
  }
}
```

```

    update();
  }

  function handleClick(d, i) {
    action('setSelectedCircle', i);
  }

  function update() {
    d3.select('#chart')
      .selectAll('circle')
      .data(data)
      .join('circle')
      ...
      .attr('cy', 100)
      .style('fill', function(d, i) {
        return state.selectedCircle === i ? 'red' : null;
      })
      .on('click', handleClick);
  }

```

Now when a circle is clicked:

- handleClick is called which calls action passing in setSelectedCircle as the action type and the circle index i as the second argument
- action updates state.selectedCircle with the circle index and calls update
- update updates the fill colour of each circle. If the circle index matches state.selectedCircle it'll get coloured red

Navigate to <https://codepen.io/createwithdata/pen/YzwGaBw><sup>66</sup> to view this example in CodePen.

As a further example, we can modify action so that if the clicked circle is already selected it'll get deselected (by setting state.selectedCircle to null):

```

...

function action(type, param) {
  switch(type) {
    case 'setSelectedCircle':
      if(state.selectedCircle === param) {
        state.selectedCircle = null;
      } else {
        state.selectedCircle = param;
      }
  }
}

```



```
        break;
    }

    update();
}

...
```

As an exercise, modify the CodePen example with the modified `action` function. Now when you click a selected circle it'll get deselected.

## 25.2 Summary

We've introduced a state management pattern based on the Flux pattern. There are two main aspects:

- an object (named `state`) which contains application state
- a function (named `action`) which modifies state and triggers a redraw

This is a simple pattern but it's surprisingly powerful. It's by far the pattern I use the most when building interactive data visualisations.

## Notes

65 <https://github.com/facebook/flux/tree/master/examples/flux-concepts>

66 <https://codepen.io/createwithdata/pen/YzwGaBw>

# 26. Practical: Add State Management

In this section we'll add state management to Energy Explorer using the approach covered in the State Management chapter. We'll add a state object (that represents the current state) and an action function (for managing state changes).

## 26.1 Overview

Open `d3-start-to-finish-code/step11`. The file structure is:

```
step11
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   ├── d3.min.js
    │   └── popup-v1.1.1.min.js
    ├── main.js
    ├── popup.js
    └── update.js
```

In this practical we:

1. Add a new module `js/store.js`.
2. Add a new object state for storing state and a new function `action` that manages state changes.

## 26.2 Add a new module

Create a new file within the `js` directory and name it `store.js`. (It's named `store.js` because `store` is the name commonly used in the Flux pattern.)

Your directory structure should now look like:

```
step11
├─ css
│   └─ style.css
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   ├─ d3.min.js
    │   └─ popup-v1.1.1.min.js
    ├─ main.js
    ├─ popup.js
    └─ store.js
        └─ update.js
```

In index.html link to the new file:

#### index.html

---

```
<!DOCTYPE html>
<html lang="en">

...
<script src="js/lib/d3.min.js"></script>
<script src="js/lib/popup-v1.1.1.min.js"></script>

<script src="js/config.js"></script>
<script src="js/store.js"></script>
<script src="js/layout.js"></script>
<script src="js/update.js"></script>
<script src="js/popup.js"></script>
<script src="js/main.js"></script>
</body>
</html>
```

---

## 26.3 Add an empty state object and action function

In js/store.js add a new empty state object and an action function:

js/store.js

---

```
let state = {  
};  
  
function action(type, param) {  
  switch(type) {  
  }  
  
  update();  
}
```

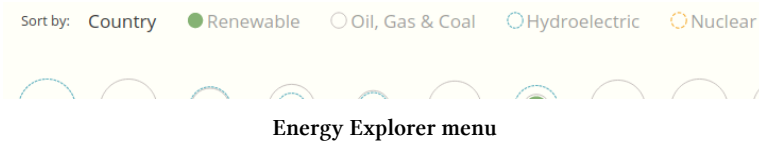
---

We haven't added any properties to the state object (nor any associated actions). We'll do this in the next practical.

Save `index.html` and `store.js`, refresh your browser (making sure it's loading `step11`) and check Energy Explorer is still working.

## 27. Practical: Add a Menu

In this practical we add a **menu** to Energy Explorer that lets the user select which indicator to sort the countries by:



The menu consists of 5 items: 'Country', 'Renewable', 'Oil, Gas & Coal', 'Hydroelectric' and 'Nuclear'. We'll add a property to the state object that stores which indicator has been selected. When a menu item is clicked we update the state accordingly. We'll implement sorting in a later practical.

### 27.1 Overview

A new module named `menu.js` will be added which contains an array of menu items:

```
[
  {
    id: 'country',
    label: 'Country'
  },
  {
    id: 'renewable',
    label: 'Renewable'
  },
  ...
]
```

We use D3 to join the menu item array to `div` elements. The resulting HTML will look something like:

```

<div class="menu">
  <div class="items">
    <div class="item">Country</div>
    <div class="item">Renewable</div>
    <div class="item">Oil, Gas & Coal</div>
    <div class="item">Hydroelectric</div>
    <div class="item">Nuclear</div>
  </div>
</div>

```

When a menu item is clicked a new state property `selectedIndicator` is updated and the chart and menu are redrawn.

Open `d3-start-to-finish-code/step11` (which we already worked on in the previous practical). The file structure is:

```

step11
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   ├── d3.min.js
    │   └── popup-v1.1.1.min.js
    ├── main.js
    ├── popup.js
    ├── store.js
    └── update.js

```

`js/store.js` was added in the previous practical.

In this practical we:

1. Add a container for the menu in `index.html`.
2. Add a new property `selectedIndicator` to the state object.
3. Add a new module `js/menu.js`.
4. Add code to construct and manage the menu.
5. Add basic styling to the menu.
6. Modify the update function so that it also updates the menu.

## 27.2 Add a container for the menu

In `index.html` make the following changes:

`index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <div id="wrapper">
      <div id="controls">
        <div class="menu">
          <div class="items"></div>
        </div>
      </div>
      <div id="chart-wrapper">
        <svg width="1200" height="1200">
          <g id="chart"></g>
        </svg>
      </div>
    </div>

    <script src="js/lib/d3.min.js"></script>
    <script src="js/lib/popup-v1.1.1.min.js"></script>

    <script src="js/config.js"></script>
    <script src="js/store.js"></script>
    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/popup.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

We add two new `<div>` elements to the main wrapper `<div id="wrapper">`. The first (with `id="controls"`) acts as a container for Energy Explorer's controls (including the menu). The second `<div>` (with `id="chart-wrapper"`) acts as a container for the chart's `<svg>` element.

## 27.3 Add selectedIndicator state property

In `js/store.js` add a new property `selectedIndicator` which indicates which energy indicator the user has selected. Also add a new action type `setSelectedIndicator` which updates `state.selectedIndicator`.

`js/store.js`

---

```
let state = {  
  selectedIndicator: 'country'  
};  
  
function action(type, param) {  
  switch(type) {  
    case 'setSelectedIndicator':  
      state.selectedIndicator = param;  
      break;  
  }  
  
  update();  
}
```

---

## 27.4 Add a new module for the menu

Create a new file within the `js` directory and name it `menu.js`. Your directory structure should now look like:

```
step11  
├─ css  
│   └─ style.css  
├─ data  
│   └─ data.csv  
├─ index.html  
└─ js  
    ├─ config.js  
    ├─ layout.js  
    ├─ lib  
    │   ├─ d3.min.js  
    │   └─ popup-v1.1.1.min.js  
    ├─ main.js  
    └─ menu.js  
    ├─ popup.js  
    ├─ store.js  
    └─ update.js
```



In `index.html` link to the new file:

#### `index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>

  <body>
    ...

    <script src="js/lib/d3.min.js"></script>
    <script src="js/lib/popup-v1.1.1.min.js"></script>

    <script src="js/config.js"></script>
    <script src="js/store.js"></script>
    <script src="js/layout.js"></script>
    <script src="js/update.js"></script>
    <script src="js/popup.js"></script>
    <script src="js/menu.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

---

## 27.5 Add code to construct and manage the menu

In `menu.js` add code to create and update the menu items using a D3 join. Also add code to handle menu item clicks:

#### `js/menu.js`

---

```
let menuItems = [
  {
    id: 'country',
    label: 'Country'
  },
  {
    id: 'renewable',
    label: 'Renewable'
  },
  {
    id: 'oilgascoal',
```

```

        label: 'Oil, Gas & Coal'
    },
    {
        id: 'hydroelectric',
        label: 'Hydroelectric'
    },
    {
        id: 'nuclear',
        label: 'Nuclear'
    }
];

function handleMenuClick(e, d) {
    action('setSelectedIndicator', d.id);
}

function updateMenu() {
    d3.select('#controls .menu .items')
        .selectAll('.item')
        .data(menuItems)
        .join('div')
        .classed('item', true)
        .classed('selected', function(d) {
            return state.selectedIndicator === d.id;
        })
        .text(function(d) {
            return d.label;
        })
        .on('click', handleMenuClick);
}

```

---

menuItems is an array containing objects for each menu item. Each object consists of an id and label. updateMenu uses D3 to join menuItems to <div> elements. You might be surprised to see D3 used in this way - D3 can be used to join arrays to HTML elements, not just SVG elements! This code is standard join code, as covered in the Data Joins chapter. We also register an event handler handleMenuClick for click events.

When a menu item is clicked, handleMenuClick is called. This calls action which updates state.selectedIndicator and calls update. We'll modify update later on to call updateMenu. When updateMenu is called, it'll add a class attribute selected to the selected menu item. This'll allow us to style the selected menu item accordingly.

## 27.6 Basic styling of the menu

Now add some CSS rules to `css/style.css` for styling the menu:

`css/style.css`

---

```
body {
    background-color: #FFFFFF7;
    cursor: default;
}

#wrapper {
    width: 1200px;
    margin: 0 auto;
}

.menu .items {
    display: flex;
}

.menu .item {
    padding: 0 1rem;
    color: #333;
    opacity: 0.5;
}

.menu .item.selected, .menu .item:hover {
    opacity: 1;
}

circle.renewable {
    fill: #7FB069;
}

...
```

---

We set `cursor` to `default` on the `body` element so that the cursor pointer always displays as an arrow. (Without this, the text select pointer is displayed when menu items are hovered.)

The menu item container (selected by `.menu .items`) is given a `display` of `flex` which arranges the menu items in a row. (You can find out more about Flexbox in the [Fundamentals of HTML, SVG, CSS & JavaScript for Data Visualisation<sup>67</sup>](#) book.) Each menu item is given some horizontal padding and given an opacity of 0.5. If a menu item is hovered or selected it's given an opacity of 1.

## 27.7 Call updateMenu from update function

Finally we need to make sure the menu is redrawn each time the state changes. In `js/update.js` rename the function `update` to `updateChart`. Then create a new function named `update` which calls `updateChart` and `updateMenu`:

`js/update.js`

---

```
function initialiseGroup(g) {
  ...
}

function updateGroup(d, i) {
  ...
}

function updateChart() {
  let layoutData = layout(data);

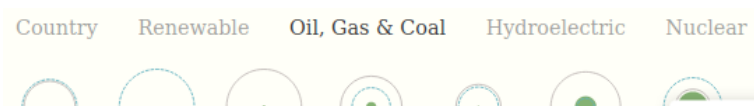
  d3.select('#chart')
    .selectAll('g')
    .data(layoutData)
    .join('g')
    .each(updateGroup);
}

function update() {
  updateChart();
  updateMenu();
}
```

---

Whenever `update` is called, the chart and menu are updated. For example, when the `setSelectedIndicator` action is invoked, `update` is called which updates both the chart and menu.

Now make sure `index.html`, `css/style.css`, `js/store.js`, `js/update.js` and `js/menu.js` are saved and view `step11` in your browser. There should now be a menu at the top. When you click an item it should highlight in a darker gray:



Energy Explorer menu with Oil, Gas & Coal selected

The chart itself won't change when a menu item is clicked – that'll be added in an upcoming practical.

## 27.8 Summary

In this practical you added a menu to Energy Explorer. When a menu item is clicked the `handleMenuClick` event handler (in `js/menu.js`) is called. This event handler calls `action` and passes in an action type of `setSelectedIndicator` and an action parameter of the menu item's id (e.g. `'renewable'`). The action function updates `state.selectedIndicator` and calls `update` which causes the chart and menu to be redrawn.

Currently only the menu responds to `state.selectedIndicator` changing. However in an upcoming practical you'll add code so that the circles are sorted according to the selected indicator. Another thing to note is that the menu was created using D3. This shows that D3 is actually a general purpose library that can create simple user interfaces, as well as data visualisations.

For simple interfaces such as this menu, D3 is fine to use. However I'd recommend a library such as React or Vue for more complicated interfaces.

## Notes

<sup>67</sup> <https://leanpub.com/html-svg-css-js-for-data-visualisation>

## 28. Data Manipulation

This chapter introduces another JavaScript library called Lodash (<https://lodash.com/><sup>68</sup>). Lodash is a general purpose utility library that provides a large number of functions that perform common tasks. It's especially useful when working with data (in particular arrays and objects). Later on you'll use it in Energy Explorer to sort the countries.

One of the advantages of using a library such as Lodash is that it overcomes web browser differences. It's perhaps not as prevalent as it once was, but some commonly used functions weren't available in certain web browsers. Using Lodash was a convenient way of overcoming this issue. It's beyond this book to show everything that Lodash can do, but we'll look at some of the functions I commonly use when building data visualisations.

If you wish to try any of the following examples out you can do so by visiting jsconsole at <https://jsconsole.com><sup>69</sup> and entering:

```
:load lodash
```

This loads Lodash into jsconsole and you can type in statements like `_.uniq([40, 30, 40, 10, 10])`.

### 28.1 Installing Lodash

Lodash can be installed in a number of ways. In this book, we'll keep things simple and include it by downloading it from the home page at <https://lodash.com/><sup>70</sup> and including it using a script tag in `index.html`.

### 28.2 Lodash syntax

Lodash is an **object** that's represented by the symbol `_` (underscore) and has a large number of **methods** defined on it. (A method is a function that's been assigned to an object property.) For example there's a method `_.sortBy` which takes an array and returns a sorted copy of the array. You invoke it using:

```
let sortedNumbers = _.sortBy([10, 50, 30, 20]);
```



It may seem strange to have a library named `_` but it's a valid character as far as variable names are concerned.

Typically Lodash doesn't change (or mutate) objects or arrays. Instead it generally returns a new object or array. The fact that it doesn't change anything (or produce 'side effects') makes your code more predictable. Let's look at some of the most useful Lodash methods when working with data.

## 28.3 `_.uniq`

The `_.uniq` method takes an array and returns a copy of the array, with all duplicates removed. Here's an example:

```
_.uniq([10, 20, 50, 40, 20, 20, 10]); // [10, 20, 50, 40]
```

I encourage you to try out each of these examples using [jsconsole](#)<sup>71</sup>.

## 28.4 `_.includes`

The `_.includes` method takes an array and a value and returns `true` if the value is in the array. For example:

```
let data = [10, 20, 40, 50];  
_.includes(data, 20); // true  
_.includes(data, 30); // false
```

## 28.5 `_.without`

The `_.without` method takes an array and a value and returns a copy of the array, with instances of the provided value excluded. For example:

```
_.without([10, 20, 30, 40, 50], 20); // [10, 30, 40, 50]
```

## 28.6 `_.filter`

The `_.filter` method takes an array and a function. It returns a copy of the array containing only the elements for which the function returns `true`. For example, let's filter an array and only include elements with a value over 30:

```
_.filter([10, 20, 30, 40, 50], function(d) {  
  return d > 30;  
});  
// [40, 50]
```

`_.filter` can also operate across arrays of objects. For example:

```
let data = [  
  {  
    name: "Paris",  
    region: "Europe",  
    indicator1: 9030,  
    indicator2: 13.45  
  },  
  {  
    name: "Madrid",  
    region: "Europe",  
    indicator1: 3912,  
    indicator2: 45.41  
  },  
  {  
    name: "New York",  
    region: "North America",  
    indicator1: 19481,  
    indicator2: 32.53  
  }  
]  
  
_.filter(data, function(d) {  
  return d.region === "Europe";  
});  
  
// Returns:  
[  
  {  
    name: "Paris",  
    region: "Europe",  
    indicator1: 9030,
```



```
        indicator2: 13.45
    },
    {
        name: "Madrid",
        region: "Europe",
        indicator1: 3912,
        indicator2: 45.41
    }
];

_.filter(data, function(d) {
    return d.indicator1 > 10000;
});

// Returns:
[
  {
    name: "New York",
    region: "North America",
    indicator1: 19481,
    indicator2: 32.53
  }
]
```

## 28.7 \_.sortBy

The `_.sortBy` method takes an array and returns a sorted copy of the array. For example:

```
_.sortBy([50, 30, 100, 3, 20]) // [ 3, 20, 30, 50, 100 ]
```

If you have an array of objects you can pass a function as the second argument to specify which property to sort by:

```
let data = [
  {
    name: "Paris",
    region: "Europe",
    indicator1: 9030,
    indicator2: 13.45
  },
  {
    name: "Madrid",
    region: "Europe",
    indicator1: 3912,
    indicator2: 45.41
  },
  {
    name: "New York",
    region: "North America",
    indicator1: 19481,
    indicator2: 32.53
  }
];

_.sortBy(data, function(d) {
  return d.indicator1;
});

// Returns:
[
  {
    name: "Madrid",
    region: "Europe",
    indicator1: 3912,
    indicator2: 45.41
  },
  {
    name: "Paris",
    region: "Europe",
    indicator1: 9030,
    indicator2: 13.45
  },
  {
    name: "New York",
    region: "North America",
    indicator1: 19481,
    indicator2: 32.53
  }
]
```

You can also pass in the name of a property instead of a function. The following two calls are equivalent:

```
_.sortBy(data, function(d) {  
  return d.indicator1;  
});  
  
_.sortBy(data, 'indicator1');
```

JavaScript has a built in sort function but `_.sortBy` has two differences: it returns a new array (rather than replacing the original array) and its number sorting is sane:

```
let data = [50, 30, 100, 3, 20];  
  
_.sortBy(data); // [ 3, 20, 30, 50, 100 ]  
  
data.sort(); // data is now [ 100, 20, 3, 30, 50 ]
```

The built in JavaScript sort function converts numbers to strings and sorts alphanumerically which is why you get a surprising result.

## 28.8 `_.orderBy`

The `_.orderBy` method is similar to `_.sortBy` except a string `'asc'` or `'desc'` can be passed in as the third parameter to specify whether the sort should be ascending or descending. For example:

```
let data = [  
  {  
    name: "Paris",  
    region: "Europe",  
    indicator1: 9030,  
    indicator2: 13.45  
  },  
  {  
    name: "Madrid",  
    region: "Europe",  
    indicator1: 3912,  
    indicator2: 45.41  
  },  
  {  
    name: "New York",
```

```
    region: "North America",
    indicator1: 19481,
    indicator2: 32.53
  }
];

_.orderBy(data, function(d) {
  return d.indicator1;
}, 'desc');

// Returns:
[
  {
    name: "New York",
    region: "North America",
    indicator1: 19481,
    indicator2: 32.53
  },
  {
    name: "Paris",
    region: "Europe",
    indicator1: 9030,
    indicator2: 13.45
  },
  {
    name: "Madrid",
    region: "Europe",
    indicator1: 3912,
    indicator2: 45.41
  }
]
```

## 28.9 Summary

This chapter has introduced the Lodash library and looked at a few of its methods that are useful for data processing. In the next chapter you'll use Lodash to sort the Energy Explorer countries according to the chosen indicator.

## Notes

68 <https://lodash.com/>

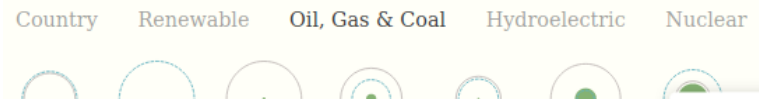
69 <https://jsconsole.com>

70 <https://lodash.com/>

71 <https://jsconsole.com/?%3Aload%20lodash>

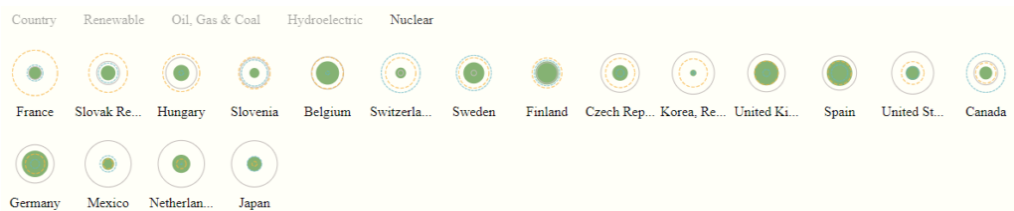
## 29. Practical: Sort the Countries

In this practical we'll add code that sorts the countries according to `state.selectedIndicator`. The sorting will be carried out **in the layout function** using Lodash's `_.orderBy` method. Remember that `state.selectedIndicator` is updated whenever a menu item is clicked.



Energy Explorer menu

We also add a feature whereby countries which have a zero value (or no data) for the selected indicator will be hidden. This isn't an essential feature but I think it's a nice one to have. It's also an opportunity to add a simple filter to Energy Explorer. For example, when Nuclear is selected you'll see:



When Nuclear is selected, countries with no nuclear energy are filtered out

### 29.1 Overview

Open `d3-start-to-finish-code/step12`. The file structure is:

```

step12
├─ css
│   └─ style.css
├─ data
│   └─ data.csv
├─ index.html
└─ js
    ├─ config.js
    ├─ layout.js
    ├─ lib
    │   └─ d3.min.js
    │   └─ lodash.min.js
    └─ popup-v1.1.1.min.js
        ├─ main.js
        ├─ menu.js
        ├─ popup.js
        ├─ store.js
        └─ update.js

```

Note that lodash has been added for you. (It was downloaded from <https://lodash.com/><sup>72</sup>.) In this practical we:

1. Link to lodash in `index.html`.
2. Add a function to sort the data.
3. Hide countries which have a zero value (or no data) for the selected indicator.

## 29.2 Link to Lodash

In `index.html` add a script tag to load `js/lib/lodash.min.js`:

**index.html**

---

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    ...

    <script src="js/lib/d3.min.js"></script>

```

```
<script src="js/lib/popup-v1.1.1.min.js"></script>
<script src="js/lib/lodash.min.js"></script>

<script src="js/config.js"></script>
<script src="js/store.js"></script>
<script src="js/layout.js"></script>
<script src="js/update.js"></script>
<script src="js/popup.js"></script>
<script src="js/menu.js"></script>
<script src="js/main.js"></script>
</body>
</html>
```

---

Now save index.html.

## 29.3 Sort the data

Recall that data looks something like:

```
[
  {
    "id": "ALB",
    "name": "Albania",
    "hydroelectric": 100,
    "nuclear": null,
    "oilgascoal": 0,
    "renewable": 0
  },
  {
    "id": "DZA",
    "name": "Algeria",
    "hydroelectric": 0.2,
    "nuclear": null,
    "oilgascoal": 99.7,
    "renewable": 0.1
  },
  ...
]
```

The selected menu item will determine the property by which the array will be sorted. For example, if ‘Oil, Gas & Coal’ is selected we sort using the property `oilgascoal`. A special case is if the first menu item ‘Country’ is selected we sort using the `name` property.

Make the following changes in `js/layout.js`:



js/layout.js

---

```

function sortAccessor(d) {
  let value = d[state.selectedIndicator];
  if(isNaN(value)) value = 0;
  return value;
}

function getSortedData(data) {
  let sorted;

  if(state.selectedIndicator === 'country') {
    sorted = _.orderBy(data, 'name');
  } else {
    sorted = _.orderBy(data, sortAccessor, 'desc');
  }

  return sorted;
}

function isVisible(d) {
  return state.selectedIndicator === 'country' || d[state.selectedIndicator] > 0;
}

function getTruncatedLabel(text) {
  return text.length <= 10 ? text : text.slice(0, 10) + '...';
}

function layout(data) {
  let labelHeight = 20;
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth + labelHeight;

  let maxRadius = 0.35 * cellWidth;

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let sortedData = getSortedData(data);

  let layoutData = data.map(function(d, i) {
  let layoutData = sortedData.map(function(d, i) {
    let item = {};

    let column = i % config.numColumns;
    let row = Math.floor(i / config.numColumns);

```

```
    item.x = column * cellWidth + 0.5 * cellWidth;
    item.y = row * cellHeight + 0.5 * cellHeight;

    item.visible = isVisible(d);

    item.renewableRadius = radiusScale(d.renewable);
    item.oilGasCoalRadius = radiusScale(d.oilgascoal);
    item.hydroelectricRadius = radiusScale(d.hydroelectric);
    item.nuclearRadius = radiusScale(d.nuclear);

    item.labelText = getTruncatedLabel(d.name);
    item.labelOffset = maxRadius + labelHeight;

    item.popupOffset = -0.8 * maxRadius;
    item.popupData = {
        name: d.name,
        renewable: d.renewable,
        oilgascoal: d.oilgascoal,
        hydroelectric: d.hydroelectric,
        nuclear: d.nuclear
    };

    return item;
});

return layoutData;
}
```

---

Near the top of function `layout`, before iterating through the data, we call `getSortedData` and assign the output to `sortedData`.

`getSortedData` uses `lodash's orderBy` function to create a sorted copy of data. If `state.selectedIndicator` is `'country'` it sorts by the `name` property (so the countries are sorted alphabetically). Otherwise it sorts using the `sortAccessor` function which extracts the selected indicator value from a data item. (This was covered in the `Data Manipulation` chapter.)

For example `sortAccessor` might get called with `d` equal to:

```
{
  "name": "Angola",
  "id": "AGO",
  "hydroelectric": 53.2,
  "nuclear": NaN,
  "oilgascoal": 46.8,
  "renewable": 0.0
}
```

If `state.selectedIndicator` is `'oilgascoal'`, we evaluate `d['oilgascoal']` which returns 46.8. This value gets assigned to `value`. If `value` is `NaN` we return zero, which results in the items with no data appearing at the end of the sorted array.

To summarise, `getSortedData` returns a sorted copy of data. If 'Country' is selected in the menu, the data is sorted alphabetically. Otherwise it's sorted by the selected indicator, in descending order. Countries with missing data are placed at the end of the sorted array.

We also add a new property `visible` to the layout items. This indicates whether the country should be visible. If the country's selected indicator value is zero (or missing), we set `visible` to `false`, otherwise it's set to `true`.

## 29.4 Hide countries with a zero or missing value

In `js/update.js` update the country's opacity according to the `visible` property:

`js/update.js`

---

```
function initialiseGroup(g) {
  ...
}

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) initialiseGroup(g);

  g.attr('transform', 'translate(' + d.x + ', ' + d.y + ')')
    .style('opacity', d.visible ? 1 : 0)
    .style('pointer-events', d.visible ? 'all' : 'none');

  g.select('.popup-center')
    .attr('cy', d.popupOffset);

  g.select('.renewable')
    .attr('r', d.renewableRadius);
```

```

    ...
  }

function updateChart() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('g')
    .data(layoutData)
    .join('g')
    .each(updateGroup);
}

function update() {
  updateChart();
  updateMenu();
}

```

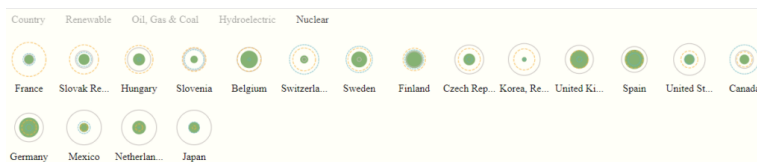
---

We set the opacity of the country group according to the new `visible` property. We could've set the display CSS property to `inline` or `none` in order to show or hide the country group but we use opacity so that it can be animated later on.

We also need to set the `pointer-events` property so that the popup doesn't appear on hidden groups. (Pointer events are still active if the opacity is zero.)

Now save `layout.js` and `update.js`. Load `step12` in your browser and click on the menu items. The circles should sort according to the selected indicator. You should also see that countries with a zero or missing indicator value are hidden.

For example click on 'Nuclear' and you'll see:



Countries sorted by Nuclear energy

## 29.5 Summary

In this practical we added sort functionality to Energy Explorer. All the important sort logic was added to the layout module and we didn't need to change any of the rendering code in `update.js`. This is a nice separation of concerns.

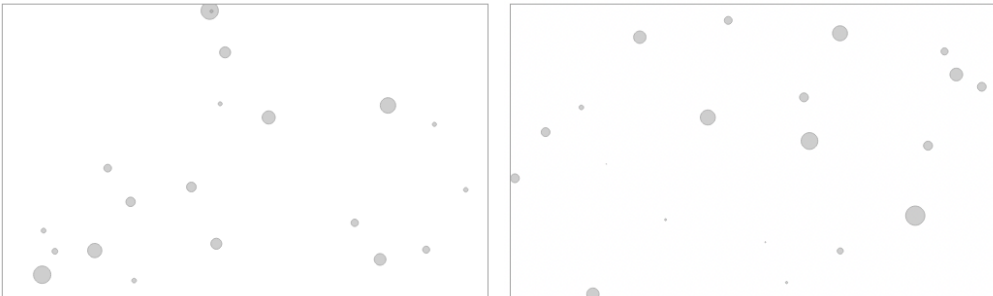
We also added filtering so that if an energy indicator such as 'Oil, Gas & Coal' is selected only countries with a value above zero are visible. We used the `opacity` property to show or hide country groups. This is so that when animations are added (which will be done in an upcoming practical) the countries will fade in or out.

## Notes

<sup>72</sup> <https://lodash.com/>

## 30. D3 Transitions

A powerful effect that adds visual flair to a data visualisation is animation. This can make a visualisation more engaging but it also serves a practical purpose. Suppose you've a visualisation consisting of a number of circles and the positions of the circles change (for example, a new indicator has been selected). If the circles animate into their new positions it's possible to track individual circles and to see where they move to. Without animation it's not obvious where each circle has moved to.



Before and after positions. Can you determine where each circle has moved to?

Open the CodePen example <https://codepen.io/createwithdata/pen/pogPwOy><sup>73</sup>. If animations are switched off ('Use transitions' is unchecked) the circles jump immediately to their new positions. It's not possible to track the movement of each circle. Furthermore, it's not discernable whether you're seeing circle movement, or an entirely new set of circles. With animations switched on, it's clear that the circles are changing position. This is known as **object constancy**.

D3 has a powerful system for animating (or 'transitioning') HTML/SVG elements. You'll be glad to know you've already done a lot of the hard work and adding animations to a selection of elements is relatively straightforward.

### 30.1 Creating D3 transitions

A D3 transition is a **special type of selection** that animates (or 'transitions') style and attribute updates. For example if you use `.style` to update the `opacity` of a selection:

```
let s = d3.select('circle');  
  
s.style('opacity', 0);
```

a D3 transition will animate `opacity` from its current value to its new value.

To create a transition add a call to `.transition()` before the `.style` and `.attr` calls. For example:

```
let s = d3.select('circle');  
  
s.transition()  
  .style('opacity', 0);
```

This results in the opacity animating from its current value to zero. (If its current value is already zero there'll be no animation.)

Let's look at a more substantial example. Suppose you have the following code which repeatedly draws a set of circles at random positions:

```
let myData = [];  
let width = 800, height = 600, numCircles = 20;  
  
function updateData() {  
  for(let i=0; i<numCircles; i++) {  
    myData[i] = {  
      x: Math.floor(Math.random() * width),  
      y: Math.floor(Math.random() * height),  
      r: Math.random() * 30  
    };  
  }  
}  
  
function update() {  
  d3.select('#chart')  
    .selectAll('circle')  
    .data(myData)  
    .join('circle')  
    .attr('cx', function(d) {  
      return d.x;  
    })  
    .attr('cy', function(d) {  
      return d.y;  
    })  
    .attr('r', function(d) {
```

```

        return d.r;
    });
}

window.setInterval(function() {
    updateData();
    update();
}, 1000);

```

`window.setInterval` creates a timer which calls the provided function every 1000 milliseconds. This results in `updateData` and `update` being called every second.

`updateData` updates each element of `myData` with an object containing `x`, `y` and `r` properties. `update` joins `myData` to `circle` elements and updates the circle center (`cx` and `cy`) and radius (`r`) using the joined value.

Navigate to <https://codepen.io/createwithdata/pen/jOqbBzp><sup>74</sup> to view this example in CodePen.

Currently the circles jump to their new positions and radii. To introduce a transition, add a call to `.transition()` just before the first `.attr` call:

```

...

function update() {
    d3.select('#chart')
      .selectAll('circle')
      .data(myData)
      .join('circle')
      .transition()
      .attr('cx', function(d) {
        return d.x;
      })
      .attr('cy', function(d) {
        return d.y;
      })
      .attr('r', function(d) {
        return d.r;
      });
}

...

```

This causes the `cx`, `cy` and `r` attribute updates to animate to their new values.

Navigate to <https://codepen.io/createwithdata/pen/BaKoWQZ><sup>75</sup> to view this example in CodePen.



Only style and attribute updates after the `.transition()` will animate. Thus the following only animates the circle radius:

```
d3.select('#chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('cx', function(d) {
    return d.x;
  })
  .attr('cy', function(d) {
    return d.y;
  })
  .transition()
  .attr('r', function(d) {
    return d.r;
  });
```

## 30.2 Transition duration

The duration of the transition can be modified by calling `.duration` after the `.transition` call and passing in the transition duration (in milliseconds). For example:

```
function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(myData)
    .join('circle')
    .transition()
    .duration(1000)
    .attr('cx', function(d) {
      return d.x;
    })
    .attr('cy', function(d) {
      return d.y;
    })
    .attr('r', function(d) {
      return d.r;
    })
  };
}
```

The above sets the transition duration to 1 second. (The default duration is 250 milliseconds.)

As an exercise try adding `.duration(1000)` to the previous Codepen example.

## 30.3 Transition delay

You can add a delay to a transition by calling `.delay` and passing in the delay (in milliseconds). For example:

```
function update() {  
  d3.select('#chart')  
    .selectAll('circle')  
    .data(myData)  
    .join('circle')  
    .transition()  
    .duration(1000)  
    .delay(250)  
    .attr('cx', function(d) {  
      return d.x;  
    })  
    .attr('cy', function(d) {  
      return d.y;  
    })  
    .attr('r', function(d) {  
      return d.r;  
    })  
    .attr('r', function(d) {  
      return d.r;  
    });  
}
```

The above delays the transition for 250 milliseconds. Adding a delay becomes more interesting when you pass in a function:

```
function update() {  
  d3.select('#chart')  
    .selectAll('circle')  
    .data(myData)  
    .join('circle')  
    .transition()  
    .duration(1000)  
    .delay(function(d, i) {  
      return i * 20;  
    })  
    .attr('cx', function(d) {  
      return d.x;  
    })  
    .attr('cy', function(d) {  
      return d.y;  
    })  
    .attr('r', function(d) {  
      return d.r;  
    });  
}
```

```
        return d.r;  
    });  
}
```

The above delays each element in the selection by  $i * 20$  where  $i$  is the element's index. This results in a staggered transition (the first element in the selection moves immediately, the second element after 20 milliseconds etc.). You'll use this technique in the next practical.

As an example, try adding:

```
.delay(function(d, i) {  
    return i * 20;  
})
```

to the previous CodePen example <https://codepen.io/createwithdata/pen/BaKoWQZ<sup>76</sup>>.

## 30.4 Key functions

When you join an array to HTML/SVG elements, D3 joins the 1st array element to the 1st HTML/SVG element in the selection, the 2nd array element to the 2nd HTML/SVG element and so on. This means that if you sort the array, and perform the join again, the array elements might get joined to different HTML/SVG elements.

In a lot of cases this doesn't cause issues but if you're positioning the HTML/SVG elements according to the array index  $i$  and using transitions you'll lose object constancy.

For example, examine the following code:

```
let myData;  
  
function updateData() {  
    myData = [];  
  
    for(let i=0; i<7; i++) {  
        myData[i] = {  
            id: i,  
            value: Math.floor(Math.random() * 50)  
        };  
    }  
  
    myData = _.sortBy(myData, function(d) {  
        return d.value;  
    });  
}
```

```

    });
}

function update() {
  d3.select('#chart')
    .selectAll('circle')
    .data(myData)
    .join('circle')
    .transition()
    .attr('cx', function(d, i) {
      return i * 100;
    })
    .attr('r', function(d) {
      return d.value;
    });
}

window.setInterval(function() {
  updateData();
  update();
}, 1000);

```

Every 1 second, `updateData` and `update` are called. `updateData` updates `myData` with objects containing two properties `id` and `value`. `myData` is then sorted by `value`.

`update` joins `myData` to `circle` elements and updates the `cx` and `r` attributes of each circle. `cx` is set according to the array index `i`. (Because `myData` is sorted in ascending order, the leftmost circle will always be the smallest circle.)

When the above code runs the circles change radius, but they don't shuffle into their new locations. Circles stay in same positions even though array has shuffled.

Navigate to <https://codepen.io/createwithdata/pen/wvyrOyd><sup>77</sup> to view this example in CodePen.

To ensure that each array element stays joined to the same `circle` element you need to specify a **key function**. This is a function that's passed into `.data` and returns a unique `id` for each element:

```
function update() {  
  d3.select('#chart')  
    .selectAll('circle')  
    .data(myData, function(d) {  
      return d.id;  
    })  
    .join('circle')  
    .transition()  
    .attr('cx', function(d, i) {  
      return i * 100;  
    })  
    .attr('r', function(d) {  
      return d.value;  
    });  
}
```

Now the circles will shuffle into their new locations:

Navigate to <https://codepen.io/createwithdata/pen/QWQgomO><sup>78</sup> to view this example in CodePen.

Key functions are simple to add but not particularly easy to understand. The situations in which they are essential are quite subtle. I tend to add a key function if I run into the situation described above but in general you won't go wrong if you always add a key function when joining data.

## 30.5 Summary

This section has shown how you can add animations (or 'transitions') to a D3 selection. It's as simple as preceding the `.style` and `.attr` calls (which you want to animate) with a single `.transition()` call. You can change the duration of transitions using `.duration` and can also create staggered transitions by passing a function into `.delay`. You also saw the importance of adding a key function if your array changes order, you're positioning by index and are using transitions.

In the next practical you'll add transitions to Energy Explorer so that when a new indicator is chosen the country groups animate into their new locations.

## Notes

<sup>73</sup> <https://codepen.io/createwithdata/pen/pogPwOy>

<sup>74</sup> <https://codepen.io/createwithdata/pen/jOqbBzp>

75 <https://codepen.io/createwithdata/pen/BaKoWQZ>

76 <https://codepen.io/createwithdata/pen/BaKoWQZ>

77 <https://codepen.io/createwithdata/pen/wvyrOyd>

78 <https://codepen.io/createwithdata/pen/QWQqomO>

# 31. Practical: Add Transitions

In this chapter we add transitions to Energy Explorer so that:

- The countries fade in when the explorer first loads.
- The circles expand out from zero radius when the explorer loads.
- The countries animate into new locations when the sort order changes.

## 31.1 Overview

Open `d3-start-to-finish-code/step13`. The file structure is:

```
step13
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   ├── d3.min.js
    │   ├── lodash.min.js
    │   └── popup-v1.1.1.min.js
    ├── main.js
    ├── menu.js
    ├── popup.js
    ├── store.js
    └── update.js
```

In this practical we:

1. Add a key function to the data join.
2. Add a transition to the countries (so their position and opacity is animated).
3. Add a configurable transition duration and delay.
4. Initialise the country group positions.
5. Add a transition to the circle radii (so they expand when the page loads).

## 31.2 Add a key function to the data join

In the previous chapter we discussed the importance of adding a key function to the data join if:

- the array is sorted
- the elements are **positioned according to array index** and
- **transitions** are used

Energy Explorer meets all 3 of these criteria so we need to add a key function to the data join.

In `js/layout.js` add an `id` property to each item:

`js/layout.js`

---

```
function sortAccessor(d) { ... }

function getSortedData(data) { ... }

function isVisible(d) { ... }

function getTruncatedLabel(text) { ... }

function layout(data) {
  ...

  let sortedData = getSortedData(data);

  let layoutData = sortedData.map(function(d, i) {
    let item = {};

    item.id = d.id;

    let column = i % config.numColumns;
    let row = Math.floor(i / config.numColumns);

    item.x = column * cellWidth + 0.5 * cellWidth;
    item.y = row * cellHeight + 0.5 * cellHeight;

    ...

    return item;
  });
};
```



```
    return layoutData;
}
```

---

(Each item in `data` and `sortedData` has a property named `id` so you can use that to set `item.id`. See the Load the Data practical for a reminder of the structure of `data`.)

Now in `updateChart` in `js/update.js` add a key function to the `data` join:

`js/update.js`

---

```
...

function updateChart() {
    let layoutData = layout(data);

    d3.select('#chart')
      .selectAll('g')
      .data(layoutData)
      .data(layoutData, function(d) {
        return d.id;
      })
      .join('g')
      .each(updateGroup);
}

function update() {
    updateChart();
    updateMenu();
}
```

---

The key function returns the `id` property which we added in the previous step. Adding this key function ensures that each array item remains associated with the same HTML/SVG element whenever the join occurs, even if the array is in a different order.

## 31.3 Add transition to the country groups

We now add a transition to the country groups (the `<g>` elements that contain the 4 circles). This will result in the countries animating into new positions when the sort order changes. The opacity will also animate if its value changes, such as when the country's `visible` property changes.

In `updateGroup` (`update.js`) add a call to `.transition()` before the `.attr` call:

js/update.js

---

```
...

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) initialiseGroup(g);

  g.transition()
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')')
    .style('opacity', d.visible ? 1 : 0)
    .style('pointer-events', d.visible ? 'all' : 'none');

  g.select('.popup-center')
    .attr('cy', d.popupOffset);

  g.select('.renewable')
    .transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('r', d.renewableRadius);

  ...
}

...
```

---

Save `js/layout.js` and `js/update.js` and load `step13` in the browser. Now when Energy Explorer loads the circles animate into view and when you choose a new indicator the circles animate into new positions. Also notice that the circles fade out if they have a zero value (or no data) for the chosen indicator.

The transition feels a bit too quick so we'll change the duration in the next section.

## 31.4 Add transition duration and delay

Add two new properties to the `config` object:

### js/config.js

---

```
let config = {  
  width: 1200,  
  numColumns: 14,  
  transitionDuration: 500,  
  transitionDelay: 8  
};
```

---

`transitionDuration` specifies transition duration and `transitionDelay` specifies the transition delay.

Now add calls to `.duration` and `.delay` after the `.transition` in `updateGroup`:

### js/update.js

---

```
...  
  
function updateGroup(d, i) {  
  let g = d3.select(this);  
  
  if(g.selectAll('*').empty()) initialiseGroup(g, d);  
  
  g.transition()  
    .duration(config.transitionDuration)  
    .delay(i * config.transitionDelay)  
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')')  
    .style('opacity', d.visible ? 1 : 0)  
    .style('pointer-events', d.visible ? 'all' : 'none');  
  
  g.select('.popup-center')  
    .attr('cy', d.popupOffset);  
  
  g.select('.renewable')  
    .transition()  
    .duration(config.transitionDuration)  
    .delay(i * config.transitionDelay)  
    .attr('r', d.renewableRadius);  
  
  ...  
}
```

---

Now save `js/config.js` and `js/update.js` and refresh the browser. The transition should be a bit slower now. The transition is also staggered across the country groups, which is a pleasing effect.

## 31.5 Initialise the country group positions

Each country group flies in from the top left. This is because the initial translation transform of each country group is  $(0,0)$ . Rather than flying in we'll fade in each group in its actual position. This is achieved by initialising the opacity and transform of each g element:

js/update.js

---

```
function initialiseGroup(g, d) {
  g.classed('country', true)
  .style('opacity', 0)
  .attr('transform', 'translate(' + d.x + ', ' + d.y + ')')
  .on('mouseover', handleMouseover)
  .on('mouseout', handleMouseout);

  g.append('circle')
    .classed('popup-center', true)
    .attr('r', 1);

  g.append('circle')
    .classed('renewable', true);

  g.append('circle')
    .classed('oilgascoal', true);

  g.append('circle')
    .classed('hydroelectric', true);

  g.append('circle')
    .classed('nuclear', true);

  g.append('text')
    .classed('label', true);
}

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) initialiseGroup(g, d);

  g.transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')')
    .style('opacity', d.visible ? 1 : 0)
```

```

        .style('pointer-events', d.visible ? 'all' : 'none');

    g.select('.popup-center')
      .attr('cy', d.popupOffset);

    ...
  }

function updateChart() {
  let layoutData = layout(data);

  d3.select('#chart')
    .selectAll('g')
    .data(layoutData, function(d) {
      return d.id;
    })
    .join('g')
    .each(updateGroup);
}

function update() {
  updateChart();
  updateMenu();
}

```

---

In `initialiseGroup` we set the opacity on each group to 0 and the transform to `'translate(' + d.x + ', ' + d.y + '')`. Remember that `initialiseGroup` is called when a new country group is created, so this is a good place to initialise the group's style and attributes.

In `updateGroup` we pass the joined data `d` into `initialiseGroup` so that it has access to the group's `x` and `y` properties.

Save `js/update.js` and refresh the browser. Now the circles fade in, without flying in from the top left.

## 31.6 Add a transition to the circle radii

Our final change is to make the circles grow from zero radius. This is a subtle change but one that adds a final bit of polish to the transitions. The default radius of `circle` elements is zero so we don't need to initialise the radii. Make the following changes:

**js/update.js**

---

```
function initialiseGroup(g, d) {
  ...
}

function updateGroup(d, i) {
  let g = d3.select(this);

  if(g.selectAll('*').empty()) initialiseGroup(g, d);

  g.transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('transform', 'translate(' + d.x + ', ' + d.y + ')')
    .style('opacity', d.visible ? 1 : 0)
    .style('pointer-events', d.visible ? 'all' : 'none');

  g.select('.popup-center')
    .attr('cy', d.popupOffset);

  g.select('.renewable')
    .transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('r', d.renewableRadius);

  g.select('.oilgascoal')
    .transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('r', d.oilGasCoalRadius);

  g.select('.hydroelectric')
    .transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('r', d.hydroelectricRadius);

  g.select('.nuclear')
    .transition()
    .duration(config.transitionDuration)
    .delay(i * config.transitionDelay)
    .attr('r', d.nuclearRadius);

  g.select('.label')
    .attr('y', d.labelOffset)
```

```
        .text(d.labelText);
    }

    function updateChart() {
        let layoutData = layout(data);

        d3.select('#chart')
            .selectAll('g')
            .data(layoutData, function(d) {
                return d.id;
            })
            .join('g')
            .each(updateGroup);
    }

    function update() {
        updateChart();
        updateMenu();
    }
}
```

---

Save `js/update.js` and refresh the browser. Now each circle should transition from zero radius to its final radius.

The completed code containing the transitions added in this practical can be found in `step13-complete`.

## 31.7 Summary

We added transitions to Energy Explorer. Adding a basic transition to the group `transform` attribute was a relatively straightforward change but it made a huge difference to the visualisation. Country groups animate to new positions and fade in or out if their visibility changes. We also fine tuned the transitions, modifying the duration and adding a staggered transition for a nice visual effect.

We also saw how the transitions can be fine tuned. We added code to fade in each group in its actual position (rather than flying in).

If you've made it this far and are reasonably happy with what you've learned you deserve a big well done! Most of Energy Explorer is complete and all that's left is adding a legend and some detailed styling.

## 32. Practical: Add a Legend

In this practical we add a legend that indicates how the circle size relates to the percentage value. It'll be located in the top right of the display and consists of a circle and label.



Energy Explorer legend (top right)

The legend circle is sized such that it represents a value of 100%.

### 32.1 Overview

Open `d3-start-to-finish-code/step14`. The file structure is:

```
step14
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   ├── d3.min.js
    │   ├── lodash.min.js
    │   └── popup-v1.1.1.min.js
    ├── main.js
    ├── menu.js
    ├── popup.js
    ├── store.js
    └── update.js
```

In this practical we:



1. Add the legend to `index.html`.
2. Add a function to update the radius of legend circle.
3. Style the legend.

## 32.2 Add the legend to `index.html`

In `index.html` we add the following HTML and SVG for the legend:

### `index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>

  <body>
    <div id="wrapper">
      <div id="controls">
        <div class="menu">
          <div class="items"></div>
        </div>
        <div class="legend">
          <svg width="70" height="70">
            <g transform="translate(35,35)">
              <circle />
              <text>100%</text>
            </g>
          </svg>
        </div>
      </div>
      <div id="chart-wrapper">
        <svg width="1200" height="1200">
          <g id="chart"></g>
        </svg>
      </div>
    </div>

    <script src="js/lib/d3.min.js"></script>
    <script src="js/lib/popup-v1.1.1.min.js"></script>
    <script src="js/lib/lodash.min.js"></script>

    <script src="js/config.js"></script>
    ...
  </body>
</html>
```

---

The legend is added to `div#controls` and consists of an SVG element. The SVG element contains a `<g>` element which is centred within the SVG element. The `<g>` element contains a `<circle>` element and a `<text>` element.

Save `index.html`.

## 32.3 Add a function to update the radius of the legend circle

In `js/layout.js` we create a new function that returns the maximum radius of the country circles. This value is based on the width and number of columns of the chart:

`js/layout.js`

---

```
function sortAccessor(d) { ... }

function getSortedData(data) { ... }

function isVisible(d) { ... }

function getTruncatedLabel(text) { ... }

function getMaxRadius() {
  let cellWidth = config.width / config.numColumns;
  let maxRadius = 0.35 * cellWidth;
  return maxRadius;
}

function layout(data) {
  let labelHeight = 20;
  let cellWidth = config.width / config.numColumns;
  let cellHeight = cellWidth + labelHeight;

  let maxRadius = 0.35 * cellWidth;
  let maxRadius = getMaxRadius();

  let radiusScale = d3.scaleSqrt()
    .domain([0, 100])
    .range([0, maxRadius]);

  let sortedData = getSortedData(data);

  let layoutData = sortedData.map(function(d, i) { ... });
```

```
    return layoutData;
}
```

---

`getMaxRadius` computes the maximum circle radius in Energy Explorer using the same calculation as before. We call `getMaxRadius` in function `layout` and we'll also use it when updating the legend circle.

In `update.js` we add a new function `updateLegend` which uses D3 to update the legend circle:

**js/update.js**

---

```
...

function updateChart() {
    let layoutData = layout(data);

    d3.select('#chart')
      .selectAll('g')
      .data(layoutData, function(d) {
        return d.id;
      })
      .join('g')
      .each(updateGroup);
}
```

```
function updateLegend() {
    d3.select('.legend circle')
      .attr('r', getMaxRadius());
}
```

```
function update() {
    updateChart();
    updateMenu();
    updateLegend();
}
```

---

`updateLegend` selects the legend circle and sets its radius using `getMaxRadius`. This means that if we change the chart width or number of columns, the legend circle will still display the correct radius.

## 32.4 Style the legend

We add some CSS rules for the legend in `css/style.css`:

**css/style.css**

---

```
body { ... }

#wrapper { ... }

#controls {
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.menu .items {
  display: flex;
}

.menu .item {
  padding: 0 1rem;
  color: #333;
  opacity: 0.5;
}

.menu .item.selected, .menu .item:hover {
  opacity: 1;
}

.legend circle {
  stroke: #aaa;
  fill: none;
}

.legend text {
  fill: #777;
  text-anchor: middle;
  dominant-baseline: middle;
}

circle.renewable {
  fill: #7FB069;
}

...
```

---

The `div#controls` element contains the menu and legend. We'd like the menu to be on the left and the legend to be on the right and we use Flexbox to achieve this. (You can find out more about Flexbox in the [Fundamentals of HTML, SVG, CSS & JavaScript for](#)

[Data Visualisation](#)<sup>79</sup> book.)

We set `display` to `flex` on the controls container. We also set `justify-content` to `space-between` to maximise the horizontal space between the menu and legend. Setting `align-items` to `center` aligns the menu and legend vertically.

On the legend itself we set the circle stroke to `#aaa` and `fill` to `none`. The legend's `<text>` element has a `fill` of `#777`. To center the label horizontally `text-anchor` is set to `middle` and to center the label vertically `dominant-baseline` is set to `middle`.

Now save `index.html`, `js/layout.js`, `js/update.js` and `css/style.css`. Load `step14` in the browser and the legend looks like:



Energy Explorer legend (top right)

The completed code for this practical is in `step14-complete`.

## 32.5 Summary

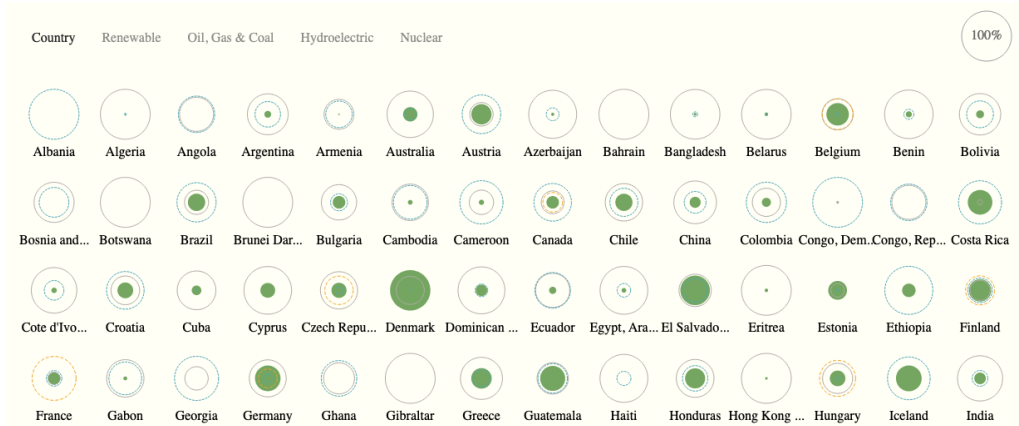
In this practical we added a legend to show the relationship between circle size and percentage value. We added code so that the legend radius is updated using the same logic as the circles in the visualisation. This ensures that the legend stays in step with any changes in visualisation width and number of columns.

## Notes

<sup>79</sup> <https://leanpub.com/html-svg-css-js-for-data-visualisation>

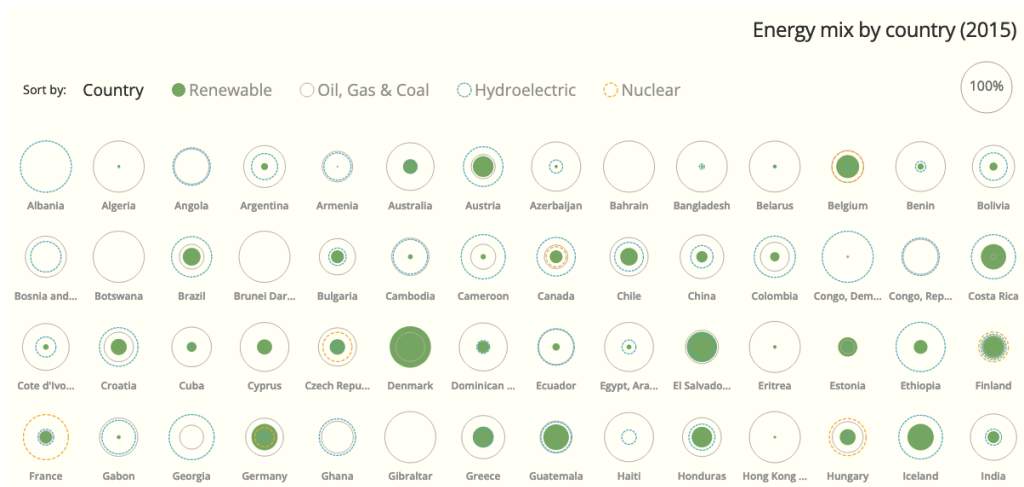
# 33. Practical: Finishing Touches

In this final practical we add some finishing touches to Energy Explorer. The changes are cosmetic and take Energy Explorer from being functional, but not particularly pretty:



Energy Explorer before finishing touches

to a more polished and professional looking data visualisation:



The finished Energy Explorer

These are subtle changes but when added together they make a big difference.

## 33.1 Overview

Open `d3-start-to-finish-code/step15`. The file structure is:

```
step15
├── css
│   └── style.css
├── data
│   └── data.csv
├── index.html
└── js
    ├── config.js
    ├── layout.js
    ├── lib
    │   ├── d3.min.js
    │   ├── lodash.min.js
    │   └── popup-v1.1.1.min.js
    ├── main.js
    ├── menu.js
    ├── popup.js
    ├── store.js
    └── update.js
```

In this practical we:

1. Add and apply the Open Sans Google font.
2. Style the menu.
3. Add a header and footer.
4. Other CSS tweaks.

## 33.2 Add and apply Open Sans font

Currently Energy Explorer is using the web browser's **default font** for all text elements. This is usually a serif font such as Times or Times New Roman. In general sans-serif fonts (such as Helvetica) are recommended for data visualisation text because they tend to be clearer (they have less visual 'noise'). Font support varies across platforms. For example, Helvetica is built into MacOS but not Windows. For this reason I tend to use Google Fonts as this ensures the same font is used regardless of operating system.

There's a huge amount of resources and opinion surrounding font choice and unless you're a trained designer it can be hard to know where to turn. I usually turn to Google

Fonts and filter by Sans Serif and sort by most popular. Currently this returns Roboto, Open Sans and Lato as popular sans serif fonts. I usually try each of these and use my instinct to choose one. In the case of Energy Explorer I've chosen Open Sans. Feel free to choose a different font if you prefer.

In `index.html` add the following line to the `<head>` element:

#### `index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Energy mix by Country 2015</title>
    <link href="https://fonts.googleapis.com/css2?family=Open+Sans:wght@400;600&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    ...
  </body>
</html>
```

---

This line was provided by Google Fonts. To get the line yourself:

- navigate to Google Fonts and select Open Sans
- select the Regular 400 and Semi-bold 600 sizes
- view your selected families (currently this is an icon at the top right of the screen, but this might change!)
- click on Embed and copy the `<link>` code

If you see a continuation character `\` please ignore it (it's been added by the typesetting software). The entire `<link>` element should occupy a single line.

Now add a font-family declaration to the body rule in `css/style.css`:



## css/style.css

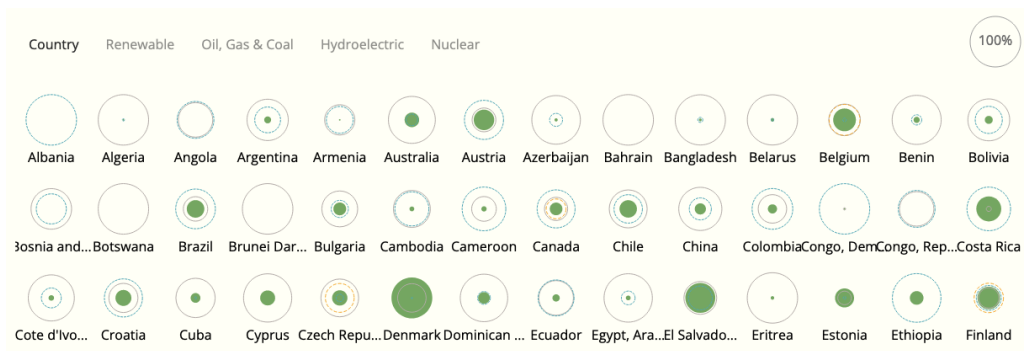
---

```
body {
  background-color: #FFFFFF7;
  font-family: 'Open Sans', sans-serif;
  cursor: default;
}
```

---

This will result in Open Sans being used throughout the visualisation. `sans-serif` is also added to the list of fonts as a fallback in case Open Sans doesn't load.

Save `index.html` and `css/style.css` and load `step15` in the browser. The font has changed to Open Sans:



Energy Explorer with Open Sans font

## 33.3 Style the menu

Before styling the menu add the following headings to bring some order to `style.css`:

## css/style.css

---

```
/* Global */
body { ... }

#wrapper { ... }

/* Controls */
#controls { ... }

/* Menu */
.menu .items { ... }
```

---

```

.menu .item.selected, .menu .item:hover { ... }

/* Legend */
.legend circle { ... }

.legend text { ... }

/* Circles (menu and chart) */
circle.renewable { ... }

circle.oilgascoal { ... }

circle.hydroelectric { ... }

circle.nuclear { ... }

/* Chart */
.country { ... }

.country .label { ... }

/* Popup */
.popup-center { ... }

.flourish-popup { ... }

```

---

We'll now style the menu so that'll it'll look like:

Sort by: Country ● Renewable ○ Oil, Gas & Coal ○ Hydroelectric ○ Nuclear

### Styled menu

We need to add a circle to each indicator and add a 'Sort by' label. In `js/menu.js` change how the menu items are created:

`js/menu.js`

---

```

let menuItems = [
  {
    id: 'country',
    label: 'Country'
  },
  {
    id: 'renewable',
    label: 'Renewable'
  },
  {

```

```

      id: 'oilgascoal',
      label: 'Oil, Gas & Coal'
    },
    {
      id: 'hydroelectric',
      label: 'Hydroelectric'
    },
    {
      id: 'nuclear',
      label: 'Nuclear'
    }
  ]
};

function getCircle(id) {
  let svg = '<svg width="18" height="18"><circle class="'
    + id + '" cx="9" cy="9" r="8"></svg>';
  return svg;
}

function getHtml(d) {
  let circle = d.id === 'country' ? '' : getCircle(d.id);
  let label = '<div class="label">' + d.label + '</div>';
  return circle + label;
}

function handleMenuClick(e, d) {
  action('setSelectedIndicator', d.id);
}

function updateMenu() {
  d3.select('#controls .menu .items')
    .selectAll('.item')
    .data(menuItems)
    .join('div')
    .classed('item', true)
    .classed('selected', function(d) {
      return state.selectedIndicator === d.id;
    })
    .text(function(d) {
      return d.label;
    })
    .html(getHtml)
    .on('click', handleMenuClick);
}

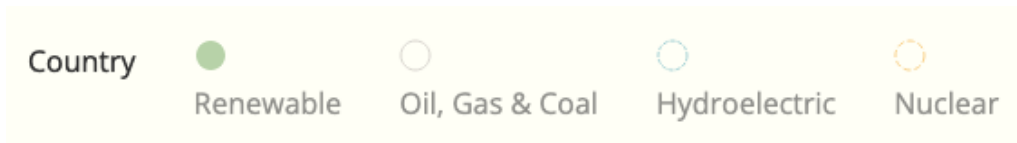
```

---

We use the `.html` method instead of `.text` in `updateMenu` to update the content of each

menu item using HTML instead of a text string. We pass in function `getHtml` which returns an HTML string consisting of an SVG circle and the label. Each circle is given the same class as its counterpart in the chart, so that it adopts the existing circle styles in `css/style.css`. (The ‘Country’ item in the menu doesn’t have a circle.)

Save `js/menu.js` and refresh the browser. The menu now looks like:



Menu with circles

In `index.html` add a Sort by: heading to the menu:

#### `index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>

  <body>
    <div id="wrapper">
      <div id="header">Energy mix by country (2015)</div>
      <div id="controls">
        <div class="menu">
          <div>Sort by:</div>
          <div class="items"></div>
        </div>
        <div class="legend">
          ...
        </div>
      </div>
      <div id="chart-wrapper">
        ...
      </div>
    </div>

    <script src="js/lib/d3.min.js"></script>
    ...
  </body>
</html>
```

---

Now make the following changes to the menu CSS:

## css/style.css

---

```
/* Menu */
.menu {
  font-size: 0.9rem;
  display: flex;
  align-items: center;
}

.menu .items {
  display: flex;
}

.menu .item {
  padding: 0 1rem;
  color: #333;
  opacity: 0.5;
  display: flex;
  align-items: center;
}

.menu .item .label {
  font-size: 1.2rem;
  padding-left: 0.2rem;
  opacity: 0.5;
  transition: opacity 0.3s;
}

.menu .item.selected, .menu .item:hover {
  .menu .item.selected .label, .menu .item:hover .label {
    opacity: 1;
  }
}

.legend circle {
  stroke: #aaa;
  fill: none;
}

...
```

---

We've made the following changes:

- Changed menu font size to 0.9rem.
- Set the menu's display to flex so that the 'Sort by:' label and menu items occupy the same row.

- Set the menu's `align-items` to `center` so that the label and menu are vertically aligned.
- Set each menu item's `display` to `flex` so that the circle and label occupy the same row.
- Set each menu item's `align-items` to `center` so that the each menu item's circle and label are vertically aligned.
- Set the font-size of each menu item to `1.2rem`.
- Give the menu labels an opacity of 0.5 (if not selected) or 1 (if selected).
- Add a subtle CSS transition to the label opacity.

Now the menu looks like:

Sort by: Country ● Renewable ○ Oil, Gas & Coal ○ Hydroelectric ○ Nuclear

### The styled menu

When an item is hovered, the label changes opacity but the circles always have an opacity of 1 (so that they look the same as the circles in the chart).

The menu is now complete. We added circles to the menu items indicating which energy type each circle style represents. There was quite a lot of detailed CSS to add. This is fairly typical when adding finishing touches to a data visualisation. It's often a significant portion of the work and sometimes requires a lot of experimentation and iteration.

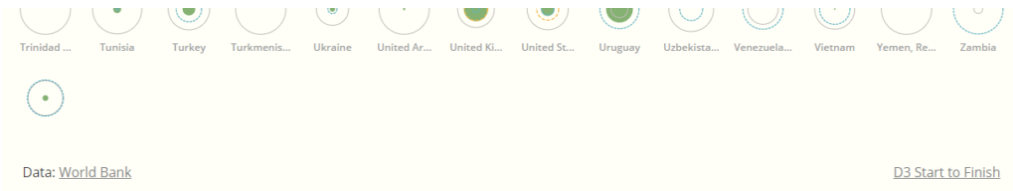
## 33.4 Add header and footer

The header will contain a main title **Energy mix by country (2015):**



### Main title

and the footer will contain information on the data source and attribution information:



Footer

In `index.html` make the following changes:

## index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>

  <body>
    <div id="wrapper">
      <div id="header">Energy mix by country (2015)</div>
      <div id="controls">
        ...
      </div>
      <div id="chart-wrapper">
        ...
      </div>
      <div id="footer">
        <div>Data: <a href="https://datacatalog.worldbank.org/dataset/world-development-i
indicators">World Bank</a></div>
        <div><a href="https://www.createwithdata.com">D3 Start to Finish</a></div>
      </div>
    </div>

    <script src="js/lib/d3.min.js"></script>
    ...
  </body>
</html>
```

We've added a `<div>` element for the header just before the controls. We've also added a `<div>` element for the footer.

Now add CSS for the header and footer:

## css/style.css

---

```
/* Global */
...

/* Header */
#header {
    font-size: 1.5rem;
    letter-spacing: -0.03rem;
    padding: 1rem;
    text-align: right;
}

/* Controls */
...

/* Menu */
...

/* Legend */
...

/* Circles (menu and chart) */
...

/* Chart */
...

/* Footer */
#footer {
    padding: 1rem 1rem 2rem 1rem;
    display: flex;
    justify-content: space-between;
}

/* Popup */
...

```

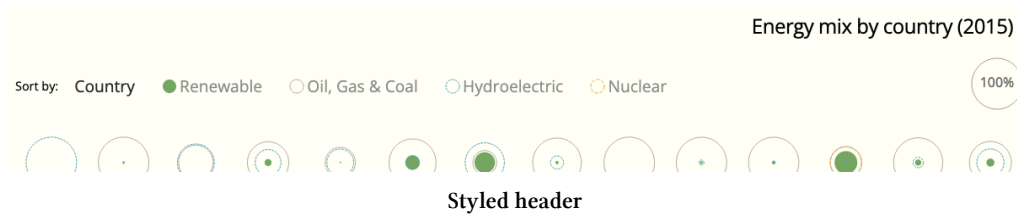
---

The header's font-size is set to 1.5rem. We close up the letter spacing a bit by setting letter-spacing to -0.03rem. We also added some padding around the header and aligned the text to the right.

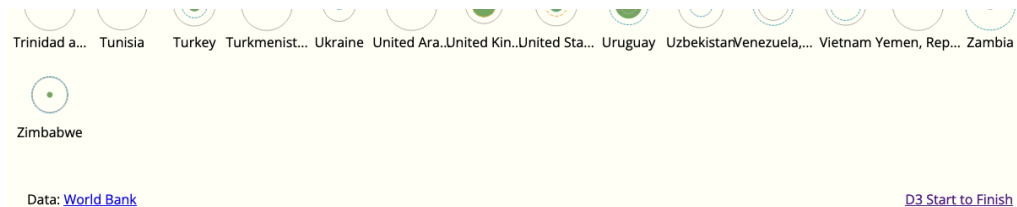
The footer's display is set to flex so that the items are arranged horizontally. We set justify-content to space-between so that the items fill the width of the container (similarly to what was done with the controls section). We also add some padding to the footer.



Save `index.html` and `style.css` and refresh the browser. The header now looks like:



and the footer looks like:



## 33.5 Other CSS tweaks

We're nearly finished! We've a few final CSS tweaks to add a final layer of polish to Energy Explorer.

In `css/style.css` make the following changes:

`css/style.css`

```
/* Global */
body {
  background-color: #FFFFFF;
  font-family: 'Open Sans', sans-serif;
  color: #333;
  margin: 0;
  cursor: default;
}

#wrapper {
  width: 1200px;
  margin: 0 auto;
}

a {
  color: #777;
  transition: opacity 0.3s;
```

```
}

a:hover {
  opacity: 0.5;
}

/* Header */
...

/* Controls */
#controls {
  padding: 0 1rem;
  display: flex;
  justify-content: space-between;
  align-items: center;
}

/* Menu */
...

/* Legend */
...

/* Circles (menu and chart) */
circle.renewable {
  fill: #7FB069;
}

circle.oilgascoal {
  fill: none;
  stroke: #BCB5B2;
}

circle.hydroelectric {
  fill: none;
  stroke: #5EB1BF;
  stroke-width: 1.25;
  stroke-dasharray: 3,1;
}

circle.nuclear {
  fill: none;
  stroke: #F6AE2D;
  stroke-width: 1.25;
}
```

```
        stroke-dasharray: 4,2;
    }

/* Chart */
.country {
    pointer-events: all;
}

.country .label {
    text-anchor: middle;
    font-size: 0.75rem;
    fill: #999;
    font-weight: 600;
    transition: fill 0.3s;
}

.country:hover .label {
    fill: #333;
}

/* Footer */
...

/* Popup */
.popup-center {
    opacity: 0;
}

.flourish-popup {
    pointer-events: none;
}

.flourish-popup-content {
    font-size: 0.8rem;
    color: #333;
}

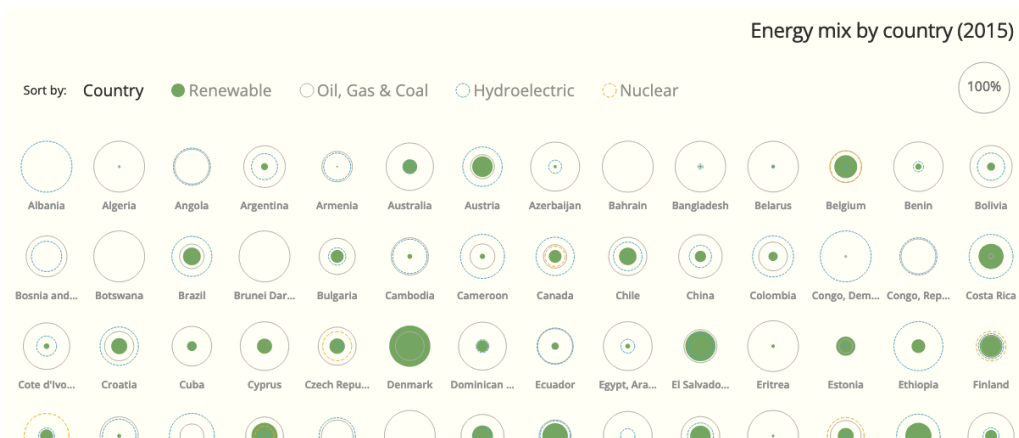
.flourish-popup-content h3 {
    text-align: center;
    margin: 0.25rem 0;
}
```

---

The following changes were made:

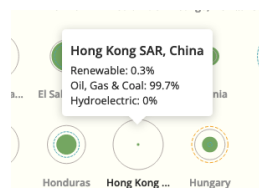
- the global text colour has been set to a dark gray (instead of black)
- the body element's margin has been set to 0 (to remove the default margins around the whole page)
- the links in the footer are coloured grey and a subtle hover over effect added
- the controls container padding has been tweaked to bring the menu and legend in a bit
- the stroke width of hydroelectric and nuclear circles has been increased slightly
- the country labels in the main chart have been styled, including a hover over effect
- set the font size, colour, alignment and margin in the popup

Save `css/style.css` and the Energy Explorer is now complete! Refresh the browser and it looks like:



The finished Energy Explorer

The popup has also been styled and looks like:



Styled popup

## 33.6 Summary

In this practical we made plenty of cosmetic changes to Energy Explorer, including: adding Open Sans font, adding a header and footer, styling the menu (including adding a

circle to each item) and many other subtle styling changes. Each change is quite a small one, but they add up to make a big difference. Prior to the changes, Energy Explorer was functional but not particularly polished, but now it looks finished and professional.

# 34. Wrapping Up

First of all, congratulations for getting this far. It's been a long journey and hopefully you've learned a lot along the way.

The aim of the book was to teach you the fundamentals of D3 and to show you how a real world, interactive data visualisation is built. Hopefully this book has succeeded in both.

Some of the things you learned were:

- what D3 is and what it's used for
- D3's approach to constructing data visualisations – how it's more of a building blocks approach rather than a collection of ready made charts
- the main functional aspects of D3, namely selections and it's large collection of modules
- in depth looks at selections and data joins
- data requests using D3
- D3's scale functions
- layout functions (to separate the geometric computations from the rendering code)
- different approaches to JavaScript modules
- joining arrays to groups of elements
- event handling with D3
- Flourish's popup library
- simple state management
- using lodash to manipulate data
- D3's transitions
- detailed styling with CSS

Along the way you built Energy Explorer. Some key aspects of the build were:

- using a layout function to perform the geometric computations. This makes the application easier to test and also would make it easier to switch to a different rendering library if necessary
- joining an array to a group of elements in order to place a country's circles in a single `<g>` element

- a significant amount of detailed styling, which hopefully has shown you the amount of CSS work that is often necessary to build a real world data visualisation
- adding an information popup to the visualisation
- handling state using something similar to the Flux pattern. This is a simple, but powerful pattern that's useful when building data visualisations (and other interactive applications)
- using D3 to create a menu
- sorting the countries using lodash
- adding transitions (animations) to the visualisation
- adding a legend, header and footer to the visualisation

Most (if not all) of what we've covered can apply to other visualisations and hopefully you've learned enough to help you build your own interactive data visualisation using D3.

## 34.1 Where next?

It's really up to you where you'd like to take the knowledge you've learned in this book. However, some ideas are:

- modify Energy Explorer. For example, you could try changing the color scheme, or other aspects of the look and feel. If you're feeling ambitious you might want to adapt it to a different data set!
- read the advanced edition of [D3 Start to Finish](#)<sup>80</sup> which includes chapters on data transformation, making Energy Explorer responsive, dark mode and ES2015 modules. If you haven't already bought it, email [info@createwithdata.com](mailto:info@createwithdata.com) for more information.
- try building your own data visualisation from scratch, using similar techniques to those you learned in this book. You could use the same data set, or one of your own
- share your experiences on social media (I'm [@createwithdata](#) and [@peter\\_r\\_cook](#) on Twitter) and let others know how you got on
- learn more about D3 through its homepage or other resources such as my own [D3 in Depth](#)<sup>81</sup>, or tutorials from [Nadieh Bremer](#)<sup>82</sup>, [Curran Kelleher](#)<sup>83</sup>, [Shirley Wu](#)<sup>84</sup> and [Amelia Wattenberger](#)<sup>85</sup>.

## 34.2 Wrapping up

Thank you so much for reading D3 Start to Finish, and for getting this far! I hope you found this book both useful and enjoyable. And I truly hope you've learned lots from it. I hope you've gained the confidence to work on other data visualisations written using D3 and perhaps even to build your own visualisation from scratch.

Stay in touch, either by email at [hello@createwithdata.com](mailto:hello@createwithdata.com) or via Twitter ([@createwithdata](https://twitter.com/createwithdata)).

All the best, and keep coding!

Peter



## Notes

80 <https://leanpub.com/d3-start-to-finish>

81 <https://www.d3indepth.com/>

82 <https://www.visualcinnamon.com/blog/>

83 <https://www.youtube.com/user/currankelleher>

84 <https://sxywu.com/>

85 <https://wattenberger.com/blog>

# Appendix A: Tools and Set-up

This book uses CodePen for many of the code examples. It's a reasonable tool for building small applications but larger applications are typically developed locally. This means using tools and files **on your computer** (rather than on the web). This appendix outlines a typical set up for developing web applications on your own computer. This is similar to the set-up most web developers use. The first section gives an overview of the necessary tools and the second section walks you through setting up these tools.

This content originally appeared in my [Fundamentals of HTML, SVG, CSS and JavaScript for Data Visualisation<sup>86</sup>](#) book.

## Web development tools

The tools necessary for web development are:

- a **code editor**
- a **web browser**
- a **web server** (not so important if you're not loading data)

You could also add a **terminal emulator** to this list. A terminal emulator allows you to interact with your operating system via text commands. For example Terminal (on MacOS) and the Command shell (on Windows). You can work through all the material in this book without a terminal emulator so I won't cover them in this book. If you'd like to learn more I suggest a web search for 'command line tutorial'.

## Code editor

A code editor is a desktop application that allows you to edit HTML, CSS and JavaScript files. You can use simple applications such as Notepad (on Windows) and TextEdit (on Mac). It's important that these editors work in plain text mode so that they don't insert extra characters. However I recommend using a dedicated code editor such as [Brackets<sup>87</sup>](#) or [Visual Studio Code<sup>88</sup>](#). Both of these are free and multi-platform. Advantages of using a dedicated code editor include:

- you can treat your application as a project making it easier to open and navigate

- syntax highlighting (code is coloured according to whether its a keyword, a variable etc.)
- numerous plug-ins (including web servers!) that make your life as a coder easier



Adobe dropped support for Brackets but at the time of writing it's been resurrected by the open source community. It does seem to be in a state of flux so keep checking its status if you're interested.

## Web browser

A modern web browser such as Chrome, Firefox, Safari or Edge is recommended for web development. Create With Data books assume that **Chrome** is being used. The majority of material will apply regardless of your browser choice but there may be differences if the debugging tools are used, as these differ from browser to browser.

## Web server

A web server typically returns (or 'serves') HTML, CSS and JavaScript files in response to requests from a web browser. For example if you type `creativewithdata.com` into a web browser the browser connects with a web server and requests files that make up that website. The web server responds by sending the requested files back to the browser.

When you create a web site on your own computer you don't necessarily need your own web server running. (You can open an HTML file directly in most web browsers.) However if you need to make requests from within your web application (as is usually the case with web based data visualisations) you will need your own local web server. (This is due to security restrictions in web browsers.) If you're an experienced developer and are comfortable setting up a local web server, feel free to skip ahead.

The options for running a local web server (starting at the easiest) include:

- Brackets editor built-in web server
- Visual Studio Code plug-ins
- Python, Ruby, PHP or Node servers

Another option which I've not personally explored is [Servez](#)<sup>89</sup> which is a simple GUI based web server.



Personally I use a Node server (such as [http-server<sup>90</sup>](#)) and I recommend this if you get serious about web development. However if you're starting out Brackets is a great choice.

## Brackets web server

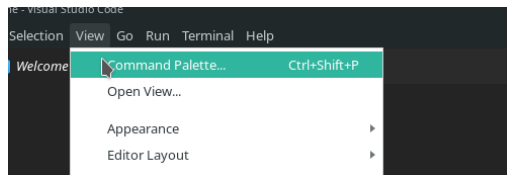
[Brackets<sup>91</sup>](#) has a built in web server so this is by far the easiest option. This [video<sup>92</sup>](#) gives a good demonstration. If you're new to code editors I recommend using Brackets.

## VS Code Live Server

Visual Studio Code has a web server plug-in called [Live Server<sup>93</sup>](#).

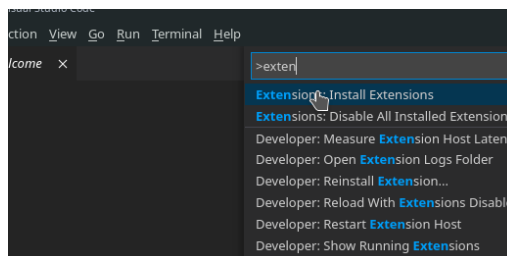
To install Live Server:

- select 'View' and then 'Command Palette...' from the menu bar



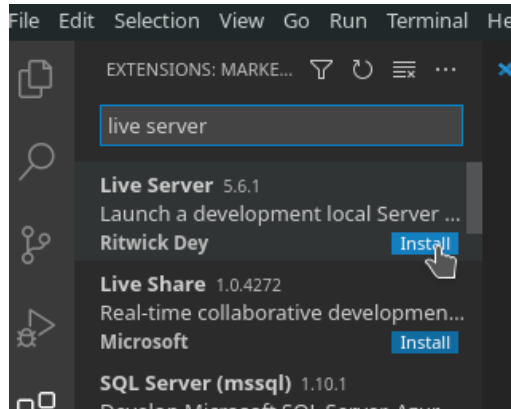
Open Command Palette in VS Code

- type 'extension' and select 'Extensions: Install Extensions'



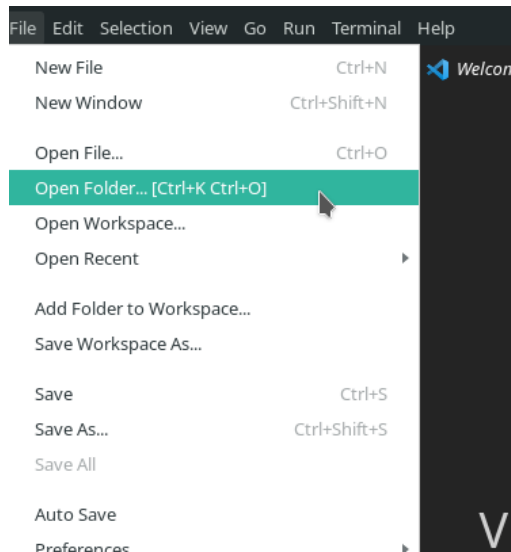
Select Extensions: Install Extensions

- search for 'live server' (by Ritwick Dey) in the extensions search box and click Install



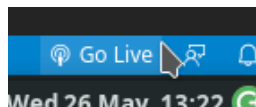
Search for 'live server'

- open a directory containing your code



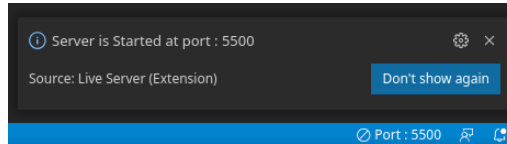
Open a directory cotaining your code

- click the Go Live button in the bottom bar



Click the Go Live button

Your web browser should then open your web application. If it doesn't, open a web browser and go to `localhost:5500`. (If a different port number to 5500 is shown in the 'Server is Started' message, use that number instead.) More detailed instructions are [here](#)<sup>94</sup>.



Server has started

## Python, Ruby, PHP, Node

If you're a Python, Ruby or PHP developer you can use the language's built in webserver. There are some useful instructions [here](#)<sup>95</sup>.

(MacOS includes Python and Ruby so you can use one of these if you're a Mac user.)

If you're a JavaScript developer using Node you can use [http-server](#)<sup>96</sup>.

## Notes

86 <https://leanpub.com/html-svg-css-js-for-data-visualisation>

87 <http://brackets.io/>

88 <https://code.visualstudio.com/>

89 <https://greggman.github.io/servez/>

90 <https://github.com/http-party/http-server><https://github.com/http-party/http-server>

91 <http://brackets.io/>

92 <https://youtu.be/KJXdvaY9lTA?t=122>

93 <https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

94 <https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

95 <https://gist.github.com/willurd/5720255>

96 <https://www.npmjs.com/package/http-server>

# Appendix B: Example Set-Up

This appendix takes you through setting up a web development environment and the creation of a simple website (using HTML, CSS and JavaScript).

The steps are:

- **create a directory** for your project
- choose and install a **code editor**
- create a simple **HTML** file
- get a local **web server** up and running
- add **CSS** and **JavaScript** files

We'll use VS Code and its Live Server extension. If you're comfortable using a different editor and web server then please feel free to do so.

## Create a project directory

Using File Explorer (Windows) or Finder (Mac) create a directory (or folder) that'll contain your project. Name it `my_website` (or something similar). You might want to place it in another directory named `coding_projects` (or similar) if you plan on having more than one project.

## Install a code editor

1. Download VS Code from [here](#)<sup>97</sup> and install
2. Run VS Code

## Create a minimal HTML file

In this section you'll create a simple HTML file. In VS Code select **New** from the **File** menu. VS Code creates a new (unsaved and untitled) file.

The following code is just about the shortest and valid HTML document possible. It serves as a good starting point to a new web page. Copy it and paste into your new document.

**index.html**

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My webpage</title>
  </head>

  <body>
    <h1>Hello! </h1>
  </body>
</html>
```

Usually the only code you'll need to change is the title (between the `<title>` tags and the content (between the `<body>`) tags.



The `<!DOCTYPE>`, `lang="en"` and `charset="utf-8"` are standard bits of code that tell the browser how to interpret your HTML file. Although your webpage may appear to work without these it's best to leave them in. (See [here](#)<sup>98</sup> for more information about character encoding if you're interested.)

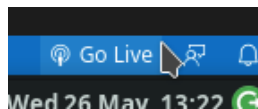
Select **Save** from the **File** menu, navigate to your project directory, name your file **index.html** and click Save. (It's important to name your file **index.html**.) Once your file is saved you should see **index.html** appear in the sidebar.

## Install Live Server extension

Follow the instructions in the VS Code Live Server section of the Web development tools chapter to install the Live Server extension.

## Start Server

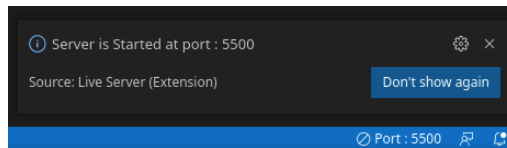
If you haven't already, select **Open Folder...** from the **File** menu and select your my\_-website directory. You should see your new directory appear in the sidebar. Now click on the Go Live button that appears in the bottom bar.



Click the Go Live button in VS Code



Make sure your browser is pointing at 'localhost:5500' (use whichever port number is displayed in the bottom bar).



Server has started message (with active port number displayed)

Your page should now be visible in the web browser (it should say 'Hello!'). Try editing your HTML file, saving it, then refreshing your browser. The changes you've made should be visible.

## Add CSS and JavaScript files

In this section you'll add a CSS file and a JavaScript file to your web page. There are usually two steps to adding a file to a web page:

- create the file and save it (with extension `.css` for CSS files or `.js` for JavaScript files)
- include the file in your `index.html` file

### Add a CSS file

Select **New** from the **File** menu and add the following CSS code to the new file:

style.css

```
h1 {  
  color: orange;  
}
```

Select **Save** from the **File** menu and when the dialog appears create a new directory called `css` within your project directory. Go into the `css` directory then save your new file as `style.css`. You're aiming for the following directory structure:

```
my_website
  css
    style.css
  index.html
```

You now need link to your CSS file in `index.html`. Start editing your HTML file by clicking `index.html` in the left sidebar. Add the line with the `<link>` tags (highlighted below):

#### `index.html`

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My webpage</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <h1>Hello there!</h1>
  </body>
</html>
```

---

Save `index.html` and check your browser (you might need to refresh the page). The title should now be **orange**.

## Add a JavaScript file

Select **New** from the **File** menu and add the following JavaScript code to the new file:

#### `main.js`

---

```
alert("Hello!");
```

---

Select **Save** from the **File** menu and when the dialog appears create a new directory within your project directory called `src`. Go into the `src` directory then save your new file as `main.js`. You're aiming for the following directory structure:

```
my_website
  css
    style.css
  src
    main.js
  index.html
```



If your files aren't in the above structure you can drag them into their correct location in the left sidebar of VS Code. You can also right click files and directories for more options.

You now need to add a link to your JavaScript file in `index.html`. Start editing your HTML file by clicking `index.html` in the left sidebar. Add the line containing the `<script>` tag (highlighted below):

#### **index.html**

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My webpage</title>
    <link rel="stylesheet" href="css/style.css">
  </head>

  <body>
    <h1>Hello there!</h1>
    <script src="src/main.js"></script>
  </body>
</html>
```

---

Save `index.html` and check your browser (you might need to refresh the page). An alert should appear saying 'Hello!'.

## Summary

This section has shown how to set up a web development environment consisting of:

- a code editor
- a web browser
- a local web server

You saw how to create a simple web site consisting of an HTML file, a CSS file and a JavaScript file. This is the basis for most web applications including data visualisations.

## Notes

97 <https://code.visualstudio.com/>

98 <https://www.w3.org/International/questions/qa-what-is-encoding>