2-2, Lesson 1 - D3 Helpers: TopoJSON, MapShaper, and d3.queue

This module will build upon the basics of D3 covered before to explore the library's powerful mapping functionality. First, to prepare our data for mapping, we will cover some useful ancillary tools—namely, the TopoJSON data format, the MapShaper web application, and d3.queue.

The first lesson of the module will introduce you to TopoJSON and walk you through the process of using MapShaper to simplify and convert GeoJSON data into this more compact format. It will also cover how to import multiple data files into a single AJAX callback using *a d3 queue* , and how to convert your TopoJSON back into displayable GeoJSON data using *topojson.js*. The second lesson will tackle the somewhat complex but vital topic of D3 projections. It will then walk through how to use D3 projection and path generators to map spatial data as vector linework in the browser, and finally, how to create a graticule for background reference.

When you have finished this module, you should be able to:

- Convert shapefile and GeoJSON data into TopoJSON format, import data from multiple files to the DOM, and translate TopoJSON data into GeoJSON for display in the browser.
- Implement an appropriate projection for your choropleth map using a D3 projection generator, correctly manipulate its projection parameters, and draw geometries from spatial data and elements of a graticule using path generators.

> **Note!**
>
> There are two reference pieces that you may find useful while completing this module:
>
> - Murray 2013, ***Interactive Data Visualization for the Web* *(http://chimera.labs.oreilly.com/books/1230000000345/index.html)*** , Chapter 12 ("Geomapping").
> - Bostock 2012, ***Let's Make a Map* *(http://bost.ocks.org/mike/map/)*** .

1.1 An Introduction to TopoJSON

TopoJSON (https://github.com/mbostock/topojson/wiki) is the first geospatial data format for the Web that encodes topology. Recall from your introductory GIS course that **topology** is the encoding of spatial relationships into digitized representations of geography. In desktop GIS software, **topology (http://webhelp.esri.com/arcgisserver/9.3/java/index.htm#geodatabases/topology_basics.htm)** is encoded by coverage and geodatabase file formats (but not in shapefiles). The advantages of topological over "spaghetti model" data formats such as shapefiles and GeoJSON are threefold:

1. Data integrity can be maintained when features are edited (that is, editing feature boundaries will not result in gaps or overlaps between features);
2. Spatial analysis using the relationships between features is easier to perform;
3. File size is significantly reduced because overlapping vertices and feature edges are eliminated.

For mapping vector data on the Web, this last point is especially important. While bandwidths seems to always be increasing and browsers are getting better at handling vector linework, as of this writing there is still a significant performance penalty for drawing very large datasets as SVG in the browser. The GeoJSON format was designed based on the **Simple Features standard (http://www.opengeospatial.org/standards/sfa)** , and sees points, lines, and polygons as individual features in a `FeatureCollection`. Thus, in any polygon dataset, each line that forms an edge between two polygons is duplicated—it's stored separately for each feature that uses it.

The TopoJSON format, created by D3 creator Mike Bostock, eliminates all of this duplicate data by implementing topology using JSON syntax. TopoJSON files are a fraction of the size of the equivalent data as GeoJSON, making them much faster to draw in the browser. Rather than store all of the vertices as coordinate arrays for each feature, TopoJSON stores a list of Arcs for each feature, a separate list of arc coordinates, and a mathematical transform to georeference those coordinates within the WGS 84 coordinate reference system.

Examples 1.1 and 1.2 show a single, relatively simple polygon feature stored as GeoJSON and TopoJSON, respectively. Compare these two examples and notice the differences in data structure.

Example 1.1: A single polygon feature stored as GeoJSON

JSON

```
{
    "type":"FeatureCollection",
    "features":[
        {
            "type":"Feature",
            "properties":{
                "adm1_code":"FRA-2669",
                "name":"Midi-Pyrénées"
            },
            "geometry":{
                "type":"Polygon",
                "coordinates":[[[1.546190219702851,45.027639472483656],[1.799559766669859,44.93286489529868],
[2.058717075132904,44.97511037934049],[2.18160363120262,44.63122955969641],[2.454713575870812,44.661563625760
72],[2.726738314921306,44.93534536399733],[2.926571079291875,44.768585516808116],[2.925795932935955,44.690812
48620722],[3.066252476370664,44.56420522708953],[3.140718215073548,44.420958157492976],[3.123354933283508,44.
26339671468662],[3.249497105307171,44.19115306316331],[3.306444532589069,44.06914500623702],[3.44240522572056
3,44.000234483234806],[3.050904575285927,43.701467189958805],[2.775624220281543,43.61790639876125],[2.6413688
48593459,43.6330992702151],[2.656716749678253,43.460784207325275],[2.567523228414245,43.422827867112346],[2.2
61495396195585,43.442258206451015],[2.031690300966602,43.42256948469395],[1.742302280126125,43.32779490840829
],[1.719047885851239,43.18341095635054],[1.952728712859823,43.095741889376654],[1.977223340584715,42.86374054
628914],[2.086157261348546,42.73814097842342],[1.938310987761838,42.571794542384225],[1.466814411955511,42.64
145518001254],[1.068182413787383,42.77612396285059],[0.656321248027155,42.838419902056785],[0.608158813078774
,42.6879898078833],[0.275388428433189,42.66868866065312],[0.169167521264058,42.72646291113443],[-0.0389334715
02503,42.68514760397869],[-0.305134937527328,42.83081010690091],[-0.29002844865397,42.98760895428978],[-0.080
687222437348,43.17250722872558],[0.023234084085686,43.339163723127285],[-0.047278407572435,43.5222274849105],
[-0.267006598375815,43.62317739542044],[-0.195279709890713,43.738855089390825],[-0.16747779026781,43.93091054
908257],[-0.00885698116474,43.92398590782335],[0.592501662122004,44.07281403334093],[0.65890587710453,44.0375
70705823384],[0.905919223593287,44.201023261114585],[1.051336704425353,44.3677831083038],[1.012992790584804,4
4.54756541571135],[1.287963088125991,44.71737417327944],[1.546190219702851,45.027639472483656]]]
            }
        }
    ]
}
```

Example 1.2: The same polygon feature as in Example 1.1, stored as TopoJSON

JSON

```
{
    "type":"Topology",
    "transform":{
        "scale":[0.0036525732585262097,0.0025214013656051654],
        "translate":[-0.305134937527328,42.571794542384225]
    },
    "arcs":[[[507,974],[69,-38],[71,17],[34,-136],[75,12],[74,108],[55,-66],[0,-31],[38,-50],[20,-57],[-4,-62
],[34,-29],[16,-48],[37,-27],[-107,-119],[-76,-33],[-36,6],[4,-68],[-25,-15],[-83,7],[-63,-8],[-79,-37],[-7,-
57],[64,-35],[7,-92],[30,-50],[-41,-66],[-129,28],[-109,53],[-113,25],[-13,-60],[-91,-8],[-29,23],[-57,-16],[
-73,58],[4,62],[57,73],[29,66],[-19,73],[-61,40],[20,46],[8,76],[43,-3],[165,59],[18,-14],[68,65],[39,66],[-1
0,72],[75,67],[71,123]]],
    "objects":{
        "example":{
            "type":"GeometryCollection",
            "geometries":[
                {
                    "arcs":[[0]],
                    "type":"Polygon",
                    "properties":{
                        "adm1_code":"FRA-2669",
                        "name":"Midi-Pyrénées"
                    }
                }
            ]
        }
    }
}
```

In Example 1.1, all data related to the polygon feature is stored in a single `"Feature"` object within the `"features"` array (lines 4-14). In Example 1.2, identifying data related to the polygon feature is stored in a `"Polygon"` object within the '"geometries"' array (lines 12-19), while the `"arcs"` used by that feature are stored in a separate array that contains other arrays with integers (line 7). Since each decimal must be stored in the computer's memory as an 8-bit character, storing integers rather than float values cuts way down on the file size, in addition to the reduction achieved by eliminating line duplication.
The `"transform"` (Example 1.2 lines 3-6)—like the information stored in the .prj file of a shapefile—is a mathematical function applied to each integer to turn it into a geographic coordinate. The coordinate reference system used as a default is EPSG:4326/WGS 84 (to understand why, review Pre-Module).

While Example 1.2 appears to contain more lines of code, keep in mind that the line breaks were added to improve human readability. If you were to save each example as a separate file, you would discover that the size of the GeoJSON file is 2.03 KB, whereas the TopoJSON is only 771 bytes—less than half the GeoJSON size, even without any shared feature edges to eliminate.

1.2 Using Mapshaper to Simplify and Convert Spatial Data

The major downside to using TopoJSON is that (as of this writing) it is not yet supported by any major desktop software. This makes converting data to TopoJSON a bit tricky. There is a **command-line tool (https://github.com/topojson/topojson/blob/master/README.md#command-line-reference)** available, but it can be difficult to install and work with. Fortunately, there are now at least two Web applications that can do the work for us. You already know about one of them, **geojson.io (http://geojson.io/)**, from Pre-Module part 3, Lesson 1. In this tutorial, we will make use of the other one, **mapshaper (http://mapshaper.org/)**.

Mapshaper is the brainchild of New York Times Graphics Editor and UW-Madison alumn Matthew Bloch. The feature that makes this tool particularly useful for our purposes is its ability to simplify overly-complex geometry. For the D3 lab assignment, you will want to balance keeping your geographic areas recognizable with minimizing the data size to maximize the speed of drawing and interaction in the browser. This tradeoff will almost certainly require simplifying your chosen spatial data, and mapshaper does this beautifully.

Before you create your TopoJSON, you will need to have gathered the spatial dataset(s) you will use for your D3 lab assignment. This tutorial will make use of the *EuropeCountries* and *FranceRegions*files from the lab assignment handout. You should replace these with your own chosen spatial datasets. A good source for global- and country-scale data is **Natural Earth (http://www.naturalearthdata.com/)**, but a shapefile or GeoJSON from any reliable source should work. Make sure you have assigned EPSG:4326 (WGS 84) as the coordinate reference system (we recommend using QGIS to complete these tasks). You should edit the attribute table of the spatial dataset to include only the field you will use to join the spatial data to your multivariate CSV data.

> **Practice!** Create a directory for your D3 lab assignment website and name it *d3-coordinated-viz.* Create a Git repository for it and sync with GitHub. Add your spatial datasets and multivariate attribute CSV to the *data* folder. Add D3 to your *lib* folder. Create a boilerplate HTML file with script links to *d3.js* and *main.js.* Create blank *main.js* and *style.css* files within the appropriate folders.

The next step is to navigate to **mapshaper.org (http://mapshaper.org/)** and import your spatial dataset(s). This is fairly self-explanatory; simply drag-and-drop each GeoJSON or zipped shapefile into the browser. Your data should appear immediately as linework against a blank backdrop. If it does not appear, go back to your GIS software and check that you have removed any projection by assigning EPSG:4326 as the CRS. Once you see your data, you can zoom and pan until it is positioned the way you want it. Then, click "Simplify" in the upper-right-hand corner of the web page. You will be presented with a choice of three simplification methods; which you choose does not really matter. Click "Next" and then use the slider at the top of the page to simplify the linework (Figure 1.1). When you are satisfied with the appearance of the linework (use your cartographic judgement!), Click on "Export" in the upper-right corner, then click "TopoJSON". Save the file in the *data* folder of your *d3-coordinated-viz* website directory and change the file extension from *.json* to *.topojson.*

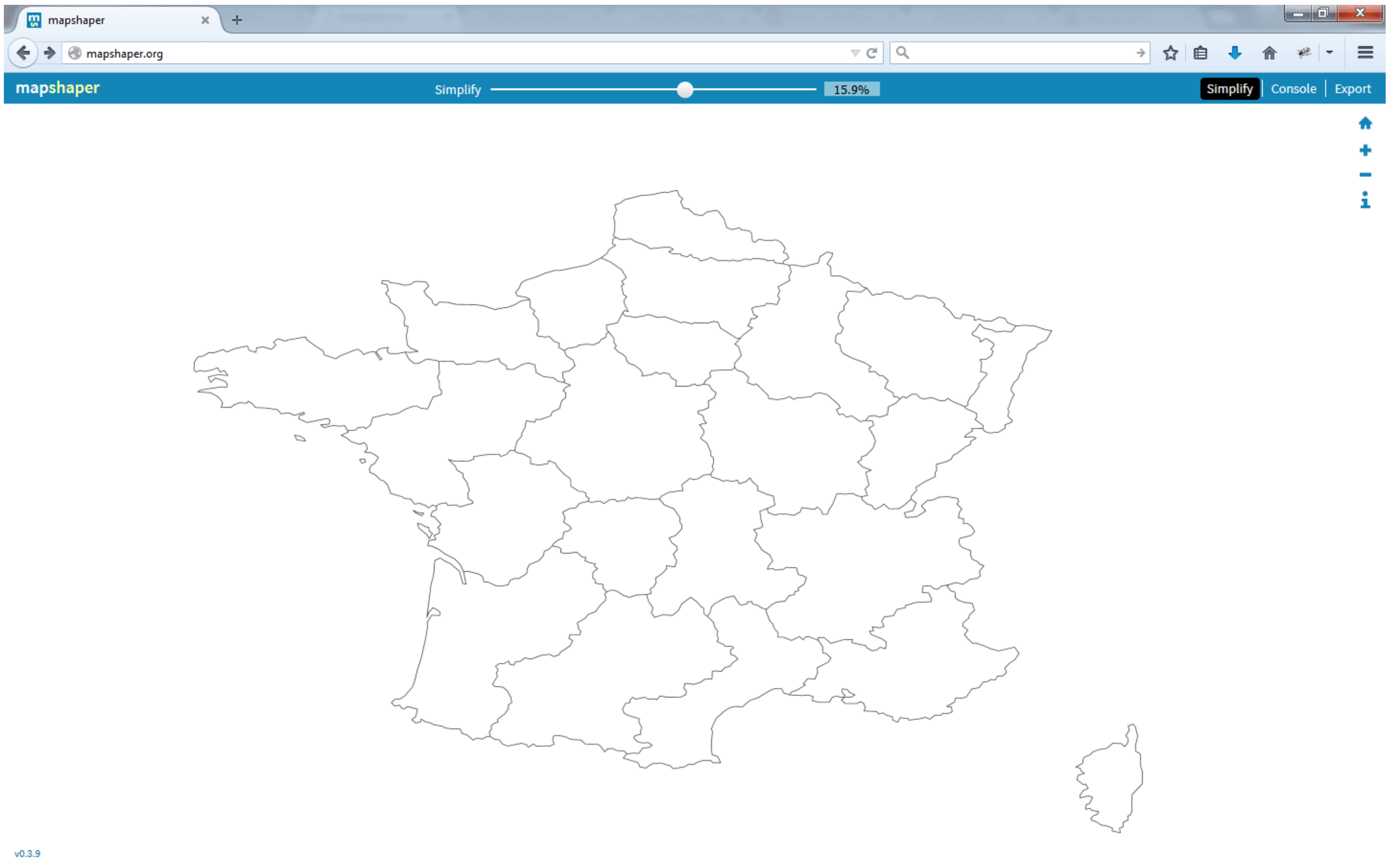
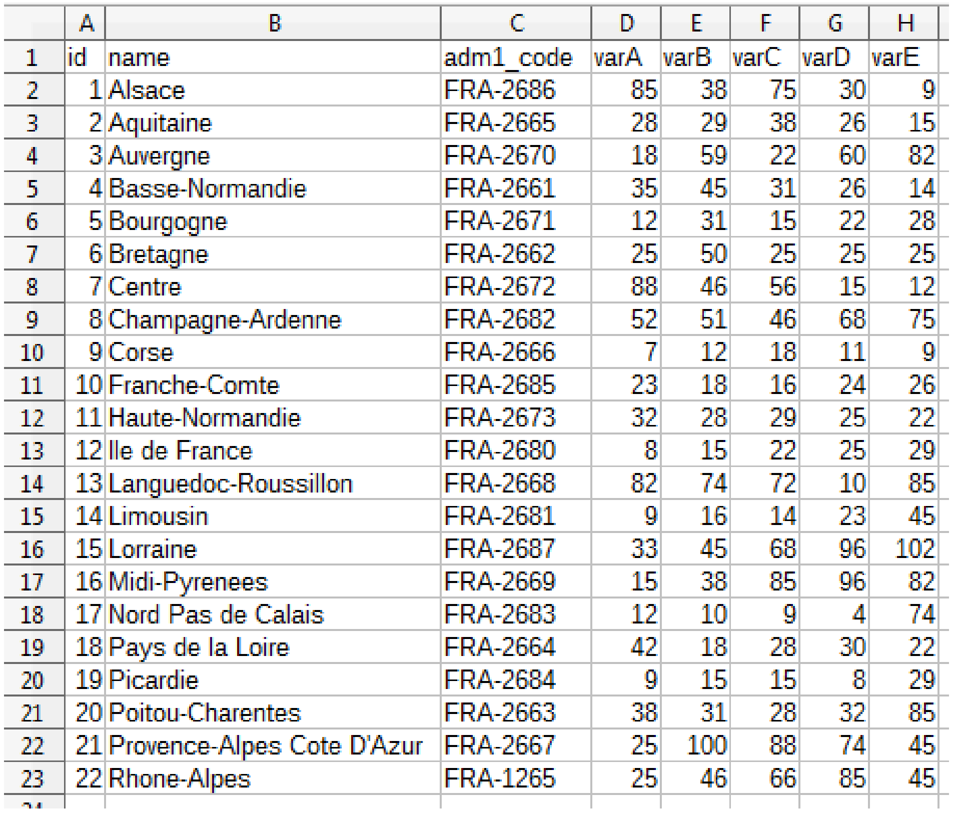


Figure 1.1: Simplifying spatial data in mapshaper

> **Practice!** Simplify your spatial data and convert it to TopoJSON format using mapshaper.

1.3 Using d3.queue to Load Data into the DOM

At this point, you should have at least one TopoJSON file for your spatial data (this tutorial uses two) and one CSV file for your attribute data. The attribute CSV should be a table of geographic features that includes an identifying attribute shared with the spatial data (Figure 1.2, column C) and at least five quantitative attributes that are of interest to you (columns D-H). Replace the dummy data in Figure 1.2 with your own chosen dataset.



	A	B	C	D	E	F	G	H
1	id	name	adm1_code	varA	varB	varC	varD	varE
2	1	Alsace	FRA-2686	85	38	75	30	9
3	2	Aquitaine	FRA-2665	28	29	38	26	15
4	3	Auvergne	FRA-2670	18	59	22	60	82
5	4	Basse-Normandie	FRA-2661	35	45	31	26	14
6	5	Bourgogne	FRA-2671	12	31	15	22	28
7	6	Bretagne	FRA-2662	25	50	25	25	25
8	7	Centre	FRA-2672	88	46	56	15	12
9	8	Champagne-Ardenne	FRA-2682	52	51	46	68	75
10	9	Corse	FRA-2666	7	12	18	11	9
11	10	Franche-Comte	FRA-2685	23	18	16	24	26
12	11	Haute-Normandie	FRA-2673	32	28	29	25	22
13	12	Ile de France	FRA-2680	8	15	22	25	29
14	13	Languedoc-Roussillon	FRA-2668	82	74	72	10	85
15	14	Limousin	FRA-2681	9	16	14	23	45
16	15	Lorraine	FRA-2687	33	45	68	96	102
17	16	Midi-Pyrenees	FRA-2669	15	38	85	96	82
18	17	Nord Pas de Calais	FRA-2683	12	10	9	4	74
19	18	Pays de la Loire	FRA-2664	42	18	28	30	22
20	19	Picardie	FRA-2684	9	15	15	8	29
21	20	Poitou-Charentes	FRA-2663	38	31	28	32	85
22	21	Provence-Alpes Cote D'Azur	FRA-2667	25	100	88	74	45
23	22	Rhone-Alpes	FRA-1265	25	46	66	85	45

Figure 1.2: an example multivariate dataset

The next task is to load *all* of our data files into the DOM. It might help you at this point to review the AJAX concepts in Pre-Module. In particular, think about how AJAX callbacks work: after each data file is loaded, the data is passed to a function. It can only be accessed within that function because it is loaded asynchronously with the rest of the script. But what if you want to access data from multiple external files (data streams)? You could load the files in **series**, calling an AJAX method for the third file within the callback of the second file and calling the AJAX method for the second file within the callback of the first—in essence, nesting the callback functions and accessing the data from the innermost callback. However, it quickly becomes unweildy to keep track of which callback you are writing script inside of, making this a less than ideal solution.

It turns out there is a simpler *and* more processor-efficient way to load multiple data streams in JavaScript. **D3 queue** **(https://github.com/d3/d3-queue)** allows you to conduct multiple AJAX calls in **parallel**, or at the same time rather than one after another. Once all of the data has loaded, it fires a single callback function to which it passes each of the data streams.

> **Practice!** Follow the instructions on the *queue.js* **GitHub site** **(https://github.com/d3/d3-queue#installing)** to **download queue.js** **(https://d3js.org/d3-queue.v2.min.js)** and place it in your *lib* folder (as of this writing, the current version is *d3-queue.v2.min.js*). Add a script link to *index.html* so you have access to the library.

Mike Bostock initially wrote Queue as its own separate library (queue.js), but it's now integrated into D3, so we do not need to load any additional tools into the DOM. To use queue, in main.js we will write a single, unnamed D3-style block starting with a call to the *d3.queue()* method:

Example 1.3: loading data streams with queue in *main.js*

JavaScript

```
//begin script when window loads
window.onload = setMap();

//set up choropleth map
function setMap(){
    //use queue to parallelize asynchronous data loading
    d3.queue()
        .defer(d3.csv, "data/unitsData.csv") //load attributes from csv
        .defer(d3.json, "data/EuropeCountries.topojson") //load background spatial data
        .defer(d3.json, "data/FranceProvinces.topojson") //load choropleth spatial data
        .await(callback);
};
```

In Example 1.3, after the call to `d3.queue()` on line 7, we use three `.defer()` operators, one for each dataset (lines 8-10). The block ends with an `.await()` operator that fires when all of the data has loaded and sends the loaded data to the callback function (line 11). Each `.defer()` operator takes two parameters: the AJAX method used to load the data and the URL of the data. The methods `d3.csv()` (https://github.com/d3/d3-request/blob/master/README.md#csv) and `d3.json()` (https://github.com/d3/d3-request/blob/master/README.md#json) are AJAX methods similar to `$.ajax()` and `$.getJSON()` in jQuery. D3 provides many convenient **AJAX methods** (https://github.com/d3/d3-request/blob/master/README.md). These would normally be called with a URL and callback as their own parameters, but `d3.queue().defer()` only uses the AJAX method name and takes care of the rest.

Once we have set up our `d3.queue()` block, we can write the callback function. We will place this function within `setMap()` so that it can make use of variables that we will add above it later on.

Example 1.4: adding a callback to `setMap()` in *main.js*

JavaScript

```
//Example 1.3 line 4...set up choropleth map
function setMap(){
    //use d3.queue to parallelize asynchronous data loading
    d3.queue()
        .defer(d3.csv, "data/unitsData.csv") //load attributes from csv
        .defer(d3.json, "data/EuropeCountries.topojson") //load background spatial data
        .defer(d3.json, "data/FranceProvinces.topojson") //load choropleth spatial data
        .await(callback);

    function callback(error, csvData, europe, france){
        console.log(error);
        console.log(csvData);
        console.log(europe);
        console.log(france);
    };
};
```

The callback function takes four parameters: an error report if any errors occur and the three datasets. The `console.log()` statements print the results to separate lines of the Console. As you can see in Figure 1.3, `d3.csv()` automatically formats the imported CSV data as an array, and `d3.json()` formats the spatial data as an object:

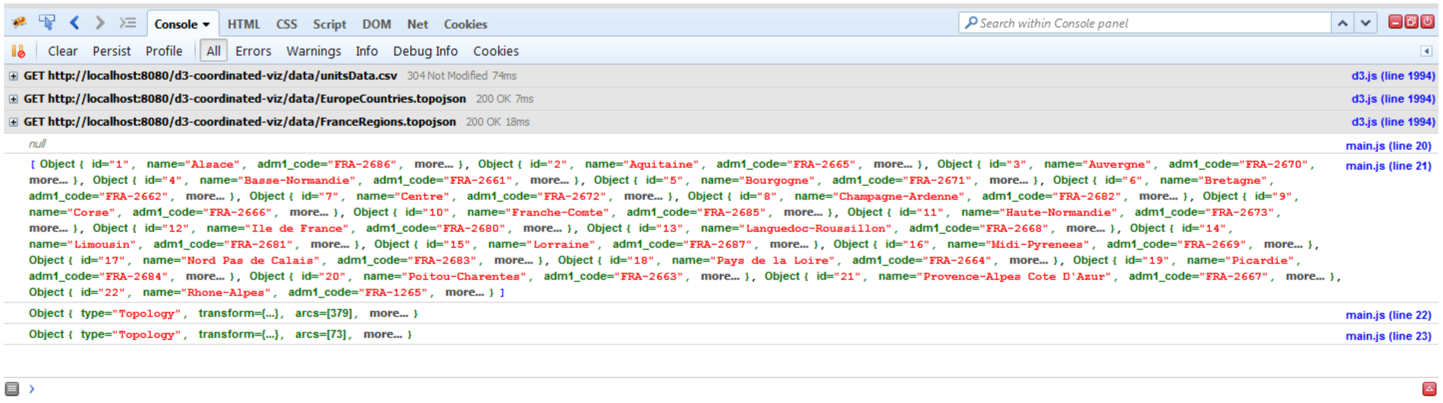

```
GET http://localhost:8080/d3-coordinated-viz/data/unitsData.csv 304 Not Modified 74ms          d3.js (line 1994)
GET http://localhost:8080/d3-coordinated-viz/data/EuropeCountries.topojson 200 OK 7ms          d3.js (line 1994)
GET http://localhost:8080/d3-coordinated-viz/data/FranceRegions.topojson 200 OK 18ms           d3.js (line 1994)
null                                                                                           main.js (line 20)
[ Object { id="1", name="Alsace", adm1_code="FRA-2686", more... }, Object { id="2", name="Aquitaine", adm1_code="FRA-2665", more... }, Object { id="3", name="Auvergne", adm1_code="FRA-2670",     main.js (line 21)
more... }, Object { id="4", name="Basse-Normandie", adm1_code="FRA-2661", more... }, Object { id="5", name="Bourgogne", adm1_code="FRA-2671", more... }, Object { id="6", name="Bretagne",
adm1_code="FRA-2662", more... }, Object { id="7", name="Centre", adm1_code="FRA-2672", more... }, Object { id="8", name="Champagne-Ardenne", adm1_code="FRA-2682", more... }, Object { id="9",
name="Corse", adm1_code="FRA-2666", more... }, Object { id="10", name="Franche-Comte", adm1_code="FRA-2685", more... }, Object { id="11", name="Haute-Normandie", adm1_code="FRA-2673",
more... }, Object { id="12", name="Ile de France", adm1_code="FRA-2680", more... }, Object { id="13", name="Languedoc-Roussillon", adm1_code="FRA-2668", more... }, Object { id="14",
name="Limousin", adm1_code="FRA-2681", more... }, Object { id="15", name="Lorraine", adm1_code="FRA-2687", more... }, Object { id="16", name="Midi-Pyrenees", adm1_code="FRA-2669", more... },
Object { id="17", name="Nord Pas de Calais", adm1_code="FRA-2683", more... }, Object { id="18", name="Pays de la Loire", adm1_code="FRA-2664", more... }, Object { id="19", name="Picardie",
adm1_code="FRA-2684", more... }, Object { id="20", name="Poitou-Charentes", adm1_code="FRA-2663", more... }, Object { id="21", name="Provence-Alpes Cote D'Azur", adm1_code="FRA-2667", more... },
Object { id="22", name="Rhone-Alpes", adm1_code="FRA-1265", more... } ]
Object { type="Topology", transform={...}, arcs=[379], more... }                              main.js (line 22)
Object { type="Topology", transform={...}, arcs=[73], more... }                               main.js (line 23)
```

Figure 1.3: results of d3.queue() AJAX calls

The Firebug Console also conveniently displays each AJAX call on the first three lines. Clicking on these lines will display more information about the parameters sent to the server as well as the data received, displayed as plain text.

1.4 Using Topojson.js to Translate TopoJSON

D3, Leaflet, and other web mapping libraries do not natively support TopoJSON data. Rather, to use our spatial data, we need to convert it *back* to GeoJSON within the DOM. For this, we will use Mike Bostock's small `topojson.js` (https://github.com/mbostock/topojson) library.

> **Practice!** Download topojson.js from the link above and place it in your *lib* folder. Add a script link in *index.html*.

As the topojson.js **API Reference** (https://github.com/mbostock/topojson/wiki/API-Reference) explains, to translate to GeoJSON, we will use the `topojson.feature()` method, passing it the variable with our TopoJSON data and the object within that data that we want to convert:

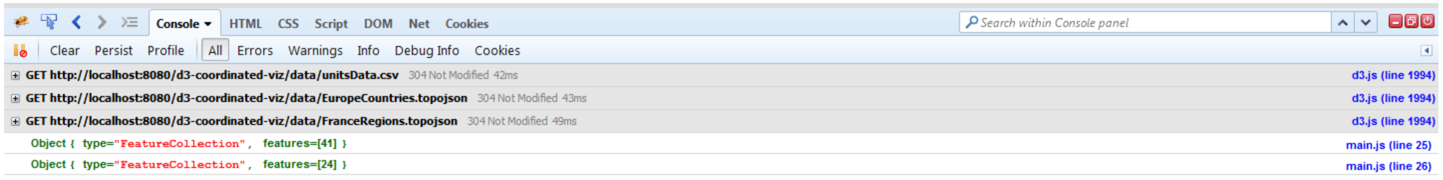
Example 1.5: converting TopoJSON to GeoJSON in *main.js*

JavaScript

```
    //Example 1.4 line 10
    function callback(error, csvData, europe, france){
        //translate europe TopoJSON
        var europeCountries = topojson.feature(europe, europe.objects.EuropeCountries),
            franceRegions = topojson.feature(france, france.objects.FranceRegions).features;

        //examine the results
        console.log(europeCountries);
        console.log(franceRegions);
    };
```

In Example 1.5, each TopoJSON object is passed as the first parameter to `topojson.feature()`. The second parameter is the object that holds the details unique to each dataset. In Example 1.2 (line 9), this object was named `"example"`; for our tutorial spatial data, it retains the name of the original file that was passed through mapshaper. Once the data has been translated and assigned to variables, we can examine those variables in the Console and see that they are now GeoJSON `FeatureCollection`s:



```
GET http://localhost:8080/d3-coordinated-viz/data/unitsData.csv 304 Not Modified 42ms                 d3.js (line 1994)
GET http://localhost:8080/d3-coordinated-viz/data/EuropeCountries.topojson 304 Not Modified 43ms      d3.js (line 1994)
GET http://localhost:8080/d3-coordinated-viz/data/FranceRegions.topojson 304 Not Modified 49ms        d3.js (line 1994)
  Object { type="FeatureCollection", features=[41] }                                                  main.js (line 25)
  Object { type="FeatureCollection", features=[24] }                                                  main.js (line 26)
```

Figure 1.4: GeoJSON data created in the DOM by topojson.js

> **Practice!** In *main.js*, use *queue.js* to load your TopoJSON and CSV data into the DOM. Use *topojson.js* to translate the imported data into GeoJSON format.

> **Self-Check:**
> 1. **Which of the following is *not* an advantage of TopoJSON over GeoJSON?**
>   a. it preserves topology, maintaining data integrity when the data is edited (such as by mapshaper)
>   b. it drastically reduces the file size by eliminating duplication and using integers instead of float values for arc coordinates
>   c. its reduced file size allows data to be drawn in the browser more quickly and improves the speed of interactions
>   d. it can be passed directly to D3 or Leaflet without requiring script to change formats in the DOM
> 2. **True/False:** Queue improves the processing efficiency of AJAX by loading each file in series (one after another) rather than in parallel (at the same time).