CHAPTER 2

Data

Data visualization would not be possible without data. In this chapter, we will explore how to load, parse, and transform various types of data. Most datasets used in data visualization are structured as tables, with rows representing individual data points and columns representing different attributes of those data points. This is known as "tabular data". Other types of data include trees, graphs, and geospatial data.

One of the most common formats of data is the comma separated values (CSV) file. We will start by loading and parsing CSV files in various ways. We will then explore how to load and parse other types of data, such as trees and graphs represented as JSON, and geospatial data represented as GeoJSON or TopoJSON. Finally, we will discuss how to transform data using sorting, filtering and binned aggregation to prepare it for visualization.

Note that it may make sense to use technology other than JavaScript for data processing, especially for large datasets. For example, Python is a popular choice for data processing, and there are many libraries available for data manipulation, such as Pandas. In many real-world applications, there is a dedicated data pipeline that may consist of various database and analytics technologies. However, for the purposes of this book, we will focus on using JavaScript and D3 for data loading and processing.

2.1 CSV: COMMA SEPARATED VALUES

One of the most common formats of data is the Comma Separated Values (CSV) format. We will start by loading and parsing CSV files in various ways. One way to load a CSV file is to leverage special "loaders" if you are using a modern build tool such as Rollup or Webpack. These loaders can be configured to automatically load and parse CSV files as JavaScript objects. Another way is to use the Fetch API to load the CSV file asynchronously and then parse it using D3. We will explore both methods in this chapter.

In order to get familiar with the concept of CSV data and how it maps to JavaScript objects, let's start by modifying our previous Circles With D3 example to represent the data as CSV instead of an array of objects. Even though this particular example with circles does not depend on the data being in a CSV file,

it's a nice way of seeing clearly the correspondence between arrays of objects and tabular data.

Various spreadsheet programs like Excel and Google Sheets can import and export data in CSV format. This makes it easy to "hand craft" data in a spreadsheet program and then export it as a CSV file for visualization. If you were to formulate the data for our circles examples as a data table in Google Sheets, it would look like this:

	A	B	C	D
1	x	y	r	fill
2	155	382	20	#D4089D
3	340	238	52	#FF0AAE
4	531	59	20	#00FF88
5	482	275	147	#7300FF
6	781	303	61	#0FFB33
7	668	229	64	#D400FF
8	316	396	85	#0FF0FF

Figure 2.1 Data as Spreadsheet

If you were to export this data as a CSV file, it would look like this:

importing-csv/data.csv

```
x,y,r,fill
155,382,20,#D4089D
340,238,52,#FF0AAE
531,59,20,#00FF88
482,275,147,#7300FF
781,303,61,#0FFB33
668,229,64,#D400FF
316,396,85,#0FF0FF
```

Figure 2.2 Data as CSV

Note that with CSV format, there is no information about the types of each column. Everything is represented as a string. This is why we need to convert the `x`, `y`, and `r` properties to numbers in the JavaScript code after this data is imported or loaded.

Importing CSV

In this example, we import the data from a CSV file called `data.csv`. This technique is possible because we are using a modern build tool that supports importing from CSV files. If you are using Rollup or Webpack, there are plugins available that enable use of this `import` syntax with CSV files (see NPM packages

`@rollup/plugin-dsv` and `csv-loader`). The data is read as an array of objects, where each object represents a row in the CSV file.

There are pros and cons to this approach. The main advantage is that the data is loaded and parsed automatically, and the code is more concise. We don't need to think at all about asynchronous control flow or hosting the data, because it is bundled with the JavaScript code at build time. For small datasets, this can work well, but as the data size gets larger, this approach does not work well because it makes the build too large. This approach also does not work well for dynamic data, such as from an API. We start here because it is conceptually simpler, but we will explore more flexible and scalable approaches in the next section.

importing-csv/data.js

```
import data from './data.csv';
for (const d of data) {
  d.x = +d.x;
  d.y = +d.y;
  d.r = +d.r;
}
export { data };
```

Figure 2.3 Importing from a CSV File

In this example, we use the `import` syntax to import the data from the `data.csv` file, which is imported as an array of objects where each value is a string. We then loop over each data point and convert the `x`, `y`, and `r` properties from strings to numbers using the unidirectional plus operator (a commonly used alternative to `parseFloat`). This is necessary because the data is read as strings from the CSV file, but in the visualization logic it's best to represent values as numbers. Finally, we export the data as a named export. This allows us to import the data in other files using the same exact import statements we had before.

2.1.1 Fetching CSV Asynchronously

Typically, data for visualization is not bundled with the JavaScript and is instead hosted separately, either as a static file or as an API endpoint. Either way, we can make an HTTP (HyperText Transfer Protocol) request to retreive the data. Since this network request may take some time to come back with the data, we need to deal with asynchronous control flow.

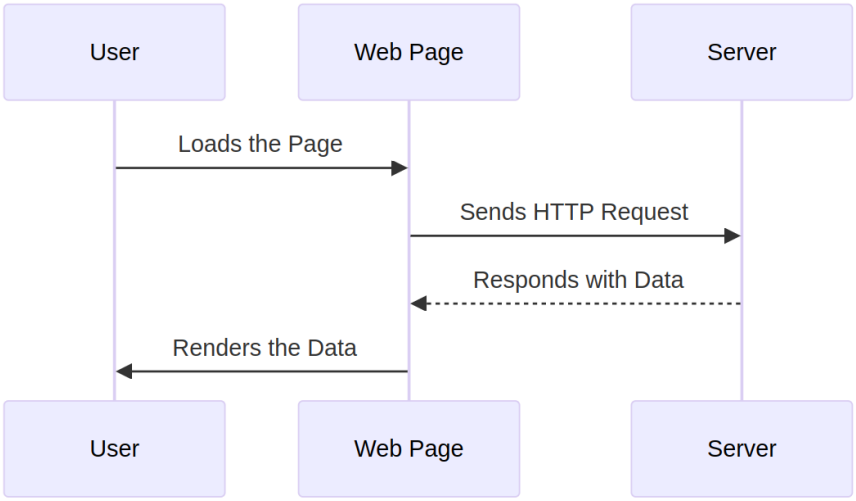

Figure 2.4 Control Flow of an Asynchronous Request

Asynchronous control flow is a way of dealing with control flow (the order in which parts of the program run) such that it accounts for things that you need to wait for, such as network requests. In JavaScript, there is a mechanism for this called `Promises` that helps with this. There is also a special syntax for working with promises called `async` and `await`. The `async` keyword is used to define a function that returns a promise, and the `await` keyword is used to pause the execution of the function until the promise is resolved.

fetching-csv/fetchAndParseData.js

```
import { csv } from 'd3';
const url =
  'https://gist.githubusercontent.com/curran/a08a1080b88344b0
  '/raw/0e7a9b0a5d22642a06d3d5b9bcbad9890c8ee534/iris.csv';
export const fetchAndParseData = async () =>
  await csv(url, (d) => {
    d.sepal_length = +d.sepal_length;
    d.sepal_width = +d.sepal_width;
    d.petal_length = +d.petal_length;
    d.petal_width = +d.petal_width;
    return d;
  });
```

Figure 2.5 Fetching and Parsing Data

In this example, we define an asynchronous function called `fetchAndParseData` that uses the `csv` function from D3 to fetch and parse data from a URL. The URL

points to a CSV file hosted on GitHub, which is a great place to host public data. The `csv` function takes two arguments: the URL of the CSV file and a function that parses each row of the CSV file. The `csv` function returns a promise that resolves to the parsed data. We use the `await` keyword to pause the execution of the function until the promise is resolved, and then return the parsed data. This is how you can fetch and parse data asynchronously.

fetching-csv/useData.js

```
import { fetchAndParseData } from './fetchAndParseData';
export const useData = ({ state, setState }) => {
  const data = state.data;
  const setData = (data) => {
    setState((state) => ({ ...state, data }));
  };
  if (!data) {
    fetchAndParseData().then(setData);
  }
  return data;
};
```

Figure 2.6 Using Data

Now that we have a function that fetches and parses data, we can integrate that into our unidirectional data flow paragm for later use in the rendering logic. To do this, we define a function called `useData` that takes the current state and a function to set the state as arguments. This function returns the data from the state, and if the data is not already in the state, it fetches and parses the data using the `fetchAndParseData` function and sets the state with the parsed data. This is how you can use asynchronous data loading in a unidirectional data flow paradigm.

fetching-csv/index.js

```
import { select } from 'd3';
import { useData } from './useData';
export const main = (container, { state, setState }) => {
  const data = useData({ state, setState });
  if (!data) return;
  select(container)
    .selectAll('pre')
    .data(data)
    .join('pre')
    .style('font-size', '16px')
    .text((d) => JSON.stringify(d));
};
```

Figure 2.7 Loading and Displaying Data

Finally, pulling it all together, we define our `main` entry point. We use the `useData` function to get the data, and if the data is not yet loaded, we return early. After the data loads and `setState` is invoked, then the `main` function runs again, and this time `data` is defined. We then use D3 render one `pre` element for each row of data, setting the text content of each `pre` element to a JSON string representation of the data. This is how you can load, parse, and render data asynchronously with unidirectional data flow and D3.