Chart.js Data Visualization

readbytes

Chart.js Data Visualization

Interactive and responsive charts

readbytes.github.io

2025-07-18

This page is intentionally left blank.

Contents

Chapter 1.

Introduction

1. What Is Chart.js?
2. Why Use Chart.js?
3. Setting Up Your Development Environment
4. Basic Concepts and Terminology

1 Introduction

1.1 What Is Chart.js?

Chart.js is a popular open-source JavaScript library designed to help developers create visually appealing, interactive, and responsive charts for web applications with minimal effort. At its core, Chart.js aims to make data visualization accessible and straightforward by providing a simple yet powerful API that abstracts much of the complexity typically involved in building charts from scratch.

Goals of Chart.js

The primary goal of Chart.js is to enable developers—regardless of their experience level—to quickly produce clean, elegant, and informative charts that enhance the user experience. It focuses on:

- **Simplicity:** Offering an intuitive API that requires little configuration to get started.
- **Responsiveness:** Making charts that adapt smoothly to different screen sizes and devices.
- **Customization:** Allowing flexible styling and interactivity options for developers who want more control.
- **Performance:** Efficient rendering using the HTML5 `<canvas>` element, balancing visual quality and speed.

Strengths of Chart.js

Chart.js stands out for several reasons:

- **Ease of Use:** It requires only a few lines of code to render a basic chart, which lowers the barrier to entry for developers new to data visualization.
- **Rich Set of Chart Types:** It supports many common chart types such as bar, line, pie, doughnut, radar, polar area, bubble, and scatter charts, covering most standard visualization needs.
- **Responsiveness and Animation:** Built-in animations and automatic responsiveness help make the charts engaging and mobile-friendly without extra coding.
- **Extensibility:** While simple by default, Chart.js can be customized deeply through plugins, options, and callbacks.
- **Open Source and Community Driven:** It has a vibrant community, frequent updates, and a wealth of examples and plugins.

Chart.js in the JavaScript Visualization Ecosystem

The JavaScript ecosystem for data visualization is rich and diverse, ranging from low-level libraries like D3.js, which offer unparalleled flexibility but require detailed coding, to high-level charting tools that trade some flexibility for ease of use.

Chart.js occupies an important niche as a **lightweight, easy-to-use, and visually attractive** charting library that sits comfortably between:

- **Low-level visualization libraries** like D3.js, which provide fine control over every aspect of the visualization but have a steep learning curve.
- **Full-featured commercial libraries** such as Highcharts or amCharts, which often offer more features out-of-the-box but may come with licensing costs.

Because of this balance, Chart.js is widely adopted for projects that need **quick, straightforward, and visually pleasing charts without a heavy overhead or complex setup**.

A Brief History

Chart.js was created by Nick Downie in 2013 and quickly gained popularity thanks to its minimalistic approach and ease of use. Over time, it has evolved through multiple versions, improving its API, adding new chart types, enhancing performance, and expanding its customization options.

Today, Chart.js is maintained by a community of contributors on GitHub and has become one of the most widely used JavaScript charting libraries, powering countless dashboards, reports, and data-driven applications.

1.2 Why Use Chart.js?

Choosing the right charting library can be crucial for building effective and engaging data visualizations. Chart.js offers several key benefits that make it an excellent choice for many projects, especially when you want to combine ease of use with solid functionality and flexibility. Let's explore the main reasons why developers often choose Chart.js for their visualization needs.

Simplicity and Ease of Use

One of Chart.js's greatest strengths is its simplicity. With just a few lines of code, you can create beautiful, functional charts without having to dive into complex configuration or low-level drawing commands. Its clean and straightforward API abstracts much of the complexity involved in rendering charts, making it accessible to beginners and efficient for experienced developers.

This simplicity doesn't mean limited power — Chart.js strikes a great balance between ease and capability, allowing you to focus on your data and the story you want to tell instead of wrestling with intricate library details.

Responsive Design by Default

In today's multi-device world, charts must look great on everything from desktops to smartphones. Chart.js is built on the HTML5 `<canvas>` element and supports responsive resizing out of the box. When the browser window changes size, your charts automatically adjust to fit, preserving readability and aesthetics.

This responsiveness saves you time and effort that you might otherwise spend managing layouts or writing custom resizing code.

Built-In Chart Types

Chart.js comes with a rich collection of commonly used chart types such as bar, line, pie, doughnut, radar, polar area, bubble, and scatter charts. These built-in types cover most of the standard visualization needs for dashboards, reports, and data analysis applications.

You don't need to hunt for plugins or build your own custom charts from scratch — Chart.js provides a solid foundation to create a wide variety of visualizations quickly.

Extensibility and Customization

While Chart.js works great with minimal configuration, it also offers powerful customization options when you need them. You can modify colors, fonts, animations, tooltips, legends, and more. Additionally, Chart.js supports plugins that extend its functionality, letting you add new features or tailor behavior to your specific requirements.

This flexibility means Chart.js can grow with your project, from simple charts to complex, interactive data presentations.

Easy Integration with JavaScript Frameworks

Whether you're using vanilla JavaScript or popular frameworks like React, Angular, or Vue, Chart.js integrates smoothly. There are many wrapper libraries and components designed to help you embed Chart.js charts within your chosen framework, making it easy to build reactive, data-driven interfaces without friction.

This ease of integration accelerates development and helps maintain clean, modular code.

1.2.1 How Chart.js Compares with Other Popular Libraries

To better understand where Chart.js shines, it helps to briefly compare it with some other widely used JavaScript visualization libraries:

- **D3.js**: D3 is a highly flexible and powerful library that lets you create custom visualizations by manipulating every aspect of the DOM and SVG elements. However, it has a steep learning curve and requires more setup. Chart.js, in contrast, offers ready-made chart types with less complexity, making it a faster choice for standard visualizations.
- **Highcharts**: Highcharts is a feature-rich commercial library with extensive chart types and advanced capabilities. It offers excellent support and more built-in features but comes with licensing costs for commercial use. Chart.js is free and open-source, making it ideal for budget-conscious projects or those wanting open customization.
- **ECharts**: ECharts provides highly interactive and visually rich charts with a strong focus on enterprise-level features and complex visualizations. It can be more complex

to learn and configure than Chart.js, which is designed to keep things simpler and lighter for quick development cycles.

1.3 Setting Up Your Development Environment

Before you can start building interactive charts with Chart.js, you need to set up a working development environment. Whether you're experimenting with quick prototypes or developing a full-featured application, Chart.js offers flexible setup options to suit your workflow.

This section will walk you through two common approaches:

- **Using a CDN for quick testing**
- **Installing via NPM or Yarn for project-based development**

We'll also cover useful tools like text editors and browser developer tools to streamline your Chart.js development experience.

Option 1: Quick Start with a CDN

If you need to jump straight into trying out Chart.js, the easiest way is to include it directly in your HTML file using a Content Delivery Network (CDN). This approach requires no installation and is ideal for quick demos, tutorials, or learning exercises.

Here's a basic HTML template using Chart.js via CDN:

```
<canvas id="myChart" width="400" height="200"></canvas>

<!-- Include Chart.js from CDN -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<script>
  const ctx = document.getElementById('myChart').getContext('2d');
  const myChart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
      datasets: [{
        label: 'Votes',
        data: [12, 19, 3, 5, 2, 3],
        backgroundColor: 'rgba(75, 192, 192, 0.5)',
        borderColor: 'rgba(75, 192, 192, 1)',
        borderWidth: 1
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
```

```
    });
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Chart.js Demo</title>
</head>
<body>
  <canvas id="myChart" width="400" height="200"></canvas>

  <!-- Include Chart.js from CDN -->
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

  <script>
    const ctx = document.getElementById('myChart').getContext('2d');
    const myChart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
        datasets: [{
          label: 'Votes',
          data: [12, 19, 3, 5, 2, 3],
          backgroundColor: 'rgba(75, 192, 192, 0.5)',
          borderColor: 'rgba(75, 192, 192, 1)',
          borderWidth: 1
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>
</body>
</html>
```

Save this as an `.html` file and open it in any modern browser to see your first Chart.js chart in action.

Option 2: Project-Based Setup with NPM or Yarn

For long-term projects, web applications, or any development requiring modular structure and version control, it's better to install Chart.js using a package manager like NPM or Yarn.

Step 1: Create a Project Folder Open your terminal and run:

```
mkdir chartjs-project
cd chartjs-project
```

```
npm init -y
```

Step 2: Initialize a New Project This creates a `package.json` file to track your project's dependencies.

```
npm install chart.js
```

Step 3: Install Chart.js Or with Yarn:

```
yarn add chart.js
```

Step 4: Create an HTML and JavaScript File In your project folder, create the following files:

- `index.html`
- `main.js`

Your `index.html` might look like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js with NPM</title>
</head>
<body>
  <canvas id="myChart"></canvas>

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

Your `main.js` would import Chart.js as a module:

```
import Chart from 'chart.js/auto';

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

new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr'],
    datasets: [{
      label: 'Sales',
      data: [10, 20, 15, 25],
      backgroundColor: 'rgba(54, 162, 235, 0.5)',
      borderColor: 'rgba(54, 162, 235, 1)',
      borderWidth: 2
    }]
  },
  options: {
```

```
    responsive: true
  }
});
```

To preview this locally, you'll need to run a local development server.

Step 5: Run a Local Development Server Modern browsers block JavaScript module imports (`type="module"`) on local files for security reasons, so you need a local server. Here are a few quick options:

- **Using VS Code's Live Server extension**
- **Using Node's built-in `http-server`**
  ```
  npx http-server .
  ```
- **Using Python's HTTP server**
  ```
  python -m http.server
  ```

Once the server is running, open the provided URL (usually `http://localhost:8080`) in your browser.

Recommended Tools for Chart.js Development

To make development smoother and more productive, consider the following tools:

- **VS Code (Visual Studio Code):** A powerful, free code editor with support for JavaScript, HTML, and extensions like Live Server and Prettier.
- **Browser Developer Tools:** All modern browsers (Chrome, Firefox, Edge, etc.) offer built-in dev tools. Use them to inspect your HTML canvas, debug scripts, and view console output.
- **Version Control (Git):** Use Git to manage your project history and collaborate with others. Services like GitHub or GitLab integrate seamlessly.

1.3.1 Summary

Whether you're quickly prototyping a chart or building a full-fledged data dashboard, Chart.js offers flexible setup options to match your workflow. For fast experimentation, a CDN-based setup is ideal. For scalable projects, using NPM or Yarn gives you better control and maintainability. With the help of a good code editor and browser tools, you're now ready to begin your Chart.js journey.

1.4 Basic Concepts and Terminology

Before you start building charts with Chart.js, it's important to understand the key concepts that form the foundation of how the library works. Whether you're configuring a simple bar chart or extending functionality with plugins, these core ideas will help you navigate the Chart.js API more effectively.

In this section, we'll introduce and explain the following fundamental terms:

- **Chart**
- **Dataset**
- **Options**
- **Scales**
- **Plugins**
- **Canvas Context**

We'll also include a conceptual diagram and a minimal code example to help you visualize how these pieces work together.

Chart

A **chart** is the central object in Chart.js. It represents the entire visualization rendered on the screen. You create a chart by calling the `Chart` constructor and passing it a configuration object that includes the data, chart type, and various options.

Every chart is tied to a `<canvas>` element, which serves as the drawing surface.

```
const myChart = new Chart(ctx, {
  type: 'bar',     // chart type
  data: { ... },   // chart data (labels and datasets)
  options: { ... } // chart configuration
});
```

Dataset

A **dataset** represents a single series of data values displayed in the chart. A dataset includes:

- The data points (e.g., `[10, 20, 30]`)
- Visual styling (e.g., color, border width)
- A label that appears in the legend and tooltips

You can have multiple datasets in a chart, allowing for comparisons between different series.

```
data: {
  labels: ['Q1', 'Q2', 'Q3'],
  datasets: [{
    label: 'Revenue',
    data: [15000, 20000, 18000],
    backgroundColor: 'steelblue'
  }]
}
```

Options

The **options** object lets you customize the behavior, appearance, and interactivity of your chart. This includes:

- Layout (padding, margins)
- Legends and tooltips
- Animation settings
- Responsiveness
- Axis configuration (through `scales`)
- Plugin settings

```
options: {
  responsive: true,
  plugins: {
    legend: {
      display: true,
      position: 'top'
    }
  }
}
```

Scales

Scales define how data values are mapped to visual positions on the chart. They control axes for Cartesian charts (like line, bar, scatter) and radial scaling for polar charts.

You can customize scale type (e.g., `linear`, `logarithmic`, `time`), labels, grid lines, and tick formatting.

```
options: {
  scales: {
    y: {
      beginAtZero: true,
      title: {
        display: true,
        text: 'Sales ($)'
      }
    }
  }
}
```

Plugins

Plugins are a powerful way to extend Chart.js's capabilities. They let you hook into the chart lifecycle (e.g., before/after render), modify data or visuals, and add custom behavior like annotations, zooming, or event tracking.

You can write your own plugins or use community-developed ones.

```
options: {
  plugins: {
    tooltip: {
      enabled: true,
      callbacks: {
        label: context => `$${context.formattedValue}`
```

```
      }
    }
  }
}
```

Canvas Context

Every Chart.js chart is rendered on an HTML5 `<canvas>` element. The **canvas context** (`ctx`) is the drawing environment passed into the `Chart` constructor. It provides access to the 2D rendering API used by Chart.js to draw charts.

```
<canvas id="myChart"></canvas>
```

```
const canvas = document.getElementById('myChart');
const ctx = canvas.getContext('2d'); // 2D rendering context
```

1.4.1 Conceptual Diagram: Chart.js Rendering Pipeline

```
[Canvas Element]
      V
[Canvas Context (ctx)]
      V
new Chart(ctx, {
      type: 'bar',
      data: {
        labels,
        datasets
      },
      options: {
        scales,
        plugins,
        layout
      }
})
```

Rendering Flow:

1. You select a `<canvas>` and get its `2D` context.
2. You pass the context and configuration to `new Chart(...)`.
3. Chart.js reads the configuration:
   - **`type`** defines the chart style (bar, line, etc.)
   - **`data`** includes labels and datasets
   - **`options`** control styling, scales, plugins, etc.
4. Chart.js renders the chart using the context.

1.4.2 Minimal Example: All Core Concepts in Action

```
<canvas id="demoChart"></canvas>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
  const ctx = document.getElementById('demoChart').getContext('2d');

  new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Jan', 'Feb', 'Mar'],
      datasets: [{
        label: 'Visitors',
        data: [120, 190, 300],
        backgroundColor: 'rgba(54, 162, 235, 0.6)'
      }]
    },
    options: {
      responsive: true,
      plugins: {
        legend: { display: true }
      },
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Chart.js Bar Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      text-align: center;
    }
    canvas {
      max-width: 600px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Monthly Visitors</h1>
  <canvas id="demoChart"></canvas>
```

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

    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Jan', 'Feb', 'Mar'],
        datasets: [{
          label: 'Visitors',
          data: [120, 190, 300],
          backgroundColor: 'rgba(54, 162, 235, 0.6)'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: { display: true }
        },
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

1.4.3 Summary

Understanding the basic building blocks of Chart.js — **charts**, **datasets**, **options**, **scales**, **plugins**, and the **canvas context** — will help you read, modify, and create charts with confidence. These concepts form the foundation of everything you'll build with Chart.js throughout this book. As you dive deeper, you'll discover how these elements interact and how to customize them for more advanced visualizations.

Chapter 2.

Getting Started with Chart.js

1. Installing Chart.js (CDN, NPM, Yarn)
2. Your First Chart: A Simple Bar Chart
3. Chart.js Anatomy: Canvas, Context, and Configuration
4. Chart Types Overview

2 Getting Started with Chart.js

2.1 Installing Chart.js (CDN, NPM, Yarn)

Before you can begin building interactive and responsive charts with Chart.js, you need to include the library in your project. Chart.js can be added in two primary ways:

- **Via CDN** for quick prototyping or small standalone projects
- **Via NPM or Yarn** for structured, package-managed development in larger applications

Let's walk through each installation method step by step, and help you decide which one is best for your use case.

2.1.1 Option 1: Using a CDN (Content Delivery Network)

This is the **simplest and fastest way** to start using Chart.js — especially useful for beginners, quick demos, testing, or simple HTML pages that don't require build tools.

When to Use:

- Prototyping or testing ideas quickly
- Learning Chart.js basics
- Embedding charts in static HTML pages
- No need for build steps like bundlers (e.g., Webpack, Vite)

How to Install:

1. Add the `<script>` tag in your HTML file, pointing to the official Chart.js CDN:

```
<canvas id="myChart" width="400" height="200"></canvas>

<!-- Chart.js from CDN -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<script>
  const ctx = document.getElementById('myChart').getContext('2d');
  new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Red', 'Blue', 'Yellow'],
      datasets: [{
        label: 'Votes',
        data: [12, 19, 3],
        backgroundColor: ['#f87171', '#60a5fa', '#facc15']
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
```

```
            beginAtZero: true
          }
        }
      }
    });
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js via CDN</title>
</head>
<body>
  <canvas id="myChart" width="400" height="200"></canvas>

  <!-- Chart.js from CDN -->
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

  <script>
    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Red', 'Blue', 'Yellow'],
        datasets: [{
          label: 'Votes',
          data: [12, 19, 3],
          backgroundColor: ['#f87171', '#60a5fa', '#facc15']
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>
</body>
</html>
```

That's it — no installation or build process needed.

2.1.2 Option 2: Using NPM (Node Package Manager)

NPM is the **recommended approach** for modern JavaScript development, especially when working with frameworks (React, Vue, Angular) or using bundlers.

When to Use:

- Larger or multi-file web apps
- Framework-based development
- Wanting version control and package management
- Using module bundlers like Webpack, Vite, or Parcel

How to Install:

1. Initialize your project (if not already done):

```
npm init -y
```

2. Install Chart.js as a dependency:

```
npm install chart.js
```

3. Use it in your JavaScript code (via ES modules):

```
// main.js
import Chart from 'chart.js/auto';

const ctx = document.getElementById('myChart').getContext('2d');
new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar'],
    datasets: [{
      label: 'Revenue',
      data: [500, 700, 600],
      borderColor: 'green',
      fill: false
    }]
  },
  options: {
    responsive: true
  }
});
```

4. Make sure your `index.html` uses the script as a module:

```
<canvas id="myChart"></canvas>
<script type="module" src="./main.js"></script>
```

5. Run a local development server (e.g., `npx http-server`, Vite, or Live Server in VS Code) to test your chart.

2.1.3 Option 3: Using Yarn (Alternative Package Manager)

Yarn is an alternative to NPM. It works similarly but has different dependency management mechanics. If your project uses Yarn, you can install Chart.js like this:

How to Install:

```
yarn add chart.js
```

Then use the same `import` and setup steps as with NPM.

2.1.4 Summary: Which Method Should You Choose?

Use Case	Recommended Method
Quick test, prototype, or demo	**CDN**
Small static HTML pages	**CDN**
Modern web apps with a build system	**NPM or Yarn**
Using frameworks (React, Vue, etc.)	**NPM or Yarn**
Wanting package version control	**NPM or Yarn**

Whether you want the convenience of a CDN or the control of a package manager, Chart.js makes it easy to get started. In the next section, you'll use one of these setups to build your first real chart.

2.2 Your First Chart: A Simple Bar Chart

Let's dive in and create your very first chart using Chart.js — a **bar chart**. This is one of the most common chart types and a great way to visualize comparative values such as sales, votes, or performance metrics.

In this section, you'll learn how to:

- Set up a canvas element as the chart container
- Write the JavaScript to define the chart configuration
- Use the `Chart` constructor to render your chart

By the end, you'll have a fully working, responsive bar chart!

2.2.1 Step-by-Step Guide

Here's the complete HTML and JavaScript code needed to create a basic bar chart using Chart.js via CDN.

```
<!-- Step 1: The Canvas Element -->
<canvas id="myBarChart" width="400" height="300"></canvas>

<!-- Step 2: Include Chart.js from CDN -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<!-- Step 3: Chart Configuration and Rendering -->
<script>
  // Get 2D drawing context from the canvas
  const ctx = document.getElementById('myBarChart').getContext('2d');

  // Create the bar chart
  new Chart(ctx, {
    type: 'bar', // Type of chart: bar
    data: {
      labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
      datasets: [{
        label: 'Votes',
        data: [12, 19, 3, 5, 2, 3],
        backgroundColor: [
          'rgba(255, 99, 132, 0.6)',
          'rgba(54, 162, 235, 0.6)',
          'rgba(255, 206, 86, 0.6)',
          'rgba(75, 192, 192, 0.6)',
          'rgba(153, 102, 255, 0.6)',
          'rgba(255, 159, 64, 0.6)'
        ],
        borderColor: [
          'rgba(255, 99, 132, 1)',
          'rgba(54, 162, 235, 1)',
          'rgba(255, 206, 86, 1)',
          'rgba(75, 192, 192, 1)',
          'rgba(153, 102, 255, 1)',
          'rgba(255, 159, 64, 1)'
        ],
        borderWidth: 1
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true,
          title: {
            display: true,
            text: 'Number of Votes'
          }
        },
        x: {
          title: {
            display: true,
            text: 'Colors'
          }
        }
      }
    }
  });
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Simple Bar Chart</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      max-width: 600px;
      margin: 2rem auto;
      padding: 1rem;
      text-align: center;
    }
    canvas {
      max-width: 100%;
    }
  </style>
</head>
<body>

  <h2>Favorite Colors Survey</h2>
  <!-- Step 1: The Canvas Element -->
  <canvas id="myBarChart" width="400" height="300"></canvas>

  <!-- Step 2: Include Chart.js from CDN -->
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

  <!-- Step 3: Chart Configuration and Rendering -->
  <script>
    // Get 2D drawing context from the canvas
    const ctx = document.getElementById('myBarChart').getContext('2d');

    // Create the bar chart
    new Chart(ctx, {
      type: 'bar', // Type of chart: bar
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
        datasets: [{
          label: 'Votes',
          data: [12, 19, 3, 5, 2, 3],
          backgroundColor: [
            'rgba(255, 99, 132, 0.6)',
            'rgba(54, 162, 235, 0.6)',
            'rgba(255, 206, 86, 0.6)',
            'rgba(75, 192, 192, 0.6)',
            'rgba(153, 102, 255, 0.6)',
            'rgba(255, 159, 64, 0.6)'
          ],
          borderColor: [
            'rgba(255, 99, 132, 1)',
            'rgba(54, 162, 235, 1)',
            'rgba(255, 206, 86, 1)',
            'rgba(75, 192, 192, 1)',
            'rgba(153, 102, 255, 1)',
            'rgba(255, 159, 64, 1)'
```

```
          ],
          borderWidth: 1
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Number of Votes'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Colors'
            }
          }
        }
      }
    });
  </script>
</body>
</html>
```

2.2.2 Explanation of Key Parts

Let's break down what each part of the code does:

Canvas Element

```
<canvas id="myBarChart" width="400" height="300"></canvas>
```

- This is where Chart.js will render your chart.
- You can control its size directly or let it resize responsively.
- The `id` is used to reference the element in JavaScript.

Chart.js CDN

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

- This pulls in the Chart.js library from a public CDN.
- No installation or build process required — perfect for quick setups.

JavaScript for the Chart

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

- This retrieves the 2D drawing context from the canvas. It's required for Chart.js to render graphics.

```
new Chart(ctx, {
  type: 'bar',
  ...
});
```

- The `Chart` constructor takes the context and a configuration object.
- `type: 'bar'` tells Chart.js to render a vertical bar chart.

```
data: {
  labels: [...],
  datasets: [...]
}
```

- `labels`: Categories that appear on the x-axis.
- `datasets`: An array of one or more data series (in this case, one set of votes).
- You can style each dataset with `backgroundColor`, `borderColor`, and `borderWidth`.

```
options: {
  responsive: true,
  scales: { ... }
}
```

- `responsive: true` ensures the chart resizes with the screen.
- `scales`: Lets you customize the x- and y-axes (e.g., start from zero, show titles).

2.2.3 Live Preview (CodePen)

You can try the code live here: Open in CodePen *(Editable template — open and paste the code above)*

2.2.4 What You Just Learned

By creating your first bar chart, you've learned:

- How to set up a canvas for Chart.js
- How to define the chart type, labels, and dataset
- How to configure basic options like scales and responsiveness
- How the `Chart` constructor ties it all together

In the next section, you'll dig deeper into how Chart.js works behind the scenes — exploring the canvas, context, and how the configuration object drives the rendering process.

2.3 Chart.js Anatomy: Canvas, Context, and Configuration

To truly understand how Chart.js works under the hood, it's helpful to break it down into its three core components:

- The HTML `<canvas>` element
- The JavaScript 2D rendering **context**
- The **configuration object** that defines what the chart looks like and how it behaves

These three parts work together in the Chart.js rendering pipeline. Once you understand how they connect, you'll be well equipped to build, customize, and troubleshoot your own charts with confidence.

2.3.1 The `canvas` Element

Chart.js renders charts using the HTML5 `<canvas>` element — a blank, pixel-based drawing surface built into modern web browsers. Unlike SVG (which uses markup to describe graphics), the canvas provides a lower-level drawing API that is faster and more efficient for rendering complex or animated visuals.

A typical canvas element looks like this:

```
<canvas id="myChart" width="600" height="400"></canvas>
```

Key Points:

- The `id` is used to reference the canvas in your JavaScript code.
- `width` and `height` set the initial resolution; CSS can further control layout.
- The canvas is completely empty by default — it doesn't know what to draw until you tell it.

2.3.2 The Canvas Context (`getContext('2d')`)

To draw on a canvas, you must access its **rendering context** — the environment that contains the drawing methods and properties.

In Chart.js, you typically retrieve the **2D context** like this:

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

This `ctx` is passed into the Chart.js constructor. Chart.js uses it to draw lines, bars, labels, grids, and everything else your chart displays.

You don't need to use drawing commands directly — Chart.js handles them behind the scenes. But it's helpful to know that the canvas is essentially a blank canvas (pun intended),

and Chart.js is the artist holding the brush.

2.3.3 The Chart.js Configuration Object

The **configuration object** is where you define every aspect of the chart. This object is passed to the `Chart` constructor and tells Chart.js what to render, how to render it, and how it should behave.

```
new Chart(ctx, {
  type: 'bar',
  data: { ... },
  options: { ... }
});
```

Let's break down each major section:

`type`: Chart Type

Specifies the kind of chart to draw: bar, line, pie, radar, doughnut, etc.

```
type: 'bar'
```

This determines the visual layout and default behavior.

`data`: Labels and Datasets

The `data` section contains:

- `labels`: Categories shown on the x-axis (or along angles for radial charts)
- `datasets`: One or more sets of values and their visual styles

```
data: {
  labels: ['Red', 'Blue', 'Green'],
  datasets: [{
    label: 'Votes',
    data: [12, 19, 3],
    backgroundColor: ['#f87171', '#60a5fa', '#34d399']
  }]
}
```

Each dataset includes:

- `label`: Text shown in the legend and tooltips
- `data`: Numeric values aligned with `labels`
- `backgroundColor`, `borderColor`, etc. for appearance

If you have multiple datasets, Chart.js will group or overlay them depending on the chart type.

`options`: Appearance, Behavior, and Interactivity

The `options` object controls how the chart behaves and appears. You can configure:

- **Responsiveness**: Does the chart resize with the window?
- **Scales**: How do axes behave?
- **Legend and tooltips**: Should they be displayed?
- **Plugins**: Enable or disable extra features
- **Animations**: Customize how the chart appears or updates

```
options: {
  responsive: true,
  scales: {
    y: {
      beginAtZero: true,
      title: {
        display: true,
        text: 'Count'
      }
    }
  },
  plugins: {
    legend: {
      display: true,
      position: 'top'
    }
  }
}
```

2.3.4 How It All Comes Together

Let's tie it all into one simple example:

Full runnable code:

```
<canvas id="myChart"></canvas>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
  const ctx = document.getElementById('myChart').getContext('2d');

  new Chart(ctx, {
    type: 'bar', // Type of chart

    data: {
      labels: ['A', 'B', 'C'],
      datasets: [{
        label: 'Scores',
        data: [10, 20, 15],
        backgroundColor: ['#93c5fd', '#a7f3d0', '#fcd34d']
      }]
    },

    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true
```

```
        }
      }
    }
  });
</script>
```

2.3.5 Visualization of the Flow

```
[<canvas id="myChart">]
         V
ctx = getContext('2d')
         V
new Chart(ctx, {
  type: 'bar',
  data: {...},
  options: {...}
})
         V
Chart.js interprets config and draws chart on canvas
```

2.3.6 Summary

Every Chart.js chart is powered by a simple architecture:

- A **<canvas> element** provides the space
- A **2D context** gives access to drawing tools
- A **configuration object** defines what to draw and how it should look

Understanding this structure will help you as you build increasingly advanced visualizations. In the next section, we'll explore the wide variety of chart types available in Chart.js so you can choose the right one for your data.

2.4 Chart Types Overview

Chart.js comes with a powerful set of **built-in chart types** that cover most common visualization needs. Whether you're displaying categorical comparisons, trends over time, proportions, or multivariate data, Chart.js has you covered.

This section introduces each chart type with a brief description of when to use it, and includes small code snippets to help you visualize the structure. In a web-based version of this book, you could also include interactive previews or thumbnail images for each chart.

2.4.1 Bar Chart

Use Case: Compare values across categories (e.g., survey results, product sales).

Bar charts are one of the most common chart types. Bars can be vertical (`type: 'bar'`) or horizontal (`type: 'bar'` + `indexAxis: 'y'`).

```
new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Apples', 'Oranges', 'Bananas'],
    datasets: [{
      label: 'Quantity',
      data: [12, 19, 8],
      backgroundColor: 'steelblue'
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Fruit Quantity Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Fruit Quantity</h1>
  <canvas id="fruitChart"></canvas>

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

    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Apples', 'Oranges', 'Bananas'],
        datasets: [{
          label: 'Quantity',
          data: [12, 19, 8],
          backgroundColor: 'steelblue'
        }]
```

```
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.2 Line Chart

Use Case: Show trends or changes over time (e.g., stock prices, temperature).

Line charts connect data points with straight lines, often used for time-series data.

```
new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar'],
    datasets: [{
      label: 'Revenue',
      data: [100, 150, 130],
      borderColor: 'green',
      fill: false
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Revenue Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
    }
```

```
  </style>
</head>
<body>

  <h1>Monthly Revenue</h1>
  <canvas id="revenueChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar'],
        datasets: [{
          label: 'Revenue',
          data: [100, 150, 130],
          borderColor: 'green',
          fill: false,
          tension: 0.3,
          pointBackgroundColor: 'green'
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.3 Pie Chart

Use Case: Display part-to-whole relationships (e.g., market share, budget breakdown).

Pie charts show how each segment contributes to a total. Best used with limited categories.

```
new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ['Red', 'Blue', 'Yellow'],
    datasets: [{
      data: [10, 20, 30],
      backgroundColor: ['#ef4444', '#3b82f6', '#facc15']
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Pie Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 400px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Pie Chart: Color Distribution</h1>
  <canvas id="pieChart"></canvas>

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

    new Chart(ctx, {
      type: 'pie',
      data: {
        labels: ['Red', 'Blue', 'Yellow'],
        datasets: [{
          data: [10, 20, 30],
          backgroundColor: ['#ef4444', '#3b82f6', '#facc15']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'bottom'
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.4 Doughnut Chart

Use Case: Similar to pie charts, but with a hollow center (great for visual emphasis or stats).

Doughnut charts function like pie charts, with a customizable inner radius.

```
new Chart(ctx, {
  type: 'doughnut',
  data: {
    labels: ['Chrome', 'Firefox', 'Safari'],
    datasets: [{
      data: [45, 25, 30],
      backgroundColor: ['#6366f1', '#f97316', '#10b981']
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Doughnut Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 400px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Browser Usage Share</h1>
  <canvas id="doughnutChart"></canvas>

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

    new Chart(ctx, {
      type: 'doughnut',
      data: {
        labels: ['Chrome', 'Firefox', 'Safari'],
        datasets: [{
          data: [45, 25, 30],
          backgroundColor: ['#6366f1', '#f97316', '#10b981']
        }]
      },
      options: {
```

```
        responsive: true,
        plugins: {
          legend: {
            position: 'bottom'
          }
        },
        cutout: '60%' // optional: customize the hollow center
      }
    });
  </script>

</body>
</html>
```

2.4.5 Radar Chart

Use Case: Compare multiple variables across categories (e.g., skill ratings, performance metrics).

Radar charts plot values in a circular layout, useful for comparing profiles.

```
new Chart(ctx, {
  type: 'radar',
  data: {
    labels: ['Speed', 'Strength', 'Agility', 'Endurance'],
    datasets: [{
      label: 'Athlete A',
      data: [65, 75, 80, 60],
      backgroundColor: 'rgba(59, 130, 246, 0.2)',
      borderColor: '#3b82f6'
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Radar Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
```

```
  </style>
</head>
<body>

  <h1>Athlete Performance Radar</h1>
  <canvas id="radarChart"></canvas>

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

    new Chart(ctx, {
      type: 'radar',
      data: {
        labels: ['Speed', 'Strength', 'Agility', 'Endurance'],
        datasets: [{
          label: 'Athlete A',
          data: [65, 75, 80, 60],
          backgroundColor: 'rgba(59, 130, 246, 0.2)',
          borderColor: '#3b82f6',
          pointBackgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          },
          title: {
            display: false
          }
        },
        scales: {
          r: {
            beginAtZero: true,
            suggestedMin: 0,
            suggestedMax: 100
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.6 Polar Area Chart

Use Case: Similar to pie/doughnut charts, but emphasizes magnitude more than proportion.

Segments grow outward based on value, rather than occupying angular space.

```
new Chart(ctx, {
  type: 'polarArea',
```

```
  data: {
    labels: ['A', 'B', 'C', 'D'],
    datasets: [{
      data: [11, 16, 7, 14],
      backgroundColor: ['#a78bfa', '#f472b6', '#34d399', '#60a5fa']
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Polar Area Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Polar Area Chart</h1>
  <canvas id="polarChart"></canvas>

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

    new Chart(ctx, {
      type: 'polarArea',
      data: {
        labels: ['A', 'B', 'C', 'D'],
        datasets: [{
          data: [11, 16, 7, 14],
          backgroundColor: ['#a78bfa', '#f472b6', '#34d399', '#60a5fa']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'right'
          },
          title: {
            display: false
          }
        },
```

```
        scales: {
          r: {
            beginAtZero: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.7 Scatter Chart

Use Case: Plot data points on a 2D plane — great for correlations or raw value distribution.

Each data point has an `x` and `y` coordinate, offering precision and flexibility.

```
new Chart(ctx, {
  type: 'scatter',
  data: {
    datasets: [{
      label: 'Experiments',
      data: [
        { x: 5, y: 10 },
        { x: 15, y: 12 },
        { x: 10, y: 8 }
      ],
      backgroundColor: '#f87171'
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Scatter Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
```

```
</head>
<body>

  <h1>Scatter Plot: Experiments</h1>
  <canvas id="scatterChart"></canvas>

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

    new Chart(ctx, {
      type: 'scatter',
      data: {
        datasets: [{
          label: 'Experiments',
          data: [
            { x: 5, y: 10 },
            { x: 15, y: 12 },
            { x: 10, y: 8 }
          ],
          backgroundColor: '#f87171'
        }]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            type: 'linear',
            position: 'bottom',
            title: {
              display: true,
              text: 'X Axis'
            }
          },
          y: {
            title: {
              display: true,
              text: 'Y Axis'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: function(context) {
                const point = context.raw;
                return `(${point.x}, ${point.y})`;
              }
            }
          }
        }
      }
    });
  </script>

</body>
```

```
</html>
```

2.4.8 Bubble Chart

Use Case: Like scatter charts, but each point also has a size (radius) — ideal for multivariate data.

Bubble charts extend scatter plots by adding a third dimension (`r` for radius).

```
new Chart(ctx, {
  type: 'bubble',
  data: {
    datasets: [{
      label: 'Population Bubbles',
      data: [
        { x: 20, y: 30, r: 15 },
        { x: 10, y: 20, r: 10 },
        { x: 30, y: 10, r: 25 }
      ],
      backgroundColor: 'rgba(59, 130, 246, 0.5)'
    }]
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Bubble Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h1>Population Bubbles</h1>
  <canvas id="bubbleChart"></canvas>

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

```
    new Chart(ctx, {
      type: 'bubble',
      data: {
        datasets: [{
          label: 'Population Bubbles',
          data: [
            { x: 20, y: 30, r: 15 },
            { x: 10, y: 20, r: 10 },
            { x: 30, y: 10, r: 25 }
          ],
          backgroundColor: 'rgba(59, 130, 246, 0.5)'
        }]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            title: {
              display: true,
              text: 'X Value'
            }
          },
          y: {
            title: {
              display: true,
              text: 'Y Value'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: context => {
                const { x, y, r } = context.raw;
                return `x: ${x}, y: ${y}, r: ${r}`;
              }
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

2.4.9 Summary Table

Chart Type	Use Case
Bar	Compare values across categories
Line	Show trends over time
Pie	Part-to-whole proportions
Doughnut	Like pie, but with a center hole
Radar	Multivariate comparison in a circular form
Polar Area	Pie-like, with variable radius
Scatter	Raw x/y data points
Bubble	Scatter + variable size (r) per point

2.4.10 What's Next?

Now that you've seen what types of charts you can build, the rest of this book will show you **how** to use and customize each of them. You'll learn how to:

- Choose the right chart for your data
- Modify styles and labels
- Handle multiple datasets
- Add interactivity and animations

In the next chapter, we'll start working with individual chart types in depth, starting with **bar and line charts** — the most commonly used and most flexible options in the Chart.js toolkit.

Chapter 3.

Understanding Chart Configuration

1. Chart Data Structure
2. Labels, Datasets, and Data Points
3. Chart Options: Layout, Scales, and Legends
4. Customizing Colors and Styles

3 Understanding Chart Configuration

3.1 Chart Data Structure

At the core of every Chart.js chart is the **`data` object** — a structured representation of what you want to visualize. This object defines the categories (labels) along the axes and the values (datasets) that will be plotted.

In this section, we'll break down the two main components of the `data` object:

- `labels`: Defines the categories or coordinates on the chart's axes
- `datasets`: Contains the data values and their visual styles

Understanding how these two pieces interact will help you build clear and accurate charts across all chart types.

3.1.1 The `data` Object: Overview

The `data` object is passed to the `Chart` constructor inside the configuration object:

```
new Chart(ctx, {
  type: 'bar',
  data: {
    labels: [...],
    datasets: [...]
  },
  options: { ... }
});
```

Here's what each part does:

- **`labels`**: Defines the values shown along the *x-axis* (or angles for circular charts)
- **`datasets`**: Defines one or more series of values to display on the chart

3.1.2 `labels`: Chart Axis Categories

The `labels` array holds the names of each category or point on the x-axis (or radial angle for pie/doughnut/radar charts). Each label maps directly to a data point in the datasets.

Example:

```
labels: ['January', 'February', 'March']
```

This will display three ticks or slices labeled *January*, *February*, and *March*.

3.1.3 `datasets`: One or More Data Series

The `datasets` array contains objects that define each data series to be plotted. Each object includes:

- `label`: A description shown in the chart legend and tooltips
- `data`: An array of values (matching the order of `labels`)
- Optional visual styles: `backgroundColor`, `borderColor`, `fill`, etc.

Example of a single dataset:

```
datasets: [{
  label: 'Sales',
  data: [150, 200, 170],
  backgroundColor: '#3b82f6'
}]
```

Each number in `data` corresponds to a label in the `labels` array:

- January → 150
- February → 200
- March → 170

3.1.4 Putting It Together: Annotated Example

```
const config = {
  type: 'bar',
  data: {
    // Labels define categories on the x-axis
    labels: ['Q1', 'Q2', 'Q3', 'Q4'],

    // Each dataset represents a series of values
    datasets: [{
      label: 'Revenue ($)',
      data: [12000, 15000, 10000, 17000],
      backgroundColor: 'rgba(75, 192, 192, 0.6)', // Fill color
      borderColor: 'rgba(75, 192, 192, 1)',       // Outline color
      borderWidth: 1
    }]
  },
  options: {
    responsive: true,
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
};
```

Full runnable code:

```html
<!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: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Quarterly Revenue</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
          label: 'Revenue ($)',
          data: [12000, 15000, 10000, 17000],
          backgroundColor: 'rgba(75, 192, 192, 0.6)',
          borderColor: 'rgba(75, 192, 192, 1)',
          borderWidth: 1
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.1.5 Multiple Datasets

You can visualize more than one dataset in a single chart — ideal for comparisons.

```
datasets: [
  {
    label: '2024',
    data: [12, 19, 3],
    backgroundColor: 'rgba(59, 130, 246, 0.6)'
  },
  {
    label: '2025',
    data: [17, 11, 6],
    backgroundColor: 'rgba(234, 88, 12, 0.6)'
  }
]
```

Each dataset will appear side-by-side (bar), as lines (line chart), or stacked (depending on the chart type and configuration).

3.1.6 Visual Mapping: How Data Is Drawn

```
Labels:   ['A', 'B', 'C']
Dataset:  [  5,  10, 15 ]

Result:
- A → 5
- B → 10
- C → 15
```

The position of each number in the `data` array aligns with the label at the same index. This simple structure makes Chart.js intuitive and powerful.

3.1.7 Summary

Property	Description
`labels`	Defines categories on the x-axis or radial layout
`datasets`	An array of data series; each has values and styling
`data`	Values mapped to labels in order
`label`	Text shown in the legend/tooltips for a dataset

The `data` object is the heart of any Chart.js configuration. Once you understand how labels and datasets connect, you'll be able to feed your charts with any data — from simple counts to multi-series comparisons.

Next, we'll explore how to customize **chart options**, including layout, scales, and legends.

3.2 Labels, Datasets, and Data Points

3.2.1 Labels, Datasets, and Data Points

At the core of every Chart.js visualization is the relationship between **labels**, **datasets**, and **data points**. These elements work together to map your raw data onto the chart.

In this section, you'll learn:

- What labels are and how they define chart categories
- How data points align with labels
- How multiple datasets enable chart comparisons
- How stacking and overlaying datasets works
- A visual diagram to make this connection clear

3.2.2 What Are Labels?

Labels are an array of strings that define categories or axis coordinates. They are used across many chart types:

- In **bar** and **line** charts, they appear along the x-axis
- In **pie** and **doughnut** charts, they define each slice
- In **radar** charts, they label each spoke

Example:

```
labels: ['Q1', 'Q2', 'Q3', 'Q4']
```

This means the chart will display four categories: Q1, Q2, Q3, and Q4.

3.2.3 What Are Data Points?

A **data point** is a single numeric value in a dataset. It aligns with one label in the `labels` array by **position (index)**.

Example with one dataset:

```
data: {
  labels: ['Jan', 'Feb', 'Mar'],
```

```
  datasets: [{
    label: 'Sales',
    data: [100, 150, 130]
  }]
}
```

Label	Data Point
Jan	100
Feb	150
Mar	130

Chart.js will map:

- Jan → 100
- Feb → 150
- Mar → 130

3.2.4 What Are Datasets?

A **dataset** is a collection of related data points, all mapped to the chart using the `labels` array. Each dataset can have its own visual style (e.g., color, border, fill).

Example of multiple datasets:

```
datasets: [
  {
    label: '2024',
    data: [120, 140, 100],
    backgroundColor: '#60a5fa'
  },
  {
    label: '2025',
    data: [160, 130, 110],
    backgroundColor: '#f97316'
  }
]
```

If `labels` is:

```
labels: ['Product A', 'Product B', 'Product C']
```

Then the mapping is:

Product	2024	2025
A	120	160
B	140	130
C	100	110

Product	2024	2025

Chart.js renders these side by side (in bar charts) or as separate lines (in line charts), depending on the chart type.

3.2.5 Example: Side-by-Side Bar Chart (Grouped)

```
const config = {
  type: 'bar',
  data: {
    labels: ['Q1', 'Q2', 'Q3'],
    datasets: [
      {
        label: '2023',
        data: [10, 15, 20],
        backgroundColor: '#3b82f6'
      },
      {
        label: '2024',
        data: [12, 18, 22],
        backgroundColor: '#f59e0b'
      }
    ]
  },
  options: {
    responsive: true,
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
};
```

This will render **grouped bars** for each quarter:

- Q1 shows two bars: 10 (2023), 12 (2024)
- Q2 shows two bars: 15 (2023), 18 (2024)
- Q3 shows two bars: 20 (2023), 22 (2024)

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bar Chart Comparison</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
```

```
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Quarterly Revenue Comparison (2023 vs 2024)</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
        datasets: [
          {
            label: '2023',
            data: [10, 15, 20],
            backgroundColor: '#3b82f6'
          },
          {
            label: '2024',
            data: [12, 18, 22],
            backgroundColor: '#f59e0b'
          }
        ]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.2.6 Example: Stacked Bar Chart

Stacking is enabled via `options.scales`:

```
options: {
  scales: {
    x: { stacked: true },
    y: { stacked: true }
  }
}
```

In a stacked bar chart:

- Q1 will show a single bar with height `10 + 12 = 22`
- Q2 → `15 + 18 = 33`
- Q3 → `20 + 22 = 42`

Stacked bars are ideal when you want to compare both **individual** and **total** values.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Stacked Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Stacked Bar Chart: Quarterly Revenue (2023 vs 2024)</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
        datasets: [
          {
            label: '2023',
            data: [10, 15, 20],
            backgroundColor: '#3b82f6'
          },
          {
            label: '2024',
            data: [12, 18, 22],
            backgroundColor: '#f59e0b'
          }
        ]
```

```
      },
      options: {
        responsive: true,
        scales: {
          x: {
            stacked: true
          },
          y: {
            stacked: true,
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.2.7 Example: Multi-Line Chart

In a line chart, datasets are plotted as separate lines sharing the same `labels`.

```
type: 'line',
data: {
  labels: ['Week 1', 'Week 2', 'Week 3'],
  datasets: [
    {
      label: 'Visitors (Site A)',
      data: [100, 120, 140],
      borderColor: '#10b981',
      fill: false
    },
    {
      label: 'Visitors (Site B)',
      data: [80, 110, 150],
      borderColor: '#ef4444',
      fill: false
    }
  ]
}
```

Each dataset is rendered as an individual line over the same x-axis labels.

3.2.8 Visual Representation: Mapping Labels to Datasets

```
labels:      ['Jan', 'Feb', 'Mar']
```

```
dataset[0]:  [  10 ,   20 ,   30 ]   // Line 1 or bar group 1
dataset[1]:  [  15 ,   25 ,   35 ]   // Line 2 or bar group 2
```

Visualization:

```
Jan             (10)
                  (15)

Feb               (20)
                    (25)

Mar               (30)
                    (35)
```

Each column groups values by label, and each color bar (or line) represents a dataset.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Multi-Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Website Visitors Over Time</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'line',
      data: {
        labels: ['Week 1', 'Week 2', 'Week 3'],
        datasets: [
          {
            label: 'Visitors (Site A)',
            data: [100, 120, 140],
            borderColor: '#10b981',
            fill: false,
            tension: 0.3
          },
          {
```

```
            label: 'Visitors (Site B)',
            data: [80, 110, 150],
            borderColor: '#ef4444',
            fill: false,
            tension: 0.3
          }
        ]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.2.9 Summary

Term	Description
Label	Defines categories or coordinates on the x-axis or angular/radial layout
Data Point	Numeric value corresponding to one label in a dataset
Dataset	A collection of data points rendered together; can be styled independently

Chart.js uses a simple but powerful indexing system: **each data point aligns with a label by index**. When using multiple datasets, Chart.js renders them either side-by-side, stacked, or overlaid — depending on chart type and configuration.

Next, we'll dive into **chart options**, where you'll learn how to control layout, axes, legends, and interaction behaviors.

3.3 Chart Options: Layout, Scales, and Legends

The `options` object in Chart.js gives you fine-grained control over how your chart looks and behaves. While the `type` and `data` properties define *what* is displayed, the `options` object determines *how* it's displayed — including layout, axes, legends, tooltips, responsiveness, and

interactivity.

In this section, we'll explore the most commonly used parts of the `options` object:

- `layout`: Adjusts padding and spacing inside the chart canvas
- `scales`: Controls the appearance and behavior of the x and y axes
- `plugins.legend`: Manages the display and positioning of the chart legend
- `plugins.tooltip`: Customizes hover tooltips

We'll look at examples to show how these settings impact the chart visually and functionally.

3.3.1 The `options` Object: Structure

Here's a quick look at the typical shape of the `options` object:

```
options: {
  layout: { ... },
  scales: { ... },
  plugins: {
    legend: { ... },
    tooltip: { ... }
  }
}
```

Let's walk through each of these sections.

3.3.2 Layout: Padding Around the Chart

The `layout` property controls the padding inside the chart canvas — between the chart area and the edges.

Example:

```
options: {
  layout: {
    padding: {
      top: 20,
      right: 30,
      bottom: 10,
      left: 15
    }
  }
}
```

You can set padding globally (as a number) or individually per side (as an object). This is helpful if your labels or legends are getting clipped or overlapping other elements.

3.3.3 Scales: Configuring Axes

The `scales` object controls each axis on the chart. You can modify how ticks, grid lines, labels, and axis titles appear.

Example: Configuring the Y-axis

```
options: {
  scales: {
    y: {
      beginAtZero: true,
      title: {
        display: true,
        text: 'Revenue ($)'
      },
      ticks: {
        color: '#333',
        stepSize: 20
      },
      grid: {
        color: '#ccc'
      }
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Y-Axis Config Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Revenue by Quarter</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
```

```
          label: 'Revenue',
          data: [40, 80, 60, 100],
          backgroundColor: 'rgba(75, 192, 192, 0.6)',
          borderColor: 'rgba(75, 192, 192, 1)',
          borderWidth: 1
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Revenue ($)'
            },
            ticks: {
              color: '#333',
              stepSize: 20
            },
            grid: {
              color: '#ccc'
            }
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

Example: Horizontal Bar Chart

To flip a bar chart horizontally, set the `indexAxis` to `'y'`:

```
options: {
  indexAxis: 'y'
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Horizontal Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
```

```
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Sales by Region</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['North', 'South', 'East', 'West'],
        datasets: [{
          label: 'Sales ($)',
          data: [12000, 18000, 10000, 15000],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        indexAxis: 'y', // <- This makes it horizontal
        responsive: true,
        scales: {
          x: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

Available Scale Options Include:

Property	Description
`beginAtZero`	Forces axis to start at zero
`title.text`	Sets the axis label
`ticks`	Controls tick marks (font, size, spacing, color, etc.)
`grid`	Adjusts grid line color, display, and style

3.3.4 Legends: Dataset Labels and Placement

Legends are displayed by default if your chart has more than one dataset. You can show, hide, or reposition the legend using `plugins.legend`.

Example:

```
options: {
  plugins: {
    legend: {
      display: true,
      position: 'top', // 'top', 'bottom', 'left', 'right'
      labels: {
        font: {
          size: 14
        },
        color: '#333'
      }
    }
  }
}
```

You can also completely hide the legend:

```
legend: {
  display: false
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart.js Legend Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Monthly Traffic Comparison</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'line',
      data: {
```

```
        labels: ['Jan', 'Feb', 'Mar', 'Apr'],
        datasets: [
          {
            label: 'Site A',
            data: [120, 150, 180, 170],
            borderColor: '#3b82f6',
            fill: false
          },
          {
            label: 'Site B',
            data: [100, 140, 160, 190],
            borderColor: '#f59e0b',
            fill: false
          }
        ]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            display: true,
            position: 'top', // Try: 'bottom', 'left', 'right'
            labels: {
              font: {
                size: 14
              },
              color: '#333'
            }
          }
        },
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.3.5 Tooltips: Hover Behavior

Tooltips appear when users hover over data points. You can customize their appearance and content.

Example:

```
options: {
  plugins: {
    tooltip: {
      enabled: true,
      backgroundColor: '#111',
      titleColor: '#fff',
      bodyColor: '#eee',
      callbacks: {
        label: function(context) {
          return `Value: ${context.parsed.y}`;
        }
      }
    }
  }
}
```

Tooltip Features:

Option	Description
`enabled`	Enable/disable tooltips
`backgroundColor`	Tooltip background color
`callbacks`	Functions to customize tooltip content

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart.js Tooltip Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Custom Tooltip Example</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
```

```
        labels: ['Apples', 'Bananas', 'Cherries', 'Dates'],
        datasets: [{
          label: 'Fruit Sales',
          data: [12, 19, 7, 14],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            enabled: true,
            backgroundColor: '#111',
            titleColor: '#fff',
            bodyColor: '#eee',
            callbacks: {
              label: function(context) {
                return `Value: ${context.parsed.y}`;
              }
            }
          }
        },
        scales: {
          y: {
            beginAtZero: true
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.3.6 Complete Example

Here's a full example that combines all of these options in a bar chart:

```
const config = {
  type: 'bar',
  data: {
    labels: ['Q1', 'Q2', 'Q3'],
    datasets: [{
      label: '2024 Revenue',
      data: [100, 150, 130],
      backgroundColor: '#3b82f6'
    }]
  },
  options: {
    layout: {
      padding: {
```

```
        top: 20,
        bottom: 10
      }
    },
    scales: {
      y: {
        beginAtZero: true,
        title: {
          display: true,
          text: 'Revenue ($)'
        },
        ticks: {
          stepSize: 50
        }
      }
    },
    plugins: {
      legend: {
        position: 'bottom',
        labels: {
          font: {
            size: 12
          }
        }
      },
      tooltip: {
        callbacks: {
          label: context => `Revenue: $${context.parsed.y}`
        }
      }
    }
  }
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart.js Custom Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>2024 Quarterly Revenue</h2>
```

```
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
        datasets: [{
          label: '2024 Revenue',
          data: [100, 150, 130],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        layout: {
          padding: {
            top: 20,
            bottom: 10
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Revenue ($)'
            },
            ticks: {
              stepSize: 50
            }
          }
        },
        plugins: {
          legend: {
            position: 'bottom',
            labels: {
              font: {
                size: 12
              }
            }
          },
          tooltip: {
            callbacks: {
              label: context => `Revenue: $${context.parsed.y}`
            }
          }
        }
      }
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.3.7 Summary

Option Section	Purpose
`layout`	Adds padding around the chart area
`scales`	Controls x/y axes: labels, ticks, titles, and grid
`plugins.legend`	Displays or customizes the chart legend
`plugins.tooltip`	Configures how tooltips behave on hover

The `options` object is your primary tool for tailoring your chart's layout, axis behavior, and interactivity. Once you're comfortable here, you'll have full control over how your chart is presented.

Next, we'll explore how to style your chart visually — customizing **colors, borders, fonts, gradients**, and more to match your app or brand design.

3.4 Customizing Colors and Styles

Chart.js makes it easy to transform plain charts into visually engaging, branded visualizations. From colors and gradients to point shapes and line widths, you can tailor every aspect of your chart's appearance.

This section shows how to:

- Apply solid and gradient **colors**
- Control **line width**, **border style**, and **fill options**
- Customize **points** in line/scatter/bubble charts
- Style **individual datasets** or **specific data points**
- Use **brand-friendly themes** for polished presentations

3.4.1 Applying Colors to Charts

The simplest way to style charts is by setting colors for chart elements like bars, lines, and points.

Example: Bar Chart with Custom Colors

```
datasets: [{
  label: 'Sales',
  data: [120, 180, 150],
  backgroundColor: ['#3b82f6', '#10b981', '#f59e0b'],
  borderColor: '#1e40af',
```

```
  borderWidth: 2
}]
```

- `backgroundColor`: Sets fill color (can be an array or a single color)
- `borderColor`: Sets the outline color
- `borderWidth`: Thickness of the border

You can supply an array of colors to assign different styles per data point.

3.4.2 Using Gradients for Fill and Lines

Gradients add visual depth and style. To use a gradient, you must access the 2D context and create it manually.

Example: Line Chart with Gradient Fill

```
const ctx = document.getElementById('myChart').getContext('2d');
const gradient = ctx.createLinearGradient(0, 0, 0, 400);
gradient.addColorStop(0, 'rgba(59, 130, 246, 0.4)');
gradient.addColorStop(1, 'rgba(59, 130, 246, 0)');

new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Mon', 'Tue', 'Wed'],
    datasets: [{
      label: 'Visitors',
      data: [100, 120, 110],
      borderColor: '#3b82f6',
      backgroundColor: gradient,
      fill: true
    }]
  }
});
```

Gradients can also be used on bar charts or backgrounds.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Line Chart with Gradient Fill</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
```

```
      max-width: 600px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Visitors Over Days</h2>
  <canvas id="myChart" width="600" height="400"></canvas>

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

    // Create vertical gradient
    const gradient = ctx.createLinearGradient(0, 0, 0, 400);
    gradient.addColorStop(0, 'rgba(59, 130, 246, 0.4)');
    gradient.addColorStop(1, 'rgba(59, 130, 246, 0)');

    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Mon', 'Tue', 'Wed'],
        datasets: [{
          label: 'Visitors',
          data: [100, 120, 110],
          borderColor: '#3b82f6',
          backgroundColor: gradient,
          fill: true,
          tension: 0.3,
          pointRadius: 4,
          pointHoverRadius: 6
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: ctx => `Visitors: ${ctx.parsed.y}`
            }
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Number of Visitors'
            }
          }
        }
      }
    });
  </script>
```

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

3.4.3 Line Widths, Styles, and Point Shapes

For **line** and **scatter** charts, you can style each dataset's appearance in more detail.

Example: Line Width, Dash Style, and Point Customization

```
datasets: [{
  label: 'Temperature',
  data: [22, 24, 21],
  borderColor: '#ef4444',
  borderWidth: 3,
  borderDash: [5, 5],        // Dashed line
  pointBackgroundColor: '#fff',
  pointBorderColor: '#ef4444',
  pointRadius: 6,
  pointStyle: 'rectRot'      // Circle, triangle, rectRot, star, etc.
}]
```

You can choose from built-in `pointStyle` options: `'circle'`, `'triangle'`, `'rect'`, `'rectRot'`, `'cross'`, `'crossRot'`, `'star'`, `'line'`

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Line Width, Dash, and Point Customization</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Temperature Over Days</h2>
  <canvas id="myChart"></canvas>

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

```
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Mon', 'Tue', 'Wed'],
        datasets: [{
          label: 'Temperature',
          data: [22, 24, 21],
          borderColor: '#ef4444',
          borderWidth: 3,
          borderDash: [5, 5],       // Dashed line pattern: 5px dash, 5px gap
          pointBackgroundColor: '#fff',
          pointBorderColor: '#ef4444',
          pointRadius: 6,
          pointStyle: 'rectRot'     // Rotated rectangle points
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: false,
            title: {
              display: true,
              text: 'Temperature (°C)'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: ctx => `Temp: ${ctx.parsed.y}°C`
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

3.4.4 Customizing Individual Data Points

Want to style a single bar, point, or slice differently? Just pass an array of styles matching the `data` array.

Example: Styling Specific Points

```
datasets: [{
  label: 'Revenue',
```

```
  data: [200, 300, 400],
  backgroundColor: [
    '#f87171', // Jan
    '#fbbf24', // Feb
    '#34d399'  // Mar
  ],
  borderColor: [
    '#b91c1c',
    '#92400e',
    '#065f46'
  ],
  borderWidth: [1, 2, 3]
}]
```

This is especially useful for:

- Highlighting outliers
- Emphasizing peaks
- Encoding categories visually

3.4.5 Brand-Friendly Styling Tips

When building dashboards or public-facing visualizations, it's important to match your brand's aesthetic. Here are some best practices:

Use a Consistent Color Palette

Use tools like Coolors or Adobe Color to define your palette. Apply consistent shades to datasets, legends, and backgrounds.

```
const brandBlue = '#1d4ed8';
const brandOrange = '#f97316';

datasets: [
  { label: '2023', data: [...], backgroundColor: brandBlue },
  { label: '2024', data: [...], backgroundColor: brandOrange }
]
```

Minimize Visual Noise

- Avoid too many colors in one chart
- Use subtle `borderWidth` and soft shadows for polish
- Use `hoverBackgroundColor` to improve interactivity

Use Fonts and Grid Colors That Match Your Theme

```
options: {
  plugins: {
    legend: {
      labels: {
        color: '#1f2937',
```

```
        font: {
          family: 'Helvetica',
          size: 14
        }
      }
    }
  },
  scales: {
    x: {
      ticks: {
        color: '#4b5563'
      },
      grid: {
        color: '#e5e7eb'
      }
    },
    y: {
      ticks: {
        color: '#4b5563'
      }
    }
  }
}
```

3.4.6 Complete Example: Branded Multi-Line Chart

```
const config = {
  type: 'line',
  data: {
    labels: ['Q1', 'Q2', 'Q3', 'Q4'],
    datasets: [
      {
        label: 'Product A',
        data: [120, 130, 115, 140],
        borderColor: '#2563eb',
        backgroundColor: 'rgba(37, 99, 235, 0.1)',
        borderWidth: 2,
        pointStyle: 'circle',
        pointRadius: 5
      },
      {
        label: 'Product B',
        data: [90, 105, 95, 100],
        borderColor: '#f59e0b',
        backgroundColor: 'rgba(245, 158, 11, 0.1)',
        borderDash: [5, 5],
        borderWidth: 2,
        pointStyle: 'rectRot',
        pointRadius: 5
      }
    ]
  },
  options: {
```

```
    plugins: {
      legend: {
        position: 'top',
        labels: {
          color: '#1f2937',
          font: { size: 14 }
        }
      }
    },
    scales: {
      y: {
        beginAtZero: true,
        ticks: { color: '#4b5563' },
        grid: { color: '#e5e7eb' }
      },
      x: {
        ticks: { color: '#4b5563' }
      }
    }
  }
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Branded Multi-Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 700px;
      margin: auto;
      display: block;
    }
  </style>
</head>
<body>

  <h2>Quarterly Sales for Products A & B</h2>
  <canvas id="myChart"></canvas>

  <script>
    const config = {
      type: 'line',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [
          {
            label: 'Product A',
            data: [120, 130, 115, 140],
```

```
          borderColor: '#2563eb',
          backgroundColor: 'rgba(37, 99, 235, 0.1)',
          borderWidth: 2,
          pointStyle: 'circle',
          pointRadius: 5,
          fill: true,
          tension: 0.3
        },
        {
          label: 'Product B',
          data: [90, 105, 95, 100],
          borderColor: '#f59e0b',
          backgroundColor: 'rgba(245, 158, 11, 0.1)',
          borderDash: [5, 5],
          borderWidth: 2,
          pointStyle: 'rectRot',
          pointRadius: 5,
          fill: true,
          tension: 0.3
        }
      ]
    },
    options: {
      responsive: true,
      plugins: {
        legend: {
          position: 'top',
          labels: {
            color: '#1f2937',
            font: { size: 14 }
          }
        }
      },
      scales: {
        y: {
          beginAtZero: true,
          ticks: { color: '#4b5563' },
          grid: { color: '#e5e7eb' },
          title: {
            display: true,
            text: 'Sales (units)',
            color: '#4b5563',
            font: { size: 14 }
          }
        },
        x: {
          ticks: { color: '#4b5563' },
          title: {
            display: true,
            text: 'Quarter',
            color: '#4b5563',
            font: { size: 14 }
          }
        }
      }
    }
  };
```

```
    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, config);
  </script>

</body>
</html>
```

3.4.7 Summary

Feature	Configuration Keys
Bar/line fill color	`backgroundColor`
Line color/width	`borderColor`, `borderWidth`, `borderDash`
Point shape/size	`pointStyle`, `pointRadius`, `pointBorderColor`
Gradient fill	Use `ctx.createLinearGradient()` with `backgroundColor`
Per-point customization	Arrays for `backgroundColor`, `borderColor`, etc.

Styling charts isn't just about looks — it improves **clarity, emphasis, and accessibility**. With Chart.js, you can start with sensible defaults and scale up to detailed, branded visuals as your project demands.

Next, we'll move on to working with **dynamic data** and updating charts in real-time.

Chapter 4.

Core Chart Types

1. Bar and Horizontal Bar Charts
2. Line Charts
3. Pie and Doughnut Charts
4. Radar and Polar Area Charts
5. Bubble and Scatter Charts

4 Core Chart Types

4.1 Bar and Horizontal Bar Charts

Bar charts are one of the most widely used chart types, ideal for comparing values across categories. Whether you're visualizing product sales, survey results, or student scores, bar charts are a great starting point.

In Chart.js, both vertical and horizontal bars use the same chart type (`'bar'`). The direction is controlled by a single configuration setting.

4.1.1 When to Use Bar vs Horizontal Bar Charts

Chart Type	Best For
Vertical Bar	Time-based comparisons, when categories fit well along the x-axis
Horizontal Bar	Long category labels, many data points, or better visual balance

Use horizontal bars when your labels are too long or when you're ranking items like top performers or survey responses.

4.1.2 Basic Vertical Bar Chart

Here's how to create a simple vertical bar chart for quarterly product sales:

```
<canvas id="salesChart"></canvas>
<script>
const ctx = document.getElementById('salesChart').getContext('2d');
new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Q1', 'Q2', 'Q3', 'Q4'],
    datasets: [{
      label: 'Product A Sales',
      data: [150, 200, 180, 220],
      backgroundColor: '#3b82f6'
    }]
  },
  options: {
    responsive: true,
    scales: {
      y: { beginAtZero: true }
    }
  }
});
</script>
```

Each bar represents a quarter's sales for Product A. The `labels` array maps to the x-axis, and the `data` values map to bar heights.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Vertical Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f3f4f6;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    canvas {
      max-width: 600px;
      width: 100%;
    }
  </style>
</head>
<body>

  <h2>Product A Quarterly Sales</h2>
  <canvas id="salesChart"></canvas>

  <script>
    const ctx = document.getElementById('salesChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
          label: 'Product A Sales',
          data: [150, 200, 180, 220],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Sales'
            }
          },
          x: {
            title: {
              display: true,
```

```
            text: 'Quarter'
          }
        }
      }
    }
  });
</script>

</body>
</html>
```

4.1.3 Creating a Horizontal Bar Chart

To flip the chart horizontally, set `indexAxis: 'y'` in the `options` object:

```
options: {
  indexAxis: 'y'
}
```

4.1.4 Example: Horizontal Survey Results

```
<canvas id="surveyChart"></canvas>
<script>
const ctx = document.getElementById('surveyChart').getContext('2d');
new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Excellent', 'Good', 'Average', 'Poor'],
    datasets: [{
      label: 'Responses',
      data: [55, 80, 45, 20],
      backgroundColor: '#10b981'
    }]
  },
  options: {
    indexAxis: 'y',
    responsive: true,
    scales: {
      x: { beginAtZero: true }
    }
  }
});
</script>
```

Horizontal bars give more space for long labels and naturally suit rating-based or ranking data.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Horizontal Survey Results</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    canvas {
      max-width: 600px;
      width: 100%;
    }
  </style>
</head>
<body>

  <h2>Survey Results</h2>
  <canvas id="surveyChart"></canvas>

  <script>
    const ctx = document.getElementById('surveyChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Excellent', 'Good', 'Average', 'Poor'],
        datasets: [{
          label: 'Responses',
          data: [55, 80, 45, 20],
          backgroundColor: '#10b981'
        }]
      },
      options: {
        indexAxis: 'y', // horizontal bars
        responsive: true,
        scales: {
          x: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Number of Responses'
            }
          },
          y: {
            title: {
              display: true,
              text: 'Rating'
            }
          }
        },
        plugins: {
          legend: {
```

```
            display: true,
            position: 'top'
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.1.5 Grouped Bar Charts

Grouped bar charts are useful for comparing multiple datasets side-by-side within each category (e.g., comparing sales between two products over several quarters).

4.1.6 Example:

```
data: {
  labels: ['Q1', 'Q2', 'Q3'],
  datasets: [
    {
      label: 'Product A',
      data: [150, 170, 160],
      backgroundColor: '#3b82f6'
    },
    {
      label: 'Product B',
      data: [130, 140, 155],
      backgroundColor: '#f59e0b'
    }
  ]
}
```

By default, Chart.js will group the bars within each category.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Grouped Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
```

```
      background: #f3f4f6;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    canvas {
      max-width: 700px;
      width: 100%;
    }
  </style>
</head>
<body>

  <h2>Quarterly Sales Comparison</h2>
  <canvas id="groupedBarChart"></canvas>

  <script>
    const ctx = document.getElementById('groupedBarChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
        datasets: [
          {
            label: 'Product A',
            data: [150, 170, 160],
            backgroundColor: '#3b82f6'
          },
          {
            label: 'Product B',
            data: [130, 140, 155],
            backgroundColor: '#f59e0b'
          }
        ]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Sales'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Quarter'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          }
        }
      }
```

```
    });
  </script>

</body>
</html>
```

4.1.7 Stacked Bar Charts

Stacked bar charts help show cumulative totals while still breaking down components.

4.1.8 Enable stacking:

```
options: {
  scales: {
    x: { stacked: true },
    y: { stacked: true }
  }
}
```

4.1.9 Example: Stacked Quarterly Sales by Region

```
data: {
  labels: ['Q1', 'Q2', 'Q3'],
  datasets: [
    {
      label: 'North',
      data: [50, 60, 55],
      backgroundColor: '#6366f1'
    },
    {
      label: 'South',
      data: [70, 80, 75],
      backgroundColor: '#f87171'
    }
  ]
}
```

In a stacked chart:

- Each category (Q1, Q2, Q3) displays one combined bar
- Each segment of the bar represents a dataset (e.g., North/South region)

You can stack either vertically or horizontally by adjusting the `indexAxis` and enabling stacking on the correct axis.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Stacked Bar Chart - Quarterly Sales by Region</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    canvas {
      max-width: 700px;
      width: 100%;
    }
  </style>
</head>
<body>

  <h2>Stacked Quarterly Sales by Region</h2>
  <canvas id="stackedSalesChart"></canvas>

  <script>
    const ctx = document.getElementById('stackedSalesChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
        datasets: [
          {
            label: 'North',
            data: [50, 60, 55],
            backgroundColor: '#6366f1'
          },
          {
            label: 'South',
            data: [70, 80, 75],
            backgroundColor: '#f87171'
          }
        ]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            stacked: true,
            title: {
              display: true,
              text: 'Quarter'
            }
          },
          y: {
```

```
            stacked: true,
            beginAtZero: true,
            title: {
              display: true,
              text: 'Sales'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.1.10 Adjusting Bar Spacing

Use `categoryPercentage` and `barPercentage` to fine-tune bar width and spacing:

```
options: {
  scales: {
    x: {
      categoryPercentage: 0.8, // space between categories (default: 0.8)
      barPercentage: 0.9       // width of bars within a group (default: 0.9)
    }
  }
}
```

Lower values create more space between bars; higher values make bars wider and closer together.

4.1.11 Summary: Key Bar Chart Features

Feature	Configuration Key	Purpose
Horizontal bars	`options.indexAxis: 'y'`	Rotate the chart for long labels
Grouped bars	Multiple `datasets[]`	Compare values side-by-side
Stacked bars	`scales.x.stacked`, `scales.y.stacked`	Show cumulative totals
Bar spacing	`categoryPercentage`, `barPercentage`	Adjust layout and density
Colors and styling	`backgroundColor`, `borderColor`	Customize appearance

4.1.12 When to Use Bar Charts

Use Case	Chart Type
Quarterly or monthly sales	**Vertical bar**
Survey results with long labels	**Horizontal bar**
Compare multiple groups	**Grouped bar**
Show breakdowns by category	**Stacked bar**

Bar charts are an excellent default choice for categorical data and comparisons. Next, we'll explore **line charts**, which are better suited for tracking trends over time.

4.2 Line Charts

Line charts are essential for visualizing trends, changes over time, or continuous data. They are commonly used for things like stock prices, temperature over days, web traffic analytics, and more.

In Chart.js, line charts are created by setting `type: 'line'` in your chart configuration. You can style each line, use multiple datasets, add dashed strokes, enable smooth curves, and customize data points.

4.2.1 When to Use Line Charts

Use a line chart when your data is:

- Time-based (e.g. days, months, years)
- Ordered (e.g. distance, rank, or progress)
- Continuous (e.g. temperature, sales trends)

4.2.2 Basic Line Chart

Here's a simple example showing temperature over a week:

```
<canvas id="lineChart"></canvas>
<script>
const ctx = document.getElementById('lineChart').getContext('2d');
new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
```

```
    datasets: [{
      label: 'Temperature (°C)',
      data: [22, 24, 23, 25, 21],
      borderColor: '#3b82f6',
      backgroundColor: 'rgba(59, 130, 246, 0.2)',
      fill: true
    }]
  },
  options: {
    responsive: true,
    scales: {
      y: {
        beginAtZero: false
      }
    }
  }
});
</script>
```

This chart shows a single smooth line with a filled background, perfect for tracking change over time.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic Line Chart - Temperature</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background-color: #f3f4f6;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    canvas {
      max-width: 700px;
      width: 100%;
    }
  </style>
</head>
<body>

  <h2>Weekly Temperature (°C)</h2>
  <canvas id="lineChart"></canvas>

  <script>
    const ctx = document.getElementById('lineChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
        datasets: [{
```

```
          label: 'Temperature (°C)',
          data: [22, 24, 23, 25, 21],
          borderColor: '#3b82f6',
          backgroundColor: 'rgba(59, 130, 246, 0.2)',
          fill: true,
          tension: 0.4
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: false,
            title: {
              display: true,
              text: 'Temperature (°C)'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Day of the Week'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: context => `Temp: ${context.parsed.y}°C`
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.2.3 Multiple Lines

To show comparisons, you can add multiple datasets—each dataset creates a separate line.

4.2.4 Example: Website Visitors Over Time

```
data: {
  labels: ['Week 1', 'Week 2', 'Week 3', 'Week 4'],
  datasets: [
    {
      label: 'Site A',
      data: [1200, 1500, 1100, 1600],
      borderColor: '#10b981',
      fill: false
    },
    {
      label: 'Site B',
      data: [1000, 1400, 1300, 1700],
      borderColor: '#f97316',
      fill: false
    }
  ]
}
```

This renders two distinct lines with different colors, allowing you to compare values across the same x-axis categories.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Website Visitors Over Time</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background-color: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 700px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Website Visitors Over Time</h2>
  <canvas id="visitorsChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Week 1', 'Week 2', 'Week 3', 'Week 4'],
        datasets: [
          {
            label: 'Site A',
```

```
            data: [1200, 1500, 1100, 1600],
            borderColor: '#10b981',
            backgroundColor: 'rgba(16, 185, 129, 0.1)',
            fill: false,
            tension: 0.3
          },
          {
            label: 'Site B',
            data: [1000, 1400, 1300, 1700],
            borderColor: '#f97316',
            backgroundColor: 'rgba(249, 115, 22, 0.1)',
            fill: false,
            tension: 0.3
          }
        ]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: false,
            title: {
              display: true,
              text: 'Visitors'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Week'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: context => `${context.dataset.label}: ${context.parsed.y} visitors`
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.2.5 Dashed and Styled Lines

You can style lines using properties like `borderWidth`, `borderDash`, and `borderColor`.

4.2.6 Example: Solid vs Dashed Lines

```
datasets: [
  {
    label: 'Projected Sales',
    data: [100, 120, 140],
    borderColor: '#6366f1',
    borderDash: [5, 5], // dashed line
    fill: false
  },
  {
    label: 'Actual Sales',
    data: [90, 110, 135],
    borderColor: '#3b82f6',
    borderWidth: 2,
    fill: false
  }
]
```

- `borderDash: [5, 5]` creates a dashed pattern
- `borderWidth` controls line thickness

Dashed lines are useful to differentiate projections, benchmarks, or goals from actual values.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Dashed and Styled Lines</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 700px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Projected vs Actual Sales</h2>
  <canvas id="salesChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Q1', 'Q2', 'Q3'],
```

```
        datasets: [
          {
            label: 'Projected Sales',
            data: [100, 120, 140],
            borderColor: '#6366f1',
            borderDash: [5, 5], // dashed line
            borderWidth: 2,
            fill: false,
            tension: 0.3
          },
          {
            label: 'Actual Sales',
            data: [90, 110, 135],
            borderColor: '#3b82f6',
            borderWidth: 2,
            fill: false,
            tension: 0.3
          }
        ]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          },
          tooltip: {
            callbacks: {
              label: ctx => `${ctx.dataset.label}: ${ctx.parsed.y}`
            }
          }
        },
        scales: {
          y: {
            beginAtZero: false,
            title: {
              display: true,
              text: 'Sales'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Quarter'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.2.7 Smooth Curves Using `tension`

You can control the curvature of lines using the `tension` property (a value between `0` and `1`).

- `0`: straight lines between points (default)
- `1`: maximum curvature

4.2.8 Example: Smooth Trend Line

```
datasets: [{
  label: 'Downloads',
  data: [300, 500, 400, 600],
  borderColor: '#0ea5e9',
  fill: true,
  tension: 0.4  // smooth curves
}]
```

Use smooth curves for better visual flow when the data isn't expected to jump sharply between points.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Smooth Curve Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 700px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Smooth Download Trend</h2>
  <canvas id="trendChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
```

```
        labels: ['Jan', 'Feb', 'Mar', 'Apr'],
        datasets: [{
          label: 'Downloads',
          data: [300, 500, 400, 600],
          borderColor: '#0ea5e9',
          backgroundColor: 'rgba(14, 165, 233, 0.2)',
          fill: true,
          tension: 0.4
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Download Count'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Month'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.2.9 Custom Point Styles

You can style each data point using:

- `pointStyle`: shape of the point
- `pointRadius`: size of the point
- `pointBorderColor`, `pointBackgroundColor`

4.2.10 Available `pointStyle` values:

- `'circle'` (default)
- `'triangle'`
- `'rect'`
- `'rectRot'`
- `'cross'`
- `'star'`
- `'line'`
- `'dash'`

4.2.11 Example: Custom Points

```
datasets: [{
  label: 'Users',
  data: [500, 600, 580, 610],
  borderColor: '#facc15',
  pointStyle: 'star',
  pointRadius: 6,
  pointBackgroundColor: '#facc15',
  pointBorderColor: '#92400e'
}]
```

This customizes each point's shape and color to stand out, which is helpful for charts with few points or high interaction.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Custom Point Styles</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #fefce8;
      text-align: center;
    }
    canvas {
      max-width: 700px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Users Per Week with Star Points</h2>
```

```
  <canvas id="pointChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Week 1', 'Week 2', 'Week 3', 'Week 4'],
        datasets: [{
          label: 'Users',
          data: [500, 600, 580, 610],
          borderColor: '#facc15',
          borderWidth: 2,
          pointStyle: 'star',
          pointRadius: 6,
          pointBackgroundColor: '#facc15',
          pointBorderColor: '#92400e',
          fill: false
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'User Count'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Week'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.2.12 Example: Time Series Chart

For true time-based data, use a `time` scale on the x-axis. You must install Chart.js adapters like `luxon` or `date-fns`.

Here's a basic example using ISO date strings:

```
type: 'line',
data: {
  datasets: [{
    label: 'CPU Usage (%)',
    data: [
      { x: '2025-01-01', y: 50 },
      { x: '2025-01-02', y: 65 },
      { x: '2025-01-03', y: 58 }
    ],
    borderColor: '#ef4444',
    tension: 0.3
  }]
},
options: {
  scales: {
    x: {
      type: 'time',
      time: {
        unit: 'day'
      }
    },
    y: {
      beginAtZero: true,
      max: 100
    }
  }
}
```

This chart plots real timestamps on the x-axis, making it perfect for monitoring logs, performance metrics, or time-based analytics.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Time Series Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/luxon"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 800px;
```

```
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>CPU Usage Over Time</h2>
  <canvas id="timeChart"></canvas>

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

    new Chart(ctx, {
      type: 'line',
      data: {
        datasets: [{
          label: 'CPU Usage (%)',
          data: [
            { x: '2025-01-01', y: 50 },
            { x: '2025-01-02', y: 65 },
            { x: '2025-01-03', y: 58 }
          ],
          borderColor: '#ef4444',
          backgroundColor: 'rgba(239, 68, 68, 0.1)',
          tension: 0.3,
          fill: true
        }]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            type: 'time',
            time: {
              unit: 'day'
            },
            title: {
              display: true,
              text: 'Date'
            }
          },
          y: {
            beginAtZero: true,
            max: 100,
            title: {
              display: true,
              text: 'CPU Usage (%)'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          }
        }
      }
    });
  </script>
```

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

4.2.13 Summary: Line Chart Features

Feature	Config Keys	Purpose
Basic line	`type: 'line'`	Visualize trends or ordered values
Multiple lines	Multiple `datasets[]`	Compare datasets side-by-side
Dashed lines	`borderDash: [5, 5]`	Distinguish projections or targets
Smooth curves	`tension: 0.1-1.0`	Soften sharp angle transitions
Custom points	`pointStyle`, `pointRadius`	Stylize data markers
Time series	`scales.x.type: 'time'`	Plot data by dates or timestamps

4.2.14 When to Use Line Charts

Use Case	Chart Style
Temperature over a week	Single line with fill and tension
Website traffic comparison	Multiple solid lines
Forecast vs actual	Dashed + solid lines
Time-stamped CPU monitoring	Time series with x-axis = `'time'`

Line charts help reveal patterns and direction in your data. Up next, we'll explore **Pie and Doughnut charts**, which are perfect for showing part-to-whole relationships.

4.3 Pie and Doughnut Charts

Pie and **doughnut** charts are ideal for showing **part-to-whole relationships** — how individual pieces contribute to a total. Common use cases include **budget allocation**, **market share**, and **survey responses**.

In Chart.js, pie and doughnut charts are closely related. They both use circular slices to represent values, but doughnut charts have a central hole, offering a slightly different visual emphasis.

4.3.1 Pie vs Doughnut: Key Differences

Feature	Pie Chart (`type: 'pie'`)	Doughnut Chart (`type: 'doughnut'`)
Shape	Full circle	Circle with hollow center
Readability	Better for small slice counts	Better for comparison and aesthetics
Use case example	Market share breakdown	Budget distribution by category

4.3.2 Creating a Basic Pie Chart

Here's how to create a pie chart showing market share among three companies:

```
<canvas id="pieChart"></canvas>
<script>
const ctx = document.getElementById('pieChart').getContext('2d');
new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ['Company A', 'Company B', 'Company C'],
    datasets: [{
      data: [45, 30, 25],
      backgroundColor: ['#3b82f6', '#f97316', '#10b981']
    }]
  },
  options: {
    responsive: true,
    plugins: {
      legend: {
        position: 'bottom'
      }
    }
  }
});
</script>
```

- The `labels` array defines slice labels.
- The `data` array defines the values for each slice.
- The `backgroundColor` array gives each slice its color.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic Pie Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
```

```
      background-color: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 400px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Market Share Distribution</h2>
  <canvas id="pieChart"></canvas>

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

    new Chart(ctx, {
      type: 'pie',
      data: {
        labels: ['Company A', 'Company B', 'Company C'],
        datasets: [{
          data: [45, 30, 25],
          backgroundColor: ['#3b82f6', '#f97316', '#10b981']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'bottom',
            labels: {
              font: { size: 14 },
              color: '#374151'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.3.3 Creating a Doughnut Chart

You can switch to a doughnut chart simply by changing the `type`:

```
type: 'doughnut'
```

4.3.4 Example: Department Budget Allocation

```
<canvas id="doughnutChart"></canvas>
<script>
const ctx = document.getElementById('doughnutChart').getContext('2d');
new Chart(ctx, {
  type: 'doughnut',
  data: {
    labels: ['Marketing', 'R&D', 'Operations', 'HR'],
    datasets: [{
      data: [30000, 25000, 20000, 10000],
      backgroundColor: ['#6366f1', '#f59e0b', '#ef4444', '#10b981']
    }]
  },
  options: {
    responsive: true,
    cutout: '60%', // controls size of doughnut hole
    plugins: {
      legend: {
        position: 'right'
      }
    }
  }
});
</script>
```

The `cutout` property sets the size of the inner hole (`'60%'` is a common default for doughnut charts).

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Doughnut Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background-color: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Department Budget Allocation</h2>
  <canvas id="doughnutChart"></canvas>

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

```
    new Chart(ctx, {
      type: 'doughnut',
      data: {
        labels: ['Marketing', 'R&D', 'Operations', 'HR'],
        datasets: [{
          data: [30000, 25000, 20000, 10000],
          backgroundColor: ['#6366f1', '#f59e0b', '#ef4444', '#10b981']
        }]
      },
      options: {
        responsive: true,
        cutout: '60%',
        plugins: {
          legend: {
            position: 'right',
            labels: {
              font: { size: 14 },
              color: '#374151'
            }
          },
          tooltip: {
            callbacks: {
              label: context => {
                const value = context.raw.toLocaleString();
                return `${context.label}: $${value}`;
              }
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.3.5 Customizing Colors and Styles

Each slice can be styled individually with the `backgroundColor`, `borderColor`, and `borderWidth` properties.

4.3.6 Example: Styling Slices

```
datasets: [{
  data: [60, 25, 15],
  backgroundColor: ['#3b82f6', '#f59e0b', '#10b981'],
  borderColor: '#ffffff',
  borderWidth: 2
}]
```

This adds white borders between slices to improve contrast and clarity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Styled Pie Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background-color: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
</head>
<body>

  <h2>Market Share Distribution</h2>
  <canvas id="styledPieChart"></canvas>

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

    new Chart(ctx, {
      type: 'pie',
      data: {
        labels: ['Company A', 'Company B', 'Company C'],
        datasets: [{
          data: [60, 25, 15],
          backgroundColor: ['#3b82f6', '#f59e0b', '#10b981'],
          borderColor: '#ffffff',
          borderWidth: 2
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'bottom',
            labels: {
              font: { size: 14 },
              color: '#374151'
            }
          },
          tooltip: {
            callbacks: {
              label: context => {
                const value = context.raw;
                return `${context.label}: ${value}%`;
              }
```

```
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.3.7 Labeling Tips and Legends

By default, Chart.js shows a **legend** using the `labels` array. To customize legend appearance:

```
plugins: {
  legend: {
    position: 'bottom',
    labels: {
      font: {
        size: 14
      },
      color: '#4b5563'
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Legend Styling Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f3f4f6;
      text-align: center;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
</head>
<body>

<h2>Pie Chart with Custom Legend</h2>
<canvas id="legendChart"></canvas>

<script>
```

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

new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ['Red', 'Blue', 'Yellow'],
    datasets: [{
      data: [30, 50, 20],
      backgroundColor: ['#ef4444', '#3b82f6', '#facc15']
    }]
  },
  options: {
    responsive: true,
    plugins: {
      legend: {
        position: 'bottom',
        labels: {
          font: {
            size: 14
          },
          color: '#4b5563'
        }
      }
    }
  }
});
</script>

</body>
</html>
```

4.3.8 Tooltips and Hover Effects

Tooltips are enabled by default but can be customized.

4.3.9 Example: Custom Tooltip Format

```
plugins: {
  tooltip: {
    callbacks: {
      label: function(context) {
        const label = context.label || '';
        const value = context.parsed;
        const total = context.dataset.data.reduce((a, b) => a + b, 0);
        const percentage = ((value / total) * 100).toFixed(1);
        return `${label}: ${value} (${percentage}%)`;
      }
    }
  }
```

```
}
```

This shows value **and** percentage inside the tooltip.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Custom Tooltip Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      text-align: center;
    }
    canvas {
      max-width: 500px;
      margin: auto;
    }
  </style>
</head>
<body>

<h2>Pie Chart with Custom Tooltip Showing Value and Percentage</h2>
<canvas id="tooltipChart"></canvas>

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

new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ['Apples', 'Oranges', 'Bananas'],
    datasets: [{
      data: [50, 30, 20],
      backgroundColor: ['#f87171', '#fbbf24', '#34d399']
    }]
  },
  options: {
    responsive: true,
    plugins: {
      tooltip: {
        callbacks: {
          label: function(context) {
            const label = context.label || '';
            const value = context.parsed;
            const total = context.dataset.data.reduce((a, b) => a + b, 0);
            const percentage = ((value / total) * 100).toFixed(1);
            return `${label}: ${value} (${percentage}%)`;
          }
        }
      },
      legend: {
```

```
        position: 'bottom'
      }
    }
  }
});
</script>

</body>
</html>
```

4.3.10 Use Case: Budget Breakdown Example

Let's say you want to visualize a $100,000 budget by department:

```
labels: ['Engineering', 'Marketing', 'HR', 'IT'],
data: [40000, 30000, 15000, 15000]
```

Choose:

- **Pie chart** if you need a compact, simple view
- **Doughnut chart** if you need to show totals or icons in the center

4.3.11 Interactivity and Animations

Pie/doughnut charts support built-in hover effects and animations.

```
options: {
  animation: {
    animateRotate: true,
    animateScale: true
  },
  hover: {
    mode: 'nearest',
    intersect: true
  }
}
```

These enhance usability and guide users to interact with specific slices.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Pie Chart with Interactivity and Animations</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
```

```
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f0f4f8;
      text-align: center;
    }
    canvas {
      max-width: 480px;
      margin: auto;
    }
  </style>
</head>
<body>

<h2>Pie Chart with Hover and Animation Effects</h2>
<canvas id="interactivePieChart"></canvas>

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

new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ['Red', 'Blue', 'Yellow'],
    datasets: [{
      data: [30, 45, 25],
      backgroundColor: ['#ef4444', '#3b82f6', '#facc15'],
      borderColor: '#fff',
      borderWidth: 2
    }]
  },
  options: {
    responsive: true,
    animation: {
      animateRotate: true,
      animateScale: true
    },
    hover: {
      mode: 'nearest',
      intersect: true
    },
    plugins: {
      legend: {
        position: 'bottom'
      },
      tooltip: {
        enabled: true
      }
    }
  }
});
</script>

</body>
</html>
```

4.3.12 Summary: Pie vs Doughnut Charts

Feature	Pie Chart	Doughnut Chart
Central hole	NO No	YES Yes (`cutout`)
Best for	Few values, simple layout	Balanced, modern aesthetics
Can show percentages	YES With tooltips/plugins	YES With tooltips/plugins
Custom slice styles	YES Via `backgroundColor`	YES Same

4.3.13 When to Use Each Chart

Scenario	Recommended Chart
Market share by brand	Pie
Budget allocation by team	Doughnut
Survey: preferred platforms	Pie or doughnut
Comparing part-to-whole data	Either

Pie and doughnut charts give immediate insight into proportions and help your audience interpret data quickly. Coming up next, we'll dive into **Radar and Polar Area charts**, which are great for multivariate comparisons and distribution visualization.

4.4 Radar and Polar Area Charts

Radar and **polar area** charts are radial chart types in Chart.js that are ideal for comparing multidimensional data or distributions across categories. Both use a circular layout, but they serve different purposes and display data in distinct ways.

4.4.1 What Are Radial Charts?

Radial charts plot data around a circle rather than along Cartesian (x/y) axes. They help visualize relative differences between variables, often making it easier to see outliers or patterns in complex datasets.

Chart.js supports two types of radial charts:

Chart Type	Description
Radar Chart	Plots variables on axes from the center, connects points to form a polygon
Polar Area Chart	Divides a circle into equal segments, with radius indicating magnitude

4.4.2 Radar Charts

A **radar chart** (also called a spider chart or web chart) is ideal for comparing attributes that share the same scale.

4.4.3 Best for:

- Skill level comparison
- Performance metrics
- Survey results across dimensions

4.4.4 Basic Radar Chart Example

Let's visualize programming skills for a developer:

```
<canvas id="radarChart"></canvas>
<script>
const ctx = document.getElementById('radarChart').getContext('2d');
new Chart(ctx, {
  type: 'radar',
  data: {
    labels: ['JavaScript', 'Python', 'C++', 'HTML', 'CSS', 'SQL'],
    datasets: [{
      label: 'Skill Level',
      data: [8, 7, 6, 9, 8, 7],
      backgroundColor: 'rgba(59, 130, 246, 0.2)',
      borderColor: '#3b82f6',
      pointBackgroundColor: '#3b82f6'
    }]
  },
  options: {
    responsive: true,
    scales: {
      r: {
        beginAtZero: true,
        max: 10
      }
```

```
    }
  }
});
</script>
```

This chart shows how a single person performs across several categories, all centered around a shared numeric scale (0–10).

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Radar Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      display: flex;
      justify-content: center;
    }
    canvas {
      max-width: 500px;
      max-height: 500px;
    }
  </style>
</head>
<body>

<canvas id="radarChart"></canvas>

<script>
const ctx = document.getElementById('radarChart').getContext('2d');
new Chart(ctx, {
  type: 'radar',
  data: {
    labels: ['JavaScript', 'Python', 'C++', 'HTML', 'CSS', 'SQL'],
    datasets: [{
      label: 'Skill Level',
      data: [8, 7, 6, 9, 8, 7],
      backgroundColor: 'rgba(59, 130, 246, 0.2)',
      borderColor: '#3b82f6',
      pointBackgroundColor: '#3b82f6'
    }]
  },
  options: {
    responsive: true,
    scales: {
      r: {
        beginAtZero: true,
        max: 10,
        ticks: {
          stepSize: 1
        },
        pointLabels: {
```

```
          font: {
            size: 14
          }
        }
      }
    }
  }
});
</script>

</body>
</html>
```

4.4.5 Comparing Multiple Datasets

You can add more datasets to compare multiple individuals or groups:

```
datasets: [
  {
    label: 'Developer A',
    data: [8, 7, 6, 9, 8, 7],
    borderColor: '#3b82f6',
    backgroundColor: 'rgba(59,130,246,0.2)'
  },
  {
    label: 'Developer B',
    data: [6, 8, 7, 7, 6, 8],
    borderColor: '#f59e0b',
    backgroundColor: 'rgba(245,158,11,0.2)'
  }
]
```

Each dataset forms its own polygon on the same axes, making it easy to spot strengths and weaknesses across categories.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Radar Chart Multiple Datasets</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      display: flex;
      justify-content: center;
    }
    canvas {
      max-width: 600px;
      max-height: 600px;
```

```
    }
  </style>
</head>
<body>

<canvas id="radarChart"></canvas>

<script>
const ctx = document.getElementById('radarChart').getContext('2d');
new Chart(ctx, {
  type: 'radar',
  data: {
    labels: ['JavaScript', 'Python', 'C++', 'HTML', 'CSS', 'SQL'],
    datasets: [
      {
        label: 'Developer A',
        data: [8, 7, 6, 9, 8, 7],
        borderColor: '#3b82f6',
        backgroundColor: 'rgba(59,130,246,0.2)',
        pointBackgroundColor: '#3b82f6',
      },
      {
        label: 'Developer B',
        data: [6, 8, 7, 7, 6, 8],
        borderColor: '#f59e0b',
        backgroundColor: 'rgba(245,158,11,0.2)',
        pointBackgroundColor: '#f59e0b',
      }
    ]
  },
  options: {
    responsive: true,
    scales: {
      r: {
        beginAtZero: true,
        max: 10,
        ticks: {
          stepSize: 1
        },
        pointLabels: {
          font: {
            size: 14
          }
        }
      }
    },
    plugins: {
      legend: {
        position: 'top',
        labels: {
          font: { size: 14 }
        }
      }
    }
  }
});
</script>
```

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

4.4.6 Pros and Cons of Radar Charts

Pros	Cons
Great for multivariate comparisons	Can become cluttered with many datasets
Easy to see strengths/weaknesses at a glance	Not intuitive for untrained users
Circular layout is compact and visually appealing	Harder to compare exact values

4.4.7 Polar Area Charts

A **polar area chart** is like a pie chart, but instead of slice angles, the **radius** represents the value. All segments are equally spaced around the circle.

4.4.8 Best for:

- Category-based metrics with varied magnitude
- Environmental factors
- Distribution comparisons

4.4.9 Basic Polar Area Chart Example

Here's a chart showing environmental metrics by category:

```
<canvas id="polarChart"></canvas>
<script>
const ctx = document.getElementById('polarChart').getContext('2d');
new Chart(ctx, {
  type: 'polarArea',
  data: {
    labels: ['Air Quality', 'Water Use', 'Energy', 'Recycling', 'Emissions'],
    datasets: [{
      data: [60, 80, 70, 50, 90],
      backgroundColor: [
        '#10b981',
        '#3b82f6',
```

```
        '#f59e0b',
        '#ef4444',
        '#8b5cf6'
      ]
    }]
  },
  options: {
    responsive: true,
    scales: {
      r: {
        beginAtZero: true
      }
    },
    plugins: {
      legend: {
        position: 'right'
      }
    }
  }
});
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Radar Chart Multiple Datasets</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      display: flex;
      justify-content: center;
    }
    canvas {
      max-width: 600px;
      max-height: 600px;
    }
  </style>
</head>
<body>
<canvas id="polarChart"></canvas>
<script>
const ctx = document.getElementById('polarChart').getContext('2d');
new Chart(ctx, {
  type: 'polarArea',
  data: {
    labels: ['Air Quality', 'Water Use', 'Energy', 'Recycling', 'Emissions'],
    datasets: [{
      data: [60, 80, 70, 50, 90],
      backgroundColor: [
        '#10b981',
        '#3b82f6',
        '#f59e0b',
```

```
        '#ef4444',
        '#8b5cf6'
      ]
    }]
  },
  options: {
    responsive: true,
    scales: {
      r: {
        beginAtZero: true
      }
    },
    plugins: {
      legend: {
        position: 'right'
      }
    }
  }
});
</script>
</body>
</html>
```

Each segment represents a category, and its radius reflects the value.

4.4.10 Styling Polar Area Charts

You can customize each slice:

```
backgroundColor: ['#3b82f6', '#10b981', '#f59e0b', '#ef4444'],
borderColor: '#fff',
borderWidth: 1
```

To enhance interactivity, tooltips and hover effects are enabled by default.

4.4.11 Pros and Cons of Polar Area Charts

Pros	Cons
Good for simple comparisons across categories	Not suitable for precise comparisons
Visually engaging and colorful	Limited space for labels
Easier to read than radar charts with fewer values	Not ideal for more than 6–8 categories

4.4.12 When to Use Each Radial Chart

Use Case	Recommended Chart
Skill comparison	Radar
Environmental impact breakdown	Polar Area
Product performance metrics	Radar
Resource usage or distribution	Polar Area

4.4.13 Summary: Radar vs Polar Area

Feature	Radar Chart	Polar Area Chart
Layout	Spiderweb (circular grid)	Circle with radial bars
Data display	Connected points (polygon)	Radius indicates value
Axes	Each label on a radial axis	No axes, just radius
Best for	Multi-attribute comparison	Category distribution
Dataset count	1–3 preferred	Usually just 1

Radar and polar area charts offer a compelling visual option for multidimensional data. When used appropriately, they reveal patterns that traditional bar and line charts may hide.

Next, we'll explore **bubble and scatter charts**, which are perfect for visualizing relationships between multiple numerical variables.

4.5 Bubble and Scatter Charts

Scatter and **bubble** charts are powerful tools for visualizing relationships between **two or more numeric variables**. Unlike categorical charts (like bar or pie), these charts plot data in a continuous coordinate system, making them ideal for identifying **correlations**, **clusters**, or **outliers**.

In Chart.js, both scatter and bubble charts use `{ x, y }` data points. The **bubble chart** extends this with an optional `r` (radius) to represent a **third dimension**.

4.5.1 Whats the Difference?

Chart Type	Data Format	Dimensions Visualized	Use Case
Scatter	`{ x, y }`	2D (x and y axes)	Correlation, trend analysis
Bubble	`{ x, y, r }`	3D (x, y + radius for magnitude)	Multivariable comparisons

4.5.2 Basic Scatter Chart

Let's say we want to analyze the relationship between **height and weight** for a group of individuals:

```
<canvas id="scatterChart"></canvas>
<script>
const ctx = document.getElementById('scatterChart').getContext('2d');
new Chart(ctx, {
  type: 'scatter',
  data: {
    datasets: [{
      label: 'Individuals',
      data: [
        { x: 160, y: 60 },
        { x: 170, y: 65 },
        { x: 180, y: 75 },
        { x: 190, y: 85 }
      ],
      backgroundColor: '#3b82f6'
    }]
  },
  options: {
    responsive: true,
    scales: {
      x: {
        title: { display: true, text: 'Height (cm)' }
      },
      y: {
        title: { display: true, text: 'Weight (kg)' }
      }
    }
  }
});
</script>
```

This scatter plot shows how weight varies with height. Each dot represents one individual's measurement.

Full runnable code:

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

```
  <title>Scatter Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Height vs Weight Scatter Chart</h2>
  <canvas id="scatterChart"></canvas>

  <script>
    const ctx = document.getElementById('scatterChart').getContext('2d');
    new Chart(ctx, {
      type: 'scatter',
      data: {
        datasets: [{
          label: 'Individuals',
          data: [
            { x: 160, y: 60 },
            { x: 170, y: 65 },
            { x: 180, y: 75 },
            { x: 190, y: 85 }
          ],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            position: 'top'
          }
        },
        scales: {
          x: {
            title: {
              display: true,
              text: 'Height (cm)'
            }
          },
          y: {
            title: {
              display: true,
              text: 'Weight (kg)'
            }
          }
        }
      }
    });
```

```
  </script>

</body>
</html>
```

4.5.3 Bubble Chart: Adding a Third Dimension

Now suppose we want to add **age** as a third variable — we'll use the `r` value (radius of the bubble) to show it:

```
<canvas id="bubbleChart"></canvas>
<script>
const ctx = document.getElementById('bubbleChart').getContext('2d');
new Chart(ctx, {
  type: 'bubble',
  data: {
    datasets: [{
      label: 'People',
      data: [
        { x: 160, y: 60, r: 6 },  // age ~ 30
        { x: 170, y: 65, r: 8 },  // age ~ 40
        { x: 180, y: 75, r: 10 }, // age ~ 50
        { x: 190, y: 85, r: 12 }  // age ~ 60
      ],
      backgroundColor: 'rgba(16, 185, 129, 0.5)',
      borderColor: '#10b981'
    }]
  },
  options: {
    responsive: true,
    scales: {
      x: {
        title: { display: true, text: 'Height (cm)' },
        beginAtZero: false
      },
      y: {
        title: { display: true, text: 'Weight (kg)' },
        beginAtZero: false
      }
    },
    plugins: {
      tooltip: {
        callbacks: {
          label: function(context) {
            const { x, y, r } = context.raw;
            return `Height: ${x} cm, Weight: ${y} kg, Age: ${r * 5} yrs`;
          }
        }
      }
    }
  }
});
</script>
```

Here:

- `x`: height
- `y`: weight
- `r`: age (scaled for display)

Each bubble's size gives an intuitive sense of age distribution while comparing other dimensions.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bubble Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f0fdf4;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Height vs Weight (Bubble = Age)</h2>
  <canvas id="bubbleChart"></canvas>

  <script>
    const ctx = document.getElementById('bubbleChart').getContext('2d');
    new Chart(ctx, {
      type: 'bubble',
      data: {
        datasets: [{
          label: 'People',
          data: [
            { x: 160, y: 60, r: 6 },  // ~30 yrs
            { x: 170, y: 65, r: 8 },  // ~40 yrs
            { x: 180, y: 75, r: 10 }, // ~50 yrs
            { x: 190, y: 85, r: 12 }  // ~60 yrs
          ],
          backgroundColor: 'rgba(16, 185, 129, 0.5)',
          borderColor: '#10b981'
        }]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            title: { display: true, text: 'Height (cm)' },
            beginAtZero: false
```

```
          },
          y: {
            title: { display: true, text: 'Weight (kg)' },
            beginAtZero: false
          }
        },
        plugins: {
          tooltip: {
            callbacks: {
              label: function(context) {
                const { x, y, r } = context.raw;
                return `Height: ${x} cm, Weight: ${y} kg, Age: ${r * 5} yrs`;
              }
            }
          },
          legend: {
            display: true,
            position: 'top'
          }
        }
      }
    });
  </script>

</body>
</html>
```

4.5.4 Example: Pricing Scenarios

Bubble charts are also helpful in business analytics. For example, comparing **product pricing**:

Price ($)	Sales Volume	Profit Margin
25	500	10
30	450	15
35	300	20
40	200	25

4.5.5 Configured as:

```
data: [
  { x: 25, y: 500, r: 10 },
  { x: 30, y: 450, r: 15 },
  { x: 35, y: 300, r: 20 },
  { x: 40, y: 200, r: 25 }
]
```

- `x`: Price
- `y`: Sales volume
- `r`: Profit margin

This lets decision-makers **balance revenue vs margin** with ease.

4.5.6 Styling Scatter and Bubble Charts

You can customize each dataset or point with properties like:

- `backgroundColor`
- `borderColor`
- `pointRadius` (scatter only)
- `hoverRadius`
- `pointStyle`: `'circle'`, `'rect'`, `'triangle'`, etc.

4.5.7 Example: Styling Individual Points

```
datasets: [{
  label: 'Dataset',
  data: [{ x: 5, y: 10 }, { x: 15, y: 20 }],
  pointBackgroundColor: ['#3b82f6', '#f59e0b'],
  pointRadius: [5, 8]
}]
```

You can also apply hover effects and animations for interactivity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Product Pricing Bubble Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f8fafc;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
```

```
<body>

  <h2>Pricing Scenarios: Revenue vs Margin</h2>
  <p><em>Bubble size represents profit margin</em></p>
  <canvas id="pricingChart"></canvas>

  <script>
    const ctx = document.getElementById('pricingChart').getContext('2d');
    new Chart(ctx, {
      type: 'bubble',
      data: {
        datasets: [{
          label: 'Product Pricing',
          data: [
            { x: 25, y: 500, r: 10 },
            { x: 30, y: 450, r: 15 },
            { x: 35, y: 300, r: 20 },
            { x: 40, y: 200, r: 25 }
          ],
          backgroundColor: ['#10b981', '#3b82f6', '#f59e0b', '#ef4444'],
          borderColor: '#334155',
          hoverBackgroundColor: '#111827'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            callbacks: {
              label: (context) => {
                const { x, y, r } = context.raw;
                return `Price: $${x}, Sales: ${y}, Margin: ${r}%`;
              }
            }
          },
          legend: {
            position: 'top'
          },
          title: {
            display: true,
            text: 'Product Pricing vs Sales & Margin'
          }
        },
        scales: {
          x: {
            title: {
              display: true,
              text: 'Price ($)'
            },
            beginAtZero: false
          },
          y: {
            title: {
              display: true,
              text: 'Sales Volume'
            },
            beginAtZero: true
          }
```

```
        }
      }
    });
  </script>

</body>
</html>
```

4.5.8 Summary: Scatter vs Bubble

Feature	Scatter Chart	Bubble Chart
Dimensions supported	2 (x, y)	3 (x, y, r)
Use case	Trends, correlation	Multivariate comparison
Size of point	Fixed (`pointRadius`)	Dynamic (`r`)
Default shape	Circle	Circle

4.5.9 When to Use

Scenario	Recommended Chart
Plotting height vs weight	Scatter
Comparing age, height, weight	Bubble
Price vs volume vs margin	Bubble
Analyzing distribution clusters	Scatter

Scatter and bubble charts allow you to visualize **rich, numeric datasets** in a compact, expressive format. They're especially useful when categorical charts can't capture the nuances of relationships and proportions.

In the next chapter, we'll explore how to **extend Chart.js with plugins** and build more advanced features like custom tooltips, labels, and interactions.

Chapter 5.

Advanced Chart Options

1. Axes and Scale Customization
2. Tooltips and Legends Customization
3. Animations and Transitions
4. Responsive and Adaptive Charts

5 Advanced Chart Options

5.1 Axes and Scale Customization

Chart.js offers flexible and powerful tools to customize chart **axes** and **scales**, allowing you to tailor visualizations to suit any dataset—whether it's linear, logarithmic, or time-based.

This section walks through how to configure **tick intervals**, **label formats**, **gridlines**, and **scale types**, with examples like formatting currency on the Y-axis and using time on the X-axis.

5.1.1 Chart.js Axis System Overview

Every Chart.js chart using Cartesian coordinates (`bar`, `line`, `scatter`, etc.) includes at least one **x-axis** and one **y-axis**. Each axis is controlled using the `options.scales` object:

```
options: {
  scales: {
    x: { ... },
    y: { ... }
  }
}
```

Each axis is a **scale** object with its own `type`, ticks, gridlines, and labels.

5.1.2 Tick Intervals and Step Size

To control how often ticks appear, use the `ticks.stepSize` property.

5.1.3 Example: Y-Axis in \10 Increments

```
options: {
  scales: {
    y: {
      beginAtZero: true,
      ticks: {
        stepSize: 10
      }
    }
  }
}
```

You can also use `min` and `max` to limit the visible range:

```
y: {
  min: 0,
  max: 100
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tick Intervals Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Y-Axis in Steps of 10</h2>
  <canvas id="stepChart"></canvas>

  <script>
    const ctx = document.getElementById('stepChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
        datasets: [{
          label: 'Performance Score',
          data: [15, 23, 45, 62, 78, 91],
          fill: false,
          borderColor: '#3b82f6',
          tension: 0.3
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            beginAtZero: true,
            min: 0,
            max: 100,
            ticks: {
              stepSize: 10
            },
            title: {
              display: true,
              text: 'Score'
```

```
            }
          },
          x: {
            title: {
              display: true,
              text: 'Month'
            }
          }
        },
        plugins: {
          legend: {
            position: 'top'
          },
          title: {
            display: true,
            text: 'Tick Step Size & Axis Limits Example'
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.1.4 Custom Tick Label Formatting

To format tick labels (e.g., currency, percentages, units), use the `ticks.callback` function:

5.1.5 Example: Format Y-axis as Currency

```
y: {
  ticks: {
    callback: function(value) {
      return '$' + value.toLocaleString();
    }
  }
}
```

This converts numbers like `1000` into `$1,000`.

5.1.6 Example: Format X-axis as Time

```
x: {
  type: 'time',
```

```
  time: {
    unit: 'month'
  },
  ticks: {
    callback: function(value) {
      return value; // You can format with moment.js or date-fns
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Custom Tick Label Formatting</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/luxon"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f1f5f9;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Revenue Over Time</h2>
  <canvas id="customTicksChart"></canvas>

  <script>
    const ctx = document.getElementById('customTicksChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        datasets: [{
          label: 'Monthly Revenue',
          data: [
            { x: '2024-01-01', y: 12000 },
            { x: '2024-02-01', y: 15000 },
            { x: '2024-03-01', y: 18000 },
            { x: '2024-04-01', y: 16000 },
            { x: '2024-05-01', y: 22000 }
          ],
          borderColor: '#3b82f6',
          backgroundColor: '#3b82f6',
          tension: 0.3,
          fill: false
        }]
      },
```

```
      options: {
        responsive: true,
        scales: {
          x: {
            type: 'time',
            time: {
              unit: 'month',
              tooltipFormat: 'MMM yyyy'
            },
            ticks: {
              callback: function(value, index, ticks) {
                // Format with Luxon via Chart.js adapter
                return luxon.DateTime.fromMillis(value).toFormat('MMM');
              }
            },
            title: {
              display: true,
              text: 'Month'
            }
          },
          y: {
            ticks: {
              callback: function(value) {
                return '$' + value.toLocaleString();
              }
            },
            title: {
              display: true,
              text: 'Revenue'
            }
          }
        },
        plugins: {
          title: {
            display: true,
            text: 'Custom Tick Formatting Example'
          },
          legend: {
            display: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.1.7 Scale Types: Linear, Log, Time, Category

`linear` (default for numeric axes)

```
type: 'linear'
```

Best for continuous numeric data.

logarithmic

```
type: 'logarithmic'
```

Useful for datasets with exponential growth (e.g., population, revenue).

```
y: {
  type: 'logarithmic',
  min: 1,
  max: 1000,
  ticks: {
    callback: function(value) {
      return value.toLocaleString();
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Logarithmic Axis Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Exponential Growth (Logarithmic Y-Axis)</h2>
  <canvas id="logChart"></canvas>

  <script>
    const ctx = document.getElementById('logChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5'],
        datasets: [{
          label: 'Population',
          data: [2, 10, 50, 200, 1000],
          borderColor: '#f59e0b',
          backgroundColor: '#f59e0b',
          tension: 0.3,
          fill: false
```

```
        }]
      },
      options: {
        responsive: true,
        scales: {
          y: {
            type: 'logarithmic',
            min: 1,
            max: 1000,
            ticks: {
              callback: function(value) {
                return value.toLocaleString();
              }
            },
            title: {
              display: true,
              text: 'Population Size (log scale)'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Time'
            }
          }
        },
        plugins: {
          title: {
            display: true,
            text: 'Logarithmic Y-Axis Chart'
          },
          legend: {
            display: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

time

Use this for time-series charts. You'll also need a time adapter like **date-fns**, **moment.js**, or **luxon**.

```
x: {
  type: 'time',
  time: {
    unit: 'day'
  },
  adapters: {
    date: {
      locale: yourLocaleObject
    }
  }
}
```

Example dataset using `{ x: date, y: value }` format:

```
data: [
  { x: '2025-01-01', y: 100 },
  { x: '2025-01-02', y: 120 },
  { x: '2025-01-03', y: 150 }
]
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Time Scale Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/luxon"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f8fafc;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Time-Series Data (Daily Values)</h2>
  <canvas id="timeChart"></canvas>

  <script>
    const ctx = document.getElementById('timeChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        datasets: [{
          label: 'Sales',
          data: [
            { x: '2025-01-01', y: 100 },
            { x: '2025-01-02', y: 120 },
            { x: '2025-01-03', y: 150 },
            { x: '2025-01-04', y: 170 },
            { x: '2025-01-05', y: 160 }
          ],
          borderColor: '#3b82f6',
          backgroundColor: '#3b82f6',
          tension: 0.3,
          fill: false
        }]
      },
      options: {
        responsive: true,
        scales: {
```

```
          x: {
            type: 'time',
            time: {
              unit: 'day'
            },
            title: {
              display: true,
              text: 'Date'
            }
          },
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Sales ($)'
            }
          }
        },
        plugins: {
          title: {
            display: true,
            text: 'Time-Series Line Chart (X = Date)'
          },
          legend: {
            display: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

category

For discrete string-based labels (e.g., months, categories):

```
x: {
  type: 'category',
  labels: ['Jan', 'Feb', 'Mar']
}
```

5.1.8 Gridline Customization

You can customize or remove gridlines with the `grid` configuration:

```
y: {
  grid: {
    color: '#e5e7eb',
    lineWidth: 1,
    drawBorder: false,
    borderDash: [4, 2]
  }
```

```
}
```

5.1.9 Remove gridlines entirely:

```
grid: {
  display: false
}
```

5.1.10 Axis Title and Styling

Use `title` to add axis labels:

```
x: {
  title: {
    display: true,
    text: 'Date',
    font: {
      size: 14,
      weight: 'bold'
    }
  }
}
```

5.1.11 Example: Y-Axis in Currency, X-Axis as Time

```
options: {
  scales: {
    x: {
      type: 'time',
      time: {
        unit: 'month'
      },
      title: {
        display: true,
        text: 'Month'
      }
    },
    y: {
      beginAtZero: true,
      ticks: {
        callback: value => '$' + value.toLocaleString()
      },
      title: {
        display: true,
        text: 'Revenue'
```

```
      }
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Revenue Over Time</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/luxon"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 720px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Monthly Revenue (Formatted as Currency)</h2>
  <canvas id="currencyTimeChart"></canvas>

  <script>
    const ctx = document.getElementById('currencyTimeChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        datasets: [{
          label: 'Revenue',
          data: [
            { x: '2025-01-01', y: 12000 },
            { x: '2025-02-01', y: 15000 },
            { x: '2025-03-01', y: 18000 },
            { x: '2025-04-01', y: 17000 },
            { x: '2025-05-01', y: 21000 }
          ],
          borderColor: '#10b981',
          backgroundColor: '#10b981',
          tension: 0.4,
          fill: false
        }]
      },
      options: {
        responsive: true,
        scales: {
          x: {
            type: 'time',
```

```
            time: {
              unit: 'month'
            },
            title: {
              display: true,
              text: 'Month'
            }
          },
          y: {
            beginAtZero: true,
            ticks: {
              callback: value => '$' + value.toLocaleString()
            },
            title: {
              display: true,
              text: 'Revenue'
            }
          }
        },
        plugins: {
          title: {
            display: true,
            text: 'Time-Series Chart with Currency Axis'
          },
          legend: {
            display: true
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.1.12 Summary: Axis Customization Features

Feature	Key Properties
Tick spacing	`ticks.stepSize`, `min`, `max`
Tick formatting	`ticks.callback`
Grid styling	`grid.color`, `grid.lineWidth`, `display`
Axis titles	`title.text`, `title.display`
Scale type (linear, log)	`type: 'linear'`, `'logarithmic'`, etc.

5.1.13 Use Cases at a Glance

Goal	How to Do It
Format Y-axis as currency	`ticks.callback` + prefix `$`
Show time on X-axis	`type: 'time'` + time adapter
Limit Y-axis to 0–100 range	`min: 0`, `max: 100`
Customize gridline appearance	`grid.color`, `grid.display`
Use a logarithmic Y-axis	`type: 'logarithmic'`

With these tools, you can shape your charts to communicate clearly and effectively. In the next section, we'll explore **tooltips and legends customization**—another critical part of building user-friendly visualizations.

5.2 Tooltips and Legends Customization

Tooltips and legends are vital interactive elements in Chart.js charts that help users **interpret data** easily. Customizing their appearance and behavior can improve clarity and user experience, especially in complex visualizations.

This section explains how to configure tooltips and legends through the `options.plugins.tooltip` and `options.plugins.legend` settings, including formatting tooltip content and adjusting legend position and style.

5.2.1 Customizing Tooltips

Tooltips appear when users hover over chart elements, showing details about the data point.

5.2.2 Basic Tooltip Setup

By default, tooltips show dataset labels and values. You can customize their content using **callback functions** in the `tooltip` plugin.

5.2.3 Formatting Tooltip Content

Use the `callbacks.label` function to control what appears inside tooltips.

```
options: {
  plugins: {
```

```
    tooltip: {
      callbacks: {
        label: function(context) {
          const label = context.dataset.label || '';
          const value = context.parsed.y !== undefined ? context.parsed.y : context.parsed;
          return `${label}: ${value.toLocaleString()}`;
        }
      }
    }
  }
}
```

- `context.dataset.label` provides the dataset name.
- `context.parsed` or `context.parsed.y` gives the value of the hovered point.
- You can format numbers using `.toLocaleString()` for better readability.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tooltip Formatting Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f1f5f9;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Custom Tooltip Content</h2>
  <canvas id="tooltipChart"></canvas>

  <script>
    const ctx = document.getElementById('tooltipChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
          label: 'Earnings',
          data: [12000, 17500, 14250, 19800],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
```

```
            callbacks: {
              label: function(context) {
                const label = context.dataset.label || '';
                const value = context.parsed.y !== undefined ? context.parsed.y : context.parsed;
                return `${label}: $${value.toLocaleString()}`;
              }
            }
          },
          title: {
            display: true,
            text: 'Earnings by Quarter'
          },
          legend: {
            display: true
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Earnings ($)'
            },
            ticks: {
              callback: value => '$' + value.toLocaleString()
            }
          },
          x: {
            title: {
              display: true,
              text: 'Quarter'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.4 Advanced Tooltip Formatting

For charts with multiple dimensions, like bubble charts, you can access all data properties:

```
label: function(context) {
  const { x, y, r } = context.raw;
  return `X: ${x}, Y: ${y}, Size: ${r}`;
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bubble Chart Tooltip</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f0fdf4;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Bubble Chart with Advanced Tooltip</h2>
  <canvas id="bubbleTooltipChart"></canvas>

  <script>
    const ctx = document.getElementById('bubbleTooltipChart').getContext('2d');
    new Chart(ctx, {
      type: 'bubble',
      data: {
        datasets: [{
          label: 'Samples',
          data: [
            { x: 10, y: 100, r: 5 },
            { x: 20, y: 200, r: 10 },
            { x: 30, y: 150, r: 15 },
            { x: 40, y: 250, r: 20 }
          ],
          backgroundColor: '#10b981'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            callbacks: {
              label: function(context) {
                const { x, y, r } = context.raw;
                return `X: ${x}, Y: ${y}, Size: ${r}`;
              }
            }
          },
          title: {
            display: true,
            text: 'Advanced Tooltip Formatting'
          }
        },
        scales: {
          x: {
            title: {
```

```
            display: true,
            text: 'X Value'
          },
          beginAtZero: true
        },
        y: {
          title: {
            display: true,
            text: 'Y Value'
          },
          beginAtZero: true
        }
      }
    }
  });
</script>

</body>
</html>
```

5.2.5 Positioning Tooltips

You can control where tooltips appear relative to the cursor:

```
tooltip: {
  position: 'nearest',  // default: 'average'
  yAlign: 'top',        // 'top', 'bottom', 'center'
  xAlign: 'center'
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Tooltip Positioning</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Custom Tooltip Positioning</h2>
```

```
  <canvas id="positionTooltipChart"></canvas>

  <script>
    const ctx = document.getElementById('positionTooltipChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        datasets: [{
          label: 'Visitors',
          data: [120, 190, 300, 250, 400],
          borderColor: '#3b82f6',
          backgroundColor: '#3b82f6',
          fill: false,
          tension: 0.2
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            position: 'nearest', // positions tooltip near the hovered point (default is 'average')
            yAlign: 'top',       // vertical alignment: 'top', 'bottom', 'center'
            xAlign: 'center',    // horizontal alignment: 'left', 'right', 'center'
            callbacks: {
              label: ctx => `Visitors: ${ctx.parsed.y}`
            }
          },
          title: {
            display: true,
            text: 'Tooltip Positioned Near Cursor, Aligned Top Center'
          },
          legend: {
            display: true
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: { display: true, text: 'Visitors' }
          },
          x: {
            title: { display: true, text: 'Month' }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.6 Styling Tooltips

Customize tooltip appearance using properties like:

```
tooltip: {
  backgroundColor: 'rgba(0,0,0,0.7)',
  titleFont: { size: 14, weight: 'bold' },
  bodyFont: { size: 12 },
  padding: 10,
  cornerRadius: 6,
  caretSize: 8
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Styled Tooltip Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Custom Styled Tooltip</h2>
  <canvas id="styledTooltipChart"></canvas>

  <script>
    const ctx = document.getElementById('styledTooltipChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Apples', 'Oranges', 'Bananas', 'Grapes'],
        datasets: [{
          label: 'Fruit Sales',
          data: [120, 150, 180, 90],
          backgroundColor: '#3b82f6'
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            backgroundColor: 'rgba(0,0,0,0.7)',
            titleFont: { size: 14, weight: 'bold' },
            bodyFont: { size: 12 },
            padding: 10,
```

```
            cornerRadius: 6,
            caretSize: 8,
            callbacks: {
              label: ctx => `${ctx.dataset.label}: ${ctx.parsed.y}`
            }
          },
          title: {
            display: true,
            text: 'Bar Chart with Styled Tooltip'
          },
          legend: {
            display: true
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: { display: true, text: 'Sales' }
          },
          x: {
            title: { display: true, text: 'Fruit' }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.7 Customizing Legends

Legends describe the chart datasets and help users identify colors or categories.

5.2.8 Legend Positioning

You can change where the legend appears using `position`:

```
legend: {
  display: true,
  position: 'top',   // 'top', 'left', 'bottom', 'right'
}
```

Full runnable code:

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

```
  <title>Legend Positioning</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f0f4f8;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Legend Positioned at the Top</h2>
  <canvas id="legendPositionChart"></canvas>

  <script>
    const ctx = document.getElementById('legendPositionChart').getContext('2d');
    new Chart(ctx, {
      type: 'pie',
      data: {
        labels: ['Chrome', 'Firefox', 'Edge', 'Safari'],
        datasets: [{
          label: 'Browser Usage',
          data: [55, 25, 10, 10],
          backgroundColor: ['#3b82f6', '#f59e0b', '#10b981', '#ef4444']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            display: true,
            position: 'top' // Change to 'left', 'bottom', 'right' to test
          },
          title: {
            display: true,
            text: 'Browser Usage with Legend at Top'
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.9 Styling Legend Labels

Adjust label font, color, and box sizes for better clarity:

```
legend: {
  labels: {
    color: '#374151',  // text color
    font: {
      size: 14,
      family: 'Arial',
      weight: 'bold'
    },
    boxWidth: 20,
    padding: 15
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Styled Legend Labels</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Legend Labels with Custom Style</h2>
  <canvas id="styledLegendChart"></canvas>

  <script>
    const ctx = document.getElementById('styledLegendChart').getContext('2d');
    new Chart(ctx, {
      type: 'doughnut',
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green'],
        datasets: [{
          label: 'Colors',
          data: [25, 35, 20, 20],
          backgroundColor: ['#ef4444', '#3b82f6', '#fbbf24', '#10b981']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            display: true,
            position: 'right',
            labels: {
```

```
            color: '#374151', // dark gray text
            font: {
              size: 14,
              family: 'Arial',
              weight: 'bold'
            },
            boxWidth: 20,
            padding: 15
          }
        },
        title: {
          display: true,
          text: 'Doughnut Chart with Styled Legend Labels'
        }
      }
    }
  });
</script>

</body>
</html>
```

5.2.10 Legend Interaction

You can control what happens on legend item clicks (e.g., toggling datasets):

```
legend: {
  onClick: function(e, legendItem, legend) {
    const index = legendItem.datasetIndex;
    const chart = legend.chart;
    chart.toggleDataVisibility(index);
    chart.update();
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Legend Interaction Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
```

```
  </style>
</head>
<body>

  <h2>Legend Click Toggles Dataset Visibility</h2>
  <canvas id="legendInteractionChart"></canvas>

  <script>
    const ctx = document.getElementById('legendInteractionChart').getContext('2d');
    const myChart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        datasets: [
          {
            label: 'Dataset 1',
            data: [10, 20, 30, 40, 50],
            borderColor: '#3b82f6',
            backgroundColor: '#3b82f6',
            fill: false,
            tension: 0.3
          },
          {
            label: 'Dataset 2',
            data: [50, 40, 30, 20, 10],
            borderColor: '#ef4444',
            backgroundColor: '#ef4444',
            fill: false,
            tension: 0.3
          }
        ]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            display: true,
            onClick: function(e, legendItem, legend) {
              const index = legendItem.datasetIndex;
              const chart = legend.chart;
              chart.toggleDataVisibility(index);
              chart.update();
            }
          },
          title: {
            display: true,
            text: 'Click Legend Items to Toggle Visibility'
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Value'
            }
          },
          x: {
```

```
            title: {
              display: true,
              text: 'Month'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.11 Putting It Together: Example

```
options: {
  plugins: {
    tooltip: {
      backgroundColor: '#111827',
      titleFont: { size: 16, weight: 'bold' },
      bodyFont: { size: 14 },
      callbacks: {
        label: function(context) {
          return `${context.dataset.label}: ${context.parsed.y.toFixed(2)}`;
        }
      }
    },
    legend: {
      position: 'bottom',
      labels: {
        color: '#1f2937',
        font: { size: 14 },
        boxWidth: 18
      }
    }
  }
}
```

This configuration styles a dark tooltip with formatted numbers and places a clearly visible legend at the bottom.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Styled Tooltip & Legend Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: system-ui, sans-serif;
```

```
      padding: 2rem;
      background: #f3f4f6;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Styled Tooltip and Legend</h2>
  <canvas id="styledChart"></canvas>

  <script>
    const ctx = document.getElementById('styledChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        datasets: [{
          label: 'Revenue',
          data: [1200.5, 1500.75, 1800.2, 1600, 2100.1],
          borderColor: '#2563eb',
          backgroundColor: '#2563eb',
          fill: false,
          tension: 0.3
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            backgroundColor: '#111827',   // dark background
            titleFont: { size: 16, weight: 'bold' },
            bodyFont: { size: 14 },
            callbacks: {
              label: function(context) {
                return `${context.dataset.label}: ${context.parsed.y.toFixed(2)}`;
              }
            }
          },
          legend: {
            position: 'bottom',
            labels: {
              color: '#1f2937',  // dark gray
              font: { size: 14 },
              boxWidth: 18
            }
          },
          title: {
            display: true,
            text: 'Revenue Over Months'
          }
        },
        scales: {
          y: {
            beginAtZero: true,
```

```
            title: {
              display: true,
              text: 'Revenue ($)'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Month'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.2.12 Summary: Customizing Tooltips and Legends

Feature	Configuration Property	Description
Tooltip content format	`plugins.tooltip.callbacks.label`	Custom text for each tooltip item
Tooltip position	`plugins.tooltip.position`	Controls tooltip placement
Tooltip styling	`backgroundColor`, `titleFont`, `padding`	Visual appearance customization
Legend position	`plugins.legend.position`	Where the legend appears
Legend label styles	`plugins.legend.labels`	Font, color, box size, padding
Legend interactivity	`plugins.legend.onClick`	Custom click behavior

Mastering tooltips and legends helps you make charts **intuitive and user-friendly**, guiding your audience to insights quickly and effectively. Next, we'll explore how to bring your charts to life with **animations and transitions**.

5.3 Animations and Transitions

Animations bring charts to life by smoothly transitioning between states, helping users follow data changes and enhancing visual appeal. Chart.js offers built-in animation support with customizable options to control **duration**, **easing**, and which chart properties animate.

This section explains how to leverage Chart.js animations and demonstrates transitioning between two datasets with smooth interpolation.

5.3.1 Animation Basics in Chart.js

By default, Chart.js animates charts when they first render or update. Animations are controlled in the `options.animation` object:

```
options: {
  animation: {
    duration: 1000,    // Duration in milliseconds
    easing: 'easeOutQuart',  // Easing function for smoothness
    animateRotate: true,     // Animate rotation for pie/doughnut charts
    animateScale: false      // Animate scaling effect for pie/doughnut charts
  }
}
```

- **`duration`** controls how long the animation lasts.
- **`easing`** affects the acceleration/deceleration curve (e.g., `'linear'`, `'easeInOutQuad'`, `'easeOutBounce'`).
- Other flags control specific animation aspects depending on chart type.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Animation Basics</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Pie Chart with Custom Animation</h2>
  <canvas id="animationChart"></canvas>

  <script>
    const ctx = document.getElementById('animationChart').getContext('2d');
    new Chart(ctx, {
      type: 'doughnut',
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green'],
        datasets: [{
          label: 'Votes',
          data: [12, 19, 3, 5],
          backgroundColor: ['#ef4444', '#3b82f6', '#fbbf24', '#10b981']
        }]
      },
      options: {
```

```
        responsive: true,
        animation: {
          duration: 1500,          // 1.5 seconds animation
          easing: 'easeOutQuart',  // smooth easing
          animateRotate: true,     // animate rotation (default true)
          animateScale: false      // disable scaling effect
        },
        plugins: {
          title: {
            display: true,
            text: 'Animated Doughnut Chart'
          },
          legend: {
            position: 'right'
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.3.2 What Properties Animate?

For Cartesian charts (bar, line, scatter), Chart.js animates:

- Data point positions (x/y coordinates)
- Bar heights and widths
- Line paths

For radial charts (pie, doughnut):

- Rotation and scaling of arcs

5.3.3 Example: Smooth Transition Between Two Datasets

Suppose you want to update a line chart with new data values and have it animate smoothly.

5.3.4 Initial Dataset

```
const data1 = [10, 20, 30, 25, 15];
```

5.3.5 New Dataset

```
const data2 = [15, 25, 20, 35, 30];
```

5.3.6 HTML JavaScript Setup

```
<canvas id="myChart"></canvas>
<button id="updateBtn">Update Data</button>

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

const chart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    datasets: [{
      label: 'Sales',
      data: data1,
      borderColor: '#3b82f6',
      fill: false,
      tension: 0.4
    }]
  },
  options: {
    animation: {
      duration: 800,
      easing: 'easeInOutCubic'
    },
    responsive: true
  }
});

document.getElementById('updateBtn').addEventListener('click', () => {
  chart.data.datasets[0].data = data2;
  chart.update();  // triggers animated transition
});
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Animated Data Update</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f0f4f8;
```

```
    }
    canvas {
      max-width: 700px;
      height: 400px;
      display: block;
      margin-bottom: 1rem;
    }
    button {
      padding: 0.6rem 1.2rem;
      font-size: 1rem;
      background-color: #3b82f6;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    button:hover {
      background-color: #2563eb;
    }
  </style>
</head>
<body>

  <canvas id="myChart"></canvas>
  <button id="updateBtn">Update Data</button>

  <script>
    const data1 = [12, 19, 3, 5, 2];
    const data2 = [5, 10, 15, 20, 25];

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

    const chart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
        datasets: [{
          label: 'Sales',
          data: data1,
          borderColor: '#3b82f6',
          fill: false,
          tension: 0.4
        }]
      },
      options: {
        animation: {
          duration: 800,
          easing: 'easeInOutCubic'
        },
        responsive: true,
        plugins: {
          title: {
            display: true,
            text: 'Sales Over Months'
          },
          legend: {
            display: true
          }
```

```
        },
        scales: {
          y: {
            beginAtZero: true,
            title: { display: true, text: 'Sales' }
          },
          x: {
            title: { display: true, text: 'Month' }
          }
        }
      }
    });

    document.getElementById('updateBtn').addEventListener('click', () => {
      chart.data.datasets[0].data = data2;
      chart.update();  // triggers animated transition
    });
  </script>

</body>
</html>
```

5.3.7 Explanation

- The chart initially renders with `data1`.
- Clicking the **Update Data** button replaces the dataset with `data2`.
- Calling `chart.update()` animates the transition between old and new points using configured duration and easing.
- The line smoothly interpolates between values instead of jumping abruptly.

5.3.8 Animation Easing Options

Chart.js supports many easing functions such as:

Easing Name	Description
`linear`	Constant speed
`easeInQuad`	Starts slow, accelerates
`easeOutQuad`	Starts fast, slows down
`easeInOutCubic`	Slow start and end, faster middle
`easeOutBounce`	Bounce effect at end

Experiment to find the effect that best fits your data story.

5.3.9 Advanced Animation Controls

You can target specific properties for animation with `animations` options:

```
options: {
  animations: {
    x: {
      duration: 500,
      easing: 'linear'
    },
    y: {
      duration: 1000,
      easing: 'easeOutBounce'
    }
  }
}
```

This example animates the x-axis and y-axis data positions with different timings and easing.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Advanced Animation Controls</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

  <h2>Chart.js with Targeted Property Animations</h2>
  <canvas id="advancedAnimationChart"></canvas>

  <script>
    const ctx = document.getElementById('advancedAnimationChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple'],
        datasets: [{
          label: 'Votes',
          data: [12, 19, 3, 5, 2],
          backgroundColor: ['#ef4444', '#3b82f6', '#fbbf24', '#10b981', '#8b5cf6']
        }]
      },
      options: {
        responsive: true,
```

```
        animations: {
          x: {
            duration: 500,
            easing: 'linear'
          },
          y: {
            duration: 1000,
            easing: 'easeOutBounce'
          }
        },
        plugins: {
          title: {
            display: true,
            text: 'Different Animation for X and Y Axes'
          },
          legend: {
            display: true
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: { display: true, text: 'Votes' }
          },
          x: {
            title: { display: true, text: 'Colors' }
          }
        }
      }
    });
  </script>

</body>
</html>
```

5.3.10 Summary: Animations and Transitions

Feature	Configuration Property	Description
Animation duration	`animation.duration`	How long animation runs (ms)
Easing function	`animation.easing`	Controls animation smoothness
Animate specific props	`animations.x`, `animations.y`	Customize duration/easing per axis
Trigger animation	Call `chart.update()`	Updates chart with animation

Animations help users visually track changes and improve engagement. Using Chart.js's flexible animation system, you can create polished, dynamic charts that respond smoothly to data updates.

Next, we'll cover how to make charts **responsive and adaptive** to different screen sizes

and devices.

5.4 Responsive and Adaptive Charts

Creating charts that look great and remain readable across devices—from large desktop monitors to small mobile screens—is crucial. Chart.js includes built-in **responsive design** features to help your charts adapt automatically to different viewport sizes, but as a developer, you can fine-tune this behavior for the best results.

This section explains how Chart.js handles responsiveness and shows how you can improve chart display on smaller screens using configuration options like `responsive`, `maintainAspectRatio`, and programmatic resizing.

5.4.1 Chart.js Responsive Basics

By default, Chart.js charts are **responsive**. This means they automatically resize to fit the size of their parent container when the browser window changes size.

```
options: {
  responsive: true   // default is true
}
```

Setting `responsive: true` makes the chart listen to window resize events and redraw accordingly.

5.4.2 Controlling Aspect Ratio

Sometimes, automatic resizing can distort your chart's shape. To control this, use:

```
options: {
  maintainAspectRatio: true  // default is true
}
```

- When `true`, the chart keeps its initial width-to-height ratio as it scales.
- When `false`, the chart fills the container fully, adjusting width and height independently.

5.4.3 Example: Maintaining Aspect Ratio

```
options: {
  responsive: true,
```

```
  maintainAspectRatio: true
}
```

This keeps charts proportional, preventing them from becoming stretched or squished on resize.

5.4.4 Example: Flexible Height for Mobile

```
options: {
  responsive: true,
  maintainAspectRatio: false
}
```

This allows the chart height to shrink on narrow mobile screens, fitting the container better.

5.4.5 Desktop vs Mobile Layouts

On desktops, charts often have ample space and can maintain fixed aspect ratios. On mobile devices, smaller width calls for adaptive height or simplified content.

You can dynamically adjust chart options depending on screen size using JavaScript:

```
function getChartOptions() {
  if (window.innerWidth < 600) {
    return {
      responsive: true,
      maintainAspectRatio: false,
      plugins: { legend: { display: false } }  // hide legend on small screens
    };
  } else {
    return {
      responsive: true,
      maintainAspectRatio: true,
      plugins: { legend: { display: true, position: 'top' } }
    };
  }
}

const chart = new Chart(ctx, {
  type: 'bar',
  data: data,
  options: getChartOptions()
});
```

You can listen for `resize` events and update the chart configuration dynamically as needed.

5.4.6 Resizing Programmatically

If your chart's container size changes due to UI interactions (like a sidebar toggle), you can manually call:

```
chart.resize();
```

This forces Chart.js to recalculate dimensions and redraw the chart to fit the new container size.

5.4.7 Styling for Small Screens

Consider simplifying visual elements on small devices:

- Reduce or hide legends and tooltips
- Use larger font sizes for readability (`options.plugins.legend.labels.font.size`)
- Decrease point radius or bar thickness
- Adjust axis tick labels (rotate or abbreviate)

5.4.8 Example: Responsive Chart with Adaptive Legend

```
options: {
  responsive: true,
  maintainAspectRatio: false,
  plugins: {
    legend: {
      display: window.innerWidth > 480,
      position: 'top'
    }
  },
  scales: {
    x: {
      ticks: {
        maxRotation: window.innerWidth > 480 ? 0 : 45,
        minRotation: window.innerWidth > 480 ? 0 : 45
      }
    }
  }
}
```

This example hides the legend and rotates X-axis labels on narrow screens to save space.

Full runnable code:

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

```
  <title>Responsive Chart with Adaptive Legend</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body, html {
      margin: 0; padding: 0; height: 100%;
      font-family: Arial, sans-serif;
    }
    #chartContainer {
      height: 80vh; /* flexible height */
      width: 90vw;
      max-width: 900px;
      margin: 2rem auto;
    }
    canvas {
      width: 100% !important;
      height: 100% !important;
    }
  </style>
</head>
<body>

  <div id="chartContainer">
    <canvas id="responsiveChart"></canvas>
  </div>

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

    function createChartOptions() {
      const showLegend = window.innerWidth > 480;
      const rotation = showLegend ? 0 : 45;
      return {
        responsive: true,
        maintainAspectRatio: false,
        plugins: {
          legend: {
            display: showLegend,
            position: 'top'
          },
          title: {
            display: true,
            text: 'Responsive Chart with Adaptive Legend & Tick Rotation'
          }
        },
        scales: {
          x: {
            ticks: {
              maxRotation: rotation,
              minRotation: rotation
            },
            title: {
              display: true,
              text: 'Month'
            }
          },
          y: {
            beginAtZero: true,
            title: {
```

```
            display: true,
            text: 'Value'
          }
        }
      }
    };
  }

  let chart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug'],
      datasets: [{
        label: 'Sales',
        data: [12, 19, 10, 15, 25, 22, 18, 30],
        backgroundColor: '#3b82f6'
      }]
    },
    options: createChartOptions()
  });

  // Update chart options on window resize
  window.addEventListener('resize', () => {
    chart.options = createChartOptions();
    chart.update();
  });
</script>

</body>
</html>
```

5.4.9 Summary: Responsive and Adaptive Charts

Feature	Description	Key Property / Method
Enable responsive layout	Chart resizes with container/window	`responsive: true` (default)
Maintain width/height ratio	Preserve aspect ratio on resize	`maintainAspectRatio: true/false`
Adapt to screen size	Customize options dynamically per viewport size	Use JS to update `options`
Manual resizing	Trigger chart redraw after container size change	`chart.resize()`
Simplify UI on small devices	Hide legends, rotate labels, adjust font size	Customize `plugins.legend` and `scales`

By understanding and controlling Chart.js's responsive features, you can create visualizations that are **beautiful, usable, and readable** on any device—maximizing your audience reach and engagement.

In the next chapter, we'll explore **plugin development and extending Chart.js** for truly customized charting experiences.

Chapter 6.

Working with Data

1. Dynamic Data Updates
2. Loading External Data (JSON, API)
3. Filtering and Transforming Data for Charts
4. Real-Time Charting with Live Data

6 Working with Data

6.1 Dynamic Data Updates

One of the powerful features of Chart.js is the ability to **update your charts dynamically** after they have been rendered. This is essential for interactive dashboards, live data feeds, or any situation where the data changes over time.

Chart.js provides a straightforward API to modify the data and re-render the chart without recreating it from scratch. Key methods include updating the `chart.data` object and calling `chart.update()` to reflect changes on the screen.

6.1.1 How Dynamic Updates Work

After you create a chart instance, the chart's data is stored inside the `chart.data` object, which contains:

- `datasets` — An array of datasets shown on the chart.
- `labels` — The labels for the chart's axes (often the X-axis).

To update or add data dynamically, you modify these properties and then call:

```
chart.update();
```

This triggers Chart.js to animate and redraw the chart based on the new data.

6.1.2 Common Data Update Methods

- **Add a new data point**: Append a value to a dataset's `data` array and add a corresponding label.
- **Add a new dataset**: Push a new dataset object to `chart.data.datasets`.
- **Remove or modify data**: Remove or change entries within `chart.data.datasets` or `chart.data.labels`.

6.1.3 Practical Example: Adding Data Points on Button Click

Let's create a simple line chart where clicking a button adds a new point with random data.

6.1.4 HTML Setup

```
<canvas id="lineChart" width="600" height="400"></canvas>
<button id="addDataBtn">Add Data Point</button>
```

6.1.5 JavaScript Setup

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

const lineChart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['January', 'February', 'March', 'April'],
    datasets: [{
      label: 'Sales',
      data: [65, 59, 80, 81],
      borderColor: 'rgba(75, 192, 192, 1)',
      fill: false,
      tension: 0.3
    }]
  },
  options: {
    responsive: true,
    animation: {
      duration: 500,
      easing: 'easeOutQuad'
    }
  }
});

// Function to generate a random sales value between 50 and 100
function getRandomSales() {
  return Math.floor(Math.random() * 50) + 50;
}

// Function to generate next month label
const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August'];
let nextMonthIndex = 4;

document.getElementById('addDataBtn').addEventListener('click', () => {
  if (nextMonthIndex >= months.length) {
    alert('No more months available');
    return;
  }

  // Add new label
  lineChart.data.labels.push(months[nextMonthIndex]);

  // Add new data point to the first dataset
  lineChart.data.datasets[0].data.push(getRandomSales());

  nextMonthIndex++;
```

```
  // Update chart to reflect changes
  lineChart.update();
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Line Chart: Add Data Points</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f3f4f6;
    }
    canvas {
      display: block;
      max-width: 700px;
      height: 400px;
      margin-bottom: 1rem;
    }
    button {
      padding: 0.6rem 1.2rem;
      font-size: 1rem;
      background-color: #3b82f6;
      border: none;
      color: white;
      border-radius: 5px;
      cursor: pointer;
    }
    button:hover {
      background-color: #2563eb;
    }
  </style>
</head>
<body>

  <canvas id="lineChart" width="600" height="400"></canvas>
  <button id="addDataBtn">Add Data Point</button>

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

    const lineChart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['January', 'February', 'March', 'April'],
        datasets: [{
          label: 'Sales',
          data: [65, 59, 80, 81],
          borderColor: 'rgba(75, 192, 192, 1)',
          fill: false,
          tension: 0.3
        }]
```

```
    },
    options: {
      responsive: true,
      animation: {
        duration: 500,
        easing: 'easeOutQuad'
      },
      plugins: {
        title: {
          display: true,
          text: 'Sales Over Months'
        }
      },
      scales: {
        y: {
          beginAtZero: true,
          title: {
            display: true,
            text: 'Sales Value'
          }
        },
        x: {
          title: {
            display: true,
            text: 'Month'
          }
        }
      }
    }
  });

  // Generate random sales value between 50 and 100
  function getRandomSales() {
    return Math.floor(Math.random() * 50) + 50;
  }

  const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August'];
  let nextMonthIndex = 4;

  document.getElementById('addDataBtn').addEventListener('click', () => {
    if (nextMonthIndex >= months.length) {
      alert('No more months available');
      return;
    }

    // Add new label
    lineChart.data.labels.push(months[nextMonthIndex]);

    // Add new data point
    lineChart.data.datasets[0].data.push(getRandomSales());

    nextMonthIndex++;

    // Update the chart with animation
    lineChart.update();
  });
</script>
```

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

6.1.6 Explanation

- The chart initializes with 4 months of sales data.
- When the **Add Data Point** button is clicked:
  - A new label (month) is added to `lineChart.data.labels`.
  - A random sales number is pushed into the dataset's `data` array.
  - Calling `lineChart.update()` redraws the chart with a smooth animation.
- The example limits the additions to the predefined `months` array.

6.1.7 Summary: Dynamic Data Updates

Action	Chart.js Method or Property
Add a new label	`chart.data.labels.push('New Label')`
Add data point to dataset	`chart.data.datasets[index].data.push(value)`
Add a new dataset	`chart.data.datasets.push(newDatasetObject)`
Reflect changes on chart	`chart.update()`

With these tools, you can build charts that respond fluidly to user input or real-time data changes, creating engaging and interactive visualizations.

6.2 Loading External Data (JSON, API)

In real-world applications, data often comes from external sources such as JSON files or REST APIs. Chart.js can easily integrate with these sources by fetching data asynchronously, transforming it as needed, and rendering charts dynamically.

This section guides you through loading external data using JavaScript's native `fetch()` API, mapping the data to Chart.js datasets, and handling asynchronous rendering.

6.2.1 Fetching Data with `fetch()`

The `fetch()` function allows you to make HTTP requests and retrieve data asynchronously in JSON or other formats.

6.2.2 Basic Fetch Example

```
fetch('data/sales.json')
  .then(response => response.json())
  .then(data => {
    console.log(data); // Inspect fetched JSON
  })
  .catch(error => console.error('Error fetching data:', error));
```

6.2.3 Mapping External Data to Chart.js

Suppose the external JSON data looks like this:

```
{
  "labels": ["January", "February", "March", "April"],
  "sales": [120, 150, 170, 140]
}
```

You can use this data to populate Chart.js:

```
fetch('data/sales.json')
  .then(response => response.json())
  .then(data => {
    const chartData = {
      labels: data.labels,
      datasets: [{
        label: 'Sales',
        data: data.sales,
        borderColor: 'rgba(54, 162, 235, 1)',
        backgroundColor: 'rgba(54, 162, 235, 0.2)',
        fill: true,
        tension: 0.3
      }]
    };

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, {
      type: 'line',
      data: chartData,
      options: {
        responsive: true
      }
    });
  });
```

6.2.4 Handling Asynchronous Rendering

Because fetching data is asynchronous, you must create or update the chart **inside the `.then()` callback** after data is loaded.

6.2.5 Option 1: Create Chart After Fetch

Create the chart only once data is available, as shown above.

6.2.6 Option 2: Create Chart First, Update Later

Alternatively, initialize an empty chart and update it after fetching data:

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

const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: [],
    datasets: [{
      label: 'Sales',
      data: [],
      backgroundColor: 'rgba(75, 192, 192, 0.5)'
    }]
  },
  options: { responsive: true }
});

fetch('data/sales.json')
  .then(res => res.json())
  .then(data => {
    chart.data.labels = data.labels;
    chart.data.datasets[0].data = data.sales;
    chart.update();
  });
```

This approach is useful when you want the chart structure upfront, perhaps showing a loading spinner or placeholder.

6.2.7 Fetching Data from REST APIs

The process is the same for REST APIs that return JSON. For example:

```
fetch('https://api.example.com/sales')
  .then(response => response.json())
  .then(apiData => {
```

```
    // Assume apiData has similar structure or map it accordingly
    chart.data.labels = apiData.months;
    chart.data.datasets[0].data = apiData.values;
    chart.update();
  });
```

Make sure to adjust the mapping depending on your API's response format.

6.2.8 Error Handling

Always handle network errors gracefully to improve user experience:

```
fetch('data/sales.json')
  .then(res => {
    if (!res.ok) throw new Error('Network response was not ok');
    return res.json();
  })
  .then(data => { /* use data */ })
  .catch(error => {
    console.error('Fetch error:', error);
    alert('Failed to load data.');
  });
```

6.2.9 Summary: Loading External Data

Step	Key Code or Concept
Fetch JSON or API data	`fetch(url).then(res => res.json())`
Map data to Chart.js	Set `chart.data.labels` and `chart.data.datasets`
Create or update chart after data	Initialize chart inside `.then()` or update existing chart with `.update()`
Handle errors gracefully	Use `.catch()` and check `response.ok`

With these techniques, you can seamlessly integrate Chart.js with external data sources, making your charts dynamic and connected to live or stored data.

Next, we'll explore how to **filter and transform data** before visualizing it, enhancing clarity and insight.

6.3 Filtering and Transforming Data for Charts

Raw data from sources like APIs or databases often requires **filtering**, **sorting**, or **transforming** before it's ready for visualization. Chart.js expects data in a specific format—usually arrays of labels and matching datasets—so preparing your data correctly ensures your charts are meaningful and accurate.

This section shows common techniques to manipulate raw data using JavaScript and demonstrates filtering transactions by month and sales above a threshold.

6.3.1 Why Filter and Transform Data?

- **Remove irrelevant data:** Focus on a specific time range or category.
- **Aggregate or group:** Summarize detailed data into meaningful buckets.
- **Format data:** Convert raw timestamps into readable labels.
- **Sort:** Display data in chronological or value order for clarity.

6.3.2 Example Dataset: Transactions

Consider an array of sales transactions with date and amount:

```
const transactions = [
  { date: '2024-01-15', amount: 200 },
  { date: '2024-02-05', amount: 150 },
  { date: '2024-01-25', amount: 300 },
  { date: '2024-03-10', amount: 50 },
  { date: '2024-02-20', amount: 400 },
  { date: '2024-03-15', amount: 100 }
];
```

6.3.3 Goal: Filter transactions to only show sales in February and March above 100, then aggregate by month for charting.

Step 1: Filter by Date and Amount

```
const filtered = transactions.filter(t => {
  const month = new Date(t.date).getMonth() + 1; // JS months 0-based
  return (month === 2 || month === 3) && t.amount > 100;
});
```

- We extract the month from the date.
- Keep only February (2) or March (3) sales above 100.

Step 2: Group and Sum Sales by Month

We want total sales per month:

```
const salesByMonth = filtered.reduce((acc, curr) => {
  const month = new Date(curr.date).toLocaleString('default', { month: 'short' });
  acc[month] = (acc[month] || 0) + curr.amount;
  return acc;
}, {});
```

Result:

```
// salesByMonth = { Feb: 550, Mar: 100 }
```

Step 3: Prepare Data for Chart.js

Extract labels and data arrays:

```
const labels = Object.keys(salesByMonth);       // ['Feb', 'Mar']
const data = Object.values(salesByMonth);       // [550, 100]
```

Final Chart Data Object

```
const chartData = {
  labels: labels,
  datasets: [{
    label: 'Filtered Sales',
    data: data,
    backgroundColor: ['#3b82f6', '#f59e0b'],
  }]
};
```

Putting It All Together in Chart.js

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

const chart = new Chart(ctx, {
  type: 'bar',
  data: chartData,
  options: {
    responsive: true,
    plugins: {
      legend: { display: false }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Filtered Sales Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
```

```
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    canvas {
      max-width: 600px;
      width: 100%;
      height: 400px;
    }
  </style>
</head>
<body>

  <canvas id="myChart"></canvas>

  <script>
    // Step 0: Dataset
    const transactions = [
      { date: '2024-01-15', amount: 200 },
      { date: '2024-02-05', amount: 150 },
      { date: '2024-01-25', amount: 300 },
      { date: '2024-03-10', amount: 50 },
      { date: '2024-02-20', amount: 400 },
      { date: '2024-03-15', amount: 100 }
    ];

    // Step 1: Filter by month (Feb or Mar) and amount > 100
    const filtered = transactions.filter(t => {
      const month = new Date(t.date).getMonth() + 1; // 1-based
      return (month === 2 || month === 3) && t.amount > 100;
    });

    // Step 2: Aggregate sales by month short name
    const salesByMonth = filtered.reduce((acc, curr) => {
      const month = new Date(curr.date).toLocaleString('default', { month: 'short' });
      acc[month] = (acc[month] || 0) + curr.amount;
      return acc;
    }, {});

    // Step 3: Extract labels and data arrays
    const labels = Object.keys(salesByMonth);    // e.g. ['Feb', 'Mar']
    const data = Object.values(salesByMonth);    // e.g. [550, 100]

    // Chart data object
    const chartData = {
      labels: labels,
      datasets: [{
        label: 'Filtered Sales',
        data: data,
        backgroundColor: ['#3b82f6', '#f59e0b']
      }]
    };

    // Chart initialization
```

```
    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: chartData,
      options: {
        responsive: true,
        plugins: {
          legend: { display: false },
          title: {
            display: true,
            text: 'Filtered Sales for Feb and Mar > $100'
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Sales Amount ($)'
            }
          },
          x: {
            title: {
              display: true,
              text: 'Month'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

6.3.4 Summary: Filtering and Transforming Data

Technique	Purpose	Example Method
Filtering	Remove unwanted data by condition	`.filter(item => condition)`
Grouping/Aggregating	Summarize by key (e.g., month, category)	`.reduce((acc, curr) => { ... })`
Formatting Labels	Convert raw dates/numbers to readable labels	`toLocaleString()`, `Date` methods
Sorting	Order data for clarity	`.sort((a, b) => a - b)`

Properly preparing data ensures your charts tell the right story. Filtering and transforming with JavaScript before feeding Chart.js will help you build clean, insightful visualizations

tailored to your audience.

Next, we'll explore **real-time charting** techniques to visualize live data streams.

6.4 Real-Time Charting with Live Data

Real-time charts enable you to visualize data that changes continuously, such as live sensor readings, stock prices, or user activity. Chart.js, combined with JavaScript's timing and networking features, makes it easy to build charts that update dynamically as new data arrives.

In this section, you'll learn how to implement real-time updating charts using either **polling with `setInterval`** or **WebSocket streams**, including how to add new data points while removing old ones for a smooth, live visualization.

6.4.1 Method 1: Polling with `setInterval`

Polling involves periodically fetching or generating new data and updating the chart at fixed intervals.

6.4.2 Example: Real-Time Line Chart with Random Data

```
<canvas id="realtimeChart" width="600" height="400"></canvas>
```

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

const maxPoints = 20; // maximum number of points shown on chart

const realtimeChart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: [], // timestamps or sequential labels
    datasets: [{
      label: 'Live Data',
      data: [],
      borderColor: 'rgba(75,192,192,1)',
      backgroundColor: 'rgba(75,192,192,0.2)',
      fill: true,
      tension: 0.3
    }]
  },
  options: {
    animation: { duration: 0 }, // disable animation for smooth updates
    responsive: true,
```

```
    scales: {
      x: {
        type: 'time',
        time: {
          unit: 'second',
          tooltipFormat: 'HH:mm:ss'
        }
      },
      y: { beginAtZero: true }
    }
  }
});

// Function to generate current timestamp label
function getCurrentTimeLabel() {
  return new Date().toLocaleTimeString();
}

// Function to generate new random data point
function generateRandomValue() {
  return Math.floor(Math.random() * 100);
}

// Update chart every second with new data point
setInterval(() => {
  const now = getCurrentTimeLabel();

  // Add new label and data
  realtimeChart.data.labels.push(now);
  realtimeChart.data.datasets[0].data.push(generateRandomValue());

  // Remove oldest data if exceeding max points
  if (realtimeChart.data.labels.length > maxPoints) {
    realtimeChart.data.labels.shift();
    realtimeChart.data.datasets[0].data.shift();
  }

  // Update chart without animation
  realtimeChart.update('none');
}, 1000);
```

6.4.3 Explanation

- We start with empty data arrays.
- Every second, a new timestamp label and random value are added.
- If the data exceeds `maxPoints`, the oldest point is removed to keep the chart size manageable.
- Animations are disabled (`update('none')`) for smooth real-time updates.

6.4.4 Method 2: Real-Time Data via WebSockets

For truly live streaming data, WebSockets provide a persistent connection allowing the server to push data instantly.

6.4.5 Example: Listening for Data and Updating Chart

```
const socket = new WebSocket('wss://example.com/live-data');

socket.addEventListener('message', event => {
  const incomingData = JSON.parse(event.data); // Expecting { time: 'HH:mm:ss', value: number }

  realtimeChart.data.labels.push(incomingData.time);
  realtimeChart.data.datasets[0].data.push(incomingData.value);

  if (realtimeChart.data.labels.length > maxPoints) {
    realtimeChart.data.labels.shift();
    realtimeChart.data.datasets[0].data.shift();
  }

  realtimeChart.update('none');
});
```

6.4.6 Explanation

- The client connects to the WebSocket server.
- Whenever the server sends a message, the chart adds the new data point and label.
- Old data is removed once max points are exceeded.
- The chart updates instantly without animation for smoothness.

6.4.7 Tips for Real-Time Charting

Tip	Description
Limit data points	Avoid performance issues by capping displayed points
Disable animations during updates	Use `update('none')` for smoother, faster redraws
Use time-based scales for X-axis	`type: 'time'` helps with timestamp labeling
Handle disconnections gracefully	Add retry logic or fallbacks for WebSocket disconnects
Debounce or throttle updates	Control update frequency for smoother UX and less CPU

6.4.8 Summary: Real-Time Charting

Technique	How It Works	Chart.js Methods & Concepts
Polling	Use `setInterval` to fetch/generate data	`chart.data.labels.push()`, `chart.update()`
WebSocket Streaming	Receive pushed data from server	`WebSocket` API, update chart inside message handler
Manage data size	Remove old points to keep chart performant	`Array.shift()` to drop oldest data
Optimize rendering	Disable animations during frequent updates	`chart.update('none')`

Real-time charting adds interactivity and immediacy to your visualizations, empowering users to monitor dynamic data effortlessly. With Chart.js and JavaScript's event and timing APIs, you can build smooth, live-updating charts tailored to your needs.

Next, we will explore **plugin development and extending Chart.js** to customize behavior beyond built-in features.

Chapter 7.

Interactivity and Plugins

1. Event Handling (Click, Hover)
2. Custom Tooltips and Callbacks
3. Using and Creating Plugins
4. Zooming and Panning

7 Interactivity and Plugins

7.1 Event Handling (Click, Hover)

Interactivity is key to engaging and insightful data visualizations. Chart.js offers built-in support for event handling, letting you respond to user actions such as clicks and mouse hovers on chart elements. By leveraging `onClick` and `onHover` handlers, you can create dynamic charts that reveal detailed information, highlight related content, or trigger custom behaviors.

7.1.1 Using `onClick` to Respond to User Clicks

The `onClick` event fires whenever a user clicks on the chart canvas. You can determine which chart element (bar, point, slice, etc.) was clicked and respond accordingly.

7.1.2 Basic `onClick` Example: Show Data Details on Bar Click

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

const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Apples', 'Bananas', 'Cherries'],
    datasets: [{
      label: 'Fruit Sales',
      data: [12, 19, 7],
      backgroundColor: ['red', 'yellow', 'purple']
    }]
  },
  options: {
    onClick: (event, elements) => {
      if (elements.length > 0) {
        // Get first clicked element
        const element = elements[0];
        const datasetIndex = element.datasetIndex;
        const index = element.index;

        const label = chart.data.labels[index];
        const value = chart.data.datasets[datasetIndex].data[index];

        alert(`You clicked on ${label}: ${value} units sold.`);
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js onClick Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #fafafa;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <canvas id="myChart"></canvas>

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

    const chart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Apples', 'Bananas', 'Cherries'],
        datasets: [{
          label: 'Fruit Sales',
          data: [12, 19, 7],
          backgroundColor: ['red', 'yellow', 'purple']
        }]
      },
      options: {
        responsive: true,
        onClick: (event, elements) => {
          if (elements.length > 0) {
            // Get first clicked element
            const element = elements[0];
            const datasetIndex = element.datasetIndex;
            const index = element.index;

            const label = chart.data.labels[index];
            const value = chart.data.datasets[datasetIndex].data[index];

            alert(`You clicked on ${label}: ${value} units sold.`);
          }
        }
      }
    });
  </script>

</body>
</html>
```

How it works:

- The callback receives `event` and an `elements` array.
- `elements` contains information about chart elements at the click position.
- You extract the dataset and data index to identify the clicked data point.
- Use this info to display details, update the page, or perform other actions.

7.1.3 Using `onHover` to Highlight Related Content

The `onHover` event fires when the user moves the mouse over the chart. You can change cursor style, highlight elements, or trigger changes elsewhere on the page.

7.1.4 Example: Change Cursor and Highlight on Hover

```
const chart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr'],
    datasets: [{
      label: 'Visitors',
      data: [100, 150, 130, 170],
      borderColor: 'blue',
      fill: false
    }]
  },
  options: {
    onHover: (event, elements) => {
      const canvas = event.native.target;
      if (elements.length) {
        canvas.style.cursor = 'pointer';  // Change cursor to pointer when hovering data points
      } else {
        canvas.style.cursor = 'default';
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js onHover Cursor Change</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
```

```
    }
    canvas {
      max-width: 600px;
      height: 400px;
      display: block;
      margin: 0 auto;
    }
  </style>
</head>
<body>

  <canvas id="lineChart"></canvas>

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

    const chart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr'],
        datasets: [{
          label: 'Visitors',
          data: [100, 150, 130, 170],
          borderColor: 'blue',
          fill: false,
          tension: 0.3,
          pointRadius: 6,
          pointHoverRadius: 8,
        }]
      },
      options: {
        responsive: true,
        onHover: (event, elements) => {
          const canvas = event.native.target;
          if (elements.length) {
            canvas.style.cursor = 'pointer'; // Pointer when hovering a point
          } else {
            canvas.style.cursor = 'default'; // Default otherwise
          }
        }
      }
    });
  </script>

</body>
</html>
```

7.1.5 Enhancing Hover Interaction

You can combine `onHover` with DOM manipulation. For example, highlighting a related HTML element elsewhere on the page:

```
options: {
  onHover: (event, elements) => {
```

```
    if (elements.length) {
      const index = elements[0].index;
      document.querySelectorAll('.detail-item').forEach((el, i) => {
        el.style.backgroundColor = (i === index) ? '#ffc' : '';
      });
    } else {
      document.querySelectorAll('.detail-item').forEach(el => el.style.backgroundColor = '');
    }
  }
}
```

Here, hovering over a data point highlights the matching `.detail-item` element.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js Hover Interaction with DOM Highlight</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      display: flex;
      gap: 2rem;
      max-width: 900px;
      margin: auto;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
    .details {
      flex: 1;
      border: 1px solid #ccc;
      border-radius: 8px;
      padding: 1rem;
      background: #fff;
      max-height: 400px;
      overflow-y: auto;
    }
    .detail-item {
      padding: 0.5rem;
      margin-bottom: 0.5rem;
      border-radius: 4px;
      transition: background-color 0.3s;
      cursor: default;
    }
  </style>
</head>
<body>

  <canvas id="lineChart"></canvas>
```

```
  <div class="details">
    <h3>Visitor Details</h3>
    <div class="detail-item">Jan: 100 visitors</div>
    <div class="detail-item">Feb: 150 visitors</div>
    <div class="detail-item">Mar: 130 visitors</div>
    <div class="detail-item">Apr: 170 visitors</div>
  </div>

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

    const chart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: ['Jan', 'Feb', 'Mar', 'Apr'],
        datasets: [{
          label: 'Visitors',
          data: [100, 150, 130, 170],
          borderColor: 'blue',
          fill: false,
          tension: 0.3,
          pointRadius: 6,
          pointHoverRadius: 8,
        }]
      },
      options: {
        responsive: true,
        onHover: (event, elements) => {
          if (elements.length) {
            const index = elements[0].index;
            document.querySelectorAll('.detail-item').forEach((el, i) => {
              el.style.backgroundColor = (i === index) ? '#ffc' : '';
            });
            event.native.target.style.cursor = 'pointer';
          } else {
            document.querySelectorAll('.detail-item').forEach(el => el.style.backgroundColor = '');
            event.native.target.style.cursor = 'default';
          }
        }
      }
    });
  </script>

</body>
</html>
```

7.1.6 Summary of Event Parameters

Parameter	Description
`event`	The native DOM event object
`elements`	Array of chart elements under the event point

`elements` is empty if the event does not target any chart data element.

7.1.7 Summary: Event Handling in Chart.js

Feature	Usage
`onClick`	React to clicks on bars, points, slices to show info or trigger actions
`onHover`	Detect mouse movement over chart elements to highlight or change cursor
Access elements	Use `elements[0].datasetIndex` and `.index` to identify the data item
Combine with DOM	Use event callbacks to update or highlight related page elements

Using `onClick` and `onHover`, you can transform static charts into interactive, user-friendly visualizations that respond intelligently to user input—making your data stories clearer and more compelling.

Next, we'll explore **custom tooltips and callbacks** to further enhance interactivity and information delivery.

7.2 Custom Tooltips and Callbacks

Tooltips are essential for providing contextual information about data points in your charts. Chart.js offers a flexible **callbacks API** that lets you customize every aspect of tooltips — from labels and titles to colors and formatting — based on the data or the user's interaction.

This section shows you how to use the `options.plugins.tooltip.callbacks` object to tailor tooltips dynamically.

7.2.1 Tooltip Callbacks Overview

The most commonly customized tooltip callbacks include:

Callback	Purpose
`label`	Customize the text for each data point
`title`	Customize the tooltip title text
`afterLabel`	Add additional lines after the label
`labelColor`	Change the color box next to the label
`footer`	Add a footer section to the tooltip

Callback	Purpose

7.2.2 Example: Customizing Tooltip Labels and Titles

Suppose you have a bar chart showing sales and want the tooltip to display the value formatted as currency, and the title to show a custom prefix.

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

const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Apples', 'Bananas', 'Cherries'],
    datasets: [{
      label: 'Sales',
      data: [500, 300, 400],
      backgroundColor: ['#ff6384', '#36a2eb', '#cc65fe']
    }]
  },
  options: {
    plugins: {
      tooltip: {
        callbacks: {
          // Customize the tooltip title
          title: (context) => {
            const label = context[0].label;
            return `Product: ${label}`;
          },

          // Customize the label text with currency formatting
          label: (context) => {
            const value = context.parsed.y || context.parsed; // Support for different chart types
            return `Revenue: $${value.toLocaleString()}`;
          },

          // Add extra info after the label
          afterLabel: (context) => {
            return 'Includes taxes and fees';
          }
        }
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom Tooltip Labels Example</title>
```

```
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f7fafc;
      display: flex;
      justify-content: center;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <canvas id="myChart"></canvas>

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

    const chart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Apples', 'Bananas', 'Cherries'],
        datasets: [{
          label: 'Sales',
          data: [500, 300, 400],
          backgroundColor: ['#ff6384', '#36a2eb', '#cc65fe']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            callbacks: {
              // Tooltip title with custom prefix
              title: (context) => {
                const label = context[0].label;
                return `Product: ${label}`;
              },

              // Label with currency formatting
              label: (context) => {
                // context.parsed can be a number or object (for bar charts y value)
                const value = (typeof context.parsed === 'object') ? context.parsed.y : context.parsed;
                return `Revenue: $${value.toLocaleString()}`;
              },

              // Extra info line below label
              afterLabel: () => 'Includes taxes and fees'
            }
          }
        }
      }
    });
  </script>
```

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

7.2.3 Example: Dynamic Tooltip Colors

You can also customize the color indicator shown next to each tooltip label by returning a color object in the `labelColor` callback.

```
tooltip: {
  callbacks: {
    labelColor: (context) => {
      const value = context.parsed.y || context.parsed;
      // Show green if sales > 400, else red
      return {
        borderColor: value > 400 ? 'green' : 'red',
        backgroundColor: value > 400 ? 'lightgreen' : 'pink'
      };
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Dynamic Tooltip Colors Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 2rem;
      background: #f9fafb;
      display: flex;
      justify-content: center;
    }
    canvas {
      max-width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>

  <canvas id="myChart"></canvas>

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

    const chart = new Chart(ctx, {
      type: 'bar',
      data: {
```

```
        labels: ['Apples', 'Bananas', 'Cherries', 'Dates'],
        datasets: [{
          label: 'Sales',
          data: [500, 300, 400, 450],
          backgroundColor: ['#ff6384', '#36a2eb', '#cc65fe', '#ffa500']
        }]
      },
      options: {
        responsive: true,
        plugins: {
          tooltip: {
            callbacks: {
              label: (context) => {
                const value = typeof context.parsed === 'object' ? context.parsed.y : context.parsed;
                return `Revenue: $${value.toLocaleString()}`;
              },
              labelColor: (context) => {
                const value = typeof context.parsed === 'object' ? context.parsed.y : context.parsed;
                return {
                  borderColor: value > 400 ? 'green' : 'red',
                  backgroundColor: value > 400 ? 'lightgreen' : 'pink'
                };
              }
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

7.2.4 Additional Tips for Tooltip Customization

- Use `context.dataset` and `context.dataIndex` inside callbacks to access dataset-specific properties.
- You can format numbers, dates, or append units for better readability.
- Customize the tooltip footer or multiple lines to provide richer information.
- Combine tooltip callbacks with dataset properties to make tooltips highly dynamic.

7.2.5 Summary: Custom Tooltips with Callbacks

Callback	Description
`title`	Customize the heading of the tooltip
`label`	Customize the main label text for each point

Callback	Description
`afterLabel`	Add extra lines after the label
`labelColor`	Change the color box (border and background)
Use `context`	Access data, dataset, and index for dynamic info

By mastering tooltip callbacks, you can create intuitive, data-rich tooltips that help users understand your charts at a glance, enhancing overall interactivity and user experience.

Next, we will explore how to **use and create plugins** to extend Chart.js functionality beyond its core features.

7.3 Using and Creating Plugins

Chart.js plugins provide a powerful way to extend and customize the behavior of your charts beyond the built-in features. Whether you want to add zooming functionality, display data labels, or create custom visual effects, plugins let you hook into the chart lifecycle and manipulate its rendering or data.

This section introduces how to use existing plugins and guides you through creating a simple custom plugin that draws a watermark overlay on your chart.

7.3.1 What Are Chart.js Plugins?

Plugins are JavaScript objects or classes that interact with Chart.js charts at specific lifecycle stages such as before drawing, after updating, or before datasets are rendered. They allow you to add, modify, or remove visual elements and behavior globally or per chart.

7.3.2 Using Existing Plugins

Many useful plugins are available in the Chart.js ecosystem. Here are two popular ones:

7.3.3 Zoom Plugin (`chartjs-plugin-zoom`)

- Adds zooming and panning support via mouse wheel, pinch, or drag.
- Easy to configure with options for zoom modes and limits.

Installation with NPM:

```
npm install chartjs-plugin-zoom
```

Usage:

```
import Chart from 'chart.js/auto';
import zoomPlugin from 'chartjs-plugin-zoom';

Chart.register(zoomPlugin);

const chart = new Chart(ctx, {
  // ... chart config ...
  options: {
    plugins: {
      zoom: {
        zoom: {
          wheel: { enabled: true },
          pinch: { enabled: true },
          mode: 'xy',
        }
      }
    }
  }
});
```

7.3.4 Data Labels Plugin (`chartjs-plugin-datalabels`)

- Displays labels directly on chart elements for better readability.
- Supports custom positioning, styling, and formatting.

Installation with NPM:

```
npm install chartjs-plugin-datalabels
```

Usage:

```
import Chart from 'chart.js/auto';
import ChartDataLabels from 'chartjs-plugin-datalabels';

Chart.register(ChartDataLabels);

const chart = new Chart(ctx, {
  // ... chart config ...
  options: {
    plugins: {
      datalabels: {
        color: 'white',
        anchor: 'end',
        align: 'top',
      }
    }
  }
});
```

7.3.5 Creating a Simple Custom Plugin

You can also write your own plugins to add unique customizations. Let's create a plugin that draws a semi-transparent watermark text over the chart canvas.

7.3.6 Step 1: Define the Plugin Object

```
const watermarkPlugin = {
  id: 'watermarkPlugin', // unique plugin ID

  beforeDraw(chart) {
    const ctx = chart.ctx;
    const width = chart.width;
    const height = chart.height;

    ctx.save();
    ctx.font = '48px Arial';
    ctx.fillStyle = 'rgba(0, 0, 0, 0.1)';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText('Demo Watermark', width / 2, height / 2);
    ctx.restore();
  }
};
```

7.3.7 Step 2: Register the Plugin

```
Chart.register(watermarkPlugin);
```

7.3.8 Step 3: Use It in a Chart

```
const chart = new Chart(ctx, {
  type: 'line',
  data: { /* your data */ },
  options: { /* your options */ },
  plugins: [watermarkPlugin] // optional, since registered globally
});
```

Full runnable code:

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

```
  <title>Chart.js Watermark Plugin</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      background: #f9fafb;
    }
    canvas {
      max-width: 700px;
      height: 400px;
    }
  </style>
</head>
<body>

<canvas id="myChart"></canvas>

<script>
  // Step 1: Define the plugin
  const watermarkPlugin = {
    id: 'watermarkPlugin',
    beforeDraw(chart) {
      const ctx = chart.ctx;
      const width = chart.width;
      const height = chart.height;

      ctx.save();
      ctx.font = 'bold 48px Arial';
      ctx.fillStyle = 'rgba(0, 0, 0, 0.07)';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.fillText('Demo Watermark', width / 2, height / 2);
      ctx.restore();
    }
  };

  // Step 2: Register globally (optional if you use it per chart)
  Chart.register(watermarkPlugin);

  // Step 3: Create the chart
  const ctx = document.getElementById('myChart').getContext('2d');

  const chart = new Chart(ctx, {
    type: 'line',
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
      datasets: [{
        label: 'Visitors',
        data: [120, 200, 150, 180, 220],
        borderColor: '#3b82f6',
        tension: 0.4,
        fill: false
      }]
    },
    options: {
      responsive: true,
      plugins: {
```

```
        legend: { position: 'top' }
      }
    },
    plugins: [watermarkPlugin] // Use it here (optional if globally registered)
  });
</script>

</body>
</html>
```

7.3.9 How It Works

- The `beforeDraw` hook runs before the chart renders.
- We access the canvas context (`ctx`) and chart dimensions.
- Using canvas APIs, we draw the watermark text centered with low opacity.
- `ctx.save()` and `ctx.restore()` ensure no side effects on other drawing.

7.3.10 Summary: Plugins in Chart.js

Aspect	Details
What	Extend/customize chart rendering and behavior
Use Existing Plugins	Register popular plugins like zoom, datalabels
Create Custom Plugins	Define objects with lifecycle hooks (e.g., `beforeDraw`)
Register Plugins	Use `Chart.register(plugin)` globally or per chart
Common Hooks	`beforeDraw`, `afterDraw`, `beforeDatasetsDraw`, etc.

Plugins empower you to tailor Chart.js exactly to your needs, whether by integrating community-built tools or adding your own unique features.

Next, we'll look at **zooming and panning** techniques to enhance data exploration in your charts.

7.4 Zooming and Panning

Exploring detailed data in your charts becomes intuitive with zooming and panning. The chartjs-plugin-zoom adds this powerful interactivity to Chart.js, enabling users to zoom in/out and pan across data, especially useful for dense or time-series charts.

This section guides you through installing, configuring, and using the plugin to add smooth

zoom and pan functionality.

7.4.1 Step 1: Install the Zoom Plugin

You can add the plugin via NPM or CDN.

7.4.2 Using NPM

```
npm install chartjs-plugin-zoom
```

7.4.3 Using CDN

Add this script tag in your HTML after including Chart.js:

```
<script src="https://cdn.jsdelivr.net/npm/chartjs-plugin-zoom@2.0.1/dist/chartjs-plugin-zoom.min.js"></s
```

7.4.4 Step 2: Register the Plugin

If you're using ES modules or bundlers:

```
import Chart from 'chart.js/auto';
import zoomPlugin from 'chartjs-plugin-zoom';

Chart.register(zoomPlugin);
```

If using CDN, the plugin is registered automatically.

7.4.5 Step 3: Configure Zoom and Pan in Chart Options

The plugin is enabled via the `options.plugins.zoom` configuration object.

Here's a simple example on a time-series line chart enabling both zoom and pan:

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

const chart = new Chart(ctx, {
  type: 'line',
  data: {
```

```
    labels: [/* array of date strings or timestamps */],
    datasets: [{
      label: 'Stock Price',
      data: [120, 130, 125, 140, 150, 160, 155],
      borderColor: 'blue',
      fill: false,
      tension: 0.2
    }]
  },
  options: {
    responsive: true,
    scales: {
      x: {
        type: 'time',
        time: {
          unit: 'day'
        }
      },
      y: {
        beginAtZero: false
      }
    },
    plugins: {
      zoom: {
        zoom: {
          wheel: {
            enabled: true  // Enable zooming with mouse wheel
          },
          pinch: {
            enabled: true  // Enable zooming on touch devices with pinch gesture
          },
          mode: 'x',       // Zoom only horizontally (x-axis)
        },
        pan: {
          enabled: true,   // Enable panning
          mode: 'x',       // Pan horizontally only
          modifierKey: 'ctrl'  // Optional: require Ctrl key to pan
        }
      }
    }
  }
});
```

7.4.6 How It Works

- **Zooming:** Users can zoom horizontally by scrolling the mouse wheel or pinching on touch screens.
- **Panning:** Drag the chart left or right while holding the `Ctrl` key (configurable via `modifierKey`).
- The zoom and pan are constrained to the x-axis (`mode: 'x'`), suitable for time-series navigation.

7.4.7 Optional: Adding Reset Zoom Button

You can programmatically reset zoom with:

```
document.getElementById('resetZoom').addEventListener('click', () => {
  chart.resetZoom();
});
```

Add a button in your HTML:

```
<button id="resetZoom">Reset Zoom</button>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart.js Zoom + Pan Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-plugin-zoom@2.0.1/dist/chartjs-plugin-zoom.min.js">
  <style>
    body {
      font-family: sans-serif;
      margin: 2rem;
    }
    canvas {
      max-width: 100%;
      height: 400px;
    }
    button {
      margin-top: 1rem;
    }
  </style>
</head>
<body>

<h2>Zoom & Pan Example (Ctrl+Drag or Scroll to Zoom)</h2>
<canvas id="myChart"></canvas>
<button id="resetZoom">Reset Zoom</button>

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

  const chart = new Chart(ctx, {
    type: 'line',
    data: {
      labels: [
        '2024-01-01', '2024-01-02', '2024-01-03',
        '2024-01-04', '2024-01-05', '2024-01-06',
        '2024-01-07', '2024-01-08'
      ],
      datasets: [{
        label: 'Stock Price',
        data: [120, 130, 125, 140, 150, 160, 155, 165],
        borderColor: 'blue',
        fill: false,
```

```
        tension: 0.2
      }]
    },
    options: {
      responsive: true,
      scales: {
        x: {
          type: 'time',
          time: {
            unit: 'day'
          },
          title: {
            display: true,
            text: 'Date'
          }
        },
        y: {
          beginAtZero: false,
          title: {
            display: true,
            text: 'Price'
          }
        }
      },
      plugins: {
        zoom: {
          zoom: {
            wheel: {
              enabled: true
            },
            pinch: {
              enabled: true
            },
            mode: 'x'
          },
          pan: {
            enabled: true,
            mode: 'x',
            modifierKey: 'ctrl'
          }
        },
        legend: {
          position: 'top'
        },
        tooltip: {
          enabled: true
        }
      }
    }
  });

  // Reset zoom button
  document.getElementById('resetZoom').addEventListener('click', () => {
    chart.resetZoom();
  });
</script>

</body>
```

```
</html>
```

7.4.8 Summary: Zoom and Pan Plugin Features

Feature	Description
Zoom via Wheel	Mouse wheel scroll zooms chart in/out
Pinch Zoom	Pinch gestures zoom on touch devices
Pan by Drag	Click and drag (optionally with modifier key) pans chart
Axis Modes	Control zoom/pan on x, y, or both axes
Programmatic Control	Methods like `resetZoom()` to manage zoom state

Adding zoom and pan transforms static charts into interactive data explorers, letting users dive into details or zoom out for an overview—all with a few lines of configuration using `chartjs-plugin-zoom`.

In the next chapter, we'll dive into **data management and dynamic updates** to handle live and changing datasets.

Chapter 8.

Styling and Theming

1. Global Chart Styles and Defaults
2. Custom Fonts and Colors
3. Gradients and Patterns
4. Dark Mode and Accessibility

8 Styling and Theming

8.1 Global Chart Styles and Defaults

When building multiple charts in an application, maintaining a consistent look and feel is crucial for a polished user experience. Chart.js makes this easy by allowing you to define **global default options** that apply to all charts unless overridden locally.

You can configure the global defaults through the `Chart.defaults` object to set fonts, colors, sizes, animation settings, and more — ensuring style consistency without repeating configuration in every chart.

8.1.1 How to Set Global Defaults

`Chart.defaults` contains default options for all chart types (`Chart.defaults`), as well as defaults scoped to specific chart types (e.g., `Chart.defaults.bar`, `Chart.defaults.line`).

You can modify these objects before creating any charts, and the settings will be applied globally.

8.1.2 Example 1: Set a Default Font Family and Size

```
// Set default font family and size for all charts
Chart.defaults.font.family = "'Roboto', 'Helvetica Neue', Arial, sans-serif";
Chart.defaults.font.size = 14;
```

This ensures all charts use the specified font and size unless individually overridden.

8.1.3 Example 2: Set Default Border Width for Bars and Line Width for Lines

```
// Default border width for bar charts
Chart.defaults.bar.borderWidth = 2;

// Default line width for line charts
Chart.defaults.line.borderWidth = 3;
```

These settings standardize the thickness of borders and lines, making charts visually consistent.

8.1.4 Example 3: Set Default Animation Duration

```
// Set animation duration globally to 1000ms (1 second)
Chart.defaults.animation.duration = 1000;
```

This will make all animations across charts last one second, creating a uniform feel.

8.1.5 Example 4: Set Default Tooltip Font Color

```
Chart.defaults.plugins.tooltip.bodyFont.color = '#333';
```

Adjust the tooltip text color globally for better readability.

8.1.6 Applying Global Defaults

```
// Apply global defaults before creating any charts
Chart.defaults.font.family = "'Open Sans', sans-serif";
Chart.defaults.font.size = 12;
Chart.defaults.plugins.legend.position = 'bottom';
Chart.defaults.animation.duration = 700;

// Then create charts as usual
const ctx = document.getElementById('myChart').getContext('2d');
const chart = new Chart(ctx, {
  type: 'line',
  data: { /* ... */ },
  options: { /* ... */ }
});
```

8.1.7 Why Use Global Defaults?

- **Consistency:** Keeps styling uniform across multiple charts.
- **Maintainability:** Change styles in one place instead of updating each chart.
- **Simplicity:** Reduce repetitive code by avoiding duplicate style declarations.

8.1.8 Summary: Key Global Defaults Properties

Property	Description
`Chart.defaults.font`	Controls font family, size, weight
`Chart.defaults.animation`	Controls animation duration, easing
`Chart.defaults.plugins.legend`	Controls legend position, labels styling
`Chart.defaults.plugins.tooltip`	Controls tooltip font, colors, callbacks
Chart type defaults (e.g., `bar`, `line`)	Specific defaults per chart type (e.g., border width)

Setting global styles with `Chart.defaults` empowers you to create a cohesive and professional visualization experience throughout your application with minimal effort.

Next, we'll explore **custom fonts and colors** to take your chart styling even further.

8.2 Custom Fonts and Colors

Customizing fonts and colors allows you to tailor your charts to fit a brand identity or design system perfectly. Chart.js provides flexible options to style labels, axes, datasets, and tooltips individually, giving you fine-grained control over the appearance.

In this section, we'll explore how to apply custom fonts and colors across different chart components with practical examples aligned to a hypothetical brand.

8.2.1 Applying Custom Fonts

You can specify fonts globally using `Chart.defaults.font`, or locally inside chart options for specific elements like axis labels or tooltips.

8.2.2 Example: Set a Custom Font for Axis Labels and Tooltips

```
const chart = new Chart(ctx, {
  type: 'bar',
  data: { /* ... */ },
  options: {
    scales: {
      x: {
        ticks: {
          font: {
            family: "'Montserrat', sans-serif",
            size: 14,
            weight: 'bold'
          },
```

```
          color: '#333' // font color for x-axis labels
        }
      },
      y: {
        ticks: {
          font: {
            family: "'Montserrat', sans-serif",
            size: 14,
            weight: 'bold'
          },
          color: '#333' // font color for y-axis labels
        }
      }
    },
    plugins: {
      tooltip: {
        bodyFont: {
          family: "'Open Sans', sans-serif",
          size: 12
        },
        titleFont: {
          family: "'Open Sans', sans-serif",
          weight: '600',
          size: 14
        }
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Chart.js with Custom Fonts</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

  <!-- Load Google Fonts -->
  <link href="https://fonts.googleapis.com/css2?family=Montserrat:wght@700&family=Open+Sans:wght@400;60

  <style>
    body {
      font-family: 'Open Sans', sans-serif;
      margin: 2rem;
    }
    canvas {
      max-width: 100%;
      height: 400px;
    }
  </style>
</head>
<body>

<h2>Chart.js with Custom Axis and Tooltip Fonts</h2>
<canvas id="myChart"></canvas>
```

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

  const chart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Apples', 'Bananas', 'Cherries'],
      datasets: [{
        label: 'Sales',
        data: [300, 500, 400],
        backgroundColor: ['#ff6384', '#36a2eb', '#cc65fe']
      }]
    },
    options: {
      scales: {
        x: {
          ticks: {
            font: {
              family: 'Montserrat',
              size: 14,
              weight: 'bold'
            },
            color: '#333'
          }
        },
        y: {
          ticks: {
            font: {
              family: 'Montserrat',
              size: 14,
              weight: 'bold'
            },
            color: '#333'
          }
        }
      },
      plugins: {
        tooltip: {
          bodyFont: {
            family: 'Open Sans',
            size: 12
          },
          titleFont: {
            family: 'Open Sans',
            weight: '600',
            size: 14
          }
        },
        legend: {
          labels: {
            font: {
              family: 'Open Sans',
              size: 13
            }
          }
        }
      }
    }
```

```
  });
</script>

</body>
</html>
```

8.2.3 Applying Custom Colors to Datasets

Customize dataset colors for backgrounds, borders, and points to match brand palettes.

8.2.4 Example: Brand Colors for Bars and Hover States

```
const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['Q1', 'Q2', 'Q3', 'Q4'],
    datasets: [{
      label: 'Revenue',
      data: [100, 200, 150, 250],
      backgroundColor: '#0052cc',      // Primary brand blue
      borderColor: '#003d99',          // Darker blue border
      borderWidth: 2,
      hoverBackgroundColor: '#0073e6' // Lighter blue on hover
    }]
  },
  options: { /* ... */ }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Brand Colors Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background: #f4f4f4;
      padding: 2rem;
    }
    canvas {
      max-width: 600px;
      margin: auto;
      background: white;
      box-shadow: 0 0 10px rgba(0,0,0,0.1);
      border-radius: 8px;
```

```
    }
  </style>
</head>
<body>

  <canvas id="revenueChart" width="600" height="400"></canvas>

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

    const chart = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Q1', 'Q2', 'Q3', 'Q4'],
        datasets: [{
          label: 'Revenue',
          data: [100, 200, 150, 250],
          backgroundColor: '#0052cc',          // Primary brand color
          borderColor: '#003d99',              // Border color
          borderWidth: 2,
          hoverBackgroundColor: '#0073e6',     // Lighter on hover
          hoverBorderColor: '#001a66',         // Optional hover border
          hoverBorderWidth: 2
        }]
      },
      options: {
        responsive: true,
        plugins: {
          legend: {
            labels: {
              color: '#333',
              font: {
                size: 14,
                weight: 'bold'
              }
            }
          },
          tooltip: {
            callbacks: {
              label: (context) => `Revenue: $${context.parsed.y}`
            }
          }
        },
        scales: {
          y: {
            beginAtZero: true,
            ticks: {
              color: '#333'
            },
            grid: {
              color: '#ddd'
            }
          },
          x: {
            ticks: {
              color: '#333'
            },
            grid: {
```

```
              color: '#eee'
            }
          }
        }
      }
    });
  </script>

</body>
</html>
```

8.2.5 Styling Axis Lines and Gridlines

Customize axis lines and gridlines to create a clean, branded look.

```
options: {
  scales: {
    x: {
      grid: {
        color: '#e0e0e0',   // Light gray grid lines
        borderColor: '#cccccc' // Slightly darker axis line
      },
      ticks: { color: '#555' }
    },
    y: {
      grid: {
        color: '#e0e0e0',
        borderColor: '#cccccc'
      },
      ticks: { color: '#555' }
    }
  }
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Styled Axis & Brand Colors Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: 'Segoe UI', sans-serif;
      background: #f9f9f9;
      padding: 2rem;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
```

```
    canvas {
      background: white;
      padding: 1rem;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>

<canvas id="myChart" width="600" height="400"></canvas>

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

  const myChart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Q1', 'Q2', 'Q3', 'Q4'],
      datasets: [{
        label: 'Revenue',
        data: [120, 190, 170, 220],
        backgroundColor: '#0052cc',          // Brand primary blue
        borderColor: '#003d99',
        borderWidth: 2,
        hoverBackgroundColor: '#0073e6',
        hoverBorderColor: '#001a66',
        hoverBorderWidth: 2
      }]
    },
    options: {
      responsive: true,
      plugins: {
        legend: {
          labels: {
            color: '#333',
            font: {
              size: 14,
              weight: 'bold'
            }
          }
        },
        tooltip: {
          callbacks: {
            label: (ctx) => `Revenue: $${ctx.parsed.y}`
          }
        }
      },
      scales: {
        x: {
          ticks: {
            color: '#555'
          },
          grid: {
            color: '#e0e0e0',           // Light gray gridlines
            borderColor: '#cccccc'      // Axis line
          }
        },
```

```
        y: {
          beginAtZero: true,
          ticks: {
            color: '#555'
          },
          grid: {
            color: '#e0e0e0',
            borderColor: '#cccccc'
          }
        }
      }
    }
  });
</script>

</body>
</html>
```

8.2.6 Customizing Tooltip Colors

Tooltips can be styled with custom background and text colors for brand consistency and readability.

```
options: {
  plugins: {
    tooltip: {
      backgroundColor: '#222', // Dark background
      titleColor: '#f0f0f0',   // Light title text
      bodyColor: '#ccc',       // Light body text
      borderColor: '#0073e6',  // Accent border color
      borderWidth: 1
    }
  }
}
```

8.2.7 Putting It All Together: Brand-Aligned Chart Example

```
const chart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    datasets: [{
      label: 'Active Users',
      data: [120, 150, 170, 160, 180],
      borderColor: '#ff6f61',  // Brand coral color
      backgroundColor: 'rgba(255, 111, 97, 0.2)',
      borderWidth: 3,
      pointBackgroundColor: '#ff6f61',
      pointRadius: 6
```

```
    }]
  },
  options: {
    scales: {
      x: {
        ticks: {
          font: {
            family: "'Lato', sans-serif",
            size: 13,
            weight: '600'
          },
          color: '#444'
        },
        grid: { color: '#f5f5f5' }
      },
      y: {
        ticks: {
          font: {
            family: "'Lato', sans-serif",
            size: 13,
            weight: '600'
          },
          color: '#444'
        },
        grid: { color: '#f5f5f5' }
      }
    },
    plugins: {
      tooltip: {
        backgroundColor: '#ff6f61',
        titleColor: '#fff',
        bodyColor: '#fff',
        borderColor: '#d9574a',
        borderWidth: 1,
        bodyFont: { family: "'Lato', sans-serif" }
      },
      legend: {
        labels: {
          font: {
            family: "'Lato', sans-serif",
            size: 14,
            weight: 'bold'
          },
          color: '#333'
        }
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
```

```
  <title>Brand Aligned Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <link href="https://fonts.googleapis.com/css2?family=Lato:wght@400;600;700&display=swap" rel="stylesh
  <style>
    body {
      font-family: 'Lato', sans-serif;
      background: #fafafa;
      margin: 0;
      padding: 2rem;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    canvas {
      background: white;
      padding: 1rem;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0,0,0,0.05);
    }
  </style>
</head>
<body>

<canvas id="myChart" width="700" height="400"></canvas>

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

  const chart = new Chart(ctx, {
    type: 'line',
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
      datasets: [{
        label: 'Active Users',
        data: [120, 150, 170, 160, 180],
        borderColor: '#ff6f61',  // Brand coral color
        backgroundColor: 'rgba(255, 111, 97, 0.2)',
        borderWidth: 3,
        pointBackgroundColor: '#ff6f61',
        pointRadius: 6,
        tension: 0.3,
        fill: true
      }]
    },
    options: {
      responsive: true,
      scales: {
        x: {
          ticks: {
            font: {
              family: "'Lato', sans-serif",
              size: 13,
              weight: '600'
            },
            color: '#444'
          },
          grid: { color: '#f5f5f5' }
```

```
        },
        y: {
          ticks: {
            font: {
              family: "'Lato', sans-serif",
              size: 13,
              weight: '600'
            },
            color: '#444'
          },
          grid: { color: '#f5f5f5' }
        }
      },
      plugins: {
        tooltip: {
          backgroundColor: '#ff6f61',
          titleColor: '#fff',
          bodyColor: '#fff',
          borderColor: '#d9574a',
          borderWidth: 1,
          titleFont: {
            family: "'Lato', sans-serif",
            weight: 'bold'
          },
          bodyFont: {
            family: "'Lato', sans-serif"
          }
        },
        legend: {
          labels: {
            font: {
              family: "'Lato', sans-serif",
              size: 14,
              weight: 'bold'
            },
            color: '#333'
          }
        }
      }
    }
  });
</script>

</body>
</html>
```

8.2.8 Summary: Custom Fonts and Colors

Element	How to Customize
Labels (Axes)	`scales.x.ticks.font`, `scales.y.ticks.font`, `color`
Dataset Colors	`backgroundColor`, `borderColor`, `hoverBackgroundColor`

Element	How to Customize
Points (Line/Scatter)	`pointBackgroundColor`, `pointRadius`, `pointStyle`
Tooltips	`plugins.tooltip` for fonts, colors, backgrounds
Legends	`plugins.legend.labels.font` and `color`

By thoughtfully applying fonts and colors aligned with your brand guidelines or design system, your charts become a seamless part of your overall user interface—both visually appealing and easy to read.

Next, we will explore how to use **gradients and patterns** to add texture and depth to your charts.

8.3 Gradients and Patterns

Adding gradients and patterns to your charts can create visually engaging and polished data visualizations. Chart.js leverages the underlying HTML `<canvas>` API, which allows you to create complex fills such as linear or radial gradients and repeating patterns.

In this section, you'll learn how to use canvas gradients and patterns to enhance chart backgrounds and dataset fills, making your charts stand out with rich color effects.

8.3.1 Using Canvas Gradients in Chart.js

Canvas gradients are created with the `CanvasRenderingContext2D` methods like `createLinearGradient()` and `createRadialGradient()`. You define color stops, then assign the gradient object as a fill style.

8.3.2 Example 1: Linear Gradient Fill for Bar Chart

```
<canvas id="barChart" width="600" height="400"></canvas>
```

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

// Create a vertical linear gradient from top (0) to bottom (400px)
const gradient = ctx.createLinearGradient(0, 0, 0, 400);
gradient.addColorStop(0, '#4facfe');   // Light blue at top
gradient.addColorStop(1, '#00f2fe');   // Cyan at bottom

const chart = new Chart(ctx, {
  type: 'bar',
```

```
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    datasets: [{
      label: 'Sales',
      data: [30, 45, 28, 50, 42],
      backgroundColor: gradient, // Use the gradient as fill
      borderColor: '#007bbd',
      borderWidth: 1
    }]
  },
  options: { responsive: true }
});
```

Here, the bars will be filled with a smooth vertical gradient transitioning from light blue to cyan.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Gradient Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f5f7fa;
      margin: 0;
      padding: 2rem;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    canvas {
      background: white;
      padding: 1rem;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0,0,0,0.05);
    }
  </style>
</head>
<body>

<canvas id="barChart" width="600" height="400"></canvas>

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

  // Create a vertical gradient from top (0) to bottom (400px)
  const gradient = ctx.createLinearGradient(0, 0, 0, 400);
  gradient.addColorStop(0, '#4facfe');   // Light blue at top
  gradient.addColorStop(1, '#00f2fe');   // Cyan at bottom

  const chart = new Chart(ctx, {
```

```
    type: 'bar',
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
      datasets: [{
        label: 'Sales',
        data: [30, 45, 28, 50, 42],
        backgroundColor: gradient, // Use the gradient as fill
        borderColor: '#007bbd',
        borderWidth: 1
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true,
          ticks: { color: '#333' },
          grid: { color: '#e0e0e0' }
        },
        x: {
          ticks: { color: '#333' },
          grid: { display: false }
        }
      },
      plugins: {
        legend: {
          labels: {
            color: '#333',
            font: {
              size: 14,
              weight: 'bold'
            }
          }
        }
      }
    }
  });
</script>

</body>
</html>
```

8.3.3 Example 2: Gradient Line Chart with Transparent Fill

```
<canvas id="lineChart" width="600" height="400"></canvas>
```

```
const ctx2 = document.getElementById('lineChart').getContext('2d');

// Create a vertical gradient for the line fill area
const gradientFill = ctx2.createLinearGradient(0, 0, 0, 400);
gradientFill.addColorStop(0, 'rgba(255, 99, 132, 0.5)');
gradientFill.addColorStop(1, 'rgba(255, 99, 132, 0)');

const chart2 = new Chart(ctx2, {
```

```
  type: 'line',
  data: {
    labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'],
    datasets: [{
      label: 'Visitors',
      data: [120, 150, 130, 180, 170, 160, 190],
      borderColor: '#ff6384',
      backgroundColor: gradientFill, // Gradient fill under the line
      fill: true,
      tension: 0.3
    }]
  },
  options: { responsive: true }
});
```

This creates a smooth red line with a gradient fill that fades from semi-transparent red to fully transparent, adding depth beneath the line.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Gradient Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f9fafc;
      margin: 0;
      padding: 2rem;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    canvas {
      background: #fff;
      border-radius: 10px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.05);
    }
  </style>
</head>
<body>

<canvas id="lineChart" width="600" height="400"></canvas>

<script>
  const ctx2 = document.getElementById('lineChart').getContext('2d');

  // Create a vertical gradient for the area below the line
  const gradientFill = ctx2.createLinearGradient(0, 0, 0, 400);
  gradientFill.addColorStop(0, 'rgba(255, 99, 132, 0.5)');
  gradientFill.addColorStop(1, 'rgba(255, 99, 132, 0)');
```

```
  const chart2 = new Chart(ctx2, {
    type: 'line',
    data: {
      labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'],
      datasets: [{
        label: 'Visitors',
        data: [120, 150, 130, 180, 170, 160, 190],
        borderColor: '#ff6384',
        backgroundColor: gradientFill,
        fill: true,
        tension: 0.3,
        pointBackgroundColor: '#ff6384',
        pointRadius: 5
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true,
          ticks: { color: '#333' },
          grid: { color: '#e0e0e0' }
        },
        x: {
          ticks: { color: '#333' },
          grid: { display: false }
        }
      },
      plugins: {
        legend: {
          labels: {
            color: '#333',
            font: {
              size: 14,
              weight: 'bold'
            }
          }
        },
        tooltip: {
          backgroundColor: '#ff6384',
          titleColor: '#fff',
          bodyColor: '#fff',
          borderColor: '#e55374',
          borderWidth: 1
        }
      }
    }
  });
</script>

</body>
</html>
```

8.3.4 Using Pattern Fills

Canvas patterns let you fill areas with repeating images or shapes. To use patterns, create an image or canvas pattern and assign it to the dataset's `backgroundColor`.

8.3.5 Example: Using a Simple Canvas Pattern for Bars

```
const patternCanvas = document.createElement('canvas');
patternCanvas.width = 20;
patternCanvas.height = 20;

const pctx = patternCanvas.getContext('2d');
pctx.fillStyle = '#007bbd';
pctx.fillRect(0, 0, 20, 20);
pctx.moveTo(0, 0);
pctx.lineTo(20, 20);
pctx.strokeStyle = '#00f2fe';
pctx.lineWidth = 2;
pctx.stroke();

const pattern = ctx.createPattern(patternCanvas, 'repeat');

const patternChart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ['A', 'B', 'C', 'D'],
    datasets: [{
      label: 'Pattern Fill',
      data: [15, 25, 20, 30],
      backgroundColor: pattern,
      borderColor: '#004080',
      borderWidth: 1
    }]
  }
});
```

This example creates a diagonal stripe pattern that repeats to fill each bar, adding texture.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart.js Pattern Fill Example</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: #f0f2f5;
      display: flex;
      justify-content: center;
      align-items: center;
```

```
      height: 100vh;
      margin: 0;
    }
    canvas {
      background: #fff;
      padding: 1rem;
      border-radius: 8px;
      box-shadow: 0 4px 12px rgba(0,0,0,0.1);
    }
  </style>
</head>
<body>

<canvas id="patternChart" width="600" height="400"></canvas>

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

  // Step 1: Create a custom pattern using an offscreen canvas
  const patternCanvas = document.createElement('canvas');
  patternCanvas.width = 20;
  patternCanvas.height = 20;

  const pctx = patternCanvas.getContext('2d');
  pctx.fillStyle = '#007bbd'; // solid blue background
  pctx.fillRect(0, 0, 20, 20);
  pctx.beginPath();
  pctx.moveTo(0, 0);
  pctx.lineTo(20, 20);
  pctx.strokeStyle = '#00f2fe'; // cyan diagonal stripe
  pctx.lineWidth = 2;
  pctx.stroke();

  // Step 2: Use it as a pattern fill in your chart
  const pattern = ctx.createPattern(patternCanvas, 'repeat');

  const patternChart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['A', 'B', 'C', 'D'],
      datasets: [{
        label: 'Pattern Fill',
        data: [15, 25, 20, 30],
        backgroundColor: pattern,
        borderColor: '#004080',
        borderWidth: 2
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true,
          ticks: { color: '#333' },
          grid: { color: '#ddd' }
        },
        x: {
          ticks: { color: '#333' },
```

```
          grid: { display: false }
        }
      },
      plugins: {
        legend: {
          labels: {
            font: {
              size: 14,
              weight: 'bold'
            },
            color: '#222'
          }
        },
        tooltip: {
          backgroundColor: '#007bbd',
          titleColor: '#fff',
          bodyColor: '#fff',
          borderColor: '#004080',
          borderWidth: 1
        }
      }
    }
  });
</script>

</body>
</html>
```

8.3.6 Tips for Using Gradients and Patterns

- **Create gradients and patterns after getting canvas context.** You need the context to create them.
- **Reuse gradient/pattern objects** for multiple datasets or charts for consistency.
- **Be mindful of performance**—complex patterns or large gradients can slow down rendering on low-end devices.
- **Combine with transparency** for subtle effects without overpowering chart data.

8.3.7 Summary: Applying Gradients and Patterns

Use Case	Method	Chart.js Property
Smooth color transitions	`ctx.createLinearGradient()` or `createRadialGradient()`	`backgroundColor`, `borderColor`
Textured fills	`ctx.createPattern(imageOrCanvas, repeatStyle)`	`backgroundColor`

Use Case	Method	Chart.js Property
Gradient fill under lines	Use gradient as `backgroundColor` with `fill: true` in line charts	`backgroundColor` and `fill`

By leveraging canvas gradients and patterns, you can create visually rich and brand-aligned charts that captivate users and elevate your data storytelling.

Next, we'll explore **dark mode** styling and accessibility best practices to make your charts usable and beautiful for everyone.

8.4 Dark Mode and Accessibility

Supporting **dark mode** and ensuring **accessibility** are essential steps toward creating inclusive, user-friendly data visualizations. Chart.js offers the flexibility to customize chart styles to fit dark themes, while accessibility best practices ensure your charts are clear and usable for all users, including those with visual impairments.

8.4.1 Supporting Dark Mode

Dark mode typically involves a dark background with lighter text and UI elements to reduce eye strain in low-light environments. To adapt your charts for dark mode, adjust colors thoughtfully:

8.4.2 Key Adjustments for Dark Mode

- **Background:** Use a dark or near-black background color instead of white.
- **Text:** Set labels, ticks, tooltips, and legends to light colors for high contrast.
- **Gridlines:** Use subtle but visible lighter gridlines to avoid overwhelming the chart.
- **Dataset Colors:** Choose colors that stand out against the dark background without causing glare.

8.4.3 Example Dark Mode Configuration

```
const darkModeOptions = {
  plugins: {
```

```
    legend: {
      labels: {
        color: '#EEE' // Light text for legend labels
      }
    },
    tooltip: {
      backgroundColor: '#222',
      titleColor: '#fff',
      bodyColor: '#ccc'
    }
  },
  scales: {
    x: {
      grid: {
        color: '#444' // Subtle gridlines
      },
      ticks: {
        color: '#ddd' // Light axis labels
      }
    },
    y: {
      grid: {
        color: '#444'
      },
      ticks: {
        color: '#ddd'
      }
    }
  },
  backgroundColor: '#121212' // Use in container or chart background as needed
};
```

You can toggle between light and dark mode by dynamically applying different option sets or by modifying `Chart.defaults` accordingly.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Chart.js Dark Mode Example</title>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<style>
  body {
    background-color: #121212;
    color: #eee;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
    font-family: Arial, sans-serif;
  }
  #chartContainer {
    background-color: #121212;
    padding: 20px;
```

```
    border-radius: 10px;
    box-shadow: 0 0 15px #000;
  }
  canvas {
    background-color: #1e1e1e;
    border-radius: 8px;
  }
</style>
</head>
<body>

<div id="chartContainer">
  <canvas id="darkModeChart" width="600" height="400"></canvas>
</div>

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

  const darkModeOptions = {
    plugins: {
      legend: {
        labels: {
          color: '#EEE' // Light text for legend labels
        }
      },
      tooltip: {
        backgroundColor: '#222',
        titleColor: '#fff',
        bodyColor: '#ccc'
      }
    },
    scales: {
      x: {
        grid: {
          color: '#444' // Subtle gridlines
        },
        ticks: {
          color: '#ddd' // Light axis labels
        }
      },
      y: {
        grid: {
          color: '#444'
        },
        ticks: {
          color: '#ddd'
        }
      }
    }
  };

  const data = {
    labels: ['Q1', 'Q2', 'Q3', 'Q4'],
    datasets: [{
      label: 'Revenue',
      data: [150, 200, 180, 220],
      backgroundColor: '#4caf50',
      borderColor: '#388e3c',
```

```
      borderWidth: 1
    }]
  };

  const darkModeChart = new Chart(ctx, {
    type: 'bar',
    data: data,
    options: {
      responsive: true,
      ...darkModeOptions
    }
  });
</script>

</body>
</html>
```

8.4.4 Accessibility Best Practices

Creating accessible charts ensures that all users, including those with disabilities, can interpret your data effectively.

8.4.5 Label Readability

- Use sufficiently large font sizes and high-contrast colors for labels and tooltips.
- Avoid using color alone to convey information; use shapes, patterns, or text labels where possible.

8.4.6 Color Contrast

- Ensure text and important elements meet WCAG contrast guidelines (minimum 4.5:1 contrast ratio).
- Test your chosen palette with tools like WebAIM Contrast Checker.

8.4.7 Keyboard and Focus Management

- Although Chart.js charts are canvas-based, consider adding accessible HTML elements (like a data table or summaries) to complement charts.
- Provide keyboard navigation for interacting with chart controls (e.g., zoom, filter

buttons).
- Use `aria-labels` and descriptive text to provide context for screen readers.

8.4.8 Example: Accessible Chart Container

```
<div role="region" aria-label="Monthly Sales Chart showing sales from January to June">
  <canvas id="salesChart"></canvas>
  <table>
    <caption>Monthly Sales Data</caption>
    <thead>
      <tr><th>Month</th><th>Sales</th></tr>
    </thead>
    <tbody>
      <tr><td>January</td><td>100</td></tr>
      <tr><td>February</td><td>120</td></tr>
      <!-- more rows -->
    </tbody>
  </table>
</div>
```

This provides textual data alongside the chart for screen readers and keyboard users.

8.4.9 Summary: Dark Mode and Accessibility Checklist

Aspect	Recommendation
Background & Text	Dark background, light text with high contrast
Gridlines	Use subtle lighter gridlines for clarity
Dataset Colors	Bright, distinguishable colors on dark bg
Font Sizes	Use readable font sizes (12-16px minimum)
Color Contrast	Follow WCAG 4.5:1 contrast ratio
Non-Color Coding	Supplement color with shapes, labels, patterns
Keyboard Accessibility	Provide alternative controls and focus management
Screen Reader Support	Add ARIA roles, labels, and accessible data tables

By thoughtfully implementing dark mode styles and prioritizing accessibility, you ensure your Chart.js visualizations are beautiful and usable for every audience, regardless of their environment or abilities.

Next, we'll explore how to extend styling even further with **custom gradients and interactive theming** techniques.

Chapter 9.

Combining Multiple Charts

1. Mixed Chart Types (e.g., Bar + Line)
2. Dashboard Layouts with Multiple Charts
3. Synchronizing Interactions Across Charts

9 Combining Multiple Charts

9.1 Mixed Chart Types (e.g., Bar + Line)

Combining multiple chart types into a single visualization is a powerful way to convey complex data relationships clearly and intuitively. Chart.js supports **mixed charts** that let you overlay different types—such as bars and lines—on the same canvas, often sharing the same or dual axes.

This approach is ideal for comparing related but differently scaled metrics, such as sales volume (bar) alongside revenue trends (line).

9.1.1 How to Create a Mixed Chart

The core idea is to specify multiple datasets in the `data.datasets` array, each with its own `type` property defining the chart style (e.g., `'bar'`, `'line'`). You can also assign datasets to different y-axes by configuring the `yAxisID` property.

9.1.2 Example: Sales Volume (Bar) Revenue Trend (Line)

Imagine you want to visualize monthly sales volume as bars and monthly revenue as a line over the same period.

```
<canvas id="mixedChart" width="700" height="400"></canvas>
```

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

const mixedChart = new Chart(ctx, {
  type: 'bar', // Default chart type for the first dataset (can be overridden)
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
    datasets: [
      {
        type: 'bar', // Sales volume as bars
        label: 'Sales Volume',
        data: [500, 700, 800, 600, 900, 750],
        backgroundColor: '#4caf50',
        yAxisID: 'yVolume'
      },
      {
        type: 'line', // Revenue as line
        label: 'Revenue ($)',
        data: [4500, 6700, 8000, 5900, 9200, 7600],
        borderColor: '#ff5722',
        borderWidth: 3,
        fill: false,
        yAxisID: 'yRevenue',
```

```
        tension: 0.3,
        pointRadius: 5,
        pointBackgroundColor: '#ff5722'
      }
    ]
  },
  options: {
    responsive: true,
    interaction: {
      mode: 'index',
      intersect: false
    },
    scales: {
      yVolume: {
        type: 'linear',
        position: 'left',
        title: {
          display: true,
          text: 'Sales Volume (units)'
        },
        beginAtZero: true
      },
      yRevenue: {
        type: 'linear',
        position: 'right',
        title: {
          display: true,
          text: 'Revenue ($)'
        },
        beginAtZero: true,
        grid: {
          drawOnChartArea: false // avoids grid lines overlapping on left y-axis
        }
      },
      x: {
        title: {
          display: true,
          text: 'Month'
        }
      }
    },
    plugins: {
      legend: {
        position: 'top'
      },
      tooltip: {
        enabled: true,
        mode: 'index',
        intersect: false
      }
    }
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Mixed Bar and Line Chart Example</title>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<style>
  body {
    font-family: Arial, sans-serif;
    margin: 40px;
    background: #fafafa;
  }
  #mixedChart {
    max-width: 700px;
    max-height: 400px;
  }
</style>
</head>
<body>

<canvas id="mixedChart" width="700" height="400"></canvas>

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

  const mixedChart = new Chart(ctx, {
    type: 'bar', // default dataset type, can be overridden per dataset
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
      datasets: [
        {
          type: 'bar',
          label: 'Sales Volume',
          data: [500, 700, 800, 600, 900, 750],
          backgroundColor: '#4caf50',
          yAxisID: 'yVolume'
        },
        {
          type: 'line',
          label: 'Revenue ($)',
          data: [4500, 6700, 8000, 5900, 9200, 7600],
          borderColor: '#ff5722',
          borderWidth: 3,
          fill: false,
          yAxisID: 'yRevenue',
          tension: 0.3,
          pointRadius: 5,
          pointBackgroundColor: '#ff5722'
        }
      ]
    },
    options: {
      responsive: true,
      interaction: {
        mode: 'index',
        intersect: false
      },
      scales: {
```

```
        yVolume: {
          type: 'linear',
          position: 'left',
          title: {
            display: true,
            text: 'Sales Volume (units)'
          },
          beginAtZero: true
        },
        yRevenue: {
          type: 'linear',
          position: 'right',
          title: {
            display: true,
            text: 'Revenue ($)'
          },
          beginAtZero: true,
          grid: {
            drawOnChartArea: false
          }
        },
        x: {
          title: {
            display: true,
            text: 'Month'
          }
        }
      },
      plugins: {
        legend: {
          position: 'top'
        },
        tooltip: {
          enabled: true,
          mode: 'index',
          intersect: false
        }
      }
    }
  });
</script>

</body>
</html>
```

9.1.3 Explanation

- The chart's **default type** is set to `'bar'`, but the line dataset overrides this with its own `type: 'line'`.
- **Dual y-axes** are created via `yVolume` (left) and `yRevenue` (right), allowing each dataset to use an appropriate scale.
- `grid.drawOnChartArea: false` on the right y-axis prevents grid lines from overlapping,

keeping the chart clean.
- Interaction mode is set to `'index'` with `intersect: false` to highlight all datasets at the hovered label simultaneously.
- Visual styles like colors, line tension (smooth curves), and point sizes are customized per dataset.

9.1.4 When to Use Mixed Charts

- Comparing **different units or scales** on the same timeline or categories.
- Combining **quantitative** and **trend** data (e.g., sales vs. revenue, expenses vs. profit).
- Enhancing data storytelling with **multiple perspectives** in a single glance.

9.1.5 Tips for Mixed Charts

- Clearly **label axes** and provide legends to avoid confusion.
- Use contrasting colors and line styles to distinguish datasets visually.
- Avoid overcrowding—limit datasets to a manageable number for clarity.
- Use tooltips and interaction modes to help users interpret combined data points.

Mixed charts unlock richer insights by blending chart types effectively. Next, we'll explore how to arrange **dashboard layouts** with multiple charts for a comprehensive overview.

9.2 Dashboard Layouts with Multiple Charts

When building data dashboards, it's common to display **multiple charts side-by-side or stacked** to provide a comprehensive view of key performance indicators (KPIs) or related datasets. Organizing these charts effectively on a single page requires thoughtful layout strategies to ensure clarity, responsiveness, and ease of comparison.

In this section, we'll explore how to use HTML and CSS techniques—such as Flexbox and CSS Grid—to arrange multiple Chart.js charts in clean, adaptable dashboard layouts.

9.2.1 Layout Strategies for Multiple Charts

Using Flexbox

Flexbox is great for creating flexible, one-dimensional layouts (either row or column). It allows charts to adjust and wrap as the viewport changes.

Example: Two-Column Dashboard Using Flexbox

```
<div class="dashboard-flex">
  <div class="chart-container">
    <canvas id="chart1"></canvas>
  </div>
  <div class="chart-container">
    <canvas id="chart2"></canvas>
  </div>
  <div class="chart-container">
    <canvas id="chart3"></canvas>
  </div>
  <div class="chart-container">
    <canvas id="chart4"></canvas>
  </div>
</div>
```

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

.chart-container {
  flex: 1 1 45%; /* Grow and shrink, base width ~45% */
  min-width: 300px;
  background: #f9f9f9;
  padding: 15px;
  box-shadow: 0 2px 6px rgba(0,0,0,0.1);
  border-radius: 8px;
}
```

This layout creates a flexible two-column dashboard that wraps to one column on narrower screens (due to `flex-wrap: wrap` and `min-width`), keeping charts readable and accessible.

Using CSS Grid

CSS Grid is ideal for two-dimensional layouts where you want explicit control over rows and columns.

Example: Two-Column Dashboard Using CSS Grid

```
<div class="dashboard-grid">
  <div class="chart-container"><canvas id="chartA"></canvas></div>
  <div class="chart-container"><canvas id="chartB"></canvas></div>
  <div class="chart-container"><canvas id="chartC"></canvas></div>
  <div class="chart-container"><canvas id="chartD"></canvas></div>
</div>
```

```css
.dashboard-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
  gap: 20px;
}

.chart-container {
  background: #fff;
  padding: 15px;
  border-radius: 6px;
  box-shadow: 0 1px 5px rgba(0,0,0,0.1);
}
```

With CSS Grid, the layout automatically adapts by fitting as many 300px-wide columns as the viewport allows, distributing charts evenly.

9.2.2 Organizing Related Charts

- **Group related KPIs** or metrics visually close for easier comparison (e.g., sales charts together, customer data together).
- Use **consistent sizing** and padding for visual harmony.
- Employ headings or section dividers to clarify chart categories.

9.2.3 Optimizing for Responsiveness

- Use relative widths (`%`, `fr`) and flexible units (`minmax()`, `auto-fit`) so charts resize smoothly.
- Set minimum widths on containers to avoid overly squished charts.
- Use media queries if you need custom layout tweaks for tablets and phones.

9.2.4 Putting It All Together: Two-Column KPI Dashboard Example

```html
<section class="dashboard-grid">
  <div class="chart-container">
    <h3>Monthly Sales</h3>
    <canvas id="salesChart"></canvas>
  </div>
  <div class="chart-container">
    <h3>Revenue Trend</h3>
    <canvas id="revenueChart"></canvas>
  </div>
  <div class="chart-container">
    <h3>Customer Growth</h3>
```

```
    <canvas id="customerChart"></canvas>
  </div>
  <div class="chart-container">
    <h3>Product Performance</h3>
    <canvas id="productChart"></canvas>
  </div>
</section>
```

```
.dashboard-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(320px, 1fr));
  gap: 24px;
  padding: 20px;
}

.chart-container {
  background-color: #ffffff;
  border-radius: 8px;
  padding: 20px;
  box-shadow: 0 3px 8px rgba(0,0,0,0.12);
}

.chart-container h3 {
  margin-bottom: 12px;
  font-weight: 600;
  color: #333;
}
```

9.2.5 Summary

Layout Method	When to Use	Benefits
Flexbox	Simple row or column layouts with wrapping	Easy, flexible, good browser support
CSS Grid	Complex 2D layouts with explicit rows & cols	Powerful control, responsive by design

By combining these layout techniques with Chart.js charts, you can build dashboards that are both **informative** and **responsive**, enhancing the overall user experience on any device.

Next, we'll cover how to **synchronize interactions across multiple charts** to create seamless, interactive dashboards.

9.3 Synchronizing Interactions Across Charts

9.3.1 Synchronizing Interactions Across Charts

When working with multiple charts on a dashboard, synchronizing interactions such as **hovering**, **tooltips**, and **zooming** can greatly enhance the user experience by providing coordinated views of related data. This technique—often called **linked brushing** or **coordinated highlighting**—helps users explore complex datasets more intuitively.

9.3.2 Why Synchronize Chart Interactions?

- **Consistent highlighting:** Hovering over a data point in one chart highlights corresponding data in others.
- **Unified tooltips:** Showing related information across charts simultaneously.
- **Synchronized zoom and pan:** When zooming or panning in one chart, all related charts adjust their view to match.
- Improves **comparative analysis** and storytelling by connecting disparate datasets visually.

9.3.3 Techniques to Synchronize Interactions in Chart.js

Chart.js does not provide built-in synchronization, but it can be achieved using JavaScript event listeners and the Chart.js API. Here are common strategies:

9.3.4 Synchronizing Hover and Tooltips

By listening to the `mousemove` or `mouseout` events on one chart's canvas, you can programmatically show or hide tooltips and highlight elements on other charts.

Example: Sync Hover Between Two Charts

```
const ctx1 = document.getElementById('chart1').getContext('2d');
const ctx2 = document.getElementById('chart2').getContext('2d');

const chart1 = new Chart(ctx1, { /* chart1 config */ });
const chart2 = new Chart(ctx2, { /* chart2 config */ });

function syncHover(sourceChart, targetChart) {
  sourceChart.canvas.addEventListener('mousemove', (event) => {
    const points = sourceChart.getElementsAtEventForMode(event, 'nearest', { intersect: true }, true);
    if (points.length) {
```

```
      const index = points[0].index;
      // Set active elements on the target chart for the same index
      targetChart.setActiveElements([
        { datasetIndex: 0, index }
      ]);
      targetChart.tooltip.setActiveElements([
        { datasetIndex: 0, index }
      ], { x: event.x, y: event.y });
      targetChart.update();
    } else {
      // Clear highlights if no point hovered
      targetChart.setActiveElements([]);
      targetChart.tooltip.setActiveElements([], { x: 0, y: 0 });
      targetChart.update();
    }
  });

  sourceChart.canvas.addEventListener('mouseout', () => {
    targetChart.setActiveElements([]);
    targetChart.tooltip.setActiveElements([], { x: 0, y: 0 });
    targetChart.update();
  });
}

syncHover(chart1, chart2);
syncHover(chart2, chart1);
```

This sets up mutual highlighting: hovering over a point in one chart highlights the corresponding point in the other.

9.3.5 Synchronizing Zoom and Pan

Using the `chartjs-plugin-zoom`, you can listen for zoom and pan events on one chart and apply the same scale adjustments to others.

Example: Sync Zoom and Pan

```
// Assuming chart1 and chart2 are created with chartjs-plugin-zoom enabled

chart1.options.plugins.zoom.onZoom = ({ chart }) => {
  const xScale = chart.scales.x;
  const yScale = chart.scales.y;

  // Apply the same scale limits to chart2
  chart2.options.scales.x.min = xScale.min;
  chart2.options.scales.x.max = xScale.max;
  chart2.options.scales.y.min = yScale.min;
  chart2.options.scales.y.max = yScale.max;
  chart2.update('none'); // 'none' disables animation for smoother sync
};

chart1.options.plugins.zoom.onPan = chart1.options.plugins.zoom.onZoom; // Use same handler for pan
```

This ensures that when users zoom or pan on one chart, the other chart's view updates accordingly.

9.3.6 Coordinated Brushing

Brushing means selecting a range or group of data points on one chart to filter or highlight related data in another.

While Chart.js does not have native brushing tools, you can implement a custom UI overlay or use click events combined with dataset filtering.

Basic Concept:

- User clicks or drags on a chart to select points or ranges.
- Capture selected indices or values.
- Update the other charts by filtering or highlighting data corresponding to the selection.

9.3.7 Best Practices for Synchronization

- Use **shared labels or indices** across datasets to correctly map data points.
- **Throttle or debounce** event handlers for smooth performance.
- Provide clear visual feedback for linked interactions to avoid confusion.
- Test across devices and screen sizes to ensure usability.

9.3.8 Summary

Interaction	Technique	Key API/Tools
Hover & Tooltip Sync	Event listeners + `setActiveElements()`	`getElementsAtEventForMode()`, `update()`
Zoom & Pan Sync	Listen to zoom/pan events, update scales	`chartjs-plugin-zoom`, scale APIs
Coordinated Brushing	Custom event capture + dataset filtering	Custom UI + `update()`

Synchronizing interactions across multiple charts makes dashboards more intuitive and insightful. In the next chapter, we'll explore **styling and theming** techniques to polish your visualizations.

Chapter 10.

Exporting and Sharing Visualizations

1. Exporting Charts as Images (PNG, JPEG)
2. Saving Chart Data and Configurations
3. Embedding Charts in Websites and Reports
4. Printing and PDF Export Tips

10 Exporting and Sharing Visualizations

10.1 Exporting Charts as Images (PNG, JPEG)

Exporting charts as images is a common requirement for sharing visual insights via reports, presentations, or social media. Chart.js makes this straightforward by leveraging the underlying `<canvas>` element's native image-export capabilities.

10.1.1 How Chart.js Enables Image Export

Chart.js charts render inside an HTML `<canvas>`. This canvas element provides methods like:

- `canvas.toDataURL()` — Returns a base64-encoded data URL representing the image.
- Chart.js convenience method `chart.toBase64Image()` — Returns the same base64 image URL for the chart's current state.

You can use these methods to:

- Display the image elsewhere.
- Trigger downloads.
- Send the image to a server.
- Share via APIs.

10.1.2 Exporting and Downloading an Image on Button Click

Here's a step-by-step example demonstrating how to export the current chart view as a PNG file and download it when a button is clicked.

```
<canvas id="myChart" width="600" height="400"></canvas>
<button id="downloadBtn">Download Chart as PNG</button>
```

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

const myChart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    datasets: [{
      label: 'Sales',
      data: [150, 200, 170, 220, 190],
      borderColor: 'blue',
      fill: false,
      tension: 0.4
    }]
  },
  options: {
```

```
    responsive: false
  }
});

document.getElementById('downloadBtn').addEventListener('click', () => {
  // Use Chart.js method to get base64 image string
  const base64Image = myChart.toBase64Image();

  // Create a temporary link element
  const link = document.createElement('a');
  link.href = base64Image;
  link.download = 'chart-image.png'; // Filename for download

  // Programmatically click the link to trigger download
  link.click();
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Export Chart as PNG Example</title>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<style>
  body {
    font-family: Arial, sans-serif;
    margin: 40px;
  }
  #myChart {
    border: 1px solid #ddd;
    margin-bottom: 10px;
  }
</style>
</head>
<body>

<canvas id="myChart" width="600" height="400"></canvas>
<br />
<button id="downloadBtn">Download Chart as PNG</button>

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

  const myChart = new Chart(ctx, {
    type: 'line',
    data: {
      labels: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
      datasets: [{
        label: 'Sales',
        data: [150, 200, 170, 220, 190],
        borderColor: 'blue',
        fill: false,
        tension: 0.4
      }]
    },
```

```
    options: {
      responsive: false
    }
  });

  document.getElementById('downloadBtn').addEventListener('click', () => {
    const base64Image = myChart.toBase64Image();

    const link = document.createElement('a');
    link.href = base64Image;
    link.download = 'chart-image.png';

    // For Firefox, link must be added to the DOM to trigger click
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);
  });
</script>

</body>
</html>
```

10.1.3 Explanation

- `myChart.toBase64Image()` returns a PNG image data URL representing the chart exactly as rendered.
- We create a hidden anchor (`<a>`) element with the `href` set to the image data URL and a `download` attribute specifying the filename.
- Triggering `link.click()` initiates the browser's download dialog.
- This method works without server interaction and is widely supported in modern browsers.

10.1.4 Exporting as JPEG or Custom Format

By default, `toBase64Image()` returns PNG. To export JPEG or customize quality, use the native canvas method:

```
const canvas = document.getElementById('myChart');
const jpegUrl = canvas.toDataURL('image/jpeg', 0.9); // 90% quality JPEG

// Then use jpegUrl like before to download or share
```

10.1.5 Summary

Method	Description	Use Case
`chart.toBase64Image()`	Easy Chart.js wrapper, returns PNG data URL	Quick export/download
`canvas.toDataURL()`	Native method, supports formats & quality	Export JPEG, WebP, or custom use

With these simple techniques, your users can **capture, download, and share beautiful Chart.js visualizations** instantly. In the next section, we'll cover saving and exporting chart data and configurations.

10.2 Saving Chart Data and Configurations

Preserving your Chart.js charts' data and configuration is essential for use cases like **saving user settings**, **sharing charts**, or **restoring state** after a page reload. Chart.js makes this easy because both the chart's **data** and **options** are represented as JavaScript objects, which can be serialized into JSON.

10.2.1 Why Save Chart Configuration?

- Reload charts with the exact same settings and data.
- Transfer charts between clients and servers.
- Enable users to save and load personalized views.
- Support offline usage or caching.

10.2.2 How to Serialize Chart Data and Configuration

Since Chart.js stores all relevant information inside its config object, you can save:

```
const chartConfig = {
  type: myChart.config.type,
  data: myChart.data,
  options: myChart.options
};
```

You can then convert this to JSON using:

```
const jsonString = JSON.stringify(chartConfig);
```

This JSON string can be:

- Saved to **localStorage** in the browser.
- Sent to a **backend API** for persistent storage.
- Exported as a file for manual sharing.

10.2.3 Example: Saving and Reloading Chart State with localStorage

```
// Save chart state to localStorage
function saveChartState(chart) {
  const chartState = {
    type: chart.config.type,
    data: chart.data,
    options: chart.options
  };
  localStorage.setItem('savedChart', JSON.stringify(chartState));
}

// Load chart state from localStorage and recreate the chart
function loadChartState(ctx) {
  const saved = localStorage.getItem('savedChart');
  if (saved) {
    const config = JSON.parse(saved);
    return new Chart(ctx, config);
  }
  return null;
}

// Usage
const ctx = document.getElementById('myChart').getContext('2d');
let myChart = new Chart(ctx, {
  type: 'bar',
  data: { /* initial data */ },
  options: { /* initial options */ }
});

// Save button triggers saving current chart state
document.getElementById('saveBtn').addEventListener('click', () => {
  saveChartState(myChart);
});

// Load button recreates the chart with saved state
document.getElementById('loadBtn').addEventListener('click', () => {
  if (myChart) myChart.destroy(); // Destroy previous instance
  myChart = loadChartState(ctx);
});
```

10.2.4 Example: Sending Chart Config to a Backend API

```
async function sendChartToServer(chart) {
  const chartConfig = {
```

```
    type: chart.config.type,
    data: chart.data,
    options: chart.options
  };

  const response = await fetch('/api/saveChart', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(chartConfig)
  });

  if (response.ok) {
    console.log('Chart saved successfully');
  } else {
    console.error('Failed to save chart');
  }
}
```

This method lets you persist user charts on the server or share them with others.

10.2.5 Notes and Best Practices

- Large datasets can produce very big JSON strings—consider compressing or paginating if needed.
- When reloading charts, always **destroy** existing instances before creating new ones to avoid memory leaks.
- Some Chart.js options or plugins might include non-serializable properties (like functions). Focus on saving **data and basic options** for best results.

10.2.6 Summary

Use Case	Technique	Example APIs
Save chart locally	`localStorage` + JSON	`JSON.stringify()`, `localStorage.setItem()`
Share/transfer charts	Serialize and POST JSON	`fetch()` with JSON body
Reload saved charts	Parse JSON + new Chart()	`JSON.parse()`, `Chart()`

By leveraging JSON serialization, you enable powerful workflows to **save, share, and restore Chart.js visualizations seamlessly**. Next, we will explore embedding these charts into websites and reports.

10.3 Embedding Charts in Websites and Reports

Embedding Chart.js visualizations into websites, blogs, or markdown-based reports allows you to showcase dynamic and interactive data insights directly within your content. This section covers practical ways to integrate charts smoothly, along with best practices to maintain styling consistency and performance.

10.3.1 Embedding Charts Directly in Webpages

The most straightforward way to embed a Chart.js chart is to include the necessary HTML, Chart.js script, and chart initialization code directly within your webpage or blog post.

10.3.2 Basic Example

```
<!-- Include Chart.js via CDN -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<!-- Canvas element where the chart will be rendered -->
<canvas id="myChart" width="600" height="400"></canvas>

<script>
  const ctx = document.getElementById('myChart').getContext('2d');
  new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['Q1', 'Q2', 'Q3', 'Q4'],
      datasets: [{
        label: 'Revenue',
        data: [12000, 15000, 13000, 17000],
        backgroundColor: 'rgba(54, 162, 235, 0.6)'
      }]
    },
    options: { responsive: true }
  });
</script>
```

Best Practices:

- Place your `<canvas>` element inside a container div with CSS to control size and responsiveness.
- Use the `responsive: true` option for automatic resizing.
- Load Chart.js via a CDN for fast, reliable delivery without extra setup.

10.3.3 Embedding in Markdown-Based Reports

Many modern markdown renderers (e.g., those used in Jupyter notebooks, GitHub Pages, or static site generators) allow embedding HTML and JavaScript, making it possible to insert Chart.js charts.

10.3.4 Approaches:

- **Inline HTML blocks:** Insert `<canvas>` and `<script>` tags directly in markdown files (if the renderer supports it).
- **External JavaScript files:** Reference a `.js` file containing the chart code.
- **Shortcodes or plugins:** Use platform-specific extensions to embed charts.

Example markdown snippet:

```
### Quarterly Revenue Chart

<div style="width: 100%; max-width: 600px;">
  <canvas id="revenueChart"></canvas>
</div>

<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
  const ctx = document.getElementById('revenueChart').getContext('2d');
  new Chart(ctx, {
    type: 'line',
    data: {
      labels: ['Q1', 'Q2', 'Q3', 'Q4'],
      datasets: [{
        label: 'Revenue',
        data: [12000, 15000, 13000, 17000],
        borderColor: 'rgba(75, 192, 192, 1)',
        fill: false
      }]
    }
  });
</script>
```

10.3.5 Embedding Charts Using Iframes

If you need to embed a Chart.js chart hosted separately (for example, in an external HTML file or visualization platform), you can use an `<iframe>` tag.

10.3.6 Example:

```
<iframe src="https://yourdomain.com/charts/sales-chart.html"
        width="650" height="450" style="border:none;">
</iframe>
```

Advantages:

- Encapsulates the chart code and dependencies, preventing conflicts.
- Easier to update the chart independently.
- Ideal for embedding in CMS platforms or environments where inline JavaScript is restricted.

10.3.7 Styling Embedded Charts

To maintain visual consistency:

- Use CSS to control the container size and aspect ratio.
- Override Chart.js global styles if needed with CSS variables or Chart.js global defaults.
- Make sure the chart container is responsive for different screen sizes.

Example CSS for responsive container:

```
.chart-container {
  position: relative;
  width: 100%;
  max-width: 700px;
  height: 400px;
  margin: auto;
}
```

Then wrap the canvas:

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

10.3.8 Summary

Embedding Method	Pros	Cons
Inline Script + Canvas	Simple, flexible, no extra hosting	Requires JavaScript support
Markdown with HTML	Integrates well with markdown reports	Some platforms restrict inline JS

Iframe	Isolated, reusable, easy updates	Slightly more complex, external hosting

By choosing the appropriate embedding method and following these best practices, you can seamlessly integrate Chart.js visualizations into websites, blogs, or reports—offering interactive, insightful, and visually appealing data stories to your audience.

In the next section, we'll explore tips for printing charts and exporting to PDF.

10.4 Printing and PDF Export Tips

Printing charts or exporting them to PDF is a common requirement when generating reports, presentations, or documentation. Ensuring your Chart.js visualizations look crisp and professional on paper or in PDF requires some preparation. This section covers best practices for **print-friendly styling**, **high-resolution rendering**, and using tools for smooth PDF exports.

10.4.1 Making Charts Print-Friendly

Use Print-Specific CSS

Apply CSS rules that optimize the chart container and page layout for printing:

```
@media print {
  body {
    background: white;
    color: black;
  }

  .chart-container {
    width: 100% !important;
    height: auto !important;
  }

  canvas {
    max-width: 100% !important;
    height: auto !important;
  }
}
```

This ensures:

- Background colors are removed or simplified.
- The chart resizes to fit the printed page width.
- Text and labels use print-optimized colors and fonts.

Increase Canvas Resolution for Crisp Prints

The default canvas resolution may appear pixelated on print or PDF. To improve quality:

- Increase the canvas size by setting its width and height attributes (not just CSS).
- Use the `devicePixelRatio` to scale the canvas for high-DPI output.

Example:

```
const canvas = document.getElementById('myChart');
const ctx = canvas.getContext('2d');
const scale = window.devicePixelRatio || 1;

// Set actual canvas size
canvas.width = canvas.clientWidth * scale;
canvas.height = canvas.clientHeight * scale;
ctx.scale(scale, scale);

// Now create the chart as usual
const myChart = new Chart(ctx, {
  /* config */
});
```

This renders a sharper chart for printing or PDF.

10.4.2 Exporting Charts to PDF

Using Browser Print Features

- Simply open the page containing your chart.
- Use **Print Preview** to check layout.
- Print to a physical printer or "Save as PDF" in modern browsers.

Make sure your print CSS optimizations are active.

Using JavaScript Libraries Like `html2pdf`

To programmatically export charts to PDF, you can use libraries like `html2pdf.js`, which convert HTML content (including canvas) into PDFs.

Example usage:

```
const element = document.getElementById('chart-container');

html2pdf()
  .from(element)
  .set({
    margin: 1,
    filename: 'chart.pdf',
    image: { type: 'jpeg', quality: 0.98 },
    html2canvas: { scale: 2 }, // Increase resolution
    jsPDF: { unit: 'in', format: 'letter', orientation: 'landscape' }
  })
  .save();
```

This method preserves styles, fonts, and chart visuals in the exported PDF.

10.4.3 Additional Tips for Print and PDF Quality

- **Colors:** Use high-contrast colors or print-optimized palettes to maintain readability.
- **Fonts:** Choose system-safe or embedded fonts that render consistently in PDFs.
- **Margins and Spacing:** Leave sufficient whitespace around charts to prevent clipping.
- **Test across devices and printers:** Different printers and PDF viewers can render colors and fonts differently—always test before distributing.
- **Avoid animations:** Animated charts may not render correctly when exporting; use static snapshots instead.

10.4.4 Summary

Technique	Description	Benefits
Print CSS media queries	Customize styles for paper output	Better layout and color fidelity
High-resolution canvas	Scale canvas for crisp print	Sharper charts on paper and PDF
Browser print/save to PDF	Built-in simple export	Easy for quick print jobs
`html2pdf.js` library	Programmatic HTML to PDF	Full control over PDF generation

By following these printing and PDF export tips, you can ensure your Chart.js visualizations look professional and clear—whether viewed on paper or in a digital document.

This concludes our chapter on exporting and sharing visualizations. With these techniques, your charts will shine across multiple platforms and formats!

Chapter 11.

Performance and Troubleshooting

1. Optimizing Chart Performance
2. Common Issues and Fixes
3. Debugging Chart Configurations
4. Handling Large Datasets

11 Performance and Troubleshooting

11.1 Optimizing Chart Performance

As your charts grow in complexity or the amount of data increases, maintaining smooth rendering and responsiveness becomes critical. Chart.js provides several ways to optimize performance, ensuring a seamless user experience even with large datasets or complex visualizations.

11.1.1 Reduce or Disable Animations

Animations can be visually appealing but add computational overhead, especially with large or frequently updated charts.

- **Disable animations completely:**

```
options: {
  animation: false
}
```

- **Reduce animation duration or easing for faster rendering:**

```
options: {
  animation: {
    duration: 200, // shorter duration
    easing: 'linear'
  }
}
```

11.1.2 Limit the Number of Data Points

Rendering thousands of points can slow down the chart drastically.

- **Aggregate or sample your data** before feeding it to Chart.js.
- **Use data decimation** (see next section) for automatic downsampling.

11.1.3 Use the Decimation Plugin for Large Datasets

Chart.js offers a built-in decimation plugin that intelligently reduces the number of points drawn without significantly affecting visual fidelity.

Enable it in your chart options:

```
options: {
  plugins: {
    decimation: {
      enabled: true,
      algorithm: 'min-max', // or 'lttb' (largest triangle three buckets)
      samples: 1000 // max points to display
    }
  }
}
```

This drastically improves performance by drawing fewer points while preserving the data's shape.

11.1.4 Avoid Excessive Chart Updates

Re-rendering the entire chart frequently (e.g., inside a tight loop or on every data tick) can degrade responsiveness.

- Batch updates and call `chart.update()` sparingly.
- Use `update('none')` if you need to update data silently without animations.

11.1.5 Optimize Dataset Styling

- Use simpler styles (solid colors vs. gradients or patterns).
- Limit shadow, border, and point effects.
- Reduce the number of datasets where possible.

11.1.6 Use Web Workers for Heavy Data Processing

Offload complex calculations or data transformations to Web Workers to avoid blocking the UI thread. Process data asynchronously before passing it to Chart.js.

11.1.7 Summary Table

Optimization Technique	When to Use	Benefits
Disable or reduce animations	Large/complex charts	Faster rendering
Limit or sample data points	Very large datasets	Reduced draw calls

Optimization Technique	When to Use	Benefits
Decimation plugin	Time series or scatter plots	Intelligent downsampling
Minimize update frequency	Real-time or interactive charts	Smoother user experience
Simplify styling	Visual complexity affects speed	Lower CPU/GPU usage
Web Workers	Heavy preprocessing or filtering	Non-blocking UI updates

By applying these optimization techniques, you ensure your Chart.js visualizations remain performant and responsive, providing a smooth experience regardless of data size or complexity.

Next, we will look at common issues and how to troubleshoot them efficiently.

11.2 Common Issues and Fixes

When working with Chart.js, you may encounter several common issues that can disrupt your visualization experience. This section outlines typical problems and provides practical solutions along with a step-by-step debugging checklist to help you quickly resolve them.

11.2.1 Chart Not Rendering or Blank Canvas

Symptoms:

- The `<canvas>` element is visible but empty.
- No chart appears after initialization.

Causes & Fixes:

- **Missing or incorrect Chart.js script:** Ensure the Chart.js library is properly loaded (via CDN or package).
- **Incorrect canvas ID or selector:** Confirm the canvas element's ID matches the one used in JavaScript.
- **Context not retrieved properly:** Use `getContext('2d')` on the canvas element before creating the chart.
- **Chart configuration errors:** Check the config object for syntax errors or missing required fields like `type` or `data`.
- **Canvas size is zero:** Set explicit width and height on the canvas or container to ensure it's visible.

11.2.2 Incorrect Axis Scale or Labels

Symptoms:

- Axis values look skewed or unexpected.
- Labels are missing or misaligned.

Causes & Fixes:

- **Data and labels mismatch:** Ensure `data.labels.length` matches the number of points in each dataset.
- **Improper scale configuration:** Check scale type (`linear`, `logarithmic`, `time`, etc.) and adjust options like `min`, `max`, and `stepSize`.
- **Time scale parsing errors:** Verify that date strings are in a supported format or use `moment.js`/`date-fns` integration.
- **Hidden axes:** Confirm axes are not accidentally disabled with `display: false`.

11.2.3 Missing or Incorrect Tooltips

Symptoms:

- Tooltips do not appear on hover.
- Tooltip content is wrong or empty.

Causes & Fixes:

- **Tooltip plugin disabled:** Check if tooltips are enabled under `options.plugins.tooltip.enabled`.
- **Custom callbacks returning undefined:** Review callback functions (`label`, `title`, etc.) to ensure they return valid strings.
- **Interaction mode conflicts:** Adjust `interaction.mode` (e.g., `'nearest'`, `'index'`) to ensure the tooltip triggers properly.

11.2.4 Legend Not Displaying or Misbehaving

Symptoms:

- Legend does not show or overlaps the chart.
- Clicking legend items doesn't toggle datasets.

Causes & Fixes:

- **Legend plugin disabled:** Verify `options.plugins.legend.display` is `true`.
- **Legend position causing overlap:** Adjust `position` property (`top`, `bottom`, `left`, `right`).

- **Custom legend callbacks causing errors:** Ensure callbacks like `labels.generateLabels` return proper label objects.

11.2.5 Performance Issues or Freezing

Symptoms:

- Chart rendering is slow or unresponsive.
- Browser freezes on update.

Causes & Fixes:

- Refer to **Chapter 11, Section 1: Optimizing Chart Performance** for detailed solutions like disabling animations and using the decimation plugin.

11.2.6 Debugging Checklist

When your chart isn't working as expected, systematically check the following:

1. **Verify Canvas Element**
   - Is the `<canvas>` element present in the DOM?
   - Does it have the correct ID or selector?
   - Is the canvas size set explicitly or via CSS?
2. **Check Chart.js Library Inclusion**
   - Is the Chart.js script loaded without errors?
   - Are there any JavaScript console errors?
3. **Validate Chart Configuration Object**
   - Does it have a valid `type`, `data`, and `options`?
   - Are datasets and labels properly structured?
   - Are scale and plugin options correctly defined?
4. **Inspect Data Integrity**
   - Are data arrays numeric and consistent in length?
   - Are labels matching the number of data points?
5. **Review Plugin and Interaction Settings**
   - Are tooltips and legends enabled?
   - Is the interaction mode appropriate for the chart type?
6. **Test with Minimal Example**

- Simplify your config to a minimal working example.
- Gradually add options to isolate the issue.

7. **Use Browser Developer Tools**

   - Look for errors or warnings in the console.
   - Use breakpoints or `console.log()` to inspect variables.

11.2.7 Summary Table of Common Issues

Problem	Possible Cause	Fix Approach
Chart not rendering	Missing script, wrong canvas ID	Verify library load and selectors
Axis scale issues	Data-label mismatch, bad config	Check data length and scale options
Missing tooltips	Disabled tooltip plugin	Enable tooltips, fix callbacks
Legend problems	Disabled legend, positioning	Enable and reposition legend
Performance slowdowns	Large data, animations on	Optimize per performance section

By following these troubleshooting steps, you can quickly diagnose and resolve common Chart.js problems, ensuring your visualizations work reliably across projects.

Next, we'll explore advanced debugging techniques to fine-tune your chart configurations.

11.3 Debugging Chart Configurations

When working with complex Chart.js configurations, subtle mistakes or conflicting settings can cause unexpected behavior or errors. Effective debugging techniques can save you time and help you quickly identify and fix issues. This section guides you through practical methods for debugging your chart configurations.

11.3.1 Step 1: Use Browser Developer Tools

Modern browsers like Chrome, Firefox, and Edge provide powerful developer tools for debugging JavaScript applications.

- **Open the Console Tab:** Check for any error messages or warnings emitted by

Chart.js or your JavaScript code. These messages often indicate syntax errors, missing properties, or plugin conflicts.

- **Inspect the Canvas Element:** Use the Elements tab to confirm that the `<canvas>` element exists, has proper dimensions, and that no CSS rules are hiding or shrinking it.
- **Debugging with Breakpoints:** You can set breakpoints in your JavaScript code where the chart is created or updated to step through execution and inspect variables.

11.3.2 Step 2: Console Logging Configuration and Data

- **Log Your Chart Configuration:** Before initializing the chart, log the entire config object:

```
console.log('Chart configuration:', chartConfig);
```

  This helps verify that your data arrays, labels, options, and plugins are structured as expected.

- **Inspect Dataset and Labels Length:** Mismatched lengths often cause axis or rendering issues. Check:

```
console.log('Labels length:', chartConfig.data.labels.length);
chartConfig.data.datasets.forEach((dataset, index) => {
  console.log(`Dataset ${index} length:`, dataset.data.length);
});
```

- **Validate Data Types:** Confirm that data points are numbers (or objects if using scatter/bubble charts) and labels are strings or dates.

11.3.3 Step 3: Isolate Problematic Settings

If the chart behaves unexpectedly, isolate the cause by simplifying the configuration:

- **Start Minimal:** Remove optional options, plugins, or datasets. Create a basic working chart.
- **Add Features One-by-One:** Gradually reintroduce settings like custom scales, animations, or callbacks, testing after each change.
- **Identify Conflicts:** Some plugins or options may interfere with each other. Disable plugins temporarily to test.

11.3.4 Step 4: Validate Plugin Compatibility

- **Check Plugin Versions:** Ensure plugins are compatible with your Chart.js version.
- **Review Plugin Options:** Incorrect or outdated plugin options can cause errors or no effect.
- **Test Without Plugins:** Remove all plugins to confirm if a plugin is causing issues.

11.3.5 Step 5: Use Online Tools and Validators

- **JSON Validators:** If your config is generated or loaded as JSON, validate it with online JSON validators to avoid syntax issues.
- **Chart.js Sandbox:** Use online playgrounds like the official Chart.js Sample Gallery to prototype and debug configurations.

11.3.6 Debugging Example

```
const config = {
  type: 'bar',
  data: {
    labels: ['Jan', 'Feb', 'Mar'],
    datasets: [{
      label: 'Sales',
      data: [100, 200], // Error: length mismatch
      backgroundColor: 'blue'
    }]
  },
  options: {}
};

console.log('Labels:', config.data.labels.length); // 3
console.log('Dataset data length:', config.data.datasets[0].data.length); // 2

// Fix: Add the missing data point or remove the extra label.
```

11.3.7 Summary Checklist

Debugging Step	What to Check	Tools/Methods
Console Errors	Syntax, runtime errors	Browser DevTools Console
Canvas Element	Presence and sizing	Elements Panel

Debugging Step	What to Check	Tools/Methods
Configuration Logging	Data shapes and option correctness	`console.log()`
Minimal Config Testing	Identify conflicting settings	Manual config simplification
Plugin Validation	Compatibility and proper usage	Plugin docs and test removal
JSON Validation	Syntax correctness	Online JSON Validators

By methodically applying these debugging techniques, you can efficiently track down and fix issues in complex Chart.js configurations, making your charts reliable and polished.

Next, we'll explore strategies for handling very large datasets in Chart.js.

11.4 Handling Large Datasets

Visualizing large datasets in Chart.js can challenge performance and usability. To maintain smooth interactions and clear insights, you need strategies that reduce the rendering workload without sacrificing data quality. This section covers practical techniques such as sampling, aggregation, and leveraging zoom plugins to efficiently handle large volumes of data.

11.4.1 Sampling and Downsampling

Rendering every single data point in a large dataset can overwhelm the browser and cause sluggish performance. Sampling reduces the number of points by selecting a representative subset.

- **Manual Sampling:** Select every n-th data point or use domain knowledge to pick important points.
- **Automated Downsampling with Decimation Plugin:** Chart.js includes a built-in decimation plugin that intelligently reduces points while preserving chart shape.

```
options: {
  plugins: {
    decimation: {
      enabled: true,
      algorithm: 'lttb', // largest triangle three buckets
      samples: 500 // max number of points to display
    }
  }
}
```

This method works especially well for time-series data by smoothing out less important fluctuations.

11.4.2 Aggregation (Grouping)

Another approach is to aggregate raw data into summarized values over intervals, such as hourly averages or monthly totals. Aggregation reduces data points and improves readability.

- **Example: Grouping sales data by month**

```
const rawData = [ /* daily sales */ ];
const monthlyData = aggregateByMonth(rawData); // custom function to sum or average

const chartData = {
  labels: monthlyData.map(d => d.month),
  datasets: [{ data: monthlyData.map(d => d.total) }]
};
```

Aggregation is often done before passing data to Chart.js, either on the server or client side.

11.4.3 Zoom and Pan Plugins for Lazy Loading

Instead of rendering all data at once, zooming and panning allow users to focus on subsets dynamically, improving performance.

- **chartjs-plugin-zoom:** This plugin enables zoom and pan gestures on your charts, which lets you load or highlight only relevant data ranges.

```
plugins: [ChartZoom],

options: {
  plugins: {
    zoom: {
      zoom: {
        wheel: { enabled: true },
        pinch: { enabled: true },
        mode: 'x'
      },
      pan: {
        enabled: true,
        mode: 'x'
      }
    }
  }
}
```

- **Lazy loading data:** Combine zoom with backend or client-side logic to load data chunks as the user navigates different time windows.

11.4.4 Example: Time-Series Chart with Downsampling and Zoom

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

const chart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: largeTimeLabels,  // thousands of timestamps
    datasets: [{
      label: 'Sensor Data',
      data: largeSensorData,
      borderColor: 'blue',
      fill: false,
    }]
  },
  options: {
    plugins: {
      decimation: {
        enabled: true,
        algorithm: 'lttb',
        samples: 1000
      },
      zoom: {
        zoom: {
          wheel: { enabled: true },
          mode: 'x',
        },
        pan: {
          enabled: true,
          mode: 'x',
        }
      }
    },
    scales: {
      x: { type: 'time' }
    }
  },
  plugins: [ChartZoom]
});
```

11.4.5 Summary Table

Technique	Purpose	Benefits	Notes
Sampling	Reduce point count	Faster rendering, less clutter	May lose fine details
Aggregation	Summarize data over intervals	Clear trends, fewer points	Needs preprocessing
Zoom & Pan Plugins	Dynamic data exploration	Focused views, lazy loading	Requires plugin and logic

By applying these techniques, you can effectively visualize large datasets with Chart.js while preserving performance and interactivity. Combining downsampling, aggregation, and zooming enables smooth, responsive charts that scale with your data size.

Next, you will learn about enhancing interactivity and integrating plugins to extend Chart.js functionality.