ECharts Data Visualization

readbytes

ECharts Data Visualization

Interactive, high-performance charts

readbytes.github.io

2025-07-18

This page is intentionally left blank.

Contents

1 Introduction **12**
- 1.1 What Is ECharts? . . . 12
- 1.2 Overview of ECharts Features . . . 13
- 1.3 Setting Up Your Environment (CDN, NPM, and Development Tools) . . . 19
  - 1.3.1 Using Webpack . . . 21
  - 1.3.2 Using Vite . . . 22
  - 1.3.3 Summary . . . 23
- 1.4 Basic Concepts: Canvas, SVG, and Rendering Modes . . . 23
  - 1.4.1 What Is a Rendering Mode? . . . 24
  - 1.4.2 Canvas Mode (Default) . . . 24
  - 1.4.3 SVG Mode . . . 24
  - 1.4.4 Switching Between Rendering Modes . . . 25
  - 1.4.5 When to Use Each Rendering Mode . . . 26
  - 1.4.6 Summary . . . 26

2 Getting Started with ECharts **29**
- 2.1 Installing and Importing ECharts . . . 29
  - 2.1.1 Option 1: Using ECharts via CDN (Quick Setup) . . . 29
  - 2.1.2 Option 2: Installing ECharts via NPM (Modern Projects) . . . 30
  - 2.1.3 Importing ECharts in a Modern Setup . . . 30
  - 2.1.4 Notes on Module Bundlers . . . 31
  - 2.1.5 Summary . . . 31
- 2.2 Creating Your First Chart (Bar Chart) . . . 31
  - 2.2.1 Creating Your First Chart (Bar Chart) . . . 31
  - 2.2.2 What Well Build . . . 31
  - 2.2.3 Step-by-Step Example Using CDN . . . 32
  - 2.2.4 Explanation of Each Part . . . 33
  - 2.2.5 Notes . . . 34
  - 2.2.6 Whats Next? . . . 34
- 2.3 ECharts Core Concepts: Option Object and API . . . 34
  - 2.3.1 The `option` Object: Chart Configuration . . . 35
  - 2.3.2 Core Chart Instance API . . . 36
  - 2.3.3 Example: Option API Together . . . 37
  - 2.3.4 Summary . . . 38
- 2.4 Anatomy of an ECharts Chart . . . 39
  - 2.4.1 Core Layers of an ECharts Chart . . . 39
  - 2.4.2 Annotated ECharts Chart Example . . . 39
  - 2.4.3 Breakdown of the Example . . . 40
  - 2.4.4 Rendering Engine: Canvas vs. SVG . . . 41
  - 2.4.5 Summary . . . 41

3 Chart Fundamentals **43**

3.1 Understanding the Option Object Structure . . . 43
3.1.1 The Hierarchical Structure . . . 43
3.1.2 Annotated `option` Object Example . . . 43
3.1.3 How These Parts Work Together . . . 46
3.1.4 Dynamic Behavior with `setOption` . . . 46
3.1.5 Modular Components . . . 46
3.1.6 Summary . . . 47
3.2 Series, Data, and Components . . . 47
3.2.1 What Is the `series` Object? . . . 47
3.2.2 Different Chart Types, Different Series Formats . . . 48
3.2.3 Defining Data Arrays . . . 51
3.2.4 Customizing Series . . . 52
3.2.5 How `series` Fits in the Big Picture . . . 52
3.2.6 Summary . . . 52
3.3 Configuring Titles, Legends, and Tooltips . . . 52
3.3.1 Chart Titles . . . 53
3.3.2 Legends . . . 54
3.3.3 Tooltips . . . 56
3.3.4 Putting It All Together: Example Chart with Multiple Series . . . 58
3.3.5 Summary . . . 60
3.4 Basic Styling and Themes . . . 60
3.4.1 Applying Built-in Themes . . . 60
3.4.2 Defining Custom Themes . . . 61
3.4.3 Styling Key Visual Elements . . . 63
3.4.4 Switching Themes Dynamically . . . 65
3.4.5 Summary . . . 66

4 Core Chart Types **69**
4.1 Bar Charts (Vertical and Horizontal) . . . 69
4.1.1 Vertical vs. Horizontal Bar Charts . . . 69
4.1.2 Switching Axis Orientation . . . 69
4.1.3 Grouped Bar Charts . . . 72
4.1.4 Stacked Bar Charts . . . 74
4.1.5 Real-World Example: Sales by Region . . . 76
4.1.6 Summary . . . 78
4.2 Line Charts and Area Charts . . . 78
4.2.1 Creating a Simple Line Chart . . . 78
4.2.2 Multi-Series Line Chart with Smooth Curves and Symbols . . . 80
4.2.3 Converting a Line Chart into an Area Chart . . . 82
4.2.4 Full Area Chart Example . . . 83
4.2.5 Summary . . . 86
4.3 Pie and Donut Charts . . . 86
4.3.1 Creating a Basic Pie Chart . . . 86
4.3.2 Turning a Pie Chart into a Donut Chart . . . 89
4.3.3 Exploding Slices (Highlighting) . . . 91

4.3.4 Real-World Use Cases . . . 92
4.3.5 Summary . . . 93
4.4 Scatter Plots and Bubble Charts . . . 93
4.4.1 Scatter Plots: Basic Two-Dimensional Data . . . 93
4.4.2 Example: Basic Scatter Plot . . . 93
4.4.3 Bubble Charts: Adding a Third Dimension with Size . . . 95
4.4.4 Example: Bubble Chart with Size Data . . . 95
4.4.5 Summary . . . 97
4.5 Radar Charts . . . 98
4.5.1 Understanding Radar Charts . . . 98
4.5.2 Defining Indicators and Axis Ranges . . . 98
4.5.3 Example: Comparing Multiple Product Attributes . . . 98
4.5.4 When to Use Radar Charts . . . 101
4.5.5 Summary . . . 101
4.6 Gauge and Funnel Charts . . . 101
4.6.1 Gauge Charts: Visualizing KPIs Like a Speedometer . . . 101
4.6.2 Basic Gauge Chart Example . . . 101
4.6.3 Funnel Charts: Visualizing Stages in a Process . . . 103
4.6.4 Basic Funnel Chart Example . . . 103
4.6.5 Customization Tips . . . 105
4.6.6 Summary . . . 106

5 Advanced Visualizations 108
5.1 Heatmaps . . . 108
5.1.1 What Is a Heatmap? . . . 108
5.1.2 Creating a Basic Heatmap in ECharts . . . 108
5.1.3 Example: Simple Heatmap with Color Gradient . . . 108
5.1.4 Practical Use Case: Correlation Matrix Heatmap . . . 111
5.1.5 Summary . . . 113
5.2 Tree Maps . . . 114
5.2.1 Creating a Tree Map with Nested Data . . . 114
5.2.2 Breakdown of Key Options . . . 117
5.2.3 Customizing Appearance . . . 118
5.2.4 Best Practices . . . 118
5.3 Sunburst Charts . . . 118
5.3.1 Understanding Sunburst Charts . . . 118
5.3.2 Structure of Hierarchical Data for Sunburst . . . 119
5.3.3 Creating a Basic Sunburst Chart . . . 119
5.3.4 Interactive Features . . . 121
5.3.5 Summary . . . 122
5.4 Graphs and Network Diagrams . . . 122
5.4.1 Use Cases for Graph and Network Diagrams . . . 122
5.4.2 Defining Nodes, Edges, and Categories . . . 123
5.4.3 Example Data Structure . . . 123
5.4.4 Key Concepts and Customizations . . . 125

5.4.5 Interaction Features . . . 126
5.4.6 Summary . . . 126
5.5 Geographic and Map Visualizations . . . 126
5.5.1 Using Built-in Maps with `geo` . . . 126
5.5.2 Example: Basic China Map with Regional Data . . . 126
5.5.3 Importing and Using Custom GeoJSON Maps . . . 128
5.5.4 Using the `geo` Component for Scatter and Heatmap Layers . . . 129
5.5.5 Choropleth Maps: Coloring Regions by Value . . . 131
5.5.6 Summary . . . 131
5.6 Parallel Coordinates and Other Specialized Charts . . . 132
5.6.1 What Are Parallel Coordinate Charts? . . . 132
5.6.2 Creating a Parallel Coordinate Chart in ECharts . . . 132
5.6.3 Other Specialized Chart Types in ECharts . . . 135
5.6.4 Summary . . . 135

6 Interactivity and Animation 137
6.1 Tooltip Customization . . . 137
6.1.1 Basic Tooltip Configuration . . . 137
6.1.2 Formatting Tooltip Content . . . 138
6.1.3 Using String Template . . . 138
6.1.4 Using a Callback Function . . . 140
6.1.5 Conditional Showing and Hiding . . . 141
6.1.6 Rich Text and Styling . . . 143
6.1.7 Practical Example: Comparing Multiple Series with Shared Tooltip . . . 145
6.1.8 Summary . . . 146
6.2 Zooming, Panning, and Data Zoom Components . . . 147
6.2.1 What is the `dataZoom` Component? . . . 147
6.2.2 Types of `dataZoom` . . . 147
6.2.3 Enabling Inside Zooming and Panning . . . 148
6.2.4 Adding a Slider Zoom Control . . . 149
6.2.5 Common Use Case: Timeline-Based Data Navigation . . . 150
6.2.6 Multiple Axes and Data Zoom . . . 152
6.2.7 Summary . . . 153
6.3 Highlighting and Selecting Data . . . 153
6.3.1 Highlighting and Downplaying Data . . . 153
6.3.2 Programmatic API . . . 154
6.3.3 Selecting Data Points . . . 155
6.3.4 API for Selection . . . 155
6.3.5 Enabling Interactive Highlighting on Hover . . . 157
6.3.6 Example: Custom Highlight on Hover . . . 157
6.3.7 Example: Click to Select/Deselect Points . . . 160
6.3.8 Styling Highlighted and Selected Items . . . 162
6.3.9 Summary . . . 164
6.4 Animation Settings and Transitions . . . 164
6.4.1 How Animation Works in ECharts . . . 164

6.4.2 Customizing Animation on Initial Render . . . 165
6.4.3 Animation During Data Updates . . . 166
6.4.4 Staggered Animations for Storytelling . . . 167
6.4.5 Common Animation Easing Options . . . 168
6.4.6 Summary . . . 169

7 Responsive Design and Theming 171
7.1 Making Charts Responsive . . . 171
7.1.1 Using Percentage-Based Dimensions . . . 171
7.1.2 Calling `resize()` on Window Resize . . . 171
7.1.3 Responsive Option Changes with `media` Queries . . . 172
7.1.4 Responsive Dashboards: Handling Multiple Charts . . . 174
7.1.5 Summary . . . 177
7.2 Using Built-in Themes and Creating Custom Themes . . . 178
7.2.1 Built-in ECharts Themes . . . 178
7.2.2 Applying a Built-in Theme . . . 178
7.2.3 Creating Your Own Custom Theme . . . 180
7.2.4 Step 1: Define a Theme Object . . . 180
7.2.5 Step 2: Register the Theme with ECharts . . . 181
7.2.6 Step 3: Use Your Custom Theme . . . 181
7.2.7 Tips for Custom Themes . . . 183
7.2.8 Summary . . . 183
7.3 Dark Mode and Accessibility Considerations . . . 183
7.3.1 Maintain Sufficient Contrast . . . 183
7.3.2 Use Legible Labels and Fonts . . . 184
7.3.3 Choose Accessible Color Palettes . . . 184
7.3.4 Detecting and Supporting Dark Mode . . . 184
7.3.5 Detecting Dark Mode with CSS Media Query . . . 184
7.3.6 Toggling Dark Mode in ECharts . . . 185
7.3.7 Example: Accessible Color Palette in Dark Mode . . . 185
7.3.8 Summary . . . 185

8 Data Management 188
8.1 Loading and Updating Data Dynamically . . . 188
8.1.1 Loading External Data Using `fetch` . . . 188
8.1.2 Example: Fetching Data from an API and Updating a Chart . . . 188
8.1.3 Explanation . . . 189
8.1.4 Notes on `setOption` . . . 189
8.1.5 Loading Local Data Files . . . 189
8.1.6 Summary . . . 190
8.2 Working with Large Datasets . . . 190
8.2.1 Challenges with Large Datasets . . . 190
8.2.2 Enable `large` Mode for Line and Scatter Series . . . 191
8.2.3 Use Sampling to Reduce Data Points . . . 191
8.2.4 Disable or Limit Animations . . . 191

8.2.5 Simplify Visual Elements . . . 192
8.2.6 Pagination or Data Windowing . . . 192
8.2.7 Use Aggregation or Alternative Visualizations . . . 192
8.2.8 Summary Table . . . 192
8.2.9 Example: Large Mode with Sampling . . . 192
8.3 Data Transformation and Filtering Techniques . . . 193
8.3.1 Why Transform and Filter Data? . . . 193
8.3.2 Common Data Transformation Methods in JavaScript . . . 193
8.3.3 `map()` Reshape or extract fields . . . 194
8.3.4 `filter()` Select subsets . . . 194
8.3.5 `reduce()` Aggregate or summarize . . . 194
8.3.6 Dynamic Data Filtering with ECharts . . . 194
8.3.7 Example: Filter Chart by Region . . . 194
8.3.8 Summary . . . 197

9 Customization and Extensibility 199
9.1 Creating Custom Series and Chart Types . . . 199
9.1.1 When to Use Custom Series . . . 199
9.1.2 Key APIs for Custom Series . . . 199
9.1.3 Example: Creating a Simple Spiral Series . . . 199
9.1.4 Step 1: Register the Series Model . . . 200
9.1.5 Step 2: Extend the Chart View . . . 200
9.1.6 Step 3: Use the Custom Series in an Option . . . 201
9.1.7 How It Works . . . 202
9.1.8 Summary . . . 203
9.2 Writing and Using Plugins . . . 203
9.2.1 Using Community Plugins . . . 203
9.2.2 How to use a community plugin: . . . 203
9.2.3 Writing Your Own Basic Plugin . . . 204
9.2.4 Plugin Structure . . . 204
9.2.5 Example: Adding a Watermark Plugin . . . 204
9.2.6 How This Works . . . 207
9.2.7 Summary . . . 207
9.3 Advanced Styling with Rich Text and SVG Paths . . . 207
9.3.1 Rich Text Formatting for Labels . . . 208
9.3.2 Example: Multi-line and Colored Axis Labels . . . 208
9.3.3 Embedding SVG Paths for Custom Icons . . . 210
9.3.4 Example: Using SVG Path in a MarkPoint . . . 210
9.3.5 Combining Rich Text and SVG Paths . . . 211
9.3.6 Summary . . . 211

10 Exporting and Sharing 214
10.1 Exporting Charts as Images and PDFs . . . 214
10.1.1 Exporting Charts as Images . . . 214
10.1.2 Example: Export Chart as PNG and Trigger Download . . . 214

10.1.3 Exporting Charts to PDF with jsPDF . . . 216
10.1.4 Example: Export Chart as PDF . . . 216
10.1.5 Tips for Best Results . . . 217
10.1.6 Summary . . . 217
10.2 Sharing via Embeds and IFrames . . . 217
10.2.1 Using IFrames to Embed ECharts Charts . . . 217
10.2.2 How to create an embeddable iframe: . . . 217
10.2.3 Sharing Live Examples via CodePen or Similar Platforms . . . 218
10.2.4 Example: CodePen Embed . . . 218
10.2.5 Best Practices for Embedding . . . 219
10.2.6 Summary . . . 219
10.3 Printing and Offline Usage . . . 219
10.3.1 Preparing Charts for Printing . . . 219
10.3.2 Adjust Chart Size and DPI . . . 220
10.3.3 Remove or Simplify Interactive Elements . . . 220
10.3.4 Use Static Image Snapshots . . . 220
10.3.5 Strategies for Offline Usage . . . 220
10.3.6 Bundle All Assets Locally . . . 221
10.3.7 Preload Data and Avoid Remote Fetching . . . 221
10.3.8 Use Build Tools for Bundling . . . 221
10.3.9 Summary . . . 221

11 Performance Optimization 223
11.1 Optimizing for Large Data and High FPS . . . 223
11.1.1 Enable `large` Mode for Series . . . 223
11.1.2 Use `progressive` Rendering . . . 223
11.1.3 Simplify Tooltips and Interactions . . . 223
11.1.4 Reduce Visual Complexity . . . 224
11.1.5 Performance Comparison Example . . . 224
11.1.6 Summary of Optimization Options . . . 224
11.2 Lazy Loading and Data Chunking . . . 225
11.2.1 Why Lazy Loading and Data Chunking Matter . . . 225
11.2.2 Load Data on Demand (e.g., on Zoom or Scroll) . . . 225
11.2.3 Data Chunking Strategies . . . 226
11.2.4 Use `setTimeout` to Load Data in Chunks . . . 226
11.2.5 Use `requestIdleCallback` for Background Loading (if supported) . . . 226
11.2.6 Benefits of Incremental Loading . . . 227
11.2.7 Summary . . . 227
11.3 Best Practices for Smooth Animations . . . 227
11.3.1 Key Animation Settings in ECharts . . . 227
11.3.2 Example: Smooth Transition on Data Update . . . 228
11.3.3 When to Disable Animations . . . 230
11.3.4 Summary of Animation Tuning . . . 230

Chapter 1.

Introduction

1. What Is ECharts?
2. Overview of ECharts Features
3. Setting Up Your Environment (CDN, NPM, and Development Tools)
4. Basic Concepts: Canvas, SVG, and Rendering Modes

1 Introduction

1.1 What Is ECharts?

ECharts (short for *Enterprise Charts*) is a powerful, open-source JavaScript library for building interactive, customizable, and high-performance data visualizations in the browser. Developed by Baidu, one of China's leading technology companies, ECharts has become a cornerstone of web-based charting, particularly in applications requiring responsiveness, smooth interactions, and support for large-scale datasets.

Released in 2013, ECharts has steadily grown in popularity, gaining global adoption across industries such as finance, logistics, education, and government. As a library, it strikes a balance between ease of use and advanced customization. It provides a declarative option-based API, meaning developers describe what they want to display, and ECharts takes care of rendering it efficiently, using either Canvas or SVG under the hood.

ECharts in the Data Visualization Landscape

To understand ECharts' role in the data visualization ecosystem, it's helpful to compare it to other popular libraries:

- **D3.js** is a low-level, powerful toolkit for creating bespoke visualizations by binding data to DOM elements. While it offers unmatched flexibility and control, it requires a steep learning curve and deep knowledge of SVG, the DOM, and functional JavaScript.
- **Chart.js** is a simpler, high-level library focused on ease of use, making it great for quick dashboards and common chart types. However, it offers less flexibility when customizing complex interactions or handling large datasets.
- **ECharts**, by contrast, offers a middle ground: it supports a rich variety of chart types out-of-the-box (including bar charts, line charts, pie charts, scatter plots, maps, and more) and comes with interactive features like tooltips, zooming, brushing, and real-time data updates. At the same time, it provides options for fine-tuned customization and extensions, making it suitable for both beginner developers and data professionals.

When and Why to Use ECharts

You should consider using ECharts when:

- You need **interactive** and **responsive** charts for web applications.
- Your data is **large or dynamic**, and performance is a priority.
- You want **rich features** like tooltips, legends, zoom controls, or geographic maps without building them from scratch.
- You prefer a **declarative, JSON-like configuration** rather than imperative DOM manipulation.
- You're building **dashboards** or **data platforms** that require a polished and consistent charting framework.

Whether you're developing enterprise-grade analytics tools or simply exploring data trends in a web app, ECharts is a versatile and scalable solution for turning raw data into meaningful

visual stories.

1.2 Overview of ECharts Features

ECharts offers a comprehensive set of features that make it a top-tier library for interactive, high-performance data visualizations. Whether you're building dashboards, data reports, or analytical tools, ECharts provides the components and flexibility needed to create stunning, responsive charts. This section outlines the core features that make ECharts powerful and versatile.

Theming and Styling

ECharts supports a flexible theming system that allows you to apply consistent colors, fonts, and styles across all your charts. You can choose from built-in themes such as `light`, `dark`, or `vintage`, or define your own custom themes.

```
echarts.registerTheme('myTheme', {
  backgroundColor: '#f4f4f4',
  color: ['#5470C6', '#91CC75', '#EE6666']
});
```

Then apply it:

```
const chart = echarts.init(domElement, 'myTheme');
```

Themes help maintain visual consistency across dashboards, especially in enterprise applications that require branding alignment.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ECharts Theming Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

<div id="main"></div>

<script>
  // Define and register a custom theme
```

```
  echarts.registerTheme('myTheme', {
    backgroundColor: '#f4f4f4',
    color: ['#5470C6', '#91CC75', '#EE6666'],
    textStyle: {
      fontFamily: 'Arial, sans-serif'
    },
    title: {
      textStyle: {
        color: '#333'
      }
    }
  });

  // Initialize chart with the custom theme
  const chart = echarts.init(document.getElementById('main'), 'myTheme');

  // Define chart options
  const option = {
    title: {
      text: 'Monthly Sales'
    },
    tooltip: {},
    xAxis: {
      data: ['Jan', 'Feb', 'Mar', 'Apr', 'May']
    },
    yAxis: {},
    series: [{
      name: 'Sales',
      type: 'bar',
      data: [120, 200, 150, 80, 70]
    }]
  };

  // Set the options on the chart
  chart.setOption(option);
</script>

</body>
</html>
```

Responsive Design

ECharts charts are responsive by default. They automatically adapt to changes in the container's size, including window resizes, sidebar toggles, and layout shifts. Developers can also trigger manual resize with:

```
chart.resize();
```

This is especially useful in mobile-friendly web apps or when embedding charts in dynamic layouts.

Rich Interactivity

ECharts offers a rich set of interactive behaviors, built in and ready to use with minimal configuration:

- **Tooltips** that follow the cursor and display detailed data
- **Data zooming and panning** (via mouse, touch, or controls)
- **Legend toggles** to filter series on the fly
- **Click and hover events** for drill-down or custom UI behaviors
- **Brush tools** for selecting ranges in scatter plots or timelines

```
tooltip: {
  trigger: 'axis'
},
dataZoom: [
  {
    type: 'slider',
    start: 30,
    end: 70
  }
]
```

These features make your charts feel alive—empowering users to explore and understand data directly.

Smooth Animations

ECharts provides smooth animations for loading, updating, and transitioning between states. Animations are not just visual flair—they improve usability by making changes easier to follow.

```
animationDuration: 1000,
animationEasing: 'cubicOut'
```

Animations can be configured per series or globally and are fully controllable for fine-tuning transitions.

Extensive Chart and Component Library

ECharts supports dozens of chart types and layout components out of the box:

- **Standard charts**: bar, line, pie, scatter
- **Advanced types**: radar, heatmap, gauge, funnel, candlestick
- **Maps and geographic data**: choropleth maps, geo-coordinates
- **Relational data**: graph, Sankey, tree diagrams
- **Layouts**: grid, polar, parallel, calendar, treemap

You can compose multiple charts in a single canvas and control their layout with the `grid`, `tooltip`, and `axis` systems.

Plugin and Extension Ecosystem

ECharts is modular by design. You can load only the components you need to reduce bundle size, or use community extensions like:

- **ECharts GL** for 3D rendering and GPU acceleration
- **ECharts LiquidFill** for decorative liquid charts
- **ECharts WordCloud** for text-based visualizations

- **ECharts Map Extensions** for enhanced geo features

```
import * as echarts from 'echarts/core';
import { LineChart } from 'echarts/charts';
import { TooltipComponent } from 'echarts/components';
```

This flexibility ensures your application stays lightweight and performant.

Custom Visualizations

ECharts allows developers to create entirely custom series and render functions. The `custom` series type lets you draw arbitrary shapes using low-level primitives, giving you the power of D3.js with the structure of ECharts.

```
series: [{
  type: 'custom',
  renderItem: function (params, api) {
    return {
      type: 'rect',
      shape: {
        x: api.value(0),
        y: api.value(1),
        width: 20,
        height: 20
      },
      style: {
        fill: '#5470C6'
      }
    };
  }
}]
```

This feature is ideal for unique, domain-specific charts like biological models, industrial pipelines, or scientific visuals.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ECharts Custom Visualization</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="main"></div>

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

```
  const option = {
    title: {
      text: 'Custom Rectangles Example'
    },
    xAxis: {
      type: 'value',
      min: 0,
      max: 100
    },
    yAxis: {
      type: 'value',
      min: 0,
      max: 100
    },
    series: [{
      type: 'custom',
      coordinateSystem: 'cartesian2d',
      data: [
        [10, 20],
        [30, 50],
        [60, 80]
      ],
      renderItem: function (params, api) {
        const x = api.coord([api.value(0), 0])[0];
        const y = api.coord([0, api.value(1)])[1];
        return {
          type: 'rect',
          shape: {
            x: x - 10,
            y: y - 10,
            width: 20,
            height: 20
          },
          style: {
            fill: '#5470C6'
          }
        };
      }
    }]
  };

  chart.setOption(option);
</script>

</body>
</html>
```

Example: A Responsive Bar Chart with Tooltip and Zoom

Here's a simple yet powerful example combining several core features:

```
const option = {
  title: {
    text: 'Sales by Region'
  },
  tooltip: {
    trigger: 'axis'
  },
```

```
  dataZoom: [{ type: 'slider' }],
  xAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  yAxis: {
    type: 'value'
  },
  series: [{
    name: 'Sales',
    type: 'bar',
    data: [120, 200, 150, 80]
  }]
};
```

This chart is interactive, responsive, and fully theme-able—all with minimal code.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Sales by Region Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #chart {
      width: 100%;
      max-width: 700px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="chart"></div>

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

  const option = {
    title: {
      text: 'Sales by Region'
    },
    tooltip: {
      trigger: 'axis'
    },
    dataZoom: [{ type: 'slider' }],
    xAxis: {
      type: 'category',
      data: ['North', 'South', 'East', 'West']
    },
    yAxis: {
      type: 'value'
    },
```

```
    series: [{
      name: 'Sales',
      type: 'bar',
      data: [120, 200, 150, 80],
      itemStyle: {
        color: '#5470C6'
      }
    }]
  };

  chart.setOption(option);
  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

ECharts offers more than just charting—it's a complete visualization framework designed to scale with your data, your users, and your application's complexity. Whether you're visualizing a dozen points or a million, ECharts gives you the tools to do it beautifully and efficiently.

1.3 Setting Up Your Environment (CDN, NPM, and Development Tools)

Before you start building visualizations with ECharts, you need to set up your development environment. ECharts can be integrated in multiple ways depending on your project's size, complexity, and toolchain. This section walks you through three common approaches:

- Using a **CDN** for quick prototyping
- Installing via **NPM** for full-scale projects
- Configuring **build tools** like Webpack and Vite for modern front-end workflows

Option 1: Using ECharts via CDN (Quick Prototyping)

The fastest way to start using ECharts is by including it directly from a CDN (Content Delivery Network) in an HTML file. This is ideal for learning, demos, and prototypes.

```
  <script>
    const chart = echarts.init(document.getElementById('main'));
    const option = {
      title: { text: 'Simple Bar Chart' },
      xAxis: { type: 'category', data: ['A', 'B', 'C', 'D'] },
      yAxis: { type: 'value' },
      series: [{ type: 'bar', data: [10, 22, 28, 43] }]
    };
    chart.setOption(option);
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>ECharts Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <script>
    const chart = echarts.init(document.getElementById('main'));
    const option = {
      title: { text: 'Simple Bar Chart' },
      xAxis: { type: 'category', data: ['A', 'B', 'C', 'D'] },
      yAxis: { type: 'value' },
      series: [{ type: 'bar', data: [10, 22, 28, 43] }]
    };
    chart.setOption(option);
  </script>
</body>
</html>
```

Use this approach when you need zero setup or just want to experiment with ECharts features.

Option 2: Installing ECharts via NPM (For Full Projects)

For professional or production applications, it's best to install ECharts as an NPM package. This allows version control, modular usage, and better integration with frameworks like React, Vue, or Angular.

```
mkdir echarts-project
cd echarts-project
npm init -y
```

Step 1: Initialize a Node project

```
npm install echarts
```

Step 2: Install ECharts

```
// main.js
import * as echarts from 'echarts';

const chart = echarts.init(document.getElementById('main'));
const option = {
```

```
  title: { text: 'Line Chart' },
  xAxis: { type: 'category', data: ['Jan', 'Feb', 'Mar'] },
  yAxis: { type: 'value' },
  series: [{ type: 'line', data: [120, 200, 150] }]
};
chart.setOption(option);
```

Step 3: Use it in JavaScript

You'll need an HTML file with a `<div id="main">` and a bundler like Webpack or Vite to compile and serve your project (see below).

Option 3: Using ECharts with Webpack or Vite (Modern Setup)

To get the most out of ECharts in modern front-end applications, integrate it with a build tool. Below are setup examples using **Webpack** and **Vite**.

1.3.1 Using Webpack

```
npm install echarts webpack webpack-cli webpack-dev-server html-webpack-plugin --save-dev
```

Step 1: Install dependencies

Step 2: Project Structure

```
echarts-webpack-demo/
+-- src/
    +-- index.js
+-- public/
    +-- index.html
+-- webpack.config.js
```

```
import * as echarts from 'echarts';

const chart = echarts.init(document.getElementById('main'));
chart.setOption({
  title: { text: 'Webpack + ECharts' },
  xAxis: { type: 'category', data: ['X', 'Y', 'Z'] },
  yAxis: { type: 'value' },
  series: [{ type: 'bar', data: [5, 20, 36] }]
});
```

src/index.js

```
<!DOCTYPE html>
<html>
```

```
  <head><title>Webpack ECharts</title></head>
  <body>
    <div id="main" style="width: 600px; height: 400px;"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

public/index.html

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html'
    })
  ],
  devServer: {
    static: './dist',
    open: true
  }
};
```

webpack.config.js

```
npx webpack serve
```

Step 3: Run it

1.3.2 Using Vite

Vite is a fast, modern build tool that works great with ECharts.

```
npm create vite@latest echarts-vite-demo -- --template vanilla
cd echarts-vite-demo
npm install
```

Step 1: Create a Vite project

```
npm install echarts
```

Step 2: Install ECharts

```
// main.js
import * as echarts from 'echarts';
import './style.css';

const chart = echarts.init(document.getElementById('main'));
chart.setOption({
  title: { text: 'Vite + ECharts' },
  xAxis: { type: 'category', data: ['Q1', 'Q2', 'Q3'] },
  yAxis: { type: 'value' },
  series: [{ type: 'line', data: [100, 200, 150] }]
});
```

Step 3: Modify `main.js`

```
<div id="main" style="width: 600px; height: 400px;"></div>
```

Step 4: Add target container in `index.html`

```
npm run dev
```

Step 5: Start development server

1.3.3 Summary

Setup Method	Use Case	Pros	Cons
CDN	Demos, quick testing	Zero setup, fast	Not scalable or modular
NPM	Apps, dashboards	Version control, reusable	Requires bundling
Webpack/Vite	Modern apps	Fast dev, build optimization	Initial setup needed

No matter your setup, ECharts is flexible enough to fit seamlessly into your workflow—whether you're experimenting or building complex production tools.

1.4 Basic Concepts: Canvas, SVG, and Rendering Modes

ECharts supports two main rendering modes: **Canvas** and **SVG**. Understanding the differences between them is essential for choosing the right rendering backend based on your

performance needs, interactivity, and target platforms.

1.4.1 What Is a Rendering Mode?

A *rendering mode* determines how ECharts draws graphics in the browser. This impacts rendering speed, styling flexibility, and integration with other web technologies.

- **Canvas** is a raster-based rendering engine using the `<canvas>` HTML element.
- **SVG** (Scalable Vector Graphics) is a vector-based rendering system that uses the DOM to draw and style graphics.

ECharts uses **Canvas by default**, but you can explicitly choose either mode depending on your use case.

1.4.2 Canvas Mode (Default)

Canvas draws graphics pixel-by-pixel in a single bitmap layer, providing **high performance** and **smooth animations** for complex or large datasets.

Advantages of Canvas

- **Better performance** for large datasets (thousands of points or objects)
- **Faster animation** and transitions
- **Lower memory usage** for dynamic charts
- Ideal for dashboards and real-time visualizations

Limitations of Canvas

- Harder to **style individual elements** (since no DOM elements are created)
- Not accessible for screen readers
- Interactions like hover/click require extra event handling

1.4.3 SVG Mode

SVG uses the browser's DOM tree to represent each chart element as an object. This makes styling and interaction easier but can impact performance with many elements.

Advantages of SVG

- **Easier to manipulate** via CSS and JavaScript
- **Better accessibility** (can be read by screen readers)

- Works better with **printing**, exporting, or embedding in documents
- Ideal for static charts, infographics, or when accessibility is a concern

Limitations of SVG

- **Slower rendering** with large datasets (hundreds or thousands of shapes)
- **Memory-intensive** with complex or animated charts

1.4.4 Switching Between Rendering Modes

You can switch the rendering mode when initializing the chart using the `echarts.init()` function's third parameter:

```
const chart = echarts.init(domElement, null, {
  renderer: 'svg' // or 'canvas'
});
```

- `'canvas'` (default): Fast, efficient raster rendering
- `'svg'`: Precise, styleable vector rendering

Example

```
const chart = echarts.init(document.getElementById('main'), null, {
  renderer: 'svg'
});

const option = {
  title: { text: 'SVG Rendered Chart' },
  xAxis: { type: 'category', data: ['A', 'B', 'C'] },
  yAxis: { type: 'value' },
  series: [{ type: 'bar', data: [10, 20, 30] }]
};

chart.setOption(option);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ECharts SVG Renderer</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
      border: 1px solid #ccc;
    }
  </style>
</head>
```

```
<body>

<div id="main"></div>

<script>
  // Initialize ECharts with SVG renderer
  const chart = echarts.init(document.getElementById('main'), null, {
    renderer: 'svg' // Switch between 'svg' and 'canvas' here
  });

  // Define chart options
  const option = {
    title: { text: 'SVG Rendered Chart' },
    tooltip: { trigger: 'axis' },
    xAxis: { type: 'category', data: ['A', 'B', 'C'] },
    yAxis: { type: 'value' },
    series: [{
      type: 'bar',
      data: [10, 20, 30],
      itemStyle: { color: '#91cc75' }
    }]
  };

  // Apply options
  chart.setOption(option);

  // Make chart responsive
  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

1.4.5 When to Use Each Rendering Mode

Use Case	Recommended Renderer
High-performance dashboards	`canvas`
Real-time data with animations	`canvas`
Static charts or infographics	`svg`
Accessibility or print-ready output	`svg`
Small datasets (few elements)	`svg` or `canvas`

You can experiment with both modes to see which works best for your data and device constraints.

1.4.6 Summary

Feature	Canvas	SVG
Rendering Type	Raster (bitmap)	Vector (DOM-based)
Performance	Excellent for large data	Slower for large data
Styling	Limited (no CSS or DOM access)	Flexible (CSS and DOM support)
Accessibility	Poor (no semantic elements)	Good (screen reader support)
Animation	Smooth and efficient	Slower and heavier
Use Case	Dashboards, dynamic visualizations	Static charts, accessibility-focused charts

ECharts gives you the freedom to choose the best rendering mode for your needs—whether you prioritize speed, accessibility, or graphic precision.

Chapter 2.

Getting Started with ECharts

1. Installing and Importing ECharts
2. Creating Your First Chart (Bar Chart)
3. ECharts Core Concepts: Option Object and API
4. Anatomy of an ECharts Chart

2 Getting Started with ECharts

2.1 Installing and Importing ECharts

Before creating charts with ECharts, you'll need to install it in your project. ECharts supports two primary installation methods:

- **CDN** for quick prototyping or small projects
- **NPM** for scalable, modular applications with modern development workflows

This section covers both methods and shows how to import ECharts modules properly in a modern JavaScript setup, including guidance for efficient bundling and tree-shaking.

2.1.1 Option 1: Using ECharts via CDN (Quick Setup)

If you're working with plain HTML and want to try ECharts without any build tools, use the CDN method.

Example

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>ECharts via CDN</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="main" style="width: 600px; height: 400px;"></div>
  <script>
    const chart = echarts.init(document.getElementById('main'));
    const option = {
      title: { text: 'Basic Example' },
      xAxis: { type: 'category', data: ['A', 'B', 'C'] },
      yAxis: { type: 'value' },
      series: [{ type: 'bar', data: [5, 20, 36] }]
    };
    chart.setOption(option);
  </script>
</body>
</html>
```

Use this approach for quick tests or embedding charts in static pages.

2.1.2 Option 2: Installing ECharts via NPM (Modern Projects)

For larger or production-level applications, install ECharts as a dependency using NPM. This enables better code management, modular usage, and integration with build tools like Webpack or Vite.

Step 1: Install ECharts

```
npm install echarts
```

2.1.3 Importing ECharts in a Modern Setup

ECharts is fully modular, which means you can import only the components and chart types you need. This enables **tree-shaking**, reducing your final bundle size.

Minimal Example (ES Modules)

```
// Import the core ECharts object
import * as echarts from 'echarts/core';

// Import specific chart type and components
import { BarChart } from 'echarts/charts';
import {
  TitleComponent,
  TooltipComponent,
  GridComponent,
  LegendComponent
} from 'echarts/components';

// Import the renderer (Canvas or SVG)
import { CanvasRenderer } from 'echarts/renderers';

// Register the components you plan to use
echarts.use([
  TitleComponent,
  TooltipComponent,
  GridComponent,
  LegendComponent,
  BarChart,
  CanvasRenderer
]);

// Initialize the chart
const chart = echarts.init(document.getElementById('main'));
const option = {
  title: { text: 'Bar Chart' },
  tooltip: {},
  xAxis: { type: 'category', data: ['Red', 'Blue', 'Green'] },
  yAxis: { type: 'value' },
  series: [{ type: 'bar', data: [23, 45, 31] }]
};
chart.setOption(option);
```

This approach supports **tree-shaking** when using Webpack, Vite, or Rollup—removing unused code from the final bundle.

2.1.4 Notes on Module Bundlers

- **Webpack**, **Vite**, and **Rollup** work seamlessly with ECharts' modular imports.
- If you're using a framework (like React, Vue, or Angular), this modular structure helps you optimize load times and reduce unnecessary chart code.

2.1.5 Summary

Method	Use When	Pros	Cons
CDN	Prototyping, static HTML	Fast and easy	No bundling, larger file
NPM + Modules	Full apps, SPAs, frameworks	Tree-shaking, modular, scalable	Requires bundler setup

By selecting the right setup for your project and importing only what you need, you can take full advantage of ECharts' performance and flexibility from the very start.

2.2 Creating Your First Chart (Bar Chart)

2.2.1 Creating Your First Chart (Bar Chart)

Now that you have ECharts installed and imported, it's time to create your first visualization—a **vertical bar chart**. In this section, we'll build a basic bar chart step by step, explaining how the HTML and JavaScript parts work together to render the chart on the screen.

2.2.2 What Well Build

A simple vertical bar chart that displays sales figures for four products.

2.2.3 Step-by-Step Example Using CDN

This example uses the ECharts **CDN version** for quick setup. You can later apply the same concepts in NPM-based projects.

Full HTML Example

```
  <script>
    // Step 1: Select the container element
    const chartDom = document.getElementById('main');

    // Step 2: Initialize the chart
    const myChart = echarts.init(chartDom);

    // Step 3: Define the chart options
    const option = {
      title: {
        text: 'Product Sales (Q1)'
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['Product A', 'Product B', 'Product C', 'Product D']
      },
      yAxis: {
        type: 'value'
      },
      series: [
        {
          name: 'Sales',
          type: 'bar',
          data: [120, 200, 150, 80]
        }
      ]
    };

    // Step 4: Set the option on the chart
    myChart.setOption(option);
  </script>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>My First ECharts Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: auto;
    }
  </style>
</head>
```

```
<body>
  <h2 style="text-align: center;">Product Sales</h2>
  <div id="main"></div>

  <script>
    // Step 1: Select the container element
    const chartDom = document.getElementById('main');

    // Step 2: Initialize the chart
    const myChart = echarts.init(chartDom);

    // Step 3: Define the chart options
    const option = {
      title: {
        text: 'Product Sales (Q1)'
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['Product A', 'Product B', 'Product C', 'Product D']
      },
      yAxis: {
        type: 'value'
      },
      series: [
        {
          name: 'Sales',
          type: 'bar',
          data: [120, 200, 150, 80]
        }
      ]
    };

    // Step 4: Set the option on the chart
    myChart.setOption(option);
  </script>
</body>
</html>
```

2.2.4 Explanation of Each Part

HTML Structure

```
<div id="main"></div>
```

- This `div` acts as the container where the chart will be drawn.
- Its dimensions are defined in CSS to control the size of the chart.

Initialize the Chart

```
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);
```

- `echarts.init()` attaches a new chart instance to the DOM element.
- You can later reuse `myChart` to update or resize the chart.

Define the `option` Object

```
const option = { ... };
```

- This object tells ECharts **what to render and how**.
- Common properties include:
  - `title`: Sets the chart title.
  - `tooltip`: Enables tooltip interactivity.
  - `xAxis`, `yAxis`: Define the axes (category vs. value).
  - `series`: Holds the main data to visualize.

Render the Chart

```
myChart.setOption(option);
```

- This applies the configuration and draws the chart in the selected container.

2.2.5 Notes

- If the chart doesn't display, make sure the container has a non-zero width and height.
- You can call `myChart.resize()` if the chart container changes size (e.g., after a window resize or layout shift).

2.2.6 Whats Next?

You've now created a fully functional ECharts bar chart with just a few lines of code. In the next sections, we'll dive deeper into the **option object structure** and explore how to build more complex and interactive charts.

2.3 ECharts Core Concepts: Option Object and API

At the heart of every ECharts chart is the **`option` object**. This configuration object defines *what* the chart should display and *how* it should behave—from chart type and data to axes, tooltips, legends, and more.

In addition to the `option` object, ECharts provides a small but powerful **API** that lets you

interact with and manage chart instances programmatically.

This section introduces the key parts of the `option` object and the core methods you'll use to control an ECharts chart instance.

2.3.1 The `option` Object: Chart Configuration

The `option` object is a plain JavaScript object that describes every visual and interactive aspect of your chart. Once passed into the chart via `setOption`, ECharts uses it to render and update the chart.

Key Sections of the `option` Object

Let's look at the most commonly used sections:

`series` The Heart of the Data

The `series` array defines the actual data and chart type. Each object in the array represents a dataset with a specific chart type.

```
series: [{
  name: 'Sales',
  type: 'bar',           // Chart type: bar, line, pie, etc.
  data: [120, 200, 150, 80]
}]
```

> Each chart type has its own set of options. For example, `line` charts can use `smooth: true`, while `pie` charts use `radius` and `center`.

`xAxis` / `yAxis` Coordinate Axes

These sections define the axes in Cartesian (grid-based) charts like bar or line charts.

```
xAxis: {
  type: 'category',         // category for labels, value for numeric
  data: ['Q1', 'Q2', 'Q3']
},
yAxis: {
  type: 'value'
}
```

- Use `type: 'category'` for discrete labels (e.g., months).
- Use `type: 'value'` for numeric or continuous data.

`tooltip` Interactive Data Hints

Tooltips provide contextual information when hovering over chart elements.

```
tooltip: {
  trigger: 'axis',          // 'item' or 'axis'
  formatter: '{b}: {c}'     // Optional custom text
}
```

- `trigger: 'item'`: Tooltip for a single point (e.g., pie slice).
- `trigger: 'axis'`: Tooltip shared across an axis (e.g., line chart).

`legend` Series Control

The legend lets users toggle visibility of different series in the chart.

```
legend: {
  data: ['Sales', 'Profit'],
  top: 'top'
}
```

- Each `series.name` should be included in the `legend.data` array.
- You can position the legend with `top`, `left`, `right`, or `bottom`.

Other Common Options

```
title: {
  text: 'Quarterly Revenue'
},
grid: {
  left: '10%',
  right: '10%',
  containLabel: true
},
color: ['#5470C6', '#91CC75'] // Custom series colors
```

These enhance layout, appearance, and theming.

2.3.2 Core Chart Instance API

Once the chart is created using `echarts.init()`, you control it with several key methods:

`setOption(option)`

Applies the configuration and renders the chart. You can call this multiple times to update or animate changes.

```
myChart.setOption(option);
```

> If you pass a new `option` that modifies only part of the chart (e.g., updates the data), ECharts will automatically merge it with the existing configuration.

`resize()`

Adjusts the chart size when the container's dimensions change (e.g., after a window resize or layout shift).

```
window.addEventListener('resize', () => {
  myChart.resize();
});
```

dispose()

Completely destroys the chart instance and frees memory. Use this before removing the chart from the DOM or replacing it.

```
echarts.dispose(myChart);
```

2.3.3 Example: Option API Together

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

const option = {
  title: { text: 'Monthly Sales' },
  tooltip: { trigger: 'axis' },
  legend: { data: ['2024'] },
  xAxis: {
    type: 'category',
    data: ['Jan', 'Feb', 'Mar']
  },
  yAxis: { type: 'value' },
  series: [{
    name: '2024',
    type: 'bar',
    data: [500, 600, 800]
  }]
};

chart.setOption(option);

// Responsive
window.addEventListener('resize', () => {
  chart.resize();
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Monthly Sales Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="main"></div>
```

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

  const option = {
    title: { text: 'Monthly Sales' },
    tooltip: { trigger: 'axis' },
    legend: { data: ['2024'] },
    xAxis: {
      type: 'category',
      data: ['Jan', 'Feb', 'Mar']
    },
    yAxis: { type: 'value' },
    series: [{
      name: '2024',
      type: 'bar',
      data: [500, 600, 800],
      itemStyle: {
        color: '#5470C6'
      }
    }]
  };

  chart.setOption(option);

  // Handle responsive resize
  window.addEventListener('resize', () => {
    chart.resize();
  });
</script>

</body>
</html>
```

2.3.4 Summary

Concept	Purpose
option	Describes the full chart: data, axes, tooltips, appearance
setOption	Applies or updates the chart configuration
resize	Adjusts the chart when container size changes
dispose	Destroys the chart instance to free resources

The `option` object is your main tool for building and customizing charts, while the API gives you control over the chart's lifecycle and behavior. Mastering these core concepts is key to building rich, dynamic visualizations with ECharts.

2.4 Anatomy of an ECharts Chart

An ECharts chart is more than just a picture—it's a layered, interactive system composed of rendering logic, visual components, and data bindings. To fully understand how to build and customize charts, it's important to learn how ECharts is structured internally.

In this section, we'll break down a complete ECharts chart into its core parts:

- **Canvas container**
- **Rendering engine**
- **Option object (data and configuration)**
- **Chart components (like axes, tooltip, legend, and grid)**

We'll also walk through an annotated code example that shows how these parts come together to render a typical chart.

2.4.1 Core Layers of an ECharts Chart

Here's a high-level overview of the main building blocks:

Layer	Description
1. DOM Container	The HTML element (usually a `<div>`) where the chart is rendered
2. Rendering Engine	Either `Canvas` or `SVG` — draws the chart on screen
3. Option Object	The JavaScript object that defines the chart structure, data, and appearance
4. Data-Binding Layer	Binds your data to visual elements like bars, lines, or points
5. Components	Modular visual elements: grid, axes, tooltips, legends, titles, etc.

2.4.2 Annotated ECharts Chart Example

```
<div id="main" style="width: 600px; height: 400px;"></div>

<script>
import * as echarts from 'echarts';

const chart = echarts.init(document.getElementById('main'), null, {
  renderer: 'canvas' // Or 'svg'
});

const option = {
  title: {
    text: 'Sales Overview'          // Chart Title (Component)
```

```
  },
  tooltip: {
    trigger: 'axis'                // Tooltip (Component)
  },
  legend: {
    data: ['2024']                 // Legend (Component)
  },
  grid: {
    left: '10%',                   // Grid (Component) - spacing/layout
    right: '10%',
    containLabel: true
  },
  xAxis: {
    type: 'category',              // xAxis (Component)
    data: ['Q1', 'Q2', 'Q3', 'Q4']
  },
  yAxis: {
    type: 'value'                  // yAxis (Component)
  },
  series: [{
    name: '2024',
    type: 'bar',                   // Chart Type: bar, line, etc.
    data: [120, 200, 150, 80]      // Data-binding layer
  }]
};

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

2.4.3 Breakdown of the Example

Code Part	What It Does
`document.getElementById('main')`	Selects the DOM container (canvas placeholder)
`echarts.init(..., { renderer })`	Initializes the chart with rendering mode (`canvas` or `svg`)
`title`	Adds a main chart title
`tooltip`	Enables hover-based data tooltips
`legend`	Displays a clickable legend to show/hide data series
`grid`	Sets internal spacing (e.g., between axes and the chart area)
`xAxis` / `yAxis`	Defines coordinate axes and their type
`series`	Binds the actual data and selects a chart type (e.g., bar)

2.4.4 Rendering Engine: Canvas vs. SVG

You can control how the chart is drawn by specifying the `renderer` when initializing:

```
echarts.init(domElement, null, {
  renderer: 'canvas' // default
});
```

- `canvas`: Better for large datasets and fast animations.
- `svg`: Easier to style with CSS and supports better accessibility.

2.4.5 Summary

Part	Role
Container	Holds the chart (`<div id="main">`)
Renderer	Canvas or SVG drawing backend
Option	Configuration object that defines everything
Components	Modular chart features: grid, tooltip, legend, axes
Series	Chart type and bound data

By understanding the anatomy of an ECharts chart, you can confidently structure and customize your visualizations. In later chapters, you'll explore how to add, remove, or style each of these components for more advanced use cases.

Chapter 3.

Chart Fundamentals

1. Understanding the Option Object Structure
2. Series, Data, and Components
3. Configuring Titles, Legends, and Tooltips
4. Basic Styling and Themes

3 Chart Fundamentals

3.1 Understanding the Option Object Structure

The **`option` object** is the foundation of every ECharts visualization. It defines *what* the chart should display and *how* it should appear and behave. This object follows a **hierarchical structure** where each key corresponds to a chart component or configuration area.

In this section, we'll break down the structure of the `option` object, explain the role of its main sections, and walk through an annotated example to show how these parts work together.

3.1.1 The Hierarchical Structure

The `option` object is organized like a nested configuration tree:

```
option
+-- title        → Chart title
+-- tooltip      → Hover tooltips
+-- legend       → Series labels for toggling visibility
+-- xAxis        → Horizontal axis config
+-- yAxis        → Vertical axis config
+-- series       → Chart type and data
+-- grid         → Layout and spacing
+-- color        → Series color palette
+-- animation    → Animation settings
+-- ...          → Many more optional components
```

Each key in the `option` object corresponds to a *component*, and the values define how that component behaves or appears.

3.1.2 Annotated `option` Object Example

Here's a complete `option` object for a simple bar chart, with comments explaining each part:

```
const option = {
  // Title at the top of the chart
  title: {
    text: 'Quarterly Sales',
    left: 'center'  // Center-align the title
  },

  // Tooltip that appears when hovering over data
  tooltip: {
```

```
    trigger: 'axis' // Trigger tooltip on axis hover (shared)
  },

  // Legend to identify and toggle data series
  legend: {
    data: ['2024'],
    top: '10px'
  },

  // Layout settings for the grid (chart area)
  grid: {
    left: '10%',
    right: '10%',
    bottom: '15%',
    containLabel: true
  },

  // X-axis configuration (categorical labels)
  xAxis: {
    type: 'category',
    data: ['Q1', 'Q2', 'Q3', 'Q4']
  },

  // Y-axis configuration (numeric values)
  yAxis: {
    type: 'value'
  },

  // The actual data and chart type
  series: [
    {
      name: '2024',
      type: 'bar',
      data: [120, 200, 150, 80],
      barWidth: '50%',
      itemStyle: {
        color: '#5470C6' // Custom bar color
      }
    }
  ]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Quarterly Sales Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 50px auto;
    }
  </style>
```

```
</head>
<body>

<div id="main"></div>

<script>
  // Initialize the chart
  const chart = echarts.init(document.getElementById('main'));

  // Option object with full configuration
  const option = {
    // Title at the top of the chart
    title: {
      text: 'Quarterly Sales',
      left: 'center'  // Center-align the title
    },

    // Tooltip that appears when hovering over data
    tooltip: {
      trigger: 'axis' // Trigger tooltip on axis hover (shared)
    },

    // Legend to identify and toggle data series
    legend: {
      data: ['2024'],
      top: '10px'
    },

    // Layout settings for the grid (chart area)
    grid: {
      left: '10%',
      right: '10%',
      bottom: '15%',
      containLabel: true
    },

    // X-axis configuration (categorical labels)
    xAxis: {
      type: 'category',
      data: ['Q1', 'Q2', 'Q3', 'Q4']
    },

    // Y-axis configuration (numeric values)
    yAxis: {
      type: 'value'
    },

    // The actual data and chart type
    series: [
      {
        name: '2024',
        type: 'bar',
        data: [120, 200, 150, 80],
        barWidth: '50%',
        itemStyle: {
          color: '#5470C6' // Custom bar color
        }
      }
```

```
    ]
  };

  // Apply the options
  chart.setOption(option);

  // Make it responsive
  window.addEventListener('resize', () => {
    chart.resize();
  });
</script>

</body>
</html>
```

3.1.3 How These Parts Work Together

Component	Role
`title`	Displays a descriptive heading above the chart
`tooltip`	Enhances interactivity by showing dynamic info on hover
`legend`	Lets users identify and toggle visible data series
`xAxis/yAxis`	Define the horizontal and vertical axis behavior and appearance
`grid`	Controls spacing and layout within the canvas area
`series`	Contains the actual data and chart type (e.g., bar, line)

3.1.4 Dynamic Behavior with `setOption`

Once you've defined the `option` object, pass it to a chart instance using:

```
chart.setOption(option);
```

You can call `setOption()` again later to update specific parts—like changing the `series` data or title—without rebuilding the entire chart.

3.1.5 Modular Components

ECharts supports dozens of additional optional components that follow the same structure:

Component	Key in `option`
Chart title	`title`

Component	Key in `option`
Tooltip	`tooltip`
Legend	`legend`
Grid layout	`grid`
Axes	`xAxis`, `yAxis`
VisualMap	`visualMap`
Dataset	`dataset`
Toolbox buttons	`toolbox`
Media queries	`media`

Each component is **self-contained**, meaning you can enable or remove them without affecting unrelated parts.

3.1.6 Summary

- The `option` object is the **central configuration** for ECharts.
- It follows a **hierarchical structure** where each top-level key represents a visual or interactive component.
- Main components include: `title`, `tooltip`, `legend`, `xAxis`, `yAxis`, and `series`.
- Each component is modular and highly customizable.
- You use `chart.setOption(option)` to render or update the chart.

  YES Mastering the `option` object is the key to unlocking ECharts' full power. In the next sections, we'll explore each major component in more depth—starting with the `series` and data structure.

3.2 Series, Data, and Components

At the core of every ECharts visualization lies the **`series`** object. This is where your data meets the chart type, defining exactly *how* the information should be visualized. Understanding the `series` object is essential to harnessing the flexibility and power of ECharts.

3.2.1 What Is the `series` Object?

- The `series` key in the `option` object is **an array** of one or more data series.
- Each element in this array represents a *dataset* rendered as a specific chart type (bar, line, pie, radar, etc.).

- You can define multiple series to combine different datasets or chart types in one visualization.

3.2.2 Different Chart Types, Different Series Formats

The shape and options of each `series` object depend on the chart type. Here are two examples to illustrate:

Example 1: Bar Chart Series

```
series: [{
  name: 'Sales',
  type: 'bar',          // Bar chart type
  data: [120, 200, 150, 80], // Array of numerical values
  barWidth: '40%',      // Customize bar width
  itemStyle: {
    color: '#5470C6'    // Bar color
  }
}]
```

- The `data` array contains numeric values corresponding to the categories on the x-axis.
- Each value renders as a vertical bar.
- You can customize appearance via `itemStyle`.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Quarterly Sales Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 50px auto;
    }
  </style>
</head>
<body>

<div id="main"></div>

<script>
  // Initialize the chart
  const chart = echarts.init(document.getElementById('main'));

  // Option object with full configuration
  const option = {
    // Title at the top of the chart
    title: {
```

```
      text: 'Quarterly Sales',
      left: 'center'  // Center-align the title
    },

    // Tooltip that appears when hovering over data
    tooltip: {
      trigger: 'axis' // Trigger tooltip on axis hover (shared)
    },

    // Legend to identify and toggle data series
    legend: {
      data: ['2024'],
      top: '10px'
    },

    // Layout settings for the grid (chart area)
    grid: {
      left: '10%',
      right: '10%',
      bottom: '15%',
      containLabel: true
    },

    // X-axis configuration (categorical labels)
    xAxis: {
      type: 'category',
      data: ['Q1', 'Q2', 'Q3', 'Q4']
    },

    // Y-axis configuration (numeric values)
    yAxis: {
      type: 'value'
    },

    // The actual data and chart type
    series: [
      {
        name: '2024',
        type: 'bar',
        data: [120, 200, 150, 80],
        barWidth: '50%',
        itemStyle: {
          color: '#5470C6' // Custom bar color
        }
      }
    ]
  };

  // Apply the options
  chart.setOption(option);

  // Make it responsive
  window.addEventListener('resize', () => {
    chart.resize();
  });
</script>

</body>
```

```
</html>
```

Example 2: Radar Chart Series

```
series: [{
  name: 'Performance',
  type: 'radar',          // Radar chart type
  data: [{
    value: [80, 90, 70, 85, 75],  // Array matching radar indicators
    name: 'Team A',
    areaStyle: {           // Fill color under the radar shape
      color: 'rgba(84,112,198,0.4)'
    }
  }]
}]
```

- The `data` here is an array of objects instead of just numbers.
- Each `value` array corresponds to the radar's indicators (dimensions).
- You can customize styling with `areaStyle`, `lineStyle`, and more.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Radar Chart - Team Performance</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #radarChart {
      width: 600px;
      height: 400px;
      margin: 50px auto;
    }
  </style>
</head>
<body>

<div id="radarChart"></div>

<script>
  const radar = echarts.init(document.getElementById('radarChart'));

  const option = {
    title: {
      text: 'Team A Performance',
      left: 'center'
    },
    tooltip: {},
    radar: {
      indicator: [
        { name: 'Speed', max: 100 },
        { name: 'Strength', max: 100 },
        { name: 'Endurance', max: 100 },
        { name: 'Skill', max: 100 },
        { name: 'Teamwork', max: 100 }
```

```
      ]
    },
    series: [{
      name: 'Performance',
      type: 'radar',
      data: [{
        value: [80, 90, 70, 85, 75],
        name: 'Team A',
        areaStyle: {
          color: 'rgba(84,112,198,0.4)'
        },
        lineStyle: {
          color: '#5470C6'
        },
        symbol: 'circle',
        symbolSize: 6,
        itemStyle: {
          color: '#5470C6'
        }
      }]
    }]
  };

  radar.setOption(option);

  window.addEventListener('resize', () => radar.resize());
</script>

</body>
</html>
```

3.2.3 Defining Data Arrays

- For **Cartesian charts** (bar, line, scatter), `data` is usually an array of numbers or `[x, y]` pairs.
- For **pie charts**, `data` is an array of objects with `value` and `name` properties:

```
series: [{
  type: 'pie',
  data: [
    { value: 1048, name: 'Apples' },
    { value: 735, name: 'Oranges' },
    { value: 580, name: 'Bananas' }
  ]
}]
```

- For **custom or complex charts**, data formats can vary widely, but always correspond to the chart type's requirements.

3.2.4 Customizing Series

You can fine-tune many aspects of a series:

Option	Description
`name`	Label used in legends and tooltips
`type`	Chart type (bar, line, pie, radar…)
`data`	Data values or objects
`itemStyle`	Styling for individual data elements
`label`	Text labels on data points
`emphasis`	Styles on hover or focus
`stack`	Stack multiple series (for bar/line)
`smooth`	Smooth curves for line charts

3.2.5 How `series` Fits in the Big Picture

The `series` data visually maps to other components:

- **Legend** uses `series.name` to display labels.
- **Tooltip** shows `series.name` and `data` details on hover.
- **Axes** provide coordinate systems for Cartesian series.

3.2.6 Summary

- The `series` array defines the **core data and chart type**.
- Different chart types require **different data formats** within the series.
- You can customize appearance and behavior with many series-level options.
- Multiple series enable **multi-layered or combined charts**.

Mastering the `series` object lets you transform raw data into meaningful visual insights using ECharts' rich chart types and features.

3.3 Configuring Titles, Legends, and Tooltips

Titles, legends, and tooltips are essential components that enhance the usability and readability of your charts. This section demonstrates how to add and customize these elements in ECharts, using examples with multiple series to show how legends and tooltips work together to improve interactivity.

3.3.1 Chart Titles

The **title** component displays descriptive text above your chart, helping users quickly understand what the visualization represents.

Basic Title Configuration

```
title: {
  text: 'Monthly Revenue',
  left: 'center',       // Horizontal alignment: 'left', 'center', 'right'
  top: '10px',          // Vertical position (can be pixels or percentages)
  textStyle: {
    color: '#333',
    fontSize: 18
  }
}
```

- `text` specifies the title text.
- `left` and `top` control the position.
- Use `textStyle` to customize font, color, and size.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chart Title Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #chart {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="chart"></div>

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

  const option = {
    title: {
      text: 'Monthly Revenue',
      left: 'center',    // 'left' | 'center' | 'right'
      top: '10px',
      textStyle: {
        color: '#333',
        fontSize: 18
      }
    },
    tooltip: { trigger: 'axis' },
    xAxis: {
```

```
      type: 'category',
      data: ['Jan', 'Feb', 'Mar', 'Apr', 'May']
    },
    yAxis: { type: 'value' },
    series: [{
      name: 'Revenue',
      type: 'bar',
      data: [3000, 4500, 4200, 5200, 4800],
      itemStyle: {
        color: '#5470C6'
      }
    }]
  };

  chart.setOption(option);

  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

3.3.2 Legends

The **legend** helps users identify different series and toggle their visibility. This is particularly useful for charts with multiple data series.

Legend Example with Multiple Series

```
legend: {
  data: ['Sales 2023', 'Sales 2024'],
  orient: 'horizontal',     // 'horizontal' (default) or 'vertical'
  top: 'bottom',            // Position: 'top', 'bottom', 'left', 'right' or pixel value
  left: 'center',
  textStyle: {
    color: '#666'
  }
}
```

- `data` contains the names of the series to display in the legend.
- `orient` changes the layout direction.
- `top` and `left` control positioning.
- Clicking legend items toggles the visibility of corresponding series.

Full runnable code:

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

```
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #chart {
      width: 700px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="chart"></div>

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

  const option = {
    title: {
      text: 'Sales Comparison',
      left: 'center'
    },
    tooltip: { trigger: 'axis' },
    legend: {
      data: ['Sales 2023', 'Sales 2024'],
      orient: 'horizontal',     // 'horizontal' or 'vertical'
      top: 'bottom',
      left: 'center',
      textStyle: {
        color: '#666'
      }
    },
    xAxis: {
      type: 'category',
      data: ['Q1', 'Q2', 'Q3', 'Q4']
    },
    yAxis: { type: 'value' },
    series: [
      {
        name: 'Sales 2023',
        type: 'bar',
        data: [3200, 4000, 3500, 3900],
        itemStyle: { color: '#91CC75' }
      },
      {
        name: 'Sales 2024',
        type: 'bar',
        data: [3700, 4200, 3900, 4500],
        itemStyle: { color: '#5470C6' }
      }
    ]
  };

  chart.setOption(option);

  window.addEventListener('resize', () => chart.resize());
</script>

</body>
```

```
</html>
```

3.3.3 Tooltips

Tooltips provide contextual information when users hover over chart elements.

Configuring Tooltips

```
tooltip: {
  trigger: 'axis',           // 'item' for individual points, 'axis' for all points along axis
  formatter: function(params) {
    let tooltipText = params[0].axisValue + '<br/>';
    params.forEach(item => {
      tooltipText += `${item.seriesName}: ${item.data}<br/>`;
    });
    return tooltipText;
  },
  backgroundColor: 'rgba(50, 50, 50, 0.7)',
  textStyle: {
    color: '#fff'
  }
}
```

- `trigger: 'axis'` shows tooltips for all series aligned on the same axis value.
- `trigger: 'item'` shows tooltips for individual data points.
- `formatter` customizes the tooltip content.
- Style the tooltip background and text for better appearance.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tooltip Formatter Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #chart {
      width: 700px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="chart"></div>

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

  const option = {
```

```
    title: {
      text: 'Monthly Sales',
      left: 'center'
    },
    tooltip: {
      trigger: 'axis',
      formatter: function(params) {
        let tooltipText = params[0].axisValue + '<br/>';
        params.forEach(item => {
          tooltipText += `${item.seriesName}: ${item.data} units<br/>`;
        });
        return tooltipText;
      },
      backgroundColor: 'rgba(50, 50, 50, 0.7)',
      textStyle: {
        color: '#fff'
      }
    },
    legend: {
      data: ['2023', '2024'],
      top: 'bottom'
    },
    xAxis: {
      type: 'category',
      data: ['Jan', 'Feb', 'Mar', 'Apr']
    },
    yAxis: {
      type: 'value'
    },
    series: [
      {
        name: '2023',
        type: 'bar',
        data: [500, 700, 600, 800],
        itemStyle: { color: '#91CC75' }
      },
      {
        name: '2024',
        type: 'bar',
        data: [550, 750, 650, 850],
        itemStyle: { color: '#5470C6' }
      }
    ]
  };

  chart.setOption(option);
  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

3.3.4 Putting It All Together: Example Chart with Multiple Series

```
const option = {
  title: {
    text: 'Quarterly Sales Comparison',
    left: 'center'
  },
  tooltip: {
    trigger: 'axis',
    formatter: function(params) {
      let content = params[0].axisValue + '<br/>';
      params.forEach(item => {
        content += `${item.seriesName}: ${item.data}<br/>`;
      });
      return content;
    }
  },
  legend: {
    data: ['2023', '2024'],
    top: 'bottom',
    left: 'center'
  },
  xAxis: {
    type: 'category',
    data: ['Q1', 'Q2', 'Q3', 'Q4']
  },
  yAxis: {
    type: 'value'
  },
  series: [
    {
      name: '2023',
      type: 'bar',
      data: [150, 230, 224, 218],
      itemStyle: { color: '#5470C6' }
    },
    {
      name: '2024',
      type: 'bar',
      data: [180, 260, 210, 230],
      itemStyle: { color: '#91CC75' }
    }
  ]
};
```

This configuration:

- Displays a centered title.
- Shows a legend at the bottom for two series.
- Uses axis-triggered tooltips that show values for both series on hover.
- Uses different colors to distinguish the series visually.

Full runnable code:

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

```
<head>
  <meta charset="UTF-8">
  <title>Quarterly Sales Comparison</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #chart {
      width: 800px;
      height: 450px;
      margin: 40px auto;
    }
  </style>
</head>
<body>

<div id="chart"></div>

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

  const option = {
    title: {
      text: 'Quarterly Sales Comparison',
      left: 'center'
    },
    tooltip: {
      trigger: 'axis',
      formatter: function(params) {
        let content = params[0].axisValue + '<br/>';
        params.forEach(item => {
          content += `${item.seriesName}: ${item.data}<br/>`;
        });
        return content;
      }
    },
    legend: {
      data: ['2023', '2024'],
      top: 'bottom',
      left: 'center'
    },
    xAxis: {
      type: 'category',
      data: ['Q1', 'Q2', 'Q3', 'Q4']
    },
    yAxis: {
      type: 'value'
    },
    series: [
      {
        name: '2023',
        type: 'bar',
        data: [150, 230, 224, 218],
        itemStyle: { color: '#5470C6' }
      },
      {
        name: '2024',
        type: 'bar',
        data: [180, 260, 210, 230],
        itemStyle: { color: '#91CC75' }
```

```
      }
    ]
  };

  chart.setOption(option);

  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

3.3.5 Summary

Component	Key Configuration Options	Purpose
Title	`text`, `left`, `top`, `textStyle`	Display chart heading
Legend	`data`, `orient`, `top`, `left`, `textStyle`	Show series labels and toggle visibility
Tooltip	`trigger`, `formatter`, `backgroundColor`, `textStyle`	Show interactive hover info

By customizing titles, legends, and tooltips, you improve the clarity and interactivity of your charts—making your data insights easier to understand and explore.

3.4 Basic Styling and Themes

Styling is a powerful way to make your charts visually appealing and aligned with your design requirements. ECharts offers **built-in themes**, supports **custom theme definitions**, and provides many options to fine-tune colors, labels, and other visual elements.

This section explains how to apply and customize themes, modify chart appearance using style properties, and dynamically switch themes.

3.4.1 Applying Built-in Themes

ECharts comes with several built-in themes like `'light'`, `'dark'`, and others. You can apply a theme when initializing the chart:

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

- The second argument specifies the theme name.
- Using `'dark'` switches the chart's background, axis colors, and default palette to a dark style.
- If you omit the theme, it defaults to the `'light'` theme.

3.4.2 Defining Custom Themes

You can create your own theme by defining a JavaScript object that overrides colors, fonts, and styles.

Example custom theme registration:

```
echarts.registerTheme('myTheme', {
  color: ['#5470C6', '#91CC75', '#FAC858', '#EE6666'],
  backgroundColor: '#f0f2f5',
  textStyle: {
    fontFamily: 'Arial, sans-serif'
  },
  title: {
    textStyle: {
      color: '#333',
      fontWeight: 'bold'
    }
  },
  tooltip: {
    backgroundColor: 'rgba(50,50,50,0.7)',
    textStyle: { color: '#fff' }
  }
});
```

Then initialize with the custom theme:

```
const chart = echarts.init(document.getElementById('main'), 'myTheme');
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Custom Theme Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main {
      width: 700px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
```

```
<body>

<div id="main"></div>

<script>
  // Register custom theme
  echarts.registerTheme('myTheme', {
    color: ['#5470C6', '#91CC75', '#FAC858', '#EE6666'],
    backgroundColor: '#f0f2f5',
    textStyle: {
      fontFamily: 'Arial, sans-serif'
    },
    title: {
      textStyle: {
        color: '#333',
        fontWeight: 'bold'
      }
    },
    tooltip: {
      backgroundColor: 'rgba(50,50,50,0.7)',
      textStyle: { color: '#fff' }
    }
  });

  // Initialize with custom theme
  const chart = echarts.init(document.getElementById('main'), 'myTheme');

  // Chart option
  const option = {
    title: {
      text: 'Custom Themed Sales Chart',
      left: 'center'
    },
    tooltip: {
      trigger: 'axis'
    },
    legend: {
      data: ['Q1', 'Q2'],
      top: 'bottom'
    },
    xAxis: {
      type: 'category',
      data: ['Product A', 'Product B', 'Product C']
    },
    yAxis: {
      type: 'value'
    },
    series: [
      {
        name: 'Q1',
        type: 'bar',
        data: [120, 200, 150]
      },
      {
        name: 'Q2',
        type: 'bar',
        data: [180, 160, 190]
      }
```

```
    ]
  };

  chart.setOption(option);
  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

3.4.3 Styling Key Visual Elements

You can customize many aspects of your chart's look and feel via the `option` object:

Color Palette

Specify a custom array of colors for your series:

```
color: ['#5470C6', '#91CC75', '#FAC858', '#EE6666'],
```

ECharts will cycle through these colors for series by default.

Labels

Labels show data values on or near chart elements. Customize them with the `label` option inside series or components:

```
series: [{
  type: 'bar',
  data: [120, 200, 150],
  label: {
    show: true,
    position: 'top',
    color: '#333',
    fontWeight: 'bold'
  }
}]
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Bar Chart with Labels</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
```

```
</head>
<body>

<div id="main"></div>

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

  const option = {
    title: {
      text: 'Product Sales',
      left: 'center'
    },
    xAxis: {
      type: 'category',
      data: ['Product A', 'Product B', 'Product C']
    },
    yAxis: {
      type: 'value'
    },
    series: [{
      type: 'bar',
      data: [120, 200, 150],
      label: {
        show: true,
        position: 'top',      // Show label above each bar
        color: '#333',
        fontWeight: 'bold',
        fontSize: 14
      },
      itemStyle: {
        color: '#5470C6'
      }
    }]
  };

  chart.setOption(option);
  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

Item Style

Control the appearance of individual graphical items (bars, points, slices) via `itemStyle`:

```
itemStyle: {
  color: '#5470C6',
  borderColor: '#333',
  borderWidth: 1,
  borderType: 'solid'
}
```

Area Style

For area charts or radar charts, the `areaStyle` adds fill colors beneath lines or polygons:

```
series: [{
  type: 'line',
  data: [100, 120, 130, 90],
  areaStyle: {
    color: 'rgba(84, 112, 198, 0.4)'
  }
}]
```

3.4.4 Switching Themes Dynamically

You can switch themes at runtime by disposing the current chart and reinitializing it with a new theme:

```
function switchTheme(themeName) {
  const container = document.getElementById('main');

  // Dispose current chart instance
  if (echarts.getInstanceByDom(container)) {
    echarts.dispose(container);
  }

  // Initialize chart with new theme
  const chart = echarts.init(container, themeName);
  chart.setOption(option);
}

// Example usage:
// switchTheme('dark');
// switchTheme('myTheme');
```

This approach allows you to provide users with theme toggles (e.g., light/dark mode) without reloading the page.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ECharts Theme Switcher</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5"></script>
  <style>
    #main { width: 600px; height: 400px; margin: 20px auto; }
    #themeBtn { display: block; margin: 0 auto 20px; padding: 8px 16px; }
  </style>
</head>
<body>

<button id="themeBtn">Switch Theme</button>
<div id="main"></div>

<script>
  // Define a custom theme
```

```
  echarts.registerTheme('myDark', {
    backgroundColor: '#2e2e2e',
    textStyle: { color: '#eee' },
    title: { textStyle: { color: '#fff' } },
    tooltip: { backgroundColor: '#444' },
    legend: { textStyle: { color: '#eee' } },
    color: ['#91cc75', '#fac858', '#ee6666']
  });

  const option = {
    title: { text: 'Quarterly Sales' },
    tooltip: { trigger: 'axis' },
    legend: { data: ['2023', '2024'] },
    xAxis: {
      type: 'category',
      data: ['Q1', 'Q2', 'Q3', 'Q4']
    },
    yAxis: { type: 'value' },
    series: [
      { name: '2023', type: 'bar', data: [120, 200, 150, 80] },
      { name: '2024', type: 'bar', data: [160, 250, 180, 100] }
    ]
  };

  let currentTheme = 'light';

  function switchTheme(themeName) {
    const container = document.getElementById('main');
    if (echarts.getInstanceByDom(container)) {
      echarts.dispose(container);
    }
    const chart = echarts.init(container, themeName);
    chart.setOption(option);
    return chart;
  }

  let chart = switchTheme(null); // Start with default theme

  document.getElementById('themeBtn').addEventListener('click', () => {
    currentTheme = currentTheme === 'light' ? 'myDark' : 'light';
    chart = switchTheme(currentTheme === 'light' ? null : 'myDark');
  });

  window.addEventListener('resize', () => chart.resize());
</script>

</body>
</html>
```

3.4.5 Summary

Styling Aspect	How to Customize	Example Property
Theme	Use built-in or register custom	`echarts.init(dom, 'dark')` or `echarts.registerTheme()`
Colors	Define color arrays for series	`color: ['#5470C6', '#91CC75']`
Labels	Show and style data labels	`label: { show: true, color: '#333' }`
Item Style	Customize individual elements' appearance	`itemStyle: { color: '#5470C6' }`
Area Style	Fill areas under lines or polygons	`areaStyle: { color: 'rgba(...)' }`

With styling and themes, you control the entire look and feel of your charts, tailoring them to fit any visual identity or user preference—while preserving the full power of ECharts' interactive and high-performance capabilities.

Chapter 4.

Core Chart Types

1. Bar Charts (Vertical and Horizontal)
2. Line Charts and Area Charts
3. Pie and Donut Charts
4. Scatter Plots and Bubble Charts
5. Radar Charts
6. Gauge and Funnel Charts

4 Core Chart Types

4.1 Bar Charts (Vertical and Horizontal)

Bar charts are one of the most common and versatile chart types, ideal for comparing quantities across different categories. ECharts supports both **vertical** and **horizontal** bar charts, as well as **grouped** and **stacked** variations to visualize multiple datasets effectively.

4.1.1 Vertical vs. Horizontal Bar Charts

- **Vertical bar charts** display bars extending upward along the y-axis.
- **Horizontal bar charts** display bars extending horizontally along the x-axis.

You can switch between these layouts simply by swapping the `xAxis` and `yAxis` configurations.

4.1.2 Switching Axis Orientation

Vertical Bar Chart (default)

```
option = {
  xAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  yAxis: {
    type: 'value'
  },
  series: [{
    type: 'bar',
    data: [320, 450, 300, 500]
  }]
};
```

- `xAxis` is categorical (regions).
- `yAxis` is numeric (sales values).
- Bars grow vertically.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Vertical Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
```

```
    html, body {
      margin: 0;
      padding: 0;
      height: 100%;
    }
    #main {
      width: 100%;
      height: 100%;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <script>
    const chart = echarts.init(document.getElementById('main'));

    const option = {
      title: {
        text: 'Regional Sales (Vertical Bar)',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        name: 'Sales',
        type: 'bar',
        data: [320, 450, 300, 500],
        itemStyle: {
          color: '#5470C6'
        },
        label: {
          show: true,
          position: 'top'
        }
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

Horizontal Bar Chart

```
option = {
  xAxis: {
    type: 'value'
  },
```

```
  yAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  series: [{
    type: 'bar',
    data: [320, 450, 300, 500]
  }]
};
```

- `xAxis` becomes numeric (sales values).
- `yAxis` becomes categorical (regions).
- Bars grow horizontally.

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/echarts@5/dist/echarts.min.js"></script>
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100%;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <script>
    const chart = echarts.init(document.getElementById('main'));

    const option = {
      title: {
        text: 'Regional Sales (Horizontal Bar)',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'value'
      },
      yAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      series: [{
        name: 'Sales',
        type: 'bar',
        data: [320, 450, 300, 500],
        itemStyle: {
          color: '#91CC75'
        },
        label: {
          show: true,
```

```
          position: 'right'
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.1.3 Grouped Bar Charts

Grouped bar charts compare multiple series side-by-side within each category.

```
option = {
  xAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  yAxis: {
    type: 'value'
  },
  legend: {
    data: ['2023', '2024']
  },
  series: [
    {
      name: '2023',
      type: 'bar',
      data: [320, 450, 300, 500]
    },
    {
      name: '2024',
      type: 'bar',
      data: [400, 470, 320, 550]
    }
  ]
};
```

- Each category has two bars, one per series.
- Legend allows toggling visibility.

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/echarts@5/dist/echarts.min.js"></script>
  <style>
```

```
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
    }
  </style>
</head>
<body>
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Sales Comparison by Region',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        axisPointer: { type: 'shadow' }
      },
      legend: {
        data: ['2023', '2024'],
        top: '10px',
        left: 'center'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      yAxis: {
        type: 'value',
        name: 'Sales'
      },
      series: [
        {
          name: '2023',
          type: 'bar',
          data: [320, 450, 300, 500],
          itemStyle: {
            color: '#5470C6'
          }
        },
        {
          name: '2024',
          type: 'bar',
          data: [400, 470, 320, 550],
          itemStyle: {
            color: '#91CC75'
          }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
```

```
</html>
```

4.1.4 Stacked Bar Charts

Stacked bars aggregate values within each category, emphasizing the total.

```
option = {
  xAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  yAxis: {
    type: 'value'
  },
  legend: {
    data: ['Online', 'In-Store', 'Wholesale']
  },
  series: [
    {
      name: 'Online',
      type: 'bar',
      stack: 'sales',         // Group bars to stack
      data: [120, 150, 180, 200]
    },
    {
      name: 'In-Store',
      type: 'bar',
      stack: 'sales',
      data: [80, 90, 70, 100]
    },
    {
      name: 'Wholesale',
      type: 'bar',
      stack: 'sales',
      data: [100, 120, 90, 150]
    }
  ]
};
```

- The `stack` property groups bars to stack vertically.
- Total height of the stacked bar represents combined 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/echarts@5/dist/echarts.min.js"></script>
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
```

```
    }
  </style>
</head>
<body>
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Sales by Channel (Stacked)',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        axisPointer: { type: 'shadow' }
      },
      legend: {
        data: ['Online', 'In-Store', 'Wholesale'],
        top: '10px',
        left: 'center'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      yAxis: {
        type: 'value',
        name: 'Sales'
      },
      series: [
        {
          name: 'Online',
          type: 'bar',
          stack: 'sales',
          data: [120, 150, 180, 200],
          itemStyle: { color: '#5470C6' }
        },
        {
          name: 'In-Store',
          type: 'bar',
          stack: 'sales',
          data: [80, 90, 70, 100],
          itemStyle: { color: '#91CC75' }
        },
        {
          name: 'Wholesale',
          type: 'bar',
          stack: 'sales',
          data: [100, 120, 90, 150],
          itemStyle: { color: '#FAC858' }
        }
      ]
    };

    chart.setOption(option);
```

```
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.1.5 Real-World Example: Sales by Region

Let's visualize sales performance by region for two years, using a grouped vertical bar chart:

```
const option = {
  title: {
    text: 'Sales by Region (2023 vs 2024)',
    left: 'center'
  },
  tooltip: {
    trigger: 'axis'
  },
  legend: {
    data: ['2023', '2024'],
    top: 'bottom'
  },
  xAxis: {
    type: 'category',
    data: ['North', 'South', 'East', 'West']
  },
  yAxis: {
    type: 'value',
    name: 'Sales (in $1000)'
  },
  series: [
    {
      name: '2023',
      type: 'bar',
      data: [320, 450, 300, 500],
      itemStyle: { color: '#5470C6' }
    },
    {
      name: '2024',
      type: 'bar',
      data: [400, 470, 320, 550],
      itemStyle: { color: '#91CC75' }
    }
  ]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Sales by Region Grouped Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
```

```
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
      display: flex; justify-content: center; align-items: center;
    }
    #main {
      width: 700px; height: 400px;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <script>
    const chart = echarts.init(document.getElementById('main'));

    const option = {
      title: {
        text: 'Sales by Region (2023 vs 2024)',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        axisPointer: { type: 'shadow' }
      },
      legend: {
        data: ['2023', '2024'],
        top: 'bottom',
        left: 'center'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      yAxis: {
        type: 'value',
        name: 'Sales (in $1000)'
      },
      series: [
        {
          name: '2023',
          type: 'bar',
          data: [320, 450, 300, 500],
          itemStyle: { color: '#5470C6' }
        },
        {
          name: '2024',
          type: 'bar',
          data: [400, 470, 320, 550],
          itemStyle: { color: '#91CC75' }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
```

```
</html>
```

4.1.6 Summary

Chart Type	How to Configure	Use Case
Vertical Bar	Categorical `xAxis`, numeric `yAxis`	Standard categorical comparisons
Horizontal Bar	Numeric `xAxis`, categorical `yAxis`	Long category names, ranking
Grouped Bars	Multiple series, no `stack` property	Comparing multiple datasets side-by-side
Stacked Bars	Multiple series with the same `stack` value	Showing part-to-whole relationships

Bar charts in ECharts are flexible and easy to customize, making them ideal for displaying discrete category data in both simple and complex scenarios.

4.2 Line Charts and Area Charts

Line charts are perfect for visualizing trends and changes over time or continuous data. Area charts are a variation where the space beneath the line is filled, emphasizing volume. ECharts makes it easy to create both simple and multi-series line and area charts with customization options like smooth curves, symbols, and interactive tooltips.

4.2.1 Creating a Simple Line Chart

Here's a basic example of a single-series line chart showing monthly sales:

```
const option = {
  title: {
    text: 'Monthly Sales'
  },
  tooltip: {
    trigger: 'axis'
  },
  xAxis: {
    type: 'category',
    data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
  },
  yAxis: {
```

```
    type: 'value'
  },
  series: [{
    name: 'Sales',
    type: 'line',
    data: [120, 200, 150, 80, 70, 110]
  }]
};
```

- `type: 'line'` defines the chart as a line chart.
- `tooltip` is set to trigger on axis hover, showing values for the hovered point.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
      display: flex; justify-content: center; align-items: center;
    }
    #main {
      width: 700px; height: 400px;
    }
  </style>
</head>
<body>
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Monthly Sales'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        name: 'Sales',
        type: 'line',
        data: [120, 200, 150, 80, 70, 110]
      }]
    };
```

```
    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.2.2 Multi-Series Line Chart with Smooth Curves and Symbols

You can add multiple lines by including more series, enable smooth curves, and customize symbols to enhance readability.

```
const option = {
  title: {
    text: 'Monthly Sales Comparison'
  },
  tooltip: {
    trigger: 'axis'
  },
  legend: {
    data: ['2023', '2024']
  },
  xAxis: {
    type: 'category',
    data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
  },
  yAxis: {
    type: 'value'
  },
  series: [
    {
      name: '2023',
      type: 'line',
      data: [120, 200, 150, 80, 70, 110],
      smooth: true,              // Smooth curves
      symbol: 'circle',          // Symbol shape at data points
      symbolSize: 8,
      lineStyle: {
        width: 3
      }
    },
    {
      name: '2024',
      type: 'line',
      data: [130, 180, 160, 90, 100, 120],
      smooth: true,
      symbol: 'rect',
      symbolSize: 8,
      lineStyle: {
        width: 3
      }
    }
  ]
};
```

- `smooth: true` creates curved lines instead of sharp angles.
- `symbol` and `symbolSize` control the marker shape and size at each data point.
- The legend allows toggling series visibility.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Multi-Series Smooth Line Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
      display: flex; justify-content: center; align-items: center;
    }
    #main {
      width: 700px; height: 400px;
    }
  </style>
</head>
<body>
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Monthly Sales Comparison'
      },
      tooltip: {
        trigger: 'axis'
      },
      legend: {
        data: ['2023', '2024']
      },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
      },
      yAxis: {
        type: 'value'
      },
      series: [
        {
          name: '2023',
          type: 'line',
          data: [120, 200, 150, 80, 70, 110],
          smooth: true,
          symbol: 'circle',
          symbolSize: 8,
          lineStyle: { width: 3 }
        },
        {
          name: '2024',
```

```
          type: 'line',
          data: [130, 180, 160, 90, 100, 120],
          smooth: true,
          symbol: 'rect',
          symbolSize: 8,
          lineStyle: { width: 3 }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.2.3 Converting a Line Chart into an Area Chart

Add the `areaStyle` property to a line series to fill the area beneath the line, turning it into an area chart.

```
series: [{
  name: 'Sales',
  type: 'line',
  data: [120, 200, 150, 80, 70, 110],
  smooth: true,
  areaStyle: {
    color: 'rgba(84, 112, 198, 0.4)'
  }
}]
```

The filled area adds visual weight to emphasize volume or magnitude changes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Area Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    html, body, #main {
      margin: 0; padding: 0; width: 100%; height: 100vh;
      display: flex; justify-content: center; align-items: center;
    }
    #main {
      width: 700px; height: 400px;
    }
  </style>
</head>
<body>
```

```
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Monthly Sales as Area Chart'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        name: 'Sales',
        type: 'line',
        data: [120, 200, 150, 80, 70, 110],
        smooth: true,
        areaStyle: {
          color: 'rgba(84, 112, 198, 0.4)'
        },
        lineStyle: {
          width: 3,
          color: '#5470C6'
        },
        symbol: 'circle',
        symbolSize: 8
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.2.4 Full Area Chart Example

```
const option = {
  title: {
    text: 'Monthly Sales with Area Fill'
  },
  tooltip: {
    trigger: 'axis'
  },
  xAxis: {
```

```
    type: 'category',
    data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
  },
  yAxis: {
    type: 'value'
  },
  series: [
    {
      name: '2023',
      type: 'line',
      data: [120, 200, 150, 80, 70, 110],
      smooth: true,
      areaStyle: {
        color: 'rgba(84, 112, 198, 0.4)'
      },
      symbol: 'circle',
      symbolSize: 6
    },
    {
      name: '2024',
      type: 'line',
      data: [130, 180, 160, 90, 100, 120],
      smooth: true,
      areaStyle: {
        color: 'rgba(145, 204, 117, 0.4)'
      },
      symbol: 'rect',
      symbolSize: 6
    }
  ]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Full Area Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 700px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="main"></div>

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

    const option = {
      title: {
        text: 'Monthly Sales with Area Fill',
```

```
        left: 'center'
      },
      tooltip: {
        trigger: 'axis'
      },
      legend: {
        data: ['2023', '2024'],
        top: 'bottom'
      },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
      },
      yAxis: {
        type: 'value'
      },
      series: [
        {
          name: '2023',
          type: 'line',
          data: [120, 200, 150, 80, 70, 110],
          smooth: true,
          areaStyle: {
            color: 'rgba(84, 112, 198, 0.4)'
          },
          symbol: 'circle',
          symbolSize: 6,
          lineStyle: {
            color: '#5470C6',
            width: 2
          }
        },
        {
          name: '2024',
          type: 'line',
          data: [130, 180, 160, 90, 100, 120],
          smooth: true,
          areaStyle: {
            color: 'rgba(145, 204, 117, 0.4)'
          },
          symbol: 'rect',
          symbolSize: 6,
          lineStyle: {
            color: '#91CC75',
            width: 2
          }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.2.5 Summary

Feature	Configuration Option	Effect
Basic line chart	`series.type = 'line'`	Displays data as connected points
Multiple series	Add multiple objects in `series` array	Compare multiple datasets
Smooth curves	`series.smooth = true`	Makes lines smooth and curved
Symbols on points	`series.symbol` and `symbolSize`	Custom markers at data points
Area fill	`series.areaStyle`	Fills area under the line (area chart)
Tooltip on axis	`tooltip.trigger = 'axis'`	Shows data for all series on hover

By mastering line and area charts in ECharts, you can effectively communicate trends, comparisons, and volumes across time or continuous data.

4.3 Pie and Donut Charts

Pie charts are a popular way to visualize proportions and percentages in a dataset, making them ideal for showing market share, budget allocation, or survey results. Donut charts are a variation with a hollow center that improves readability and allows for additional information in the middle.

ECharts makes it easy to create both pie and donut charts, customize labels and interactions, and highlight key slices.

4.3.1 Creating a Basic Pie Chart

Here's an example of a simple pie chart showing market share by company:

```
const option = {
  title: {
    text: 'Market Share by Company',
    left: 'center'
  },
  tooltip: {
    trigger: 'item',
    formatter: '{b}: {c} ({d}%)' // Show name, value, and percent
  },
  legend: {
    orient: 'vertical',
    left: 'left',
    data: ['Company A', 'Company B', 'Company C', 'Company D']
```

```
  },
  series: [
    {
      name: 'Market Share',
      type: 'pie',
      radius: '50%',              // Radius defines pie size
      data: [
        { value: 335, name: 'Company A' },
        { value: 310, name: 'Company B' },
        { value: 234, name: 'Company C' },
        { value: 135, name: 'Company D' }
      ],
      emphasis: {
        itemStyle: {
          shadowBlur: 10,
          shadowOffsetX: 0,
          shadowColor: 'rgba(0, 0, 0, 0.5)'
        }
      },
      label: {
        show: true,
        formatter: '{b}\n{d}%',
        fontWeight: 'bold'
      },
      labelLine: {
        smooth: true,
        length: 15,
        length2: 10
      }
    }
  ]
};
```

- `radius: '50%'` sets the size of the pie.
- Labels show the name and percent, formatted neatly.
- `emphasis` highlights a slice with a shadow on hover.
- `labelLine` controls the lines connecting labels to slices.

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/echarts@5/dist/echarts.min.js"></script>
  <style>
    #pieChart {
      width: 600px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="pieChart"></div>
```

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

    const option = {
      title: {
        text: 'Market Share by Company',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{b}: {c} ({d}%)'
      },
      legend: {
        orient: 'vertical',
        left: 'left',
        data: ['Company A', 'Company B', 'Company C', 'Company D']
      },
      series: [
        {
          name: 'Market Share',
          type: 'pie',
          radius: '50%',
          data: [
            { value: 335, name: 'Company A' },
            { value: 310, name: 'Company B' },
            { value: 234, name: 'Company C' },
            { value: 135, name: 'Company D' }
          ],
          emphasis: {
            itemStyle: {
              shadowBlur: 10,
              shadowOffsetX: 0,
              shadowColor: 'rgba(0, 0, 0, 0.5)'
            }
          },
          label: {
            show: true,
            formatter: '{b}\n{d}%',
            fontWeight: 'bold'
          },
          labelLine: {
            smooth: true,
            length: 15,
            length2: 10
          }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.3.2 Turning a Pie Chart into a Donut Chart

To create a donut chart, set the `radius` property to an array with inner and outer radius values:

```
series: [{
  type: 'pie',
  radius: ['40%', '70%'],   // Inner radius creates hollow center
  data: [
    { value: 335, name: 'Company A' },
    { value: 310, name: 'Company B' },
    { value: 234, name: 'Company C' },
    { value: 135, name: 'Company D' }
  ],
  label: {
    show: true,
    position: 'outside',
    formatter: '{b}: {d}%',
    fontWeight: 'bold'
  },
  labelLine: {
    length: 20,
    length2: 10,
    smooth: true
  },
  emphasis: {
    itemStyle: {
      shadowBlur: 15,
      shadowColor: 'rgba(0, 0, 0, 0.5)'
    }
  }
}]
```

- The first value (`'40%'`) is the inner radius, creating the hollow center.
- The second value (`'70%'`) is the outer radius, controlling the donut's thickness.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Donut Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #donutChart {
      width: 600px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="donutChart"></div>

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

```
    const option = {
      title: {
        text: 'Market Share by Company (Donut Chart)',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{b}: {c} ({d}%)'
      },
      legend: {
        orient: 'vertical',
        left: 'left',
        data: ['Company A', 'Company B', 'Company C', 'Company D']
      },
      series: [
        {
          name: 'Market Share',
          type: 'pie',
          radius: ['40%', '70%'],  // Inner and outer radius for donut shape
          data: [
            { value: 335, name: 'Company A' },
            { value: 310, name: 'Company B' },
            { value: 234, name: 'Company C' },
            { value: 135, name: 'Company D' }
          ],
          label: {
            show: true,
            position: 'outside',
            formatter: '{b}: {d}%',
            fontWeight: 'bold'
          },
          labelLine: {
            length: 20,
            length2: 10,
            smooth: true
          },
          emphasis: {
            itemStyle: {
              shadowBlur: 15,
              shadowColor: 'rgba(0, 0, 0, 0.5)'
            }
          }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.3.3 Exploding Slices (Highlighting)

To highlight or "explode" a specific slice, use the `selected` and `selectedOffset` properties:

```
data: [
  { value: 335, name: 'Company A', selected: true, selectedOffset: 20 },
  { value: 310, name: 'Company B' },
  { value: 234, name: 'Company C' },
  { value: 135, name: 'Company D' }
]
```

- `selected: true` marks the slice as exploded initially.
- `selectedOffset` moves the slice outward by the specified pixels.

You can also enable slice explosion on hover by setting:

```
series: [{
  type: 'pie',
  // ...
  emphasis: {
    scale: true,           // Slightly enlarges the hovered slice
    scaleSize: 10
  }
}]
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Donut Chart with Exploding Slice</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #donutChart {
      width: 600px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="donutChart"></div>

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

    const option = {
      title: {
        text: 'Market Share with Exploded Slice',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{b}: {c} ({d}%)'
      },
      legend: {
        orient: 'vertical',
```

```
        left: 'left',
        data: ['Company A', 'Company B', 'Company C', 'Company D']
      },
      series: [
        {
          name: 'Market Share',
          type: 'pie',
          radius: ['40%', '70%'],
          data: [
            { value: 335, name: 'Company A', selected: true, selectedOffset: 20 },
            { value: 310, name: 'Company B' },
            { value: 234, name: 'Company C' },
            { value: 135, name: 'Company D' }
          ],
          label: {
            show: true,
            position: 'outside',
            formatter: '{b}: {d}%',
            fontWeight: 'bold'
          },
          labelLine: {
            length: 20,
            length2: 10,
            smooth: true
          },
          emphasis: {
            scale: true,
            scaleSize: 10,
            itemStyle: {
              shadowBlur: 20,
              shadowColor: 'rgba(0, 0, 0, 0.5)'
            }
          }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.3.4 Real-World Use Cases

- **Market Share Analysis:** Show percentage of sales or users by company or product.
- **Budget Breakdown:** Visualize how total budget is allocated across departments.
- **Survey Results:** Display proportions of responses for different categories.
- **Population Demographics:** Represent age or income group distribution.

The visual impact of pie and donut charts makes them excellent for communicating proportions at a glance.

4.3.5 Summary

Feature	Configuration Property	Purpose
Pie Chart Size	`radius: '50%'`	Sets pie size
Donut Chart Hollow	`radius: ['40%', '70%']`	Creates hollow center for donut
Labels	`label.formatter`	Customize label text and position
Label Lines	`labelLine.length`, `length2`, `smooth`	Connect labels to slices
Hover Highlight	`emphasis.itemStyle`, `emphasis.scale`	Highlight slices on hover
Exploded Slice	`selected`, `selectedOffset`	Move slice out for emphasis

With these techniques, you can craft clear, interactive pie and donut charts that highlight key parts of your data and engage your audience effectively.

4.4 Scatter Plots and Bubble Charts

Scatter plots are powerful for visualizing the relationship or distribution between two continuous variables. Bubble charts extend scatter plots by adding a third dimension—usually represented by the size of each point—allowing you to visualize an additional metric simultaneously.

4.4.1 Scatter Plots: Basic Two-Dimensional Data

A scatter plot displays points on an X-Y plane, where each point corresponds to two numerical values.

4.4.2 Example: Basic Scatter Plot

```
const option = {
  title: {
    text: 'Height vs Weight'
  },
  tooltip: {
    trigger: 'item',
    formatter: params => `Height: ${params.data[0]} cm<br/>Weight: ${params.data[1]} kg`
  },
```

```
  xAxis: {
    name: 'Height (cm)',
    type: 'value'
  },
  yAxis: {
    name: 'Weight (kg)',
    type: 'value'
  },
  series: [{
    type: 'scatter',
    data: [
      [170, 65],
      [160, 55],
      [180, 75],
      [175, 70],
      [165, 60]
    ]
  }]
};
```

- Each data point is an array `[x, y]` — height and weight.
- The tooltip displays the X and Y values when hovering over points.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Scatter Plot: Height vs Weight</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #scatterPlot {
      width: 600px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="scatterPlot"></div>

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

    const option = {
      title: {
        text: 'Height vs Weight'
      },
      tooltip: {
        trigger: 'item',
        formatter: params => `Height: ${params.data[0]} cm<br/>Weight: ${params.data[1]} kg`
      },
      xAxis: {
        name: 'Height (cm)',
        type: 'value'
      },
```

```
      yAxis: {
        name: 'Weight (kg)',
        type: 'value'
      },
      series: [{
        type: 'scatter',
        data: [
          [170, 65],
          [160, 55],
          [180, 75],
          [175, 70],
          [165, 60]
        ],
        symbolSize: 12,
        itemStyle: {
          color: '#5470C6'
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.4.3 Bubble Charts: Adding a Third Dimension with Size

Bubble charts build on scatter plots by adding a third dimension to represent the size of each point, useful for highlighting magnitude or importance.

4.4.4 Example: Bubble Chart with Size Data

```
const option = {
  title: {
    text: 'Height vs Weight with Age (Bubble Size)'
  },
  tooltip: {
    trigger: 'item',
    formatter: params => {
      const [height, weight, age] = params.data;
      return `Height: ${height} cm<br/>Weight: ${weight} kg<br/>Age: ${age} years`;
    }
  },
  xAxis: {
    name: 'Height (cm)',
    type: 'value'
  },
```

```
  yAxis: {
    name: 'Weight (kg)',
    type: 'value'
  },
  series: [{
    type: 'scatter',
    symbolSize: function (data) {
      return data[2]; // Use the third value as bubble size
    },
    data: [
      [170, 65, 20],  // height, weight, age (size)
      [160, 55, 25],
      [180, 75, 30],
      [175, 70, 22],
      [165, 60, 28]
    ]
  }]
};
```

- Each data point is `[x, y, size]`.
- The `symbolSize` function uses the third value (`age`) to determine the bubble size.
- Tooltip shows all three dimensions on hover.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Bubble Chart: Height vs Weight with Age</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #bubbleChart {
      width: 600px;
      height: 400px;
      margin: 0 auto;
    }
  </style>
</head>
<body>
  <div id="bubbleChart"></div>

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

    const option = {
      title: {
        text: 'Height vs Weight with Age (Bubble Size)'
      },
      tooltip: {
        trigger: 'item',
        formatter: params => {
          const [height, weight, age] = params.data;
          return `Height: ${height} cm<br/>Weight: ${weight} kg<br/>Age: ${age} years`;
        }
      },
      xAxis: {
```

```
        name: 'Height (cm)',
        type: 'value'
      },
      yAxis: {
        name: 'Weight (kg)',
        type: 'value'
      },
      series: [{
        type: 'scatter',
        symbolSize: function (data) {
          return data[2]; // bubble size from age
        },
        data: [
          [170, 65, 20],  // height, weight, age (bubble size)
          [160, 55, 25],
          [180, 75, 30],
          [175, 70, 22],
          [165, 60, 28]
        ],
        itemStyle: {
          color: '#91CC75'
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.4.5 Summary

Chart Type	Data Format	Visual Encoding	Use Case Example
Scatter Plot	`[x, y]`	Position of points	Height vs. Weight correlation
Bubble Chart	`[x, y, size]`	Position + bubble size	Adding Age or Population size

Scatter and bubble charts help reveal patterns, clusters, and outliers in multi-dimensional data with a straightforward, intuitive visual format.

4.5 Radar Charts

Radar charts (also known as spider charts or web charts) are ideal for visualizing multivariate data where each variable is represented along a separate axis starting from the same point. They are particularly useful to compare multiple items or groups across several dimensions or attributes simultaneously.

4.5.1 Understanding Radar Charts

- Each axis (or "indicator") corresponds to one dimension or attribute.
- Values are plotted along each axis and connected to form a polygon.
- Multiple series can be drawn on the same chart to compare different entities.

4.5.2 Defining Indicators and Axis Ranges

Indicators define the attributes being measured, along with their maximum values to scale the axes:

```
radar: {
  indicator: [
    { name: 'Quality', max: 100 },
    { name: 'Performance', max: 100 },
    { name: 'Durability', max: 100 },
    { name: 'Price', max: 100 },
    { name: 'Appearance', max: 100 }
  ]
}
```

- `name`: The attribute label.
- `max`: The maximum possible value for that attribute, used to normalize the scale.

4.5.3 Example: Comparing Multiple Product Attributes

This example compares two products across five key attributes:

```
const option = {
  title: {
    text: 'Product Attribute Comparison'
  },
  tooltip: {},
  legend: {
    data: ['Product A', 'Product B'],
    bottom: 0
  },
```

```
  radar: {
    indicator: [
      { name: 'Quality', max: 100 },
      { name: 'Performance', max: 100 },
      { name: 'Durability', max: 100 },
      { name: 'Price', max: 100 },
      { name: 'Appearance', max: 100 }
    ],
    shape: 'circle',
    splitNumber: 5, // Number of concentric circles
    name: {
      textStyle: {
        color: '#555',
        fontSize: 14
      }
    }
  },
  series: [{
    name: 'Product Comparison',
    type: 'radar',
    data: [
      {
        value: [85, 90, 70, 60, 75],
        name: 'Product A',
        areaStyle: { opacity: 0.2 }
      },
      {
        value: [80, 85, 75, 70, 80],
        name: 'Product B',
        areaStyle: { opacity: 0.2 }
      }
    ]
  }]
};
```

- The `radar.indicator` array defines the attributes and scales.
- The `series.data` array contains values for each product.
- `areaStyle` adds a translucent fill for better visual comparison.
- The legend identifies each product polygon.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Radar Chart: Product Attribute Comparison</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #radarChart {
      width: 600px;
      height: 450px;
      margin: 20px auto;
    }
  </style>
</head>
```

```
<body>
  <div id="radarChart"></div>

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

    const option = {
      title: {
        text: 'Product Attribute Comparison'
      },
      tooltip: {},
      legend: {
        data: ['Product A', 'Product B'],
        bottom: 0
      },
      radar: {
        indicator: [
          { name: 'Quality', max: 100 },
          { name: 'Performance', max: 100 },
          { name: 'Durability', max: 100 },
          { name: 'Price', max: 100 },
          { name: 'Appearance', max: 100 }
        ],
        shape: 'circle',
        splitNumber: 5,
        name: {
          textStyle: {
            color: '#555',
            fontSize: 14
          }
        }
      },
      series: [{
        name: 'Product Comparison',
        type: 'radar',
        data: [
          {
            value: [85, 90, 70, 60, 75],
            name: 'Product A',
            areaStyle: { opacity: 0.2 }
          },
          {
            value: [80, 85, 75, 70, 80],
            name: 'Product B',
            areaStyle: { opacity: 0.2 }
          }
        ]
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.5.4 When to Use Radar Charts

- Comparing multiple entities on several attributes.
- Displaying skill sets, performance metrics, or survey responses.
- Visualizing strengths and weaknesses across categories.

4.5.5 Summary

Concept	Configuration Key	Description
Indicators	`radar.indicator`	Attributes with axis names and max values
Multiple Series	`series.data`	Values for each entity plotted on the chart
Axis Style	`radar.name.textStyle`	Styling for attribute labels
Area Fill	`series.areaStyle`	Adds translucent polygon fill

Radar charts provide a clear and compact way to compare complex, multidimensional data at a glance, making them invaluable for performance reviews, product analysis, and more.

4.6 Gauge and Funnel Charts

Gauge and funnel charts serve different purposes but are both useful for displaying progress, performance, or stages in a process. ECharts offers flexible options to create visually appealing and interactive gauge and funnel charts with animations and custom labels.

4.6.1 Gauge Charts: Visualizing KPIs Like a Speedometer

Gauge charts are semicircular or circular dials that represent a single value on a scale—perfect for KPIs such as speed, utilization, or completion percentage.

4.6.2 Basic Gauge Chart Example

```
const option = {
  title: {
    text: 'CPU Usage',
    left: 'center'
  },
```

```
  series: [{
    name: 'Usage',
    type: 'gauge',
    min: 0,
    max: 100,
    splitNumber: 10,          // Number of segments on the dial
    detail: {
      formatter: '{value} %',  // Format value display
      fontSize: 20
    },
    data: [{ value: 65, name: 'CPU' }],
    axisLine: {
      lineStyle: {
        width: 15,
        color: [
          [0.3, '#91c7ae'],
          [0.7, '#63869e'],
          [1, '#c23531']
        ]
      }
    },
    animationDuration: 1000    // Animate progress to current value
  }]
};
```

- The dial ranges from `min` to `max`.
- `splitNumber` divides the scale into equal parts.
- The `detail.formatter` controls the label formatting inside the gauge.
- `axisLine.lineStyle.color` sets gradient colors for ranges.
- The gauge needle animates smoothly on load via `animationDuration`.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Gauge Chart - CPU Usage</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #gaugeChart {
      width: 400px;
      height: 300px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="gaugeChart"></div>

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

    const option = {
      title: {
        text: 'CPU Usage',
```

```
        left: 'center'
      },
      series: [{
        name: 'Usage',
        type: 'gauge',
        min: 0,
        max: 100,
        splitNumber: 10,
        detail: {
          formatter: '{value} %',
          fontSize: 20
        },
        data: [{ value: 65, name: 'CPU' }],
        axisLine: {
          lineStyle: {
            width: 15,
            color: [
              [0.3, '#91c7ae'],
              [0.7, '#63869e'],
              [1, '#c23531']
            ]
          }
        },
        animationDuration: 1000
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.6.3 Funnel Charts: Visualizing Stages in a Process

Funnel charts visualize progressively decreasing values, such as sales pipelines, conversion rates, or customer drop-offs, by stacking segments in descending order.

4.6.4 Basic Funnel Chart Example

```
const option = {
  title: {
    text: 'Sales Funnel',
    left: 'center'
  },
  tooltip: {
    trigger: 'item',
    formatter: '{b}: {c} ({d}%)'
```

```
  },
  series: [{
    name: 'Sales',
    type: 'funnel',
    left: '10%',
    top: 60,
    bottom: 60,
    width: '80%',
    minSize: '0%',
    maxSize: '100%',
    sort: 'descending',        // Sort funnel from large to small
    gap: 2,                    // Space between funnel sections
    label: {
      show: true,
      position: 'inside',
      formatter: '{b}\n{c} items'
    },
    labelLine: {
      show: false
    },
    data: [
      { value: 1000, name: 'Leads' },
      { value: 800, name: 'Qualified Leads' },
      { value: 600, name: 'Proposals' },
      { value: 300, name: 'Negotiations' },
      { value: 150, name: 'Closed Deals' }
    ]
  }]
};
```

- The funnel is sorted from highest to lowest value.
- Labels display stage name and item count.
- `gap` controls spacing between sections for clarity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Funnel Chart - Sales Funnel</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #funnelChart {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="funnelChart"></div>

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

    const option = {
```

```
      title: {
        text: 'Sales Funnel',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{b}: {c} ({d}%)'
      },
      series: [{
        name: 'Sales',
        type: 'funnel',
        left: '10%',
        top: 60,
        bottom: 60,
        width: '80%',
        minSize: '0%',
        maxSize: '100%',
        sort: 'descending',
        gap: 2,
        label: {
          show: true,
          position: 'inside',
          formatter: '{b}\n{c} items'
        },
        labelLine: {
          show: false
        },
        data: [
          { value: 1000, name: 'Leads' },
          { value: 800, name: 'Qualified Leads' },
          { value: 600, name: 'Proposals' },
          { value: 300, name: 'Negotiations' },
          { value: 150, name: 'Closed Deals' }
        ]
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

4.6.5 Customization Tips

- **Progress Animation:** Use `animationDuration` and `animationEasing` to animate gauge needle or funnel sections.
- **Label Formatting:** Customize labels using `formatter` to show percentages, counts, or combined info.
- **Colors:** Use `itemStyle.color` or `axisLine.lineStyle.color` gradients to visually indicate performance zones or priorities.

- **Sizes and Layout:** Control funnel dimensions with `left`, `top`, `width`, and `height` for responsive design.

4.6.6 Summary

Chart Type	Key Configurations	Typical Use Case
Gauge	`series.type = 'gauge'` `min`, `max`, `axisLine` colors	Visualizing single KPI like CPU usage or speed
Funnel	`series.type = 'funnel'` `sort`, `gap`, `label.formatter`	Showing stages in a sales or conversion funnel

Both gauge and funnel charts provide intuitive visualizations for progress and processes, complementing your dashboard or reporting toolkit with clear and engaging graphics.

Chapter 5.

Advanced Visualizations

1. Heatmaps
2. Tree Maps
3. Sunburst Charts
4. Graphs and Network Diagrams
5. Geographic and Map Visualizations
6. Parallel Coordinates and Other Specialized Charts

5 Advanced Visualizations

5.1 Heatmaps

Heatmaps are a powerful visualization technique used to represent data intensity or magnitude across two dimensions using color gradients. They are especially useful for showing patterns such as frequency, correlation, or activity intensity over a grid-like structure.

Common use cases for heatmaps include:

- **Calendar heatmaps** showing activity over days or hours.
- **Correlation matrices** displaying relationships between variables.
- **Geospatial data** intensity over regions (when combined with map coordinates).
- **Performance metrics** across two categorical dimensions.

5.1.1 What Is a Heatmap?

A heatmap visualizes data points on a 2D grid, where each cell's color intensity corresponds to the value's magnitude. Higher values might be shown as darker or warmer colors (e.g., red), while lower values might be lighter or cooler colors (e.g., blue).

5.1.2 Creating a Basic Heatmap in ECharts

ECharts provides the `heatmap` series type combined with the `visualMap` component to map data values to colors dynamically.

5.1.3 Example: Simple Heatmap with Color Gradient

```
const option = {
  title: {
    text: 'Activity Heatmap'
  },
  tooltip: {
    position: 'top',
    formatter: params => `X: ${params.data[0]}<br/>Y: ${params.data[1]}<br/>Value: ${params.data[2]}`
  },
  grid: {
    height: '50%',
    top: '15%'
  },
  xAxis: {
    type: 'category',
```

```
    data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'],
    splitArea: {
      show: true
    }
  },
  yAxis: {
    type: 'category',
    data: ['Morning', 'Afternoon', 'Evening'],
    splitArea: {
      show: true
    }
  },
  visualMap: {
    min: 0,
    max: 100,
    calculable: true,
    orient: 'horizontal',
    left: 'center',
    bottom: '5%',
    inRange: {
      color: ['#dbe9f4', '#4682b4', '#023858'] // Light to dark blue gradient
    }
  },
  series: [{
    name: 'Activity',
    type: 'heatmap',
    data: [
      [0, 0, 15], [0, 1, 30], [0, 2, 55],
      [1, 0, 25], [1, 1, 40], [1, 2, 65],
      [2, 0, 35], [2, 1, 50], [2, 2, 75],
      [3, 0, 20], [3, 1, 45], [3, 2, 70],
      [4, 0, 30], [4, 1, 60], [4, 2, 90],
      [5, 0, 40], [5, 1, 70], [5, 2, 95],
      [6, 0, 50], [6, 1, 80], [6, 2, 100]
    ],
    label: {
      show: true
    }
  }]
};
```

- The `data` array holds `[xIndex, yIndex, value]` triplets.
- `xAxis` and `yAxis` define categories for the two dimensions (days and time of day).
- `visualMap` dynamically maps data values to colors, with a draggable legend for fine-tuning.
- Labels on each cell show the numeric value for clarity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple Heatmap with Color Gradient</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
```

```
    #heatmapChart {
      width: 700px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="heatmapChart"></div>

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

    const option = {
      title: {
        text: 'Activity Heatmap'
      },
      tooltip: {
        position: 'top',
        formatter: params => `X: ${params.data[0]}<br/>Y: ${params.data[1]}<br/>Value: ${params.data[2]}
      },
      grid: {
        height: '50%',
        top: '15%'
      },
      xAxis: {
        type: 'category',
        data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'],
        splitArea: {
          show: true
        }
      },
      yAxis: {
        type: 'category',
        data: ['Morning', 'Afternoon', 'Evening'],
        splitArea: {
          show: true
        }
      },
      visualMap: {
        min: 0,
        max: 100,
        calculable: true,
        orient: 'horizontal',
        left: 'center',
        bottom: '5%',
        inRange: {
          color: ['#dbe9f4', '#4682b4', '#023858'] // Light to dark blue gradient
        }
      },
      series: [{
        name: 'Activity',
        type: 'heatmap',
        data: [
          [0, 0, 15], [0, 1, 30], [0, 2, 55],
          [1, 0, 25], [1, 1, 40], [1, 2, 65],
          [2, 0, 35], [2, 1, 50], [2, 2, 75],
          [3, 0, 20], [3, 1, 45], [3, 2, 70],
```

```
          [4, 0, 30], [4, 1, 60], [4, 2, 90],
          [5, 0, 40], [5, 1, 70], [5, 2, 95],
          [6, 0, 50], [6, 1, 80], [6, 2, 100]
        ],
        label: {
          show: true
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.1.4 Practical Use Case: Correlation Matrix Heatmap

Heatmaps can display correlations between variables in a matrix format, where the diagonal represents self-correlation (value 1).

```
const option = {
  title: { text: 'Correlation Matrix' },
  tooltip: {
    position: 'top',
    formatter: params => `${params.data[3]} correlation: ${params.data[2]}`
  },
  xAxis: {
    type: 'category',
    data: ['Var A', 'Var B', 'Var C', 'Var D'],
    splitArea: { show: true }
  },
  yAxis: {
    type: 'category',
    data: ['Var A', 'Var B', 'Var C', 'Var D'],
    splitArea: { show: true }
  },
  visualMap: {
    min: -1,
    max: 1,
    calculable: true,
    orient: 'horizontal',
    left: 'center',
    bottom: '5%',
    inRange: {
      color: ['#d73027', '#ffffbf', '#1a9850']  // Red to yellow to green
    }
  },
  series: [{
    name: 'Correlation',
    type: 'heatmap',
    data: [
      [0, 0, 1, 'Var A'], [0, 1, 0.8, 'Var B'], [0, 2, -0.5, 'Var C'], [0, 3, 0, 'Var D'],
```

```
      [1, 0, 0.8, 'Var A'], [1, 1, 1, 'Var B'], [1, 2, -0.3, 'Var C'], [1, 3, 0.2, 'Var D'],
      [2, 0, -0.5, 'Var A'], [2, 1, -0.3, 'Var B'], [2, 2, 1, 'Var C'], [2, 3, 0.7, 'Var D'],
      [3, 0, 0, 'Var A'], [3, 1, 0.2, 'Var B'], [3, 2, 0.7, 'Var C'], [3, 3, 1, 'Var D']
    ],
    label: { show: true, formatter: '{c}' }
  }]
};
```

- Colors represent correlation strength and direction.
- Labels display correlation coefficients.
- Useful for quickly identifying strong positive or negative relationships.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Correlation Matrix Heatmap</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #correlationChart {
      width: 600px;
      height: 450px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="correlationChart"></div>

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

    const option = {
      title: {
        text: 'Correlation Matrix',
        left: 'center'
      },
      tooltip: {
        position: 'top',
        formatter: params => `${params.data[3]} correlation: ${params.data[2]}`
      },
      xAxis: {
        type: 'category',
        data: ['Var A', 'Var B', 'Var C', 'Var D'],
        splitArea: { show: true },
        axisLabel: { rotate: 45 }
      },
      yAxis: {
        type: 'category',
        data: ['Var A', 'Var B', 'Var C', 'Var D'],
        splitArea: { show: true }
      },
      visualMap: {
        min: -1,
        max: 1,
```

```
        calculable: true,
        orient: 'horizontal',
        left: 'center',
        bottom: '5%',
        inRange: {
          color: ['#d73027', '#ffffbf', '#1a9850'] // Red to yellow to green
        }
      },
      series: [{
        name: 'Correlation',
        type: 'heatmap',
        data: [
          [0, 0, 1, 'Var A'], [0, 1, 0.8, 'Var B'], [0, 2, -0.5, 'Var C'], [0, 3, 0, 'Var D'],
          [1, 0, 0.8, 'Var A'], [1, 1, 1, 'Var B'], [1, 2, -0.3, 'Var C'], [1, 3, 0.2, 'Var D'],
          [2, 0, -0.5, 'Var A'], [2, 1, -0.3, 'Var B'], [2, 2, 1, 'Var C'], [2, 3, 0.7, 'Var D'],
          [3, 0, 0, 'Var A'], [3, 1, 0.2, 'Var B'], [3, 2, 0.7, 'Var C'], [3, 3, 1, 'Var D']
        ],
        label: {
          show: true,
          formatter: '{c}'
        },
        emphasis: {
          itemStyle: {
            borderColor: '#333',
            borderWidth: 1
          }
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.1.5 Summary

Feature	Configuration	Purpose
Data Format	`[xIndex, yIndex, value]`	Map values onto grid cells
Visual Mapping	`visualMap`	Maps values to colors dynamically
Axes	`xAxis` and `yAxis` as categories	Define row and column categories
Labeling	`series.label`	Display data values on heatmap cells
Use Cases	Calendar data, correlation matrices	Highlight intensity or relationships

Heatmaps in ECharts provide an intuitive way to visualize complex data distributions and uncover hidden patterns across two dimensions using color as a natural and immediate visual

cue.

5.2 Tree Maps

Tree maps are an effective way to visualize hierarchical (tree-structured) data in a compact, space-filling layout. They show part-to-whole relationships by representing each node as a rectangle whose area is proportional to a numeric value. Nested rectangles correspond to child nodes, allowing you to see both the overall structure and the relative weight of each branch.

When to Use Tree Maps

- **Disk usage**: Visualize folder/file sizes in a directory tree.
- **Financial reports**: Show budget breakdowns by department and sub-department.
- **Market share**: Compare sales by region, then by product within each region.
- **Portfolio analysis**: Display allocation across asset classes and individual holdings.

5.2.1 Creating a Tree Map with Nested Data

Below is an example that visualizes sales data by region and product category. Each **region** contains **categories**, and each category contains individual **products**.

```
const option = {
  title: {
    text: 'Sales Breakdown by Region and Category',
    left: 'center'
  },
  tooltip: {
    formatter: params => {
      // params.value is the numeric value;
      // params.data.name is the node name
      return `${params.data.name}: $${params.value}k`;
    }
  },
  series: [{
    type: 'treemap',
    // Define the root of the tree
    data: [
      {
        name: 'North America',
        value: 800,
        children: [
          { name: 'Electronics', value: 400 },
          { name: 'Furniture',   value: 200 },
          { name: 'Books',       value: 200 }
        ]
      },
      {
```

```
        name: 'Europe',
        value: 700,
        children: [
          { name: 'Electronics', value: 300 },
          { name: 'Clothing',    value: 250 },
          { name: 'Books',       value: 150 }
        ]
      },
      {
        name: 'Asia',
        value: 900,
        children: [
          { name: 'Electronics', value: 500 },
          { name: 'Automotive',  value: 200 },
          { name: 'Clothing',    value: 200 }
        ]
      }
    ],
    leafDepth: 2,          // Render two levels of depth
    roam: false,           // Disable panning/zooming
    nodeClick: false,      // Disable drill-down on click
    visualMin: 100,        // Minimum size for visual mapping
    visualMax: 500,        // Maximum size for visual mapping
    // Color gradient: smaller values light, larger values dark
    visualDimension: 0,    // Use 'value' for color mapping
    visualMap: {
      show: false,
      min: 100,
      max: 500,
      inRange: {
        color: ['#e0f3f8', '#abd9e9', '#74add1', '#4575b4']
      }
    },
    label: {
      show: true,
      formatter: '{b}\n{c}k',  // Name and value with 'k' suffix
      fontSize: 12,
      color: '#333'
    },
    upperLabel: {
      show: true,
      height: 30,
      formatter: '{b}',
      fontWeight: 'bold'
    }
  }]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Tree Map Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
```

```
    #treemapChart {
      width: 700px;
      height: 500px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="treemapChart"></div>

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

    const option = {
      title: {
        text: 'Sales Breakdown by Region and Category',
        left: 'center'
      },
      tooltip: {
        formatter: params => `${params.data.name}: $${params.value}k`
      },
      series: [{
        type: 'treemap',
        data: [
          {
            name: 'North America',
            value: 800,
            children: [
              { name: 'Electronics', value: 400 },
              { name: 'Furniture', value: 200 },
              { name: 'Books', value: 200 }
            ]
          },
          {
            name: 'Europe',
            value: 700,
            children: [
              { name: 'Electronics', value: 300 },
              { name: 'Clothing', value: 250 },
              { name: 'Books', value: 150 }
            ]
          },
          {
            name: 'Asia',
            value: 900,
            children: [
              { name: 'Electronics', value: 500 },
              { name: 'Automotive', value: 200 },
              { name: 'Clothing', value: 200 }
            ]
          }
        ],
        leafDepth: 2,
        roam: false,
        nodeClick: false,
        visualMin: 100,
        visualMax: 500,
        visualDimension: 0,
```

```
        visualMap: {
          show: false,
          min: 100,
          max: 500,
          inRange: {
            color: ['#e0f3f8', '#abd9e9', '#74add1', '#4575b4']
          }
        },
        label: {
          show: true,
          formatter: '{b}\n{c}k',
          fontSize: 12,
          color: '#333'
        },
        upperLabel: {
          show: true,
          height: 30,
          formatter: '{b}',
          fontWeight: 'bold'
        }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.2.2 Breakdown of Key Options

Feature	Configuration	Description
Nested Data	`data: [{ name, value, children: […] }, …]`	Each object can contain `children` to represent sub-nodes
Node Sizing	`value`	Determines rectangle area; larger values → larger rectangles
Depth Control	`leafDepth: 2`	Controls how many levels of the hierarchy are rendered
Color Gradients	`visualMap.inRange.color`	Maps `value` to a gradient palette, enhancing pattern recognition
Labels for Leaf Nodes	`label.formatter`	Shows node name and value on leaf rectangles
Labels for Parent Nodes	`upperLabel.formatter`	Displays parent folder/category names in a distinct style
Interactivity	`roam`, `nodeClick`, `tooltip`	Configure panning/zooming, drill-down behavior, and hover tooltips

5.2.3 Customizing Appearance

- **Rounded corners**:
  ```
  itemStyle: { borderRadius: 4 }
  ```
- **Border styling**:
  ```
  itemStyle: { borderColor: '#fff', borderWidth: 2 }
  ```
- **Drill-down navigation**:
  ```
  series: [{ type: 'treemap', nodeClick: 'zoomToNode' }]
  ```

5.2.4 Best Practices

1. **Limit depth**: Too many levels can make labels unreadable—use `leafDepth` to control.
2. **Meaningful thresholds**: Tune `visualMin`/`visualMax` so color gradients emphasize important ranges.
3. **Clear labels**: Use `upperLabel` for category headers and `label` for leaf details.
4. **Tooltips**: Always include `tooltip.formatter` to give precise numeric context on hover.

Tree maps turn complex hierarchies into intuitive, area-based visuals, making them an ideal choice whenever you need to show how parts contribute to a whole across multiple levels.

5.3 Sunburst Charts

Sunburst charts are a radial, circular visualization of hierarchical data, serving as an intuitive alternative to tree maps. Instead of nested rectangles, sunburst charts use concentric rings to represent different levels of the hierarchy. Each ring is divided into segments proportional to the value or size of each node, making it easy to see part-to-whole relationships across multiple levels at a glance.

5.3.1 Understanding Sunburst Charts

- The **center** represents the root node.
- Each **ring** outward corresponds to a deeper level in the hierarchy.
- Segments within each ring represent child nodes, sized by their values.
- Interactive features often include zooming into child nodes by clicking segments and smooth transition animations.

5.3.2 Structure of Hierarchical Data for Sunburst

The data format is similar to tree maps, where each node can have children:

```
{
  name: 'Root',
  value: 100,
  children: [
    {
      name: 'Category A',
      value: 60,
      children: [
        { name: 'Item A1', value: 30 },
        { name: 'Item A2', value: 30 }
      ]
    },
    {
      name: 'Category B',
      value: 40,
      children: [
        { name: 'Item B1', value: 40 }
      ]
    }
  ]
}
```

5.3.3 Creating a Basic Sunburst Chart

Here is an example visualizing product sales across categories and subcategories:

```
const option = {
  title: {
    text: 'Sales Distribution by Category',
    left: 'center'
  },
  tooltip: {
    trigger: 'item',
    formatter: params => `${params.data.name}: $${params.data.value}k`
  },
  series: [{
    type: 'sunburst',
    radius: [0, '90%'],  // Inner radius to outer radius
    data: [
      {
        name: 'Electronics',
        value: 100,
        children: [
          { name: 'Phones', value: 60 },
          { name: 'Computers', value: 40 }
        ]
      },
      {
        name: 'Furniture',
        value: 80,
        children: [
```

```
          { name: 'Chairs', value: 30 },
          { name: 'Tables', value: 50 }
        ]
      }
    ],
    label: {
      rotate: 'radial',
      fontSize: 12
    },
    itemStyle: {
      borderColor: '#fff',
      borderWidth: 2
    },
    emphasis: {
      focus: 'ancestor'  // Highlight ancestors on hover
    },
    animationDuration: 800,
    animationEasing: 'cubicOut'
  }]
};
```

- `radius: [0, '90%']` defines the full sunburst from the center out.
- `label.rotate: 'radial'` aligns labels along the arcs for better readability.
- `itemStyle` with border color separates segments visually.
- `emphasis.focus: 'ancestor'` highlights the hovered segment and its parent path.
- Smooth animation on load and interaction enhances user experience.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Sunburst Chart Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #chart {
      width: 700px;
      height: 500px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="chart"></div>

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

    const option = {
      title: {
        text: 'Sales Distribution by Category',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
```

```
        formatter: params => `${params.data.name}: $${params.data.value}k`
      },
      series: [{
        type: 'sunburst',
        radius: [0, '90%'],
        data: [
          {
            name: 'Electronics',
            value: 100,
            children: [
              { name: 'Phones', value: 60 },
              { name: 'Computers', value: 40 }
            ]
          },
          {
            name: 'Furniture',
            value: 80,
            children: [
              { name: 'Chairs', value: 30 },
              { name: 'Tables', value: 50 }
            ]
          }
        ],
        label: {
          rotate: 'radial',
          fontSize: 12
        },
        itemStyle: {
          borderColor: '#fff',
          borderWidth: 2
        },
        emphasis: {
          focus: 'ancestor'
        },
        animationDuration: 800,
        animationEasing: 'cubicOut'
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.3.4 Interactive Features

- **Click to zoom:** By default, clicking a segment zooms into that node, focusing the view on its descendants.
- **Hover highlighting:** Ancestor paths are highlighted on hover to clarify hierarchy context.

- **Tooltips:** Show detailed info about any segment.

You can customize interaction behavior using the `nodeClick` property:

```
series: [{
  type: 'sunburst',
  nodeClick: 'zoomToNode' // Default zoom on click
  // or 'rootToNode' to show full path on click
}]
```

5.3.5 Summary

Feature	Configuration	Purpose
Hierarchical Data	Nested `children` arrays	Represent multi-level categories and items
Ring Layout	`radius: [inner, outer]`	Controls sunburst size and inner hole size
Labels	`label.rotate: 'radial'`	Align text along curved segments
Interaction	`nodeClick`, `emphasis.focus`	Zoom and highlight for user engagement
Animations	`animationDuration`, `animationEasing`	Smooth transitions on load and interaction

Sunburst charts combine hierarchy and proportion in a visually striking radial layout, making them ideal for exploring complex nested data intuitively and interactively.

5.4 Graphs and Network Diagrams

Graphs and network diagrams are essential for visualizing relationships and connections between entities. They are widely used in fields such as social network analysis, communication flow mapping, dependency graphs, and transportation networks. ECharts provides powerful tools to create interactive, customizable graph visualizations that reveal patterns, clusters, and link structures.

5.4.1 Use Cases for Graph and Network Diagrams

- **Social Networks:** Visualize friendships, followers, or professional connections.
- **Communication Flows:** Map email exchanges, call records, or message traffic.
- **Dependency Graphs:** Show task dependencies or software module relations.

- **Transportation Networks:** Display routes, stops, and connections between locations.

5.4.2 Defining Nodes, Edges, and Categories

In ECharts, a graph consists mainly of three components:

- **Nodes:** Represent entities or points in the network.
- **Edges (Links):** Define the connections between nodes.
- **Categories:** Optional groups that classify nodes by type or role, often used for color-coding and legend grouping.

5.4.3 Example Data Structure

```
const option = {
  title: {
    text: 'Social Network Example'
  },
  tooltip: {},
  legend: [{
    data: ['Person', 'Company', 'Project']
  }],
  series: [{
    type: 'graph',
    layout: 'force',   // Force-directed layout for natural positioning
    roam: true,        // Enable panning and zooming
    label: {
      show: true,
      position: 'right'
    },
    categories: [
      { name: 'Person', itemStyle: { color: '#ff7f50' } },
      { name: 'Company', itemStyle: { color: '#87cefa' } },
      { name: 'Project', itemStyle: { color: '#da70d6' } }
    ],
    data: [
      { id: '1', name: 'Alice', category: 0 },
      { id: '2', name: 'Bob', category: 0 },
      { id: '3', name: 'Acme Corp', category: 1 },
      { id: '4', name: 'Project X', category: 2 }
    ],
    links: [
      { source: '1', target: '2' },
      { source: '1', target: '3' },
      { source: '2', target: '3' },
      { source: '3', target: '4' }
    ],
    force: {
      repulsion: 200,       // Distance between nodes to avoid overlap
      edgeLength: [50, 100] // Desired edge length range
```

```
    },
    edgeSymbol: ['none', 'arrow'],  // Arrow at the target end of links
    edgeSymbolSize: [0, 10],
    lineStyle: {
      color: 'source',
      width: 2,
      curveness: 0.2   // Slightly curved edges for clarity
    }
  }]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Social Network Graph</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #chart {
      width: 100%;
      max-width: 900px;
      height: 600px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="chart"></div>

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

    const option = {
      title: {
        text: 'Social Network Example',
        left: 'center'
      },
      tooltip: {},
      legend: [{
        data: ['Person', 'Company', 'Project'],
        top: 30
      }],
      series: [{
        type: 'graph',
        layout: 'force',
        roam: true,
        label: {
          show: true,
          position: 'right'
        },
        categories: [
          { name: 'Person', itemStyle: { color: '#ff7f50' } },
          { name: 'Company', itemStyle: { color: '#87cefa' } },
          { name: 'Project', itemStyle: { color: '#da70d6' } }
        ],
```

```
        data: [
          { id: '1', name: 'Alice', category: 0 },
          { id: '2', name: 'Bob', category: 0 },
          { id: '3', name: 'Acme Corp', category: 1 },
          { id: '4', name: 'Project X', category: 2 }
        ],
        links: [
          { source: '1', target: '2' },
          { source: '1', target: '3' },
          { source: '2', target: '3' },
          { source: '3', target: '4' }
        ],
        force: {
          repulsion: 200,
          edgeLength: [50, 100]
        },
        edgeSymbol: ['none', 'arrow'],
        edgeSymbolSize: [0, 10],
        lineStyle: {
          color: 'source',
          width: 2,
          curveness: 0.2
        }
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.4.4 Key Concepts and Customizations

Concept	Configuration	Description
Nodes	`series.data`	Each node with `id`, `name`, and `category` to group and style
Edges (Links)	`series.links`	Connect nodes with `source` and `target` ids
Categories	`series.categories`	Define groups with colors and legends
Force Layout	`series.layout = 'force'`	Automatic node placement simulating physical forces
Repulsion	`force.repulsion`	Controls spacing between nodes to avoid clutter
Edge Length	`force.edgeLength`	Sets preferred distances between connected nodes
Edge Symbols	`edgeSymbol` and `edgeSymbolSize`	Add arrowheads or other markers to edges
Curveness	`lineStyle.curveness`	Adds curvature to edges for readability

5.4.5 Interaction Features

- **Roaming:** Enable zooming and panning for large networks using `roam: true`.
- **Dragging:** Nodes can be dragged interactively by default in force layouts, allowing manual rearrangement.
- **Tooltips:** Show detailed information about nodes or edges on hover.
- **Highlighting:** Categories can be highlighted via legend interaction.

5.4.6 Summary

Graphs and network diagrams in ECharts provide a rich toolkit for representing complex relational data in a clear, interactive format. With force-directed layouts, customizable node and edge styles, and grouping by categories, you can build insightful visualizations that reveal structure and connectivity within your data.

5.5 Geographic and Map Visualizations

Geographic and map visualizations enable you to display spatial data on geographic regions or coordinate systems, making it easier to analyze patterns across locations such as countries, states, or cities. ECharts supports powerful geographic rendering with `geo` and `map` components, allowing you to work with built-in maps or import your own GeoJSON data.

5.5.1 Using Built-in Maps with `geo`

ECharts provides built-in support for common maps like countries or provinces (e.g., China map). The `geo` component sets up the geographic coordinate system, which can be combined with series such as `scatter`, `map`, or `heatmap` to plot data.

5.5.2 Example: Basic China Map with Regional Data

```
const option = {
  title: {
    text: 'Sales by Province in China',
    left: 'center'
  },
  tooltip: {
    trigger: 'item',
```

```
    formatter: '{b}<br/>Sales: {c}'
  },
  visualMap: {
    min: 0,
    max: 1000,
    left: 'left',
    top: 'bottom',
    text: ['High', 'Low'],
    calculable: true,
    inRange: {
      color: ['#e0ffff', '#006edd']
    }
  },
  series: [{
    name: 'Sales',
    type: 'map',
    map: 'china',          // Use built-in China map
    roam: true,            // Enable zooming and panning
    data: [
      { name: 'Beijing', value: 800 },
      { name: 'Shanghai', value: 700 },
      { name: 'Guangdong', value: 900 },
      { name: 'Sichuan', value: 300 }
      // ... more provinces
    ]
  }]
};
```

- The `map` property selects a built-in map by name.
- `data` associates values with regions by name.
- `visualMap` colors regions based on value ranges.
- `roam` enables user interaction such as zoom and pan.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>China Map Example</title>
  <style>
    html, body {
      margin: 0;
      padding: 0;
    }
    #chart {
      width: 100%;
      height: 600px;
    }
  </style>
  <!-- ECharts Core -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <!-- China Map Data -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/map/js/china.js"></script>
</head>
<body>
```

```
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Sales by Province in China',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{b}<br/>Sales: {c}'
      },
      visualMap: {
        min: 0,
        max: 1000,
        left: 'left',
        top: 'bottom',
        text: ['High', 'Low'],
        calculable: true,
        inRange: {
          color: ['#e0ffff', '#006edd']
        }
      },
      series: [{
        name: 'Sales',
        type: 'map',
        map: 'china',
        roam: true,
        label: {
          show: true
        },
        data: [
          { name: 'Beijing', value: 800 },
          { name: 'Shanghai', value: 700 },
          { name: 'Guangdong', value: 900 },
          { name: 'Sichuan', value: 300 },
          { name: 'Zhejiang', value: 600 },
          { name: 'Hunan', value: 500 }
        ]
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.5.3 Importing and Using Custom GeoJSON Maps

To visualize custom regions or countries not included in the built-in maps, import GeoJSON data and register it with ECharts:

```
// Load GeoJSON (can be via AJAX or import)
fetch('path/to/custom-region.geojson')
  .then(response => response.json())
  .then(geoJson => {
    echarts.registerMap('customRegion', geoJson);

    const option = {
      series: [{
        type: 'map',
        map: 'customRegion',  // Use the registered custom map
        roam: true,
        data: [
          { name: 'Region A', value: 120 },
          { name: 'Region B', value: 300 }
        ]
      }]
    };

    myChart.setOption(option);
  });
```

- `echarts.registerMap` registers the custom GeoJSON under a name.
- Then the series `map` property references this name.
- This allows visualization of any geographic area defined by the GeoJSON.

5.5.4 Using the geo Component for Scatter and Heatmap Layers

The `geo` component defines the coordinate system, and series like `scatter` or `heatmap` plot points or regions over the map:

```
const option = {
  geo: {
    map: 'world',
    roam: true,
    label: { show: false },
    itemStyle: { areaColor: '#eee', borderColor: '#111' }
  },
  visualMap: {
    min: 0,
    max: 100,
    calculable: true,
    inRange: { color: ['#50a3ba', '#eac736', '#d94e5d'] }
  },
  series: [
    {
      name: 'Air Quality',
      type: 'scatter',
      coordinateSystem: 'geo',
      data: [
        { name: 'New York', value: [-74.006, 40.7128, 65] },
        { name: 'London', value: [-0.1276, 51.5074, 50] },
        { name: 'Beijing', value: [116.4074, 39.9042, 85] }
      ],
```

```
      symbolSize: val => val[2] / 2,  // Size proportional to value
      encode: { longitude: 0, latitude: 1, value: 2 },
      label: { formatter: '{b}', position: 'right', show: false },
      emphasis: { label: { show: true } }
    }
  ]
};
```

- Coordinates use `[longitude, latitude]` in the `value` array.
- `coordinateSystem: 'geo'` links the series to the `geo` map.
- `symbolSize` scales points by the data value.
- Tooltips and labels provide interactivity and clarity.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Geo Scatter Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- ECharts core library -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <!-- World map data -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/map/js/world.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      geo: {
        map: 'world',
        roam: true,
        label: { show: false },
        itemStyle: {
          areaColor: '#eee',
          borderColor: '#111'
        }
      },
      visualMap: {
        min: 0,
        max: 100,
        calculable: true,
        inRange: {
          color: ['#50a3ba', '#eac736', '#d94e5d']
```

```
        },
        left: 'left',
        bottom: '5%'
      },
      series: [
        {
          name: 'Air Quality',
          type: 'scatter',
          coordinateSystem: 'geo',
          data: [
            { name: 'New York', value: [-74.006, 40.7128, 65] },
            { name: 'London', value: [-0.1276, 51.5074, 50] },
            { name: 'Beijing', value: [116.4074, 39.9042, 85] }
          ],
          symbolSize: val => val[2] / 2,
          encode: {
            longitude: 0,
            latitude: 1,
            value: 2
          },
          label: {
            formatter: '{b}',
            position: 'right',
            show: false
          },
          emphasis: {
            label: {
              show: true
            }
          }
        }
      ]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

5.5.5 Choropleth Maps: Coloring Regions by Value

Using the `map` series with `visualMap` allows coloring of regions to create choropleth maps, useful for demographic or economic data visualization.

5.5.6 Summary

Feature	Description	Example Usage
Built-in Maps	Use predefined maps like countries or provinces	Visualizing sales by Chinese provinces
Custom GeoJSON	Import and register any geographic data	Visualizing custom regions, districts, or zones
Geo Component	Defines geographic coordinate system	Plotting points, scatter, or heatmaps over a map
Coordinate Format	`[longitude, latitude]`	Geospatial positioning of data points
Choropleth Maps	Color regions by data value	Showing population density or election results
Interaction	Zoom, pan, tooltip, label controls	Enhances user exploration of spatial data

By combining geographic maps with rich data overlays, ECharts makes spatial data visualization accessible and highly customizable for a wide range of applications.

5.6 Parallel Coordinates and Other Specialized Charts

When working with multidimensional datasets—such as features in machine learning, financial indicators, or experimental measurements—visualizing relationships across many variables simultaneously can be challenging. Parallel coordinate charts offer a powerful solution by representing each dimension as a vertical axis and plotting data points as lines crossing these axes.

5.6.1 What Are Parallel Coordinate Charts?

- Each **vertical axis** corresponds to one variable or feature.
- Each **data point** is drawn as a polyline intersecting the axes at positions corresponding to its values.
- Patterns, clusters, or outliers emerge from how lines group or cross.
- Useful for **exploring correlations, distributions, and trends** in high-dimensional data.

5.6.2 Creating a Parallel Coordinate Chart in ECharts

Here's a simple example visualizing three features for a set of data points:

```
const option = {
  title: {
    text: 'Parallel Coordinates Example'
  },
  tooltip: {
    trigger: 'item'
  },
  parallelAxis: [
    { dim: 0, name: 'Feature A' },
    { dim: 1, name: 'Feature B' },
    { dim: 2, name: 'Feature C' }
  ],
  parallel: {
    left: 50,
    right: 50,
    bottom: 50,
    top: 80,
    parallelAxisDefault: {
      type: 'value',
      nameLocation: 'end',
      nameGap: 20,
      nameTextStyle: { fontSize: 14 },
      axisLine: { lineStyle: { color: '#aaa' } },
      axisTick: { lineStyle: { color: '#aaa' } },
      splitLine: { show: true }
    }
  },
  series: [{
    name: 'Data',
    type: 'parallel',
    lineStyle: { width: 2 },
    data: [
      [10, 20, 30],
      [15, 25, 10],
      [25, 15, 20],
      [30, 35, 25]
    ]
  }]
};
```

- Each array in `data` corresponds to one data point across all dimensions.
- `parallelAxis` defines the individual vertical axes with labels.
- `parallel` sets the layout and style for the coordinate system.
- Tooltips help examine individual line values on hover.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Parallel Coordinates Chart</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
```

```
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- ECharts library -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Parallel Coordinates Example'
      },
      tooltip: {
        trigger: 'item'
      },
      parallelAxis: [
        { dim: 0, name: 'Feature A' },
        { dim: 1, name: 'Feature B' },
        { dim: 2, name: 'Feature C' }
      ],
      parallel: {
        left: 50,
        right: 50,
        bottom: 50,
        top: 80,
        parallelAxisDefault: {
          type: 'value',
          nameLocation: 'end',
          nameGap: 20,
          nameTextStyle: { fontSize: 14 },
          axisLine: { lineStyle: { color: '#aaa' } },
          axisTick: { lineStyle: { color: '#aaa' } },
          splitLine: { show: true }
        }
      },
      series: [{
        name: 'Data',
        type: 'parallel',
        lineStyle: { width: 2 },
        data: [
          [10, 20, 30],
          [15, 25, 10],
          [25, 15, 20],
          [30, 35, 25]
        ]
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
```

```
</html>
```

5.6.3 Other Specialized Chart Types in ECharts

ECharts also supports several other advanced chart types designed for specific use cases:

Chart Type	Use Case	Brief Description
Sankey Diagram	Visualizing flow quantities between entities	Shows how resources, energy, or money move between nodes with width-proportional links
Calendar Chart	Time-series data over a calendar layout	Maps values to calendar days for pattern discovery (e.g., daily sales or activity)
Funnel Chart	Stages in a process (e.g., sales pipeline)	Visualizes reduction across sequential stages
Gauge Chart	KPI and metric visualization	Displays progress or performance on a dial

Each of these charts comes with customizable options for data binding, styling, and interaction, enabling rich, domain-specific insights.

5.6.4 Summary

- **Parallel coordinate charts** help you analyze multidimensional datasets by drawing lines across vertical axes representing features.
- They reveal complex correlations and patterns not easily seen in traditional 2D charts.
- **ECharts** makes it straightforward to build these charts with flexible axis configuration and styling.
- Explore **Sankey**, **calendar**, **funnel**, and **gauge** charts for other specialized visualization needs.

Chapter 6.

Interactivity and Animation

1. Tooltip Customization
2. Zooming, Panning, and Data Zoom Components
3. Highlighting and Selecting Data
4. Animation Settings and Transitions

6 Interactivity and Animation

6.1 Tooltip Customization

Tooltips are a crucial part of interactive charts, providing detailed information when users hover over or click on data points. In ECharts, the `tooltip` component is highly customizable, allowing you to control when tooltips appear, what content they display, and how they look. This section covers advanced tooltip customization techniques to help you create insightful and visually appealing tooltips.

6.1.1 Basic Tooltip Configuration

At its simplest, adding a tooltip is as easy as enabling it in the chart option:

```
tooltip: {
  show: true,
  trigger: 'item'  // Show tooltip on each data item hover
}
```

The `trigger` can be `'item'` (default) or `'axis'`, the latter showing tooltips for all series aligned on a given axis value.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Tooltip Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- ECharts -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Basic Tooltip Example',
        left: 'center'
```

```
      },
      tooltip: {
        show: true,
        trigger: 'item' // Show tooltip when hovering a data item
      },
      series: [{
        type: 'pie',
        radius: '50%',
        data: [
          { value: 40, name: 'Apples' },
          { value: 30, name: 'Bananas' },
          { value: 20, name: 'Cherries' },
          { value: 10, name: 'Dates' }
        ],
        emphasis: {
          itemStyle: {
            shadowBlur: 10,
            shadowOffsetX: 0,
            shadowColor: 'rgba(0, 0, 0, 0.5)'
          }
        }
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.2 Formatting Tooltip Content

You can format the tooltip content with a string template or a callback function to customize how data is displayed.

6.1.3 Using String Template

```
tooltip: {
  formatter: '{a} <br/>{b}: {c}'
}
```

- `{a}`: series name
- `{b}`: data item name or category
- `{c}`: data item value

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tooltip String Template Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- ECharts -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Fruit Sales',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: '{a} <br/>{b}: {c} units'  // 'a' = series name, 'b' = data name, 'c' = value
      },
      series: [{
        name: 'Sales',
        type: 'pie',
        radius: '50%',
        data: [
          { value: 1048, name: 'Apples' },
          { value: 735, name: 'Bananas' },
          { value: 580, name: 'Cherries' },
          { value: 484, name: 'Dates' },
          { value: 300, name: 'Elderberries' }
        ]
      }]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.4 Using a Callback Function

For more control, provide a formatter function that receives the data and returns custom HTML or text:

```
tooltip: {
  formatter: params => {
    if (Array.isArray(params)) {
      // Multi-series tooltip (trigger: 'axis')
      return params.map(p => `
        <div>
          <strong>${p.seriesName}</strong>: ${p.data}
        </div>
      `).join('');
    } else {
      // Single data point tooltip
      return `<div>${params.seriesName}<br/>${params.name}: <em>${params.value}</em></div>`;
    }
  }
}
```

This example handles both single and multiple series tooltips, formatting content with HTML tags for styling.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tooltip Callback Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- ECharts -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Monthly Sales by Product',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        formatter: params => {
```

```
          if (Array.isArray(params)) {
            return `
              <div>
                <strong>${params[0].axisValue}</strong><br/>
                ${params.map(p => `
                  <span style="color:${p.color}">● </span>
                  ${p.seriesName}: <b>${p.data}</b>
                `).join('<br/>')}
              </div>
            `;
          } else {
            return `<div>${params.seriesName}<br/>${params.name}: <em>${params.value}</em></div>`;
          }
        }
      },
      legend: { top: '10%' },
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr']
      },
      yAxis: { type: 'value' },
      series: [
        {
          name: 'Laptops',
          type: 'line',
          data: [120, 200, 150, 80]
        },
        {
          name: 'Phones',
          type: 'line',
          data: [180, 100, 250, 130]
        }
      ]
    };

    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.5 Conditional Showing and Hiding

You can control whether a tooltip appears based on data or other conditions by returning `false` in the formatter or using the `formatter` function smartly:

```
tooltip: {
  formatter: params => {
    // Hide tooltip for zero values
    if (params.value === 0) {
      return false;
    }
    return `${params.seriesName} - ${params.name}: ${params.value}`;
  }
```

```
}
```

Returning `false` suppresses the tooltip display for that point.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Conditional Tooltip Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
  <!-- Load ECharts -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Sales Data (Tooltips Hide on Zero)',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: params => {
          //  Conditionally suppress tooltip
          if (params.value === 0) return false;
          return `${params.seriesName} - ${params.name}: ${params.value}`;
        }
      },
      xAxis: {
        type: 'category',
        data: ['Q1', 'Q2', 'Q3', 'Q4']
      },
      yAxis: { type: 'value' },
      series: [{
        name: 'Product A',
        type: 'bar',
        data: [120, 0, 90, 0],  //  Tooltips will not show for zero
        itemStyle: {
          color: '#73c0de'
        }
      }]
    };
```

```
    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.6 Rich Text and Styling

ECharts supports rich text styles inside tooltips using the `{styleName|text}` syntax, defined in the `rich` property:

```
tooltip: {
  formatter: params => {
    return `{title|${params.seriesName}}\n{value|${params.value}}`;
  },
  backgroundColor: '#222',
  borderColor: '#aaa',
  borderWidth: 1,
  textStyle: {
    color: '#fff',
    fontSize: 14
  },
  rich: {
    title: {
      color: '#ffd700',
      fontWeight: 'bold',
      fontSize: 16
    },
    value: {
      color: '#87ceeb',
      fontSize: 14,
      padding: [4, 0, 0, 0]
    }
  }
}
```

This approach lets you style different parts of the tooltip text distinctly, enhancing readability and visual appeal.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Rich Text Tooltip Example</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
    }
    #chart {
      width: 100%;
```

```
      height: 100%;
    }
  </style>
  <!-- Load ECharts -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Rich Text Tooltip Styling',
        left: 'center'
      },
      tooltip: {
        trigger: 'item',
        formatter: params => {
          return `{title|${params.seriesName}}\n{value|${params.name}: ${params.value}}`;
        },
        backgroundColor: '#222',
        borderColor: '#aaa',
        borderWidth: 1,
        textStyle: {
          color: '#fff',
          fontSize: 14,
          rich: {
            title: {
              color: '#ffd700',
              fontWeight: 'bold',
              fontSize: 16,
              lineHeight: 24
            },
            value: {
              color: '#87ceeb',
              fontSize: 14,
              padding: [4, 0, 0, 0]
            }
          }
        }
      },
      xAxis: {
        type: 'category',
        data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        name: 'Sales',
        type: 'bar',
        data: [120, 200, 150, 80, 70],
        itemStyle: {
          color: '#73c0de'
        }
      }]
    };
```

```
    chart.setOption(option);
    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.7 Practical Example: Comparing Multiple Series with Shared Tooltip

When multiple series share the same axis, a combined tooltip shows values for all series at the hovered axis coordinate:

```
tooltip: {
  trigger: 'axis',
  formatter: params => {
    let content = `<strong>${params[0].axisValue}</strong><br/>`;
    params.forEach(p => {
      content += `${p.marker} ${p.seriesName}: ${p.data}<br/>`;
    });
    return content;
  }
}
```

- `params` is an array containing info for each series.
- The tooltip shows all series values side-by-side for easy comparison.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Shared Tooltip Example</title>
  <style>
    html, body, #chart {
      margin: 0; padding: 0; height: 100vh; width: 100vw;
    }
  </style>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Sales by Region (2023 vs 2024)',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        formatter: params => {
```

```
          let content = `<strong>${params[0].axisValue}</strong><br/>`;
          params.forEach(p => {
            content += `${p.marker} ${p.seriesName}: ${p.data}<br/>`;
          });
          return content;
        }
      },
      legend: {
        data: ['2023', '2024'],
        top: 'bottom'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West']
      },
      yAxis: {
        type: 'value',
        name: 'Sales (in $1000)'
      },
      series: [
        {
          name: '2023',
          type: 'bar',
          data: [320, 450, 300, 500],
          itemStyle: { color: '#5470C6' }
        },
        {
          name: '2024',
          type: 'bar',
          data: [400, 470, 320, 550],
          itemStyle: { color: '#91CC75' }
        }
      ]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.1.8 Summary

Feature	Description	Example Use Case
`trigger: 'item'` or `'axis'`	Controls tooltip activation behavior	Show single point vs. multi-series info
Formatter string/template	Simple substitution-based content formatting	Display series name and value
Formatter callback	Full control over tooltip content and logic	Conditional display or custom HTML

Feature	Description	Example Use Case
Conditional display	Return `false` to hide tooltips for certain points	Hide zero or invalid data
Rich text styling	Apply multiple text styles within tooltip content	Highlight important data visually
Multi-series tooltips	Show all series data for a single axis coordinate	Compare values across multiple lines

With these powerful customization options, you can design tooltips that fit your chart's style and purpose perfectly, improving the clarity and interactivity of your visualizations.

6.2 Zooming, Panning, and Data Zoom Components

Interactive zooming and panning enhance the user experience by allowing users to explore detailed portions of large datasets or time series without losing context. ECharts provides the versatile `dataZoom` component to enable such interactions seamlessly.

6.2.1 What is the `dataZoom` Component?

The `dataZoom` component allows users to zoom into and pan across data along an axis dynamically. It acts as a viewport control for your chart's data, letting you focus on specific regions while still viewing overall trends.

6.2.2 Types of `dataZoom`

ECharts supports two primary types of `dataZoom`:

Type	Description	Interaction Style
Inside	Zoom and pan via mouse wheel, pinch gestures, or drag within the chart area	Mouse wheel scroll and drag directly on chart axes
Slider	A draggable slider bar (usually below or beside the chart) for zoom control	Visual slider UI for precise zooming

You can enable either or both types to offer different zooming experiences.

6.2.3 Enabling Inside Zooming and Panning

To enable zooming and panning directly inside the chart area, add an `inside` dataZoom targeting the desired axis:

```
dataZoom: [{
  type: 'inside',
  xAxisIndex: 0,       // Enable zoom on x-axis
  start: 0,            // Initial zoom window start percentage
  end: 100             // Initial zoom window end percentage
}]
```

- Users can zoom using mouse wheel or touch pinch gestures.
- Dragging horizontally pans the data view.
- Supports multiple axes by specifying different `xAxisIndex` or `yAxisIndex`.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Inside Zoom & Pan Example</title>
  <style>
    html, body, #chart {
      margin: 0; padding: 0; height: 100vh; width: 100vw;
    }
  </style>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Sales by Region with Inside Zoom & Pan',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West', 'Central', 'Northeast', 'Southwest']
      },
      yAxis: {
        type: 'value',
        name: 'Sales (in $1000)'
      },
      dataZoom: [
        {
          type: 'inside',
          xAxisIndex: 0,
          start: 0,
          end: 100
```

```
        }
      ],
      series: [{
        name: 'Sales',
        type: 'bar',
        data: [320, 450, 300, 500, 420, 380, 460],
        itemStyle: { color: '#5470C6' }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.2.4 Adding a Slider Zoom Control

A slider provides a visible UI for zooming and panning, useful when users need precise control or a timeline overview:

```
dataZoom: [{
  type: 'slider',
  xAxisIndex: 0,
  start: 20,   // Show data from 20% to 80%
  end: 80,
  bottom: 10,  // Position slider below the chart
  height: 20,
  handleSize: '100%',  // Size of the draggable handle
  fillerColor: 'rgba(167,183,204,0.4)',  // Slider fill color
  borderColor: '#aaa'
}]
```

- The slider bar can be styled and positioned.
- Users drag the handles to adjust the zoom window.
- The chart updates dynamically as the window changes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Slider Zoom Control Example</title>
  <style>
    html, body, #chart {
      margin: 0; padding: 0; height: 100vh; width: 100vw;
    }
  </style>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
```

```
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    const option = {
      title: {
        text: 'Sales by Region with Slider Zoom',
        left: 'center'
      },
      tooltip: {
        trigger: 'axis'
      },
      xAxis: {
        type: 'category',
        data: ['North', 'South', 'East', 'West', 'Central', 'Northeast', 'Southwest']
      },
      yAxis: {
        type: 'value',
        name: 'Sales (in $1000)'
      },
      dataZoom: [
        {
          type: 'slider',
          xAxisIndex: 0,
          start: 20,           // Show data from 20% to 80%
          end: 80,
          bottom: 10,          // Position slider below the chart
          height: 20,
          handleSize: '100%',  // Size of the draggable handle
          fillerColor: 'rgba(167,183,204,0.4)',
          borderColor: '#aaa'
        }
      ],
      series: [{
        name: 'Sales',
        type: 'bar',
        data: [320, 450, 300, 500, 420, 380, 460],
        itemStyle: { color: '#5470C6' }
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.2.5 Common Use Case: Timeline-Based Data Navigation

When working with time series data, a slider zoom lets users easily select date ranges without losing sight of the entire timeline.

```
option = {
  xAxis: {
    type: 'time'
  },
  dataZoom: [
    {
      type: 'slider',
      start: 30,
      end: 70
    },
    {
      type: 'inside',
      start: 30,
      end: 70
    }
  ],
  series: [{
    type: 'line',
    data: [ /* [timestamp, value] pairs */ ]
  }]
};
```

- The slider appears below the chart for selecting date ranges.
- Inside zoom lets users zoom/pan directly on the line chart.
- Synchronizing both types provides intuitive and flexible navigation.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Time Series with Slider Zoom</title>
  <style>
    html, body, #chart {
      margin: 0; padding: 0; height: 100vh; width: 100vw;
    }
  </style>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));

    // Generate sample time series data: daily values for June 2024
    const startDate = new Date(2024, 5, 1); // June 1, 2024
    const data = [];
    for (let i = 0; i < 30; i++) {
      const date = new Date(startDate.getTime() + i * 86400000);
      const value = Math.round(50 + 30 * Math.sin(i / 5) + Math.random() * 10);
      data.push([date.toISOString().slice(0,10), value]);
    }

    const option = {
      title: {
        text: 'Daily Sales in June 2024',
```

```
        left: 'center'
      },
      tooltip: {
        trigger: 'axis',
        formatter: params => {
          const p = params[0];
          return `${p.axisValue}<br/>Sales: ${p.data[1]}`;
        }
      },
      xAxis: {
        type: 'time',
        boundaryGap: false
      },
      yAxis: {
        type: 'value',
        name: 'Sales'
      },
      dataZoom: [
        {
          type: 'slider',
          start: 30,
          end: 70,
          bottom: 10,
          height: 20
        },
        {
          type: 'inside',
          start: 30,
          end: 70
        }
      ],
      series: [{
        type: 'line',
        data: data,
        smooth: true,
        lineStyle: { width: 2 },
        symbol: 'circle',
        symbolSize: 6
      }]
    };

    chart.setOption(option);

    window.addEventListener('resize', () => chart.resize());
  </script>
</body>
</html>
```

6.2.6 Multiple Axes and Data Zoom

You can enable zooming on multiple axes by adding more `dataZoom` components:

```
dataZoom: [
  { type: 'slider', xAxisIndex: 0, start: 0, end: 50 },
```

```
  { type: 'inside', yAxisIndex: 0, start: 0, end: 100 }
]
```

This allows independent zooming along horizontal and vertical axes.

6.2.7 Summary

Feature	Description
`dataZoom.type`	`'inside'` for mouse/touch zoom, `'slider'` for draggable UI slider
`start` / `end`	Initial zoom window percentage (0-100%)
`xAxisIndex` / `yAxisIndex`	Specify which axis the zoom applies to
Multiple dataZooms	Combine inside and slider zooms for versatile interaction
Timeline navigation	Perfect for selecting date ranges in time series charts

By integrating `dataZoom` components, you empower your users to dive deep into data details while maintaining a smooth, responsive experience. This capability is essential for dashboards, monitoring systems, and any application requiring exploration of large or time-based datasets.

6.3 Highlighting and Selecting Data

Interactivity in charts goes beyond tooltips and zooming—highlighting and selecting specific data points or series allows users to focus on important information and analyze subsets of data effectively. ECharts provides a powerful API to control highlighting and selection both programmatically and through user interactions like clicks and hovers.

6.3.1 Highlighting and Downplaying Data

- **Highlighting** makes a chart element visually stand out by changing its style (e.g., color, opacity).
- **Downplaying** returns the element to its normal, less prominent style.
- Useful for emphasizing data points, lines, bars, or areas on demand.

6.3.2 Programmatic API

ECharts instances provide the following methods:

```
myChart.dispatchAction({
  type: 'highlight',       // Emphasize an element
  seriesIndex: 0,          // Target series index
  dataIndex: 2             // Target data point index
});

myChart.dispatchAction({
  type: 'downplay',        // Remove emphasis
  seriesIndex: 0,
  dataIndex: 2
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Highlight/Downplay Demo</title>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="main"></div>

  <!-- Import ECharts from CDN -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    // Initialize chart
    const chartDom = document.getElementById('main');
    const myChart = echarts.init(chartDom);

    const option = {
      title: {
        text: 'Bar Chart Highlight/Downplay Demo',
        left: 'center'
      },
      xAxis: {
        type: 'category',
        data: ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
        data: [5, 20, 36, 10, 10]
      }]
    };
```

```
    myChart.setOption(option);

    // Highlight the 3rd bar (index 2)
    myChart.dispatchAction({
      type: 'highlight',
      seriesIndex: 0,
      dataIndex: 2
    });

    // After 2 seconds, remove the highlight
    setTimeout(() => {
      myChart.dispatchAction({
        type: 'downplay',
        seriesIndex: 0,
        dataIndex: 2
      });
    }, 2000);
  </script>
</body>
</html>
```

6.3.3 Selecting Data Points

Selection typically indicates a user's choice or focus and can persist beyond hover. Selected data points often change style and can trigger additional logic.

6.3.4 API for Selection

```
myChart.dispatchAction({
  type: 'select',
  seriesIndex: 0,
  dataIndex: 2
});

myChart.dispatchAction({
  type: 'unselect',
  seriesIndex: 0,
  dataIndex: 2
});
```

You can also select multiple points or entire series at once.

Full runnable code:

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

```
  <title>ECharts Select/Unselect Demo</title>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
    #controls {
      text-align: center;
      margin: 10px;
    }
    button {
      margin: 0 5px;
      padding: 8px 12px;
      font-size: 14px;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <div id="controls">
    <button onclick="selectBar(0)">Select Apple</button>
    <button onclick="selectBar(2)">Select Cherry</button>
    <button onclick="unselectBar(2)">Unselect Cherry</button>
    <button onclick="clearSelection()">Clear All Selection</button>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    const chartDom = document.getElementById('main');
    const myChart = echarts.init(chartDom);

    const option = {
      title: {
        text: 'Bar Chart Select/Unselect Demo',
        left: 'center'
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
        data: [5, 20, 36, 10, 10],
        // Enable selection style by specifying selected mode
        select: {
          itemStyle: {
            color: 'orange'
          }
        },
        emphasis: {
          itemStyle: {
            color: '#a0522d'
```

```
        }
      }
    }]
  };

  myChart.setOption(option);

  function selectBar(dataIndex) {
    myChart.dispatchAction({
      type: 'select',
      seriesIndex: 0,
      dataIndex: dataIndex
    });
  }

  function unselectBar(dataIndex) {
    myChart.dispatchAction({
      type: 'unselect',
      seriesIndex: 0,
      dataIndex: dataIndex
    });
  }

  function clearSelection() {
    myChart.dispatchAction({
      type: 'downplay',
      seriesIndex: 0
    });
    // ECharts does not have a direct clear all selections action
    // so unselect all indices explicitly if needed
    for(let i = 0; i < option.xAxis.data.length; i++) {
      unselectBar(i);
    }
  }
</script>
</body>
</html>
```

6.3.5 Enabling Interactive Highlighting on Hover

ECharts automatically highlights hovered elements, but you can customize this behavior or respond to hover events.

6.3.6 Example: Custom Highlight on Hover

```
myChart.on('mouseover', params => {
  myChart.dispatchAction({
    type: 'highlight',
    seriesIndex: params.seriesIndex,
```

```
    dataIndex: params.dataIndex
  });
});

myChart.on('mouseout', params => {
  myChart.dispatchAction({
    type: 'downplay',
    seriesIndex: params.seriesIndex,
    dataIndex: params.dataIndex
  });
});
```

This explicitly controls highlight on mouseover and removes it on mouseout, allowing for custom styles or logic.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Hover Highlight + Selection Demo</title>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
    #controls {
      text-align: center;
      margin: 10px;
    }
    button {
      margin: 0 5px;
      padding: 8px 12px;
      font-size: 14px;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <div id="controls">
    <button onclick="selectBar(0)">Select Apple</button>
    <button onclick="selectBar(2)">Select Cherry</button>
    <button onclick="unselectBar(2)">Unselect Cherry</button>
    <button onclick="clearSelection()">Clear All Selection</button>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    const chartDom = document.getElementById('main');
    const myChart = echarts.init(chartDom);

    const option = {
      title: {
        text: 'Bar Chart Hover Highlight + Selection Demo',
```

```
      left: 'center'
    },
    tooltip: {},
    xAxis: {
      type: 'category',
      data: ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
    },
    yAxis: {
      type: 'value'
    },
    series: [{
      type: 'bar',
      data: [5, 20, 36, 10, 10],
      select: {
        itemStyle: { color: 'orange' }
      },
      emphasis: {
        itemStyle: { color: '#a0522d' }
      }
    }]
  };

  myChart.setOption(option);

  // Custom highlight on hover
  myChart.on('mouseover', params => {
    if (params.componentType === 'series') {
      myChart.dispatchAction({
        type: 'highlight',
        seriesIndex: params.seriesIndex,
        dataIndex: params.dataIndex
      });
    }
  });

  // Remove highlight on mouseout
  myChart.on('mouseout', params => {
    if (params.componentType === 'series') {
      myChart.dispatchAction({
        type: 'downplay',
        seriesIndex: params.seriesIndex,
        dataIndex: params.dataIndex
      });
    }
  });

  // Selection API
  function selectBar(dataIndex) {
    myChart.dispatchAction({
      type: 'select',
      seriesIndex: 0,
      dataIndex: dataIndex
    });
  }

  function unselectBar(dataIndex) {
    myChart.dispatchAction({
      type: 'unselect',
```

```
        seriesIndex: 0,
        dataIndex: dataIndex
      });
    }

    function clearSelection() {
      for(let i = 0; i < option.xAxis.data.length; i++) {
        unselectBar(i);
      }
    }
  </script>
</body>
</html>
```

6.3.7 Example: Click to Select/Deselect Points

```
myChart.on('click', params => {
  if (params.selected) {
    myChart.dispatchAction({
      type: 'unselect',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  } else {
    myChart.dispatchAction({
      type: 'select',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});
```

This toggles selection state on click, useful for interactive filtering or drill-down features.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Click-to-Select Demo</title>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 40px auto;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
```

```
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);

const option = {
  title: {
    text: 'Bar Chart Click-to-Select Demo',
    left: 'center'
  },
  tooltip: {},
  xAxis: {
    type: 'category',
    data: ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
  },
  yAxis: {
    type: 'value'
  },
  series: [{
    type: 'bar',
    data: [5, 20, 36, 10, 10],
    select: {
      itemStyle: { color: 'orange' }
    },
    emphasis: {
      itemStyle: { color: '#a0522d' }
    }
  }]
};

myChart.setOption(option);

// Custom highlight on hover
myChart.on('mouseover', params => {
  if (params.componentType === 'series') {
    myChart.dispatchAction({
      type: 'highlight',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});

// Remove highlight on mouseout
myChart.on('mouseout', params => {
  if (params.componentType === 'series') {
    myChart.dispatchAction({
      type: 'downplay',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});

// Click to toggle selection
myChart.on('click', params => {
  if (params.selected) {
    myChart.dispatchAction({
      type: 'unselect',
      seriesIndex: params.seriesIndex,
```

```
          dataIndex: params.dataIndex
        });
      } else {
        myChart.dispatchAction({
          type: 'select',
          seriesIndex: params.seriesIndex,
          dataIndex: params.dataIndex
        });
      }
    });
  </script>
</body>
</html>
```

6.3.8 Styling Highlighted and Selected Items

Customize styles in your `option` to differentiate highlighted and selected states:

```
series: [{
  type: 'bar',
  data: [...],
  emphasis: {        // Highlight style
    itemStyle: {
      color: 'orange',
      opacity: 1
    }
  },
  select: {          // Selected style
    itemStyle: {
      color: 'red',
      borderWidth: 2,
      borderColor: '#900'
    }
  }
}]
```

- `emphasis` defines styles on highlight (hover or programmatic).
- `select` defines styles for persistent selection.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Highlight & Select Styling Demo</title>
<style>
  #main { width: 600px; height: 400px; margin: 40px auto; }
</style>
</head>
<body>
<div id="main"></div>
<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
```

```
<script>
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);

const option = {
  title: { text: 'Bar Chart: Highlight & Select Styling', left: 'center' },
  tooltip: {},
  xAxis: { type: 'category', data: ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry'] },
  yAxis: { type: 'value' },
  series: [{
    type: 'bar',
    data: [5, 20, 36, 10, 10],
    emphasis: {
      itemStyle: {
        color: 'orange',
        opacity: 1
      }
    },
    select: {
      itemStyle: {
        color: 'red',
        borderWidth: 3,
        borderColor: '#900'
      }
    }
  }]
};

myChart.setOption(option);

// Hover highlight
myChart.on('mouseover', params => {
  if (params.componentType === 'series') {
    myChart.dispatchAction({
      type: 'highlight',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});
myChart.on('mouseout', params => {
  if (params.componentType === 'series') {
    myChart.dispatchAction({
      type: 'downplay',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});

// Click toggles selection
myChart.on('click', params => {
  if (params.selected) {
    myChart.dispatchAction({
      type: 'unselect',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
```

```
  } else {
    myChart.dispatchAction({
      type: 'select',
      seriesIndex: params.seriesIndex,
      dataIndex: params.dataIndex
    });
  }
});
</script>
</body>
</html>
```

6.3.9 Summary

Feature	Description
`highlight`	Emphasize specific chart elements programmatically or on hover
`downplay`	Remove emphasis and restore default styling
`select` / `unselect`	Mark data points or series as selected or not
Event listeners	Use `mouseover`, `mouseout`, and `click` to trigger highlight/select actions
Styling options	Customize appearance for emphasis and selection states

Using these APIs, you can build dynamic, user-friendly charts that respond to interaction patterns like focusing on outliers, comparing data points, or filtering by selection.

6.4 Animation Settings and Transitions

Animations bring charts to life by smoothly transitioning between states, improving user engagement and making complex data changes easier to understand. ECharts offers rich animation controls that allow you to customize how charts appear, update, and transition over time.

6.4.1 How Animation Works in ECharts

When a chart is rendered or updated, ECharts animates the changes in visual elements such as bars, lines, or areas. This includes initial loading, data updates, or switching chart types.

Key animation properties let you control timing and easing:

Property	Description	Default Value
`animationDuration`	Duration of the animation in milliseconds	`1000` (1 second)
`animationEasing`	Easing function controlling animation curve	`'cubicOut'`
`animationDelay`	Delay before the animation starts (ms)	`0`

6.4.2 Customizing Animation on Initial Render

You can specify these properties globally or within individual series:

```
option = {
  animationDuration: 1500,
  animationEasing: 'elasticOut',
  series: [{
    type: 'bar',
    data: [10, 20, 30, 40],
    animationDelay: idx => idx * 300  // Stagger animation for each bar
  }]
};
```

- Here, each bar's animation starts 300ms after the previous, creating a cascading effect.
- The easing function `'elasticOut'` gives a bouncy transition.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Custom Animation Demo</title>
<style>
  #main { width: 600px; height: 400px; margin: 40px auto; }
</style>
</head>
<body>
<div id="main"></div>
<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);

const option = {
  title: { text: 'Bar Chart with Custom Animation', left: 'center' },
  animationDuration: 1500,
  animationEasing: 'elasticOut',
  xAxis: {
    type: 'category',
    data: ['A', 'B', 'C', 'D', 'E']
  },
  yAxis: { type: 'value' },
  series: [{
    type: 'bar',
```

```
    data: [10, 20, 30, 40, 50],
    animationDelay: idx => idx * 300
  }]
};

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

6.4.3 Animation During Data Updates

When you call `setOption` with new data, ECharts animates the transition from the old state to the new one, allowing smooth updates rather than abrupt changes.

```
myChart.setOption({
  series: [{
    data: [15, 25, 35, 45],
    animationDurationUpdate: 800,       // Duration for update animations
    animationEasingUpdate: 'linear'     // Easing during update
  }]
});
```

- `animationDurationUpdate` controls update animation speed.
- `animationEasingUpdate` defines how the transition curve behaves.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Animation During Data Update Demo</title>
<style>
  #main { width: 600px; height: 400px; margin: 40px auto; }
  button { display: block; margin: 20px auto; padding: 8px 16px; font-size: 16px; }
</style>
</head>
<body>
<div id="main"></div>
<button id="updateDataBtn">Update Data</button>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);

let data = [10, 20, 30, 40];

const option = {
  title: { text: 'Data Update Animation', left: 'center' },
  xAxis: { type: 'category', data: ['A', 'B', 'C', 'D'] },
  yAxis: { type: 'value' },
```

```
  series: [{
    type: 'bar',
    data,
    animationDurationUpdate: 800,
    animationEasingUpdate: 'linear'
  }]
};

myChart.setOption(option);

document.getElementById('updateDataBtn').addEventListener('click', () => {
  // Update data randomly
  data = data.map(v => v + Math.round(Math.random() * 20 - 10));
  myChart.setOption({
    series: [{
      data,
      animationDurationUpdate: 800,
      animationEasingUpdate: 'linear'
    }]
  });
});
</script>
</body>
</html>
```

6.4.4 Staggered Animations for Storytelling

Use the `animationDelay` or `animationDelayUpdate` callback functions to create staggered animation sequences that guide the viewer's attention step-by-step.

```
series: [{
  type: 'line',
  data: [5, 10, 15, 20, 25],
  animationDelay: idx => idx * 200,           // Initial staggered animation
  animationDelayUpdate: idx => idx * 100      // Staggered animation on update
}]
```

This technique is ideal for:

- Emphasizing trends as data points appear one by one.
- Creating smooth transitions in dashboards or presentations.
- Directing the user's eye through complex changes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Staggered Animation Demo</title>
<style>
  #main { width: 600px; height: 400px; margin: 40px auto; }
```

```
  button { display: block; margin: 20px auto; padding: 8px 16px; font-size: 16px; }
</style>
</head>
<body>
<div id="main"></div>
<button id="updateDataBtn">Update Data</button>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
const chartDom = document.getElementById('main');
const myChart = echarts.init(chartDom);

let data = [5, 10, 15, 20, 25];

const option = {
  title: { text: 'Staggered Animation Demo', left: 'center' },
  xAxis: { type: 'category', data: ['A', 'B', 'C', 'D', 'E'] },
  yAxis: { type: 'value' },
  series: [{
    type: 'line',
    data,
    animationDelay: idx => idx * 200,          // stagger on initial render
    animationDelayUpdate: idx => idx * 100,    // stagger on update
    lineStyle: { width: 3 },
    symbolSize: 10,
    smooth: true
  }]
};

myChart.setOption(option);

document.getElementById('updateDataBtn').addEventListener('click', () => {
  // Randomly change data points
  data = data.map(v => Math.max(0, v + Math.round(Math.random() * 10 - 5)));
  myChart.setOption({
    series: [{
      data,
      animationDelayUpdate: idx => idx * 100
    }]
  });
});
</script>
</body>
</html>
```

6.4.5 Common Animation Easing Options

Easing Name	Description
`linear`	Constant speed
`cubicOut`	Starts fast, ends slowly (default)
`elasticOut`	Bouncy effect

Easing Name	Description
`bounceOut`	Bounce effect
`quadInOut`	Smooth acceleration/deceleration

Try different easing options to match the tone and flow of your visualization.

6.4.6 Summary

Animation Feature	Purpose
`animationDuration`	Control how long initial animations last
`animationEasing`	Customize the easing curve for animations
`animationDelay`	Stagger animation start times per data item
`animationDurationUpdate`	Duration for animations during data updates
`animationEasingUpdate`	Easing for update transitions
Staggered animation callbacks	Guide user attention and storytelling

By thoughtfully applying ECharts' animation settings and transitions, you can transform static data into engaging narratives, making insights easier to grasp and more memorable.

Chapter 7.

Responsive Design and Theming

1. Making Charts Responsive
2. Using Built-in Themes and Creating Custom Themes
3. Dark Mode and Accessibility Considerations

7 Responsive Design and Theming

7.1 Making Charts Responsive

In today's multi-device world, ensuring your charts look great and function well on different screen sizes—from desktops to tablets and smartphones—is essential. ECharts provides flexible options and APIs to help you create responsive charts that adapt smoothly to layout changes and varying viewport dimensions.

7.1.1 Using Percentage-Based Dimensions

When initializing your chart container (`div`), avoid fixed pixel sizes. Instead, use CSS with relative units like percentages or viewport units to let the chart scale naturally.

```
<div id="chart" style="width: 100%; height: 400px;"></div>
```

- Setting `width: 100%` makes the chart fill its parent container horizontally.
- Adjust the height in CSS or using relative units (`vh`, `%`) to control vertical sizing.

7.1.2 Calling `resize()` on Window Resize

ECharts does **not automatically** detect container size changes. To ensure the chart resizes when the window or container changes, call the `resize` method on the chart instance:

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

// Initial option set
chart.setOption(option);

// Handle window resize events
window.addEventListener('resize', () => {
  chart.resize();
});
```

This redraws the chart to fit the new container dimensions, preserving clarity and layout.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>ECharts Resize Demo</title>
<style>
  #chart {
    width: 80vw;
    height: 400px;
```

```
    margin: 20px auto;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<div id="chart"></div>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
  const chartDom = document.getElementById('chart');
  const chart = echarts.init(chartDom);

  const option = {
    title: { text: 'Resizable Chart', left: 'center' },
    tooltip: {},
    xAxis: {
      type: 'category',
      data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    },
    yAxis: { type: 'value' },
    series: [{
      type: 'bar',
      data: [120, 200, 150, 80, 70, 110, 130]
    }]
  };

  chart.setOption(option);

  // Resize chart on window resize
  window.addEventListener('resize', () => {
    chart.resize();
  });
</script>

</body>
</html>
```

7.1.3 Responsive Option Changes with `media` Queries

ECharts supports `media` queries inside the option to change chart configurations based on viewport size or aspect ratio, enabling more granular responsiveness:

```
option = {
  // Base options for all screen sizes
  xAxis: { /* ... */ },
  yAxis: { /* ... */ },
  series: [{ /* ... */ }],
  media: [
    {
      query: { maxWidth: 600 },
      option: {
        // For narrow screens, simplify chart, e.g., fewer labels
```

```
        xAxis: { axisLabel: { show: false } },
        legend: { show: false }
      }
    },
    {
      query: { minWidth: 601 },
      option: {
        // For wider screens, show full details
        xAxis: { axisLabel: { show: true } },
        legend: { show: true }
      }
    }
  ]
};
```

- Use CSS-like queries (`maxWidth`, `minWidth`, `maxHeight`, `minHeight`) to define breakpoints.
- Provide alternative options optimized for different device sizes or orientations.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>ECharts Responsive Media Demo</title>
<style>
  #chart {
    width: 90vw;
    height: 400px;
    margin: 20px auto;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<div id="chart"></div>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
  const chartDom = document.getElementById('chart');
  const chart = echarts.init(chartDom);

  const option = {
    title: { text: 'Responsive Chart Example', left: 'center' },
    tooltip: {},
    legend: {
      data: ['Sales', 'Expenses'],
      bottom: 10
    },
    xAxis: {
      type: 'category',
      data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
      axisLabel: { show: true }
    },
    yAxis: { type: 'value' },
```

```
    series: [
      {
        name: 'Sales',
        type: 'bar',
        data: [120, 200, 150, 80, 70, 110]
      },
      {
        name: 'Expenses',
        type: 'bar',
        data: [90, 140, 130, 70, 60, 100]
      }
    ],
    media: [
      {
        query: { maxWidth: 600 },
        option: {
          legend: { show: false },
          xAxis: { axisLabel: { show: false } }
        }
      },
      {
        query: { minWidth: 601 },
        option: {
          legend: { show: true },
          xAxis: { axisLabel: { show: true } }
        }
      }
    ]
  };

  chart.setOption(option);

  // Also resize on window resize
  window.addEventListener('resize', () => {
    chart.resize();
  });
</script>

</body>
</html>
```

7.1.4 Responsive Dashboards: Handling Multiple Charts

In dashboards with multiple charts, ensure each chart resizes independently:

```
const charts = [
  echarts.init(document.getElementById('chart1')),
  echarts.init(document.getElementById('chart2')),
  echarts.init(document.getElementById('chart3'))
];

// Set options for each chart...
charts.forEach(chart => chart.setOption(option));
```

```
// Resize all charts on window resize
window.addEventListener('resize', () => {
  charts.forEach(chart => chart.resize());
});
```

This approach keeps your dashboard layout consistent and interactive across screen sizes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Multiple ECharts Example with Responsive Resize</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 10px;
      display: flex;
      flex-wrap: wrap;
      gap: 20px;
      background: #f5f5f5;
    }
    .chart-container {
      background: white;
      border: 1px solid #ccc;
      border-radius: 4px;
      flex: 1 1 300px;
      min-width: 280px;
      height: 300px;
      padding: 10px;
      box-sizing: border-box;
    }
    .chart-title {
      text-align: center;
      font-weight: bold;
      margin-bottom: 8px;
    }
  </style>
</head>
<body>

  <div id="chart1" class="chart-container">
    <div class="chart-title">Bar Chart</div>
    <div style="width:100%;height:250px;" id="chart1_inner"></div>
  </div>

  <div id="chart2" class="chart-container">
    <div class="chart-title">Line Chart</div>
    <div style="width:100%;height:250px;" id="chart2_inner"></div>
  </div>

  <div id="chart3" class="chart-container">
    <div class="chart-title">Pie Chart</div>
    <div style="width:100%;height:250px;" id="chart3_inner"></div>
  </div>
```

```
<!-- ECharts library from CDN -->
<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
  // Initialize charts
  const chart1 = echarts.init(document.getElementById('chart1_inner'));
  const chart2 = echarts.init(document.getElementById('chart2_inner'));
  const chart3 = echarts.init(document.getElementById('chart3_inner'));

  // Bar chart option with responsive media query
  const option1 = {
    title: { text: 'Sales by Region', left: 'center' },
    tooltip: {},
    xAxis: {
      type: 'category',
      data: ['North', 'South', 'East', 'West']
    },
    yAxis: { type: 'value' },
    series: [{
      type: 'bar',
      data: [320, 450, 300, 500],
      itemStyle: { color: '#5470C6' }
    }],
    media: [
      {
        query: { maxWidth: 400 },
        option: {
          xAxis: { axisLabel: { show: false } },
          title: { textStyle: { fontSize: 12 } }
        }
      }
    ]
  };

  // Line chart option with responsive media query
  const option2 = {
    title: { text: 'Monthly Sales', left: 'center' },
    tooltip: { trigger: 'axis' },
    xAxis: {
      type: 'category',
      data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
    },
    yAxis: { type: 'value' },
    series: [{
      type: 'line',
      data: [120, 200, 150, 80, 70, 110],
      smooth: true,
      areaStyle: { color: 'rgba(84, 112, 198, 0.4)' },
      symbol: 'circle',
      symbolSize: 6
    }],
    media: [
      {
        query: { maxWidth: 400 },
        option: {
          title: { textStyle: { fontSize: 12 } },
          series: [{ symbolSize: 3 }]
        }
      }
```

```
      ]
    };

    // Pie chart option with responsive media query
    const option3 = {
      title: { text: 'Market Share', left: 'center' },
      tooltip: { trigger: 'item', formatter: '{b}: {c} ({d}%)' },
      legend: { orient: 'horizontal', bottom: 10 },
      series: [{
        type: 'pie',
        radius: ['40%', '70%'],
        data: [
          { value: 335, name: 'Company A' },
          { value: 310, name: 'Company B' },
          { value: 234, name: 'Company C' },
          { value: 135, name: 'Company D' }
        ],
        label: {
          formatter: '{b}: {d}%',
          fontWeight: 'bold',
          fontSize: 12
        },
        labelLine: { smooth: true }
      }],
      media: [
        {
          query: { maxWidth: 400 },
          option: {
            legend: { show: false },
            series: [{ label: { fontSize: 8 } }]
          }
        }
      ]
    };

    // Set options on each chart
    chart1.setOption(option1);
    chart2.setOption(option2);
    chart3.setOption(option3);

    // Resize all charts on window resize
    window.addEventListener('resize', () => {
      chart1.resize();
      chart2.resize();
      chart3.resize();
    });
  </script>
</body>
</html>
```

7.1.5 Summary

Technique	Purpose
Percentage-based CSS	Allow container to scale fluidly
`chart.resize()`	Redraw chart on container or window size changes
`media` queries in option	Adjust chart configuration based on viewport size or aspect ratio
Multiple charts	Resize each chart instance individually in dashboards

By combining flexible container sizing, window resize handling, and conditional option adjustments, you can build ECharts visualizations that look sharp and function flawlessly on any device.

7.2 Using Built-in Themes and Creating Custom Themes

Themes control the overall look and feel of your charts, including colors, fonts, backgrounds, and styles. ECharts comes with several built-in themes that you can apply effortlessly, and also provides the flexibility to create and register your own custom themes to match your brand or design preferences.

7.2.1 Built-in ECharts Themes

ECharts includes several pre-defined themes you can apply when initializing a chart:

Theme Name	Description
`light`	The default theme with a clean, bright look
`dark`	Dark background with contrasting colors for nighttime or dark UI
`vintage`	Soft, muted colors with a retro feel
`macarons`	Bright and pastel color palette
`infographic`	Bold colors designed for infographics
`shine`	Glossy, shiny style
`roma`	Warm, elegant colors
`dark`	High contrast theme for dark environments

7.2.2 Applying a Built-in Theme

To apply a theme, pass its name as the second argument when initializing the chart:

```
// Use the 'dark' built-in theme
const chart = echarts.init(document.getElementById('chart'), 'dark');
```

```
chart.setOption({
  // Your chart option here
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Dark Theme Example</title>
  <style>
    /* Full viewport height and remove default margin */
    body, html {
      margin: 0;
      padding: 0;
      height: 100vh;
      background-color: #333;
    }
    /* Chart container fills viewport */
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
</head>
<body>
  <div id="chart"></div>

  <!-- ECharts library from CDN -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>

  <script>
    // Initialize chart with 'dark' theme
    const chart = echarts.init(document.getElementById('chart'), 'dark');

    const option = {
      title: {
        text: 'Sales Overview',
        left: 'center',
        textStyle: { color: '#fff' }
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
        axisLine: { lineStyle: { color: '#aaa' } }
      },
      yAxis: {
        type: 'value',
        axisLine: { lineStyle: { color: '#aaa' } },
        splitLine: { lineStyle: { color: '#555' } }
      },
      series: [{
        type: 'bar',
        data: [120, 200, 150, 80, 70, 110],
        itemStyle: {
```

```
          color: '#3398DB'
        }
      }]
    };

    chart.setOption(option);

    // Handle window resize for responsiveness
    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

7.2.3 Creating Your Own Custom Theme

Custom themes allow you to override colors, fonts, and other style properties globally across charts.

7.2.4 Step 1: Define a Theme Object

A theme object contains properties such as `color`, `backgroundColor`, `textStyle`, and component-specific styles.

Example of a simple custom theme:

```
const myTheme = {
  color: ['#5470C6', '#91CC75', '#FAC858', '#EE6666', '#73C0DE'],  // Custom color palette
  backgroundColor: '#f5f5f5',  // Chart background color
  textStyle: {
    fontFamily: 'Arial, sans-serif',
    color: '#333'
  },
  title: {
    textStyle: {
      fontWeight: 'bold',
      color: '#444'
    }
  },
  tooltip: {
    backgroundColor: 'rgba(50, 50, 50, 0.7)',
    textStyle: {
      color: '#fff'
    }
  },
  legend: {
    textStyle: {
      color: '#666'
```

```
    }
  }
};
```

7.2.5 Step 2: Register the Theme with ECharts

Register your theme under a unique name using `echarts.registerTheme` before initializing charts:

```
echarts.registerTheme('myCustomTheme', myTheme);
```

7.2.6 Step 3: Use Your Custom Theme

Apply your registered theme by name when creating chart instances:

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

chart.setOption({
  // Chart configuration here
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Custom ECharts Theme Demo</title>
  <style>
    body, html {
      margin: 0; padding: 0; height: 100vh;
      background-color: #f5f5f5;
      font-family: Arial, sans-serif;
    }
    #chart {
      width: 100%;
      height: 100%;
    }
  </style>
</head>
<body>
  <div id="chart"></div>

  <!-- ECharts library -->
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    // Step 1: Define a custom theme object
    const myTheme = {
      color: ['#5470C6', '#91CC75', '#FAC858', '#EE6666', '#73C0DE'],  // Custom palette
```

```
    backgroundColor: '#f5f5f5',  // Light gray background
    textStyle: {
      fontFamily: 'Arial, sans-serif',
      color: '#333'
    },
    title: {
      textStyle: {
        fontWeight: 'bold',
        color: '#444'
      }
    },
    tooltip: {
      backgroundColor: 'rgba(50, 50, 50, 0.7)',
      textStyle: {
        color: '#fff'
      }
    },
    legend: {
      textStyle: {
        color: '#666'
      }
    }
  };

  // Step 2: Register the custom theme
  echarts.registerTheme('myCustomTheme', myTheme);

  // Step 3: Initialize chart with the custom theme
  const chart = echarts.init(document.getElementById('chart'), 'myCustomTheme');

  // Sample chart option
  const option = {
    title: {
      text: 'Monthly Sales',
      left: 'center'
    },
    tooltip: {
      trigger: 'axis'
    },
    xAxis: {
      type: 'category',
      data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
    },
    yAxis: {
      type: 'value'
    },
    series: [{
      name: 'Sales',
      type: 'bar',
      data: [120, 200, 150, 80, 70, 110]
    }]
  };

  chart.setOption(option);

  // Resize on window size change
  window.addEventListener('resize', () => {
    chart.resize();
```

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

7.2.7 Tips for Custom Themes

- Define a consistent color palette to unify your charts' look.
- Customize component styles like `title`, `legend`, `tooltip`, and `axis` for full control.
- Use the ECharts Theme Builder online tool for visual theme creation.
- Register multiple themes to support dynamic switching in your applications.

7.2.8 Summary

Action	Example Code
Apply built-in theme	`echarts.init(dom, 'dark')`
Define custom theme	`const myTheme = { color: [...], ... }`
Register custom theme	`echarts.registerTheme('myTheme', myTheme)`
Use custom theme	`echarts.init(dom, 'myTheme')`

By leveraging built-in themes or creating your own, you can ensure your ECharts visualizations perfectly fit the visual identity and user experience of your projects.

Ready to explore how to implement dark mode or improve accessibility next?

7.3 Dark Mode and Accessibility Considerations

Designing charts that are both visually appealing and accessible is crucial for reaching a wide audience and ensuring usability across diverse viewing environments. This section covers best practices for accessibility—including contrast, readability, and color choice—and how to implement dark mode support in your ECharts visualizations.

7.3.1 Maintain Sufficient Contrast

- Ensure text, labels, and important chart elements stand out clearly against backgrounds.

- Follow WCAG guidelines recommending a contrast ratio of at least 4.5:1 for normal text.
- Avoid low-contrast color combinations that can hinder readability, especially for users with vision impairments.

7.3.2 Use Legible Labels and Fonts

- Choose readable font sizes and weights for axis labels, legends, and tooltips.
- Avoid overly decorative or condensed fonts.
- Provide clear spacing around text to avoid clutter.

7.3.3 Choose Accessible Color Palettes

- Use color palettes that are friendly to colorblind users; avoid relying solely on color to convey information.
- Utilize tools like ColorBrewer or Adobe Color to select palettes optimized for accessibility.
- Include shapes, patterns, or labels in addition to color for differentiation.

7.3.4 Detecting and Supporting Dark Mode

Dark mode reduces eye strain in low-light environments by inverting typical light backgrounds to dark tones. Supporting dark mode in your charts enhances user experience on devices and platforms that offer this feature.

7.3.5 Detecting Dark Mode with CSS Media Query

You can detect if a user prefers dark mode using JavaScript and the `prefers-color-scheme` media query:

```
const isDarkMode = window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches;
```

Use this boolean to load appropriate chart themes or styles.

7.3.6 Toggling Dark Mode in ECharts

1. **Use built-in themes:** ECharts includes a `dark` theme designed for dark backgrounds and high contrast.

```
const theme = isDarkMode ? 'dark' : 'light';
const chart = echarts.init(document.getElementById('chart'), theme);
```

2. **Dynamic theme switching:** Allow users to toggle between modes and update the chart accordingly.

```
function switchTheme(isDark) {
  const themeName = isDark ? 'dark' : 'light';
  chart.dispose();
  const newChart = echarts.init(document.getElementById('chart'), themeName);
  newChart.setOption(option);
  return newChart;
}
```

3. **Custom dark mode styling:** When creating your own themes, adapt background colors, axis lines, text colors, and tooltip styles to suit dark environments.

7.3.7 Example: Accessible Color Palette in Dark Mode

```
const darkTheme = {
  backgroundColor: '#121212',
  color: ['#90caf9', '#f48fb1', '#ce93d8', '#81c784', '#ffb74d'],
  textStyle: { color: '#e0e0e0' },
  axisLine: { lineStyle: { color: '#888' } },
  tooltip: {
    backgroundColor: 'rgba(50, 50, 50, 0.8)',
    textStyle: { color: '#fff' }
  }
};
echarts.registerTheme('customDark', darkTheme);
```

7.3.8 Summary

Accessibility Aspect	Recommendations
Contrast	Maintain high contrast ratios for readability
Fonts and Labels	Use clear, legible fonts with appropriate size
Color Palettes	Select colorblind-safe and accessible palettes
Dark Mode Detection	Use CSS media queries to detect user preferences
Theme Adaptation	Use built-in `dark` theme or custom dark themes
User Control	Allow users to toggle between light and dark modes

By thoughtfully combining accessibility best practices with dark mode support, your ECharts visualizations will be inclusive, visually comfortable, and adaptable to diverse user needs and environments.

Chapter 8.

Data Management

1. Loading and Updating Data Dynamically
2. Working with Large Datasets
3. Data Transformation and Filtering Techniques

8 Data Management

8.1 Loading and Updating Data Dynamically

One of the powerful features of ECharts is its ability to update visualizations dynamically as new data becomes available. Whether you're loading data from external APIs, local files, or real-time streams, ECharts makes it straightforward to fetch, process, and update charts efficiently.

8.1.1 Loading External Data Using `fetch`

Modern JavaScript provides the `fetch` API for asynchronous data retrieval from servers or local files. Here's how you can use it to load JSON data and update your chart.

8.1.2 Example: Fetching Data from an API and Updating a Chart

```
<div id="chart" style="width: 600px; height: 400px;"></div>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
  // Initialize chart
  const chart = echarts.init(document.getElementById('chart'));

  // Initial empty option
  const option = {
    title: { text: 'Dynamic Data Loading' },
    xAxis: { type: 'category', data: [] },
    yAxis: { type: 'value' },
    series: [{
      name: 'Sales',
      type: 'bar',
      data: []
    }]
  };

  chart.setOption(option);

  // Function to fetch data and update chart
  async function loadData() {
    try {
      const response = await fetch('https://api.example.com/sales-data');
      const jsonData = await response.json();

      // Example jsonData format:
      // { categories: ['Jan', 'Feb', 'Mar'], values: [120, 200, 150] }

      chart.setOption({
```

```
        xAxis: { data: jsonData.categories },
        series: [{ data: jsonData.values }]
      }, false);  // `notMerge: false` merges new data with existing option

    } catch (error) {
      console.error('Failed to load data:', error);
    }
  }

  // Load data initially
  loadData();

  // Optionally refresh data every 30 seconds
  setInterval(loadData, 30000);
</script>
```

8.1.3 Explanation

- **Initialization:** We create a basic bar chart with empty data arrays.
- **Fetching Data:** The `loadData` function uses `fetch` to retrieve JSON data from an API endpoint.
- **Updating Chart:** The `setOption` method is called with `notMerge: false` (default), meaning new data is merged with the current option, efficiently updating only what's changed.
- **Automatic Refresh:** Using `setInterval`, the chart refreshes with new data every 30 seconds.

8.1.4 Notes on `setOption`

- **`notMerge` parameter:** Controls whether the new option replaces the old one (`true`) or merges with it (`false`).
- Setting `notMerge: false` ensures that only updated parts are changed, maintaining other configurations like axes or legends.
- This helps achieve smooth transitions and avoids complete chart re-initialization.

8.1.5 Loading Local Data Files

You can also load data from local JSON or CSV files similarly:

```
fetch('data/local-sales.json')
  .then(res => res.json())
  .then(data => {
```

```
    chart.setOption({
      xAxis: { data: data.months },
      series: [{ data: data.sales }]
    });
  });
```

Ensure the file is served from a web server or local dev server to avoid CORS or file protocol restrictions.

8.1.6 Summary

Step	Description
Initialize Chart	Create chart instance with empty or placeholder data
Fetch Data	Use `fetch` API to asynchronously get external data
Update Chart	Call `setOption` with new data and `notMerge: false`
Refresh Dynamically	Use intervals or event triggers to reload and update

Dynamic data loading empowers you to build dashboards and visualizations that stay current and interactive, improving user engagement and insight delivery.

8.2 Working with Large Datasets

Handling large datasets efficiently is crucial for maintaining smooth and responsive visualizations in ECharts. When your data grows into thousands or even millions of points, default rendering and interactions may slow down significantly. This section covers key optimization techniques and best practices to keep your charts performant.

8.2.1 Challenges with Large Datasets

- **Rendering Overhead:** Drawing thousands of elements (points, bars, lines) can overload the browser's rendering engine.
- **Interaction Lag:** Hover effects, tooltips, and animations may become sluggish.
- **Memory Usage:** Storing and processing large data arrays consume more memory and CPU.

8.2.2 Enable `large` Mode for Line and Scatter Series

ECharts offers a special `large` mode optimized for datasets with tens of thousands of points or more. It switches from SVG-like rendering to a more efficient canvas drawing approach.

```
series: [{
  type: 'line',
  large: true,       // Enable large mode
  data: largeDataArray
}]
```

Note: Some interactive features like emphasis or symbol display may be disabled in large mode for performance.

8.2.3 Use Sampling to Reduce Data Points

Sampling reduces the number of data points by selecting a subset representative of the overall trend. This can be done before feeding data into ECharts or via ECharts' built-in sampling option:

```
series: [{
  type: 'line',
  sampling: 'lttb',  // Largest Triangle Three Buckets sampling method
  data: hugeDataArray
}]
```

Supported Sampling Methods:

- `average` — averages points in each bucket
- `max` — takes the max value in each bucket
- `min` — takes the min value in each bucket
- `lttb` — Largest Triangle Three Buckets, preserves shape better

8.2.4 Disable or Limit Animations

Animations can strain the browser with large datasets. Disable or reduce animation duration to improve responsiveness:

```
option = {
  animation: false,
  // or
  animationDuration: 100,
  animationEasing: 'linear',
  // ...
};
```

8.2.5 Simplify Visual Elements

- Remove or simplify effects like shadows, gradients, and complex symbol shapes.
- Limit the number of labels, tooltips, or hover highlights.

8.2.6 Pagination or Data Windowing

For very large data, consider:

- Displaying a subset or "window" of data with dataZoom controls.
- Paginating data in dashboards and loading only visible portions.

8.2.7 Use Aggregation or Alternative Visualizations

If raw data is too dense:

- Aggregate data into bins (e.g., hourly, daily).
- Use heatmaps or density charts to visualize concentration.
- Consider summary charts or statistical plots.

8.2.8 Summary Table

Technique	Description	Usage Scenario
`large: true`	Optimized rendering for very large data	$>$ 10,000 points in line/scatter
`sampling`	Reduce points while preserving shape	When full detail is not needed
Disable animations	Reduce browser rendering load	Smooth interactions needed
Simplify visuals	Remove complex styling	Improve rendering speed
Data windowing	Show data in chunks	Very large datasets
Aggregation	Summarize data into bins	High-density or noisy data

8.2.9 Example: Large Mode with Sampling

```
option = {
  xAxis: { type: 'category' },
```

```
  yAxis: { type: 'value' },
  series: [{
    type: 'line',
    data: largeDataArray,
    large: true,
    sampling: 'lttb'
  }],
  animation: false
};
```

By applying these techniques, you can create high-performance charts that handle large datasets gracefully without sacrificing interactivity or visual quality.

8.3 Data Transformation and Filtering Techniques

Before visualizing data with ECharts, it's often necessary to preprocess, transform, or filter your raw datasets to fit the chart's requirements or to highlight relevant information. This section explains common JavaScript techniques for manipulating data arrays and shows how to apply dynamic filtering to update charts on the fly.

8.3.1 Why Transform and Filter Data?

- **Shape data** to match chart structure (e.g., extract specific fields, aggregate values).
- **Clean data** by removing invalid or incomplete entries.
- **Filter data** to focus on subsets, such as time ranges or categories.
- **Enhance performance** by reducing data size.

8.3.2 Common Data Transformation Methods in JavaScript

Assuming your data is an array of objects, JavaScript's array methods make it easy to manipulate:

```
const data = [
  { month: 'Jan', sales: 120, region: 'North' },
  { month: 'Feb', sales: 200, region: 'South' },
  { month: 'Mar', sales: 150, region: 'North' },
  // more data...
];
```

8.3.3 `map()` Reshape or extract fields

Use `map` to transform each item into the format required by ECharts:

```
// Extract sales values for chart data
const salesData = data.map(item => item.sales);

// Extract months for xAxis categories
const months = data.map(item => item.month);
```

8.3.4 `filter()` Select subsets

Filter data by conditions, e.g., only 'North' region:

```
const northRegionData = data.filter(item => item.region === 'North');
```

8.3.5 `reduce()` Aggregate or summarize

Sum sales across all months:

```
const totalSales = data.reduce((sum, item) => sum + item.sales, 0);
```

Or aggregate sales by region:

```
const salesByRegion = data.reduce((acc, item) => {
  acc[item.region] = (acc[item.region] || 0) + item.sales;
  return acc;
}, {});
```

8.3.6 Dynamic Data Filtering with ECharts

You can enable interactive filtering by updating the chart's data based on user input, such as dropdowns or buttons.

8.3.7 Example: Filter Chart by Region

```
<select id="regionSelect">
  <option value="All">All</option>
  <option value="North">North</option>
  <option value="South">South</option>
</select>
```

```
<div id="chart" style="width: 600px; height: 400px;"></div>

<script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
<script>
  const data = [
    { month: 'Jan', sales: 120, region: 'North' },
    { month: 'Feb', sales: 200, region: 'South' },
    { month: 'Mar', sales: 150, region: 'North' },
    { month: 'Apr', sales: 80, region: 'South' }
  ];

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

  function updateChart(region) {
    let filteredData = data;
    if (region !== 'All') {
      filteredData = data.filter(item => item.region === region);
    }

    const option = {
      xAxis: {
        type: 'category',
        data: filteredData.map(item => item.month)
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
        data: filteredData.map(item => item.sales)
      }]
    };

    chart.setOption(option);
  }

  // Initial render with all data
  updateChart('All');

  // Event listener for region select dropdown
  document.getElementById('regionSelect').addEventListener('change', e => {
    updateChart(e.target.value);
  });
</script>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Dynamic Data Filtering Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
```

```
    #regionSelect {
      margin-bottom: 15px;
      font-size: 16px;
      padding: 5px;
    }
    #chart {
      width: 600px;
      height: 400px;
    }
  </style>
</head>
<body>
  <label for="regionSelect">Select Region: </label>
  <select id="regionSelect">
    <option value="All">All</option>
    <option value="North">North</option>
    <option value="South">South</option>
  </select>

  <div id="chart"></div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    // Sample data with region info
    const data = [
      { month: 'Jan', sales: 120, region: 'North' },
      { month: 'Feb', sales: 200, region: 'South' },
      { month: 'Mar', sales: 150, region: 'North' },
      { month: 'Apr', sales: 80,  region: 'South' }
    ];

    // Initialize chart instance
    const chart = echarts.init(document.getElementById('chart'));

    // Function to update chart based on selected region
    function updateChart(region) {
      let filteredData = data;
      if (region !== 'All') {
        filteredData = data.filter(item => item.region === region);
      }

      const option = {
        title: {
          text: `Sales Data (${region})`,
          left: 'center'
        },
        tooltip: {
          trigger: 'axis',
          axisPointer: { type: 'shadow' }
        },
        xAxis: {
          type: 'category',
          data: filteredData.map(item => item.month),
          axisTick: { alignWithLabel: true }
        },
        yAxis: {
          type: 'value'
        },
```

```
        series: [{
          name: 'Sales',
          type: 'bar',
          data: filteredData.map(item => item.sales),
          itemStyle: {
            color: '#5470C6'
          }
        }],
        animationDuration: 800
      };

      chart.setOption(option);
    }

    // Initial chart render (show all)
    updateChart('All');

    // Update chart when dropdown changes
    document.getElementById('regionSelect').addEventListener('change', e => {
      updateChart(e.target.value);
    });

    // Resize chart on window resize
    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

8.3.8 Summary

Method	Purpose	Example
`map()`	Transform or extract specific fields	`data.map(item => item.sales)`
`filter()`	Select subset of data based on condition	`data.filter(item => item.region === 'North')`
`reduce()`	Aggregate or summarize data	Summing sales by region
Dynamic Filtering	Update chart data based on user input	Filtering chart by selected category

Transforming and filtering data efficiently before passing it to ECharts ensures your charts remain relevant, focused, and performant. These JavaScript techniques, combined with ECharts' flexible API, allow you to build interactive and dynamic visualizations tailored to your data.

Chapter 9.

Customization and Extensibility

1. Creating Custom Series and Chart Types
2. Writing and Using Plugins
3. Advanced Styling with Rich Text and SVG Paths

9 Customization and Extensibility

9.1 Creating Custom Series and Chart Types

ECharts is designed to be highly extensible. While it offers a wide range of built-in chart types, sometimes your visualization needs may go beyond standard bar, line, or pie charts. For such cases, ECharts provides powerful APIs like `echarts.extendChartView()` and `echarts.registerSeriesModel()` that allow you to create **custom series** and completely new chart types tailored to your specific requirements.

9.1.1 When to Use Custom Series

Custom series are ideal when you want to visualize data in unique ways not covered by the built-in charts. Examples include:

- Spiral plots or radial layouts with unusual shapes
- Hexagonal binning (hexmaps)
- Specialized animations or non-rectangular coordinate systems
- Novel visual metaphors or artistic data presentations

Creating a custom series lets you control rendering at a low level, defining exactly how each data point appears and behaves.

9.1.2 Key APIs for Custom Series

`echarts.registerSeriesModel()`

This method registers the data model of your custom series. It defines the data schema, default options, and how data should be parsed or formatted.

`echarts.extendChartView()`

This creates the rendering logic (the "view") for your custom series. You implement how each element is drawn using ECharts' rendering layer, which supports Canvas and SVG under the hood.

9.1.3 Example: Creating a Simple Spiral Series

Here is a minimal example illustrating a custom series that plots points along a spiral.

9.1.4 Step 1: Register the Series Model

```
echarts.registerSeriesModel({
  type: 'customSpiral',
  getInitialData(option, ecModel) {
    // Create a list data structure for coordinates
    return new echarts.List(['value'], this);
  },
  defaultOption: {
    coordinateSystem: null,
    spiralTurns: 3,
    pointSize: 10,
    itemStyle: {
      color: '#5470C6'
    }
  }
});
```

9.1.5 Step 2: Extend the Chart View

```
echarts.extendChartView({
  type: 'customSpiral',
  render(seriesModel, ecModel, api) {
    const group = new echarts.graphic.Group();
    const data = seriesModel.getData();
    const spiralTurns = seriesModel.get('spiralTurns');
    const pointSize = seriesModel.get('pointSize');
    const itemStyle = seriesModel.getModel('itemStyle').getItemStyle();

    // Number of points to draw
    const count = 100;

    for (let i = 0; i < count; i++) {
      const angle = 2 * Math.PI * spiralTurns * (i / count);
      const radius = 5 * i;
      const x = api.getWidth() / 2 + radius * Math.cos(angle);
      const y = api.getHeight() / 2 + radius * Math.sin(angle);

      const circle = new echarts.graphic.Circle({
        shape: { cx: x, cy: y, r: pointSize },
        style: itemStyle
      });

      group.add(circle);
    }

    // Replace previous group with new rendering group
    this.group.removeAll();
    this.group.add(group);
  }
});
```

9.1.6 Step 3: Use the Custom Series in an Option

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

const option = {
  series: [{
    type: 'customSpiral',
    spiralTurns: 5,
    pointSize: 8,
    itemStyle: { color: 'tomato' }
  }]
};

chart.setOption(option);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Custom Spiral Series Example</title>
  <style>
    #main {
      width: 700px;
      height: 500px;
      margin: 20px auto;
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <div id="main"></div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    // Step 1: Register the custom series model
    echarts.registerSeriesModel({
      type: 'customSpiral',
      getInitialData(option, ecModel) {
        return new echarts.List(['value'], this);
      },
      defaultOption: {
        coordinateSystem: null,
        spiralTurns: 3,
        pointSize: 10,
        itemStyle: {
          color: '#5470C6'
        }
      }
    });

    // Step 2: Extend the chart view
    echarts.extendChartView({
      type: 'customSpiral',
      render(seriesModel, ecModel, api) {
        const group = new echarts.graphic.Group();
```

```
        const spiralTurns = seriesModel.get('spiralTurns');
        const pointSize = seriesModel.get('pointSize');
        const itemStyle = seriesModel.getModel('itemStyle').getItemStyle();

        const count = 100;
        for (let i = 0; i < count; i++) {
          const angle = 2 * Math.PI * spiralTurns * (i / count);
          const radius = 5 * i;
          const x = api.getWidth() / 2 + radius * Math.cos(angle);
          const y = api.getHeight() / 2 + radius * Math.sin(angle);

          const circle = new echarts.graphic.Circle({
            shape: { cx: x, cy: y, r: pointSize },
            style: itemStyle
          });

          group.add(circle);
        }

        this.group.removeAll();
        this.group.add(group);
      }
    });

    // Step 3: Initialize and set option
    const chart = echarts.init(document.getElementById('main'));

    const option = {
      series: [{
        type: 'customSpiral',
        spiralTurns: 5,
        pointSize: 8,
        itemStyle: { color: 'tomato' }
      }]
    };

    chart.setOption(option);

    // Resize chart on window resize
    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

9.1.7 How It Works

- The **series model** defines the custom option parameters and data structure.
- The **chart view** is responsible for drawing the graphic elements directly via ECharts' graphic utilities (`Circle`, `Group`, etc.).
- The example plots 100 circles arranged along a spiral, centered in the chart container.

- Parameters like `spiralTurns` and `pointSize` are customizable through the series options.

9.1.8 Summary

API	Purpose
`echarts.registerSeriesModel`	Define data model, default options
`echarts.extendChartView`	Implement drawing and animation logic

Creating custom series unlocks full control over rendering and interactivity, enabling truly unique visualizations tailored to your data and design goals.

9.2 Writing and Using Plugins

ECharts has a flexible plugin architecture that allows developers to extend its core functionality beyond built-in features. Plugins can add new chart types, shapes, interaction modes, or UI components, helping tailor ECharts to specific use cases or design needs.

9.2.1 Using Community Plugins

The ECharts ecosystem includes many community-built plugins that you can easily incorporate into your projects. For example, `echarts-liquidfill` creates beautiful liquid fill charts — perfect for showing progress or capacity in a fluid style.

9.2.2 How to use a community plugin:

1. **Install or include the plugin** (via NPM or CDN):

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

or

```
npm install echarts-liquidfill
```

2. **Import and register (for module bundlers):**

```
import * as echarts from 'echarts';
import 'echarts-liquidfill';
```

3. **Use the new chart type in your options:**

```
const option = {
  series: [{
    type: 'liquidFill',
    data: [0.6], // 60% fill
    radius: '80%',
    outline: { show: false },
    backgroundStyle: { color: '#e0f7fa' }
  }]
};

const chart = echarts.init(document.getElementById('main'));
chart.setOption(option);
```

Community plugins are a great way to add complex visuals quickly without reinventing the wheel.

9.2.3 Writing Your Own Basic Plugin

You can also create custom plugins to add specific features such as:

- Custom shapes or decorations
- Watermarks or logos
- Specialized interaction handlers
- Additional rendering layers

9.2.4 Plugin Structure

A plugin is essentially a JavaScript module that extends ECharts by registering new components, shapes, or behaviors.

9.2.5 Example: Adding a Watermark Plugin

Let's create a simple plugin that adds a watermark text overlay on the chart canvas.

Step 1: Define the Plugin

```
function watermarkPlugin(ec) {
  ec.registerCoordinateSystem('watermark', {
    // No coordinate system needed here, so this can be minimal
```

```
  });

  ec.extendComponentView({
    type: 'watermark',
    render: function (watermarkModel, ecModel, api) {
      const ctx = api.getZr().painter.getLayer(0).ctx; // Canvas context
      const width = api.getWidth();
      const height = api.getHeight();

      ctx.save();
      ctx.font = 'bold 40px Arial';
      ctx.fillStyle = 'rgba(0, 0, 0, 0.1)';
      ctx.textAlign = 'center';
      ctx.textBaseline = 'middle';
      ctx.translate(width / 2, height / 2);
      ctx.rotate(-Math.PI / 6); // Rotate watermark
      ctx.fillText('SAMPLE WATERMARK', 0, 0);
      ctx.restore();
    }
  });

  ec.registerComponentModel({
    type: 'watermark',
    defaultOption: {}
  });
}
```

Step 2: Register the Plugin

```
import * as echarts from 'echarts';

watermarkPlugin(echarts);
```

Step 3: Enable the Watermark in Chart Options

```
const option = {
  // Your chart options here...

  // Add watermark component
  watermark: {}
};

const chart = echarts.init(document.getElementById('main'));
chart.setOption(option);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Watermark Plugin Example</title>
  <style>
    #main {
      width: 700px;
      height: 500px;
```

```
      margin: 20px auto;
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <div id="main"></div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    // Step 1: Define the Watermark Plugin
    function watermarkPlugin(ec) {
      // Register a minimal component model for watermark
      ec.registerComponentModel({
        type: 'watermark',
        defaultOption: {}
      });

      // Extend the component view to draw watermark text
      ec.extendComponentView({
        type: 'watermark',
        render: function (watermarkModel, ecModel, api) {
          const ctx = api.getZr().painter.getLayer(0).ctx; // Canvas context
          const width = api.getWidth();
          const height = api.getHeight();

          ctx.save();
          ctx.font = 'bold 40px Arial';
          ctx.fillStyle = 'rgba(0, 0, 0, 0.1)';
          ctx.textAlign = 'center';
          ctx.textBaseline = 'middle';
          ctx.translate(width / 2, height / 2);
          ctx.rotate(-Math.PI / 6);
          ctx.fillText('SAMPLE WATERMARK', 0, 0);
          ctx.restore();
        }
      });
    }

    // Step 2: Register the plugin with ECharts
    watermarkPlugin(echarts);

    // Step 3: Use the watermark component in your chart option
    const option = {
      title: {
        text: 'Bar Chart with Watermark',
        left: 'center'
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['A', 'B', 'C', 'D', 'E']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
```

```
        data: [5, 20, 36, 10, 10],
        itemStyle: {
          color: '#5470C6'
        }
      }],
      watermark: {}  // Enable the watermark component
    };

    const chart = echarts.init(document.getElementById('main'));
    chart.setOption(option);

    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

9.2.6 How This Works

- We register a minimal component model and view named `watermark`.
- The component view's `render` method directly draws on the canvas layer.
- The watermark appears as semi-transparent text rotated across the chart.
- This approach can be adapted for logos, custom decorations, or background effects.

9.2.7 Summary

Plugin Use Case	Description
Community Plugins	Extend ECharts with ready-made features
Custom Plugins	Add tailored functionality (shapes, overlays)
Core API Used	`registerComponentModel`, `extendComponentView`

Writing plugins unlocks deeper customization, empowering you to enhance ECharts exactly how your project demands.

9.3 Advanced Styling with Rich Text and SVG Paths

ECharts provides powerful styling capabilities that go beyond basic colors and fonts. Two key techniques for advanced visual customization are **rich text formatting** for labels and embedding **SVG paths** for custom icons or marks. This section guides you through both

approaches to help you create visually compelling, unique charts.

9.3.1 Rich Text Formatting for Labels

The `rich` property allows you to apply multiple styles within a single label, supporting multi-line text, different colors, font sizes, weights, and more. This is particularly useful for emphasizing parts of the text or showing complex information clearly.

9.3.2 Example: Multi-line and Colored Axis Labels

```
const option = {
  xAxis: {
    type: 'category',
    data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
    axisLabel: {
      formatter: function (value) {
        return `{day|${value}}\n{desc|Sales}`;
      },
      rich: {
        day: {
          fontSize: 14,
          fontWeight: 'bold',
          color: '#2f4554',
          lineHeight: 20
        },
        desc: {
          fontSize: 10,
          color: '#aaa',
          fontStyle: 'italic'
        }
      }
    }
  },
  yAxis: {
    type: 'value'
  },
  series: [{
    type: 'bar',
    data: [120, 200, 150, 80, 70]
  }]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Multi-line and Colored Axis Labels</title>
```

```
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 20px auto;
    }
  </style>
</head>
<body>
  <div id="main"></div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    const option = {
      xAxis: {
        type: 'category',
        data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
        axisLabel: {
          formatter: function (value) {
            return `{day|${value}}\n{desc|Sales}`;
          },
          rich: {
            day: {
              fontSize: 14,
              fontWeight: 'bold',
              color: '#2f4554',
              lineHeight: 20
            },
            desc: {
              fontSize: 10,
              color: '#aaa',
              fontStyle: 'italic'
            }
          }
        }
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
        data: [120, 200, 150, 80, 70]
      }]
    };

    const chart = echarts.init(document.getElementById('main'));
    chart.setOption(option);

    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

Explanation:

- The `formatter` function returns a string with two parts, each wrapped in

`{styleName|text}`.

- The `rich` object defines styles named `day` and `desc`.
- The label shows the day on top in bold and a smaller description beneath in italic.

9.3.3 Embedding SVG Paths for Custom Icons

ECharts supports using SVG path data as icons in elements like `markPoint`, `legend`, or custom series. This enables highly customizable graphics, such as unique shapes or branded symbols.

9.3.4 Example: Using SVG Path in a MarkPoint

```
const option = {
  xAxis: {
    type: 'category',
    data: ['A', 'B', 'C', 'D', 'E']
  },
  yAxis: {
    type: 'value'
  },
  series: [{
    type: 'line',
    data: [10, 52, 200, 334, 390],
    markPoint: {
      symbol: 'path://M512 0C229.234 0 0 229.234 0 512s229.234 512 512 512 512-229.234 512-512S794.766 0
      symbolSize: 40,
      data: [
        { type: 'max', name: 'Max' }
      ]
    }
  }]
};
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>SVG Path as MarkPoint Icon in ECharts</title>
  <style>
    #main {
      width: 700px;
      height: 450px;
      margin: 20px auto;
    }
  </style>
</head>
```

```
<body>
  <div id="main"></div>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    const option = {
      xAxis: {
        type: 'category',
        data: ['A', 'B', 'C', 'D', 'E']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'line',
        data: [10, 52, 200, 334, 390],
        markPoint: {
          symbol: 'path://M512 0C229.234 0 0 229.234 0 512s229.234 512 512 512 512-229.234 512-512S794.
          symbolSize: 40,
          data: [
            { type: 'max', name: 'Max' }
          ]
        }
      }]
    };

    const chart = echarts.init(document.getElementById('main'));
    chart.setOption(option);

    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

Explanation:

- The `symbol` is set to a custom SVG path string (here, a circle outline path).
- `symbolSize` controls the icon's size.
- This icon marks the max data point on the line.

9.3.5 Combining Rich Text and SVG Paths

You can combine these techniques for ultimate styling flexibility — for example, create legends with SVG icons and richly formatted labels.

9.3.6 Summary

Feature	Usage Example	Notes
Rich Text (`rich`)	Multi-style axis or tooltip labels	Supports multi-line, fonts, colors
SVG Paths (`symbol`)	Custom shapes for marks or legends	Use `path://` prefix with SVG data

These tools empower you to elevate chart aesthetics and tailor visual language to your brand or data storytelling needs.

Chapter 10.

Exporting and Sharing

1. Exporting Charts as Images and PDFs
2. Sharing via Embeds and IFrames
3. Printing and Offline Usage

10 Exporting and Sharing

10.1 Exporting Charts as Images and PDFs

ECharts makes it easy to export your interactive charts as static images, such as PNG or JPEG, which can then be used for reports, presentations, or sharing. Additionally, by combining ECharts with libraries like jsPDF, you can generate PDF documents containing your charts directly from the browser.

10.1.1 Exporting Charts as Images

The key method for exporting images from an ECharts instance is:

```
echartsInstance.getDataURL({
  type: 'png',    // 'png' or 'jpeg'
  pixelRatio: 2,  // Controls resolution (higher = better quality)
  backgroundColor: '#fff' // Optional background color
});
```

This method returns a base64-encoded data URL representing the chart image.

10.1.2 Example: Export Chart as PNG and Trigger Download

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

// Your chart option here
const option = {
  // ... chart configuration ...
};

chart.setOption(option);

// Export function
function exportChartImage() {
  const url = chart.getDataURL({
    type: 'png',
    pixelRatio: 2,
    backgroundColor: '#fff'
  });

  // Create a temporary link to trigger download
  const a = document.createElement('a');
  a.href = url;
  a.download = 'echarts-export.png';
  document.body.appendChild(a);
  a.click();
  document.body.removeChild(a);
}
```

Add a button in your HTML to call `exportChartImage()` for a user-friendly way to save the chart image.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>ECharts Export as PNG Example</title>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 20px auto;
    }
    #exportBtn {
      display: block;
      margin: 10px auto;
      padding: 8px 16px;
      font-size: 16px;
    }
  </style>
</head>
<body>
  <div id="main"></div>
  <button id="exportBtn">Export Chart as PNG</button>

  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <script>
    const chart = echarts.init(document.getElementById('main'));

    const option = {
      title: {
        text: 'Sample Sales Data'
      },
      tooltip: {},
      xAxis: {
        type: 'category',
        data: ['Jan', 'Feb', 'Mar', 'Apr', 'May']
      },
      yAxis: {
        type: 'value'
      },
      series: [{
        type: 'bar',
        data: [120, 200, 150, 80, 70]
      }]
    };

    chart.setOption(option);

    document.getElementById('exportBtn').addEventListener('click', () => {
      const url = chart.getDataURL({
        type: 'png',
        pixelRatio: 2,
        backgroundColor: '#fff'
      });
```

```
      const a = document.createElement('a');
      a.href = url;
      a.download = 'echarts-export.png';
      document.body.appendChild(a);
      a.click();
      document.body.removeChild(a);
    });

    window.addEventListener('resize', () => {
      chart.resize();
    });
  </script>
</body>
</html>
```

10.1.3 Exporting Charts to PDF with jsPDF

jsPDF is a popular JavaScript library for generating PDF files in the browser. You can embed your exported chart image into a PDF page.

10.1.4 Example: Export Chart as PDF

```
<button id="exportPdfBtn">Export as PDF</button>
```

```
import jsPDF from 'jspdf';

document.getElementById('exportPdfBtn').addEventListener('click', () => {
  const pdf = new jsPDF('landscape');

  // Get chart image data URL
  const imgData = chart.getDataURL({
    type: 'png',
    pixelRatio: 3,
    backgroundColor: '#fff'
  });

  // Add image to PDF (x, y, width, height)
  pdf.addImage(imgData, 'PNG', 10, 10, 280, 150);

  // Save the PDF
  pdf.save('chart-report.pdf');
});
```

10.1.5 Tips for Best Results

- **Use higher `pixelRatio` values** for sharper images, especially for printing.
- **Set a white or suitable `backgroundColor`** to avoid transparent backgrounds that may appear strange in some contexts.
- **Adjust image size when embedding in PDFs** to fit the layout.
- For multi-page reports, you can add multiple charts or content using jsPDF's API.

10.1.6 Summary

Feature	Method / Library	Purpose
Export as PNG/JPEG	`echartsInstance.getDataURL()`	Generate base64 image URLs
Trigger download	Creating temporary `<a>` tags	Save images locally
Export as PDF	jsPDF	Embed chart images in PDFs

Exporting charts as images or PDFs allows you to create sharable, offline-ready visualizations easily, extending the usability of your ECharts projects.

10.2 Sharing via Embeds and IFrames

Sharing your ECharts visualizations as embeddable components makes it easy to showcase your interactive charts on blogs, documentation sites, or dashboards without duplicating code or setup. Two common approaches are **embedding via iframe** and **sharing live demos through services like CodePen**.

10.2.1 Using IFrames to Embed ECharts Charts

An **iframe** lets you embed a standalone HTML page containing an ECharts visualization into any other webpage. This approach encapsulates the chart independently, avoiding conflicts with the parent page's styles or scripts.

10.2.2 How to create an embeddable iframe:

1. **Create a standalone HTML file** that initializes your ECharts chart.

```
<!-- Save as chart.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>ECharts Embed</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
  <style>
    body, html { margin: 0; padding: 0; height: 100%; }
    #chart { width: 100%; height: 100vh; }
  </style>
</head>
<body>
  <div id="chart"></div>
  <script>
    const chart = echarts.init(document.getElementById('chart'));
    chart.setOption({
      title: { text: 'Embedded Chart Example' },
      tooltip: {},
      xAxis: { data: ['A', 'B', 'C', 'D'] },
      yAxis: {},
      series: [{ type: 'bar', data: [5, 20, 36, 10] }]
    });
  </script>
</body>
</html>
```

2. **Host this file** on your web server or a static hosting service (GitHub Pages, Netlify, etc.).

3. **Embed it in other pages** using an iframe tag:

```
<iframe
  src="https://yourdomain.com/chart.html"
  style="width: 600px; height: 400px; border: none;"
  title="ECharts Embed"
></iframe>
```

10.2.3 Sharing Live Examples via CodePen or Similar Platforms

For quick sharing and collaboration, online playgrounds like **CodePen**, **JSFiddle**, or **CodeSandbox** allow you to publish live ECharts demos.

10.2.4 Example: CodePen Embed

You can create a new pen with ECharts CDN loaded and your chart code, then embed it with an iframe:

```
<iframe height="400" style="width: 100%;" scrolling="no" title="ECharts Bar Chart" src="https://codepen
```

This iframe dynamically loads the interactive demo, letting users experiment directly.

10.2.5 Best Practices for Embedding

- **Responsive sizing:** Use percentage widths or CSS flexbox/grid to make embeds adapt to different screen sizes.
- **Security:** When embedding third-party URLs, ensure they use HTTPS and consider sandbox attributes on iframes.
- **Performance:** Keep embedded charts lightweight and avoid large datasets if embedding multiple charts on a page.
- **Cross-origin communication:** For advanced use cases, you can enable messaging between the parent page and iframe via `postMessage`.

10.2.6 Summary

Method	Description	Use Case
IFrame embedding	Embed standalone HTML page with ECharts	Blogs, docs, static sites
CodePen/Playground	Share live editable demos	Collaborative sharing, tutorials

Embedding charts as iframes or live demos provides a flexible way to distribute your visualizations across multiple platforms, improving reach and user engagement.

10.3 Printing and Offline Usage

Preparing your ECharts visualizations for printing or offline use requires special considerations to ensure clarity, quality, and smooth functionality even without internet connectivity. This section covers best practices to optimize charts for both scenarios.

10.3.1 Preparing Charts for Printing

Printing interactive charts often means converting them into static visuals suitable for paper or PDF output. Here are key tips:

10.3.2 Adjust Chart Size and DPI

- **Set explicit pixel dimensions** for your chart container to control printed size precisely.
- Use the `getDataURL` method with a **higher `pixelRatio`** (e.g., 3 or 4) to export high-resolution images ideal for print.

```
const highResImg = chart.getDataURL({
  type: 'png',
  pixelRatio: 3,
  backgroundColor: '#fff'
});
```

- Embed this image into your print layout or generate PDFs for crisp output.

10.3.3 Remove or Simplify Interactive Elements

- Disable tooltips, animations, and zooming for print views to avoid clutter or rendering issues.
- Use CSS media queries (`@media print`) to hide UI controls like buttons or legends that aren't needed on paper.

```
@media print {
  .echarts-tooltip, .zoom-controls {
    display: none !important;
  }
}
```

10.3.4 Use Static Image Snapshots

- Instead of printing the live chart, print a static image snapshot generated via `getDataURL()` for consistency.

10.3.5 Strategies for Offline Usage

Using ECharts in offline environments (e.g., kiosks, internal apps, or presentations) means bundling all necessary resources locally and avoiding dependencies on external CDNs.

10.3.6 Bundle All Assets Locally

- Download and include the ECharts library files (`echarts.min.js`) directly in your project.
- Reference scripts with local paths instead of CDN URLs:

```
<script src="libs/echarts.min.js"></script>
```

- Similarly, host any plugins or extensions locally.

10.3.7 Preload Data and Avoid Remote Fetching

- Include static or preprocessed datasets inside your project.
- Avoid relying on remote API calls or external JSON files that require internet access.

10.3.8 Use Build Tools for Bundling

- Use bundlers like **Webpack** or **Vite** to package ECharts and your application into a single deployable bundle.
- This ensures all dependencies are included and ready to run offline.

10.3.9 Summary

Aspect	Best Practice	Benefit
Printing	Use high-resolution images, remove UI	Clear, print-ready visuals
Offline Usage	Host all scripts and data locally	Fully functional without internet
UI Adjustments	Hide interactive elements in print mode	Cleaner printed documents

By following these guidelines, you ensure that your ECharts visualizations look great on paper and perform reliably in offline environments.

Chapter 11.

Performance Optimization

1. Optimizing for Large Data and High FPS
2. Lazy Loading and Data Chunking
3. Best Practices for Smooth Animations

11 Performance Optimization

11.1 Optimizing for Large Data and High FPS

When working with large datasets or aiming for smooth, high-frame-rate (FPS) animations, performance optimization becomes critical in ECharts. This section discusses key techniques to ensure your charts remain responsive and visually clear even under heavy data loads.

11.1.1 Enable `large` Mode for Series

The `large` mode is designed to accelerate rendering when dealing with tens of thousands of data points, especially in scatter, line, and bar charts.

- **How it works:** Internally, ECharts uses a simplified rendering pipeline optimized for large datasets by sacrificing some visual details.
- **Enable it in your series:**

```
series: [{
  type: 'line',
  large: true,
  data: largeDataArray
}]
```

11.1.2 Use `progressive` Rendering

`progressive` mode progressively renders data in chunks rather than all at once, improving initial load times and responsiveness.

- **Set `progressive` and optionally `progressiveThreshold`:**

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

11.1.3 Simplify Tooltips and Interactions

Complex tooltips or interactive elements can degrade performance when many points trigger events.

- Use **simple tooltip formats** and avoid heavy HTML or rich text.

- Disable unnecessary hover effects or animations on large datasets.

```
tooltip: {
  trigger: 'item',
  formatter: params => `X: ${params.data[0]}, Y: ${params.data[1]}`
}
```

11.1.4 Reduce Visual Complexity

- Avoid excessive shadows, gradients, or item styles.
- Use minimal `symbolSize` and turn off animations if needed (`animation: false`).
- Limit or avoid labels on large datasets.

11.1.5 Performance Comparison Example

Consider a scatter plot with 50,000 points:

```
// Without optimization
series: [{
  type: 'scatter',
  data: largeDataArray,
  animation: true,
  symbolSize: 10,
  tooltip: { formatter: '{c}' }
}]
```

- This may cause sluggish rendering and UI lag.

Now, apply optimizations:

```
series: [{
  type: 'scatter',
  data: largeDataArray,
  large: true,
  progressive: 5000,
  animation: false,
  symbolSize: 3,
  tooltip: { formatter: params => `(${params.data[0]}, ${params.data[1]})` }
}]
```

- Notice a significant improvement in render speed and responsiveness.

11.1.6 Summary of Optimization Options

Option	Purpose	Usage Example
`large`	Enable fast rendering for large data	`large: true`
`progressive`	Render data in chunks for smooth loading	`progressive: 5000`
`animation`	Disable animations to boost performance	`animation: false`
`tooltip.formatter`	Simplify tooltip content	Simple string or callback
Visual complexity	Reduce shadows, symbol sizes, labels	Minimal styling

By combining these techniques, you can maintain smooth, interactive ECharts visualizations even with very large datasets, providing your users with a seamless experience.

11.2 Lazy Loading and Data Chunking

When dealing with extremely large datasets or continuous data streams, loading and rendering all data at once can cause slowdowns and unresponsive UIs. **Lazy loading** and **data chunking** help mitigate these issues by loading data incrementally or in manageable pieces, improving performance and user experience.

11.2.1 Why Lazy Loading and Data Chunking Matter

- **Avoid UI freezing:** Loading large datasets synchronously blocks the main thread, causing lag.
- **Improve perceived performance:** Display initial data quickly, then progressively add more.
- **Enable smooth interaction:** Load only visible or relevant data (e.g., zoomed viewport).

11.2.2 Load Data on Demand (e.g., on Zoom or Scroll)

Use ECharts' `dataZoom` events to detect when users zoom or pan, then fetch or load the corresponding slice of data dynamically.

```
chart.on('datazoom', params => {
  const startIndex = Math.floor(params.start / 100 * totalData.length);
  const endIndex = Math.floor(params.end / 100 * totalData.length);
  const visibleData = totalData.slice(startIndex, endIndex);

  chart.setOption({
    series: [{ data: visibleData }]
  }, false);
```

```
});
```

11.2.3 Data Chunking Strategies

Instead of loading all data at once, break it into smaller chunks and load them asynchronously to keep the UI responsive.

11.2.4 Use `setTimeout` to Load Data in Chunks

```
const chunkSize = 1000;
let currentIndex = 0;

function loadNextChunk() {
  const chunk = totalData.slice(currentIndex, currentIndex + chunkSize);
  currentIndex += chunkSize;

  chart.setOption({
    series: [{
      data: (chart.getOption().series[0].data || []).concat(chunk)
    }]
  });

  if (currentIndex < totalData.length) {
    setTimeout(loadNextChunk, 50); // schedule next chunk after 50ms
  }
}

loadNextChunk();
```

11.2.5 Use `requestIdleCallback` for Background Loading (if supported)

```
function loadChunk() {
  if (currentIndex >= totalData.length) return;

  const chunk = totalData.slice(currentIndex, currentIndex + chunkSize);
  currentIndex += chunkSize;

  chart.setOption({
    series: [{
      data: (chart.getOption().series[0].data || []).concat(chunk)
    }]
  });

  requestIdleCallback(loadChunk);
}
```

```
requestIdleCallback(loadChunk);
```

This method lets the browser schedule loading during idle time, improving smoothness.

11.2.6 Benefits of Incremental Loading

- Maintains UI responsiveness during heavy data operations.
- Provides a better user experience by showing data progressively.
- Enables handling streaming or real-time data feeds effectively.

11.2.7 Summary

Technique	Description	Example Use Case
Lazy loading on zoom	Load visible data only on zoom events	Time series zooming
Data chunking with `setTimeout`	Load data in timed batches asynchronously	Large static datasets
Data chunking with `requestIdleCallback`	Background loading without blocking UI	Smooth incremental loading

By applying lazy loading and chunking, your ECharts visualizations can scale gracefully even with massive data volumes, ensuring a fluid and responsive interface.

11.3 Best Practices for Smooth Animations

Animations add polish and clarity to your ECharts visualizations by smoothly transitioning between data states. However, with large datasets or complex charts, animations can sometimes cause lag or dropped frames. This section covers how to tune animation settings for optimal smoothness and when to disable them for performance.

11.3.1 Key Animation Settings in ECharts

`animationThreshold`

- Controls the maximum number of graphic elements to animate.

- If the dataset exceeds this threshold, animations are automatically disabled to prevent sluggishness.

```
animationThreshold: 2000,  // Animate only if elements  2000
```

Adjusting this number helps balance visual appeal and responsiveness.

animationDuration

- Sets the length of animation in milliseconds.
- Shorter durations create snappier updates; longer durations provide smoother, slower transitions.

```
animationDuration: 500,  // Half-second animation
```

animationEasing

- Defines the easing function for animations.
- Common options include `'linear'`, `'cubicOut'`, `'quadraticInOut'`, and `'bounceOut'`.

```
animationEasing: 'cubicOut',
```

Choosing the right easing can enhance the natural feel of transitions.

11.3.2 Example: Smooth Transition on Data Update

```
chart.setOption({
  animationDuration: 800,
  animationEasing: 'cubicOut',
  series: [{
    data: updatedData
  }]
});
```

This configuration produces a smooth 800ms transition with a cubic easing curve, improving user experience when data changes.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ECharts Smooth Transition Example</title>
  <script src="https://cdn.jsdelivr.net/npm/echarts@5/dist/echarts.min.js"></script>
  <style>
    #main {
      width: 600px;
      height: 400px;
      margin: 2rem auto;
    }
```

```
    button {
      display: block;
      margin: 1rem auto;
    }
  </style>
</head>
<body>

<div id="main"></div>
<button onclick="updateData()">Update Data</button>

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

  // Initial data
  let currentData = [120, 200, 150, 80, 70, 110, 130];

  chart.setOption({
    title: {
      text: 'Smooth Transition on Data Update'
    },
    tooltip: {},
    xAxis: {
      type: 'category',
      data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    },
    yAxis: {
      type: 'value'
    },
    series: [{
      name: 'Sales',
      type: 'bar',
      data: currentData
    }]
  });

  function updateData() {
    // Generate new data
    const updatedData = currentData.map(val => Math.round(val * (0.5 + Math.random())));
    currentData = updatedData;

    chart.setOption({
      animationDuration: 800,
      animationEasing: 'cubicOut',
      series: [{
        data: updatedData
      }]
    });
  }
</script>

</body>
</html>
```

11.3.3 When to Disable Animations

Animations can be counterproductive when:

- Handling very large datasets (thousands of points or more).
- Needing real-time responsiveness or frequent updates.
- Running on low-power devices or constrained environments.

Disable animations explicitly:

```
animation: false,
animationDuration: 0,
```

11.3.4 Summary of Animation Tuning

Setting	Purpose	Recommendation
`animationThreshold`	Auto-disable animations on large data	Set according to dataset size
`animationDuration`	Control animation speed	300–800 ms for smooth yet quick updates
`animationEasing`	Adjust transition feel	Use `'cubicOut'` or `'quadraticInOut'` for natural motion
`animation`	Enable or disable animations	Disable for large or real-time data

By carefully tuning animation parameters or disabling them when needed, you can ensure your ECharts charts remain visually appealing **and** performant across all scenarios.