JavaScript Data Visualization

readbytes

JavaScript Data Visualization

From Basics to Beautiful Charts

readbytes.github.io

2025-07-16

This page is intentionally left blank.

Contents

Chapter 1.

Fundamentals of Data Visualization

1. Principles of Good Visualization
2. JavaScript and the Web Platform

1 Fundamentals of Data Visualization

1.1 Principles of Good Visualization

Data visualization is both an art and a science. The goal is to communicate information clearly and effectively through graphical means. When done well, visualization reveals insights, supports decision-making, and tells compelling stories from complex data. But to achieve this, certain core principles must guide your design choices.

In this section, we will explore the fundamental principles that make a visualization not only beautiful but also meaningful: **clarity, simplicity, integrity, and context**. We will also look at how to select the right chart type, avoid common pitfalls, and design visuals that facilitate correct interpretation.

1.1.1 Clarity: Make Your Message Clear

The primary purpose of any data visualization is to communicate data clearly. Visual clutter, ambiguous symbols, or complicated designs can confuse viewers, obscuring the message rather than illuminating it.

- **Avoid chartjunk**: Coined by Edward Tufte, *chartjunk* refers to unnecessary or distracting decorations like 3D effects, excessive gridlines, or fanciful backgrounds that do not add informational value. These elements consume visual attention but do not help in understanding the data.
- **Use legible labels and titles**: Axes, legends, and titles should be clearly labeled and easy to read. The viewer should understand what each element of the chart represents without guessing.
- **Consistent scales and units**: Ensure that scales on axes are uniform and logical. Distorted scales can mislead the viewer.

1.1.2 Simplicity: Less Is More

Simplicity aids comprehension. Strive to reduce complexity without losing important information.

- **Limit the number of colors and elements**: Using too many colors or data series can overwhelm the viewer. Choose a color palette that supports grouping and comparison.
- **Focus on the key message**: Remove anything that doesn't support the story or analysis you want to highlight.

1.1.3 Integrity: Represent Data Honestly

Your visualization must present data truthfully and avoid misleading the viewer.

- **Start axes at zero when appropriate**: Bar charts especially should have a baseline at zero to avoid exaggerating differences. For example, truncating the y-axis can make small differences appear huge.
- **Choose the right chart type**: Some chart types are better at representing certain data types. For example, using a pie chart to compare many categories is misleading because it becomes hard to discern slice sizes.

1.1.4 Context: Help Viewers Understand the Data

Providing context allows the audience to interpret the data correctly.

- **Provide clear titles and annotations**: A title explains what the visualization shows, while annotations can highlight key points or trends.
- **Add reference points**: For example, a line indicating a target or average helps viewers evaluate performance at a glance.
- **Explain units and time frames**: When dealing with time series or quantitative data, include units (e.g., dollars, percentage) and relevant time periods.

1.1.5 Choosing the Right Chart Type

Different chart types serve different purposes and data types. Here are common examples with guidance on when to use each:

Chart Type	Purpose	When to Use	When to Avoid
Bar Chart	Comparing quantities across categories	When comparing discrete categories or showing changes over time (with time on x-axis)	Avoid for part-to-whole relationships
Pie Chart	Showing proportions of a whole	When you have a small number (3-5) of categories that sum to 100%	Avoid with many slices or close percentages
Line Chart	Showing trends over time	When you want to show continuous data or changes over time	Avoid if data is categorical

Chart Type	Purpose	When to Use	When to Avoid
Scatter Plot	Showing relationships between two variables	To identify correlations or clusters	Avoid if data is categorical or too dense
Histogram	Showing distribution of numerical data	To visualize frequency distributions	Avoid for categorical data

1.1.6 Example: Pie Chart vs Bar Chart

Imagine you want to show the market share of four smartphone brands.

- **Pie Chart**: Displays slices of a circle, with each slice proportional to the market share. Good for illustrating parts of a whole if categories are few and clearly distinct.
- **Bar Chart**: Displays bars representing market share values. Better if you want to compare values side by side and include additional details like exact percentages.

In this case, the bar chart is often more precise because viewers can compare bar heights more accurately than estimating pie slice angles.

1.1.7 Common Visualization Pitfalls and How to Avoid Them

Using 3D Effects

3D charts often distort data perception. For example, a 3D pie chart tilts slices, making some appear larger than others despite equal values.

Better: Use 2D charts for accurate visual comparison.

Overloading With Too Much Data

Too many lines or bars can make a chart unreadable.

Better: Break complex data into multiple simpler charts or use interactive filters.

Misleading Axis Scales

Non-zero baselines can exaggerate differences, creating false impressions.

Better: Start axes at zero unless you explicitly want to highlight small changes, and always clarify if not.

Ignoring Color Blindness

Color choices may be indistinguishable to color-blind viewers.

Better: Use color palettes tested for accessibility and combine color with shape or pattern coding.

1.2 JavaScript and the Web Platform

In today's world, data visualization thrives on the web. The web platform provides a powerful and accessible environment for creating dynamic, interactive charts that can reach anyone with a browser. At the heart of this capability is **JavaScript**, the programming language that powers interaction and animation in web browsers.

In this section, we'll explore how JavaScript works with core web technologies — HTML, SVG, and the Document Object Model (DOM) — to bring data visualizations to life. We'll also touch on browser rendering, performance considerations, and the important role of visualization libraries like D3.js.

1.2.1 JavaScript: The Engine Behind Dynamic Visualizations

JavaScript is the language that runs in your web browser, enabling you to manipulate webpage content in real time. When you create a data visualization on the web, JavaScript is responsible for:

- Fetching and processing data,
- Creating graphical elements,
- Updating visuals based on user interaction,
- Animating transitions and effects.

Because JavaScript runs directly inside the browser, it allows your visualizations to be interactive and responsive, adapting instantly as users explore the data.

1.2.2 Understanding the Document Object Model (DOM)

The **DOM** is the browser's internal representation of the webpage's structure. It models the page as a tree of nodes, each representing elements like paragraphs, images, buttons, or charts.

- When your HTML page loads, the browser parses the HTML markup and builds the DOM.

- JavaScript can access and modify this DOM tree using APIs, allowing you to add, remove, or change elements dynamically.
- For example, to create a bar chart, JavaScript can create rectangles (`<rect>` elements) inside an SVG container, setting their size and color based on data.

This dynamic manipulation of the DOM is fundamental to interactive data visualization.

1.2.3 HTML and SVG: The Building Blocks of Visualization

HTML (HyperText Markup Language)

HTML defines the structure of your web page, including where your visualizations live. While HTML is great for text and layout, it is limited in creating scalable and complex graphics directly.

SVG (Scalable Vector Graphics)

SVG is an XML-based markup language designed specifically for describing 2D graphics. Unlike bitmap images, SVG graphics are vector-based, meaning they can scale smoothly without loss of quality.

- SVG elements include shapes like rectangles, circles, lines, paths, and text.
- Because SVG is part of the DOM, JavaScript can interact with each element individually — modifying attributes like position, size, color, and opacity.

Together, HTML and SVG allow you to create rich, interactive charts directly in the browser.

1.2.4 Browser Rendering and Performance Considerations

When you manipulate DOM or SVG elements, the browser must re-render the page to reflect those changes. While modern browsers are very efficient, excessive DOM manipulations or very large datasets can lead to performance bottlenecks, causing slow rendering or janky interactions.

To keep visualizations smooth:

- Minimize the number of DOM updates by batching changes.
- Use efficient data structures and algorithms for processing.
- Consider using **Canvas** (a raster-based drawing surface) for very large or highly dynamic datasets, though Canvas trades off direct DOM interaction for raw performance.

1.2.5 Enter Visualization Libraries: D3.js and Beyond

While you can use vanilla JavaScript to create visualizations, managing the DOM and SVG directly can quickly become complex, especially for large or interactive charts. This is where libraries like **D3.js** come in.

- **D3.js** (Data-Driven Documents) provides powerful tools to bind data directly to DOM elements.
- It abstracts many low-level operations, enabling you to write declarative code that describes *what* should appear rather than *how* to manipulate every element.
- D3 also offers utilities for scales, axes, layouts, and animations, accelerating the development of sophisticated visualizations.

Using libraries like D3, you can focus more on the data and design aspects, while the library efficiently handles the rendering and updating of the graphical elements.

Chapter 2.

Working with Data

1. Loading and Parsing Data
2. Data Preparation

2 Working with Data

2.1 Loading and Parsing Data

Before you can create meaningful visualizations, you need to get your data into your JavaScript program. In the web environment, data often comes from external sources such as CSV files, JSON APIs, or databases. This section explains how to **load external data asynchronously** using JavaScript, how to parse it into usable formats, and why careful inspection and cleaning of raw data is essential.

2.1.1 Loading Data Asynchronously with JavaScript

Web browsers load external resources asynchronously to keep applications responsive. This means your JavaScript code requests data, then continues running while waiting for the data to arrive.

2.1.2 Using `fetch()`

The modern JavaScript API for requesting external data is the `fetch()` function, which returns a **Promise** — an object representing the eventual completion (or failure) of the asynchronous operation.

Example: Loading JSON data with `fetch()`:

```
fetch('data/sample.json')
  .then(response => response.json())  // Parse response as JSON
  .then(data => {
    console.log('Loaded JSON data:', data);
    // Process your data here
  })
  .catch(error => {
    console.error('Error loading JSON:', error);
  });
```

Similarly, you can load CSV files as plain text and then parse them:

```
fetch('data/sample.csv')
  .then(response => response.text())  // Get raw text
  .then(csvText => {
    console.log('Loaded CSV text:', csvText);
    // Parse CSV text manually or with a library
  })
  .catch(error => {
    console.error('Error loading CSV:', error);
  });
```

2.1.3 Using D3.jss Data Loading Utilities

If you use D3.js, it provides convenient methods for loading and parsing common data formats:

- `d3.csv(url)`: loads a CSV file and parses each row into an object.
- `d3.json(url)`: loads a JSON file and parses it automatically.

These methods also return Promises, making them easy to integrate into asynchronous workflows.

Example: Loading a CSV file with D3:

```
d3.csv('data/sample.csv').then(data => {
  console.log('Parsed CSV data:', data);
  // Data is an array of objects, one per row
});
```

Example: Loading JSON with D3:

```
d3.json('data/sample.json').then(data => {
  console.log('Loaded JSON data:', data);
});
```

2.1.4 Parsing and Converting Data Types

Raw data often comes with all values as strings. To perform calculations or create accurate visualizations, you need to convert fields into appropriate types — numbers, dates, booleans, etc.

Consider this CSV row:

```
date,sales,profit
2023-01-01,"1000","200"
```

After loading, `sales` and `profit` will be strings `"1000"` and `"200"`. To use these numerically, convert them:

```
d3.csv('data/sample.csv', row => {
  return {
    date: new Date(row.date),          // Convert to Date object
    sales: +row.sales,                 // Convert string to number
    profit: +row.profit                // Convert string to number
  };
}).then(data => {
  console.log('Typed CSV data:', data);
});
```

The `+` operator is a shorthand for `Number()`. This step ensures the values behave correctly in calculations, scales, and other operations.

2.1.5 Understanding Promises and Asynchronous Flow

Since data loading is asynchronous, code that depends on the data must run **after** the data has finished loading. Promises enable this sequencing using `.then()` or modern `async/await` syntax.

Example using `async/await`:

```
async function loadData() {
  try {
    const data = await d3.csv('data/sample.csv', row => ({
      date: new Date(row.date),
      sales: +row.sales
    }));
    console.log('Data loaded with async/await:', data);
    // Continue processing or visualization here
  } catch (error) {
    console.error('Error loading data:', error);
  }
}
loadData();
```

This approach makes asynchronous code easier to read and maintain.

2.1.6 Inspecting and Cleaning Raw Data

Loading and parsing is just the first step. Real-world data often contains:

- Missing values,
- Inconsistent formats,
- Outliers,
- Typographical errors.

Before visualizing, inspect your data by logging it or using browser developer tools. Look for anomalies that might skew your charts or cause errors.

Example inspection tips:

- Check for `null`, `undefined`, or empty strings.
- Verify date formats are consistent.
- Confirm numerical fields contain valid numbers.

You might need to:

- Filter out incomplete or corrupted rows,
- Impute missing values,
- Normalize or transform data fields.

Proper data cleaning improves the accuracy and reliability of your visualizations.

2.1.7 Summary

- Use JavaScript's `fetch()` or D3's `d3.csv()` and `d3.json()` to load data asynchronously.
- Data loading returns Promises — use `.then()` or `async/await` to handle asynchronous flow.
- Convert raw string values to appropriate data types (numbers, dates) for meaningful analysis.
- Always inspect and clean your data before visualization to avoid misleading results.

2.2 Data Preparation

Once you have loaded and parsed your raw data, the next crucial step is **preparing it** for visualization. Raw datasets are often messy, unstructured, or too detailed to display directly. Preparing data involves **sorting, grouping, aggregating, and filtering** it into a form that reveals meaningful patterns and supports your chart design.

In this section, we'll explore how to use JavaScript's powerful array methods and D3's utilities to transform data. We will also discuss why clean, well-structured data is essential for creating clear and insightful visualizations.

2.2.1 Sorting Data

Sorting organizes data into a meaningful order. For example, you may want to sort sales figures from highest to lowest or dates chronologically.

JavaScript's `Array.prototype.sort()` method lets you specify a comparator function:

Full runnable code:

```
const salesData = [
  { category: 'Books', sales: 500 },
  { category: 'Electronics', sales: 1200 },
  { category: 'Clothing', sales: 700 }
];

// Sort descending by sales
salesData.sort((a, b) => b.sales - a.sales);

console.log(JSON.stringify(salesData,null,2));
/* Output:
[
  { category: 'Electronics', sales: 1200 },
  { category: 'Clothing', sales: 700 },
  { category: 'Books', sales: 500 }
]
```

```
*/
```

2.2.2 Filtering Data

Filtering removes unwanted data points, allowing you to focus on relevant subsets.

Example: Filter out sales below 600 units:

Full runnable code:

```
const salesData = [
  { category: 'Books', sales: 500 },
  { category: 'Electronics', sales: 1200 },
  { category: 'Clothing', sales: 700 }
];

const filteredData = salesData.filter(d => d.sales >= 600);
console.log(JSON.stringify(filteredData,null,2));
/* Output:
[
  { category: 'Electronics', sales: 1200 },
  { category: 'Clothing', sales: 700 }
]
*/
```

Filtering helps declutter visualizations by excluding noise or irrelevant entries.

2.2.3 Mapping Data

Mapping transforms each element of an array into a new form.

Example: Extract just the category names for a legend:

Full runnable code:

```
const salesData = [
  { category: 'Books', sales: 500 },
  { category: 'Electronics', sales: 1200 },
  { category: 'Clothing', sales: 700 }
];

const categories = salesData.map(d => d.category);
console.log(categories); // ['Electronics', 'Clothing', 'Books']
```

Mapping can also convert raw data values into normalized or calculated forms.

2.2.4 Grouping and Aggregating Data

Often, data must be grouped by categories or time periods and aggregated (summed, averaged, counted) to summarize it effectively.

2.2.5 Grouping and Aggregating with `reduce`

Suppose you have detailed sales records with categories and individual sales amounts:

Full runnable code:

```
const detailedSales = [
  { category: 'Books', sales: 200 },
  { category: 'Electronics', sales: 400 },
  { category: 'Books', sales: 300 },
  { category: 'Clothing', sales: 700 },
  { category: 'Electronics', sales: 800 }
];
//To calculate total sales per category:

const salesByCategory = detailedSales.reduce((acc, curr) => {
  acc[curr.category] = (acc[curr.category] || 0) + curr.sales;
  return acc;
}, {});

console.log(JSON.stringify(salesByCategory,null,2));
/* Output:
{
  Books: 500,
  Electronics: 1200,
  Clothing: 700
}
*/
```

This produces an object mapping categories to total sales.

2.2.6 Grouping with D3s `d3.group` and Aggregation

D3 provides helpful functions for grouping and aggregating:

```
const detailedSales = [
  { category: 'Books', sales: 200 },
  { category: 'Electronics', sales: 400 },
  { category: 'Books', sales: 300 },
  { category: 'Clothing', sales: 700 },
  { category: 'Electronics', sales: 800 }
];
const grouped = d3.group(detailedSales, d => d.category);
```

```
for (const [category, records] of grouped) {
  const totalSales = d3.sum(records, d => d.sales);
  console.log(`${category}: ${totalSales}`);
}
/* Output:
Books: 500
Electronics: 1200
Clothing: 700
*/
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Sales Grouping with D3</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f9f9f9;
      padding: 20px;
    }
    h2 {
      margin-bottom: 10px;
    }
    ul {
      list-style: none;
      padding-left: 0;
    }
    li {
      background: #fff;
      margin-bottom: 6px;
      padding: 10px;
      border-radius: 6px;
      box-shadow: 0 1px 3px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Sales Summary by Category</h2>
  <ul id="salesList"></ul>

  <script>
    const detailedSales = [
      { category: 'Books', sales: 200 },
      { category: 'Electronics', sales: 400 },
      { category: 'Books', sales: 300 },
      { category: 'Clothing', sales: 700 },
      { category: 'Electronics', sales: 800 }
    ];

    const grouped = d3.group(detailedSales, d => d.category);
    const salesList = document.getElementById('salesList');

    for (const [category, records] of grouped) {
```

```
      const totalSales = d3.sum(records, d => d.sales);
      console.log(`${category}: ${totalSales}`);

      const li = document.createElement('li');
      li.textContent = `${category}: $${totalSales.toLocaleString()}`;
      salesList.appendChild(li);
    }
  </script>
</body>
</html>
```

2.2.7 Calculating Daily Averages from Time Series Data

Time series data often requires aggregation by date.

Example dataset:

```
const timeSeries = [
  { date: new Date('2023-06-01'), value: 10 },
  { date: new Date('2023-06-01'), value: 20 },
  { date: new Date('2023-06-02'), value: 15 },
  { date: new Date('2023-06-02'), value: 25 }
];

//Calculate average value per day:

const groupedByDate = d3.group(timeSeries, d => d.date.toISOString().split('T')[0]);

const dailyAverages = Array.from(groupedByDate, ([date, records]) => {
  const avg = d3.mean(records, d => d.value);
  return { date, average: avg };
});

console.log(JSON.stringify(dailyAverages,null,2));
/* Output:
[
  { date: '2023-06-01', average: 15 },
  { date: '2023-06-02', average: 20 }
]
*/
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Time Series Averages</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f9f9f9;
```

```
      padding: 20px;
    }
    h2 {
      margin-bottom: 12px;
    }
    ul {
      list-style: none;
      padding-left: 0;
    }
    li {
      background: #fff;
      margin-bottom: 8px;
      padding: 10px 16px;
      border-radius: 6px;
      box-shadow: 0 1px 3px rgba(0,0,0,0.08);
    }
  </style>
</head>
<body>
  <h2>Daily Averages from Time Series</h2>
  <ul id="averageList"></ul>

  <script>
    const timeSeries = [
      { date: new Date('2023-06-01'), value: 10 },
      { date: new Date('2023-06-01'), value: 20 },
      { date: new Date('2023-06-02'), value: 15 },
      { date: new Date('2023-06-02'), value: 25 }
    ];

    const groupedByDate = d3.group(timeSeries, d => d.date.toISOString().split('T')[0]);

    const dailyAverages = Array.from(groupedByDate, ([date, records]) => {
      const avg = d3.mean(records, d => d.value);
      return { date, average: avg };
    });

    console.log("Daily Averages:", JSON.stringify(dailyAverages, null, 2));

    const list = document.getElementById('averageList');
    dailyAverages.forEach(({ date, average }) => {
      const item = document.createElement('li');
      item.textContent = `${date}: Average = ${average.toFixed(2)}`;
      list.appendChild(item);
    });
  </script>
</body>
</html>
```

2.2.8 Why Clean and Prepared Data Matters

Clean, well-structured data:

- Makes your visualizations easier to create and maintain,

- Ensures accurate and meaningful insights,
- Reduces errors and unexpected bugs,
- Improves chart readability and interpretation.

Poorly prepared data can lead to confusing, misleading, or cluttered visuals that hinder understanding.

2.2.9 Summary

- Use **sorting** to order data meaningfully.
- Apply **filtering** to focus on relevant subsets.
- Use **mapping** to transform data items.
- Group and aggregate data to summarize complex datasets, leveraging JavaScript's `reduce` or D3's `group`, `sum`, and `mean`.
- Prepare time series data by grouping by dates and calculating aggregates like daily averages.
- Clean, well-prepared data is fundamental for effective and trustworthy visualizations.

Chapter 3.

Building Visuals with D3.js

1. What is D3 and Why Use It?
2. Selections, Data Joins, and Enter/Update/Exit

3 Building Visuals with D3.js

3.1 What is D3 and Why Use It?

When it comes to creating web-based data visualizations, **D3.js** stands out as one of the most powerful and flexible tools available. But what exactly is D3, and why might you choose it over other charting libraries?

3.1.1 D3.js: Data-Driven Documents

D3 stands for **Data-Driven Documents**. At its core, D3 is a JavaScript library that **binds data directly to elements in the Document Object Model (DOM)**, allowing you to control every aspect of your visualization by manipulating HTML, SVG, or Canvas elements based on data.

Unlike many charting libraries that offer pre-built chart types (like bar charts, pie charts, or line charts) as ready-to-use components, D3 gives you **fine-grained control** over how data is translated into visual elements. This means you can create virtually any type of visualization, from simple charts to highly customized and interactive graphics.

3.1.2 How D3 Differs from Other Charting Libraries

Pre-Built Charts vs. Custom Control

Libraries like **Chart.js** or **Google Charts** provide a set of standard chart types that are easy to implement with minimal code. They are excellent when you want quick, common charts without worrying about low-level details.

However, these libraries can be limited when you need:

- Custom layouts or shapes,
- Unique interactions or animations,
- Complex data transformations,
- Integration with other web content.

D3, by contrast, does not come with built-in charts. Instead, it provides a rich set of **tools for data binding, element selection, and attribute manipulation**. This lets you build exactly what you want — no more, no less.

The Power and Flexibility of D3

- **Data binding:** Connect data arrays to DOM elements, so changes in data automatically reflect in the visuals.

- **Selections:** Select and manipulate specific elements on the page, allowing targeted updates.
- **Enter/Update/Exit pattern:** Manage elements as data changes — create new elements for new data, update existing ones, and remove obsolete elements.
- **Scales and layouts:** Built-in functions help map data values to visual properties like position, color, and size.
- **Transitions:** Smoothly animate changes in your visualizations.
- **Extensibility:** You can mix D3 with other JavaScript frameworks or libraries to create complex interactive experiences.

A Simple Hello World Example with D3

Let's create a basic example that appends a red circle to the webpage using D3:

```
  <svg width="200" height="200"></svg>

  <script>
    // Select the SVG container
    const svg = d3.select('svg');

    // Append a circle element bound to no data (simple example)
    svg.append('circle')
      .attr('cx', 100)      // x-coordinate of center
      .attr('cy', 100)      // y-coordinate of center
      .attr('r', 50)        // radius
      .attr('fill', 'red'); // fill color
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Hello World</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
  <svg width="200" height="200"></svg>

  <script>
    // Select the SVG container
    const svg = d3.select('svg');

    // Append a circle element bound to no data (simple example)
    svg.append('circle')
      .attr('cx', 100)      // x-coordinate of center
      .attr('cy', 100)      // y-coordinate of center
      .attr('r', 50)        // radius
      .attr('fill', 'red'); // fill color
  </script>
</body>
</html>
```

Open this page in a browser, and you will see a bright red circle centered inside the SVG.

Though this is a simple static example, D3's true strength emerges when you bind data arrays to elements and manipulate them dynamically — which we will explore next.

3.1.3 Summary

D3.js is not just another charting library — it is a powerful **data-driven approach** to building web visuals. Its fine control over DOM elements, combined with rich data binding and manipulation tools, makes it ideal for creating custom, interactive, and dynamic visualizations.

In the upcoming sections, you will learn how D3 manages data and elements together through selections, data joins, and the enter/update/exit pattern — the fundamental concepts for mastering D3.

3.2 Selections, Data Joins, and Enter/Update/Exit

At the heart of D3.js is a powerful pattern that manages how data and DOM elements work together. This core concept involves **selecting elements**, **binding data** to those elements, and then **updating the DOM** to reflect changes in data. Understanding this process — especially the **enter-update-exit** cycle — is key to mastering D3.

3.2.1 The Core Pattern: Select, Bind, Update

1. **Selections**: Use D3 to select existing DOM elements or create new ones.
2. **Data Binding (Data Join)**: Bind an array of data to the selection, linking each datum to a DOM element.
3. **Enter, Update, Exit**: Handle three cases:
    - **Enter**: Create new elements for new data points.
    - **Update**: Modify existing elements for updated data.
    - **Exit**: Remove elements when data is no longer present.

3.2.2 Step-by-Step Example: Rendering Rectangles from an Array

Imagine you want to display a series of rectangles representing some numeric values. You start with an initial dataset, and later update the data — adding, changing, or removing values.

3.2.3 Initial Data and SVG Setup

```
<svg width="300" height="150"></svg>
```

Javascript:

```
  const svg = d3.select('svg');
  const width = +svg.attr('width');

  // Initial dataset
  let data = [30, 80, 45, 60];

  // Function to render rectangles based on data
  function render(data) {
    // Select all rectangles and bind data
    const rects = svg.selectAll('rect')
      .data(data, (d, i) => i); // Key function: index as id

    // EXIT: Remove rectangles with no matching data
    rects.exit()
      .transition()
      .duration(500)
      .attr('height', 0)
      .remove();

    // UPDATE: Update existing rectangles
    rects
      .transition()
      .duration(500)
      .attr('y', d => 150 - d)
      .attr('height', d => d);

    // ENTER: Create new rectangles for new data
    rects.enter()
      .append('rect')
      .attr('x', (d, i) => i * 40)
      .attr('width', 30)
      .attr('y', 150)       // Start at bottom for animation
      .attr('height', 0)    // Start with zero height
      .attr('fill', 'steelblue')
      .transition()
      .duration(500)
      .attr('y', d => 150 - d)
      .attr('height', d => d);
  }

  // Initial render
  render(data);

  // Update data after 2 seconds
  setTimeout(() => {
    data = [50, 40, 70]; // Data changed: one removed, two updated
    render(data);
  }, 2000);

  // Another update after 4 seconds
  setTimeout(() => {
    data = [90, 20, 40, 60, 80]; // Data changed: new added
```

```
    render(data);
  }, 4000);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<svg width="300" height="150"></svg>
<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
  const svg = d3.select('svg');
  const width = +svg.attr('width');

  // Initial dataset
  let data = [30, 80, 45, 60];

  // Function to render rectangles based on data
  function render(data) {
    // Select all rectangles and bind data
    const rects = svg.selectAll('rect')
      .data(data, (d, i) => i); // Key function: index as id

    // EXIT: Remove rectangles with no matching data
    rects.exit()
      .transition()
      .duration(500)
      .attr('height', 0)
      .remove();

    // UPDATE: Update existing rectangles
    rects
      .transition()
      .duration(500)
      .attr('y', d => 150 - d)
      .attr('height', d => d);

    // ENTER: Create new rectangles for new data
    rects.enter()
      .append('rect')
      .attr('x', (d, i) => i * 40)
      .attr('width', 30)
      .attr('y', 150)       // Start at bottom for animation
      .attr('height', 0)    // Start with zero height
      .attr('fill', 'steelblue')
      .transition()
      .duration(500)
      .attr('y', d => 150 - d)
      .attr('height', d => d);
  }

  // Initial render
  render(data);
```

```
  // Update data after 2 seconds
  setTimeout(() => {
    data = [50, 40, 70]; // Data changed: one removed, two updated
    render(data);
  }, 2000);

  // Another update after 4 seconds
  setTimeout(() => {
    data = [90, 20, 40, 60, 80]; // Data changed: new added
    render(data);
  }, 4000);
</script>
</body>
</html>
```

3.2.4 What Happens in the Code?

- **Selection:** `svg.selectAll('rect').data(data, (d, i) => i)` selects all existing `<rect>` elements and binds the new data array. The second argument is a *key function* identifying data points by index, helping D3 match data with elements.
- **Exit:** `.exit()` selects rectangles with no corresponding data and removes them with a shrinking animation.
- **Update:** The existing rectangles matched to new data update their attributes like height and position.
- **Enter:** For any new data points without existing elements, `.enter()` creates new rectangles, starting from height 0, then animating to their correct size and position.

3.2.5 Visualizing the Enter-Update-Exit Cycle

Phase	What It Means	Visual Metaphor
Enter	New data → Create new elements	New building blocks being added
Update	Existing data → Modify elements	Renovating or repainting buildings
Exit	Removed data → Remove elements	Demolishing obsolete buildings

Each cycle ensures the DOM exactly reflects the current state of the data — no more, no less.

3.2.6 Why is This Important?

This pattern lets your visualization:

- **Dynamically respond** to changing data,
- **Efficiently update** only what needs to change,
- **Avoid unnecessary DOM manipulation**, improving performance,
- **Maintain smooth animations** and user interactions.

3.2.7 Summary

Selections, data joins, and the enter-update-exit pattern form the backbone of D3's approach to dynamic visualizations. By binding data to DOM elements and managing changes carefully, you create visuals that update seamlessly as data evolves.

In the next sections, we'll build on this foundation to explore scales, axes, and more complex interactions.

Chapter 4.

D3.js Drawing with SVG

1. Creating Shapes and Paths
2. Scaling and Positioning Elements
3. Axes and Labels

4 D3.js Drawing with SVG

4.1 Creating Shapes and Paths

One of the foundational skills in D3.js is creating visual elements using **SVG (Scalable Vector Graphics)**. SVG provides a set of basic shapes—such as rectangles, circles, lines, and complex paths—that you can manipulate with D3 to build rich data visualizations.

In this section, you'll learn how to create these shapes and specify their attributes to represent data visually. We will also walk through an example that draws multiple shapes based on dataset values.

4.1.1 Drawing Basic SVG Shapes with D3

D3 makes it easy to create and modify SVG shapes by appending elements and setting their attributes dynamically.

4.1.2 Rectangles (`rect`)

Rectangles are useful for bar charts and backgrounds.

Key attributes:

- `x`, `y`: coordinates of the top-left corner,
- `width`, `height`: size of the rectangle,
- `fill`: fill color.

```
svg.append('rect')
  .attr('x', 10)
  .attr('y', 20)
  .attr('width', 100)
  .attr('height', 50)
  .attr('fill', 'steelblue');
```

4.1.3 Circles (`circle`)

Circles are often used for scatterplots or to highlight points.

Key attributes:

- `cx`, `cy`: center coordinates,
- `r`: radius,

- `fill`: fill color.

```
svg.append('circle')
  .attr('cx', 150)
  .attr('cy', 45)
  .attr('r', 25)
  .attr('fill', 'orange');
```

4.1.4 Lines (`line`)

Lines can show relationships, connections, or axes.

Key attributes:

- `x1`, `y1`: starting point,
- `x2`, `y2`: ending point,
- `stroke`: line color,
- `stroke-width`: thickness.

```
svg.append('line')
  .attr('x1', 10)
  .attr('y1', 100)
  .attr('x2', 200)
  .attr('y2', 100)
  .attr('stroke', 'black')
  .attr('stroke-width', 2);
```

4.1.5 Paths (`path`)

Paths define complex shapes or curves with the `d` attribute, which contains a series of commands.

Example: Draw a simple triangle using `path`:

```
svg.append('path')
  .attr('d', 'M 50 150 L 150 150 L 100 50 Z')  // Move, line, line, close path
  .attr('fill', 'green');
```

4.1.6 Example: Drawing Multiple Shapes Based on Data

Let's create a small visualization that draws rectangles and circles from an array of data objects:

```
<svg width="300" height="200"></svg>
<script src="https://d3js.org/d3.v7.min.js"></script>
```

Javascript:

```
  const svg = d3.select('svg');

  // Dataset with values controlling rectangle height and circle radius
  const data = [
    { x: 20, rectHeight: 40, circleRadius: 10 },
    { x: 80, rectHeight: 70, circleRadius: 15 },
    { x: 140, rectHeight: 50, circleRadius: 12 },
    { x: 200, rectHeight: 90, circleRadius: 20 }
  ];

  // Draw rectangles
  svg.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr('x', d => d.x)
    .attr('y', d => 150 - d.rectHeight)  // Position from bottom
    .attr('width', 30)
    .attr('height', d => d.rectHeight)
    .attr('fill', 'steelblue');

  // Draw circles above rectangles
  svg.selectAll('circle')
    .data(data)
    .enter()
    .append('circle')
    .attr('cx', d => d.x + 15)            // Center horizontally on rectangle
    .attr('cy', d => 150 - d.rectHeight - d.circleRadius - 5) // Above rect
    .attr('r', d => d.circleRadius)
    .attr('fill', 'orange');
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<svg width="300" height="200"></svg>
<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
  const svg = d3.select('svg');

  // Dataset with values controlling rectangle height and circle radius
  const data = [
    { x: 20, rectHeight: 40, circleRadius: 10 },
    { x: 80, rectHeight: 70, circleRadius: 15 },
    { x: 140, rectHeight: 50, circleRadius: 12 },
    { x: 200, rectHeight: 90, circleRadius: 20 }
  ];

  // Draw rectangles
  svg.selectAll('rect')
    .data(data)
```

```
    .enter()
    .append('rect')
    .attr('x', d => d.x)
    .attr('y', d => 150 - d.rectHeight)  // Position from bottom
    .attr('width', 30)
    .attr('height', d => d.rectHeight)
    .attr('fill', 'steelblue');

  // Draw circles above rectangles
  svg.selectAll('circle')
    .data(data)
    .enter()
    .append('circle')
    .attr('cx', d => d.x + 15)            // Center horizontally on rectangle
    .attr('cy', d => 150 - d.rectHeight - d.circleRadius - 5) // Above rect
    .attr('r', d => d.circleRadius)
    .attr('fill', 'orange');
</script>
</body>
</html>
```

4.1.7 What This Does:

- Each data point creates a blue rectangle whose height corresponds to `rectHeight`.
- Above each rectangle, an orange circle is drawn with radius `circleRadius`.
- The horizontal position (`x`) spaces out the shapes evenly.

4.1.8 Summary

- Use SVG elements like `<rect>`, `<circle>`, `<line>`, and `<path>` to draw basic shapes.
- Attributes such as `x`, `y`, `cx`, `cy`, `r`, `width`, and `height` control position and size.
- Paths allow for more complex shapes with commands in the `d` attribute.
- D3's data binding allows you to generate multiple shapes dynamically from datasets.

Mastering SVG shapes is the first step toward creating rich, data-driven graphics, and D3's seamless integration with SVG empowers you to bring your data to life visually.

4.2 Scaling and Positioning Elements

When creating data visualizations, raw data values often need to be translated into pixel positions on the screen. For example, a sales number of 1000 might correspond to a bar height of 150 pixels, or a date might map to a position along a horizontal timeline.

D3's scale functions provide a powerful and flexible way to map your data values—be they numbers, categories, or dates—to visual ranges like coordinates, lengths, or colors.

In this section, we will explore how to use some of the most common D3 scales: `scaleLinear`, `scaleBand`, and `scaleTime`. We'll also discuss how margins affect your drawing space to create clean, well-structured charts.

4.2.1 Why Use Scales?

Imagine you want to draw a bar chart for sales values ranging from 0 to 1000 on a canvas that is only 300 pixels high. You cannot just use raw data values as pixel heights because 1000 pixels would not fit!

Scales let you define:

- **Input domain**: The range of your data values (e.g., 0 to 1000).
- **Output range**: The range of pixel values on the screen (e.g., 0 to 300 pixels).

Then, when you pass a data value to the scale, it returns the corresponding pixel value automatically.

4.2.2 Common D3 Scale Functions

`scaleLinear`

Maps continuous numeric input to continuous numeric output — ideal for quantitative data such as sales, temperature, or scores.

Example: Vertical scale for bar heights

```
const height = 300;

const yScale = d3.scaleLinear()
  .domain([0, 1000])          // data values from 0 to 1000
  .range([height, 0]);        // pixel range: bottom (300) to top (0)

// yScale(0) returns 300 (bottom)
// yScale(1000) returns 0 (top)
```

Note the range is reversed because SVG's y=0 is at the top, so higher data values map to smaller y-coordinates (higher up on the screen).

`scaleBand`

Maps discrete categories to distinct bands along an axis, useful for categorical data like product names or months.

- Divides the output range into evenly spaced bands.
- Supports padding between bands.

Example: Horizontal scale for bar positions

```
const width = 400;
const categories = ['Books', 'Electronics', 'Clothing'];

const xScale = d3.scaleBand()
  .domain(categories)         // categories
  .range([0, width])          // pixel range along x-axis
  .padding(0.1);              // space between bands

console.log(xScale('Books'));        // position for "Books" category
console.log(xScale.bandwidth());     // width of each band
```

scaleTime

Maps date/time values to numeric output for timelines or time series.

```
const width = 500;
const parseDate = d3.timeParse('%Y-%m-%d');

const xScale = d3.scaleTime()
  .domain([parseDate('2023-01-01'), parseDate('2023-12-31')])
  .range([0, width]);
```

4.2.3 Example: Horizontal Bar Chart Layout with Margins

Margins create padding between the edges of the SVG and your chart content, allowing space for axes, labels, and preventing clipping.

```
const svgWidth = 500, svgHeight = 300;
const margin = { top: 20, right: 20, bottom: 40, left: 60 };

// Inner drawing area
const width = svgWidth - margin.left - margin.right;
const height = svgHeight - margin.top - margin.bottom;

const svg = d3.select('svg')
  .attr('width', svgWidth)
  .attr('height', svgHeight);

const data = [
  { category: 'Books', value: 400 },
  { category: 'Electronics', value: 900 },
  { category: 'Clothing', value: 600 }
];

// Scales
const xScale = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, width])
  .padding(0.2);
```

```
const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([height, 0]);

// Group for chart elements with margin applied
const chartGroup = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);

// Draw bars
chartGroup.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', d => xScale(d.category))
  .attr('y', d => yScale(d.value))
  .attr('width', xScale.bandwidth())
  .attr('height', d => height - yScale(d.value))
  .attr('fill', 'steelblue');
```

Here:

- We reserve margins around the chart area.
- The inner group (`chartGroup`) is translated by margins.
- `xScale` positions bars horizontally by category.
- `yScale` determines the vertical height of each bar.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 40px;
      text-align: center;
    }
    svg {
      border: 1px solid #ccc;
      margin-top: 20px;
    }
  </style>
</head>
<body>

  <h1>D3 Bar Chart Example</h1>
  <svg></svg>

  <script>
    const svgWidth = 500, svgHeight = 300;
    const margin = { top: 20, right: 20, bottom: 40, left: 60 };

    const width = svgWidth - margin.left - margin.right;
```

```
    const height = svgHeight - margin.top - margin.bottom;

    const svg = d3.select('svg')
      .attr('width', svgWidth)
      .attr('height', svgHeight);

    const data = [
      { category: 'Books', value: 400 },
      { category: 'Electronics', value: 900 },
      { category: 'Clothing', value: 600 }
    ];

    // Scales
    const xScale = d3.scaleBand()
      .domain(data.map(d => d.category))
      .range([0, width])
      .padding(0.2);

    const yScale = d3.scaleLinear()
      .domain([0, d3.max(data, d => d.value)])
      .range([height, 0]);

    const chartGroup = svg.append('g')
      .attr('transform', `translate(${margin.left},${margin.top})`);

    // Draw bars
    chartGroup.selectAll('rect')
      .data(data)
      .enter()
      .append('rect')
      .attr('x', d => xScale(d.category))
      .attr('y', d => yScale(d.value))
      .attr('width', xScale.bandwidth())
      .attr('height', d => height - yScale(d.value))
      .attr('fill', 'steelblue');

    // X Axis
    chartGroup.append('g')
      .attr('transform', `translate(0,${height})`)
      .call(d3.axisBottom(xScale));

    // Y Axis
    chartGroup.append('g')
      .call(d3.axisLeft(yScale));
  </script>
</body>
</html>
```

4.2.4 Summary

- **D3 scales** map your data domain to pixel ranges, enabling accurate, flexible positioning.
- `scaleLinear` is ideal for continuous numeric data.
- `scaleBand` works well for discrete categorical data with evenly spaced bands.

- `scaleTime` handles temporal data.
- Use margins to create padding and prevent overlap between chart elements and axes.
- Combining scales with margins helps build clear, well-structured visualizations.

4.3 Axes and Labels

Axes are fundamental to most data visualizations because they provide **context**—helping viewers understand the scale and meaning of your data. D3.js simplifies creating and customizing axes through **axis generators**, which automatically create ticks, lines, and labels based on your scales.

In this section, you will learn how to use D3's axis generators (`axisBottom`, `axisLeft`, etc.), add them to your SVG, and format tick marks and labels to make your charts more readable.

4.3.1 D3 Axis Generators

D3 provides four basic axis generator functions:

- `d3.axisBottom(scale)`: Creates an axis with ticks below the horizontal line (usually for x-axes).
- `d3.axisTop(scale)`: Creates an axis with ticks above the horizontal line.
- `d3.axisLeft(scale)`: Creates a vertical axis with ticks to the left (commonly for y-axes).
- `d3.axisRight(scale)`: Creates a vertical axis with ticks to the right.

Each axis generator takes a **scale** as input and generates the SVG elements required to render the axis.

4.3.2 Adding an Axis to an SVG

Here's a step-by-step example of adding labeled x and y axes to a basic bar chart.

```
const svgWidth = 500, svgHeight = 300;
const margin = { top: 20, right: 20, bottom: 50, left: 70 };

const width = svgWidth - margin.left - margin.right;
const height = svgHeight - margin.top - margin.bottom;

const svg = d3.select('svg')
  .attr('width', svgWidth)
  .attr('height', svgHeight);

const data = [
```

```
  { category: 'Books', value: 400 },
  { category: 'Electronics', value: 900 },
  { category: 'Clothing', value: 600 }
];

// Scales
const xScale = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, width])
  .padding(0.2);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .nice()
  .range([height, 0]);

// Chart group with margin transform
const chartGroup = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);

// Draw bars
chartGroup.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', d => xScale(d.category))
  .attr('y', d => yScale(d.value))
  .attr('width', xScale.bandwidth())
  .attr('height', d => height - yScale(d.value))
  .attr('fill', 'steelblue');

// Create and add x-axis
const xAxis = d3.axisBottom(xScale);
chartGroup.append('g')
  .attr('transform', `translate(0,${height})`) // Move to bottom of chart
  .call(xAxis)
  .selectAll('text')                           // Style x-axis labels
  .attr('transform', 'rotate(-40)')
  .style('text-anchor', 'end');

// Create and add y-axis
const yAxis = d3.axisLeft(yScale)
  .ticks(5)                                    // Number of ticks
  .tickFormat(d3.format('~s'));                // Format ticks (e.g., 1k for 1000)

chartGroup.append('g')
  .call(yAxis);

// Add axis labels
// X-axis label
chartGroup.append('text')
  .attr('x', width / 2)
  .attr('y', height + margin.bottom - 5)
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('Product Category');

// Y-axis label
```

```
chartGroup.append('text')
  .attr('transform', 'rotate(-90)')
  .attr('x', -height / 2)
  .attr('y', -margin.left + 20)
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('Sales (units)');
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 40px;
      text-align: center;
    }
    svg {
      border: 1px solid #ccc;
      margin-top: 20px;
    }
  </style>
</head>
<body>

  <h1>D3 Bar Chart Example</h1>
  <svg></svg>

  <script>
const svgWidth = 500, svgHeight = 300;
const margin = { top: 20, right: 20, bottom: 50, left: 70 };

const width = svgWidth - margin.left - margin.right;
const height = svgHeight - margin.top - margin.bottom;

const svg = d3.select('svg')
  .attr('width', svgWidth)
  .attr('height', svgHeight);

const data = [
  { category: 'Books', value: 400 },
  { category: 'Electronics', value: 900 },
  { category: 'Clothing', value: 600 }
];

// Scales
const xScale = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, width])
  .padding(0.2);

const yScale = d3.scaleLinear()
```

```
  .domain([0, d3.max(data, d => d.value)])
  .nice()
  .range([height, 0]);

// Chart group with margin transform
const chartGroup = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);

// Draw bars
chartGroup.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', d => xScale(d.category))
  .attr('y', d => yScale(d.value))
  .attr('width', xScale.bandwidth())
  .attr('height', d => height - yScale(d.value))
  .attr('fill', 'steelblue');

// Create and add x-axis
const xAxis = d3.axisBottom(xScale);
chartGroup.append('g')
  .attr('transform', `translate(0,${height})`) // Move to bottom of chart
  .call(xAxis)
  .selectAll('text')                           // Style x-axis labels
  .attr('transform', 'rotate(-40)')
  .style('text-anchor', 'end');

// Create and add y-axis
const yAxis = d3.axisLeft(yScale)
  .ticks(5)                                    // Number of ticks
  .tickFormat(d3.format('~s'));                // Format ticks (e.g., 1k for 1000)

chartGroup.append('g')
  .call(yAxis);

// Add axis labels
// X-axis label
chartGroup.append('text')
  .attr('x', width / 2)
  .attr('y', height + margin.bottom - 5)
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('Product Category');

// Y-axis label
chartGroup.append('text')
  .attr('transform', 'rotate(-90)')
  .attr('x', -height / 2)
  .attr('y', -margin.left + 20)
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('Sales (units)');
  </script>

</body>
</html>
```

4.3.3 Formatting Ticks and Labels

- **Ticks** are the small marks on an axis that indicate data values.
- Use `.ticks(count)` on the axis generator to control the approximate number of ticks.
- `.tickFormat(formatFunction)` formats tick labels. For example:
  - `d3.format('~s')` formats large numbers with SI prefixes (`k`, `M`).
  - You can pass your own function for custom formatting.

4.3.4 Rotating Axis Labels

When category names or labels are long, rotating them improves readability. Use the following on the selected text elements:

```
.selectAll('text')
  .attr('transform', 'rotate(-40)')
  .style('text-anchor', 'end');
```

This rotates the labels 40 degrees counter-clockwise and aligns the text appropriately.

4.3.5 Summary

- D3's axis generators (`axisBottom`, `axisLeft`, etc.) create complete axes from scales.
- Append axes to your SVG and position them using `transform` (e.g., translate).
- Customize ticks with `.ticks()` and `.tickFormat()` for better clarity.
- Add axis labels using `<text>` elements for context.
- Rotate tick labels when needed to avoid overlap.

With well-designed axes and labels, your visualizations become easier to interpret and more professional looking.

Chapter 5.

D3.js Core Chart Types

1. Bar Charts
2. Line Charts
3. Scatter Plots
4. Pie/Donut Charts

5 D3.js Core Chart Types

5.1 Bar Charts

Bar charts are among the most common and effective ways to represent categorical data visually. They use rectangular bars whose lengths correspond to data values, making it easy to compare quantities across categories.

In this section, we will build a simple bar chart from an array of values using D3.js. You'll learn how to:

- Bind data to rectangles,
- Set up axes,
- Scale bar heights with `scaleLinear`,
- Use `scaleBand` for spacing bars,
- Add interactive tooltips for better user experience.

5.1.1 Step 1: Prepare the Data and SVG Container

Let's start with a simple array of numeric values:

```
const data = [25, 40, 15, 60, 20];
```

And create an SVG container for the chart:

```
<svg width="500" height="300"></svg>
```

5.1.2 Step 2: Define Margins and Dimensions

Margins provide padding around the chart for axes and labels:

```
const svgWidth = 500;
const svgHeight = 300;
const margin = { top: 20, right: 20, bottom: 40, left: 40 };

const width = svgWidth - margin.left - margin.right;
const height = svgHeight - margin.top - margin.bottom;
```

5.1.3 Step 3: Set Up Scales

We use:

- `scaleBand` to map each bar to a horizontal position and control its width and spacing,

- `scaleLinear` to scale the data values to bar heights.

```
const xScale = d3.scaleBand()
  .domain(data.map((d, i) => i))   // Use indices as categories
  .range([0, width])
  .padding(0.2);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data)])
  .range([height, 0]);             // Inverted range for SVG coordinates
```

5.1.4 Step 4: Append a Group for Chart Content

Translate the group to accommodate margins:

```
const svg = d3.select('svg')
  .attr('width', svgWidth)
  .attr('height', svgHeight);

const chartGroup = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);
```

5.1.5 Step 5: Draw Bars Using Data Join

Bind data to rectangles, set their positions and sizes according to scales:

```
chartGroup.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', (d, i) => xScale(i))
  .attr('y', d => yScale(d))
  .attr('width', xScale.bandwidth())
  .attr('height', d => height - yScale(d))
  .attr('fill', 'steelblue');
```

5.1.6 Step 6: Add Axes

Create and append x and y axes for better readability:

```
const xAxis = d3.axisBottom(xScale)
  .tickFormat(i => `Item ${i + 1}`);

const yAxis = d3.axisLeft(yScale)
  .ticks(5);

chartGroup.append('g')
```

```
  .attr('transform', `translate(0,${height})`)
  .call(xAxis);

chartGroup.append('g')
  .call(yAxis);
```

5.1.7 Bonus: Add Tooltips for Interactivity

Tooltips enhance user experience by showing exact data values on hover.

```
// Create a tooltip div (add to HTML body)
const tooltip = d3.select('body').append('div')
  .style('position', 'absolute')
  .style('background', '#f9f9f9')
  .style('padding', '5px 10px')
  .style('border', '1px solid #ccc')
  .style('border-radius', '4px')
  .style('pointer-events', 'none')
  .style('opacity', 0);

chartGroup.selectAll('rect')
  .on('mouseover', function(event, d) {
    tooltip.transition().duration(200).style('opacity', 0.9);
    tooltip.html(`Value: ${d}`)
      .style('left', (event.pageX + 10) + 'px')
      .style('top', (event.pageY - 28) + 'px');
    d3.select(this).attr('fill', 'orange');
  })
  .on('mouseout', function() {
    tooltip.transition().duration(500).style('opacity', 0);
    d3.select(this).attr('fill', 'steelblue');
  });
```

5.1.8 Full Working Example

Here's the complete code you can paste into an HTML file:

```
    const data = [25, 40, 15, 60, 20];

    const svgWidth = 500;
    const svgHeight = 300;
    const margin = { top: 20, right: 20, bottom: 40, left: 40 };

    const width = svgWidth - margin.left - margin.right;
    const height = svgHeight - margin.top - margin.bottom;

    const svg = d3.select('svg')
      .attr('width', svgWidth)
      .attr('height', svgHeight);
```

```
  const xScale = d3.scaleBand()
    .domain(data.map((d, i) => i))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data)])
    .range([height, 0]);

  const chartGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  // Draw bars
  chartGroup.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr('x', (d, i) => xScale(i))
    .attr('y', d => yScale(d))
    .attr('width', xScale.bandwidth())
    .attr('height', d => height - yScale(d))
    .attr('fill', 'steelblue');

  // Axes
  const xAxis = d3.axisBottom(xScale)
    .tickFormat(i => `Item ${i + 1}`);

  const yAxis = d3.axisLeft(yScale)
    .ticks(5);

  chartGroup.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(xAxis);

  chartGroup.append('g')
    .call(yAxis);

  // Tooltip
  const tooltip = d3.select('body').append('div')
    .style('position', 'absolute')
    .style('background', '#f9f9f9')
    .style('padding', '5px 10px')
    .style('border', '1px solid #ccc')
    .style('border-radius', '4px')
    .style('pointer-events', 'none')
    .style('opacity', 0);

  chartGroup.selectAll('rect')
    .on('mouseover', function(event, d) {
      tooltip.transition().duration(200).style('opacity', 0.9);
      tooltip.html(`Value: ${d}`)
        .style('left', (event.pageX + 10) + 'px')
        .style('top', (event.pageY - 28) + 'px');
      d3.select(this).attr('fill', 'orange');
    })
    .on('mouseout', function() {
      tooltip.transition().duration(500).style('opacity', 0);
      d3.select(this).attr('fill', 'steelblue');
```

```
    });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body { font-family: sans-serif; }
  </style>
</head>
<body>
  <svg width="500" height="300"></svg>

  <script>
    const data = [25, 40, 15, 60, 20];

    const svgWidth = 500;
    const svgHeight = 300;
    const margin = { top: 20, right: 20, bottom: 40, left: 40 };

    const width = svgWidth - margin.left - margin.right;
    const height = svgHeight - margin.top - margin.bottom;

    const svg = d3.select('svg')
      .attr('width', svgWidth)
      .attr('height', svgHeight);

    const xScale = d3.scaleBand()
      .domain(data.map((d, i) => i))
      .range([0, width])
      .padding(0.2);

    const yScale = d3.scaleLinear()
      .domain([0, d3.max(data)])
      .range([height, 0]);

    const chartGroup = svg.append('g')
      .attr('transform', `translate(${margin.left},${margin.top})`);

    // Draw bars
    chartGroup.selectAll('rect')
      .data(data)
      .enter()
      .append('rect')
      .attr('x', (d, i) => xScale(i))
      .attr('y', d => yScale(d))
      .attr('width', xScale.bandwidth())
      .attr('height', d => height - yScale(d))
      .attr('fill', 'steelblue');

    // Axes
    const xAxis = d3.axisBottom(xScale)
      .tickFormat(i => `Item ${i + 1}`);
```

```
    const yAxis = d3.axisLeft(yScale)
      .ticks(5);

    chartGroup.append('g')
      .attr('transform', `translate(0,${height})`)
      .call(xAxis);

    chartGroup.append('g')
      .call(yAxis);

    // Tooltip
    const tooltip = d3.select('body').append('div')
      .style('position', 'absolute')
      .style('background', '#f9f9f9')
      .style('padding', '5px 10px')
      .style('border', '1px solid #ccc')
      .style('border-radius', '4px')
      .style('pointer-events', 'none')
      .style('opacity', 0);

    chartGroup.selectAll('rect')
      .on('mouseover', function(event, d) {
        tooltip.transition().duration(200).style('opacity', 0.9);
        tooltip.html(`Value: ${d}`)
          .style('left', (event.pageX + 10) + 'px')
          .style('top', (event.pageY - 28) + 'px');
        d3.select(this).attr('fill', 'orange');
      })
      .on('mouseout', function() {
        tooltip.transition().duration(500).style('opacity', 0);
        d3.select(this).attr('fill', 'steelblue');
      });
  </script>
</body>
</html>
```

5.1.9 Summary

- Bar charts visualize data with rectangular bars sized proportionally to values.
- Use `scaleBand` for horizontal positioning and spacing of bars.
- Use `scaleLinear` to scale bar heights.
- Bind data to `<rect>` elements with D3's data join.
- Add axes for context.
- Enhance charts with interactive tooltips for better user experience.

Next, you can explore line charts or add animations to your bars for richer interaction.

5.2 Line Charts

Line charts are ideal for visualizing trends over continuous data, such as time series. They connect data points with lines, making it easy to see patterns, changes, and comparisons.

In this section, we'll learn how to create line charts with D3 by:

- Parsing date-based data,
- Using `d3.scaleTime()` for the x-axis,
- Creating lines with `d3.line()`,
- Smoothing curves,
- Plotting multiple data series on the same chart.

5.2.1 Parsing Dates and Setting Up Scales

Time data is typically stored as strings and needs to be parsed into JavaScript Date objects for D3 to work with them effectively.

Use `d3.timeParse` to convert date strings to Date objects:

```
const parseDate = d3.timeParse('%Y-%m-%d');
```

Then, create a time scale (`scaleTime`) for the x-axis and a linear scale for the y-axis:

```
const xScale = d3.scaleTime()
  .domain(d3.extent(data, d => d.date))
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([height, 0]);
```

5.2.2 Using `d3.line()` to Create Paths

`d3.line()` generates the SVG path data string from your dataset.

You define:

- How to extract x and y values from data points,
- Whether to smooth the line using curve functions like `d3.curveMonotoneX`.

```
const line = d3.line()
  .x(d => xScale(d.date))
  .y(d => yScale(d.value))
  .curve(d3.curveMonotoneX); // Smooth curve
```

5.2.3 Example: Multi-Series Smoothed Line Chart

Here is a full example with two series plotted on the same chart.

```
const svg = d3.select('svg');
const margin = { top: 30, right: 100, bottom: 50, left: 60 };
const width = +svg.attr('width') - margin.left - margin.right;
const height = +svg.attr('height') - margin.top - margin.bottom;

const parseDate = d3.timeParse('%Y-%m-%d');

// Sample multi-series data
const data = [
  {
    name: 'Product A',
    values: [
      { date: '2023-01-01', value: 30 },
      { date: '2023-02-01', value: 50 },
      { date: '2023-03-01', value: 45 },
      { date: '2023-04-01', value: 60 }
    ]
  },
  {
    name: 'Product B',
    values: [
      { date: '2023-01-01', value: 20 },
      { date: '2023-02-01', value: 35 },
      { date: '2023-03-01', value: 55 },
      { date: '2023-04-01', value: 70 }
    ]
  }
];

// Parse date strings
data.forEach(series => {
  series.values.forEach(d => {
    d.date = parseDate(d.date);
  });
});

// Flatten all data points to get combined domains
const allValues = data.flatMap(series => series.values);

const xScale = d3.scaleTime()
  .domain(d3.extent(allValues, d => d.date))
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(allValues, d => d.value)])
  .nice()
  .range([height, 0]);

const color = d3.scaleOrdinal(d3.schemeCategory10)
  .domain(data.map(d => d.name));

const line = d3.line()
  .x(d => xScale(d.date))
  .y(d => yScale(d.value))
```

```
    .curve(d3.curveMonotoneX);

  const chartGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  // Draw axes
  chartGroup.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(d3.axisBottom(xScale).ticks(5).tickFormat(d3.timeFormat('%b %Y')));

  chartGroup.append('g')
    .call(d3.axisLeft(yScale).ticks(6));

  // Draw lines for each series
  chartGroup.selectAll('.line')
    .data(data)
    .enter()
    .append('path')
    .attr('class', 'line')
    .attr('d', d => line(d.values))
    .attr('fill', 'none')
    .attr('stroke', d => color(d.name))
    .attr('stroke-width', 3);

  // Add legend
  const legend = svg.append('g')
    .attr('transform', `translate(${width + margin.left + 20},${margin.top})`);

  legend.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr('x', 0)
    .attr('y', (d, i) => i * 25)
    .attr('width', 18)
    .attr('height', 18)
    .attr('fill', d => color(d.name));

  legend.selectAll('text')
    .data(data)
    .enter()
    .append('text')
    .attr('x', 24)
    .attr('y', (d, i) => i * 25 + 14)
    .text(d => d.name)
    .attr('font-size', '14px');
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<svg width="600" height="400"></svg>
```

```
<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
  const svg = d3.select('svg');
  const margin = { top: 30, right: 100, bottom: 50, left: 60 };
  const width = +svg.attr('width') - margin.left - margin.right;
  const height = +svg.attr('height') - margin.top - margin.bottom;

  const parseDate = d3.timeParse('%Y-%m-%d');

  // Sample multi-series data
  const data = [
    {
      name: 'Product A',
      values: [
        { date: '2023-01-01', value: 30 },
        { date: '2023-02-01', value: 50 },
        { date: '2023-03-01', value: 45 },
        { date: '2023-04-01', value: 60 }
      ]
    },
    {
      name: 'Product B',
      values: [
        { date: '2023-01-01', value: 20 },
        { date: '2023-02-01', value: 35 },
        { date: '2023-03-01', value: 55 },
        { date: '2023-04-01', value: 70 }
      ]
    }
  ];

  // Parse date strings
  data.forEach(series => {
    series.values.forEach(d => {
      d.date = parseDate(d.date);
    });
  });

  // Flatten all data points to get combined domains
  const allValues = data.flatMap(series => series.values);

  const xScale = d3.scaleTime()
    .domain(d3.extent(allValues, d => d.date))
    .range([0, width]);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(allValues, d => d.value)])
    .nice()
    .range([height, 0]);

  const color = d3.scaleOrdinal(d3.schemeCategory10)
    .domain(data.map(d => d.name));

  const line = d3.line()
    .x(d => xScale(d.date))
    .y(d => yScale(d.value))
    .curve(d3.curveMonotoneX);
```

```
  const chartGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  // Draw axes
  chartGroup.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(d3.axisBottom(xScale).ticks(5).tickFormat(d3.timeFormat('%b %Y')));

  chartGroup.append('g')
    .call(d3.axisLeft(yScale).ticks(6));

  // Draw lines for each series
  chartGroup.selectAll('.line')
    .data(data)
    .enter()
    .append('path')
    .attr('class', 'line')
    .attr('d', d => line(d.values))
    .attr('fill', 'none')
    .attr('stroke', d => color(d.name))
    .attr('stroke-width', 3);

  // Add legend
  const legend = svg.append('g')
    .attr('transform', `translate(${width + margin.left + 20},${margin.top})`);

  legend.selectAll('rect')
    .data(data)
    .enter()
    .append('rect')
    .attr('x', 0)
    .attr('y', (d, i) => i * 25)
    .attr('width', 18)
    .attr('height', 18)
    .attr('fill', d => color(d.name));

  legend.selectAll('text')
    .data(data)
    .enter()
    .append('text')
    .attr('x', 24)
    .attr('y', (d, i) => i * 25 + 14)
    .text(d => d.name)
    .attr('font-size', '14px');
</script>
</body>
</html>
```

5.2.4 Explanation

- **Parsing Dates**: We convert date strings to Date objects so the time scale works correctly.
- **Scales**: `scaleTime` maps dates to horizontal positions, `scaleLinear` maps values

vertically.

- **Line Generator**: `d3.line()` creates the path; `.curve(d3.curveMonotoneX)` smooths the line while preserving monotonicity.
- **Multiple Series**: We loop through each data series, drawing a separate path with a distinct color.
- **Legend**: Colored boxes and text label each series for clarity.

5.2.5 Summary

- Use `d3.scaleTime()` to handle dates on your x-axis.
- Convert date strings to Date objects with `d3.timeParse`.
- Generate line paths with `d3.line()`, specifying x and y accessors.
- Smooth lines with curve functions like `curveMonotoneX`.
- Plot multiple data series by binding each series to a separate `<path>`.
- Add axes and legends for context and clarity.

5.3 Scatter Plots

Scatter plots visualize the relationship between two numeric variables by plotting points on a Cartesian coordinate system. They are essential for identifying patterns, correlations, or clusters in data.

In this section, we'll create a scatter plot using D3.js by:

- Plotting data points as SVG `circle` elements,
- Using `scaleLinear` for both x and y axes,
- Adding labeled axes,
- Encoding a third variable as color,
- Adding interactive tooltips for detailed insights.

5.3.1 Step 1: Prepare the Data and SVG Container

Consider the following sample dataset where each point has `x`, `y`, and `category` values:

```
const data = [
  { x: 34, y: 78, category: 'A' },
  { x: 109, y: 280, category: 'B' },
  { x: 310, y: 120, category: 'A' },
  { x: 79, y: 411, category: 'C' },
  { x: 420, y: 220, category: 'B' },
  { x: 233, y: 145, category: 'C' },
```

```
];
```

And create an SVG element:

```
<svg width="600" height="400"></svg>
```

5.3.2 Step 2: Define Margins and Dimensions

```
const svgWidth = 600;
const svgHeight = 400;
const margin = { top: 20, right: 30, bottom: 50, left: 60 };

const width = svgWidth - margin.left - margin.right;
const height = svgHeight - margin.top - margin.bottom;
```

5.3.3 Step 3: Create Scales for x, y, and Color

We will create:

- `xScale` and `yScale` as linear scales mapping data to pixels,
- A color scale to encode categories.

```
const xScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.x)]).nice()
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.y)]).nice()
  .range([height, 0]);  // Inverted range for SVG coordinate system

const colorScale = d3.scaleOrdinal()
  .domain([...new Set(data.map(d => d.category))])
  .range(d3.schemeCategory10);
```

5.3.4 Step 4: Append Chart Group and Draw Axes

```
const svg = d3.select('svg')
  .attr('width', svgWidth)
  .attr('height', svgHeight);

const chartGroup = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);

const xAxis = d3.axisBottom(xScale);
const yAxis = d3.axisLeft(yScale);
```

```
chartGroup.append('g')
  .attr('transform', `translate(0,${height})`)
  .call(xAxis)
  .append('text')
  .attr('x', width / 2)
  .attr('y', 40)
  .attr('fill', 'black')
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('X Value');

chartGroup.append('g')
  .call(yAxis)
  .append('text')
  .attr('transform', 'rotate(-90)')
  .attr('x', -height / 2)
  .attr('y', -45)
  .attr('fill', 'black')
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .text('Y Value');
```

5.3.5 Step 5: Plot Points with Circles and Color Encoding

```
chartGroup.selectAll('circle')
  .data(data)
  .enter()
  .append('circle')
  .attr('cx', d => xScale(d.x))
  .attr('cy', d => yScale(d.y))
  .attr('r', 7)
  .attr('fill', d => colorScale(d.category))
  .attr('opacity', 0.8);
```

5.3.6 Step 6: Add Tooltips for Interactivity

Tooltips show details when hovering over points.

```
const tooltip = d3.select('body').append('div')
  .style('position', 'absolute')
  .style('background', '#fff')
  .style('padding', '6px 10px')
  .style('border', '1px solid #ccc')
  .style('border-radius', '4px')
  .style('pointer-events', 'none')
  .style('opacity', 0);

chartGroup.selectAll('circle')
  .on('mouseover', (event, d) => {
```

```
    tooltip.transition().duration(200).style('opacity', 0.9);
    tooltip.html(
      `<strong>Category:</strong> ${d.category}<br/>
       <strong>X:</strong> ${d.x}<br/>
       <strong>Y:</strong> ${d.y}`
    )
    .style('left', (event.pageX + 10) + 'px')
    .style('top', (event.pageY - 28) + 'px');
  })
  .on('mouseout', () => {
    tooltip.transition().duration(500).style('opacity', 0);
  });
```

5.3.7 Full Example

Here is the complete HTML + JavaScript code you can use:

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Scatter Plot</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body { font-family: sans-serif; }
  </style>
</head>
<body>
  <svg width="600" height="400"></svg>

  <script>
    const data = [
      { x: 34, y: 78, category: 'A' },
      { x: 109, y: 280, category: 'B' },
      { x: 310, y: 120, category: 'A' },
      { x: 79, y: 411, category: 'C' },
      { x: 420, y: 220, category: 'B' },
      { x: 233, y: 145, category: 'C' },
    ];

    const svgWidth = 600;
    const svgHeight = 400;
    const margin = { top: 20, right: 30, bottom: 50, left: 60 };

    const width = svgWidth - margin.left - margin.right;
    const height = svgHeight - margin.top - margin.bottom;

    const svg = d3.select('svg')
      .attr('width', svgWidth)
      .attr('height', svgHeight);
```

```
  const chartGroup = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.x)]).nice()
    .range([0, width]);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.y)]).nice()
    .range([height, 0]);

  const colorScale = d3.scaleOrdinal()
    .domain([...new Set(data.map(d => d.category))])
    .range(d3.schemeCategory10);

  const xAxis = d3.axisBottom(xScale);
  const yAxis = d3.axisLeft(yScale);

  chartGroup.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(xAxis)
    .append('text')
    .attr('x', width / 2)
    .attr('y', 40)
    .attr('fill', 'black')
    .attr('text-anchor', 'middle')
    .attr('font-size', '14px')
    .text('X Value');

  chartGroup.append('g')
    .call(yAxis)
    .append('text')
    .attr('transform', 'rotate(-90)')
    .attr('x', -height / 2)
    .attr('y', -45)
    .attr('fill', 'black')
    .attr('text-anchor', 'middle')
    .attr('font-size', '14px')
    .text('Y Value');

  chartGroup.selectAll('circle')
    .data(data)
    .enter()
    .append('circle')
    .attr('cx', d => xScale(d.x))
    .attr('cy', d => yScale(d.y))
    .attr('r', 7)
    .attr('fill', d => colorScale(d.category))
    .attr('opacity', 0.8);

  const tooltip = d3.select('body').append('div')
    .style('position', 'absolute')
    .style('background', '#fff')
    .style('padding', '6px 10px')
    .style('border', '1px solid #ccc')
    .style('border-radius', '4px')
    .style('pointer-events', 'none')
    .style('opacity', 0);
```

```
    chartGroup.selectAll('circle')
      .on('mouseover', (event, d) => {
        tooltip.transition().duration(200).style('opacity', 0.9);
        tooltip.html(
          `<strong>Category:</strong> ${d.category}<br/>
           <strong>X:</strong> ${d.x}<br/>
           <strong>Y:</strong> ${d.y}`
        )
        .style('left', (event.pageX + 10) + 'px')
        .style('top', (event.pageY - 28) + 'px');
      })
      .on('mouseout', () => {
        tooltip.transition().duration(500).style('opacity', 0);
      });
  </script>
</body>
</html>
```

5.3.8 Summary

- Scatter plots display points representing two quantitative variables.
- Use `scaleLinear` to map both axes to pixel ranges.
- Color can encode a third categorical or numeric variable.
- Axes provide context with ticks and labels.
- Tooltips improve usability by revealing details on hover.

5.4 Pie/Donut Charts

Pie and donut charts are popular for showing parts of a whole — how categories contribute proportionally to a total. They represent data as slices of a circle, where each slice's angle corresponds to the category's value.

In this section, you'll learn how to build pie and donut charts using D3's `d3.pie()` and `d3.arc()` generators. You'll also see how to use color scales to differentiate slices and add labels or legends for clarity.

5.4.1 Understanding the Core Concepts

- **`d3.pie()`** processes your data array and computes the start and end angles for each slice.
- **`d3.arc()`** generates the SVG path commands needed to draw each slice based on those

angles.

- For donut charts, you set an inner radius to create a hole in the center.

5.4.2 Step 1: Prepare Your Data and SVG

Here's a sample dataset representing sales by category:

```
const data = [
  { category: 'Books', value: 30 },
  { category: 'Electronics', value: 70 },
  { category: 'Clothing', value: 45 },
  { category: 'Sports', value: 20 }
];
```

Create an SVG container:

```
<svg width="400" height="400"></svg>
```

5.4.3 Step 2: Set Dimensions and Radius

```
const width = 400;
const height = 400;
const margin = 40;

const radius = Math.min(width, height) / 2 - margin;
```

5.4.4 Step 3: Create Color Scale

Use a categorical color scale to assign different colors to each slice:

```
const color = d3.scaleOrdinal()
  .domain(data.map(d => d.category))
  .range(d3.schemeCategory10);
```

5.4.5 Step 4: Generate Pie Layout and Arc Path

```
const pie = d3.pie()
  .value(d => d.value)
  .sort(null);  // Keep original data order

const arc = d3.arc()
  .innerRadius(0)         // For pie chart; use > 0 for donut
```

```
  .outerRadius(radius);
```

5.4.6 Step 5: Append Group and Draw Slices

```
const svg = d3.select('svg')
  .attr('width', width)
  .attr('height', height)
  .append('g')
  .attr('transform', `translate(${width/2},${height/2})`);  // Center group

const arcs = svg.selectAll('arc')
  .data(pie(data))
  .enter()
  .append('path')
  .attr('d', arc)
  .attr('fill', d => color(d.data.category))
  .attr('stroke', 'white')
  .style('stroke-width', '2px');
```

5.4.7 Step 6: Add Labels to Slices

Place labels at the centroid of each slice:

```
svg.selectAll('text')
  .data(pie(data))
  .enter()
  .append('text')
  .attr('transform', d => `translate(${arc.centroid(d)})`)
  .attr('text-anchor', 'middle')
  .attr('font-size', '14px')
  .attr('fill', 'white')
  .text(d => d.data.category);
```

5.4.8 Step 7: Creating a Donut Chart

To convert the pie chart into a donut chart, simply set a non-zero `innerRadius`:

```
const donutArc = d3.arc()
  .innerRadius(radius * 0.5)  // Creates the hole (50% radius)
  .outerRadius(radius);

svg.selectAll('path')
  .data(pie(data))
  .join('path')
  .attr('d', donutArc)
  .attr('fill', d => color(d.data.category))
```

```
  .attr('stroke', 'white')
  .style('stroke-width', '2px');
```

5.4.9 Full Example: Pie Chart with Labels

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Pie Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body { font-family: sans-serif; }
  </style>
</head>
<body>
  <svg width="400" height="400"></svg>

  <script>
    const data = [
      { category: 'Books', value: 30 },
      { category: 'Electronics', value: 70 },
      { category: 'Clothing', value: 45 },
      { category: 'Sports', value: 20 }
    ];

    const width = 400;
    const height = 400;
    const margin = 40;
    const radius = Math.min(width, height) / 2 - margin;

    const color = d3.scaleOrdinal()
      .domain(data.map(d => d.category))
      .range(d3.schemeCategory10);

    const pie = d3.pie()
      .value(d => d.value)
      .sort(null);

    const arc = d3.arc()
      .innerRadius(0)  // Pie chart
      .outerRadius(radius);

    const svg = d3.select('svg')
      .attr('width', width)
      .attr('height', height)
      .append('g')
      .attr('transform', `translate(${width / 2},${height / 2})`);

    // Draw slices
    svg.selectAll('path')
```

```
      .data(pie(data))
      .enter()
      .append('path')
      .attr('d', arc)
      .attr('fill', d => color(d.data.category))
      .attr('stroke', 'white')
      .style('stroke-width', '2px');

    // Add labels
    svg.selectAll('text')
      .data(pie(data))
      .enter()
      .append('text')
      .attr('transform', d => `translate(${arc.centroid(d)})`)
      .attr('text-anchor', 'middle')
      .attr('font-size', '14px')
      .attr('fill', 'white')
      .text(d => d.data.category);
  </script>
</body>
</html>
```

5.4.10 Summary

- Use `d3.pie()` to compute slice angles from categorical data values.
- Use `d3.arc()` to generate SVG paths for pie or donut slices.
- Use a color scale to visually differentiate slices.
- Add text labels positioned at slice centroids for clarity.
- Convert a pie chart into a donut by setting a non-zero `innerRadius`.

Chapter 6.

D3.js Adding Interactivity

1. Tooltips and Hover Effects
2. Click Events and Dynamic Highlights
3. Filtering and Live Updates

6 D3.js Adding Interactivity

6.1 Tooltips and Hover Effects

Adding tooltips and hover effects to your visualizations greatly enhances user experience by providing dynamic, contextual information. Tooltips display detailed data when users hover over elements, making charts more interactive and informative without cluttering the view.

In this section, we'll learn how to:

- Create a styled tooltip element,
- Use D3's `mouseover`, `mousemove`, and `mouseout` events to control tooltip visibility and positioning,
- Apply tooltips to a bar or scatter plot.

6.1.1 Creating the Tooltip Element

First, add a `<div>` element that will serve as the tooltip. This element is initially hidden and styled to appear as a floating box.

```
const tooltip = d3.select('body')
  .append('div')
  .attr('class', 'tooltip')
  .style('position', 'absolute')
  .style('background', '#f9f9f9')
  .style('padding', '8px')
  .style('border', '1px solid #ccc')
  .style('border-radius', '4px')
  .style('pointer-events', 'none')  // Ignore mouse events
  .style('opacity', 0);
```

Add this CSS (either in a `<style>` tag or stylesheet) to style the tooltip smoothly:

```
.tooltip {
  font-family: sans-serif;
  font-size: 13px;
  color: #333;
  box-shadow: 0px 2px 4px rgba(0,0,0,0.2);
  transition: opacity 0.3s;
}
```

6.1.2 Attaching Tooltip Events

Use D3 event listeners on chart elements to control tooltip behavior:

- **`mouseover`**: Show the tooltip and populate it with relevant data.
- **`mousemove`**: Update the tooltip's position to follow the cursor.

- **mouseout**: Hide the tooltip.

Example event handlers:

```
selection
  .on('mouseover', (event, d) => {
    tooltip.style('opacity', 0.9)
      .html(`Value: <strong>${d.value}</strong>`);
  })
  .on('mousemove', (event) => {
    tooltip.style('left', (event.pageX + 10) + 'px')
      .style('top', (event.pageY - 28) + 'px');
  })
  .on('mouseout', () => {
    tooltip.style('opacity', 0);
  });
```

6.1.3 Working Example: Tooltip on a Bar Chart

Here's a complete example demonstrating tooltips on a simple bar chart.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart with Tooltips</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    .tooltip {
      position: absolute;
      background: #f9f9f9;
      padding: 8px;
      border: 1px solid #ccc;
      border-radius: 4px;
      pointer-events: none;
      font-family: sans-serif;
      font-size: 13px;
      color: #333;
      box-shadow: 0px 2px 4px rgba(0,0,0,0.2);
      transition: opacity 0.3s;
      opacity: 0;
    }
  </style>
</head>
<body>

<svg width="600" height="400"></svg>

<script>
  const data = [
    { name: 'A', value: 30 },
    { name: 'B', value: 80 },
```

```
    { name: 'C', value: 45 },
    { name: 'D', value: 60 },
    { name: 'E', value: 20 },
  ];

  const svg = d3.select('svg');
  const margin = { top: 20, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr('width') - margin.left - margin.right;
  const height = +svg.attr('height') - margin.top - margin.bottom;

  const chart = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleBand()
    .domain(data.map(d => d.name))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([height, 0]);

  chart.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(d3.axisBottom(xScale));

  chart.append('g')
    .call(d3.axisLeft(yScale));

  // Create tooltip div
  const tooltip = d3.select('body').append('div')
    .attr('class', 'tooltip');

  // Draw bars
  chart.selectAll('.bar')
    .data(data)
    .enter()
    .append('rect')
    .attr('class', 'bar')
    .attr('x', d => xScale(d.name))
    .attr('y', d => yScale(d.value))
    .attr('width', xScale.bandwidth())
    .attr('height', d => height - yScale(d.value))
    .attr('fill', '#4682b4')
    .on('mouseover', (event, d) => {
      tooltip.style('opacity', 0.9)
        .html(`<strong>${d.name}</strong><br/>Value: ${d.value}`);
    })
    .on('mousemove', (event) => {
      tooltip.style('left', (event.pageX + 10) + 'px')
        .style('top', (event.pageY - 28) + 'px');
    })
    .on('mouseout', () => {
      tooltip.style('opacity', 0);
    });
</script>
```

```
</body>
</html>
```

6.1.4 Explanation

- We create a hidden tooltip `<div>` styled for readability.
- Bars listen for mouse events:
  - On **`mouseover`**, tooltip content is set and opacity increased to show it.
  - On **`mousemove`**, tooltip follows the cursor using page coordinates.
  - On **`mouseout`**, tooltip fades out.
- `pointer-events: none` on the tooltip ensures it doesn't block mouse interaction with the chart.

6.1.5 Summary

- Tooltips improve user engagement by showing data on demand.
- Use D3 event listeners `mouseover`, `mousemove`, and `mouseout` to control tooltip behavior.
- Position tooltips near the cursor using event coordinates.
- Style tooltips with CSS for smooth appearance and clear readability.

6.2 Click Events and Dynamic Highlights

Click events provide another powerful layer of interactivity in your visualizations. They allow users to select, highlight, or explore data points in detail, creating more engaging and insightful charts.

In this section, we will learn how to:

- Bind click event listeners to chart elements using `.on("click", ...)`,
- Change styles dynamically to highlight selected elements,
- Implement toggling behavior to select/deselect,
- Discuss how click events can update other parts of a dashboard or show detail panels.

6.2.1 Binding Click Events to Elements

D3's `.on()` method attaches event listeners to selections. For example, to add a click handler on bar elements:

```
bars.on('click', (event, d) => {
  console.log('Clicked on:', d);
});
```

6.2.2 Highlighting Selected Elements

Highlighting typically involves changing styles such as fill color, stroke, or opacity. You can store the selection state by adding/removing CSS classes or using data properties.

6.2.3 Example: Toggle Bar Highlight

```
bars.on('click', function(event, d) {
  const isSelected = d3.select(this).classed('selected');
  d3.select(this).classed('selected', !isSelected);
});
```

Add CSS:

```
.selected {
  fill: orange;
  stroke: black;
  stroke-width: 2px;
}
```

This toggles the "selected" class, visually distinguishing clicked bars.

6.2.4 Example: Click to Highlight Bars and Show Details

Let's extend the bar chart example from the previous section by adding click highlights and a detail pane:

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
```

```
<body>
<div id="details" style="margin-top: 20px; font-family: sans-serif;"></div>
<svg width="600" height="400"></svg>
<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
  const data = [
    { name: 'A', value: 30, description: 'Category A details' },
    { name: 'B', value: 80, description: 'Category B details' },
    { name: 'C', value: 45, description: 'Category C details' },
    { name: 'D', value: 60, description: 'Category D details' },
    { name: 'E', value: 20, description: 'Category E details' },
  ];

  const svg = d3.select('svg');
  const margin = { top: 20, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr('width') - margin.left - margin.right;
  const height = +svg.attr('height') - margin.top - margin.bottom;

  const chart = svg.append('g')
    .attr('transform', `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleBand()
    .domain(data.map(d => d.name))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([height, 0]);

  chart.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(d3.axisBottom(xScale));

  chart.append('g')
    .call(d3.axisLeft(yScale));

  const details = d3.select('#details');

  const bars = chart.selectAll('.bar')
    .data(data)
    .enter()
    .append('rect')
    .attr('class', 'bar')
    .attr('x', d => xScale(d.name))
    .attr('y', d => yScale(d.value))
    .attr('width', xScale.bandwidth())
    .attr('height', d => height - yScale(d.value))
    .attr('fill', '#4682b4')
    .style('cursor', 'pointer')
    .on('click', function(event, d) {
      // Toggle selection
      const selected = d3.select(this).classed('selected');
      // Clear previous selections
      bars.classed('selected', false).attr('fill', '#4682b4');
      if (!selected) {
        d3.select(this).classed('selected', true).attr('fill', 'orange');
```

```
        details.html(`<strong>${d.name}</strong>: ${d.description}`);
      } else {
        details.html('');
      }
    });
</script>
<style>
  .selected {
    stroke: black;
    stroke-width: 2px;
  }
</style>
</body>
</html>
```

6.2.5 Explanation

- When a bar is clicked, the click handler toggles the "selected" class.
- The fill color changes to orange to highlight the selected bar.
- Only one bar can be selected at a time; clicking the selected bar deselects it.
- A detail pane below the chart updates with the selected bar's description.
- The cursor changes to pointer to indicate interactivity.

6.2.6 Advanced Uses of Click Events

Click interactions can be extended to:

- **Toggle visibility**: Show/hide subsets of data.
- **Update auxiliary charts**: Clicking a segment can filter or highlight related data elsewhere.
- **Open detail panes or modals**: Provide in-depth information or drill-down views.
- **Multi-select or lasso**: Allow selecting multiple elements with shift-click or dragging.

6.2.7 Summary

- Use `.on('click', ...)` to bind click handlers to elements.
- Change styles dynamically to highlight or toggle selection.
- Use click events to update related UI components like detail panels.
- Design clear visual feedback for selections (color, stroke, opacity).
- Think about extending interactivity for coordinated multiple views.

6.3 Filtering and Live Updates

Filtering allows users to explore subsets of data dynamically by interacting with controls such as buttons, dropdowns, or sliders. Coupled with live updates and smooth transitions, filtering makes visualizations more engaging and insightful.

In this section, we'll learn how to:

- Create UI controls to filter data (e.g., buttons, dropdowns),
- Use D3's `.data()` method to re-bind filtered data,
- Handle the enter, update, and exit selections properly,
- Apply transitions for smooth element removal and addition,
- Build a working example that filters data by category with immediate visual feedback.

6.3.1 Setting Up Filtering Controls

Filtering controls can be simple HTML elements like buttons or dropdown menus:

```
<select id="categoryFilter">
  <option value="All">All</option>
  <option value="A">Category A</option>
  <option value="B">Category B</option>
  <option value="C">Category C</option>
</select>
```

Users can pick a category to filter the displayed data.

6.3.2 Re-binding Data and Managing Selections

When data changes (after filtering), you need to:

1. **Re-bind data** with `.data(filteredData, keyFunction)` to ensure proper matching,
2. **Handle the Enter selection** to create new elements for added data,
3. **Handle the Update selection** to update existing elements,
4. **Handle the Exit selection** to remove elements no longer present,
5. Use **transitions** to animate these changes smoothly.

6.3.3 Example: Filtering a Scatter Plot by Category

Sample Data

```
const data = [
  { x: 34, y: 78, category: 'A' },
```

```
  { x: 109, y: 280, category: 'B' },
  { x: 310, y: 120, category: 'A' },
  { x: 79, y: 411, category: 'C' },
  { x: 420, y: 220, category: 'B' },
  { x: 233, y: 145, category: 'C' },
];
```

HTML Controls and SVG

```
<select id="categoryFilter">
  <option value="All">All</option>
  <option value="A">Category A</option>
  <option value="B">Category B</option>
  <option value="C">Category C</option>
</select>

<svg width="600" height="400"></svg>
```

JavaScript: Drawing and Filtering

```
const svg = d3.select('svg');
const margin = { top: 20, right: 30, bottom: 50, left: 60 };
const width = +svg.attr('width') - margin.left - margin.right;
const height = +svg.attr('height') - margin.top - margin.bottom;

const chart = svg.append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`);

const xScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.x)]).nice()
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.y)]).nice()
  .range([height, 0]);

const colorScale = d3.scaleOrdinal()
  .domain(['A', 'B', 'C'])
  .range(d3.schemeCategory10);

// Axes
chart.append('g')
  .attr('transform', `translate(0,${height})`)
  .call(d3.axisBottom(xScale));

chart.append('g')
  .call(d3.axisLeft(yScale));

// Initial draw
function update(filteredData) {
  // DATA JOIN with key function (by x and y for uniqueness)
  const circles = chart.selectAll('circle')
    .data(filteredData, d => d.x + '-' + d.y);

  // EXIT old elements not in new data
  circles.exit()
```

```
    .transition()
    .duration(500)
    .attr('r', 0)
    .remove();

  // UPDATE existing elements
  circles
    .transition()
    .duration(500)
    .attr('cx', d => xScale(d.x))
    .attr('cy', d => yScale(d.y))
    .attr('fill', d => colorScale(d.category))
    .attr('r', 7);

  // ENTER new elements
  circles.enter()
    .append('circle')
    .attr('cx', d => xScale(d.x))
    .attr('cy', d => yScale(d.y))
    .attr('r', 0) // start from 0 radius for animation
    .attr('fill', d => colorScale(d.category))
    .transition()
    .duration(500)
    .attr('r', 7);
}

// Initial render with all data
update(data);

// Filtering handler
d3.select('#categoryFilter').on('change', function() {
  const selected = this.value;
  const filteredData = selected === 'All' ? data : data.filter(d => d.category === selected);
  update(filteredData);
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Filtering Demo</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 40px;
      text-align: center;
    }

    select {
      font-size: 16px;
      padding: 6px 10px;
      margin-bottom: 20px;
    }
```

```
    svg {
      border: 1px solid #ccc;
      display: block;
      margin: 0 auto;
    }
  </style>
</head>
<body>

  <h1>D3 Scatter Plot with Category Filter</h1>

  <label for="categoryFilter">Filter by Category:</label>
  <select id="categoryFilter">
    <option value="All">All</option>
    <option value="A">Category A</option>
    <option value="B">Category B</option>
    <option value="C">Category C</option>
  </select>

  <svg width="600" height="400"></svg>

  <script>
    const data = [
      { x: 34, y: 78, category: 'A' },
      { x: 109, y: 280, category: 'B' },
      { x: 310, y: 120, category: 'A' },
      { x: 79, y: 411, category: 'C' },
      { x: 420, y: 220, category: 'B' },
      { x: 233, y: 145, category: 'C' },
    ];

    const svg = d3.select('svg');
    const margin = { top: 20, right: 30, bottom: 50, left: 60 };
    const width = +svg.attr('width') - margin.left - margin.right;
    const height = +svg.attr('height') - margin.top - margin.bottom;

    const chart = svg.append('g')
      .attr('transform', `translate(${margin.left},${margin.top})`);

    const xScale = d3.scaleLinear()
      .domain([0, d3.max(data, d => d.x)]).nice()
      .range([0, width]);

    const yScale = d3.scaleLinear()
      .domain([0, d3.max(data, d => d.y)]).nice()
      .range([height, 0]);

    const colorScale = d3.scaleOrdinal()
      .domain(['A', 'B', 'C'])
      .range(d3.schemeCategory10);

    // Axes
    chart.append('g')
      .attr('transform', `translate(0,${height})`)
      .call(d3.axisBottom(xScale));

    chart.append('g')
      .call(d3.axisLeft(yScale));
```

```
    function update(filteredData) {
      const circles = chart.selectAll('circle')
        .data(filteredData, d => d.x + '-' + d.y);

      circles.exit()
        .transition()
        .duration(500)
        .attr('r', 0)
        .remove();

      circles.transition()
        .duration(500)
        .attr('cx', d => xScale(d.x))
        .attr('cy', d => yScale(d.y))
        .attr('fill', d => colorScale(d.category))
        .attr('r', 7);

      circles.enter()
        .append('circle')
        .attr('cx', d => xScale(d.x))
        .attr('cy', d => yScale(d.y))
        .attr('r', 0)
        .attr('fill', d => colorScale(d.category))
        .transition()
        .duration(500)
        .attr('r', 7);
    }

    update(data);

    d3.select('#categoryFilter').on('change', function() {
      const selected = this.value;
      const filteredData = selected === 'All'
        ? data
        : data.filter(d => d.category === selected);
      update(filteredData);
    });
  </script>
</body>
</html>
```

6.3.4 How This Works

- When the filter changes, we create a filtered subset of the data.
- Calling `update(filteredData)` re-binds the filtered data to the circles.
- The **exit** selection removes circles that are no longer relevant, animating their radius to zero before removal.
- The **update** selection animates position and color changes for existing circles.
- The **enter** selection appends new circles, animating their radius from zero to full size.
- This creates smooth transitions that visually guide users through the data changes.

6.3.5 Extending This Approach

- Use **buttons** or **sliders** to filter by categories, ranges, or dates.
- Combine multiple filters for advanced slicing.
- Update axes scales if the data domain changes dynamically.
- Link filtering with other charts for coordinated views.
- Animate transitions of position, color, size, or opacity for richer feedback.

6.3.6 Summary

- Filtering updates the displayed dataset dynamically.
- Use `.data()` with a key function to maintain element identity.
- Handle enter, update, and exit selections to add/remove/update elements.
- Use transitions for smooth animations during filtering.
- UI controls trigger filtering logic and live chart updates.

Chapter 7.

D3.js Animating Data

1. Transitions and Tweens in D3
2. Smoothing and Timing Functions
3. Animating Between States

7 D3.js Animating Data

7.1 Transitions and Tweens in D3

Animating your data visualizations enhances their appeal and helps users understand changes over time or between states. D3's powerful transition system makes it easy to animate changes smoothly by interpolating values — a process known as *tweening*.

In this section, we'll cover:

- How to create transitions using `.transition()` and control their duration with `.duration()`,
- The concept of tweening — interpolating between starting and ending values,
- Animating attributes and styles,
- A simple example showing bars growing to their height and dots moving to new positions.

7.1.1 Creating Transitions

A **transition** in D3 is a special selection that animates changes in attributes, styles, or other properties over time.

You create a transition by chaining `.transition()` onto a selection:

```
selection.transition()
  .duration(1000)  // animation lasts 1000ms (1 second)
  .attr('height', newHeight)
  .attr('y', newY);
```

This smoothly interpolates the attribute values from their current state to the specified new values.

7.1.2 Tweening: Interpolating Between Values

Behind the scenes, D3 calculates intermediate values for each frame between the old and new states, a process called *tweening*.

For example, when you change the height of a bar from 0 to 100 pixels, D3 automatically computes the heights for frames between 0 and 100, creating smooth growth.

You can also define *custom tween functions* for more complex interpolations, but basic attribute and style changes are automatically tweened by D3.

7.1.3 Animating Attributes and Styles

D3 supports animating any SVG attribute or CSS style that can be interpolated numerically. Common animated properties include:

- `x`, `y`, `width`, `height` for SVG shapes,
- `fill`, `stroke` colors,
- `opacity` for fade effects,
- `transform` for position, rotation, or scaling.

Example of animating color:

```
selection.transition()
  .duration(800)
  .style('fill', 'orange');
```

7.1.4 Example 1: Bars Growing on Enter

Let's create a simple bar chart where bars grow from height 0 to their data-driven heights when added:

```
const data = [30, 80, 45, 60, 20];

const svg = d3.select('svg')
  .attr('width', 500)
  .attr('height', 300);

const xScale = d3.scaleBand()
  .domain(data.map((d,i) => i))
  .range([0, 480])
  .padding(0.2);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data)])
  .range([250, 0]);

const chart = svg.append('g').attr('transform', 'translate(10,10)');

chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('x', (d,i) => xScale(i))
  .attr('width', xScale.bandwidth())
  .attr('y', yScale(0))          // Start at baseline (height 0)
  .attr('height', 0)             // Initial height 0
  .attr('fill', 'steelblue')
  .transition()
  .duration(1000)
  .attr('y', d => yScale(d))     // Animate y up to data value
  .attr('height', d => 250 - yScale(d));  // Animate height growing
```

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Animated Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 40px;
    }
    svg {
      border: 1px solid #ccc;
      display: block;
      margin: 0 auto;
    }
  </style>
</head>
<body>

  <h1>Animated D3 Bar Chart</h1>
  <svg width="500" height="300"></svg>

  <script>
    const data = [30, 80, 45, 60, 20];

    const svg = d3.select('svg');
    const chartMargin = { top: 10, right: 10, bottom: 40, left: 10 };
    const chartWidth = +svg.attr('width') - chartMargin.left - chartMargin.right;
    const chartHeight = +svg.attr('height') - chartMargin.top - chartMargin.bottom;

    const xScale = d3.scaleBand()
      .domain(data.map((_, i) => i))
      .range([0, chartWidth])
      .padding(0.2);

    const yScale = d3.scaleLinear()
      .domain([0, d3.max(data)])
      .range([chartHeight, 0]);

    const chart = svg.append('g')
      .attr('transform', `translate(${chartMargin.left},${chartMargin.top})`);

    chart.selectAll('rect')
      .data(data)
      .enter()
      .append('rect')
      .attr('x', (_, i) => xScale(i))
      .attr('width', xScale.bandwidth())
      .attr('y', yScale(0))       // Start at bottom
      .attr('height', 0)          // Start with height 0
      .attr('fill', 'steelblue')
      .transition()
      .duration(1000)
      .attr('y', d => yScale(d))  // Animate upward
      .attr('height', d => chartHeight - yScale(d));  // Animate growth
  </script>
```

```
</body>
</html>
```

7.1.5 Example 2: Moving Dots to New Positions

Transitions can also animate positions. Imagine updating a scatter plot where circles move smoothly to new data points:

```
const oldData = [
  {x: 30, y: 50},
  {x: 70, y: 90},
  {x: 110, y: 30},
];

const newData = [
  {x: 100, y: 150},
  {x: 200, y: 80},
  {x: 250, y: 130},
];

const svg = d3.select('svg')
  .attr('width', 400)
  .attr('height', 200);

const circles = svg.selectAll('circle')
  .data(oldData)
  .enter()
  .append('circle')
  .attr('cx', d => d.x)
  .attr('cy', d => d.y)
  .attr('r', 10)
  .attr('fill', 'teal');

// Later update positions with transition
svg.selectAll('circle')
  .data(newData)
  .transition()
  .duration(1500)
  .attr('cx', d => d.x)
  .attr('cy', d => d.y);
```

The circles smoothly move from their old to new `(x,y)` positions over 1.5 seconds.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Circle Transition</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
```

```
      font-family: sans-serif;
      text-align: center;
      padding: 40px;
    }
    svg {
      border: 1px solid #ccc;
      margin-top: 20px;
      display: block;
      margin-left: auto;
      margin-right: auto;
    }
    button {
      margin-top: 20px;
      font-size: 16px;
      padding: 8px 16px;
      cursor: pointer;
    }
  </style>
</head>
<body>

  <h1>D3 Circle Transition Demo</h1>
  <svg></svg>
  <button id="animateBtn">Animate to New Data</button>

  <script>
    const oldData = [
      {x: 30, y: 50},
      {x: 70, y: 90},
      {x: 110, y: 30},
    ];

    const newData = [
      {x: 100, y: 150},
      {x: 200, y: 80},
      {x: 250, y: 130},
    ];

    const svg = d3.select('svg')
      .attr('width', 400)
      .attr('height', 200);

    svg.selectAll('circle')
      .data(oldData)
      .enter()
      .append('circle')
      .attr('cx', d => d.x)
      .attr('cy', d => d.y)
      .attr('r', 10)
      .attr('fill', 'teal');

    document.getElementById('animateBtn').addEventListener('click', () => {
      svg.selectAll('circle')
        .data(newData)
        .transition()
        .duration(1500)
        .attr('cx', d => d.x)
        .attr('cy', d => d.y);
```

```
    });
  </script>

</body>
</html>
```

7.1.6 Summary

- Use `.transition()` to animate changes in selections.
- Control animation speed with `.duration()`.
- D3 interpolates attribute and style values between their old and new states — this is tweening.
- Animate bar height, position, color, opacity, and more.
- Transitions add polish and clarity to data updates, helping users track changes visually.

7.2 Smoothing and Timing Functions

Animations are more engaging and natural when their pacing feels smooth and intentional, rather than mechanical or abrupt. D3 provides a set of **easing functions** that control how the animation progresses over time — essentially, how speed changes during the transition.

In this section, we'll learn about:

- What easing functions are and why they matter,
- Common easing functions like `easeLinear`, `easeCubic`, `easeBounce`, and more,
- How to apply easing functions in D3 transitions,
- A comparison example showing different easing effects side-by-side.

7.2.1 What Are Easing Functions?

An easing function defines the rate of change of a parameter over time. Instead of moving at a constant speed, easing can make animations:

- Start slow and speed up (ease-in),
- Start fast and slow down (ease-out),
- Start and end slowly, speeding up in the middle (ease-in-out),
- Bounce or elastic movements.

This makes animations feel more natural and visually pleasing.

7.2.2 Common D3 Easing Functions

D3 provides many built-in easing functions in the `d3.ease*` namespace. Here are a few popular ones:

Easing Function	Description
`d3.easeLinear`	Constant speed, no easing
`d3.easeCubicIn`	Starts slowly, accelerates
`d3.easeCubicOut`	Starts fast, decelerates
`d3.easeCubicInOut`	Slow start and end, fast middle
`d3.easeBounce`	Bounce effect, like dropping a ball
`d3.easeElastic`	Elastic effect with overshoot

7.2.3 Applying Easing to Transitions

You apply easing to a transition using the `.ease()` method:

```
selection.transition()
  .duration(1500)
  .ease(d3.easeCubicInOut)
  .attr('x', 200);
```

This transition will move the element's `x` position to 200, accelerating slowly at the start and decelerating slowly at the end.

7.2.4 Example: Comparing Different Easing Functions

Below is an example that animates three circles moving horizontally using different easing functions. You'll observe how the pace of their motion changes.

```
<svg width="600" height="100"></svg>
<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
    const svg = d3.select('svg');

    const easingFunctions = [
      { name: 'easeLinear', func: d3.easeLinear },
      { name: 'easeCubicInOut', func: d3.easeCubicInOut },
      { name: 'easeBounce', func: d3.easeBounce },
    ];

    // Create circles
    const circles = svg.selectAll('circle')
      .data(easingFunctions)
      .enter()
      .append('circle')
```

```
      .attr('cx', 50)
      .attr('cy', (d, i) => 30 + i * 30)
      .attr('r', 12)
      .attr('fill', 'steelblue');

    // Create labels
    svg.selectAll('text')
      .data(easingFunctions)
      .enter()
      .append('text')
      .attr('x', 10)
      .attr('y', (d, i) => 30 + i * 30 + 5)
      .text(d => d.name) // FIXED HERE
      .attr('font-family', 'sans-serif')
      .attr('font-size', 12)
      .attr('fill', '#333');

    // Animate each circle individually using its own easing function
    function animate() {
      circles.each(function(d) {
        d3.select(this)
          .transition()
          .duration(2000)
          .ease(d.func) // FIXED HERE
          .attr('cx', 550)
          .transition()
          .duration(2000)
          .ease(d.func)
          .attr('cx', 50)
          .on('end', animate);
      });
    }

  animate();
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Easing Demo</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 40px;
    }

    svg {
      border: 1px solid #ccc;
      display: block;
      margin: 0 auto;
    }
  </style>
```

```
</head>
<body>

  <h1>D3 Easing Functions Animation</h1>
  <svg width="600" height="100"></svg>

  <script>
    const svg = d3.select('svg');

    const easingFunctions = [
      { name: 'easeLinear', func: d3.easeLinear },
      { name: 'easeCubicInOut', func: d3.easeCubicInOut },
      { name: 'easeBounce', func: d3.easeBounce },
    ];

    // Create circles
    const circles = svg.selectAll('circle')
      .data(easingFunctions)
      .enter()
      .append('circle')
      .attr('cx', 50)
      .attr('cy', (d, i) => 30 + i * 30)
      .attr('r', 12)
      .attr('fill', 'steelblue');

    // Create labels
    svg.selectAll('text')
      .data(easingFunctions)
      .enter()
      .append('text')
      .attr('x', 10)
      .attr('y', (d, i) => 30 + i * 30 + 5)
      .text(d => d.name) // FIXED HERE
      .attr('font-family', 'sans-serif')
      .attr('font-size', 12)
      .attr('fill', '#333');

    // Animate each circle individually using its own easing function
    function animate() {
      circles.each(function(d) {
        d3.select(this)
          .transition()
          .duration(2000)
          .ease(d.func) // FIXED HERE
          .attr('cx', 550)
          .transition()
          .duration(2000)
          .ease(d.func)
          .attr('cx', 50)
          .on('end', animate);
      });
    }

    animate();
  </script>

</body>
</html>
```

7.2.5 What Youll See

- The **top circle** moves at a constant speed (`easeLinear`).
- The **middle circle** starts and ends slowly but moves fastest in the middle (`easeCubicInOut`).
- The **bottom circle** "bounces" as it reaches the end position (`easeBounce`).

The looped animation demonstrates how easing functions can completely change the feel of a motion.

7.2.6 When to Use Different Easing Styles

- Use **linear easing** for simple, uniform movements.
- Use **cubic or quadratic easing** for smooth and natural acceleration/deceleration.
- Use **bounce or elastic easing** for playful or attention-grabbing effects.
- Match easing style to the story your visualization wants to tell.

7.2.7 Summary

- Easing functions control the pacing of animations.
- D3 offers a rich library of easing options (`easeLinear`, `easeCubicInOut`, `easeBounce`, etc.).
- Apply easing with `.ease()` inside a transition.
- Experiment with easing to create animations that feel natural, smooth, or dramatic.
- Visual comparisons help in choosing the right easing for your visualization.

7.3 Animating Between States

One of the most powerful uses of animation in data visualization is smoothly transitioning between different *states* or *views* of the data. For example, toggling between monthly and yearly summaries, or morphing a stacked bar chart into a grouped bar chart.

This section guides you through structuring your code to animate between such states, focusing on:

- Preparing data for different states,
- Interpolating values between states,
- Using D3's transitions to smoothly morph chart elements,
- Writing clean, readable, and performant code.

7.3.1 Understanding State Transitions

A *state* here means a particular arrangement or representation of your data in the visualization — for example:

- Monthly sales shown as grouped bars,
- Yearly sales shown as stacked bars,
- Data sorted by one category versus another.

Animating between these states involves changing attributes like position, height, color, or shape over time to help users visually track the transition.

7.3.2 Example: Morphing Stacked Bars to Grouped Bars

Data Setup

Imagine sales data split by categories over months:

```
const data = [
  { month: 'Jan', A: 30, B: 20, C: 50 },
  { month: 'Feb', A: 40, B: 35, C: 25 },
  { month: 'Mar', A: 50, B: 30, C: 20 },
];
const categories = ['A', 'B', 'C'];
```

You want to toggle between:

- **Stacked bars**: categories stacked vertically per month,
- **Grouped bars**: categories grouped side-by-side per month.

Preparing Scales

```
const x0 = d3.scaleBand()
  .domain(data.map(d => d.month))
  .rangeRound([0, width])
  .paddingInner(0.1);

const x1 = d3.scaleBand()
  .domain(categories)
  .rangeRound([0, x0.bandwidth()])
  .padding(0.05);

const y = d3.scaleLinear()
  .rangeRound([height, 0])
  .domain([0, d3.max(data, d => d3.sum(categories, k => d[k]))]);
```

Drawing Bars in Stacked Mode

Using D3's `stack` function to compute stacked positions:

```
const stack = d3.stack()
  .keys(categories);
```

```
const series = stack(data);

svg.selectAll('g.layer')
  .data(series)
  .enter()
  .append('g')
  .attr('class', 'layer')
  .attr('fill', (d, i) => color(categories[i]))
  .selectAll('rect')
  .data(d => d)
  .enter()
  .append('rect')
  .attr('x', d => x0(d.data.month))
  .attr('y', d => y(d[1]))
  .attr('height', d => y(d[0]) - y(d[1]))
  .attr('width', x0.bandwidth());
```

Transition to Grouped Bars

In grouped mode, each category's bar sits side-by-side inside each month's band:

```
function transitionGrouped() {
  y.domain([0, d3.max(data, d => d3.max(categories, key => d[key]))]);

  svg.selectAll('g.layer rect')
    .transition()
    .duration(750)
    .delay((d, i) => i * 20)
    .attr('x', d => x0(d.data.month) + x1(d3.select(d3.event.target.parentNode).datum().key))
    .attr('width', x1.bandwidth())
    .transition()
    .attr('y', d => y(d.data[d3.select(d3.event.target.parentNode).datum().key]))
    .attr('height', d => height - y(d.data[d3.select(d3.event.target.parentNode).datum().key]));
}
```

Here, bars animate horizontally from full bandwidth (stacked) to smaller grouped widths, while adjusting vertical position and height.

Transition to Stacked Bars

To morph back:

```
function transitionStacked() {
  y.domain([0, d3.max(data, d => d3.sum(categories, k => d[k]))]);

  svg.selectAll('g.layer rect')
    .transition()
    .duration(750)
    .delay((d, i) => i * 20)
    .attr('x', d => x0(d.data.month))
    .attr('width', x0.bandwidth())
    .transition()
    .attr('y', d => y(d[1]))
    .attr('height', d => y(d[0]) - y(d[1]));
}
```

Bars animate back to stacked vertical positions and widths.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Stacked  Grouped Bar Transition</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 40px;
    }
    button {
      margin: 10px;
      padding: 8px 16px;
      font-size: 14px;
      cursor: pointer;
    }
    svg {
      border: 1px solid #ccc;
      display: block;
      margin: 20px auto;
    }
  </style>
</head>
<body>

  <h1>Stacked  Grouped Bar Chart</h1>
  <button onclick="transitionGrouped()">Grouped</button>
  <button onclick="transitionStacked()">Stacked</button>
  <svg width="600" height="400"></svg>

  <script>
    const data = [
      { month: 'Jan', A: 30, B: 20, C: 50 },
      { month: 'Feb', A: 40, B: 35, C: 25 },
      { month: 'Mar', A: 50, B: 30, C: 20 },
    ];
    const categories = ['A', 'B', 'C'];

    const svg = d3.select('svg');
    const margin = { top: 20, right: 30, bottom: 40, left: 50 };
    const width = +svg.attr('width') - margin.left - margin.right;
    const height = +svg.attr('height') - margin.top - margin.bottom;

    const chart = svg.append('g')
      .attr('transform', `translate(${margin.left},${margin.top})`);

    const color = d3.scaleOrdinal()
      .domain(categories)
      .range(d3.schemeCategory10);

    const x0 = d3.scaleBand()
      .domain(data.map(d => d.month))
      .rangeRound([0, width])
      .paddingInner(0.1);
```

```
  const x1 = d3.scaleBand()
    .domain(categories)
    .rangeRound([0, x0.bandwidth()])
    .padding(0.05);

  const y = d3.scaleLinear()
    .rangeRound([height, 0])
    .domain([0, d3.max(data, d => d3.sum(categories, k => d[k]))]);

  chart.append('g')
    .attr('transform', `translate(0,${height})`)
    .call(d3.axisBottom(x0));

  const yAxis = chart.append('g')
    .call(d3.axisLeft(y));

  const stack = d3.stack().keys(categories);
  const series = stack(data);

  const groups = chart.selectAll('g.layer')
    .data(series)
    .enter()
    .append('g')
    .attr('class', 'layer')
    .attr('fill', d => color(d.key));

  const rects = groups.selectAll('rect')
    .data(d => d.map(v => Object.assign({}, v, { key: d.key })))
    .enter()
    .append('rect')
    .attr('x', d => x0(d.data.month))
    .attr('y', d => y(d[1]))
    .attr('height', d => y(d[0]) - y(d[1]))
    .attr('width', x0.bandwidth());

  function transitionGrouped() {
    y.domain([0, d3.max(data, d => d3.max(categories, key => d[key]))]);
    yAxis.transition().duration(750).call(d3.axisLeft(y));

    rects.transition()
      .duration(750)
      .attr('x', d => x0(d.data.month) + x1(d.key))
      .attr('width', x1.bandwidth())
      .attr('y', d => y(d.data[d.key]))
      .attr('height', d => height - y(d.data[d.key]));
  }

  function transitionStacked() {
    y.domain([0, d3.max(data, d => d3.sum(categories, k => d[k]))]);
    yAxis.transition().duration(750).call(d3.axisLeft(y));

    rects.transition()
      .duration(750)
      .attr('x', d => x0(d.data.month))
      .attr('width', x0.bandwidth())
      .attr('y', d => y(d[1]))
      .attr('height', d => y(d[0]) - y(d[1]));
  }
```

```
  </script>

</body>
</html>
```

7.3.3 Key Points for Smooth Animation

- **Data binding consistency**: Ensure that your data join uses a consistent key function so D3 can match elements across states.
- **Interpolating attributes**: Animate `x`, `y`, `width`, and `height` attributes smoothly.
- **Adjust scales**: Update your scales' domains to fit new data ranges for each state.
- **Use delays**: Staggering transitions can improve visual clarity.
- **Separate logic**: Keep your transition functions modular for readability and maintenance.

7.3.4 Summary

- Animating between states enhances comprehension by visually connecting views.
- Structure code to prepare data and scales for each state.
- Use D3 transitions to smoothly interpolate attributes like position and size.
- Keep data joins consistent and update scales appropriately.
- Write modular transition functions for clarity and performance.

Chapter 8.

Chart.js

1. Fast Setup for Simple Charts
2. Customizing Appearance
3. Responsive Design

8 Chart.js

8.1 Fast Setup for Simple Charts

Chart.js is a popular, beginner-friendly JavaScript library for creating beautiful, responsive charts quickly. Unlike low-level libraries like D3, Chart.js provides ready-to-use chart types—such as bar, line, pie, and more—making it ideal for fast prototyping or dashboards where you want great visuals with minimal code.

8.1.1 Installing and Including Chart.js

You can include Chart.js in your project in several ways:

8.1.2 Via CDN

Add this `<script>` tag inside your HTML `<head>` or before your closing `</body>`:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

8.1.3 Via npm (for build tools)

If you use npm and bundlers like Webpack:

```
npm install chart.js
```

Then import it in your JavaScript:

```
import { Chart, registerables } from 'chart.js';
Chart.register(...registerables);
```

8.1.4 Creating Your First Chart

After including Chart.js, you create charts by:

1. Adding a `<canvas>` element in your HTML where the chart will render,
2. Initializing a chart with `new Chart()` in your JavaScript, passing the canvas context and configuration.

8.1.5 Example: Simple Bar Chart

```
  <canvas id="myBarChart" width="600" height="400"></canvas>
  <script>
    const ctx = document.getElementById('myBarChart').getContext('2d');

    const myBarChart = new Chart(ctx, {
      type: 'bar', // Chart type: bar chart
      data: {
        labels: ['January', 'February', 'March', 'April', 'May'],
        datasets: [{
          label: 'Sales',
          data: [12, 19, 7, 15, 10],
          backgroundColor: 'rgba(54, 162, 235, 0.7)',
          borderColor: 'rgba(54, 162, 235, 1)',
          borderWidth: 1
        }]
      },
      options: {
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Bar Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body>
  <canvas id="myBarChart" width="600" height="400"></canvas>

  <script>
    const ctx = document.getElementById('myBarChart').getContext('2d');

    const myBarChart = new Chart(ctx, {
      type: 'bar', // Chart type: bar chart
      data: {
        labels: ['January', 'February', 'March', 'April', 'May'],
        datasets: [{
          label: 'Sales',
          data: [12, 19, 7, 15, 10],
          backgroundColor: 'rgba(54, 162, 235, 0.7)',
          borderColor: 'rgba(54, 162, 235, 1)',
          borderWidth: 1
        }]
      },
      options: {
        scales: {
```

```
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>
</body>
</html>
```

8.1.6 How This Works

- The `<canvas>` element provides a drawing surface for Chart.js.
- `new Chart(ctx, config)` initializes the chart.
- The `config` object specifies:
  - The `type` of chart,
  - The `data` including labels and datasets,
  - `options` to customize scales, legends, tooltips, and more.

8.1.7 Summary

- Chart.js offers quick setup for common charts with minimal code.
- Include it via CDN or npm depending on your workflow.
- Initialize charts by selecting a canvas and calling `new Chart()`.
- The configuration object defines the chart type, data, and options.
- Great for fast prototyping or adding standard charts to any web page.

8.2 Customizing Appearance

One of Chart.js's strengths is its rich configuration system that lets you easily customize the look and feel of your charts. From colors and borders to tooltips, gridlines, legends, and titles, you can tailor charts to match your design needs without complex code.

In this section, we'll cover:

- Customizing dataset colors and borders,
- Configuring tooltips,
- Styling gridlines and axes,
- Enabling and styling legends and titles,

- Adjusting fonts and other visual options.

8.2.1 Dataset Styling: Colors, Borders, and More

Each dataset can be styled with properties such as:

- `backgroundColor`: fill color of bars, areas, or points,
- `borderColor`: outline color,
- `borderWidth`: thickness of borders,
- `hoverBackgroundColor` and `hoverBorderColor`: colors on hover.

8.2.2 Example: Colored Bars with Borders

```
const data = {
  labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
  datasets: [{
    label: 'Sales',
    data: [12, 19, 7, 15, 10],
    backgroundColor: [
      'rgba(255, 99, 132, 0.7)',
      'rgba(54, 162, 235, 0.7)',
      'rgba(255, 206, 86, 0.7)',
      'rgba(75, 192, 192, 0.7)',
      'rgba(153, 102, 255, 0.7)'
    ],
    borderColor: 'rgba(0,0,0,0.8)',
    borderWidth: 2
  }]
};
```

Here, each bar has a different fill color with a consistent black border.

8.2.3 Tooltip Customization

Tooltips display contextual information when hovering over data points. You can configure:

- Display format,
- Colors and fonts,
- Callbacks to customize content.

8.2.4 Example: Custom Tooltip Options

```
const options = {
  plugins: {
    tooltip: {
      enabled: true,
      backgroundColor: 'rgba(0, 0, 0, 0.7)',
      titleFont: { size: 16, weight: 'bold' },
      bodyFont: { size: 14 },
      callbacks: {
        label: context => `Value: ${context.parsed.y}`
      }
    }
  }
};
```

This creates a dark tooltip with custom font sizes and a label prefix.

8.2.5 Gridlines and Axes Styling

Control gridlines and axis appearance under the `scales` option:

- `grid.color` sets gridline color,
- `grid.lineWidth` controls thickness,
- Axis ticks can be styled with fonts, colors, and rotation.

8.2.6 Example: Blue Gridlines and Custom Ticks

```
const options = {
  scales: {
    x: {
      grid: {
        color: 'rgba(0, 0, 255, 0.1)',
        lineWidth: 1
      },
      ticks: {
        color: 'navy',
        font: { size: 14, weight: 'bold' },
        maxRotation: 45,
        minRotation: 45
      }
    },
    y: {
      grid: {
        color: 'rgba(0, 0, 255, 0.1)'
      },
      ticks: {
        color: 'navy',
        font: { size: 14 }
```

```
      }
    }
  }
};
```

8.2.7 Legends and Titles

Enable and style legends and chart titles under the `plugins` section:

```
const options = {
  plugins: {
    legend: {
      display: true,
      position: 'top',
      labels: {
        color: 'darkred',
        font: { size: 14, family: 'Arial' }
      }
    },
    title: {
      display: true,
      text: 'Monthly Sales Data',
      color: 'darkblue',
      font: { size: 18, weight: 'bold' }
    }
  }
};
```

8.2.8 Putting It All Together

Here's a full example combining these customizations:

```
const ctx = document.getElementById('myChart').getContext('2d');

const config = {
  type: 'bar',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    datasets: [{
      label: 'Sales',
      data: [12, 19, 7, 15, 10],
      backgroundColor: [
        'rgba(255, 99, 132, 0.7)',
        'rgba(54, 162, 235, 0.7)',
        'rgba(255, 206, 86, 0.7)',
        'rgba(75, 192, 192, 0.7)',
        'rgba(153, 102, 255, 0.7)'
      ],
      borderColor: 'rgba(0,0,0,0.8)',
      borderWidth: 2
```

```
    }]
  },
  options: {
    plugins: {
      tooltip: {
        backgroundColor: 'rgba(0,0,0,0.7)',
        titleFont: { size: 16, weight: 'bold' },
        bodyFont: { size: 14 },
      },
      legend: {
        display: true,
        position: 'top',
        labels: {
          color: 'darkred',
          font: { size: 14, family: 'Arial' }
        }
      },
      title: {
        display: true,
        text: 'Monthly Sales Data',
        color: 'darkblue',
        font: { size: 18, weight: 'bold' }
      }
    },
    scales: {
      x: {
        grid: {
          color: 'rgba(0,0,255,0.1)',
          lineWidth: 1
        },
        ticks: {
          color: 'navy',
          font: { size: 14, weight: 'bold' },
          maxRotation: 45,
          minRotation: 45
        }
      },
      y: {
        grid: {
          color: 'rgba(0,0,255,0.1)'
        },
        ticks: {
          color: 'navy',
          font: { size: 14 }
        }
      }
    }
  }
};

const myChart = new Chart(ctx, config);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <title>Chart.js Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 30px;
      background-color: #f8f9fa;
    }
    #myChart {
      max-width: 600px;
      margin: auto;
    }
  </style>
</head>
<body>
  <h2 style="text-align:center;">Monthly Sales Dashboard</h2>
  <canvas id="myChart" width="600" height="400"></canvas>

  <script>
    const ctx = document.getElementById('myChart').getContext('2d');

    const config = {
      type: 'bar',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        datasets: [{
          label: 'Sales',
          data: [12, 19, 7, 15, 10],
          backgroundColor: [
            'rgba(255, 99, 132, 0.7)',
            'rgba(54, 162, 235, 0.7)',
            'rgba(255, 206, 86, 0.7)',
            'rgba(75, 192, 192, 0.7)',
            'rgba(153, 102, 255, 0.7)'
          ],
          borderColor: 'rgba(0,0,0,0.8)',
          borderWidth: 2
        }]
      },
      options: {
        plugins: {
          tooltip: {
            backgroundColor: 'rgba(0,0,0,0.7)',
            titleFont: { size: 16, weight: 'bold' },
            bodyFont: { size: 14 },
          },
          legend: {
            display: true,
            position: 'top',
            labels: {
              color: 'darkred',
              font: { size: 14, family: 'Arial' }
            }
          },
          title: {
            display: true,
            text: 'Monthly Sales Data',
```

```
            color: 'darkblue',
            font: { size: 18, weight: 'bold' }
          }
        },
        scales: {
          x: {
            grid: {
              color: 'rgba(0,0,255,0.1)',
              lineWidth: 1
            },
            ticks: {
              color: 'navy',
              font: { size: 14, weight: 'bold' },
              maxRotation: 45,
              minRotation: 45
            }
          },
          y: {
            grid: {
              color: 'rgba(0,0,255,0.1)'
            },
            ticks: {
              color: 'navy',
              font: { size: 14 }
            }
          }
        }
      }
    };

    const myChart = new Chart(ctx, config);
  </script>
</body>
</html>
```

8.2.9 Summary

- Customize dataset colors, borders, and hover styles in the dataset object.
- Style tooltips via `plugins.tooltip` options, including fonts and callbacks.
- Control gridlines and axis ticks under `scales` for better readability.
- Enable and style legends and titles with `plugins.legend` and `plugins.title`.
- Chart.js configuration is modular and declarative, making customization straightforward.

8.3 Responsive Design

In today's multi-device world, charts need to look great and remain readable on all screen sizes—from large desktops to small mobile phones. Chart.js makes building responsive charts simple by automatically resizing charts to fit their container while preserving aspect ratios and layout integrity.

This section covers:

- How Chart.js handles responsiveness,
- Configuring the `responsive` and `maintainAspectRatio` options,
- Best practices for responsive chart containers using CSS,
- Examples with CSS flexbox layouts.

8.3.1 Chart.js Responsive Defaults

By default, Chart.js charts are **responsive**:

```
options: {
  responsive: true,             // Chart resizes with container
  maintainAspectRatio: true     // Keeps original aspect ratio
}
```

- **`responsive: true`** tells Chart.js to automatically resize the chart when its container size changes.
- **`maintainAspectRatio: true`** ensures the chart maintains its original width-to-height ratio during resizing.

If you set `responsive` to `false`, the chart size is fixed, and does not adapt when the window or container resizes.

8.3.2 Customizing Aspect Ratio

You can customize the default aspect ratio using the `aspectRatio` option:

```
options: {
  aspectRatio: 2 // Width will be twice the height
}
```

If `maintainAspectRatio` is set to `false`, the chart will fill the container's width and height regardless of aspect ratio.

8.3.3 Using CSS to Control Chart Size

The size of a Chart.js chart depends on its container element. You can control responsiveness with CSS:

8.3.4 Example: Responsive Chart Container

```
<div class="chart-container">
  <canvas id="myChart"></canvas>
</div>

<style>
.chart-container {
  width: 100%;          /* Container fills the width of its parent */
  max-width: 600px;     /* Optional max width */
  height: 400px;        /* Fixed height */
  margin: auto;         /* Center horizontally */
}
</style>
```

In this example, the chart scales to fit the container's width but keeps a fixed height of 400px.

8.3.5 Example: Flexbox Layout

Flexbox is great for layouts that adapt fluidly to screen sizes.

```
<div class="dashboard" style="display: flex; gap: 20px;">
  <div style="flex: 1; min-width: 300px;">
    <canvas id="chart1"></canvas>
  </div>
  <div style="flex: 1; min-width: 300px;">
    <canvas id="chart2"></canvas>
  </div>
</div>
```

Here, two charts share the horizontal space equally and shrink responsively down to a minimum width of 300px.

8.3.6 Example: Responsive Chart Configuration

```
const ctx = document.getElementById('myChart').getContext('2d');

const config = {
  type: 'line',
  data: {
```

```
    labels: ['Jan', 'Feb', 'Mar', 'Apr'],
    datasets: [{
      label: 'Revenue',
      data: [120, 190, 300, 250],
      borderColor: 'blue',
      fill: false,
    }]
  },
  options: {
    responsive: true,
    maintainAspectRatio: false,  // Fill the container's dimensions
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
};

const myChart = new Chart(ctx, config);
```

By setting `maintainAspectRatio` to `false` and controlling the container's CSS height, the chart adapts fluidly to container size changes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Responsive Chart.js Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
      background-color: #f0f2f5;
    }

    .chart-container {
      width: 100%;
      max-width: 600px;
      height: 400px;
      margin: 30px auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
      padding: 20px;
    }

    .dashboard {
      display: flex;
      gap: 20px;
      justify-content: center;
      flex-wrap: wrap;
    }
```

```
    .dashboard > div {
      flex: 1;
      min-width: 300px;
    }

    canvas {
      width: 100% !important;
      height: 100% !important;
    }
  </style>
</head>
<body>
  <h2 style="text-align:center;">Responsive Chart.js Dashboard</h2>

  <!-- Single Line Chart in Responsive Container -->
  <div class="chart-container">
    <canvas id="myChart"></canvas>
  </div>

  <!-- Two Charts Side by Side Using Flexbox -->
  <div class="dashboard">
    <div><canvas id="chart1"></canvas></div>
    <div><canvas id="chart2"></canvas></div>
  </div>

  <script>
    const configLine = {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr'],
        datasets: [{
          label: 'Revenue',
          data: [120, 190, 300, 250],
          borderColor: 'blue',
          backgroundColor: 'rgba(0,0,255,0.1)',
          fill: true,
        }]
      },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const configBar = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
          label: 'Profit',
          data: [100, 150, 130, 170],
          backgroundColor: 'rgba(75, 192, 192, 0.6)',
          borderColor: 'rgba(75, 192, 192, 1)',
```

```
          borderWidth: 1
        }]
      },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    new Chart(document.getElementById('myChart').getContext('2d'), configLine);
    new Chart(document.getElementById('chart1').getContext('2d'), configLine);
    new Chart(document.getElementById('chart2').getContext('2d'), configBar);
  </script>
</body>
</html>
```

8.3.7 Tips for Responsive Charts

- Use container elements with relative or fixed height; otherwise, the chart may collapse.
- Consider `maintainAspectRatio: false` if you want the chart to fill the container exactly.
- Use CSS media queries to adjust container size on different screen widths.
- Combine Chart.js responsiveness with CSS flexbox or grid for flexible dashboard layouts.

8.3.8 Summary

- Chart.js supports responsive charts that resize automatically with their containers.
- `responsive: true` enables resizing; `maintainAspectRatio` controls whether aspect ratio is preserved.
- Use CSS to define container width and height; flexbox and grid are powerful for layout.
- Adjust aspect ratio and container sizing for best user experience on different devices.

Chapter 9.

ECharts

1. Rich Interactive Dashboards
2. Handling Large Datasets
3. Themes and Advanced Options

9 ECharts

9.1 Rich Interactive Dashboards

ECharts is a powerful, open-source JavaScript visualization library developed by Apache that excels at creating rich, interactive dashboards with surprisingly little code. It offers built-in support for complex interactions such as linked tooltips, brushing (selection), zooming, and multi-panel layouts, making it ideal for building sophisticated analytics interfaces.

In this section, we'll explore how ECharts enables these interactive features and walk through an example that links a bar chart and a pie chart — a classic dashboard interaction pattern.

9.1.1 Why Use ECharts for Dashboards?

- **Minimal code for complex visuals:** ECharts abstracts many interactions you would otherwise implement manually.
- **Built-in linking and brushing:** Easily synchronize tooltips or selections across multiple charts.
- **Flexible layouts:** Compose multiple charts side-by-side or stacked with smooth integration.
- **Performance:** Optimized for large datasets and dynamic updates.

9.1.2 Linking Bar and Pie Charts: An Example

Imagine a sales dashboard where:

- The **bar chart** shows sales by product category.
- The **pie chart** breaks down the selected category's sales by region.
- Hovering or clicking a bar filters the pie chart accordingly.

9.1.3 Step 1: Setup HTML and Include ECharts

```
<div id="barChart" style="width: 50%; height: 400px; display: inline-block;"></div>
<div id="pieChart" style="width: 45%; height: 400px; display: inline-block;"></div>

<script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
```

9.1.4 Step 2: Initialize the Charts

```
const barChart = echarts.init(document.getElementById('barChart'));
const pieChart = echarts.init(document.getElementById('pieChart'));
```

9.1.5 Step 3: Define Data

```
const categories = ['Electronics', 'Clothing', 'Books', 'Home'];
const salesByCategory = [120, 90, 150, 80];

const salesByRegion = {
  Electronics: [
    { name: 'North', value: 40 },
    { name: 'South', value: 30 },
    { name: 'East', value: 25 },
    { name: 'West', value: 25 }
  ],
  Clothing: [
    { name: 'North', value: 20 },
    { name: 'South', value: 25 },
    { name: 'East', value: 30 },
    { name: 'West', value: 15 }
  ],
  Books: [
    { name: 'North', value: 50 },
    { name: 'South', value: 30 },
    { name: 'East', value: 40 },
    { name: 'West', value: 30 }
  ],
  Home: [
    { name: 'North', value: 15 },
    { name: 'South', value: 25 },
    { name: 'East', value: 20 },
    { name: 'West', value: 20 }
  ]
};
```

9.1.6 Step 4: Configure the Bar Chart

```
const barOption = {
  title: { text: 'Sales by Category' },
  tooltip: {},
  xAxis: {
    type: 'category',
    data: categories,
    axisTick: { alignWithLabel: true }
  },
  yAxis: { type: 'value' },
  series: [{
```

```
    name: 'Sales',
    type: 'bar',
    data: salesByCategory,
    emphasis: {
      focus: 'series'
    }
  }]
};
barChart.setOption(barOption);
```

9.1.7 Step 5: Configure the Pie Chart

```
const pieOption = {
  title: { text: 'Sales by Region', left: 'center' },
  tooltip: {
    trigger: 'item'
  },
  legend: {
    orient: 'vertical',
    left: 'left'
  },
  series: [{
    name: 'Region Sales',
    type: 'pie',
    radius: '60%',
    data: salesByRegion[categories[0]],
    emphasis: {
      itemStyle: {
        shadowBlur: 10,
        shadowOffsetX: 0,
        shadowColor: 'rgba(0, 0, 0, 0.5)'
      }
    }
  }]
};
pieChart.setOption(pieOption);
```

9.1.8 Step 6: Link Interactions

When hovering or clicking on a bar, update the pie chart data accordingly:

```
barChart.on('click', function (params) {
  const category = params.name;
  pieChart.setOption({
    series: [{
      data: salesByRegion[category]
    }],
    title: { text: `Sales by Region for ${category}` }
  });
```

```
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Interactive ECharts Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background-color: #f4f4f4;
      padding: 20px;
      text-align: center;
    }
    .charts {
      display: flex;
      justify-content: space-around;
      flex-wrap: wrap;
    }
    #barChart, #pieChart {
      height: 400px;
      border-radius: 8px;
      background: white;
      box-shadow: 0 2px 6px rgba(0,0,0,0.1);
    }
    #barChart {
      width: 50%;
      min-width: 300px;
    }
    #pieChart {
      width: 45%;
      min-width: 300px;
    }
  </style>
</head>
<body>
  <h2>Sales Dashboard</h2>
  <p>Click a bar to view detailed regional data</p>

  <div class="charts">
    <div id="barChart"></div>
    <div id="pieChart"></div>
  </div>

  <script>
    const categories = ['Electronics', 'Clothing', 'Books', 'Home'];
    const salesByCategory = [120, 90, 150, 80];

    const salesByRegion = {
      Electronics: [
        { name: 'North', value: 40 },
        { name: 'South', value: 30 },
        { name: 'East', value: 25 },
        { name: 'West', value: 25 }
```

```
    ],
    Clothing: [
      { name: 'North', value: 20 },
      { name: 'South', value: 25 },
      { name: 'East', value: 30 },
      { name: 'West', value: 15 }
    ],
    Books: [
      { name: 'North', value: 50 },
      { name: 'South', value: 30 },
      { name: 'East', value: 40 },
      { name: 'West', value: 30 }
    ],
    Home: [
      { name: 'North', value: 15 },
      { name: 'South', value: 25 },
      { name: 'East', value: 20 },
      { name: 'West', value: 20 }
    ]
  };

  const barChart = echarts.init(document.getElementById('barChart'));
  const pieChart = echarts.init(document.getElementById('pieChart'));

  const barOption = {
    title: { text: 'Sales by Category' },
    tooltip: {},
    xAxis: {
      type: 'category',
      data: categories,
      axisTick: { alignWithLabel: true }
    },
    yAxis: { type: 'value' },
    series: [{
      name: 'Sales',
      type: 'bar',
      data: salesByCategory,
      emphasis: { focus: 'series' }
    }]
  };

  const pieOption = {
    title: { text: 'Sales by Region', left: 'center' },
    tooltip: { trigger: 'item' },
    legend: { orient: 'vertical', left: 'left' },
    series: [{
      name: 'Region Sales',
      type: 'pie',
      radius: '60%',
      data: salesByRegion[categories[0]],
      emphasis: {
        itemStyle: {
          shadowBlur: 10,
          shadowOffsetX: 0,
          shadowColor: 'rgba(0, 0, 0, 0.5)'
        }
      }
    }]
```

```
    };

    barChart.setOption(barOption);
    pieChart.setOption(pieOption);

    barChart.on('click', function (params) {
      const category = params.name;
      pieChart.setOption({
        series: [{ data: salesByRegion[category] }],
        title: { text: `Sales by Region for ${category}` }
      });
    });
  </script>
</body>
</html>
```

9.1.9 What Happens?

- The bar chart shows total sales per category.
- Clicking a bar updates the pie chart to show the regional breakdown for that category.
- Both charts have tooltips for detailed info.
- This creates a dashboard feel where charts respond dynamically to user input.

9.1.10 Additional Interactive Features in ECharts

- **Tooltip linking:** Tooltips on multiple charts can show synchronized data.
- **Brushing:** Select ranges or points on one chart to filter or highlight others.
- **Data zoom and pan:** Zoom or pan large datasets easily.
- **Multi-panel layouts:** Organize many charts in responsive grid layouts.

9.1.11 Summary

- ECharts enables complex dashboards with minimal boilerplate code.
- Built-in interaction patterns like linked charts simplify dashboard logic.
- Powerful features like brushing, zooming, and flexible layouts are ready to use.
- The bar + pie chart example demonstrates how to create linked visualizations that respond to user events, enhancing data exploration.

9.2 Handling Large Datasets

Handling large datasets efficiently is critical in modern data visualization. ECharts is designed with powerful performance optimizations that allow you to render and interact with tens of thousands of data points smoothly. This section explores how ECharts achieves this and shows practical techniques like progressive rendering, data sampling, and zooming to maintain responsiveness even with large datasets.

9.2.1 Performance Optimization Techniques in ECharts

Progressive Rendering

For very large datasets, ECharts supports *progressive rendering*, which gradually draws data points in chunks instead of all at once. This approach prevents browser freezing and ensures smooth initial load.

You enable it by setting the `progressive` and `progressiveThreshold` options on a series:

```
series: [{
  type: 'scatter',
  progressive: 5000,          // Number of points rendered per frame
  progressiveThreshold: 10000 // Threshold to start progressive rendering
}]
```

9.2.2 Data Sampling

When plotting huge datasets, sometimes showing every point is unnecessary or impractical. Data sampling reduces the number of points by selecting representative samples, improving rendering speed while preserving visual patterns.

You can implement sampling server-side or preprocess data client-side before passing it to ECharts.

9.2.3 Zooming and Panning with `dataZoom`

ECharts' built-in `dataZoom` component lets users zoom and pan to explore subsets of data without reloading. This is essential when working with large data, as it focuses rendering on the visible range.

```
dataZoom: [
  {
    type: 'inside',  // Enables zooming and panning inside the chart
    xAxisIndex: 0
```

```
  },
  {
    type: 'slider',  // A visible slider control
    xAxisIndex: 0
  }
]
```

9.2.4 Example: Scatter Plot with 10,000 Points and Zoom

Step 1: Generate Large Dataset

```
const data = [];

for (let i = 0; i < 12000; i++) {
  data.push([
    Math.random() * 1000,  // x value
    Math.random() * 1000   // y value
  ]);
}
```

Step 2: Initialize ECharts and Configure Scatter Plot

```
const chart = echarts.init(document.getElementById('scatterPlot'));

const option = {
  title: { text: 'Large Scatter Plot with Zoom' },
  tooltip: { trigger: 'item' },
  xAxis: { type: 'value', min: 0, max: 1000 },
  yAxis: { type: 'value', min: 0, max: 1000 },
  dataZoom: [
    { type: 'inside', xAxisIndex: 0, yAxisIndex: 0 },
    { type: 'slider', xAxisIndex: 0, yAxisIndex: 0 }
  ],
  series: [{
    type: 'scatter',
    data: data,
    symbolSize: 5,
    progressive: 3000,
    progressiveThreshold: 5000
  }]
};

chart.setOption(option);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Large ECharts Scatter Plot</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
```

```
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background-color: #f0f2f5;
      padding: 20px;
      text-align: center;
    }
    #scatterPlot {
      width: 90%;
      max-width: 1200px;
      height: 600px;
      margin: auto;
      border-radius: 8px;
      background: white;
      box-shadow: 0 2px 6px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Scatter Plot: 12,000 Points with Progressive Rendering</h2>
  <p>Zoom and pan to explore high-density data without lag</p>

  <div id="scatterPlot"></div>

  <script>
    // Step 1: Generate Large Dataset
    const data = [];
    for (let i = 0; i < 12000; i++) {
      data.push([
        Math.random() * 1000,  // x value
        Math.random() * 1000   // y value
      ]);
    }

    // Step 2: Initialize and Configure Chart
    const chart = echarts.init(document.getElementById('scatterPlot'));

    const option = {
      title: { text: 'Large Scatter Plot with Zoom', left: 'center' },
      tooltip: { trigger: 'item' },
      xAxis: { type: 'value', min: 0, max: 1000 },
      yAxis: { type: 'value', min: 0, max: 1000 },
      dataZoom: [
        { type: 'inside', xAxisIndex: 0, yAxisIndex: 0 },
        { type: 'slider', xAxisIndex: 0, yAxisIndex: 0 }
      ],
      series: [{
        type: 'scatter',
        data: data,
        symbolSize: 5,
        progressive: 3000,
        progressiveThreshold: 5000
      }]
    };

    chart.setOption(option);
  </script>
</body>
```

```
</html>
```

9.2.5 What Makes This Efficient?

- **Progressive rendering** allows the browser to draw 3,000 points per animation frame, avoiding freezes.
- `dataZoom` enables users to zoom into any region, rendering fewer points at once.
- The `symbolSize` is modest, keeping visual clutter manageable.
- Tooltips show details for hovered points without lag.

9.2.6 Tips for Handling Large Data with ECharts

- Use **progressive rendering** for datasets over several thousand points.
- Apply **data sampling** where appropriate to reduce points while preserving trends.
- Employ **dataZoom** to focus user attention and improve performance.
- Optimize data formats (arrays instead of objects) for faster parsing.
- Keep visual elements lightweight (small symbols, no complex animations).

9.2.7 Summary

- ECharts is optimized to handle large datasets smoothly using progressive rendering and sampling.
- `dataZoom` lets users interactively explore large data without overwhelming the browser.
- Combining these techniques enables real-time visualization of tens of thousands of points.
- The scatter plot example demonstrates how to apply these features for responsive, performant charts.

9.3 Themes and Advanced Options

ECharts provides powerful theming and configuration capabilities that let you create visually stunning charts tailored to your application's style and complexity requirements. This section explores how to apply prebuilt themes like dark or vintage, and dives into advanced chart configurations such as stacked area charts and dual Y-axes.

9.3.1 Applying Themes in ECharts

Themes allow you to quickly change the look of your charts globally, affecting colors, fonts, and styles without rewriting your chart options.

9.3.2 Using Built-in Themes

ECharts ships with a few built-in themes such as **dark** and **vintage**. You can include them via CDN or npm and apply when initializing your chart.

9.3.3 Step 1: Include Theme Script

From CDN, include the theme JavaScript file after the core ECharts script:

```
<script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/echarts/theme/dark.js"></script>
```

9.3.4 Step 2: Initialize Chart with Theme

Pass the theme name as the second argument to `echarts.init`:

```
const chart = echarts.init(document.getElementById('chartContainer'), 'dark');
```

This applies the dark theme globally to the chart, changing default colors and background accordingly.

9.3.5 Using Custom Themes

You can also create custom themes by defining a JSON object specifying colors, fonts, and styles, then register it via:

```
echarts.registerTheme('myCustomTheme', {
  color: ['#c23531', '#2f4554', '#61a0a8', '#d48265'],
  backgroundColor: '#f0f0f0',
  textStyle: { fontFamily: 'Arial' },
  // Other style options...
});
```

Then initialize your chart with `'myCustomTheme'`.

9.3.6 Advanced Chart Configurations

ECharts' flexible option structure supports complex visualizations like stacked area charts and charts with dual Y-axes.

9.3.7 Example 1: Stacked Area Chart

Stacked area charts visualize cumulative values over a category (often time), showing part-to-whole relationships and trends.

```
const option = {
  title: { text: 'Stacked Area Chart' },
  tooltip: { trigger: 'axis' },
  legend: { data: ['Email', 'Ads', 'Direct'] },
  xAxis: { type: 'category', data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'] },
  yAxis: { type: 'value' },
  series: [
    {
      name: 'Email',
      type: 'line',
      stack: 'total',
      areaStyle: {},
      data: [120, 132, 101, 134, 90, 230, 210]
    },
    {
      name: 'Ads',
      type: 'line',
      stack: 'total',
      areaStyle: {},
      data: [220, 182, 191, 234, 290, 330, 310]
    },
    {
      name: 'Direct',
      type: 'line',
      stack: 'total',
      areaStyle: {},
      data: [150, 232, 201, 154, 190, 330, 410]
    }
  ]
};
chart.setOption(option);
```

- The `stack: 'total'` option stacks all series vertically.
- `areaStyle: {}` enables the filled area under the line.
- Tooltips show combined values per point for easy comparison.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
```

```
  <title>ECharts Stacked Area Chart - Dark Theme</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/echarts/theme/dark.js"></script>
  <style>
    body {
      margin: 0;
      background-color: #2c343c;
      font-family: 'Segoe UI', sans-serif;
      color: #ddd;
      text-align: center;
      padding: 20px;
    }
    #chartContainer {
      width: 95%;
      max-width: 1000px;
      height: 500px;
      margin: 0 auto;
      border-radius: 8px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.3);
    }
  </style>
</head>
<body>
  <h2>Stacked Area Chart with Dark Theme</h2>
  <div id="chartContainer"></div>

  <script>
    const chart = echarts.init(document.getElementById('chartContainer'), 'dark');

    const option = {
      title: { text: 'Stacked Area Chart' },
      tooltip: { trigger: 'axis' },
      legend: { data: ['Email', 'Ads', 'Direct'] },
      xAxis: {
        type: 'category',
        boundaryGap: false,
        data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
      },
      yAxis: { type: 'value' },
      series: [
        {
          name: 'Email',
          type: 'line',
          stack: 'total',
          areaStyle: {},
          data: [120, 132, 101, 134, 90, 230, 210]
        },
        {
          name: 'Ads',
          type: 'line',
          stack: 'total',
          areaStyle: {},
          data: [220, 182, 191, 234, 290, 330, 310]
        },
        {
          name: 'Direct',
          type: 'line',
          stack: 'total',
```

```
          areaStyle: {},
          data: [150, 232, 201, 154, 190, 330, 410]
        }
      ]
    };

    chart.setOption(option);
  </script>
</body>
</html>
```

9.3.8 Example 2: Dual Y-Axis Chart

When two datasets use different scales, dual Y-axes allow displaying both clearly.

```
const option = {
  title: { text: 'Dual Y-Axis Chart' },
  tooltip: { trigger: 'axis' },
  legend: { data: ['Temperature', 'Rainfall'] },
  xAxis: { type: 'category', data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'] },
  yAxis: [
    {
      type: 'value',
      name: 'Temperature (°C)',
      position: 'left',
      axisLine: { lineStyle: { color: '#ff5722' } },
      axisLabel: { formatter: '{value} °C' }
    },
    {
      type: 'value',
      name: 'Rainfall (mm)',
      position: 'right',
      axisLine: { lineStyle: { color: '#2196f3' } },
      axisLabel: { formatter: '{value} mm' }
    }
  ],
  series: [
    {
      name: 'Temperature',
      type: 'line',
      yAxisIndex: 0,
      data: [2.0, 4.9, 7.0, 23.2, 25.6, 76.7],
      itemStyle: { color: '#ff5722' }
    },
    {
      name: 'Rainfall',
      type: 'bar',
      yAxisIndex: 1,
      data: [2.6, 5.9, 9.0, 26.4, 28.7, 70.7],
      itemStyle: { color: '#2196f3' }
    }
  ]
};
chart.setOption(option);
```

- Two Y-axes are defined in the `yAxis` array with different labels and colors.
- `yAxisIndex` in each series associates it with the left or right axis.
- Mixing `line` and `bar` chart types is straightforward.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Dual Y-Axis Chart with ECharts</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
  <style>
    body {
      margin: 0;
      font-family: Arial, sans-serif;
      background-color: #f5f5f5;
      padding: 20px;
      text-align: center;
    }
    #chartContainer {
      width: 95%;
      max-width: 1000px;
      height: 500px;
      margin: auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
    }
  </style>
</head>
<body>
  <h2>Dual Y-Axis Chart</h2>
  <p>Visualizing Temperature vs. Rainfall on Separate Y-Axes</p>
  <div id="chartContainer"></div>

  <script>
    const chart = echarts.init(document.getElementById('chartContainer'));

    const option = {
      title: { text: 'Dual Y-Axis Chart' },
      tooltip: { trigger: 'axis' },
      legend: { data: ['Temperature', 'Rainfall'] },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
      },
      yAxis: [
        {
          type: 'value',
          name: 'Temperature (°C)',
          position: 'left',
          axisLine: { lineStyle: { color: '#ff5722' } },
          axisLabel: { formatter: '{value} °C' }
        },
        {
          type: 'value',
```

```
          name: 'Rainfall (mm)',
          position: 'right',
          axisLine: { lineStyle: { color: '#2196f3' } },
          axisLabel: { formatter: '{value} mm' }
        }
      ],
      series: [
        {
          name: 'Temperature',
          type: 'line',
          yAxisIndex: 0,
          data: [2.0, 4.9, 7.0, 23.2, 25.6, 76.7],
          itemStyle: { color: '#ff5722' }
        },
        {
          name: 'Rainfall',
          type: 'bar',
          yAxisIndex: 1,
          data: [2.6, 5.9, 9.0, 26.4, 28.7, 70.7],
          itemStyle: { color: '#2196f3' }
        }
      ]
    };

    chart.setOption(option);
  </script>
</body>
</html>
```

9.3.9 Summary

- ECharts themes allow consistent, easy customization of chart appearance globally.
- You can load built-in themes like `dark` or create your own for full control.
- Advanced configurations support complex charts such as stacked areas and dual Y-axes.
- The flexible option structure makes combining chart types and custom layouts seamless.

Chapter 10.

Highcharts

1. Rich Interactive Dashboards
2. Handling Large Datasets
3. Themes and Advanced Options

10 Highcharts

10.1 Rich Interactive Dashboards

Highcharts is a widely used commercial JavaScript charting library known for its rich features and extensive interactivity out of the box. It offers smooth animations, responsive design, and a variety of built-in tools that enable developers to create highly interactive dashboards with minimal effort.

This section introduces how Highcharts supports linked charts, drilldowns, and live data updates, empowering you to build engaging and insightful dashboards. We'll also walk through a practical example where clicking on a chart segment drills down to reveal detailed subcategory charts.

10.1.1 Why Choose Highcharts for Dashboards?

- **Robust interactivity:** Drilldown, zooming, panning, tooltips, and linked charts are built-in.
- **Ease of use:** Intuitive API and detailed documentation speed up development.
- **Cross-device compatibility:** Works well on desktops and mobile devices.
- **Live data:** Supports real-time updates for dynamic dashboards.

10.1.2 Drilldown Example: Exploring Categories and Subcategories

Imagine a sales dashboard where:

- The initial **column chart** displays sales by main product categories.
- Clicking on a category drills down into a detailed chart showing sales by subcategories.
- Users can easily navigate back to the main view.

10.1.3 Step 1: Include Highcharts Library

```
<script src="https://code.highcharts.com/highcharts.js"></script>
<script src="https://code.highcharts.com/modules/drilldown.js"></script>
```

10.1.4 Step 2: Create a Container for the Chart

```
<div id="container" style="width: 100%; height: 400px;"></div>
```

10.1.5 Step 3: Initialize the Chart with Drilldown Data

```
Highcharts.chart('container', {
  chart: { type: 'column' },
  title: { text: 'Sales by Category' },
  xAxis: { type: 'category' },
  yAxis: { title: { text: 'Sales (in USD)' } },
  legend: { enabled: false },
  plotOptions: {
    series: {
      borderWidth: 0,
      dataLabels: { enabled: true }
    }
  },
  series: [{
    name: 'Categories',
    colorByPoint: true,
    data: [
      { name: 'Electronics', y: 120, drilldown: 'electronics' },
      { name: 'Clothing', y: 90, drilldown: 'clothing' },
      { name: 'Books', y: 150, drilldown: 'books' }
    ]
  }],
  drilldown: {
    series: [
      {
        id: 'electronics',
        name: 'Electronics Subcategories',
        data: [
          ['Phones', 50],
          ['Laptops', 40],
          ['Accessories', 30]
        ]
      },
      {
        id: 'clothing',
        name: 'Clothing Subcategories',
        data: [
          ['Men', 40],
          ['Women', 30],
          ['Kids', 20]
        ]
      },
      {
        id: 'books',
        name: 'Books Subcategories',
        data: [
          ['Fiction', 60],
          ['Non-fiction', 50],
```

```
          ['Children', 40]
        ]
      }
    ]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highcharts Drilldown Example</title>
  <script src="https://code.highcharts.com/highcharts.js"></script>
  <script src="https://code.highcharts.com/modules/drilldown.js"></script>
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background-color: #f4f4f4;
      padding: 20px;
      text-align: center;
    }
    #container {
      width: 100%;
      max-width: 800px;
      height: 400px;
      margin: auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 6px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Sales by Category with Drilldown</h2>
  <p>Click on a column to explore subcategories</p>

  <div id="container"></div>

  <script>
    Highcharts.chart('container', {
      chart: { type: 'column' },
      title: { text: 'Sales by Category' },
      xAxis: { type: 'category' },
      yAxis: { title: { text: 'Sales (in USD)' } },
      legend: { enabled: false },
      plotOptions: {
        series: {
          borderWidth: 0,
          dataLabels: { enabled: true, format: '{point.y}' }
        }
      },
      series: [{
        name: 'Categories',
        colorByPoint: true,
        data: [
```

```
          { name: 'Electronics', y: 120, drilldown: 'electronics' },
          { name: 'Clothing', y: 90, drilldown: 'clothing' },
          { name: 'Books', y: 150, drilldown: 'books' }
        ]
      }],
      drilldown: {
        series: [
          {
            id: 'electronics',
            name: 'Electronics Subcategories',
            data: [
              ['Phones', 50],
              ['Laptops', 40],
              ['Accessories', 30]
            ]
          },
          {
            id: 'clothing',
            name: 'Clothing Subcategories',
            data: [
              ['Men', 40],
              ['Women', 30],
              ['Kids', 20]
            ]
          },
          {
            id: 'books',
            name: 'Books Subcategories',
            data: [
              ['Fiction', 60],
              ['Non-fiction', 50],
              ['Children', 40]
            ]
          }
        ]
      }
    });
  </script>
</body>
</html>
```

10.1.6 What Happens?

- The main column chart shows sales totals for each category.
- Clicking on a category triggers a drilldown, replacing the chart data with subcategory sales.
- Highcharts provides smooth animations and a built-in “Back” button to return to the main view.

10.1.7 Additional Interactive Dashboard Features in Highcharts

- **Linked charts:** You can synchronize multiple charts by updating series data or highlighting corresponding points programmatically.
- **Live updates:** Real-time data streaming updates chart points dynamically with `.addPoint()` and `.setData()`.
- **Zoom and pan:** Built-in zooming and panning let users focus on specific data ranges.
- **Advanced tooltips:** Rich HTML tooltips can include images, links, or custom formatting.

10.1.8 Summary

- Highcharts offers a comprehensive suite of interactive features ideal for building rich dashboards.
- Drilldowns simplify data exploration by revealing detailed views on demand.
- Linking charts and live updates enhance the depth and dynamism of dashboards.
- The example demonstrated how to create a drilldown-enabled column chart with minimal setup.

10.2 Handling Large Datasets

When visualizing large datasets, performance and responsiveness are key. Highcharts includes several built-in features designed to optimize rendering and interactivity when working with thousands or even millions of points. This section covers essential techniques such as using the `turboThreshold` setting, lazy loading, and navigator components to create smooth, zoomable charts with large time series data.

10.2.1 Performance Optimization Techniques in Highcharts

`turboThreshold`

Highcharts' `turboThreshold` setting limits the number of data points rendered without additional parsing. When your dataset exceeds this threshold, Highcharts optimizes rendering to maintain speed, but disables some advanced features to avoid slowdowns.

By default, `turboThreshold` is set to 1000 points, but you can increase it to accommodate larger datasets:

```
plotOptions: {
  series: {
```

```
    turboThreshold: 5000  // Allow up to 5000 points before optimization kicks in
  }
}
```

If your dataset exceeds this number, Highcharts applies faster rendering algorithms, which improves performance without significantly sacrificing visual quality.

Lazy Loading and Progressive Rendering

While Highcharts does not natively support progressive rendering like some libraries, you can implement lazy loading by loading and rendering data in chunks or by reducing data density based on zoom level (e.g., data sampling on the server).

Navigator and Scrollbar for Time Series Data

Highcharts' **navigator** and **scrollbar** components enable users to zoom, pan, and explore subsets of large time series data efficiently:

- The **navigator** shows an overview of the full dataset with a draggable window.
- The **scrollbar** provides a quick way to navigate through data.

10.2.2 Example: Zoomable Time Series Line Chart with Thousands of Points

Step 1: Generate Large Time Series Data

```
const data = [];
const startTime = new Date('2023-01-01').getTime();
const pointCount = 10000;

for (let i = 0; i < pointCount; i++) {
  data.push([
    startTime + i * 3600 * 1000, // hourly intervals
    Math.round(Math.random() * 100)  // random value
  ]);
}
```

Step 2: Initialize Highcharts with Navigator and TurboThreshold

```
Highcharts.chart('container', {
  chart: { zoomType: 'x' },
  title: { text: 'Zoomable Time Series with Large Dataset' },
  xAxis: { type: 'datetime' },
  yAxis: { title: { text: 'Value' } },
  tooltip: { xDateFormat: '%Y-%m-%d %H:%M', shared: true },
  navigator: {
    enabled: true,
    adaptToUpdatedData: true
  },
  scrollbar: { enabled: true },
  plotOptions: {
    series: {
```

```
      turboThreshold: 20000  // Adjust for large datasets
    }
  },
  series: [{
    name: 'Random Data',
    data: data,
    type: 'line',
    marker: { enabled: false }
  }]
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highcharts Large Time Series Chart</title>
  <script src="https://code.highcharts.com/highcharts.js"></script>
  <script src="https://code.highcharts.com/modules/stock.js"></script>
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background-color: #f0f2f5;
      padding: 20px;
      text-align: center;
    }
    #container {
      width: 95%;
      max-width: 1000px;
      height: 600px;
      margin: auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Zoomable Time Series with Large Dataset</h2>
  <p>Scroll, zoom, and navigate through thousands of data points</p>

  <div id="container"></div>

  <script>
    const data = [];
    const startTime = new Date('2023-01-01').getTime();
    const pointCount = 10000;

    for (let i = 0; i < pointCount; i++) {
      data.push([
        startTime + i * 3600 * 1000, // hourly intervals
        Math.round(Math.random() * 100)
      ]);
    }

    Highcharts.stockChart('container', {
```

```
      chart: { zoomType: 'x' },
      title: { text: 'Zoomable Time Series with Large Dataset' },
      xAxis: { type: 'datetime' },
      yAxis: { title: { text: 'Value' } },
      tooltip: {
        xDateFormat: '%Y-%m-%d %H:%M',
        shared: true
      },
      navigator: {
        enabled: true,
        adaptToUpdatedData: true
      },
      scrollbar: { enabled: true },
      plotOptions: {
        series: {
          turboThreshold: 20000,
          marker: { enabled: false }
        }
      },
      series: [{
        name: 'Random Data',
        data: data,
        type: 'line'
      }]
    });
  </script>
</body>
</html>
```

10.2.3 What This Does:

- The **navigator** at the bottom lets users select and zoom into a subset of the 10,000 points.
- **Scrollbar** provides quick navigation through the timeline.
- Setting `turboThreshold` higher allows rendering this large dataset without errors.
- Disabling markers with `marker: { enabled: false }` helps maintain performance.
- Zooming on the X-axis (`zoomType: 'x'`) enables detailed exploration of data segments.

10.2.4 Additional Tips for Handling Large Datasets in Highcharts

- Use **data grouping** (available in Highstock) to aggregate data points at higher zoom levels.
- Consider server-side **data sampling** or **aggregation** to reduce client-side load.
- Avoid unnecessary chart redraws; update data using `.setData()` or `.addPoint()` efficiently.
- Turn off animations when rendering large datasets to improve speed.

10.2.5 Summary

- Highcharts offers multiple options to optimize large dataset visualization, including the `turboThreshold` and navigator.
- The navigator and scrollbar components enhance user experience by enabling smooth zooming and panning through extensive time series.
- Proper configuration of markers and thresholds ensures charts remain responsive and visually clear.
- The zoomable line chart example demonstrates these techniques in practice for handling thousands of points seamlessly.

10.3 Themes and Advanced Options

Highcharts offers extensive options to customize the appearance and functionality of your charts, enabling you to create visually consistent dashboards and highly specialized visualizations. This section covers how to apply built-in and custom themes, and explores advanced chart types like polar charts, gauge charts, and heatmaps. We'll also provide a configuration-rich example that demonstrates the flexibility of Highcharts' options API and plugin system.

10.3.1 Using Built-in Themes

Highcharts includes several built-in themes such as **dark-unica**, **grid-light**, and **sand-signika** that you can easily load to change the overall look and feel of your charts.

To apply a built-in theme:

1. Include the theme script after the core Highcharts script:

```
<script src="https://code.highcharts.com/themes/dark-unica.js"></script>
```

2. Initialize your chart as usual. The theme applies automatically and overrides default colors, fonts, and styles.

10.3.2 Creating and Applying Custom Themes

You can create your own theme by defining a configuration object with your preferred styles and then apply it globally via `Highcharts.setOptions()`:

```
Highcharts.setOptions({
  chart: {
```

```
    backgroundColor: '#f0f0f0',
    style: { fontFamily: 'Arial, sans-serif' }
  },
  title: { style: { color: '#333', fontSize: '18px' } },
  colors: ['#2b908f', '#90ee7e', '#f45b5b', '#7798BF'],
  tooltip: { backgroundColor: 'rgba(0,0,0,0.75)', style: { color: '#fff' } }
  // Add more global styling here...
});
```

This approach ensures consistent styling across all charts in your application.

10.3.3 Exploring Advanced Chart Types

Polar Charts

Polar charts display data in circular layouts, useful for wind rose charts, radar charts, or cyclic data.

Basic polar chart configuration:

```
Highcharts.chart('container', {
  chart: { polar: true, type: 'line' },
  title: { text: 'Polar Chart Example' },
  pane: { size: '80%' },
  xAxis: {
    categories: ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW'],
    tickmarkPlacement: 'on',
    lineWidth: 0
  },
  yAxis: { gridLineInterpolation: 'polygon', lineWidth: 0, min: 0 },
  series: [{
    name: 'Wind Speed',
    data: [8, 7, 6, 5, 6, 7, 8, 9],
    pointPlacement: 'on'
  }]
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highcharts Polar Chart</title>
  <script src="https://code.highcharts.com/highcharts.js"></script>
  <style>
    body {
      margin: 0;
      padding: 20px;
      font-family: 'Segoe UI', sans-serif;
      background-color: #f0f2f5;
      text-align: center;
    }
```

```
    #container {
      width: 95%;
      max-width: 800px;
      height: 500px;
      margin: 0 auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Polar Chart Example</h2>
  <p>Wind speed distribution across compass directions</p>

  <div id="container"></div>

  <script>
    Highcharts.chart('container', {
      chart: { polar: true, type: 'line' },
      title: { text: 'Polar Chart Example' },
      pane: { size: '80%' },
      xAxis: {
        categories: ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW'],
        tickmarkPlacement: 'on',
        lineWidth: 0
      },
      yAxis: {
        gridLineInterpolation: 'polygon',
        lineWidth: 0,
        min: 0
      },
      tooltip: {
        shared: true,
        pointFormat: '<span style="color:{series.color}">{series.name}</span>: <b>{point.y}</b><br/>'
      },
      series: [{
        name: 'Wind Speed',
        data: [8, 7, 6, 5, 6, 7, 8, 9],
        pointPlacement: 'on'
      }]
    });
  </script>
</body>
</html>
```

Gauge Charts

Gauge charts display a single value on a dial or speedometer-like display, ideal for KPIs or performance indicators.

Example gauge chart snippet:

```
    Highcharts.chart('container', {
      chart: {
        type: 'solidgauge'
      },
      title: {
```

```
      text: 'Gauge Chart Example'
    },
    tooltip: {
      enabled: true
    },
    pane: {
      startAngle: -150,
      endAngle: 150,
      background: {
        backgroundColor: '#EEE',
        innerRadius: '60%',
        outerRadius: '100%',
        shape: 'arc'
      }
    },
    yAxis: {
      min: 0,
      max: 200,
      title: {
        text: 'Speed'
      },
      stops: [
        [0.3, '#55BF3B'],  // green
        [0.6, '#DDDF0D'],  // yellow
        [0.9, '#DF5353']   // red
      ],
      lineWidth: 0,
      tickWidth: 0,
      minorTickInterval: null,
      labels: {
        y: 16
      }
    },
    plotOptions: {
      solidgauge: {
        dataLabels: {
          y: -10,
          borderWidth: 0,
          useHTML: true,
          format: '<div style="text-align:center"><span style="font-size:22px">{y}</span><br/><span s
        }
      }
    },
    series: [{
      name: 'Speed',
      data: [80],
      tooltip: {
        valueSuffix: ' km/h'
      }
    }]
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8" />
  <title>Highcharts Solid Gauge Chart</title>
<script src="https://code.highcharts.com/highcharts.js"></script>
<script src="https://code.highcharts.com/highcharts-more.js"></script>
<script src="https://code.highcharts.com/modules/solid-gauge.js"></script>

  <style>
    body {
      margin: 0;
      font-family: 'Segoe UI', sans-serif;
      background-color: #f4f4f4;
      text-align: center;
      padding: 20px;
    }
    #container {
      width: 95%;
      max-width: 600px;
      height: 400px;
      margin: auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Gauge Chart Example</h2>
  <p>Displays speed in km/h using radial scale</p>
  <div id="container"></div>

  <script>
    Highcharts.chart('container', {
      chart: {
        type: 'solidgauge'
      },
      title: {
        text: 'Gauge Chart Example'
      },
      tooltip: {
        enabled: true
      },
      pane: {
        startAngle: -150,
        endAngle: 150,
        background: {
          backgroundColor: '#EEE',
          innerRadius: '60%',
          outerRadius: '100%',
          shape: 'arc'
        }
      },
      yAxis: {
        min: 0,
        max: 200,
        title: {
          text: 'Speed'
        },
        stops: [
```

```
          [0.3, '#55BF3B'],  // green
          [0.6, '#DDDF0D'],  // yellow
          [0.9, '#DF5353']   // red
        ],
        lineWidth: 0,
        tickWidth: 0,
        minorTickInterval: null,
        labels: {
          y: 16
        }
      },
      plotOptions: {
        solidgauge: {
          dataLabels: {
            y: -10,
            borderWidth: 0,
            useHTML: true,
            format: '<div style="text-align:center"><span style="font-size:22px">{y}</span><br/><span s
          }
        }
      },
      series: [{
        name: 'Speed',
        data: [80],
        tooltip: {
          valueSuffix: ' km/h'
        }
      }]
    });
  </script>
</body>
</html>
```

Heatmaps

Heatmaps visualize matrix-style data with colors representing values, useful for correlations, activity logs, or geospatial data.

Example heatmap setup:

```
Highcharts.chart('container', {
  chart: { type: 'heatmap' },
  title: { text: 'Heatmap Example' },
  xAxis: { categories: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'] },
  yAxis: { categories: ['Morning', 'Afternoon', 'Evening'], title: null },
  colorAxis: { min: 0, minColor: '#FFFFFF', maxColor: '#7cb5ec' },
  series: [{
    name: 'Sales per time',
    borderWidth: 1,
    data: [
      [0, 0, 10], [0, 1, 19], [0, 2, 8],
      [1, 0, 24], [1, 1, 67], [1, 2, 22],
      [2, 0, 12], [2, 1, 38], [2, 2, 15],
      [3, 0, 24], [3, 1, 70], [3, 2, 32],
      [4, 0, 20], [4, 1, 49], [4, 2, 15]
    ],
    dataLabels: { enabled: true, color: '#000' }
  }]
```

```
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Highcharts Heatmap Example</title>
  <script src="https://code.highcharts.com/highcharts.js"></script>
  <script src="https://code.highcharts.com/modules/heatmap.js"></script>
  <script src="https://code.highcharts.com/modules/exporting.js"></script>
  <style>
    body {
      margin: 0;
      font-family: 'Segoe UI', sans-serif;
      background-color: #f0f2f5;
      text-align: center;
      padding: 20px;
    }
    #container {
      width: 95%;
      max-width: 800px;
      height: 500px;
      margin: auto;
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>
  <h2>Sales Heatmap</h2>
  <p>Sales per time slot across weekdays</p>
  <div id="container"></div>

  <script>
    Highcharts.chart('container', {
      chart: { type: 'heatmap' },
      title: { text: 'Heatmap Example' },
      xAxis: {
        categories: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
        title: { text: 'Day' }
      },
      yAxis: {
        categories: ['Morning', 'Afternoon', 'Evening'],
        title: { text: 'Time of Day' }
      },
      colorAxis: {
        min: 0,
        minColor: '#FFFFFF',
        maxColor: '#7cb5ec'
      },
      legend: {
        align: 'right',
        layout: 'vertical',
        margin: 0,
```

```
        verticalAlign: 'middle',
        y: 25,
        symbolHeight: 200
      },
      tooltip: {
        formatter: function () {
          return `<b>${this.series.name}</b><br>${this.series.xAxis.categories[this.point.x]} ` +
                 `${this.series.yAxis.categories[this.point.y]}<br>` +
                 `Sales: <b>${this.point.value}</b>`;
        }
      },
      series: [{
        name: 'Sales per time',
        borderWidth: 1,
        data: [
          [0, 0, 10], [0, 1, 19], [0, 2, 8],
          [1, 0, 24], [1, 1, 67], [1, 2, 22],
          [2, 0, 12], [2, 1, 38], [2, 2, 15],
          [3, 0, 24], [3, 1, 70], [3, 2, 32],
          [4, 0, 20], [4, 1, 49], [4, 2, 15]
        ],
        dataLabels: { enabled: true, color: '#000' }
      }]
    });
  </script>
</body>
</html>
```

10.3.4 Configuration-Heavy Example: Multi-Series Polar Gauge with Custom Theme

```
// Apply custom theme globally
Highcharts.setOptions({
  chart: { backgroundColor: '#1e1e1e', style: { fontFamily: 'Verdana' } },
  colors: ['#ff7f50', '#87cefa', '#da70d6', '#32cd32', '#6495ed'],
  title: { style: { color: '#fff' } },
  legend: { itemStyle: { color: '#ccc' } },
  tooltip: { backgroundColor: '#333', style: { color: '#fff' } }
});

Highcharts.chart('container', {
  chart: { polar: true, type: 'area' },
  title: { text: 'Multi-Series Polar Chart with Custom Theme' },
  pane: { size: '75%' },
  xAxis: {
    categories: ['Speed', 'Reliability', 'Comfort', 'Safety', 'Efficiency'],
    tickmarkPlacement: 'on',
    lineWidth: 0,
    labels: { style: { color: '#eee' } }
  },
  yAxis: {
    gridLineInterpolation: 'polygon',
    lineWidth: 0,
```

```
    min: 0,
    labels: { style: { color: '#bbb' } }
  },
  tooltip: { shared: true, valueSuffix: '%' },
  legend: { align: 'right', verticalAlign: 'top', layout: 'vertical' },
  series: [
    {
      name: 'Model A',
      data: [80, 90, 70, 85, 75],
      pointPlacement: 'on',
      fillOpacity: 0.5
    },
    {
      name: 'Model B',
      data: [70, 85, 80, 90, 65],
      pointPlacement: 'on',
      fillOpacity: 0.5
    }
  ]
});
```

This example highlights:

- A **custom dark theme** applied globally.
- A **polar area chart** comparing multiple series.
- Advanced styling for axes, labels, and tooltips.
- A flexible layout with a legend and shared tooltips.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Multi-Series Polar Chart with Custom Theme</title>
  <script src="https://code.highcharts.com/highcharts.js"></script>
  <style>
    body {
      margin: 0;
      padding: 20px;
      font-family: Verdana, sans-serif;
      background-color: #1e1e1e;
      color: #fff;
      text-align: center;
    }
    #container {
      width: 90%;
      max-width: 800px;
      height: 500px;
      margin: auto;
      border-radius: 8px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.5);
      background-color: #2e2e2e;
    }
  </style>
</head>
```

```
<body>
  <h2>Vehicle Comparison (Custom Theme)</h2>
  <div id="container"></div>

  <script>
    // Apply custom theme globally
    Highcharts.setOptions({
      chart: { backgroundColor: '#1e1e1e', style: { fontFamily: 'Verdana' } },
      colors: ['#ff7f50', '#87cefa', '#da70d6', '#32cd32', '#6495ed'],
      title: { style: { color: '#fff' } },
      legend: { itemStyle: { color: '#ccc' } },
      tooltip: { backgroundColor: '#333', style: { color: '#fff' } }
    });

    Highcharts.chart('container', {
      chart: { polar: true, type: 'area' },
      title: { text: 'Multi-Series Polar Chart with Custom Theme' },
      pane: { size: '75%' },
      xAxis: {
        categories: ['Speed', 'Reliability', 'Comfort', 'Safety', 'Efficiency'],
        tickmarkPlacement: 'on',
        lineWidth: 0,
        labels: { style: { color: '#eee' } }
      },
      yAxis: {
        gridLineInterpolation: 'polygon',
        lineWidth: 0,
        min: 0,
        labels: { style: { color: '#bbb' } }
      },
      tooltip: { shared: true, valueSuffix: '%' },
      legend: { align: 'right', verticalAlign: 'top', layout: 'vertical' },
      series: [
        {
          name: 'Model A',
          data: [80, 90, 70, 85, 75],
          pointPlacement: 'on',
          fillOpacity: 0.5
        },
        {
          name: 'Model B',
          data: [70, 85, 80, 90, 65],
          pointPlacement: 'on',
          fillOpacity: 0.5
        }
      ]
    });
  </script>
</body>
</html>
```

10.3.5 Summary

- Highcharts supports easy application of built-in and custom themes to unify chart styling.
- Advanced chart types such as polar charts, gauge charts, and heatmaps extend your visualization possibilities.
- The options API is highly flexible, allowing deep customization and integration of plugins.
- Combining these features empowers you to build visually consistent and complex dashboards tailored to your needs.

Chapter 11.

Three.js for 3D Visualizations

1. Basics of 3D Visualization
2. When to Use 3D for Data
3. Simple 3D Bar and Scatter Charts

11 Three.js for 3D Visualizations

11.1 Basics of 3D Visualization

3D visualization adds a new dimension—literally—to data storytelling by leveraging depth, perspective, and interactivity. When used thoughtfully, 3D graphics can provide richer insights, clearer spatial relationships, and engaging user experiences that go beyond what traditional 2D charts can offer.

In the browser, **Three.js** is the leading JavaScript library for creating 3D graphics, providing a powerful yet approachable API to build and render complex 3D scenes using WebGL technology.

11.1.1 Core Concepts in Three.js

Creating a 3D scene with Three.js involves several fundamental components:

11.1.2 Scene

The **scene** acts as the container that holds all 3D objects, lights, and cameras. Think of it as your virtual world or stage where everything is positioned.

```
const scene = new THREE.Scene();
```

11.1.3 Camera

The **camera** defines the viewpoint from which you observe the scene. Different camera types exist, but the most common is the **PerspectiveCamera**, which simulates human eye perspective—objects farther away appear smaller, creating realistic depth.

```
const camera = new THREE.PerspectiveCamera(
  75,                // Field of view (degrees)
  window.innerWidth / window.innerHeight,  // Aspect ratio
  0.1,               // Near clipping plane
  1000               // Far clipping plane
);
```

11.1.4 Renderer

The **renderer** processes the scene and camera to display the 3D graphics on an HTML canvas element.

```
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

11.1.5 Meshes

Meshes are the visible 3D objects composed of:

- **Geometry**: Defines the shape (e.g., cube, sphere).
- **Material**: Defines the surface appearance (color, texture, reflectivity).

Example: a cube mesh.

```
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

11.1.6 Lights

Lights illuminate the scene, affecting how materials appear. Three.js offers various lights like ambient, directional, point, and spotlights.

```
const light = new THREE.AmbientLight(0xffffff); // Soft white light
scene.add(light);
```

11.1.7 2D vs 3D Visualization

Aspect	2D Visualization	3D Visualization
Dimension	Width and height	Width, height, and depth
Perspective	Orthogonal or flat	Perspective mimics real-world depth
Interaction	Pan, zoom, hover	Rotate, zoom, pan in 3D space
Data complexity	Simple relationships, fewer variables	Complex spatial or multi-dimensional data
Use cases	Standard charts, dashboards	Geospatial data, networks, volumetric data

3D visualizations can enhance understanding by showing depth cues and spatial relationships, but they also add complexity and require careful design to avoid clutter or confusion.

11.1.8 Basic Cube Example: Rendering a Rotating Cube with Three.js

This simple example sets up a Three.js scene, adds a cube, and animates it with rotation.

```
<script>
  // 1. Setup scene, camera, and renderer
  const scene = new THREE.Scene();
  const camera = new THREE.PerspectiveCamera(
    75, window.innerWidth / window.innerHeight, 0.1, 1000
  );
  const renderer = new THREE.WebGLRenderer({ antialias: true });
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // 2. Create a cube mesh
  const geometry = new THREE.BoxGeometry();
  const material = new THREE.MeshBasicMaterial({ color: 0x0077ff, wireframe: true });
  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  // 3. Position the camera
  camera.position.z = 5;

  // 4. Animation loop
  function animate() {
    requestAnimationFrame(animate);

    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;

    renderer.render(scene, camera);
  }
  animate();

  // 5. Handle window resize
  window.addEventListener('resize', () => {
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
  });
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Three.js Basic Cube</title>
  <style>body { margin: 0; }</style>
</head>
```

```
<body>
  <script src="https://cdn.jsdelivr.net/npm/three@0.152.0/build/three.min.js"></script>
  <script>
    // 1. Setup scene, camera, and renderer
    const scene = new THREE.Scene();
    const camera = new THREE.PerspectiveCamera(
      75, window.innerWidth / window.innerHeight, 0.1, 1000
    );
    const renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // 2. Create a cube mesh
    const geometry = new THREE.BoxGeometry();
    const material = new THREE.MeshBasicMaterial({ color: 0x0077ff, wireframe: true });
    const cube = new THREE.Mesh(geometry, material);
    scene.add(cube);

    // 3. Position the camera
    camera.position.z = 5;

    // 4. Animation loop
    function animate() {
      requestAnimationFrame(animate);

      cube.rotation.x += 0.01;
      cube.rotation.y += 0.01;

      renderer.render(scene, camera);
    }
    animate();

    // 5. Handle window resize
    window.addEventListener('resize', () => {
      camera.aspect = window.innerWidth / window.innerHeight;
      camera.updateProjectionMatrix();
      renderer.setSize(window.innerWidth, window.innerHeight);
    });
  </script>
</body>
</html>
```

11.1.9 Summary

- Three.js provides an accessible way to create interactive 3D scenes in the browser using WebGL.
- Key components include scenes, cameras, renderers, meshes, and lights.
- 3D visualizations add depth and perspective, offering new ways to understand complex data.
- A simple rotating cube example demonstrates the essential setup and rendering loop.

11.2 When to Use 3D for Data

While 3D visualizations can be captivating and powerful, they are not always the best choice. Adding a third dimension increases complexity and can sometimes hinder rather than help data comprehension. Knowing when 3D adds real value versus when it becomes a distraction is essential for effective data storytelling.

11.2.1 When 3D Adds Value

Spatial Data and Geographical Visualization

Data that inherently exists in three-dimensional space—such as geospatial data—benefits naturally from 3D rendering. Examples include:

- **Globe visualizations**: Showing global patterns like flight routes, weather systems, or population density on a rotating Earth model.
- **Terrain mapping**: Elevation data or 3D city models that reveal topography or building heights.
- **Astronomical data**: Star maps or solar system simulations where depth conveys real physical distances.

3D in these cases enables intuitive understanding of spatial relationships, directions, and scale that 2D maps cannot easily convey.

Simulation Environments and Time-Varying Data

In scientific and engineering fields, 3D visualization helps explore simulations with multiple interacting variables over time, such as:

- Fluid dynamics showing flow patterns around objects.
- Molecular structures and chemical simulations.
- Medical imaging (MRI, CT scans) visualized as volumetric data.

The depth dimension here is critical for grasping complex, layered structures or behaviors evolving in space.

Multi-Variable Comparisons and Complex Data

3D can help represent data with multiple quantitative dimensions, for example:

- **3D scatter plots** that plot points using X, Y, and Z to compare three variables simultaneously.
- **3D histograms or bar charts** that stack or group data along an extra axis.
- **Network graphs** with nodes and edges positioned in 3D to reduce clutter and reveal hidden connections.

When carefully designed, these visualizations enable viewers to explore rich datasets that would be difficult to parse in flat 2D.

11.2.2 When 3D Becomes a Distraction

Overcomplicating Simple Data

If your data has no natural third dimension or spatial component, adding 3D may confuse users rather than help. For example:

- Basic time series or categorical data often work best as clean 2D line or bar charts.
- Unnecessary 3D effects can obscure values, distort scales, or make comparisons harder.

Navigation and Interpretation Challenges

3D visualizations require users to manipulate views (rotate, zoom, pan) to understand data fully. This interaction can slow down comprehension if not intuitive or if controls are missing.

Visual Artifacts and Occlusion

Depth can cause occlusion, where some data points hide behind others. Lighting and perspective may create misleading shadows or distort shapes, requiring careful design and user training.

11.2.3 Example Scenarios

Scenario	2D or 3D?	Reasoning
Global flight route maps	3D	Spatial data mapped naturally on a globe with rotation.
Sales over time by region	2D	Clear trends easier to see with simple line or bar charts.
Molecular protein structures	3D	Complex spatial relationships best understood in 3D.
Basic bar chart comparing values	2D	3D bars distort perception of height; 2D is clearer.
3D scatter plot for multivariate data	3D	Enables viewing three variables simultaneously.
Pie chart with small segments	2D	Adding 3D tilt often distorts segment size perception.

11.2.4 Summary

- Use 3D visualizations primarily when your data naturally involves spatial relationships, multiple variables, or simulation environments.

- Avoid 3D if it complicates simple data, hinders interpretation, or causes visual artifacts.
- Thoughtful use of 3D can deepen insight, but it requires balancing complexity with clarity and user interaction.

11.3 Simple 3D Bar and Scatter Charts

Building 3D visualizations with Three.js involves creating geometric shapes to represent data points and arranging them meaningfully within a 3D coordinate system. In this section, we'll guide you through creating a simple 3D bar chart and a 3D scatter plot, covering geometry setup, axis creation, and user interaction such as rotation and zoom.

11.3.1 Setting Up Common Essentials

Before diving into chart-specific code, let's establish the common scene setup with a camera, renderer, and basic lighting to illuminate the scene:

```
const scene = new THREE.Scene();

const camera = new THREE.PerspectiveCamera(
  60,
  window.innerWidth / window.innerHeight,
  0.1,
  1000
);
camera.position.set(30, 30, 50);
camera.lookAt(scene.position);

const renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Lighting
const ambientLight = new THREE.AmbientLight(0x404040, 2); // Soft white light
scene.add(ambientLight);

const directionalLight = new THREE.DirectionalLight(0xffffff, 1);
directionalLight.position.set(50, 50, 50);
scene.add(directionalLight);

// Controls for rotation and zoom
const controls = new THREE.OrbitControls(camera, renderer.domElement);
```

Make sure to include the OrbitControls script from Three.js examples to enable mouse interaction:

```
<script src="https://cdn.jsdelivr.net/npm/three@0.152.0/examples/js/controls/OrbitControls.min.js"></sc
```

11.3.2 3D Bar Chart Example

Step 1: Sample Data

We will visualize sales data for 5 categories:

```
const data = [
  { category: 'A', value: 10 },
  { category: 'B', value: 15 },
  { category: 'C', value: 7 },
  { category: 'D', value: 20 },
  { category: 'E', value: 13 }
];
```

Step 2: Create Axes

For simplicity, axes can be represented by lines:

```
function createAxis() {
  const material = new THREE.LineBasicMaterial({ color: 0x000000 });
  const points = [];

  // X axis
  points.push(new THREE.Vector3(0, 0, 0));
  points.push(new THREE.Vector3(30, 0, 0));
  const xAxis = new THREE.BufferGeometry().setFromPoints(points);
  scene.add(new THREE.Line(xAxis, material));

  points.length = 0;
  // Y axis
  points.push(new THREE.Vector3(0, 0, 0));
  points.push(new THREE.Vector3(0, 25, 0));
  const yAxis = new THREE.BufferGeometry().setFromPoints(points);
  scene.add(new THREE.Line(yAxis, material));

  points.length = 0;
  // Z axis
  points.push(new THREE.Vector3(0, 0, 0));
  points.push(new THREE.Vector3(0, 0, 15));
  const zAxis = new THREE.BufferGeometry().setFromPoints(points);
  scene.add(new THREE.Line(zAxis, material));
}
createAxis();
```

Step 3: Add Bars

Each bar is a box geometry with height proportional to the data value.

```
const barWidth = 3;
const gap = 2;

data.forEach((d, i) => {
  const height = d.value;
  const geometry = new THREE.BoxGeometry(barWidth, height, barWidth);
  const material = new THREE.MeshPhongMaterial({ color: 0x0077ff });
  const bar = new THREE.Mesh(geometry, material);

  // Position bars along X-axis, raised half their height on Y so base is on axis
  bar.position.set(i * (barWidth + gap), height / 2, 0);
```

```
  scene.add(bar);
});
```

Step 4: Animate and Render

```
function animate() {
  requestAnimationFrame(animate);
  controls.update(); // Required for damping/auto rotation
  renderer.render(scene, camera);
}
animate();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>3D Bar Chart with Three.js</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      font-family: sans-serif;
    }
  </style>
  <!-- Use version 0.146.0 to ensure OrbitControls is available globally -->
  <script src="https://cdn.jsdelivr.net/npm/three@0.146.0/build/three.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/three@0.146.0/examples/js/controls/OrbitControls.js"></scri
</head>
<body>
  <script>
    // Scene Setup
    const scene = new THREE.Scene();

    const camera = new THREE.PerspectiveCamera(
      60,
      window.innerWidth / window.innerHeight,
      0.1,
      1000
    );
    camera.position.set(30, 30, 50);
    camera.lookAt(scene.position);

    const renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // Lights
    const ambientLight = new THREE.AmbientLight(0x404040, 2);
    scene.add(ambientLight);

    const directionalLight = new THREE.DirectionalLight(0xffffff, 1);
    directionalLight.position.set(50, 50, 50);
    scene.add(directionalLight);
```

```
    // Orbit Controls
    const controls = new THREE.OrbitControls(camera, renderer.domElement);

    // Sample Data
    const data = [
      { category: 'A', value: 10 },
      { category: 'B', value: 15 },
      { category: 'C', value: 7 },
      { category: 'D', value: 20 },
      { category: 'E', value: 13 }
    ];

    // Axes
    function createAxis() {
      const material = new THREE.LineBasicMaterial({ color: 0x000000 });

      const xPoints = [new THREE.Vector3(0, 0, 0), new THREE.Vector3(30, 0, 0)];
      const yPoints = [new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 25, 0)];
      const zPoints = [new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 0, 15)];

      scene.add(new THREE.Line(new THREE.BufferGeometry().setFromPoints(xPoints), material));
      scene.add(new THREE.Line(new THREE.BufferGeometry().setFromPoints(yPoints), material));
      scene.add(new THREE.Line(new THREE.BufferGeometry().setFromPoints(zPoints), material));
    }
    createAxis();

    // Bars
    const barWidth = 3;
    const gap = 2;

    data.forEach((d, i) => {
      const height = d.value;
      const geometry = new THREE.BoxGeometry(barWidth, height, barWidth);
      const material = new THREE.MeshPhongMaterial({ color: 0x0077ff });
      const bar = new THREE.Mesh(geometry, material);
      bar.position.set(i * (barWidth + gap), height / 2, 0);
      scene.add(bar);
    });

    // Animate and Render
    function animate() {
      requestAnimationFrame(animate);
      controls.update();
      renderer.render(scene, camera);
    }
    animate();

    // Responsive Resize
    window.addEventListener('resize', () => {
      camera.aspect = window.innerWidth / window.innerHeight;
      camera.updateProjectionMatrix();
      renderer.setSize(window.innerWidth, window.innerHeight);
    });
  </script>
</body>
</html>
```

11.3.3 3D Scatter Plot Example

Step 1: Sample Data

This dataset has three numeric variables (x, y, z):

```
const scatterData = [
  { x: 5, y: 10, z: 2 },
  { x: 15, y: 8, z: 5 },
  { x: 10, y: 15, z: 8 },
  { x: 20, y: 18, z: 10 },
  { x: 25, y: 5, z: 12 }
];
```

Step 2: Create Scatter Points

Use spheres for points, sized uniformly and colored distinctly:

```
scatterData.forEach((point) => {
  const geometry = new THREE.SphereGeometry(0.8, 16, 16);
  const material = new THREE.MeshPhongMaterial({ color: 0xff5533 });
  const sphere = new THREE.Mesh(geometry, material);

  sphere.position.set(point.x, point.y, point.z);
  scene.add(sphere);
});
```

Step 3: Reuse Axes and Controls

Use the same axis creation and controls setup from the bar chart example to provide spatial reference and interaction.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>3D Scatter Plot with Three.js</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      background-color: #111;
    }
  </style>
  <!-- Use Three.js 0.146.0 with global OrbitControls support -->
  <script src="https://cdn.jsdelivr.net/npm/three@0.146.0/build/three.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/three@0.146.0/examples/js/controls/OrbitControls.js"></scri
</head>
<body>
  <script>
    // Create scene and camera
    const scene = new THREE.Scene();
    scene.background = new THREE.Color('#111');

    const camera = new THREE.PerspectiveCamera(
```

```
    60,
    window.innerWidth / window.innerHeight,
    0.1,
    1000
);
camera.position.set(30, 30, 50);
camera.lookAt(scene.position);

// Renderer
const renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Lighting
const ambientLight = new THREE.AmbientLight(0x404040, 2);
scene.add(ambientLight);

const directionalLight = new THREE.DirectionalLight(0xffffff, 1);
directionalLight.position.set(50, 50, 50);
scene.add(directionalLight);

// Controls
const controls = new THREE.OrbitControls(camera, renderer.domElement);

// Sample Data
const scatterData = [
  { x: 5, y: 10, z: 2 },
  { x: 15, y: 8, z: 5 },
  { x: 10, y: 15, z: 8 },
  { x: 20, y: 18, z: 10 },
  { x: 25, y: 5, z: 12 }
];

// Create scatter points
scatterData.forEach((point) => {
  const geometry = new THREE.SphereGeometry(0.8, 16, 16);
  const material = new THREE.MeshPhongMaterial({ color: 0xff5533 });
  const sphere = new THREE.Mesh(geometry, material);

  sphere.position.set(point.x, point.y, point.z);
  scene.add(sphere);
});

// Optional: Add axes
function addAxis(from, to, color) {
  const material = new THREE.LineBasicMaterial({ color });
  const points = [new THREE.Vector3(...from), new THREE.Vector3(...to)];
  const geometry = new THREE.BufferGeometry().setFromPoints(points);
  const line = new THREE.Line(geometry, material);
  scene.add(line);
}

addAxis([0, 0, 0], [30, 0, 0], 0xff0000); // X (Red)
addAxis([0, 0, 0], [0, 30, 0], 0x00ff00); // Y (Green)
addAxis([0, 0, 0], [0, 0, 30], 0x0000ff); // Z (Blue)

// Animate
function animate() {
```

```
      requestAnimationFrame(animate);
      controls.update();
      renderer.render(scene, camera);
    }
    animate();

    // Responsive
    window.addEventListener('resize', () => {
      camera.aspect = window.innerWidth / window.innerHeight;
      camera.updateProjectionMatrix();
      renderer.setSize(window.innerWidth, window.innerHeight);
    });
  </script>
</body>
</html>
```

11.3.4 Summary

- **3D bar charts** use box geometries sized and positioned by data, laid out along axes.
- **3D scatter plots** use spheres positioned in 3D space according to multiple variables.
- Axes help ground the visualization in space, and simple line geometries can represent them.
- OrbitControls enable users to rotate, pan, and zoom, making exploration intuitive.
- Lighting and materials improve depth perception and visual clarity.

Chapter 12.

Dashboards Layout and Design Considerations

1. Grids, Flexbox, and CSS for Dashboards
2. Responsive Containers
3. Mobile-Friendly Visualizations

12 Dashboards Layout and Design Considerations

12.1 Grids, Flexbox, and CSS for Dashboards

An effective dashboard isn't just about the charts it contains—it's also about **how those charts are laid out and organized**. A well-designed layout helps users quickly find and compare information, while maintaining visual balance and clarity. This section introduces key CSS techniques—**CSS Grid** and **Flexbox**—that provide powerful tools for structuring dashboards with responsiveness and modularity.

12.1.1 Why Layout Matters in Dashboards

Dashboards often combine multiple charts, tables, and controls. Without a clear layout strategy:

- Visual elements can become cluttered or misaligned.
- Users may struggle to interpret related data side by side.
- The interface might break on different screen sizes or devices.

Good layout practices ensure:

- Logical grouping of related visualizations.
- Consistent spacing and alignment.
- Flexibility to adapt to varying screen sizes.

12.1.2 CSS Grid: The Dashboard Backbone

CSS Grid provides a two-dimensional grid system—rows and columns—that makes complex layouts intuitive and maintainable.

12.1.3 Key Features:

- Define explicit rows and columns with sizes.
- Place elements anywhere on the grid using grid lines or areas.
- Easily create modular, scalable layouts.

12.1.4 Example: Simple 2x2 Dashboard Grid

```
<div class="dashboard-grid">
  <div class="chart chart1">Bar Chart</div>
  <div class="chart chart2">Line Chart</div>
  <div class="chart chart3">Pie Chart</div>
  <div class="chart chart4">Scatter Plot</div>
</div>
```

```
.dashboard-grid {
  display: grid;
  grid-template-columns: repeat(2, 1fr);  /* Two equal columns */
  grid-template-rows: repeat(2, 300px);   /* Two rows, fixed height */
  gap: 20px;                              /* Space between grid items */
  padding: 20px;
}

.chart {
  background-color: #f5f5f5;
  border: 1px solid #ddd;
  border-radius: 6px;
  display: flex;
  align-items: center;
  justify-content: center;
  font-weight: bold;
  font-size: 1.2rem;
}
```

This layout creates a clean 2x2 grid where each chart container occupies one cell with equal width and height and consistent spacing between them.

12.1.5 Flexbox: Flexible Alignment and Distribution

Flexbox excels at laying out components in one dimension (row or column) and controlling spacing, alignment, and wrapping.

12.1.6 Key Uses in Dashboards:

- Align controls or legends horizontally or vertically.
- Stack charts in a column or row that adapts to content size.
- Distribute space evenly or with fixed gaps.

12.1.7 Example: Horizontal Toolbar with Buttons

```
<div class="toolbar">
  <button>Filter</button>
  <button>Export</button>
  <button>Refresh</button>
</div>
```

```
.toolbar {
  display: flex;
  gap: 15px;          /* Space between buttons */
  padding: 10px;
  background: #eee;
  border-radius: 4px;
  justify-content: flex-start;
  align-items: center;
}

button {
  padding: 8px 16px;
  border: none;
  background: #0077ff;
  color: white;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1rem;
}
```

Flexbox effortlessly aligns the buttons in a row with consistent spacing and vertical centering.

12.1.8 Combining Grid and Flexbox for Modular Design

You can combine CSS Grid and Flexbox to build modular dashboard components:

- Use **Grid** for the overall dashboard structure.
- Use **Flexbox** inside each chart container for internal layout (e.g., placing titles, legends, controls).

12.1.9 Example: Chart Container with Title and Chart Area

```
<div class="chart-container">
  <h3>Sales Over Time</h3>
  <div class="chart-area">[Chart renders here]</div>
</div>
```

```
.chart-container {
  display: flex;
  flex-direction: column;
  gap: 10px;
```

```
  padding: 10px;
  background: #fff;
  border-radius: 6px;
  box-shadow: 0 0 6px rgba(0,0,0,0.1);
}

.chart-area {
  flex-grow: 1;
  /* Chart SVG or Canvas will fill this area */
}
```

This structure ensures the title stays above the chart, spaced nicely, and the chart expands to fill the container.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>CSS Grid & Flexbox Dashboard</title>
  <style>
    body {
      margin: 0;
      font-family: 'Segoe UI', sans-serif;
      background-color: #fafafa;
    }

    /* Toolbar using Flexbox */
    .toolbar {
      display: flex;
      gap: 15px;
      padding: 15px 20px;
      background: #eee;
      border-bottom: 1px solid #ccc;
      align-items: center;
    }

    button {
      padding: 8px 16px;
      border: none;
      background: #0077ff;
      color: white;
      border-radius: 4px;
      cursor: pointer;
      font-size: 1rem;
    }

    /* Grid layout for dashboard */
    .dashboard-grid {
      display: grid;
      grid-template-columns: repeat(2, 1fr);
      grid-template-rows: repeat(2, 300px);
      gap: 20px;
      padding: 20px;
    }
```

```
    .chart {
      background-color: #f5f5f5;
      border: 1px solid #ddd;
      border-radius: 6px;
      display: flex;
      align-items: center;
      justify-content: center;
      font-weight: bold;
      font-size: 1.2rem;
    }

    /* Chart container using Flexbox */
    .chart-container {
      display: flex;
      flex-direction: column;
      gap: 10px;
      padding: 10px;
      background: #fff;
      border-radius: 6px;
      box-shadow: 0 0 6px rgba(0,0,0,0.1);
      margin: 20px;
    }

    .chart-area {
      flex-grow: 1;
      background: #f9f9f9;
      border: 1px dashed #ccc;
      display: flex;
      align-items: center;
      justify-content: center;
      font-size: 1rem;
      color: #777;
    }
  </style>
</head>
<body>
  <!-- Flexbox Toolbar -->
  <div class="toolbar">
    <button>Filter</button>
    <button>Export</button>
    <button>Refresh</button>
  </div>

  <!-- Grid Dashboard -->
  <div class="dashboard-grid">
    <div class="chart chart1">Bar Chart</div>
    <div class="chart chart2">Line Chart</div>
    <div class="chart chart3">Pie Chart</div>
    <div class="chart chart4">Scatter Plot</div>
  </div>

  <!-- Flexbox Chart Module -->
  <div class="chart-container">
    <h3>Sales Over Time</h3>
    <div class="chart-area">[Chart renders here]</div>
  </div>
</body>
</html>
```

12.1.10 Best Practices for Dashboard Layouts

- **Consistent spacing:** Use grid `gap` and flex `gap` to maintain uniform margins.
- **Modular containers:** Encapsulate each visualization and its controls for reusability.
- **Responsive adjustments:** Use CSS media queries to modify grid columns or flex direction on smaller screens.
- **Avoid fixed widths:** Prefer fractional (`fr`) units in Grid and flexible sizing in Flexbox for adaptability.

12.1.11 Summary

- **CSS Grid** is ideal for the main dashboard layout with rows and columns.
- **Flexbox** works best for internal component alignment and control placement.
- Combining both yields clean, flexible, and maintainable dashboards.
- Proper use of spacing, modular containers, and responsive design ensures your dashboard looks great across devices.

12.2 Responsive Containers

In modern dashboards, containers that hold charts and visualizations must **adapt gracefully** to various screen sizes—from large desktop monitors to tablets and small laptops. Responsive containers ensure your visualizations maintain clarity and usability no matter the device or window size.

12.2.1 Building Containers That Adapt

Use Relative Units for Width and Height

Rather than fixed pixel sizes, use **percentage widths**, and viewport-relative units like `vw` (viewport width) and `vh` (viewport height) to make containers flexible.

```
.chart-container {
  width: 80%;          /* 80% of the parent container */
  height: 50vh;        /* Half the viewport height */
  margin: auto;
  border: 1px solid #ccc;
  border-radius: 6px;
  background: #fff;
}
```

This example creates a container that scales with the viewport width and height, centering it horizontally.

Media Queries for Fine-Tuned Adjustments

Media queries enable you to **adjust container sizes or layout** based on specific screen widths:

```
@media (max-width: 768px) {
  .chart-container {
    width: 95%;
    height: 40vh;
  }
}

@media (max-width: 480px) {
  .chart-container {
    width: 100%;
    height: 30vh;
  }
}
```

Here, as the screen narrows, the container widens to fill more horizontal space, while its height adjusts for smaller viewports, maintaining usability on mobile devices.

Resizing Charts Inside Responsive Containers

While CSS can handle container resizing, chart libraries like **D3.js** and **Chart.js** often require **explicit resizing** calls to update the visualization to fit the new container dimensions.

Resizing Chart.js Charts

Chart.js has built-in support for responsiveness when configured properly:

```
const ctx = document.getElementById('myChart').getContext('2d');
const myChart = new Chart(ctx, {
  type: 'bar',
  data: {/*...*/},
  options: {
    responsive: true,
    maintainAspectRatio: false
  }
});
```

To make the chart resize smoothly:

- Set `responsive: true` to enable dynamic resizing.
- Set `maintainAspectRatio: false` if you want the height to adapt independently of width.
- Ensure the canvas's container resizes using CSS.

12.2.2 Handling Resizing in D3 Visualizations

D3 visualizations usually require listening for **window resize events** and recalculating scales, axes, and elements dimensions accordingly.

Example Resize Listener

```
function renderChart() {
  const container = document.querySelector('.chart-container');
  const width = container.clientWidth;
  const height = container.clientHeight;

  // Clear existing SVG or update its size
  d3.select('svg').attr('width', width).attr('height', height);

  // Recompute scales, axes, and redraw chart elements here...
}

// Initial render
renderChart();

// Listen for window resize
window.addEventListener('resize', () => {
  renderChart();
});
```

This pattern:

- Queries container size dynamically.
- Adjusts SVG size accordingly.
- Recomputes all scale domains and ranges.
- Redraws or updates chart elements.

12.2.3 Example: Responsive Container and Chart.js Chart

```
<div class="chart-container" style="height:400px;">
  <canvas id="barChart"></canvas>
</div>

<style>
  .chart-container {
    width: 90%;
    max-width: 900px;
    margin: auto;
  }

  @media (max-width: 600px) {
    .chart-container {
      width: 100%;
      height: 300px;
    }
  }
</style>

<script>
  const ctx = document.getElementById('barChart').getContext('2d');
  const barChart = new Chart(ctx, {
    type: 'bar',
```

```
    data: {
      labels: ['A', 'B', 'C', 'D'],
      datasets: [{
        label: 'Sales',
        data: [12, 19, 3, 5],
        backgroundColor: 'steelblue'
      }]
    },
    options: {
      responsive: true,
      maintainAspectRatio: false
    }
  });
</script>
```

Resize the browser window to see how the chart adapts within its container.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Responsive Bar Chart with Chart.js</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      margin: 0;
      font-family: Arial, sans-serif;
      background-color: #f4f4f4;
      padding: 20px;
      text-align: center;
    }

    .chart-container {
      width: 90%;
      max-width: 900px;
      height: 400px;
      margin: auto;
    }

    @media (max-width: 600px) {
      .chart-container {
        width: 100%;
        height: 300px;
      }
    }
  </style>
</head>
<body>
  <h2>Sales by Category</h2>
  <div class="chart-container">
    <canvas id="barChart"></canvas>
  </div>

  <script>
    const ctx = document.getElementById('barChart').getContext('2d');
```

```
    const barChart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['A', 'B', 'C', 'D'],
        datasets: [{
          label: 'Sales',
          data: [12, 19, 3, 5],
          backgroundColor: 'steelblue'
        }]
      },
      options: {
        responsive: true,
        maintainAspectRatio: false
      }
    });
  </script>
</body>
</html>
```

12.2.4 Summary

- Use relative units (`%`, `vw`, `vh`) to create fluid containers.
- Employ media queries to tweak container dimensions on different devices.
- For Chart.js, enable `responsive` and `maintainAspectRatio` options.
- For D3, implement resize event listeners that update chart dimensions and scales.
- Responsive containers combined with dynamic resizing deliver seamless visualization experiences across devices.

12.3 Mobile-Friendly Visualizations

With the growing use of smartphones and tablets, designing **mobile-friendly visualizations** has become essential. Small screens and touch interfaces introduce unique challenges and require thoughtful adaptation to ensure usability without sacrificing insight.

12.3.1 Challenges of Mobile Visualization

- **Limited screen space:** Less room for complex charts or multiple visualizations side-by-side.
- **Touch interaction:** No precise mouse pointer; finger taps require larger, well-spaced targets.
- **Performance constraints:** Mobile devices may have less processing power and slower

network speeds.
- **Readability:** Smaller text and crowded labels can become illegible.

12.3.2 Strategies for Mobile Optimization

Simplify Visualizations

Reduce chart complexity by:

- Choosing simpler chart types (e.g., bar charts over complex scatterplots).
- Showing fewer data points or summarizing data (aggregation).
- Avoiding unnecessary decorations or "chartjunk."

Example: Stacked Layout Instead of Side-by-Side

Instead of a multi-column desktop layout, stack charts vertically to maximize width and readability.

```
@media (max-width: 600px) {
  .dashboard-grid {
    display: block;
  }

  .chart {
    margin-bottom: 20px;
    width: 100%;
    height: 300px;
  }
}
```

12.3.3 Responsive Text and Labels

Ensure text scales appropriately for smaller screens:

- Use relative font units like `em` or `rem`.
- Adjust label density or abbreviate text to avoid crowding.
- Consider tooltips or tap-to-show labels instead of persistent text.

12.3.4 Tap-Friendly Interaction

Design interactive elements with touch in mind:

- Use larger clickable areas (recommended minimum: 44x44 pixels).
- Space buttons and interactive points apart.

- Replace hover effects with tap or long-press actions.
- Provide clear visual feedback on taps.

12.3.5 Progressive Enhancement and Graceful Degradation

Progressive Enhancement means building a baseline experience that works everywhere, then adding advanced features on capable devices.

Graceful Degradation means ensuring the visualization remains usable even if some features are unsupported.

For example:

- Provide a static image or simple table fallback when JavaScript or SVG isn't supported.
- Use media queries to hide or simplify elements on smaller screens.
- Disable complex animations or interactions on low-powered devices.

12.3.6 Performance Considerations

- Load smaller datasets or summaries on mobile.
- Lazy-load charts only when they enter the viewport.
- Avoid heavy animations or transitions that drain battery or cause lag.

12.3.7 Example: Tap-Friendly Bar Chart with Responsive Layout

```
<div class="chart-container">
  <canvas id="mobileBarChart"></canvas>
</div>

<style>
  .chart-container {
    width: 100%;
    max-width: 500px;
    margin: auto;
  }

  @media (max-width: 480px) {
    .chart-container {
      height: 250px;
    }
  }

  /* Make bars tappable with some padding */
  canvas {
```

```
    touch-action: manipulation; /* Improves touch responsiveness */
  }
</style>
```

JavaScript would enable tapping bars to show tooltips or highlight them, replacing hover with touch-friendly events.

12.3.8 Summary

- Mobile requires **simpler, cleaner visualizations** with stacked layouts.
- Text and labels must be **responsive and legible**.
- Interactions should be designed for **touch**, with larger targets and tap feedback.
- Employ **progressive enhancement** to serve all users well.
- Consider **performance** to keep visualizations smooth and efficient.

By thoughtfully adapting visualizations for mobile, you ensure insights are accessible wherever users go.

Chapter 13.

Dashboards Integrating Filters and Controls

1. Dropdowns, Sliders, and Inputs
2. Two-Way Binding with State
3. Live Data Updates (WebSocket or Polling)

13 Dashboards Integrating Filters and Controls

13.1 Dropdowns, Sliders, and Inputs

Interactive dashboards often require **controls** that let users filter data, select metrics, or adjust views dynamically. Common form elements like **dropdowns (`<select>`)**, **sliders (`<input type="range">`)**, and other inputs provide intuitive interfaces to modify visualizations in real time.

This section demonstrates how to build such controls and connect them to charts, enabling dynamic updates based on user input.

13.1.1 Building Form Controls for Filtering

Dropdown (`select`) for Metric Selection

A dropdown lets users choose from a list of options, such as different metrics or categories.

```
<label for="metric-select">Choose Metric:</label>
<select id="metric-select">
  <option value="sales">Sales</option>
  <option value="profit">Profit</option>
  <option value="expenses">Expenses</option>
</select>
```

You can listen for changes to update the chart accordingly.

Slider (`input type"range"`) for Date Range or Value Filtering

A range slider allows continuous adjustment, ideal for filtering data by date or value range.

```
<label for="date-range">Select Date Range (days):</label>
<input type="range" id="date-range" min="1" max="30" value="15" />
<span id="date-range-value">15</span> days
```

As users slide, you can capture the value and update the displayed data.

13.1.2 Integrating Controls with Charts

Example: Bar Chart with Metric Selector and Date Range Slider

```
<div>
  <label for="metric-select">Metric:</label>
  <select id="metric-select">
    <option value="sales">Sales</option>
    <option value="profit">Profit</option>
    <option value="expenses">Expenses</option>
  </select>
```

```
  <label for="date-range">Last N Days:</label>
  <input type="range" id="date-range" min="1" max="30" value="15" />
  <span id="date-range-value">15</span>
</div>

<canvas id="metricChart" width="600" height="400"></canvas>
```

```
const ctx = document.getElementById('metricChart').getContext('2d');

// Sample data: daily records for 30 days
const fullData = Array.from({ length: 30 }, (_, i) => ({
  day: i + 1,
  sales: Math.floor(Math.random() * 100),
  profit: Math.floor(Math.random() * 50),
  expenses: Math.floor(Math.random() * 70)
}));

const metricSelect = document.getElementById('metric-select');
const dateRange = document.getElementById('date-range');
const dateRangeValue = document.getElementById('date-range-value');

let currentMetric = metricSelect.value;
let currentRange = +dateRange.value;

// Initialize Chart.js chart
const chart = new Chart(ctx, {
  type: 'bar',
  data: getChartData(currentMetric, currentRange),
  options: {
    responsive: true,
    scales: { y: { beginAtZero: true } }
  }
});

function getChartData(metric, range) {
  const slicedData = fullData.slice(-range);
  return {
    labels: slicedData.map(d => `Day ${d.day}`),
    datasets: [{
      label: metric.charAt(0).toUpperCase() + metric.slice(1),
      data: slicedData.map(d => d[metric]),
      backgroundColor: 'steelblue'
    }]
  };
}

// Update chart on metric selection
metricSelect.addEventListener('change', (e) => {
  currentMetric = e.target.value;
  updateChart();
});

// Update chart on slider input
dateRange.addEventListener('input', (e) => {
  currentRange = +e.target.value;
  dateRangeValue.textContent = currentRange;
  updateChart();
});
```

```
function updateChart() {
  const newData = getChartData(currentMetric, currentRange);
  chart.data.labels = newData.labels;
  chart.data.datasets[0].label = newData.datasets[0].label;
  chart.data.datasets[0].data = newData.datasets[0].data;
  chart.update();
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Metric Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f7f9fa;
      padding: 20px;
      text-align: center;
    }

    label {
      margin-right: 10px;
      font-weight: bold;
    }

    select, input[type="range"] {
      margin: 10px 0;
    }

    .controls {
      margin-bottom: 20px;
    }

    canvas {
      max-width: 100%;
      height: 400px;
    }
  </style>
</head>
<body>
  <h2>Dynamic Metric Bar Chart</h2>

  <div class="controls">
    <label for="metric-select">Metric:</label>
    <select id="metric-select">
      <option value="sales">Sales</option>
      <option value="profit">Profit</option>
      <option value="expenses">Expenses</option>
    </select>

    <br>

    <label for="date-range">Last N Days:</label>
```

```
  <input type="range" id="date-range" min="1" max="30" value="15" />
  <span id="date-range-value">15</span>
</div>

<canvas id="metricChart"></canvas>

<script>
  const ctx = document.getElementById('metricChart').getContext('2d');

  // Generate 30 days of sample data
  const fullData = Array.from({ length: 30 }, (_, i) => ({
    day: i + 1,
    sales: Math.floor(Math.random() * 100),
    profit: Math.floor(Math.random() * 50),
    expenses: Math.floor(Math.random() * 70)
  }));

  const metricSelect = document.getElementById('metric-select');
  const dateRange = document.getElementById('date-range');
  const dateRangeValue = document.getElementById('date-range-value');

  let currentMetric = metricSelect.value;
  let currentRange = +dateRange.value;

  function getChartData(metric, range) {
    const slicedData = fullData.slice(-range);
    return {
      labels: slicedData.map(d => `Day ${d.day}`),
      datasets: [{
        label: metric.charAt(0).toUpperCase() + metric.slice(1),
        data: slicedData.map(d => d[metric]),
        backgroundColor: 'steelblue'
      }]
    };
  }

  const chart = new Chart(ctx, {
    type: 'bar',
    data: getChartData(currentMetric, currentRange),
    options: {
      responsive: true,
      maintainAspectRatio: false,
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });

  metricSelect.addEventListener('change', (e) => {
    currentMetric = e.target.value;
    updateChart();
  });

  dateRange.addEventListener('input', (e) => {
    currentRange = +e.target.value;
    dateRangeValue.textContent = currentRange;
```

```
      updateChart();
    });

    function updateChart() {
      const newData = getChartData(currentMetric, currentRange);
      chart.data.labels = newData.labels;
      chart.data.datasets[0].label = newData.datasets[0].label;
      chart.data.datasets[0].data = newData.datasets[0].data;
      chart.update();
    }
  </script>
</body>
</html>
```

13.1.3 Explanation

- The dropdown changes which metric (`sales`, `profit`, `expenses`) the chart displays.
- The range slider controls how many days of data to show.
- Event listeners on both inputs trigger the chart update function.
- The chart updates smoothly without page reload.

13.1.4 Extending to Other Libraries

- **D3.js:** On input events, re-bind filtered data to elements, update scales, and redraw visuals.
- **ECharts or Highcharts:** Update options or series data and call the respective update methods.

13.1.5 Summary

- Use `<select>` and `<input type="range">` to create interactive controls for filtering or switching views.
- Bind input events to functions that update your visualization data and redraw charts.
- Provide immediate feedback by dynamically changing visuals based on user selections.
- This interaction greatly enhances dashboard usability and user engagement.

13.2 Two-Way Binding with State

When building interactive dashboards, managing the **state**—the current values of filters, selections, and data views—is crucial. Two-way binding ensures that changes in form controls update the state, and any state changes update the UI and visualizations in a synchronized manner.

This section introduces a simple shared state object pattern and demonstrates how to keep filters and charts in sync using vanilla JavaScript and React.

13.2.1 Concept of Shared Application State

A **state object** holds the current parameters for your dashboard, for example:

```
const state = {
  metric: 'sales',
  dateRange: 15
};
```

Both the UI controls and charts read from and update this state. Any change triggers a re-render or update of all dependent parts.

13.2.2 Two-Way Binding with Vanilla JavaScript

In vanilla JS, you manually update the state and UI elements, wiring event listeners and update functions:

```
<label for="metric-select">Metric:</label>
<select id="metric-select">
  <option value="sales">Sales</option>
  <option value="profit">Profit</option>
  <option value="expenses">Expenses</option>
</select>

<label for="date-range">Last N Days:</label>
<input type="range" id="date-range" min="1" max="30" value="15" />
<span id="date-range-value">15</span>

<canvas id="chart"></canvas>
```

```
const state = {
  metric: 'sales',
  dateRange: 15
};

const metricSelect = document.getElementById('metric-select');
const dateRange = document.getElementById('date-range');
const dateRangeValue = document.getElementById('date-range-value');
```

```
function updateUI() {
  // Reflect state in UI controls
  metricSelect.value = state.metric;
  dateRange.value = state.dateRange;
  dateRangeValue.textContent = state.dateRange;
}

function updateChart() {
  // Update chart based on current state (pseudo-code)
  console.log(`Update chart for metric: ${state.metric}, range: ${state.dateRange}`);
  // Actual chart update logic goes here...
}

// Event listeners update state and then update UI & chart
metricSelect.addEventListener('change', (e) => {
  state.metric = e.target.value;
  updateUI();
  updateChart();
});

dateRange.addEventListener('input', (e) => {
  state.dateRange = +e.target.value;
  updateUI();
  updateChart();
});

// Initialize UI and chart on page load
updateUI();
updateChart();
```

Here:

- UI changes update the `state`.
- `updateUI()` syncs controls to state (useful if state changes programmatically).
- `updateChart()` redraws the visualization based on state.

This pattern maintains **single source of truth** and consistent UI.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Shared State Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f7f9fa;
      padding: 20px;
      text-align: center;
    }

    .controls {
      margin-bottom: 20px;
    }
```

```
    label {
      margin-right: 8px;
      font-weight: bold;
    }

    select,
    input[type="range"] {
      margin: 10px 10px;
    }

    canvas {
      max-width: 100%;
      height: 400px;
      display: block;
      margin: auto;
    }
  </style>
</head>
<body>
  <h2>Shared Application State Dashboard</h2>

  <div class="controls">
    <label for="metric-select">Metric:</label>
    <select id="metric-select">
      <option value="sales">Sales</option>
      <option value="profit">Profit</option>
      <option value="expenses">Expenses</option>
    </select>

    <br>

    <label for="date-range">Last N Days:</label>
    <input type="range" id="date-range" min="1" max="30" value="15" />
    <span id="date-range-value">15</span>
  </div>

  <canvas id="chart"></canvas>

  <script>
    const state = {
      metric: 'sales',
      dateRange: 15
    };

    const metricSelect = document.getElementById('metric-select');
    const dateRange = document.getElementById('date-range');
    const dateRangeValue = document.getElementById('date-range-value');

    // Sample data generator
    const fullData = Array.from({ length: 30 }, (_, i) => ({
      day: i + 1,
      sales: Math.floor(Math.random() * 100),
      profit: Math.floor(Math.random() * 50),
      expenses: Math.floor(Math.random() * 70)
    }));

    // YES Define getChartData BEFORE it's used
    function getChartData(metric, range) {
```

```
      const sliced = fullData.slice(-range);
      return {
        labels: sliced.map(d => `Day ${d.day}`),
        datasets: [{
          label: metric.charAt(0).toUpperCase() + metric.slice(1),
          data: sliced.map(d => d[metric]),
          backgroundColor: 'steelblue'
        }]
      };
    }

    const ctx = document.getElementById('chart').getContext('2d');
    const chart = new Chart(ctx, {
      type: 'bar',
      data: getChartData(state.metric, state.dateRange),
      options: {
        responsive: true,
        maintainAspectRatio: false,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });

    function updateUI() {
      metricSelect.value = state.metric;
      dateRange.value = state.dateRange;
      dateRangeValue.textContent = state.dateRange;
    }

    function updateChart() {
      const newData = getChartData(state.metric, state.dateRange);
      chart.data.labels = newData.labels;
      chart.data.datasets[0].label = newData.datasets[0].label;
      chart.data.datasets[0].data = newData.datasets[0].data;
      chart.update();
    }

    metricSelect.addEventListener('change', (e) => {
      state.metric = e.target.value;
      updateUI();
      updateChart();
    });

    dateRange.addEventListener('input', (e) => {
      state.dateRange = +e.target.value;
      updateUI();
      updateChart();
    });

    // Initialize UI and Chart
    updateUI();
    updateChart();
  </script>
</body>
</html>
```

13.3 Live Data Updates (WebSocket or Polling)

Real-time data visualizations are increasingly important for monitoring live systems, financial markets, sensor readings, or user activity. This section guides you through implementing **live updates** in your dashboards using either **polling** or **WebSocket** connections, ensuring smooth, flicker-free UI updates.

13.3.1 Polling with `setInterval`

Polling is the simplest approach to fetch updated data at fixed intervals.

13.3.2 Basic Polling Example

```
// Simulated data source returning updated values
function fetchData() {
  return Promise.resolve({
    timestamp: new Date().toLocaleTimeString(),
    value: Math.floor(Math.random() * 100)
  });
}

// Function to update UI/chart
function updateDashboard(data) {
  console.log(`Updated at ${data.timestamp}: value = ${data.value}`);
  // Insert chart update logic here
}

// Poll every 5 seconds
setInterval(() => {
  fetchData().then(updateDashboard);
}, 5000);
```

This pattern is easy to implement but can cause redundant network requests if data doesn't change often and may introduce latency between updates.

13.3.3 WebSockets for Real-Time Push

WebSockets allow servers to push new data to clients immediately over a persistent connection, reducing delay and unnecessary requests.

13.3.4 Basic WebSocket Client Example

```
const socket = new WebSocket('wss://example.com/data');

socket.addEventListener('open', () => {
  console.log('WebSocket connection established');
});

socket.addEventListener('message', (event) => {
  const data = JSON.parse(event.data);
  console.log(`Received data:`, data);
  // Update UI/chart with new data
});

socket.addEventListener('close', () => {
  console.log('WebSocket connection closed');
});
```

13.3.5 Mini Dashboard: Live Updating Chart with Polling

Here's a simple example using **Chart.js** with polling to append live data points every second.

```
<canvas id="liveChart" width="600" height="400"></canvas>
```

```
const ctx = document.getElementById('liveChart').getContext('2d');
const maxPoints = 20;
let labels = [];
let dataPoints = [];

const liveChart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: labels,
    datasets: [{
      label: 'Live Value',
      data: dataPoints,
      borderColor: 'teal',
      fill: false,
    }]
  },
  options: {
    animation: false, // Disable animation for performance
    scales: {
      x: { display: true },
      y: { beginAtZero: true }
    }
  }
});

// Simulate fetching new data point
function fetchLiveData() {
  return Promise.resolve(Math.floor(Math.random() * 100));
}
```

```
// Update chart by appending new data
function updateLiveChart() {
  fetchLiveData().then(value => {
    const now = new Date().toLocaleTimeString();
    if (labels.length >= maxPoints) {
      labels.shift();
      dataPoints.shift();
    }
    labels.push(now);
    dataPoints.push(value);
    liveChart.update('none'); // Update without animation for smoothness
  });
}

// Poll every second
setInterval(updateLiveChart, 1000);
```

13.3.6 Best Practices for Responsive Live Updates

- **Disable or minimize animations** during frequent updates to prevent flicker and improve performance.
- **Batch updates** when possible instead of updating on every data point.
- **Limit data points** shown to a manageable number for smooth rendering.
- Use **requestAnimationFrame** for smooth rendering tied to browser refresh rate if implementing complex custom animations.
- For WebSocket, **handle connection loss** gracefully by attempting reconnect or notifying users.

13.3.7 Summary

- Use **polling** for simple periodic data fetching; it's easy to implement but may have delays.
- Use **WebSockets** for instant, push-based real-time data updates.
- Update charts by adding/removing data points, keeping datasets within size limits.
- Minimize animation and batch updates to keep UI smooth and responsive.
- Handle network or connection issues gracefully for robust dashboards.

Chapter 14.

Performance and Optimization

1. Working with Large Datasets
2. Debouncing and Throttling Input
3. Lazy Loading and Virtual Rendering
4. Canvas-Based Rendering for Speed

14 Performance and Optimization

14.1 Working with Large Datasets

14.1.1 Working with Large Datasets

Visualizing large datasets in the browser can be a rewarding but challenging task. As your data size grows—from thousands to millions of points—performance bottlenecks may arise, affecting rendering speed, interactivity, and user experience. This section explores the common challenges and introduces techniques to optimize your visualizations when working with large-scale data.

Challenges with Large Datasets in the Browser

- **Rendering Lag:** Browsers can struggle to draw thousands or millions of SVG elements or complex charts, causing slow load times and jittery interactions.
- **Memory Usage:** Large datasets consume significant memory, potentially leading to crashes or slowdowns.
- **Slow Data Processing:** Parsing, filtering, and computing with large data arrays can freeze the UI thread.
- **Poor Responsiveness:** UI events may become sluggish when visualization logic blocks the main thread.

Techniques to Improve Performance

Data Aggregation and Downsampling Reducing data volume is often the most effective way to speed up visualization.

- **Aggregation:** Summarize data into meaningful groups (e.g., average flight delay per day instead of per flight).
- **Downsampling:** Select a representative subset of data points for display, preserving overall trends without overloading the renderer.

Example: Use binning or rolling averages to reduce noisy, dense time series.

```
function rollingAverage(data, windowSize) {
  return data.map((d, i, arr) => {
    if(i < windowSize) return null;
    const window = arr.slice(i - windowSize, i);
    const avg = d3.mean(window, v => v.value);
    return { date: d.date, avgValue: avg };
  }).filter(Boolean);
}
```

Offload Work with `requestIdleCallback()` `requestIdleCallback()` allows deferring non-urgent processing until the browser is idle, preventing UI jank.

```
requestIdleCallback(() => {
  // Expensive data processing or rendering here
  processLargeDataset(data);
});
```

This keeps the interface responsive by splitting heavy work into chunks during idle time.

Demonstration: Visualizing Flight Delays Dataset

Suppose we have a CSV file with hundreds of thousands of flight delay records, including flight times and delay durations.

```
d3.csv('flight_delays_large.csv').then(data => {
  svg.selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', d => xScale(new Date(d.departure_time)))
    .attr('cy', d => yScale(+d.delay_minutes))
    .attr('r', 1)
    .attr('fill', 'steelblue');
});
```

Baseline: Rendering All Raw Points with D3.js Performance Issues:

- Rendering so many circles causes slow initial load and lag on interactions like zoom or tooltip.

Optimized: Aggregation and Deferred Processing

1. **Aggregate delays by hour or day:**

```
const aggregated = d3.rollup(
  data,
  v => d3.mean(v, d => +d.delay_minutes),
  d => d3.timeHour.floor(new Date(d.departure_time))
);
const aggregatedArray = Array.from(aggregated, ([date, avgDelay]) => ({ date, avgDelay }));
```

2. **Render aggregated data with larger circles or line chart:**

```
svg.selectAll('circle')
  .data(aggregatedArray)
  .join('circle')
  .attr('cx', d => xScale(d.date))
  .attr('cy', d => yScale(d.avgDelay))
  .attr('r', 3)
  .attr('fill', 'orange');
```

3. **Use `requestIdleCallback()` to defer heavy parsing:**

```
requestIdleCallback(() => {
  parseAndAggregateLargeCSV(rawCSVData);
});
```

Before and After Performance

Aspect	Before	After
Initial Load	Several seconds lag	Instant or under 1 second
Rendering	Hundreds of thousands of points	Aggregated hundreds/thousands
UI Responsiveness	Slow during zoom/pan	Smooth and responsive
Memory Usage	High	Significantly reduced

Summary

- Large datasets challenge browser rendering and responsiveness.
- Aggregation and downsampling reduce data volume while preserving insights.
- Deferring expensive work with `requestIdleCallback()` prevents UI blocking.
- Combining these techniques yields fast, smooth visualizations even with massive datasets.

By applying these strategies, your charts will stay performant, ensuring users get rich insights without frustration.

Next up: learn how to manage user input efficiently with **Debouncing and Throttling** for an even smoother experience!

14.2 Debouncing and Throttling Input

14.2.1 Debouncing and Throttling Input

When building interactive data visualizations, user input can trigger frequent updates—especially when users scroll, resize, drag sliders, or type rapidly. Without control, these frequent events can lead to excessive rendering, sluggish performance, and redundant API calls.

In this section, you'll learn how to apply **debouncing** and **throttling** to optimize interactivity, preserve responsiveness, and reduce computational overhead.

14.2.2 Whats the Difference?

Technique	Behavior	When to Use
Debouncing	Waits until the user stops triggering the event, then runs the function once.	For inputs where only the final action matters (e.g., search boxes, sliders).
Throttling	Ensures the function runs at most once every X milliseconds.	For events that need periodic updates (e.g., window resize, scroll).

14.2.3 Why It Matters in Visualization

Without debouncing or throttling:

- Dragging a time slider may trigger hundreds of re-renders per second.
- Typing in a search box can fire an API call with every keystroke.
- Scrolling a zoomable map could call the redraw function too frequently.

With proper control, we:

- Reduce unnecessary work
- Maintain smooth UI performance
- Avoid overloading APIs or the DOM

14.2.4 Debouncing in Practice

Debouncing waits for a pause in activity before executing a function. Here's how to implement it in vanilla JavaScript:

Native JavaScript Debounce

```
function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Usage: delay filter update until typing stops
const searchInput = document.getElementById('search');
searchInput.addEventListener('input', debounce(handleSearch, 300));

function handleSearch(event) {
  const query = event.target.value;
  updateChartWithQuery(query);
}
```

Using Lodashs `_.debounce`

```
import _ from 'lodash';

const handleFilterChange = _.debounce((value) => {
  updateChartWithFilter(value);
}, 300);

document.getElementById('filter').addEventListener('input', (e) => {
  handleFilterChange(e.target.value);
});
```

14.2.5 Throttling in Practice

Throttling limits how often a function can execute—even if the triggering event happens constantly.

Native JavaScript Throttle

```
function throttle(fn, interval) {
  let lastTime = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastTime >= interval) {
      lastTime = now;
      fn.apply(this, args);
    }
  };
}

// Usage: limit redraw on scroll
window.addEventListener('scroll', throttle(() => {
  updateVisibleDataWindow();
}, 200));
```

Using Lodashs `_.throttle`

```
import _ from 'lodash';

const throttledResize = _.throttle(() => {
  redrawResponsiveChart();
}, 250);

window.addEventListener('resize', throttledResize);
```

14.2.6 Real-World Use Case: Throttled Map Zoom

```
const zoomHandler = _.throttle((event) => {
  const zoomLevel = event.transform.k;
  updateMapZoom(zoomLevel);
}, 100);

// Apply zoom behavior using D3
d3.select('svg')
  .call(d3.zoom().on('zoom', zoomHandler));
```

This ensures the zoomed map updates smoothly without triggering layout recalculations at every pixel-level change.

14.2.7 Performance Comparison

Interaction	Without Debounce/Throttle	With Optimization
Text input	API call every keystroke	Call once after typing pauses
Scroll	Dozens of redraws/sec	One update every 100–200 ms
Resize	UI stutter, lag	Smooth resize and redraw

14.2.8 Summary

- **Debouncing** delays action until input stops—great for sliders and search fields.
- **Throttling** limits how often a function runs—ideal for scroll, resize, and zoom.
- Both techniques are essential for responsive, high-performance visualizations.
- Implement easily using native JavaScript or utilities like **Lodash**.

Applied wisely, these tools keep your dashboards fast, fluid, and user-friendly—no matter how interactive they get.

Next: Learn how **lazy loading and virtual rendering** can help you handle even more data without performance sacrifice.

14.3 Lazy Loading and Virtual Rendering

14.3.1 Lazy Loading and Virtual Rendering

When visualizing large datasets—think thousands of rows in a table or thousands of bars or points in a chart—rendering everything at once in the DOM can overwhelm the browser. This leads to slow load times, sluggish scrolling, and unresponsive interfaces.

The solution? **Lazy loading** and **virtual rendering**. These techniques ensure that only what's visible to the user is rendered at any given moment, dramatically improving performance.

14.3.2 Why Lazy Load or Virtualize?

Rendering 10,000+ elements in the browser with SVG, DOM, or HTML is rarely necessary. Most users only interact with a small portion of what's visible on screen. Virtualization ensures the browser only creates the elements it needs to display.

Technique	Purpose
Lazy Loading	Load/render items only as they scroll into view
Virtual Rendering	Only mount visible elements in the DOM (replace others as needed)

14.3.3 Real-World Example: Virtualized List of Data Points

Let's say you're displaying a long list of company financials or climate records. Instead of rendering 10,000 rows at once, you can render only what's visible.

14.3.4 Option 1: Custom Virtual Scroller in Vanilla JavaScript

HTML structure:

```
<div id="viewport" style="height: 400px; overflow-y: scroll; position: relative;">
  <div id="content" style="position: absolute; width: 100%;"></div>
</div>
```

JavaScript logic:

```
const viewport = document.getElementById('viewport');
const content = document.getElementById('content');

const rowHeight = 30;
const totalItems = 10000;
const visibleRows = 15;
const buffer = 5;

const data = Array.from({ length: totalItems }, (_, i) => `Row ${i + 1}`);

content.style.height = `${totalItems * rowHeight}px`;

function renderRows(scrollTop) {
  const start = Math.max(0, Math.floor(scrollTop / rowHeight) - buffer);
  const end = Math.min(totalItems, start + visibleRows + buffer * 2);

  content.innerHTML = '';
  for (let i = start; i < end; i++) {
    const div = document.createElement('div');
    div.textContent = data[i];
    div.style.position = 'absolute';
    div.style.top = `${i * rowHeight}px`;
    div.style.height = `${rowHeight}px`;
    div.style.width = '100%';
    content.appendChild(div);
  }
}
```

```
viewport.addEventListener('scroll', () => {
  renderRows(viewport.scrollTop);
});

// Initial render
renderRows(0);
```

YES This gives you a blazing-fast scrollable list of 10,000 rows, with only ~25 rows ever in the DOM at once.

14.3.5 Option 2: Using `react-window` for React Apps

For React developers, react-window offers high-performance virtual lists and grids out of the box.

14.3.6 Example with `FixedSizeList`:

```
import { FixedSizeList as List } from 'react-window';

const Row = ({ index, style }) => (
  <div style={style}>Row #{index}</div>
);

<List
  height={400}
  itemCount={10000}
  itemSize={30}
  width={'100%'}
>
  {Row}
</List>
```

YES Only the visible items are rendered and React efficiently reuses DOM elements as you scroll.

14.3.7 Progressive Rendering with Charts

With D3 or other libraries, progressive rendering means:

- Drawing parts of the chart incrementally
- Using **Intersection Observer** to defer rendering until a section scrolls into view

14.3.8 Using IntersectionObserver for Lazy Chart Loading

```
const observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      renderChart(entry.target.dataset.chartId);
      observer.unobserve(entry.target);
    }
  });
}, { threshold: 0.1 });

document.querySelectorAll('.lazy-chart').forEach(el => observer.observe(el));
```

HTML Example:

```
<div class="lazy-chart" data-chart-id="region1" style="height: 400px;"></div>
```

This is especially helpful for dashboards with many charts — you defer rendering until the user actually scrolls to each one.

14.3.9 Lazy Line/Bar Chart Rendering with D3

If you're rendering a very long time-series:

```
function drawVisibleSlice(start, end) {
  const visibleData = fullDataset.slice(start, end);
  d3.select('svg').selectAll('line')
    .data(visibleData)
    .join('line') // Or any element
    .attr('x1', d => xScale(d.time))
    .attr('y1', d => yScale(d.value))
    .attr('x2', ...)
    .attr('y2', ...);
}
```

Combine this with a **scroll listener** or a **slider** to fetch only the visible portion dynamically.

14.3.10 Summary

- **Virtual rendering** and **lazy loading** reduce rendering overhead by only displaying what's visible.
- Use custom scroll logic or libraries like `react-window` to optimize long lists or data tables.
- Use **IntersectionObserver** to defer chart rendering until visible.
- In D3, draw slices of data incrementally or on-demand to avoid freezing the browser.

These techniques are critical for building **smooth, scalable dashboards**—especially as

your data grows into the tens or hundreds of thousands of elements.

Next: Take your rendering performance even further with **Canvas-based rendering** for lightning-fast data visualization.

14.4 Canvas-Based Rendering for Speed

14.4.1 Canvas-Based Rendering for Speed

When building interactive data visualizations, developers often choose between two main rendering technologies: **SVG (Scalable Vector Graphics)** and the `<canvas>` API. Both have strengths, but when performance is paramount—especially with **thousands or tens of thousands of data points**—Canvas typically wins.

When to Use `canvas` Over SVG

SVG is *declarative*, meaning each element (like a `circle` or `line`) is part of the DOM and can be individually styled, interacted with, or animated. This makes SVG ideal for small to medium-sized datasets and charts with a rich interactive layer.

However, when you're working with **large datasets (10,000+ elements)**, SVG performance degrades. Each SVG element adds to the DOM, and browsers must maintain and repaint all of them. This introduces significant overhead.

Canvas, by contrast, is *imperative*. You draw pixels directly onto a 2D bitmap using JavaScript. There's no DOM representation of each point—just raw pixel manipulation. This makes Canvas extremely efficient for rendering **dense plots, heatmaps, and real-time graphics**.

SVG vs Canvas: Performance Showdown

Let's consider a benchmark: rendering **10,000 points** in a scatter plot.

Metric	SVG	Canvas
DOM nodes created	10,000 `<circle>` elements	0
Time to render (approximate)	1,500ms – 3,000ms	20ms – 50ms
Browser memory footprint	High	Low
Interactivity (per-point)	Easy (native DOM events)	Requires manual hit testing
Ideal for	Small datasets, accessibility	Large datasets, performance

The difference is dramatic. While SVG struggles with DOM overload, Canvas easily handles tens of thousands of pixels at 60 FPS.

Example: Scatter Plot with Canvas D3

Though D3 is often associated with SVG, it works equally well to **prepare and scale data**, regardless of the rendering method.

Let's walk through a minimal example: rendering a scatter plot of 10,000 random points using Canvas and D3.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Canvas Scatter Plot with D3</title>
  <style>
    canvas {
      border: 1px solid #ccc;
      display: block;
      margin: 20px auto;
    }
  </style>
</head>
<body>
<canvas id="scatter" width="800" height="500"></canvas>

<script src="https://d3js.org/d3.v7.min.js"></script>
<script>
  const canvas = document.getElementById('scatter');
  const ctx = canvas.getContext('2d');

  const width = canvas.width;
  const height = canvas.height;

  // Generate 10,000 random points
  const data = d3.range(10000).map(() => ({
    x: Math.random() * 100,
    y: Math.random() * 100
  }));

  // Scales
  const xScale = d3.scaleLinear().domain([0, 100]).range([40, width - 20]);
  const yScale = d3.scaleLinear().domain([0, 100]).range([height - 20, 20]);

  // Clear and draw
  ctx.clearRect(0, 0, width, height);
  ctx.fillStyle = 'steelblue';

  data.forEach(d => {
    ctx.beginPath();
    ctx.arc(xScale(d.x), yScale(d.y), 2, 0, 2 * Math.PI);
    ctx.fill();
  });
</script>
</body>
</html>
```

Key Points:

- **D3** handles the data generation and scaling—just like it would for SVG.

- **Canvas** renders all 10,000 points in milliseconds with low memory usage.
- Each point is drawn manually with `ctx.arc()` and `ctx.fill()`.

Heatmaps: A Natural Fit for Canvas

Heatmaps are another great use case for Canvas. Since heatmaps are essentially a **matrix of colored pixels**, Canvas's bitmap-based rendering shines. Unlike SVG, you don't need to manage thousands of `rect` elements—just set the color of each pixel or grid cell and draw.

For performance-intensive tasks like **real-time sensor readings, geographic data density maps, or simulation outputs**, Canvas ensures smooth rendering and fast updates.

When to Stick with SVG

Canvas isn't a silver bullet. SVG is still the best choice when:

- You need **semantic structure** for accessibility or styling.
- You want **tooltips, labels, and interactivity** per element.
- The dataset is small to moderate (<1,000 points).

Conclusion

If your visualization is choking on large datasets, **Canvas is the upgrade you need**. You lose a bit of element-level interactivity but gain massive speed boosts. For hybrid solutions, some developers even use **Canvas for the heavy rendering** and **SVG for interactive overlays**.

Chapter 15.

Exporting and Sharing

1. Exporting to PNG, SVG, or PDF
2. Sharing via Embeds or Static Pages
3. Hosting Options (GitHub Pages, Netlify)

15 Exporting and Sharing

15.1 Exporting to PNG, SVG, or PDF

15.1.1 Exporting to PNG, SVG, or PDF

Creating beautiful data visualizations is only part of the journey—the next step is sharing them. Whether you want to include charts in reports, presentations, or send them to stakeholders, **exporting visualizations to image or document formats like PNG, SVG, or PDF** is a crucial skill.

This section covers practical strategies for exporting your charts using:

- Native Canvas methods like `toDataURL()`
- Utilities such as `html2canvas` and `d3-save-svg`
- PDF generation tools like `jsPDF`

15.1.2 Exporting Canvas Charts as PNG

If your chart is rendered using the `<canvas>` element, exporting to a PNG is fast and straightforward using the native `canvas.toDataURL()` method.

Example: Save a Canvas Chart as PNG

```
<canvas id="chartCanvas" width="600" height="400"></canvas>
<button id="downloadBtn">Download PNG</button>

<script>
  const canvas = document.getElementById('chartCanvas');
  const ctx = canvas.getContext('2d');

  // Draw a simple chart
  ctx.fillStyle = 'steelblue';
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.arc(Math.random() * 600, Math.random() * 400, 3, 0, 2 * Math.PI);
    ctx.fill();
  }

  // Export to PNG on button click
  document.getElementById('downloadBtn').addEventListener('click', () => {
    const url = canvas.toDataURL('image/png');
    const link = document.createElement('a');
    link.href = url;
    link.download = 'chart.png';
    link.click();
  });
</script>
```

This approach is efficient for any chart drawn with Canvas, including high-performance

scatter plots and heatmaps.

15.1.3 Exporting D3 SVG Charts as SVG or PNG

If you're using D3 to create charts with SVG (e.g., `d3.select("svg")`), you can export the raw SVG or convert it into a downloadable PNG using utilities.

a) Exporting SVG with `d3-save-svg`

`d3-save-svg` is a lightweight helper that serializes and downloads D3-generated SVG charts.

```
<svg id="chartSVG" width="600" height="400"></svg>
<button id="saveSvgBtn">Download SVG</button>

<script src="https://d3js.org/d3.v7.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/d3-save-svg@0.1.0/dist/d3-save-svg.min.js"></script>
<script>
  const svg = d3.select("#chartSVG");

  // Draw something
  svg.selectAll("circle")
    .data(d3.range(100))
    .enter()
    .append("circle")
    .attr("cx", () => Math.random() * 600)
    .attr("cy", () => Math.random() * 400)
    .attr("r", 3)
    .attr("fill", "steelblue");

  // Save SVG on click
  document.getElementById('saveSvgBtn').addEventListener('click', () => {
    d3_save_svg.save(d3.select("#chartSVG").node(), {
      filename: 'chart.svg'
    });
  });
</script>
```

Example: Download SVG This method preserves styling and structure, making it ideal for vector exports.

b) Convert SVG to PNG with `html2canvas`

To turn an SVG chart into a PNG—even if it's styled with external CSS—you can use `html2canvas`. It captures DOM content (including SVG) and renders it to a canvas, which you can then export.

```
<button id="savePngBtn">Save as PNG</button>

<script src="https://cdn.jsdelivr.net/npm/html2canvas@1.4.1/dist/html2canvas.min.js"></script>
<script>
```

```
  document.getElementById('savePngBtn').addEventListener('click', () => {
    const chart = document.getElementById('chartSVG');
    html2canvas(chart).then(canvas => {
      const url = canvas.toDataURL('image/png');
      const link = document.createElement('a');
      link.href = url;
      link.download = 'chart.png';
      link.click();
    });
  });
</script>
```

Example: Convert D3 SVG to PNG

> **Note**: `html2canvas` may not render external fonts or styles perfectly. For complex SVGs, `d3-save-svg` or direct SVG manipulation is usually more accurate.

15.1.4 Exporting Charts as PDF

For generating **PDFs that include charts**, the most popular client-side tool is `jsPDF`. It supports embedding images, text, and vector content.

You can convert a Canvas or an SVG chart into an image and insert it into a PDF.

Example: Convert a Canvas Chart to PDF

```
<button id="savePdfBtn">Save as PDF</button>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js"></script>
<script>
  document.getElementById('savePdfBtn').addEventListener('click', async () => {
    const { jsPDF } = window.jspdf;
    const canvas = document.getElementById('chartCanvas');
    const imgData = canvas.toDataURL('image/png');

    const pdf = new jsPDF();
    pdf.addImage(imgData, 'PNG', 15, 40, 180, 100);
    pdf.save('chart.pdf');
  });
</script>
```

You can also export SVGs via `html2canvas` + `jsPDF`:

1. Use `html2canvas` to convert an SVG to a Canvas.
2. Extract the PNG and add it to a PDF with `jsPDF`.

This layered approach works well when you need a high-quality, printable document that includes your visualizations.

15.1.5 Summary

Format	Best For	How
PNG	Fast sharing, presentations	`canvas.toDataURL()`, `html2canvas`
SVG	Print-ready vector graphics	`d3-save-svg`, manual serialization
PDF	Reports, downloadable documents	`jsPDF` + image data

Being able to **export your visualizations in multiple formats** is an essential part of making your work portable and professional. Whether your audience wants screenshots, high-res print graphics, or embedded documents, these tools make exporting seamless.

Next, we'll explore how to share your charts on the web—either as embeddable widgets or standalone static pages.

15.2 Sharing via Embeds or Static Pages

15.2.1 Sharing via Embeds or Static Pages

After creating an engaging chart, your next step is to **share it with the world**. This could mean embedding your chart on a blog, integrating it into a dashboard, or distributing it as a standalone HTML file. Fortunately, JavaScript visualizations are easy to package and share thanks to the web-native technologies they're built on.

In this section, you'll learn how to:

- Embed charts using `<iframe>`, `<script>`, or JavaScript bundles
- Export standalone HTML pages
- Share charts using Webpack, Vite, or GitHub Pages

15.2.2 Embedding Charts in Blogs and Dashboards

There are several ways to embed a visualization into a webpage, depending on your needs and environment.

a) Using an `iframe`

An `<iframe>` lets you encapsulate your chart inside a separate HTML page and embed it anywhere—like a blog post, CMS, or even another web app.

```
<iframe
  src="https://yourdomain.com/chart.html"
```

```
  width="700"
  height="500"
  style="border: none;">
</iframe>
```

Example: To use this approach, you'll need to host `chart.html` somewhere accessible (see GitHub Pages below).

b) Embedding with a `script` Tag

If your chart is packaged as a JavaScript bundle, you can expose it as a function and allow users to call it from a script tag.

```
<!-- Your HTML page -->
<div id="chart"></div>
<script src="https://yourdomain.com/chart.bundle.js"></script>
<script>
  renderChart(document.getElementById('chart'));
</script>
```

Example (hosted bundle): In this setup:

- `chart.bundle.js` exposes a function like `renderChart(container)`
- Users only need to include the script and call your function with a target container

This is a great option for dashboards or apps where you want fine-grained control.

15.2.3 Exporting Standalone HTML Pages

Sometimes you want to **share a single, portable HTML file** containing everything needed to view your chart. This is useful for email attachments, downloads, or static hosting.

Here's a simple D3 example wrapped in one HTML file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Standalone Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
  <svg width="600" height="400"></svg>
  <script>
    const svg = d3.select("svg");
    svg.selectAll("circle")
      .data(d3.range(100))
      .enter()
      .append("circle")
      .attr("cx", () => Math.random() * 600)
      .attr("cy", () => Math.random() * 400)
```

```
      .attr("r", 4)
      .attr("fill", "tomato");
  </script>
</body>
</html>
```

Save this as `chart.html`, and it's ready to open in any browser or embed in an `<iframe>`.

15.2.4 Hosting with Webpack or Vite

For more complex projects (with modules, styling, or multiple charts), you'll want a build tool. Popular choices are:

- **Webpack**: Customizable, widely used
- **Vite**: Fast, modern, ES module–first

Example: Vite Setup for a Shared Chart

1. **Initialize a project**:
   ```
   npm create vite@latest chart-project -- --template vanilla
   cd chart-project
   npm install
   ```

2. **Add D3**:
   ```
   npm install d3
   ```

3. **Edit `main.js`**:
   ```
   import * as d3 from 'd3';

   const svg = d3.select("svg");
   svg.selectAll("circle")
     .data(d3.range(200))
     .enter()
     .append("circle")
     .attr("cx", () => Math.random() * 600)
     .attr("cy", () => Math.random() * 400)
     .attr("r", 3)
     .attr("fill", "steelblue");
   ```

4. **Build and export**:
   ```
   npm run build
   ```

   This creates a `dist/` folder with static files you can share or deploy.

15.2.5 Hosting on GitHub Pages with `gh-pages`

GitHub Pages offers a free and easy way to host static sites directly from your repositories.

Step-by-Step: Host a Chart from GitHub

1. **Create a GitHub repository** and add your `chart.html` or `dist/` folder.
2. **Install the `gh-pages` CLI** (if using Node):
   ```
   npm install --save-dev gh-pages
   ```
3. **Add a deploy script** in `package.json`:
   ```
   {
     "scripts": {
       "build": "vite build",
       "deploy": "gh-pages -d dist"
     }
   }
   ```
4. **Deploy it**:
   ```
   npm run deploy
   ```
5. **Visit your chart** at:

   `https://<your-username>.github.io/<repo-name>/`

You can now embed this URL in an `<iframe>`, share it, or include it in presentations.

15.2.6 Summary

Method	Use Case	Hosting Required?
`<iframe>`	Embed in blogs, CMS	YES Yes
`<script>` tag	Embed in web apps or dashboards	YES Yes
Standalone HTML	Share offline or via download/email	NO No
Vite/Webpack	Build modular, scalable visualizations	YES Yes
GitHub Pages	Free, simple hosting for static content	YES Yes

Whether you're building a simple chart for a blog or a dynamic dashboard component, sharing your work is straightforward with the right tools. In the next section, we'll dive deeper into hosting options like **GitHub Pages**, **Netlify**, and more for seamless deployment.

15.3 Hosting Options (GitHub Pages, Netlify)

15.3.1 Hosting Options (GitHub Pages, Netlify)

Once your data visualization is ready to share, the final step is putting it online so others can access it. Static hosting services like **GitHub Pages** and **Netlify** make this process simple, free, and fast. Whether you're publishing a single HTML chart or an entire visualization dashboard built with Webpack or Vite, these platforms offer flexible deployment options.

In this section, we'll compare GitHub Pages and Netlify, walk through deploying a chart project with both, and explore automation tools and CI/CD workflows for teams.

15.3.2 Static Hosting Comparison

Feature	**GitHub Pages**	**Netlify**
Hosting type	Static site	Static site + build environment
Free tier	YES Yes	YES Yes
Custom domains	YES Yes	YES Yes
HTTPS/SSL	YES Auto	YES Auto
CI/CD Integration	GitHub Actions or manual	Built-in with Git integration
Deployment methods	CLI, GitHub Actions	Drag & drop, Git push, CLI
Build support	Limited (manual build)	Full build pipeline (e.g. Vite)
Best for	Simple charts, personal sites	Modern workflows, team deployments

15.3.3 Option 1: Deploy with GitHub Pages

GitHub Pages is ideal for **simple static sites** like standalone D3 charts or Canvas visualizations.

15.3.4 Method A: Use the `gh-pages` CLI (Recommended for Projects with Build Steps)

1. **Install `gh-pages`**:
   ```
   npm install --save-dev gh-pages
   ```

2. **Add deployment script in `package.json`**:

```
{
  "scripts": {
    "build": "vite build",
    "deploy": "gh-pages -d dist"
  }
}
```

3. **Deploy your project**:

```
npm run build
npm run deploy
```

4. **View your site** at:

   `https://<username>.github.io/<repo-name>/`

This approach works perfectly for Vite/Webpack projects, as long as you build to a `dist/` folder.

15.3.5 Method B: Use GitHub Actions for CI/CD Deployment

To automate deployment on every push to `main` or `master`:

1. **Create a workflow file** at `.github/workflows/deploy.yml`:

```
name: Deploy to GitHub Pages

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18

      - run: npm install
      - run: npm run build

      - uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./dist
```

2. **Push changes to GitHub**—GitHub Actions will build and deploy your site automatically.

This is a **CI/CD-friendly** method great for teams or frequently updated projects.

15.3.6 Option 2: Deploy with Netlify

Netlify is a modern hosting platform with built-in CI/CD, HTTPS, and domain management. It's especially well-suited for **projects using build tools like Vite, React, or Svelte**.

15.3.7 Method A: Drag and Drop

1. **Build your project**:
   ```
   npm run build
   ```
2. **Go to netlify.com** and log in.
3. **Click "Add new site" → "Deploy manually"**.
4. **Drag your `dist/` folder into the upload area**.
5. Done! Netlify gives you a live preview link like:

   `https://your-site-name.netlify.app`

15.3.8 Method B: Connect GitHub Repo (Git Push Deploy)

1. **Push your visualization project to GitHub**.
2. **In Netlify**, choose "Add new site from Git".
3. **Authorize GitHub**, select your repository.
4. **Configure build settings**:
   - **Build Command**: `npm run build`
   - **Publish Directory**: `dist`
5. **Click "Deploy Site"**.

Now every time you push to GitHub, Netlify will rebuild and redeploy your site automatically.

15.3.9 Bonus: Automating with CI/CD for Teams

For teams maintaining multiple charts or data dashboards, automation ensures consistency and reliability. Both GitHub Pages and Netlify support:

- **Pull request previews**: Share temporary links for PRs.
- **Branch-specific deploys**: Deploy staging and production environments.

- **Secrets management**: Store API keys for fetching live data.
- **Build plugins**: Run additional tasks like linting, testing, or bundling assets.

Popular tools that integrate well:

- **GitHub Actions** (flexible for any workflow)
- **Netlify CI/CD** (zero config for most frontend projects)
- **Vercel** (alternative to Netlify, especially for Next.js apps)

15.3.10 Summary

Deployment Method	Best For	Build Required?	Automation Support
GitHub Pages (CLI)	Static HTML or Vite builds	YES Yes	NO Manual
GitHub Pages (Actions)	Teams, CI/CD, frequent updates	YES Yes	YES Yes
Netlify (Drag & Drop)	Quick share of standalone charts	YES Yes	NO Manual
Netlify (Git push)	Full CI/CD with modern toolchains	YES Yes	YES Yes

Whether you're an individual developer sharing a single chart or part of a team maintaining a suite of dashboards, GitHub Pages and Netlify provide solid, free options to **publish, share, and maintain visualizations online**.

In the next chapter, we'll look at **interactive storytelling techniques** to make your data narratives more dynamic and user-driven.

Chapter 16.

Appendices

1. Color Palettes for Data Viz
2. Accessibility Guidelines

16 Appendices

16.1 Color Palettes for Data Viz

16.1.1 Color Palettes for Data Viz

Color is one of the most powerful tools in a data visualizer's toolkit—but also one of the easiest to misuse. The right color palette can highlight trends, clarify differences, and make your chart beautiful. The wrong one can mislead, confuse, or exclude viewers.

This section introduces **commonly used, accessible color palettes** and how to apply them effectively in different JavaScript visualization libraries including **D3**, **Chart.js**, and **ECharts**. We'll cover:

- Categorical vs Sequential palettes
- Accessibility and color blindness
- Popular palettes: ColorBrewer, Tableau, D3 built-in scales
- Integration examples with D3, Chart.js, and ECharts

16.1.2 Types of Color Palettes

Before choosing colors, understand the **data type** you're visualizing.

Categorical (Qualitative)

Used for distinct, unordered categories (e.g., product types, regions).

> YES Use **distinct colors** that are easily distinguishable.

Examples:

- `["#1f77b4", "#ff7f0e", "#2ca02c", ...]`
- Tableau 10
- D3's `schemeCategory10`

Sequential

Used for ordered data with a clear progression (e.g., temperature, population).

> YES Use **light to dark gradients** or **single-hue ramps**.

Examples:

- ColorBrewer `Blues`, `Greens`, `Oranges`
- D3's `interpolateViridis`, `interpolateBlues`

Diverging

Used to show deviation from a midpoint (e.g., positive vs negative growth).

> YES Use **two contrasting colors** that meet at a neutral center.

Examples:

- ColorBrewer `RdBu`, `PiYG`
- D3's `interpolateRdBu`

16.1.3 Color Perception & Accessibility

Not everyone sees color the same way. Around **1 in 12 men** and **1 in 200 women** have some form of color vision deficiency.

Tips for Accessibility:

- Avoid red-green combinations.
- Use textures or patterns when possible.
- Test palettes with Coblis (Color Blindness Simulator)
- Prefer **ColorBrewer's `*Colorblind Safe*`** palettes.
- Provide **legends and labels**—never rely on color alone.

16.1.4 Popular Palettes

ColorBrewer

- Created for cartography, but widely used in all types of visualizations.
- Available via D3: `d3.schemeBlues`, `d3.schemeSet1`, etc.

Tableau Palettes

- Designed for clarity, elegance, and color-blind accessibility.
- Tableau 10 is one of the most widely adopted categorical palettes.

```
// Tableau 10
const tableau10 = [
  "#4e79a7", "#f28e2b", "#e15759", "#76b7b2", "#59a14f",
  "#edc949", "#af7aa1", "#ff9da7", "#9c755f", "#bab0ab"
];
```

D3 Color Scales

- Built-in palettes for categorical and continuous data.
- Supports interpolation and color ramps.

Examples:

```
d3.schemeCategory10          // categorical (10 colors)
d3.interpolateViridis        // continuous gradient
d3.interpolateRdBu           // diverging
```

16.1.5 Integration Examples

D3.js: Applying Color Scales

```
const color = d3.scaleOrdinal(d3.schemeCategory10);

d3.selectAll("circle")
  .data(data)
  .attr("fill", d => color(d.category));
```

For sequential data:

```
const color = d3.scaleSequential(d3.interpolateBlues)
  .domain([0, 100]);

d3.selectAll("rect")
  .data(data)
  .attr("fill", d => color(d.value));
```

Chart.js: Custom Color Palettes

Chart.js supports custom colors through `backgroundColor` and `borderColor` arrays.

```
const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['A', 'B', 'C'],
    datasets: [{
      label: 'Values',
      data: [10, 20, 30],
      backgroundColor: [
        '#4e79a7', '#f28e2b', '#e15759' // Tableau colors
      ]
    }]
  }
});
```

For sequential data (e.g., heatmaps or gradient charts), you can use utility libraries to generate gradients dynamically or manually interpolate.

ECharts: Applying Color Palettes

ECharts allows palette customization using the `color` property at the global or series level.

```
const option = {
  color: ['#4e79a7', '#f28e2b', '#e15759'], // Tableau 10
  series: [{
    type: 'pie',
```

```
    data: [
      { value: 40, name: 'Apples' },
      { value: 25, name: 'Bananas' },
      { value: 35, name: 'Cherries' }
    ]
  }]
};
```

For continuous data:

```
visualMap: {
  min: 0,
  max: 100,
  inRange: {
    color: ['#f7fbff', '#08306b'] // Sequential blue ramp
  }
}
```

16.1.6 Summary

Palette Type	Use Case	Examples
Categorical	Categories, labels	`schemeCategory10`, Tableau
Sequential	Quantitative values	`interpolateBlues`, `Viridis`
Diverging	Deviations from a baseline	`interpolateRdBu`, `PiYG`

Library	Color Integration Approach
D3	`scaleOrdinal()`, `scaleSequential()`
Chart.js	`backgroundColor` array in dataset config
ECharts	`color` array or `visualMap.inRange`

By choosing palettes intentionally—and testing for accessibility—you can make your visualizations not only more appealing, but also more accurate and inclusive. In the next section, we'll explore accessibility guidelines to ensure your charts are readable and usable by everyone.

16.2 Accessibility Guidelines

16.2.1 Accessibility Guidelines

Creating effective data visualizations isn't just about aesthetics—it's about **inclusion**. Accessible visualizations ensure that **everyone**, including users with disabilities, can understand and interact with your charts.

This section outlines **core accessibility principles** for charts and shows how to implement them using HTML, SVG, and ARIA techniques. Whether you're building charts with D3, Chart.js, or ECharts, these guidelines will help you make your work usable for a broader audience.

16.2.2 Key Accessibility Considerations

Concern	Why It Matters
Color contrast	Helps users with low vision or color blindness
ARIA labels	Supports screen readers in understanding the chart
Keyboard navigation	Allows interaction without a mouse
Screen reader support	Makes charts understandable in non-visual contexts

Let's explore each in detail, along with actionable techniques.

16.2.3 Ensure High Color Contrast

Color should not be the only way to convey information.

- Use **high-contrast color pairs** (e.g., dark text on light backgrounds).
- Test with WebAIM's Contrast Checker.
- Avoid relying solely on red-green or blue-purple distinctions.
- Pair colors with **shapes, patterns, or labels**.

Example: Line Chart with Color Shape

```
d3.selectAll("path")
  .attr("stroke", (d, i) => color(i))
  .attr("stroke-dasharray", (d, i) => i % 2 === 0 ? "5,5" : "0");
```

16.2.4 Add ARIA Labels and Descriptions

For charts rendered in **SVG or Canvas**, screen readers can't interpret visuals directly. You need to provide **semantic cues** via ARIA attributes and supporting HTML.

For SVG Charts

```
<svg role="img" aria-labelledby="chart-title chart-desc">
  <title id="chart-title">Sales Over Time</title>
```

```
  <desc id="chart-desc">A line chart showing sales trends from 2020 to 2024.</desc>
  <!-- Your SVG elements here -->
</svg>
```

- `role="img"` declares the SVG as an image
- `aria-labelledby` ties the chart to its title and description

For Canvas Charts

Canvas has no built-in accessibility, so use alternative elements to describe it:

```
<canvas id="myCanvas" aria-describedby="canvas-desc" role="img"></canvas>
<p id="canvas-desc">
  A bar chart comparing revenue across product categories in 2024.
</p>
```

Consider offering an accessible **text summary** or **data table** below the chart.

16.2.5 Enable Keyboard Navigation

Many visualizations use zoom, pan, or tooltip interactions that rely on a mouse. To support **keyboard users**, follow these guidelines:

- Make interactive containers **focusable**:

```
<div tabindex="0" id="chart-container"> ... </div>
```

- Handle keyboard events like arrow keys or tab:

```
document.getElementById("chart-container").addEventListener("keydown", (e) => {
  if (e.key === "ArrowRight") zoomIn();
  if (e.key === "ArrowLeft") zoomOut();
});
```

- Provide **visible focus indicators** with CSS:

```
#chart-container:focus {
  outline: 2px solid #007acc;
}
```

16.2.6 Support Screen Readers

Screen readers can't "see" a chart, but they can read:

- Titles and descriptions
- Summary text or table
- Live regions for dynamic updates

Provide a Text Table Equivalent (when possible)

```
<table aria-hidden="true">
  <caption>Monthly Sales Data</caption>
  <thead>
    <tr><th>Month</th><th>Sales ($)</th></tr>
  </thead>
  <tbody>
    <tr><td>January</td><td>12,300</td></tr>
    <tr><td>February</td><td>14,200</td></tr>
    <!-- etc. -->
  </tbody>
</table>
```

This can supplement the visual chart and allow screen reader users to understand the data.

16.2.7 Label Axes and Data Points Clearly

Visual labels must also be **semantic**.

- Add axis labels using `<text>` or `<label>` elements.
- Include screen-reader-friendly data descriptions (especially for interactive charts).

D3 Example: Axis Labels

```
svg.append("text")
  .attr("class", "x label")
  .attr("text-anchor", "end")
  .attr("x", width)
  .attr("y", height + 40)
  .text("Year");
```

Consider also hiding this label from sighted users but **making it visible to screen readers**:

```
<span class="sr-only">X-axis: Year</span>
```

```
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  overflow: hidden;
  clip: rect(1px, 1px, 1px, 1px);
  white-space: nowrap;
}
```

16.2.8 Summary of Best Practices

Guideline	Do This
Use high-contrast colors	Pair color with shape or label
Add ARIA metadata	Use `<title>` and `<desc>` in SVG, or `aria-describedby`
Support keyboard input	Add `tabindex`, listen to `keydown` events
Provide text equivalents	Use summaries, tables, or captions
Label axes and regions	Use visible text and/or screen reader–only labels

Making your charts accessible is not just about compliance—it's about **respecting your audience** and ensuring that data is available to everyone, regardless of ability or device.

In the next section, you'll discover **resources** for high-quality datasets, open-source tools, and inspiring real-world examples of data visualization done right.