data-map-d3 Documentation

Release 1.0

Lukas Vonlanthen

January 25, 2016

Contents

Welcome to the documentation of the data-map-d3 tutorial. This documentation covers the steps taken to create a simple data map using D3.js.

The final result can be seen here: http://lvonlanthen.github.io/data-map-d3

The code is available here: https://github.com/lvonlanthen/data-map-d3

This tutorial was created for the webmap workshop being held on January 20-21 2016 at the University of Bern.

Note: Unfortunately, I was not able to add detailed comments for all the steps yet. The code is there, but additional explanations are sometimes missing. I hope to finde time soon to complete this.

Contents:

CHAPTER 1

Step 1: Basic document structure

In this step, we will create the basic HTML document which will hold our map.

Contents

1.1 Basic structure of the HTML document

Every HTML document consists HTML tags, the most basic ones is `<html>`. Within the HTML document, the sections `<head>` and `<body>` are found. Every tag has a closing matching closing tag (`<html></html>`).

See also:

For a basic introduction to HTML, refer to the W3Schools HTML Introduction.

The following is a very basic structure of a HTML document. Create a directory where you want to work in and create a new document called `index.html` and add the following content. Open the file in a webbrowser and you should see your HTML document. The title of the document was set and "Content" is visible.

Listing 1.1: index.html - very basic HTML structure

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Land use statistics map</title>
  </head>
  <body>
    Content
  </body>
</html>
```

1.2 Container for the map

The map will be placed in a container so we need to create this container first. The `<div>` tag defindes a division or a section in an HTML document, so we create one with a unique ID to later have easy access to it. While we are at it, we also create a heading using the `<h3>` tag.

Listing 1.2: index.html - with a map container

```html
<!-- ... -->
  <body>
    <h3>Land use statistics map</h3>
    <div id="map"><!-- Map container --></div>
  </body>
<!-- ... -->
```

The `<div>` element for the map is empty, as the map will be generated dynamically using JavaScript. If we refresh the browser, we see the heading but no map division, as there is no content in it.

1.3 Some very basic styling

Even though there is no content in the `<div>` element for the map, we can style it to appear in the browser. For the styling we use CSS. Within the `<head>` section, we could define a `<style>` element and define styles within this element.

However, it is better to cleanly separate the HTML from the CSS and create separate and pure CSS file. We can therefore create a new file called `style.css` and define the styles there.

We give the map container a predefined width and height to make it visible. We also give it a visible border and a background color. If we refresh the browser page, we can now see the empty container for the map.

Listing 1.3: style.css - first CSS styles

```css
body {
  margin: 25px;
}

#map {
  width: 900px;
  height: 600px;
  border: 1px solid silver;
  background: #E6E6E6;
}
```

We also added a small margin to the `<body>` section to give it some space, which looks a bit nicer.

See also:

For a basic introduction to CSS, refer to the W3Schools CSS Introduction.

We can now link the CSS file in our HTML documents so the changes are picked up.

Listing 1.4: index.html - link CSS file

```html
<!-- ... -->
  <head>
    <meta charset="UTF-8">
    <title>Land use statistics map</title>

    <!-- Custom CSS styles -->
    <link href="style.css" rel="stylesheet" type="text/css">
  </head>
<!-- ... -->
```

1.4 Load D3.js library

Since the map will be created with D3.js, we need to load the D3 library. This is a JavaScript file we need to include in our document. Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page. However, it is good practice to place scripts at the bottom of the `<body>` element to improve page load.

You can either download the D3.js library and link it locally, or you can directly link the latest release of the library. On http://d3js.org/, they provide the following link:

```
<script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
```

Hint: Please note that if you are working on your local machine without a web server running, you may need to add `http:` to the script URL in order to make it work on your machine. This is what will be used throughout this tutorial.

Listing 1.5: index.html - load the D3.js library

```
<!-- ... -->
  <body>
    <h3>Land use statistics map</h3>
    <div id="map"><!-- Map container --></div>

    <!-- JS libraries -->
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
  </body>
<!-- ... -->
```

1.5 JavaScript debugging: console.log()

The map will be created using JavaScript code that we will write shortly. We can already prepare a JavaScript document, because as with the CSS style sheet, we are going to create a separate file for the JavaScript. We create a file called `map.js` and for test purposes, we are just going to output some text in the console.

Listing 1.6: map.js - first log to console

```
// This is where the map will be coded.
console.log("Ready to create the map!");
```

We link the JavaScript file at the bottom of the `<body>` section.

Listing 1.7: index.html - log stuff to the console

```
    <!-- ... -->

    <!-- JS libraries -->
```

```html
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>

    <!-- Custom JS code -->
    <script src="map.js"></script>
  </body>
<!-- ... -->
```

To see the browser console, you need to open the developer tools by hitting `F12` and selecting “Console” in the menu. If you are using Firefox, it is recommended to install the Firebug extension.

See also:

For more information on the console and the developer tools, visit W3Schools JavaScript Debugging.

`console.log()` is very useful for debugging. You can not only output text, but also variables and data. In your browser, you can use the console.log() method to display data.

However, since not all browsers support the console, you should not have console logs in production.

Next

Proceed to *Step 2: First map*.

1.6 Code

- **For reference, the file `index.html` after step 1:** https://github.com/lvonlanthen/data-map-d3/blob/step-01/index.html
- **For reference, the file `style.css` after step 1:** https://github.com/lvonlanthen/data-map-d3/blob/step-01/style.css
- **For reference, the file `map.js` after step 1:** https://github.com/lvonlanthen/data-map-d3/blob/step-01/map.js

Step 2: First map

In this step, we will prepare the geographic data and display it on the map for the first time.

Contents

2.1 Preparation of geographic data

For this tutorial, we use data for Switzerland. The geographical base units are the Municipalities of Switzerland. The boundaries of these municipalities can be downloaded as Shapefiles. The preparation consists of using QGIS to remove unccessary attributes and save the geometries as GeoJSON.

Hint: If you want to skip the data preparation steps, you can create a new folder `data` in your working directory, download the GeoJSON file (right-click, save as ...) from the repository and save it to `data/ch_municipalities.geojson`. You can then proceed to the section *Show features on map*.

2.1.1 Download

Download the data from GEOSTAT of the Swiss Federal Statistical Office. Make sure to download the file named "Generalisierte Gemeindegrenzen: Geodaten" (gd-b-00.03-877-gg14) for the year **2014**.

Once the file was downloaded, extract the ZIP file.

2.1.2 Data structure

The zip file contains several files and folders. The folder `ggg_2014` contains the generalized municipality boundaries in various formats. We will work with Shapefiles (`shp`).

For each format, there are different files such as `g1b14.shp` or `g2g14.shp`.

- The first two characters indicate the level of cartographic generalization, where `g2` is more generalized (meaning less details and smaller files) than `g1`.
- The next letter indicates the type of geometry:
  - `g` (Gemeinden): Municipalities or Communes
  - `b` (Bezirke): Districts
  - `k` (Kantone): Cantons
  - `r` (Grossregionen): Regions
  - `l` (Landesgrenzen): Country

More information on the files can be found in the metadata file (available only in german or french).

For this tutorial, we will use the boundaries of municipalities. As the very detailed boundaries are not that important, we can use the `g2` generalization level. Therefore, we will work with the file `ggg_2014/shp/g2g14.shp`.

2.1.3 QGIS

Hint: We will work with the open source GIS software QGIS. If you do not have a GIS software installed, it is recommended that you download and install QGIS. If you already have another GIS software, you can work with the one you are more familiar with. However, the following steps are written for QGIS.

Open the file `ggg_2014/shp/g2g14.shp` in QGIS.

2.1.4 Remove unnecessary attributes

Since we will only need the geometries of the municipalities, we can delete columns of the Shapefile which are not needed. This will help to reduce the file size of the resulting GeoJSON.

There is a plugin called “Table Manager Plugin” which can be installed and used to delete the unnecessary columns. You can delete all columns except `GMDNR` (the ID of the municipality) and `GMDNAME` (the name of the municipality).

It is recommended that you save the resulting Shapefile under a new file name to not modify the original file. QGIS will automatically add the new Shapefile to the map.

2.1.5 Export as GeoJSON

To export the Shapefile as GeoJSON make sure the correct file (the one without unnecessary attributes) is selected in QGIS.

Use right-click on the file and select “Save as” which opens a dialog box and you can set the following parameters:

- Format: GeoJSON
- Save as: [working_directory]/data/ch_municipalities.geojson

  Navigate to your working directory (where you placed your index.html file), create a new folder “data” and save the file as “ch_municipalities.geojson”.
- CRS: “Selected CRS” and browse to select “WGS 84” (you can filter by “EPSG:4326”)
- Additionally, set the following parameters:

- Encoding: UTF-8
- Layer Options > COORDINATE_PRECISION: 3

  This will help to reduce the file size

Then click OK to export the Shapefile as GeoJSON.

2.2 Show features on map

Now we would like to display the GeoJSON on the map.

Listing 2.1: map.js - a first map

```
// We specify the dimensions for the map container. We use the same
// width and height as specified in the CSS above.
var width = 900,
    height = 600;

// We create a SVG element in the map container and give it some
// dimensions.
var svg = d3.select('#map').append('svg')
  .attr('width', width)
  .attr('height', height);

// We define a geographical projection
//     https://github.com/mbostock/d3/wiki/Geo-Projections
// and set the initial zoom to show the features.
var projection = d3.geo.mercator()
  // The approximate scale factor was found through try and error
  .scale(10000)
  // The geographical center of Switzerland is around 46.8°, 8.2°
  //     https://de.wikipedia.org/wiki/Älggi-Alp
  .center([8.226692, 46.80121])
  // Translate: Translate it to fit the container
  .translate([width/2, height/2]);

// We prepare a path object and apply the projection to it.
var path = d3.geo.path()
  .projection(projection);

// Load the features from the GeoJSON.
d3.json('data/ch_municipalities.geojson', function(error, features) {

  // We add a <g> element to the SVG element and give it a class to
  // style it later.
  svg.append('g')
      .attr('class', 'features')
    // D3 wants us to select the (non-existing) path objects first ...
    .selectAll('path')
      // ... and then enter the data. For each feature, a <path> element
      // is added.
      .data(features.features)
    .enter().append('path')
      // As "d" attribute, we set the path of the feature.
      .attr('d', path);

});
```

Hint: If you are using Chrome, you may not see the map. Instead, you may see an error message in the console saying that the GeoJSON file could not be loaded because cross origin requests are not supported. In this case, you either have to run a web server locally or use a different browser (eg. Firefox).

Next

Proceed to *Step 3: Dynamic map extent*.

2.3 Code

- **For reference, the file `index.html` after step 2:** https://github.com/lvonlanthen/data-map-d3/blob/step-02/index.html
- **For reference, the file `style.css` after step 2:** https://github.com/lvonlanthen/data-map-d3/blob/step-02/style.css
- **For reference, the file `map.js` after step 2:** https://github.com/lvonlanthen/data-map-d3/blob/step-02/map.js
- **The diff view of step 1 and step 2:** https://github.com/lvonlanthen/data-map-d3/compare/step-01...step-02?diff=split

CHAPTER 3

Step 3: Dynamic map extent

In this step, we will make the initial map extent dynamic.

Contents

3.1 Get extent from data

In the previous step, the initial extent was set based on hard-coded parameters which were found by try-and-error.

Listing 3.1: map.js - static initial extent

```
// ...

// We define a geographical projection
//     https://github.com/mbostock/d3/wiki/Geo-Projections
// and set the initial zoom to show the features.
var projection = d3.geo.mercator()
  // The approximate scale factor was found through try and error
  .scale(10000)
  // The geographical center of Switzerland is around 46.8°, 8.2°
  //     https://de.wikipedia.org/wiki/Älggi-Alp
  .center([8.226692, 46.80121])
  // Translate: Translate it to fit the container
  .translate([width/2, height/2]);

// ...
```

These parameters (`scale` and `center`) happen to work quite well for Switzerland, but they are not very flexible and would not work at all if the geographic data were to change.

In order to make the extent more dynamic, we cannot set the `scale` and `extent` before we read the GeoJSON and determine the extent from the data. Instead, we simply create an initial dummy scale which will be overwritten later.

Listing 3.2: map.js - set dummy initial scale

```javascript
// ...

// We define a geographical projection
//     https://github.com/mbostock/d3/wiki/Geo-Projections
// and set some dummy initial scale. The correct scale, center and
// translate parameters will be set once the features are loaded.
var projection = d3.geo.mercator()
  .scale(1);

// ...
```

To determine the real extent based on the features, we can make use of a function to calculate the `scale` and the `center` parameters for a given set of Features. We put the function at the bottom of our JavaScript block.

Listing 3.3: map.js - function to calculate extent

```javascript
// ...

/**
 * Calculate the scale factor and the center coordinates of a GeoJSON
 * FeatureCollection. For the calculation, the height and width of the
 * map container is needed.
 *
 * Thanks to: http://stackoverflow.com/a/17067379/841644
 *
 * @param {object} features - A GeoJSON FeatureCollection object
 *   containing a list of features.
 *
 * @return {object} An object containing the following attributes:
 *   - scale: The calculated scale factor.
 *   - center: A list of two coordinates marking the center.
 */
function calculateScaleCenter(features) {
  // Get the bounding box of the paths (in pixels!) and calculate a
  // scale factor based on the size of the bounding box and the map
  // size.
  var bbox_path = path.bounds(features),
      scale = 0.95 / Math.max(
        (bbox_path[1][0] - bbox_path[0][0]) / width,
        (bbox_path[1][1] - bbox_path[0][1]) / height
      );

  // Get the bounding box of the features (in map units!) and use it
  // to calculate the center of the features.
  var bbox_feature = d3.geo.bounds(features),
      center = [
        (bbox_feature[1][0] + bbox_feature[0][0]) / 2,
        (bbox_feature[1][1] + bbox_feature[0][1]) / 2];

  return {
    'scale': scale,
    'center': center
  };
}
```

Now, we can call this function when we read the GeoJSON file to get the optimal `scale` and `center` for the current geographic data.

Listing 3.4: index.html - set optimal scale and center

```
// ...
d3.json('data/ch_municipalities.geojson', function(error, features) {

  // Get the scale and center parameters from the features.
  var scaleCenter = calculateScaleCenter(features);

  // Apply scale, center and translate parameters.
  projection.scale(scaleCenter.scale)
    .center(scaleCenter.center)
    .translate([width/2, height/2]);

  // We add a <g> element to the SVG element and give it a class to
  // ...
```

If we refresh the browser, we should not see a very big difference, as the manual parameters were already rather good. However, our code is now much more dynamic and not only works for Switzerland.

3.2 Verification

To verify that our extent also works in other geographic contexts, we can test by loading a different GeoJSON file.

- You can download a GeoJSON file of Germany from Click that 'hood!.
- Save it to the data folder (next to `ch_municipalities.geojson`).
- In the code, replace the path to the GeoJSON file: `data/germany.geojson`.
- Refresh the browser, the map should now show Germany and its states, everything nicely scaled and centered.
- Don't forget to change the path to the GeoJSON file back to `data/ch_municipalities.geojson`.

Next

Proceed to *Step 4: Add some colors*.

3.3 Code

- **For reference, the file `index.html` after step 3:** https://github.com/lvonlanthen/data-map-d3/blob/step-03/index.html
- **For reference, the file `style.css` after step 3:** https://github.com/lvonlanthen/data-map-d3/blob/step-03/style.css
- **For reference, the file `map.js` after step 3:** https://github.com/lvonlanthen/data-map-d3/blob/step-03/map.js
- **The diff view of step 2 and step 3:** https://github.com/lvonlanthen/data-map-d3/compare/step-02...step-03?diff=split

CHAPTER 4

Step 4: Add some colors

In this step, we will prepare and include statistical data and use it to color the municipalities on the map.

Contents

4.1 Preparation of statistical data

As statistical data, we will use data on municipalities provided by the Swiss Federal Statistical Office. You can download the Excel file and open it, for example using LibreCalc.

Hint: If you want to skip the data preparation steps, you can download the CSV file (right-click, save as ...) from the repository and save it to `data/areastatistics.csv`. You can then proceed to the section *Select some colors*.

Unfortunately, the file is not ready to use as such because the file has some metadata and merged column labels at the top. We want it to start right with one row of data labels followed by the data rows.

Also, the file contains a lot of columns which we will not use. In order to keep file size minimal, we are going to delete all unnecessary columns from the file.

For our tutorial, we only need the following columns:

- Gemeindecode
- Gemeindename
- Gesamtfläche in km²
- Siedlungsfläche in %
- Landwirtschaftsfläche in %
- Wald und Gehölze in %
- Unproduktive Fläche in %

This means you can do the following actions in the Excel file:

- Delete columns W to AU
- Delete columns R and T
- Delete columns C to O
- We use row 9 as our new header to label the columns:
  - `id`
  - `name`
  - `area`
  - `urban`
  - `agriculture`
  - `forest`
  - `unproductive`
- Delete rows 1 to 8
- Delete rows 2354 to 2367

The table should now look like this:

id	name	area	urban	agriculture	forest	unproductive
1	Aeugst am Albis	7.9	12.6	51.3	30.9	5.2
2	Affoltern am Albis	10.6	30.8	40.1	28.2	0.9
...	...	...	...	...	...	...

We can now export it as CSV document and save it to our data folder as `data/areastatistics.csv`.

When saving it, make sure to use the following settings:

- Character set: Unicode (UTF-8)
- Field delimiter: ,
- Text delimiter: “”

4.2 Select some colors

Next we are going so select some colors for our map. For this, we are using ColorBrewer, which is great at giving color advice for maps.

We choose 9 data classes and select a color scheme that appeals to us, for example YlGnBu. We then click on “Export” and copy the CSS classes for this scheme.

These CSS classes are now pasted in the CSS style section of our HTML document.

Listing 4.1: style.css - color classes from ColorBrewer

```css
/* ... */

/* Thanks to http://colorbrewer2.org/ */
.YlGnBu .q0-9{fill:rgb(255,255,217)}
.YlGnBu .q1-9{fill:rgb(237,248,177)}
.YlGnBu .q2-9{fill:rgb(199,233,180)}
.YlGnBu .q3-9{fill:rgb(127,205,187)}
.YlGnBu .q4-9{fill:rgb(65,182,196)}
```

```
.YlGnBu .q5-9{fill:rgb(29,145,192)}
.YlGnBu .q6-9{fill:rgb(34,94,168)}
.YlGnBu .q7-9{fill:rgb(37,52,148)}
.YlGnBu .q8-9{fill:rgb(8,29,88)}
```

4.3 Color the municipalities by attribute

We have selected 9 classes from ColorBrewer so we want to split the attribute values into 9 categories. Each feature then receives a CSS class based on the category of its value and the CSS class will determine its color on the map.

To achieve the categorization, we create a quantize scale with D3. It creates a number of classes within a given domain and we use it to return the class name used by ColorBrewer.

At the same time, we prepare a D3 mapping object called `dataById` which will later permit easy access to the data we are about to read. More on that later.

Listing 4.2: map.js - object to access data and quantize scale

```
// ...

// We prepare an object to later have easier access to the data.
var dataById = d3.map();

// We create a quantize scale to categorize the values in 9 groups.
// The domain is static and has a maximum of 100 (based on the
// assumption that no value can be larger than 100%).
// The scale returns text values which can be used for the color CSS
// classes (q0-9, q1-9 ... q8-9)
var quantize = d3.scale.quantize()
    .domain([0, 100])
    .range(d3.range(9).map(function(i) { return 'q' + i + '-9'; }));

// ...
```

Since the class of the feature depends on the category of the attribute, we need to wait until the CSV is loaded until we can draw the features on the map.

We will load the CSV (using `d3.csv()`) inside the function called after loading the GeoJSON. The drawing of the features will now happen inside the function called after the CSV is available, meaning that we replace the existing function and move it inside the `d3.csv()` function.

Listing 4.3: map.js - load csv and draw the features

```
// ...

  // Read the data for the cartogram
  d3.csv('data/areastatistics.csv', function(data) {

    // This maps the data of the CSV so it can be easily accessed by
    // the ID of the municipality, for example: dataById[2196]
    dataById = d3.nest()
      .key(function(d) { return d.id; })
      .rollup(function(d) { return d[0]; })
      .map(data);

    // We add a <g> element to the SVG element and give it a class to
    // style it later. We also add a class name for Colorbrewer.
    svg.append('g')
        .attr('class', 'features YlGnBu')
      // D3 wants us to select the (non-existing) path objects first ...
      .selectAll('path')
        // ... and then enter the data. For each feature, a <path>
        // element is added.
        .data(features.features)
      .enter().append('path')
        .attr('class', function(d) {
          // Use the quantized value for the class
          return quantize(dataById[d.properties.GMDNR].urban);
        })
        // As "d" attribute, we set the path of the feature.
        .attr('d', path);

  });

// ...
```

4.4 Play around

You can try and change the key of the map which is visualized on the map. You can do this by changing the key accessed by the `dataById` object called inside the `quantize()` function.

Listing 4.4: map.js - change map key

```
        // ...

        .attr('class', function(d) {
          // Use the quantized value for the class
          return quantize(dataById[d.properties.GMDNR].urban);
        })

        // ...
```

Try to set it to `agriculture`, `forest` or `unproductive` and refresh the map.

Next

Proceed to *Step 5: Dynamic color domain*.

4.5 Code

- **For reference, the file `index.html` after step 4:** https://github.com/lvonlanthen/data-map-d3/blob/step-04/index.html
- **For reference, the file `style.css` after step 4:** https://github.com/lvonlanthen/data-map-d3/blob/step-04/style.css
- **For reference, the file `map.js` after step 4:** https://github.com/lvonlanthen/data-map-d3/blob/step-04/map.js

- **The diff view of step 3 and step 4:** https://github.com/lvonlanthen/data-map-d3/compare/step-03...step-04?diff=split

CHAPTER 5

Step 5: Dynamic color domain

In this step, we will use a dynamic domain for quantize scale.

Contents

In the previous step, we used a static domain for the quantize scale.

Listing 5.1: map.js - hard coded quantize scale domain

```
// ...

// We create a quantize scale to categorize the values in 9 groups.
// The domain is static and has a maximum of 100 (based on the
// assumption that no value can be larger than 100%).
// The scale returns text values which can be used for the color CSS
// classes (q0-9, q1-9 ... q8-9)
var quantize = d3.scale.quantize()
    .domain([0, 100])
    .range(d3.range(9).map(function(i) { return 'q' + i + '-9'; }));

// ...
```

The hard coded domain ranging from 0 to 100 works surprisingly well for our data because there is a wide range of percentage values with very low and very high values.

However, as we would change the data and move away from percentage data this would not work with values greater than 100. Also, if we had only high percentage values (eg. all values > 50%), the domain of the quantize scale would still go from 0 to 100, even though there were no values for these categories.

It is therefore much better to use a domain which is based on the actual values of the data.

5.1 Store current key in single variable

As we have seen before, it is possible to change the key of the map and we will do this later using a dropdown. In order to make it easier to change the key and access the values attributed to it, we will define a variable holding the current key and we will create a function returning the value belonging to it.

Listing 5.2: map.js - variable to hold the map key

```javascript
// We define a variable holding the current key to visualize on the map.
var currentKey = 'urban';

// ...
```

We place this at the end of the JavaScript block.

Listing 5.3: map.js - function to access value of data

```javascript
// ...

/**
 * Helper function to access the (current) value of a data object.
 *
 * Use "+" to convert text values to numbers.
 *
 * @param {object} d - A data object representing an entry (one line) of
 * the data CSV.
 */
function getValueOfData(d) {
  return +d[currentKey];
}
```

And we can use this function instead of manually accessing key `urban` in the quantize function

Listing 5.4: map.js - use value function in quantize

```javascript
      // ...
        .attr('class', function(d) {
          // Use the quantized value for the class
          return quantize(getValueOfData(dataById[d.properties.GMDNR]));
        })
        // ...
```

5.2 Domain calculation

Replace the old definition of the quantize with the following (the domain will be calculated later):

Listing 5.5: map.js - quantize without domain

```javascript
// ...

// We prepare a quantize scale to categorize the values in 9 groups.
// The scale returns text values which can be used for the color CSS
// classes (q0-9, q1-9 ... q8-9). The domain will be defined once the
// values are known.
var quantize = d3.scale.quantize()
  .range(d3.range(9).map(function(i) { return 'q' + i + '-9'; }));

// ...
```

Listing 5.6: map.js - calculate domain

```
    // ...

    // Set the domain of the values (the minimum and maximum values of
    // all values of the current key) to the quantize scale.
    quantize.domain([
      d3.min(data, function(d) { return getValueOfData(d); }),
      d3.max(data, function(d) { return getValueOfData(d); })
    ]);

    // ...
```

Next

Proceed to *Step 6: Zoom*.

5.3 Code

- **For reference, the file `index.html` after step 5:** https://github.com/lvonlanthen/data-map-d3/blob/step-05/index.html
- **For reference, the file `style.css` after step 5:** https://github.com/lvonlanthen/data-map-d3/blob/step-05/style.css
- **For reference, the file `map.js` after step 5:** https://github.com/lvonlanthen/data-map-d3/blob/step-05/map.js
- **The diff view of step 4 and step 5:** https://github.com/lvonlanthen/data-map-d3/compare/step-04...step-05?diff=split

CHAPTER 6

Step 6: Zoom

In this step, we will add a zoom functionality to our map.

Contents

D3.js draws SVG elements, which are perfectly scalable, as they are vectors. In order to implement a zoom functionality, we therefore have to be able to access the features on the map and scale them.

6.1 Map features

We extract the map features.

Listing 6.1: map.js - extract mapFeatures

```
// ...

// We add a <g> element to the SVG element and give it a class to
// style. We also add a class name for Colorbrewer.
var mapFeatures = svg.append('g')
  .attr('class', 'features YlGnBu');

// ...
```

Listing 6.2: map.js - add path to mapFeatures

```
    // ...

    // We add the features to the <g> element created before.
    // D3 wants us to select the (non-existing) path objects first ...
    mapFeatures.selectAll('path')
        // ... and then enter the data. For each feature, a <path>
        // element is added.
        // ...
```

We did not really change much, we just did some refactoring by extracting mapFeatures as a variable so we can access it.

6.2 Enable zoom

Listing 6.3: map.js - zoom definition

```
// ...

// Define the zoom and attach it to the map
var zoom = d3.behavior.zoom()
  .scaleExtent([1, 10])
  .on('zoom', doZoom);
svg.call(zoom);

// ...
```

Now we need to define the function `doZoom()` which will actually do the zoom.

Listing 6.4: map.js - zoom function

```
// ...

/**
 * Zoom the features on the map. This rescales the features on the map.
 */
function doZoom() {
  mapFeatures.attr("transform",
    "translate(" + d3.event.translate + ") scale(" + d3.event.scale + ")");
}

// ...
```

Next

Proceed to *Step 7: Show the name on mouseover*.

6.3 Code

- **For reference, the file `index.html` after step 6:** https://github.com/lvonlanthen/data-map-d3/blob/step-06/index.html
- **For reference, the file `style.css` after step 6:** https://github.com/lvonlanthen/data-map-d3/blob/step-06/style.css
- **For reference, the file `map.js` after step 6:** https://github.com/lvonlanthen/data-map-d3/blob/step-06/map.js
- **The diff view of step 5 and step 6:** https://github.com/lvonlanthen/data-map-d3/compare/step-05...step-06?diff=split

CHAPTER 7

Step 7: Show the name on mouseover

In this step, we will implement a functionality to show the name of the feature (the municipality) when we move the mouse over it.

Contents

7.1 Show the boundaries of the municipalities

So far, we did not apply much style to the features beside the color which is based on the data. We will now make the boundaries of the municipalities visible by styling it in CSS.

Listing 7.1: style.css - feature boundaries

```css
/* ... */

g.features {
  stroke: #d8d8d8;
  stroke-width: 0.5;
}
g.features path:hover {
  opacity: 0.5;
}
```

We also added a little CSS effect to highlight the features when the mouse moves over them. You can refresh the browser to see the changes.

7.2 Keep stroke width proportional on zoom

If you looked at the features and zoomed in, you might have noticed that the boundaries of the features are becoming bigger and bigger. This is because the zoom function just scales the features and their boundaries up when zooming in.

We can solve this by making the stroke width proportional to the zoom level. These changes take place in the `doZoom()` function.

Listing 7.2: map.js - keep stroke width proportional to zoom

```
// ...

/**
 * Zoom the features on the map. This rescales the features on the map.
 * Keep the stroke width proportional when zooming in.
 */
function doZoom() {
  mapFeatures.attr("transform",
    "translate(" + d3.event.translate + ") scale(" + d3.event.scale + ")")
    // Keep the stroke width proportional. The initial stroke width
    // (0.5) must match the one set in the CSS.
    .style("stroke-width", 0.5 / d3.event.scale + "px");
}

// ...
```

7.3 Helper function

We will not access the name through the feature, but rather through the data itself. This can be done with the `dataById` object we defined earlier. We need to pass this object an ID, which is the ID of the feature. Since we will have to do this more than once in the future, we now define a helper function to gain easier access to it.

This also facilitates changes should we decide to use other geographic data (where the ID is not named `GMDNR`). In this case, we would only have to change this function.

Listing 7.3: map.js - helper function getIdOfFeature

```
// ...

/**
 * Helper function to retrieve the ID of a feature. The ID is found in
 * the properties of the feature.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
function getIdOfFeature(f) {
  return f.properties.GMDNR;
}
```

We can already use this helper function inside the quantize function so we don't have to access the property `GMDNR` manually.

Listing 7.4: map.js - use helper function in quantize

```
      // ...
        .attr('class', function(d) {
          // Use the quantized value for the class
          return quantize(getValueOfData(dataById[getIdOfFeature(d)]));
        })
        // ...
```

7.4 Show the name on mouseover

We will work with a container and a function called “tooltip”, even though in a first step it is not really a tooltip yet. We will come to that in the next step.

We start off by styling the tooltip container, even though it is not there yet. We also add a helper style `hidden` which - obviously - hides elements.

Listing 7.5: style.css - style tooltip

```css
/* ... */

.tooltip {
  font-weight: bold;
  padding: 0.5rem;
  border: 1px solid silver;
}

.hidden {
  display: none;
}

/* ... */
```

Next we add a `<div>` element for the tooltip, which is hidden by default.

Listing 7.6: map.js - tooltip element

```javascript
// ...

// We add a <div> container for the tooltip, which is hidden by default.
var tooltip = d3.select("#map")
  .append("div")
  .attr("class", "tooltip hidden");

// ...
```

We define a function which will show the tooltip.

Listing 7.7: map.js - tooltip function

```javascript
// ...

/**
 * Show a tooltip with the name of the feature.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
function showTooltip(f) {
  // Get the ID of the feature.
  var id = getIdOfFeature(f);
  // Use the ID to get the data entry.
  var d = dataById[id];
  // Show the tooltip (unhide it) and set the name of the data entry.
  tooltip.classed('hidden', false)
    .html(d.name);
}
```

```

// ...
```

Now we need to trigger the tooltip action. Notice that we also used our new helper function `getIdOfFeature`.

Listing 7.8: map.js - trigger tooltip

```
        // ...
        // As "d" attribute, we set the path of the feature.
        .attr('d', path)
        // When the mouse moves over a feature, show the tooltip.
        .on('mousemove', showTooltip);

  // ...
```

Next

Proceed to *Step 8: Tooltip*.

7.5 Code

- **For reference, the file `index.html` after step 7:** https://github.com/lvonlanthen/data-map-d3/blob/step-07/index.html
- **For reference, the file `style.css` after step 7:** https://github.com/lvonlanthen/data-map-d3/blob/step-07/style.css
- **For reference, the file `map.js` after step 7:** https://github.com/lvonlanthen/data-map-d3/blob/step-07/map.js
- **The diff view of step 6 and step 7:** https://github.com/lvonlanthen/data-map-d3/compare/step-06...step-07?diff=split

CHAPTER 8

Step 8: Tooltip

In this step, we will use the mouseover functionality we created earlier and position the container showing the name to act as a tooltip which appears next to the mouse cursor.

Contents

8.1 Tooltip styling

First, we add some style for the tooltip. Most of it is just to make it look nicer. However, important is the declaration to use absolute positioning (`position: absolute;`). This allows to use the attributes `top` and `left` to position an element. Since the position will depend on the mouse cursor whose position we don't know yet, we leave that for the moment and add it later in the code.

Listing 8.1: style.css - tooltip styling

```css
/* ... */

.tooltip {
  font-weight: bold;
  padding: 0.5rem;
  border: 1px solid silver;
  color: #222;
  background: #fff;
  border-radius: 5px;
  box-shadow: 0px 0px 5px 0px #a6a6a6;
  opacity: 0.9;
  position: absolute;
}

/* ... */
```

8.2 Tooltip positioning

To calculate the position of the tooltip (the `left` and `top` CSS attributes), we take the map container as reference element and get the position of the mouse cursor. Then we add a little offset so the tooltip appears next to the cursor and not on top of it.

We also try to guess if the tooltip is shown outside the map container on the right side. In this case, we show it more to the left so it does not go over the map. *Yes, this it rather buggy and you might as well leave that part.*

Listing 8.2: map.js - tooltip positioning

```
// ...

/**
 * Show a tooltip with the name of the feature. Calculate the position
 * of the cursor to show the tooltip next to the mouse.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
function showTooltip(f) {
  // Get the ID of the feature.
  var id = getIdOfFeature(f);
  // Use the ID to get the data entry.
  var d = dataById[id];

  // Get the current mouse position (as integer)
  var mouse = d3.mouse(d3.select('#map').node()).map(
    function(d) { return parseInt(d); }
  );

  // Calculate the absolute left and top offsets of the tooltip. If the
  // mouse is close to the right border of the map, show the tooltip on
  // the left.
  var left = Math.min(width - 4 * d.name.length, mouse[0] + 5);
  var top = mouse[1] + 25;

  // Show the tooltip (unhide it) and set the name of the data entry.
  // Set the position as calculated before.
  tooltip.classed('hidden', false)
    .attr("style", "left:" + left + "px; top:" + top + "px")
    .html(d.name);
}

// ...
```

8.3 Hide tooltip

Right now, the tooltip never disappears. If we move the cursor out of the map or even out of Switzerland, we would like the tooltip to disappear.

Listing 8.3: map.js - hide tooltip function

```
// ...

/**
```

```
 * Hide the tooltip.
 */
function hideTooltip() {
  tooltip.classed('hidden', true);
}

// ...
```

And we need to trigger the action when the mouse moves out of the features.

Listing 8.4: index.html - trigger tooltip hiding

```
        // ...
        .attr('d', path)
        // When the mouse moves over a feature, show the tooltip.
        .on('mousemove', showTooltip)
        // When the mouse moves out of a feature, hide the tooltip.
        .on('mouseout', hideTooltip);

  // ...
```

Next

Proceed to *Step 9: Show details of features*.

8.4 Code

- **For reference, the file `index.html` after step 8:** https://github.com/lvonlanthen/data-map-d3/blob/step-08/index.html
- **For reference, the file `style.css` after step 8:** https://github.com/lvonlanthen/data-map-d3/blob/step-08/style.css
- **For reference, the file `map.js` after step 8:** https://github.com/lvonlanthen/data-map-d3/blob/step-08/map.js
- **The diff view of step 7 and step 8:** https://github.com/lvonlanthen/data-map-d3/compare/step-07...step-08?diff=split

CHAPTER 9

Step 9: Show details of features

In this step, we use a container to show details of a feature when clicking it.

Contents

9.1 Details container

We did not touch our HTML document in a while and now we are going to create a `<div>` container just underneath the map. In this container, we will show the details of a feature when the user clicks on it. The container is hidden initially.

Listing 9.1: index.html - details container

```
    <!-- ... -->
    <div id="map"><!-- Map container --></div>

    <div id="details" class="hidden"><!-- Details container --></div>

    <!-- JS libraries -->
    <!-- ... -->
```

9.2 Click function to show details

Similar to the function to show a tooltip, we will create a function to show the details of a feature.

We are creating a function which handles showing the details.

Listing 9.2: map.js - function to show details

```
// ...

/**
```

```
 * Show the details of a feature in the details <div> container.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
function showDetails(f) {
  // Get the ID of the feature.
  var id = getIdOfFeature(f);
  // Use the ID to get the data entry.
  var d = dataById[id];

  // The details HTML output is just the name
  var detailsHtml = d.name;

  // Put the HTML output in the details container and show (unhide) it.
  d3.select('#details').html(detailsHtml);
  d3.select('#details').classed("hidden", false);
}

// ...
```

And we trigger the function when a user clicks on a feature.

Listing 9.3: map.js - trigger function to show details

```
        // ...
        .attr('d', path)
        // When the mouse moves over a feature, show the tooltip.
        .on('mousemove', showTooltip)
        // When the mouse moves out of a feature, hide the tooltip.
        .on('mouseout', hideTooltip)
        // When a feature is clicked, show the details of it.
        .on('click', showDetails);

  // ...
```

9.3 Mustache templates

Until now, we are just showing the name of the feature in the details container. We would like to show much more information, for example a table showing all the data we have on this feature (name, total area, forest area in % etc.).

Returning HTML output from a JavaScript function can be a bit painful, as we have to create a string full of HTML tags and variables between. This can become quite ugly, especially if we want to output an HTML table.

This is why we are going to use Mustache templates and more specifically mustache.js since we are dealing with JavaScript. As stated on their homepage, Mustache is a logic-less template syntax. It can be used for HTML, config files, source code - anything. It works by expanding tags in a template using values provided in a hash or object.

First, we are going to include the mustache.js JavaScript library in our HTML document.

Listing 9.4: index.html - include mustache library

```
    <!-- ... -->

    <!-- JS libraries -->
    <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/mustache.js/2.2.1/mustache.min.js"></script>

    <!-- ... -->
```

Now we are ready to create a Mustache template. We can do so right in our HTML document. This way, we will have all of our HTML output in one file. To define Mustache markup in a HTML document, we can use the `<script>` tag with type `x-tmpl-mustache`. The script within this block will not be executed until we tell Mustache to do so.

Listing 9.5: index.html - mustache template

```html
    <!-- ... -->
    <div id="details" class="hidden"><!-- Details container --></div>

    <!-- Mustache template, rendered later to show the details of a feature -->
    <script id="template" type="x-tmpl-mustache">
      <h3>{{ name }}</h3>
      <table>
        <tr>
          <th>Total area:</th>
          <td>{{ area }} km&sup2;</td>
        </tr>
        <tr>
          <th>Urban area:</th>
          <td>{{ urban }} %</td>
        </tr>
        <tr>
          <th>Agricultural area:</th>
          <td>{{ agriculture }} %</td>
        </tr>
        <tr>
          <th>Forest area:</th>
          <td>{{ forest }} %</td>
        </tr>
        <tr>
          <th>Unproductive area:</th>
          <td>{{ unproductive }} %</td>
        </tr>
      </table>
    </script>

    <!-- JS libraries -->
    <!-- ... -->
```

Now we need to tell Mustache to load this template and parse it so it can be filled with values later.

Listing 9.6: map.js - parse mustache template

```javascript
// ...

// We get and prepare the Mustache template, parsing it speeds up future uses
var template = d3.select('#template').html();
Mustache.parse(template);

// ...
```

And finally, in our showDetails function, we replace our previously created detailsHtml output with our rendered Mustache template.

Listing 9.7: map.js - render mustache template

```
// ...

/**
 * Show the details of a feature in the details <div> container.
 * The content is rendered with a Mustache template.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
function showDetails(f) {
  // Get the ID of the feature.
  var id = getIdOfFeature(f);
  // Use the ID to get the data entry.
  var d = dataById[id];

  // Render the Mustache template with the data object and put the
  // resulting HTML output in the details container.
  var detailsHtml = Mustache.render(template, d);

  // Put the HTML output in the details container and show (unhide) it.
  d3.select('#details').html(detailsHtml);
  d3.select('#details').classed("hidden", false);
}

// ...
```

Next

Proceed to *Step 10: A bit of styling*.

9.4 Code

- **For reference, the file `index.html` after step 9:** https://github.com/lvonlanthen/data-map-d3/blob/step-09/index.html
- **For reference, the file `style.css` after step 9:** https://github.com/lvonlanthen/data-map-d3/blob/step-09/style.css
- **For reference, the file `map.js` after step 9:** https://github.com/lvonlanthen/data-map-d3/blob/step-09/map.js
- **The diff view of step 8 and step 9:** https://github.com/lvonlanthen/data-map-d3/compare/step-08...step-09?diff=split

CHAPTER 10

Step 10: A bit of styling

In this step, we are going to do some first styling of our page.

Contents

10.1 Skeleton CSS

We are going to use a CSS framework. Since we want to keep our page rather simple and we are not going to use a lot of fancy CSS features, we are going to use Skeleton. Skeleton is a very lightweight CSS library which comes with a responsive grid system, which we will look at later.

For now, let's just include the Skeleton CSS library in our HTML document.

Listing 10.1: index.html - include skeleton

```
    <!-- ... -->
    <title>Land use statistics map</title>

    <!-- CSS libraries -->
    <link href="http://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.min.css" rel="styleshee

    <!-- Custom CSS styles -->
    <!-- ... -->
```

And we see that Skeleton already applied some styles to our page, namely the font changed and the table looks much nicer now.

10.2 Some styling

While our table looks a bit nicer now, Skeleton also added a lot of space (padding) between the table cells. We can overwrite this in our custom style sheet. We also add a little margin at the top of our details container so it does not stick as close to the map.

Listing 10.2: style.css - table styling

```css
/* ... */

#details {
  margin-top: 2rem;
}

table {
  margin-bottom: 0;
}
td, th {
  padding: 6px 9px;
}
td {
  text-align: right;
}

/* ... */
```

10.3 Centering content

Skeleton provides a wrapper to center content by adding a class `container` to the element. So we put all the HTML containers we have so far in a `<div>` element with this class.

Listing 10.3: index.html - center content

```html
  <!-- ... -->
  <body>
    <div class="container">
      <h3>Land use statistics map</h3>
      <div id="map"><!-- Map container --></div>

      <div id="details" class="hidden"><!-- Details container --></div>
    </div>

    <!-- Mustache template, rendered later to show the details of a feature -->
    <!-- ... -->
```

And we see that all the content is nicely centered, at least on larger screens. If the screen width is too narrow, the map is not centered yet, but we will fix that later on.

10.4 Grid layout

Skeleton is based on a grid layout where each row contains 12 columns.

This means that the markup for a row with two columns of equal width would look like this:

Listing 10.4: markup to create columns with skeleton

```html
<div class="row">
  <div class="six columns">Column 1</div>
  <div class="six columns">Column 2</div>
</div>
```

We can use this in our details container to display the name of the feature on the left and the table with the data on the right.

First, we add the class `row` to the details container which will eventually hold the details.

Listing 10.5: index.html - class row for details container

```
    <!-- ... -->
    <div class="container">
      <h3>Land use statistics map</h3>
      <div id="map"><!-- Map container --></div>

      <div id="details" class="hidden row"><!-- Details container --></div>
    </div>
    <!-- ... -->
```

Next, we add the `columns` class in the Mustache template.

Listing 10.6: index.html - name and table in columns

```
    <!-- ... -->

    <!-- Mustache template, rendered later to show the details of a feature -->
    <script id="template" type="x-tmpl-mustache">
      <h3 class="six columns">{{ name }}</h3>
      <table class="six columns">
        <tr>
          <!-- ... -->
```

Next

Proceed to *Step 11: Responsive map and instructions*.

10.5 Code

- **For reference, the file `index.html` after step 10:** https://github.com/lvonlanthen/data-map-d3/blob/step-10/index.html
- **For reference, the file `style.css` after step 10:** https://github.com/lvonlanthen/data-map-d3/blob/step-10/style.css
- **For reference, the file `map.js` after step 10:** https://github.com/lvonlanthen/data-map-d3/blob/step-10/map.js
- **The diff view of step 9 and step 10:** https://github.com/lvonlanthen/data-map-d3/compare/step-09...step-10?diff=split

CHAPTER 11

Step 11: Responsive map and instructions

In this step, we are going make our map responsive and we will add some instructions on how to use the map.

Contents

11.1 Full screen responsive map

Since we included Skeleton, you may have noticed that the row used for the details container is wider than the map container on wide screens. This does not look very nice and comes from the fact, that Skeleton uses a max width of 960px on large screens, but our map still has a hard coded width of 900px.

We can remove the width (and height) definition in the CSS style of the map container.

Listing 11.1: style.css - full screen map

```css
/* ... */

#map {
  border: 1px solid silver;
  background: #E6E6E6;
}

/* ... */
```

While this lets the map container take up the full width of the row (same as the row in the details container), the map now has an ugly grey border when zooming in. Also, on smaller screens the map is now bigger than the container itself.

This is because the SVG element we created in the map still has a hard coded width of 900px.

To fix this, we do not set the width and height of the SVG element explicitely and use a viewbox instead. The viewbox receives initial dimensions (our width and height) and we tell it to preserve the aspect ratio so. This permits the container to handle rescaling automatically and while preserving the aspect ratio if the container is smaller or bigger than initially set.

Listing 11.2: map.js - table styling

```
// ...

// We create a SVG element in the map container and give it some
// dimensions. We can use a viewbox and preserve the aspect ratio. This
// also allows a responsive map which rescales and looks good even on
// different screen sizes
var svg = d3.select('#map').append('svg')
  .attr("preserveAspectRatio", "xMidYMid")
  .attr("viewBox", "0 0 " + width + " " + height);

// ...
```

Now the map looks and zooms quite nicely, regarding of the screen size. The only thing which is not pleasant is a small grey border at the bottom of the map, visible especially when zooming in. This seems to be an issue of the font size within the map container and it can be fixed in the CSS style by setting the font-size of the map container to 0.

Listing 11.3: style.css - set font-size in map container to 0

```
/* ... */

#map {
  font-size: 0; /* to prevent margin at bottom of map container */
  border: 1px solid silver;
  background: #E6E6E6;
}

/* ... */
```

However, this also sets the font size of the tooltip to 0, so we need to prevent that by setting a font size specifically for the tooltip.

Listing 11.4: style.css - font-size for tooltip

```
/* ... */

.tooltip {
  font-weight: bold;
  padding: 0.5rem;
  border: 1px solid silver;
  color: #222;
  background: #fff;
  border-radius: 5px;
  box-shadow: 0px 0px 5px 0px #a6a6a6;
  opacity: 0.9;
  position: absolute;
  font-size: 1.5rem;
}

/* ... */
```

The map is now fully responsive and scales nicely when the screen size changes. However, there is a lot of margin at the side of the map and especially on small screens we would rather like the map to fill up the space available. You can overwrite the margin of the container element in the CSS style sheet:

Listing 11.5: style.css - less margin for container

```css
/* ... */

.container {
  width: 100%;
}

/* ... */
```

11.2 Instructions

We would like to add some instructions on how to use the map and we can add this in the HTML document.

Listing 11.6: index.html - instructions

```html
    <!-- ... -->
    <div class="container">
      <h3>Land use statistics map</h3>
      <div id="map"><!-- Map container --></div>

      <div id="details" class="hidden row"><!-- Details container --></div>
      <div id="initial">
        <h5>Instructions</h5>
        <ul>
          <li>Select a municipality to show the details.</li>
          <li>Scroll in the map to zoom in and out.</li>
        </ul>
      </div>
    </div>
    <!-- ... -->
```

We also set some margin at the top, same as for the details container.

Listing 11.7: style.css - top margin for instructions

```css
/* ... */

#details, #initial {
  margin-top: 2rem;
}

/* ... */
```

And finally we want the initial instructions to disappear when the details of a feature are shown.

Listing 11.8: map.js - hide initial content when details are shown

```javascript
// ...

/**
 * Show the details of a feature in the details <div> container.
 * The content is rendered with a Mustache template.
 *
 * @param {object} f - A GeoJSON Feature object.
 */
```

```
function showDetails(f) {
  // Get the ID of the feature.
  var id = getIdOfFeature(f);
  // Use the ID to get the data entry.
  var d = dataById[id];

  // Render the Mustache template with the data object and put the
  // resulting HTML output in the details container.
  var detailsHtml = Mustache.render(template, d);

  // Hide the initial container.
  d3.select('#initial').classed("hidden", true);

  // Put the HTML output in the details container and show (unhide) it.
  d3.select('#details').html(detailsHtml);
  d3.select('#details').classed("hidden", false);
}

// ...
```

Next

Proceed to *Step 12: Dropdown to select map key*.

11.3 Code

- **For reference, the file `index.html` after step 11:** https://github.com/lvonlanthen/data-map-d3/blob/step-11/index.html
- **For reference, the file `style.css` after step 11:** https://github.com/lvonlanthen/data-map-d3/blob/step-11/style.css
- **For reference, the file `map.js` after step 11:** https://github.com/lvonlanthen/data-map-d3/blob/step-11/map.js
- **The diff view of step 10 and step 11:** https://github.com/lvonlanthen/data-map-d3/compare/step-10...step-11?diff=split

CHAPTER 12

Step 12: Dropdown to select map key

In this step, we are going to create a dropdown to select the key after which the map is colored.

Contents

12.1 Function for the coloring

Until now, the map is colored when the CSV inside the GeoJSON is loaded. We want to be able to change the colors without having to reload the CSV data (and the GeoJSON) because the data is already available.

So far, the data of the CSV is only available within the function which loads the file. We need to be able to access this data from outside of this function so we declare a new variable in which we store the data once it is available.

Listing 12.1: map.js - variable to store the data

```
// ...

// We define a variable to later hold the data of the CSV.
var mapData;

// ...
```

In the function where we load the CSV, we store the data to the new variable.

Listing 12.2: map.js - store the data

```
  // ...

  // Read the data for the cartogram
  d3.csv('data/areastatistics.csv', function(data) {

    // We store the data object in the variable which is accessible from
    // outside of this function.
    mapData = data;

    // This maps the data of the CSV so it can be easily accessed by
    // the ID of the municipality, for example: dataById[2196]

    // ...
```

Now we can use the map data to upate the colors any time we want. We can therefore define a function which will do exactly that. The coloring is already done in the CSV function and we can take a lot of it to create our function, namely setting the domain of the quantize scale (as this depends on which data key we are looking at) and the setting the class of the feature paths.

Listing 12.3: map.js - function to update map colors

```
// ...

/**
 * Update the colors of the features on the map. Each feature is given a
 * CSS class based on its value.
 */
function updateMapColors() {
  // Set the domain of the values (the minimum and maximum values of
  // all values of the current key) to the quantize scale.
  quantize.domain([
    d3.min(mapData, function(d) { return getValueOfData(d); }),
    d3.max(mapData, function(d) { return getValueOfData(d); })
  ]);
  // Update the class (determining the color) of the features.
  mapFeatures.selectAll('path')
    .attr('class', function(f) {
      // Use the quantized value for the class
      return quantize(getValueOfData(dataById[getIdOfFeature(f)]));
    });
}

// ...
```

Since setting the domain of the quantize scale and setting the class of the feature paths is now handled by a separate function, we can remove this part from the CSV file and instead call the function we just created. This prevents duplication of code.

Listing 12.4: map.js - call updateMapColors from within CSV

```
  // ...

  // Read the data for the cartogram
  d3.csv('data/areastatistics.csv', function(data) {

    // We store the data object in the variable which is accessible from
    // outside of this function.
    mapData = data;

    // This maps the data of the CSV so it can be easily accessed by
    // the ID of the municipality, for example: dataById[2196]
    dataById = d3.nest()
      .key(function(d) { return d.id; })
      .rollup(function(d) { return d[0]; })
      .map(data);

    // We add the features to the <g> element created before.
    // D3 wants us to select the (non-existing) path objects first ...
    mapFeatures.selectAll('path')
        // ... and then enter the data. For each feature, a <path>
        // element is added.
        .data(features.features)
      .enter().append('path')
        // As "d" attribute, we set the path of the feature.
        .attr('d', path)
        // When the mouse moves over a feature, show the tooltip.
        .on('mousemove', showTooltip)
        // When the mouse moves out of a feature, hide the tooltip.
        .on('mouseout', hideTooltip)
        // When a feature is clicked, show the details of it.
        .on('click', showDetails);

    // Call the function to update the map colors.
    updateMapColors();

  });

  // ...
```

12.2 Dropdown to select key

Now we need a possibility to change the key and for this, we will use a dropdown which we will place next to the heading above the map. We will therefore move the heading in a “row”-container and within that in the first column. In the second column of the row, we will create the dropdown.

Listing 12.5: index.html - dropdown to select key

```
    <!-- ... -->
    <div class="container">
      <div class="row">
        <h3 class="nine columns">Land use statistics map</h3>
        <select id="select-key" class="three columns">
          <option value="urban" selected="selected">Urban area in %</option>
          <option value="agriculture">Agricultural area in %</option>
          <option value="forest">Forest area in %</option>
          <option value="unproductive">Unproductive area in %</option>
        </select>
      </div>
      <div id="map"><!-- Map container --></div>

      <!-- ... -->
```

12.3 Action to change the key

Now we need to trigger an update of the map colors when an option of the dropdown is selected.

Listing 12.6: map.js - trigger action to change map colors

```
// ...

// Listen to changes of the dropdown to select the key to visualize on
// the map.
d3.select('#select-key').on('change', function(a) {
  // Change the current key and call the function to update the colors.
  currentKey = d3.select(this).property('value');
  updateMapColors();
});

// ...
```

And finally we can update the instructions as this is a new action the user than perform.

Listing 12.7: index.html - update instructions

```
      <!-- ... -->
        <h5>Instructions</h5>
        <ul>
          <li>Change the key using the dropdown above the map.</li>
          <li>Select a municipality to show the details.</li>
          <li>Scroll in the map to zoom in and out.</li>
        </ul>
      <!-- ... -->
```

Next

Proceed to *Step 13: Declare sources and show instructions again*.

12.4 Code

- **For reference, the file `index.html` after step 12:** https://github.com/lvonlanthen/data-map-d3/blob/step-12/index.html
- **For reference, the file `style.css` after step 12:** https://github.com/lvonlanthen/data-map-d3/blob/step-12/style.css
- **For reference, the file `map.js` after step 12:** https://github.com/lvonlanthen/data-map-d3/blob/step-12/map.js
- **The diff view of step 11 and step 12:** https://github.com/lvonlanthen/data-map-d3/compare/step-11...step-12?diff=split

Step 13: Declare sources and show instructions again

In this step, we are going to declare the sources we used for the map and add the possibility to return to the instructions after showing the details of a feature.

Contents

13.1 Add a footer

It is very important to declare the data sources used to create the map and we will do so by adding a footer at the bottom of the page.

Listing 13.1: index.html - footer with data sources

```html
      <!-- ... -->

      <div id="details" class="hidden row"><!-- Details container --></div>
      <div id="initial">
        <h5>Instructions</h5>
        <ul>
          <li>Change the key using the dropdown above the map.</li>
          <li>Select a municipality to show the details.</li>
          <li>Scroll in the map to zoom in and out.</li>
        </ul>
      </div>
      <div class="footer">
        <strong>Data sources</strong>: Data from the Swiss Federal Statistical Office, both the <a hr
      </div>
    </div>

    <!-- ... -->
```

And we style it with CSS

Listing 13.2: style.css - footer style

```
/* ... */

.footer {
  border-top: 1px solid silver;
  color: #888888;
  font-size: 1.25rem;
  text-align: center;
  margin-top: 1rem;
  padding: 0.5rem;
}

/* ... */
```

13.2 Return to instructions

Right now the instructions are only visible when the page is first loaded. After clicking on a feature the instructions are hidden and there is no way to bring them back.

We will now add a button to close the details and bring back the initial instructions.

First, we create a button as part of the Mustache template for the details of a feature. To do this, we reduce the number of columns the table is using up from 6 to 5. This makes room for one single column after the table which contains the button. Notice the `column` class name for single column (as opposed to `columns` for multiple columns).

Listing 13.3: index.html - close button in template

```
    <!-- ... -->

    <!-- Mustache template, rendered later to show the details of a feature -->
    <script id="template" type="x-tmpl-mustache">
      <h3 class="six columns">{{ name }}</h3>
      <table class="five columns">
        <tr>
          <th>Total area:</th>
          <td>{{ area }} km&sup2;</td>
        </tr>
        <tr>
          <th>Urban area:</th>
          <td>{{ urban }} %</td>
        </tr>
        <tr>
          <th>Agricultural area:</th>
          <td>{{ agriculture }} %</td>
        </tr>
        <tr>
          <th>Forest area:</th>
          <td>{{ forest }} %</td>
        </tr>
        <tr>
          <th>Unproductive area:</th>
          <td>{{ unproductive }} %</td>
        </tr>
      </table>
      <a href="#" id="close-details" class="one column" onclick="hideDetails(); return false;">&#x27
    </script>

    <!-- ... -->
```

Now we need to define the function `hideDetails()` which will hide the details container and show the initial instructions container again.

Listing 13.4: map.js - function to hide the details

```
// ...

/**
 * Hide the details <div> container and show the initial content instead.
 */
function hideDetails() {
  // Hide the details
  d3.select('#details').classed("hidden", true);
  // Show the initial content
  d3.select('#initial').classed("hidden", false);
}

// ...
```

Lastly, we apply some styling to the button.

Listing 13.5: style.css - close button style

```
/* ... */

#close-details {
  color: #FA5858;
  text-decoration: none;
  font-size: 4rem;
  text-align: right;
  line-height: 4.5rem;
}
#close-details:hover {
  color: red;
}

/* ... */
```

Next

Proceed to *Step 14: Map legend*.

13.3 Code

- **For reference, the file `index.html` after step 13:** https://github.com/lvonlanthen/data-map-d3/blob/step-13/index.html
- **For reference, the file `style.css` after step 13:** https://github.com/lvonlanthen/data-map-d3/blob/step-13/style.css
- **For reference, the file `map.js` after step 13:** https://github.com/lvonlanthen/data-map-d3/blob/step-13/map.js

- **The diff view of step 12 and step 13:** https://github.com/lvonlanthen/data-map-d3/compare/step-12...step-13?diff=split

Step 14: Map legend

In this step, we are going to create a legend for the colors on the map.

Contents

14.1 Legend

First, we are going to create a container in the HTML document for the legend.

Listing 14.1: index.html - legend container

```
      <!-- ... -->
      <div id="map"><!-- Map container --></div>
      <div id="legend"><!-- Legend container --></div>

      <!-- ... -->
```

We add the legend SVG and define a function which will update the legend if the key of the map changes or if the window is resized.

Listing 14.2: map.js - legend functions

```
// ...

// We prepare a number format which will always return 2 decimal places.
var formatNumber = d3.format('.2f');

// For the legend, we prepare a very simple linear scale. Domain and
// range will be set later as they depend on the data currently shown.
var legendX = d3.scale.linear();

// We use the scale to define an axis. The tickvalues will be set later
// as they also depend on the data.
var legendXAxis = d3.svg.axis()
  .scale(legendX)
  .orient("bottom")
  .tickSize(13)
  .tickFormat(function(d) {
    return formatNumber(d);
  });

// We create an SVG element in the legend container and give it some
// dimensions.
var legendSvg = d3.select('#legend').append('svg')
  .attr('width', '100%')
  .attr('height', '44');

// To this SVG element, we add a <g> element which will hold all of our
// legend entries.
var g = legendSvg.append('g')
    .attr("class", "legend-key YlGnBu")
    .attr("transform", "translate(" + 20 + "," + 20 + ")");

// We add a <rect> element for each quantize category. The width and
// color of the rectangles will be set later.
g.selectAll("rect")
    .data(quantize.range().map(function(d) {
      return quantize.invertExtent(d);
    }))
  .enter().append("rect");

// We add a <text> element acting as the caption of the legend. The text
// will be set later.
g.append("text")
    .attr("class", "caption")
    .attr("y", -6)

/**
 * Function to update the legend.
 * Somewhat based on http://bl.ocks.org/mbostock/4573883
 */
function updateLegend() {

  // We determine the width of the legend. It is based on the width of
  // the map minus some spacing left and right.
  var legendWidth = d3.select('#map').node().getBoundingClientRect().width - 50;

  // We determine the domain of the quantize scale which will be used as
  // tick values. We cannot directly use the scale via quantize.scale()
  // as this returns only the minimum and maximum values but we need all
  // the steps of the scale. The range() function returns all categories
  // and we need to map the category values (q0-9, ..., q8-9) to the
  // number values. To do this, we can use invertExtent().
  var legendDomain = quantize.range().map(function(d) {
    var r = quantize.invertExtent(d);
    return r[1];
  });
  // Since we always only took the upper limit of the category, we also
  // need to add the lower limit of the very first category to the top
  // of the domain.
  legendDomain.unshift(quantize.domain()[0]);

  // On smaller screens, there is not enough room to show all 10
  // category values. In this case, we add a filter leaving only every
  // third value of the domain.
  if (legendWidth < 400) {
    legendDomain = legendDomain.filter(function(d, i) {
      return i % 3 == 0;
    });
  }

  // We set the domain and range for the x scale of the legend. The
  // domain is the same as for the quantize scale and the range takes up
  // all the space available to draw the legend.
  legendX
    .domain(quantize.domain())
    .range([0, legendWidth]);

  // We update the rectangles by (re)defining their position and width
  // (both based on the legend scale) and setting the correct class.
  g.selectAll("rect")
    .data(quantize.range().map(function(d) {
      return quantize.invertExtent(d);
    }))
    .attr("height", 8)
    .attr("x", function(d) { return legendX(d[0]); })
    .attr("width", function(d) { return legendX(d[1]) - legendX(d[0]); })
    .attr('class', function(d, i) {
      return quantize.range()[i];
    });

  // We update the legend caption. To do this, we take the text of the
  // currently selected dropdown option.
  var keyDropdown = d3.select('#select-key').node();
  var selectedOption = keyDropdown.options[keyDropdown.selectedIndex];
  g.selectAll('text.caption')
    .text(selectedOption.text);

  // We set the calculated domain as tickValues for the legend axis.
  legendXAxis
    .tickValues(legendDomain)

  // We call the axis to draw the axis.
  g.call(legendXAxis);
}

// ...
```

We call the function after the window has been resized and when the map colors have been updated.

Listing 14.3: map.js - update legend on window resize

```
// ...

// We add a listener to the browser window, calling updateLegend when
// the window is resized.
window.onresize = updateLegend;

// ...
```

Listing 14.4: map.js - update legend after map colors change

```
  // ...

  // We call the function to update the legend.
  updateLegend();
// ...
```

Lastly, we need some style for the legend container and the legend rectangles.

Listing 14.5: style.css - legend style

```
/* ... */

#legend {
  border: 1px solid silver;
  border-top: 0;
}

.legend-key path {
  display: none;
}

.legend-key text {
  font-size: 1rem;
}

.legend-key line {
  stroke: #000;
  shape-rendering: crispEdges;
}

/* ... */
```

14.2 Code

- **For reference, the file `index.html` after step 14:** https://github.com/lvonlanthen/data-map-d3/blob/step-14/index.html
- **For reference, the file `style.css` after step 14:** https://github.com/lvonlanthen/data-map-d3/blob/step-14/style.css
- **For reference, the file `map.js` after step 14:** https://github.com/lvonlanthen/data-map-d3/blob/step-14/map.js
- **The diff view of step 13 and step 14:** https://github.com/lvonlanthen/data-map-d3/compare/step-13...step-14?diff=split

CHAPTER 15

Indices and tables

- genindex
- modindex
- search