Using D3.js, the powerful JavaScript toolkit for creating cross-platform vector graphics, you can now combine performance with maximum compatibility to build a web-based visualization and present data in an interactive and convenient way. We’ll reach top-notch reusability and testability by combining D3 graphics with our favorite web application framework, AngularJS.

This book teaches the basics of vector graphics, D3, and AngularJS integration, and then dives into controlling, manipulating, and filtering data. You will learn about the testability of components and how to implement custom interactions, filters, and controllers: discover how to parse and map data in D3.js, and get a grasp on drawing D3.js built-in shapes and curves. After reading the last few chapters, you’ll be able to bring life to your visualizations with more features of D3.js such as interactions, animations, and transitions. You will finish your journey by implementing a parser for different server application logs and display them on a Google Analytics style interactive dashboard.

Who this book is written for
If you are a web developer with experience in AngularJS and want to implement interactive visualizations using D3.js, this book is for you. Knowledge of SVG or D3.js will give you an edge to get the most out of this book.

What you will learn from this book
- Design, implement, and integrate an interactive dashboard to visualize server logs in real time using D3 graphics
- Learn cross-platform vector graphics to implement a dashboard visualization
- Perform data-driven transformations on selected HTML and SVG nodes
- Map, group, and filter datasets and create scales and axes
- Modularize data visualization information into reusable components to seamlessly integrate them into an AngularJS application
- Load, parse, and preprocess external data and autoupdate the visualization
- Design various chart types such as scatter, line, bar, or area and extend built-in shapes
- Create custom animations and transitions for the visualization
- Implement interactions and controls for the visualization preserving two-way binding between D3 and AngularJS components
- Load, parse, and preprocess external data and autoupdate the visualization

Who this book is written for
If you are a web developer with experience in AngularJS and want to implement interactive visualizations using D3.js, this book is for you. Knowledge of SVG or D3.js will give you an edge to get the most out of this book.

What you will learn from this book
- Design, implement, and integrate an interactive dashboard to visualize server logs in real time using D3 graphics
- Learn cross-platform vector graphics to implement a dashboard visualization
- Perform data-driven transformations on selected HTML and SVG nodes
- Map, group, and filter datasets and create scales and axes
- Modularize data visualization information into reusable components to seamlessly integrate them into an AngularJS application
- Load, parse, and preprocess external data and autoupdate the visualization
- Design various chart types such as scatter, line, bar, or area and extend built-in shapes
- Create custom animations and transitions for the visualization
- Implement interactions and controls for the visualization preserving two-way binding between D3 and AngularJS components
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Magic of SVG, D3.js, and AngularJS'
- A synopsis of the book’s content
- More information on Data Visualization with D3 and AngularJS

About the Author

Christoph Körner is the CTO of GESIM, a Swiss startup company, where he is responsible for its simulation software and web interface that is built using AngularJS and D3. He studied electrical engineering and information technology at Vienna University of Technology, including a 1-year exchange program at the Swiss Federal Institute of Technology, Lausanne.

Christoph is a passionate, self-taught software developer, coworker, and web enthusiast with more than 7 years of experience in designing, developing, and implementing customer-oriented web-based IT solutions. He has always been enthusiastic and curious about new technologies and started to use AngularJS and D3 since the very first version. Christoph is not only fascinated by technology, but he also aims to push cutting-edge technologies to production services.
Data Visualization with D3 and AngularJS

The question of how I could better visualize my application server's log files has always hovered around in my mind. In this book, we'll materialize this idea. After reading the book, you'll be able to analyze the log files of your application server and display them as a time series chart on a web page—a real-time autoupdating chart. I picked my two favorite JavaScript libraries—AngularJS and D3.js—to develop such a visualization.

AngularJS is a web application framework that facilitates the development of modern web applications with concepts such as the MVC/MVVM pattern, two-way data binding, testable modules, and dependency injection. I especially like the idea that one can declare templates to extend the HTML language; this enables a whole new world of applications and the sharing of work between designers and frontend engineers.

D3.js is a visualization toolkit that facilitates the generation and manipulation of web-based vector graphics with full access to the underlying SVG standard. Moreover, animations and interactive visualizations change how users perceive web applications; D3.js offers everything you need to make a visualization interactive out of the box.

In this book, I will teach you how to create an interactive real-time visualization for the Web with AngularJS as the application framework and D3.js as the visualization toolkit. I will explain the basic concepts of D3.js, when and why we use it, and why it's so much better than other DOM-transforming libraries, with a focus on its data-driven approach. Also, I will show you best practices to manipulate datasets and preprocess data—an everyday job if you want to display data from different sources. Then, we will spend more time on seeing how a visualization library can be best integrated into an AngularJS application using modules, directives, and services to make the components testable and maintainable. Later, we will focus on developing the visualization.

First, we need to load real data in the application and parse the text data to a JavaScript array of objects. Then, we need to preprocess and group the data according to the way we want to display it. Now, we can generate a curve or shape in the chart that represents our data. Next, we want to animate this curve—to show a transition between the old and the new state—whenever we update the data of the chart. We also want to zoom and pan inside the charts and create filters in order to clamp the chart to a certain range, for example, a starting time and an ending time for the time series data. Finally, we want to put everything together on to a dashboard and feed it with real server logs of a web server.

Displaying data on the web is a common task for developers. This book walks you through an exciting project, explaining best practices and common pitfalls in this domain using the two most amazing JavaScript libraries: AngularJS and D3.js.
What This Book Covers

Chapter 1, *The Magic of SVG, D3.js, and AngularJS*, introduces you to the world of vector graphics on the web and points out the advantages of D3.js.


Chapter 3, *Manipulating Data*, walks you through the important task of manipulating and preprocessing datasets in JavaScript and D3.js.

Chapter 4, *Building a Chart Directive*, lets you implement a first scatter chart directive in order to integrate the visualization into a simple AngularJS application with a focus on modularization and testability.

Chapter 5, *Loading and Parsing Data*, explains how to load files and parse real data using AngularJS and D3.js. It also explains how to aggregate information for the visualization.

Chapter 6, *Drawing Curves and Shapes*, helps you understand lines, curves, and shapes in SVG and D3.js in order to create different chart types, such as line charts, area charts, and bar charts.

Chapter 7, *Controlling Transitions and Animations*, shows how animations can be used to better visualize changes in the dataset and the visualization. You will learn about animations in D3.js and how to create plausible transitions to update chart directives.

Chapter 8, *Bringing the Chart to Life with Interactions*, explains how to add interactions to the chart directive, which is a big advantage of web-based visualizations. You will also see how to combine and encapsulate the interactions of the D3 chart and the AngularJS application.

Chapter 9, *Building a Real-time Visualization to Monitor Server Logs*, walks you through an exciting example where you will use all the acquired knowledge of the book to build a visualization dashboard to analyze server logs.
In this book, we will develop an application that combines both frameworks D3.js and AngularJS into a stunning interactive visualization, an interactive dashboard to visualize server logs in real time. First, I will discuss the outline and the purpose of this application. Then, I will explain how to modularize the visualization so that all components can be seamlessly embedded into one single application.

In the second section, we will talk about the terminology and definitions that are used in this book to introduce you to this topic or to refresh your knowledge of some basics in computer graphics. It will help you to understand why we are using vector graphics—especially SVG for the Web—for the visualization that we will develop in this book (and not, for example, Canvas or WebGL).

The next part of the chapter will give an introduction to D3.js and its data-driven approach. It will help you to understand the similarities and fundamental differences of D3.js and other DOM transforming libraries such as jQuery. Then, we will see an introduction to AngularJS and discuss its strengths for modern application development as well as the integration of D3.js into an AngularJS application.

The last section of this chapter will help you to understand why we prefer D3.js over other visualization libraries. It will give a brief outline and comparison of web visualization tools emphasizing on libraries to create and manipulate vector graphics and the evolution of web standards. We will discuss the main ideas and advantages of using D3.js in web applications and the differences between it and other tools for web visualizations (such as Raphaël or Three.js).
In this chapter, you will learn:

- How to distinguish vector and pixel graphics and to know their application areas
- When and why to use vector graphics, especially SVG
- Why D3.js is superior to other common graphical toolkits
- When and why to use D3.js for creating graphical content
- The advantages of integrating D3.js into an AngularJS application
- How to structure and encapsulate a visualization library written in D3.js to use with an AngularJS application

**Building a real-time dashboard to visualize server logs**

Did you ever wonder how many users visit your web application or how many exceptions your server application raised in the last month? The solution to various similar problems is to simply analyze and visualize the log or access files of the particular application.

In this book, we will develop a real-time dashboard to visualize application logs and the server's system status in the browser. Thus, we will build a custom Google Analytics-like visualization dashboard for our own server (see the following figure). Additionally, we want to add custom animation and user interactions—such as selecting data ranges, zooming, and panning—to the visualization.
Chapter 1

Google Analytics dashboard

The dashboard (frontend) will be capable of loading, parsing, and grouping log files of different formats, filtering data based on selected data ranges and displaying, updating, and animating multiple chart types. We will build the visualization library with D3.js and the application with AngularJS.

While developing this application, we will see all the advantages, problems, and best practices of D3.js and AngularJS in action. Also, you will learn to combine both frameworks to one single application.

During most of the chapters, we will focus on loading log data from static resources. In the last chapter, we will implement a simple web server (backend) using Node.js and Express to detect log file changes in real time and push these changes to the dashboard using WebSockets and the Socket.IO library.

Besides implementing all the functionality, you will also learn how to design and test (we will use unit and integration tests) the frontend components with the test runners, Karma and Protractor.
Terminology and definitions

Let's start from the beginning. In the following chapters, we will discuss computer graphics and visualizations for the Web. Therefore, it's important to understand the basic terminology of this domain. In this section, I will refresh your knowledge of the two most common image representations (vector and pixel graphics). I will also discuss the web standards that are relevant for graphical applications (DOM, SVG, and so on). If these definitions are not new for you, then nothing can stop you from jumping directly to the next section.

Document Object Model

The Document Object Model (DOM) is the tree representation of the hierarchical elements of an HTML document and it was specified by the World Wide Web Consortium (W3C). These elements in the DOM are called nodes (for example, html, head, body, and so on), which can have attributes (for example, class="header") and content (for example, "My Application" is the content of the h1 node). The DOM provides a public JavaScript interface with which we can access nodes and manipulate them.

Let's look at the source code of a simple HTML page to see an example of the DOM tree:

```html
<html>
  <head>
    <title>My App</title>
  </head>
  <body>
    <h1 class="header">My Application</h1>
    <p class="content">
      Lorem ipsum dolor sit amet, ...
    </p>
  </body>
</html>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Developers often visualize the DOM tree and its hierarchical elements by indenting the HTML code properly. The DOM tree of the previous example can also be displayed and modified with the developer tools of a modern browser. See the following screenshot:

![DOM tree in Chrome developer tools](image)

**Vector graphics and Scalable Vector Graphics**

A vector graphic is an image represented solely by the geometric primitives (shape) and attributes (size, margins, appearance, and so on) of its containing elements. These elements can be primitive shapes (such as lines, circles, triangles, and so on) or complex shapes that are composed by these primitives. All elements are included in the graphic; therefore, the whole graphic can be described by implicit mathematical expressions. Thus, the size of a vector graphic is only dependent on the number and complexity of the elements in the image and not on the resolution in which it will be displayed.

**Rule of thumb for vector graphics**

The more the elements in the image, the higher the cost to draw or update the graphic, but in general, the costs do not depend on the resolution of the generated image (lossless rescaling).

Let's look at a simple example. An image of a circle can be fully described by the shape of a circle with the coordinates of its center point and radius. In all modern browsers, we can generate a vector graphic by embedding Scalable Vector Graphics (SVG), a web standard for vector graphics specified by the W3C directly in the DOM, as follows:

```html
<html>
  ... 
  <body>
    <svg id="vis" width="640" height="480">
      ... 
    </svg>
  </body>
</html>
```
All containing elements of a vector graphic are usually stored in a scene graph, a tree representation of the hierarchical elements of the graphic. This concept is very similar to the DOM tree, and in the case of SVG, the scene graph is directly embedded in the DOM tree. We can see this in the previous example (by looking at the svg node) that contains the circle element.

As we mentioned previously, we can also use JavaScript to generate or modify SVGs. Let's use the partially complete helper function that generates and draws SVG nodes. Don't worry about the details of this function; just imagine that it will create a vector graphic element (you can look it up in the source code of this chapter if you are brave enough):

```javascript
<script type="text/javascript">
function VectorGraphic(parent, tag, attrs) {

  var el = document.createElementNS('http://www.w3.org/2000/svg', tag);

  ...

  return {
    on: function(event, handler){...},
    fill: function(color){...},
    stroke: function(color){...},
    draw: function() {
      parent.appendChild(el);
    }
  }
}
</script>
```

The preceding function creates a new SVG node and returns a .draw() method. It appends this new element to the parent container. Additionally, it provides methods to style the layout of the new element. Let's use this function to create the same circle from the previous example from within JavaScript:

```javascript
<script type="text/javascript">
var ctx = document.getElementById('vis');

var circle = VectorGraphic(ctx, 'circle', {cx:25, cy:25, r:20});

circle.fill('red');

circle.draw();
</script>
```
The preceding code will generate the same circle as before with the center point at the coordinates 25, 25 and a radius of 20. The following figure shows the result of the graphic in the browser, where the left-hand side shows the original image and the right-hand side shows the image after zooming in:

![A vector graphic generated in the SVG node (left: normal, right: zoom)](image)

We observe that the circle will always appear in the best resolution no matter how far we zoom into the image. The reason for this is that the vector graphics are recomputed (by the mathematical expressions), redrawn, and rasterized according to the relevant display and zoom factor.

We silently skipped another very important fact of SVGs such that all the elements of the graphic appear in the DOM tree of the SVG node. In our case, we can see in the previous example of the HTML page that the SVG node contains a circle element. This means that the browser also knows about all the elements in the image. Therefore, we can use all built-in capabilities to style and observe these elements. For example, it’s possible to attach an event listener with JavaScript to the circle and call event handlers when the user interacts with this element of the graphic. This event handler could look like this:

```javascript
<script type="text/javascript">
  var ctx = document.getElementById('vis');
  var circle = VectorGraphic(ctx, 'circle', {cx: 25, cy: 25, r: 20});
  circle.fill('red');
  circle.on('mouseover', function() {
    this.stroke('blue');
  });
  circle.draw();
</script>
```
Vector graphics (particularly generated with SVG) are used in the Web, in general, to draw graphics that contain a moderate number of elements and when interactions and controls (such as zooming, panning, selecting elements, and so on) are desired. Graphics for high performance with a big number of elements would rather use pixel graphics.

**Pixel graphics**

A pixel graphic (often called as a raster graphic) is an image that is represented solely by the pixels in the graphic. Thus, its size is only dependent on the resolution of the image.

---

**Rule of thumb for pixel graphics**

The more the pixels in the image (the higher the resolution of the image), the higher the cost to draw or update the graphic, but in general, the cost does not depend on the number of elements in the generated image.

---

In general, pixel graphics are rasterized images of a geometric representation. Therefore, an image of a circle can be just as well defined by the shape of a circle with coordinates of its center point and a radius. The description of the circle is exactly the same as for vector graphics, only the ways of storing and displaying the image are different.

Let's write a partially complete JavaScript helper function to generate pixel graphics with the Canvas API. Don't worry about the details of this function; just imagine that it will create a pixel graphic element (you can look it up in the source code of this chapter if you are brave enough):

```javascript
<script type="text/javascript">
function PixelGraphic(parent, tag, attrs) {
    var el = parent.getContext('2d');
    el.beginPath();
    ...
    return {
        stroke: function (color){... },
        fill: function (color){ ... },
        draw: function () {
            el.arc(attrs.cx, attrs.cy, attrs.r, 0, 2*Math.PI);
            ...
        }
    }
}</script>
```
If we generate such a circle with JavaScript, the resulting code looks very similar to the previous vector graphic example:

```javascript
<script type="text/javascript">
    var ctx = document.getElementById('vis');
    var circle = PixelGraphic(ctx, 'circle', {cx:25, cy:25, r:20});
    circle.fill('red');
    circle.draw();
</script>
```

The preceding code generates a very similar circle element with the center point at the coordinates 25, 25 and a radius of 20. However, if we look at the resulting DOM tree, we observe a small but important difference, that is, the circle element is not visible in the canvas node. Let's look at the HTML code of this example:

```html
<html>
...
<body>
    <canvas id="vis" width="640" height="480"></canvas>
</body>
</html>
```

The pixel graphic in the previous example is now stored in the `canvas` element. As a result, the image is rasterized and transformed to a pixel array of color values. Therefore, the `canvas` element does not contain any information about the elements of the image.

If we look at the result of this code in the browser, we see the exact same image as before with the vector circle. The only small, but very important difference will become visible once we zoom inside the graphic. We will soon start to see pixels instead of a sharp and smooth circle. This effect is visualized in the following figure:

![A pixel graphic generated in canvas (left: normal, right: zoom)](image-url)
This effect can be easily explained if we look closer at how pixel graphics are stored. First, the image is internally drawn on a container with the defined dimension of the image. Then, the image is divided by a raster, which is defined by the image resolution. Finally, the image is stored as pixel values of this raster. If we see a pixelated image, we see exactly these discrete rastered pixel values. The whole process is called **rasterization** and is visualized in the following figure:

![A pixel graphic generated and rasterized in Canvas](image1)

Besides lossy rescaling, we can spot another big difference to vector graphics: the browser does not know about the elements in the image. Therefore, it's not possible to see and address the elements of the graphics in the DOM tree anymore. It's not possible anymore to attach an event listener to the circle and call event handlers when the user interacts with an element of the graphic. In pixel graphics, we are primarily interacting with pixel values instead of objects, as we can see in the following figure:

![Interaction with the generated pixel graphic](image2)

In pixel graphics, we need to think in pixels rather than in elements of an image.
We usually prefer pixel graphics in the browser—particularly generated with Canvas (2D) or WebGL (3D)—for graphics that contain a big number of elements and performance (high frame rate) is more important than the image quality, interactions, and zooming.

Understanding Data-Driven Documents

Data-Driven Documents (D3.js) is a data-driven library for DOM manipulation and a graphical toolkit with maximum compatibility, accessibility, and performance. It utilizes fully the capabilities of modern browsers and web standards (such as HTML, CSS, and SVG).

It is open source and hosted on GitHub (https://github.com/mbostock/d3) under a slightly modified BSD 3-clause license; therefore, it can be used in commercial products without being required to release any modifications. By the way, GitHub itself uses D3.js to visualize the contribution history of repositories.

Make yourself familiar with the wiki pages and the API reference on GitHub as they will become your companions during the next weeks:


Why do we use D3.js?

D3.js is used for various different tasks, but it's mainly used for the following purposes:

- Transforming HTML or SVG elements in the DOM tree, as shown in the following code:

  ```javascript
  // Example for HTML
  // Change the background color of all p elements
  d3.selectAll('p').style('background-color', 'red');
  </script>
  ```

- Transforming data into HTML or SVG elements as follows:

  ```javascript
  // Example for SVG
  d3.selectAll('circle').data(dataArray)
  .enter()
  .append('circle');
  </script>
  ```
The Magic of SVG, D3.js, and AngularJS

• Generating or preparing complex visual content, as shown in the following code:

```html
<script type="text/javascript">
  // Create a Chord element
  var chord = d3.layout.chord()
    .sortSubgroups(d3.descending)
    .matrix(matrix);
</script>
```

• Loading data using AJAX requests as follows:

```html
<script type="text/javascript">
  // Load external data
  d3.json('data.json', function(error, data){
    // do something with the data
  });
</script>
```

D3.js is not a chart library! It provides low-level tools to build dynamic visualizations; therefore, many chart libraries are built on top of D3.js.

One reason why D3.js gained a lot of popularity is its *data-driven* approach. Instead of explicitly looping over elements in an array and drawing them on the screen, D3.js allows for an *implicit* declarative representation. In D3.js, we rather think in terms of how the visualization is composed than how each element is arranged in the scene. The second main reason for its popularity is its *clear focus on its underlying web standards* (HTML, SVG, and CSS). This brings many advantages such as the following:

• **Compatibility:** D3.js is not abstracting the underlying standards, it’s exploiting them. Therefore, developers can use all standard attributes of HTML, SVG, and CSS to compose and style their visualizations rather than learning an abstraction API for the visualization library.

• **Debugging:** D3.js will not only append all HTML elements and styles to the DOM, but it will also append all SVG elements and their CSS attributes. This makes it possible to simply open the developer tools of the browser and look at the generated and modified elements and attributes. It lets developers use their standard debugging tools and workflows that they are already familiar with. Whoever dealt with debugging of pixel graphics libraries (such as OpenGL, WebGL, Canvas, and so on) knows that good debugging capabilities are a real game changer.
• **Performance:** D3.js relies on SVG and therefore facilitates optimizing performance of interactions and animations by giving full access to all SVG features. In most other graphical libraries, one is limited to the capabilities provided by the abstraction layer and the API of the library.

### The killer feature – data joins

There is one more feature that distinguishes D3.js from other DOM transforming libraries such as jQuery: the concept of **data joins**. When binding an array of data, D3.js automatically intersects the *old* dataset with the *new* one to generate three new datasets:

- The **enter set** that stores all elements from the *new* dataset that are not in the *old* dataset and therefore need to be added
- The **update set** that stores all elements from the *new* dataset that are already in the *old* dataset and therefore need to be updated
- The **exit set** that stores all the elements from the *old* dataset that are not in the *new* dataset and therefore need to be removed

The following figure visualizes this intersection, where the *old* dataset is called **Selection** and the *new* dataset is called **Data**:

![Data joins in D3.js](image)

This technique is often referred to as **data binding** because we are literally binding an array of elements to a Selection of elements. However, now we know that data joins are not just data bindings, but they additionally intersect the datasets.
Let's look at a simple example. In general, the data-driven approach of D3.js allows developers to declare the manipulations of HTML or SVG elements based on CSS selectors. This is very similar to jQuery; therefore, I will also show the corresponding code using jQuery:

```javascript
// with jQuery
$('p').css('background-color', 'red');
// with D3.js
d3.selectAll('p').style('background-color', 'red');
</script>
```

However, the big difference is that D3.js implements data joins, which gives developers the access to match an array of elements (the new dataset) to a Selection (the old dataset). Corresponding with the enter, update, and exit sets from the previous intersection figure, D3.js can return these intersected datasets using the following functions:

- `selection.data(dataSet).enter()` for elements that are new to the dataset and not yet in the current Selection
- `selection.data(dataSet)` for elements that are already existent in the dataset
- `selection.data(dataSet).exit()` for elements that are removed from the dataset and still existent in the current Selection

Let's look at an example where we use all of the preceding methods. First, we will write a function that appends, updates, and removes p elements in the DOM. Then, we will play around with it:

```javascript
function join_p(dataSet) {
    var el = d3.select('body');

    var join = el
      // get the selection of all p elements
      .selectAll('p')
      // join the selection with the dataset
      .data(dataSet);

    // elements not yet in the selection
    // they need to be added
    join.enter().append('p');

    // elements currently in the selection
    // they need to be updated
```
Chapter 1

```javascript
join.text(function(d) { return d; });

// elements still in selection
// they need to be removed
join.exit().remove('p');
</script>

Let's play with this function in the developer tools of the browser. At first, we see a blank page without any `p` elements in the DOM. Okay, now we call the `join_p(['append', 'to', 'DOM'])` function from the console inside the browser.

We observe that three paragraphs appear with the content `append`, `to`, and `DOM`; we can also look at the DOM tree in the developer tools:

```html
<body>
  <p>append</p>
  <p>to</p>
  <p>DOM</p>
</body>
```

So what happened here? In the `join_p()` function, we first created a Selection of all `p` elements in the body using `.selectAll('p')` and then created a data join with the `['append', 'to', 'DOM']` dataset using `.data(dataSet)`. It seems weird that we call `.selectAll('p')` where not a single `p` element exists yet in the DOM. However, if we think in terms of data joins, we solely create an empty Selection of `p` elements. This makes sense immediately after calling the `enter` function, which returns all elements that are not yet existing in this Selection. In our case of the empty Selection, this function returns all the elements of the dataset. Finally, we just need to append them to the DOM using the `.append('p')` function.

In the following line, the `join` variable returns all elements of the current Selection and we just appended three new elements to it. The `.text()` method updates all elements of the current Selection and sets the value of the array element as text of the corresponding `p` tag (this method is called dynamic properties and will be explained in more detail in the following chapter). The last method, `.exit()`, returns no elements because all elements are available in the dataset and in the Selection. The following figure shows how the Selection changes with the dataset:

Adding elements to a Selection and updating them
If we now call the `join_p()` function again, this time with the following dataset `join_p(['modify', 'in', 'DOM'])`, we see that the text of the first two paragraphs will change as follows:

```html
<body>
  <p>modify</p>
  <p>in</p>
  <p>DOM</p>
</body>
```

Despite the previous function call, the Selection of `p` elements now is not empty, but contains the three previous elements. This means that both `.enter()` and `.exit()` methods will return no elements. The `join` variable solely contains the new updated elements whose paragraph text is correspondingly updated. We can see the effect on the Selection in the following figure:

![Updating elements of a Selection](image)

Finally, we can try to call `join_p([])` with an empty dataset. As we could imagine by now, this results in all paragraphs being removed. The `.exit()` function will return all elements of the Selection because the dataset contains no elements. Calling `.remove()` on these elements will remove them from the DOM. We can observe the change of the Selection in the following figure:

![Removing elements from a Selection](image)

Data joins are data bindings with access to the intersection of the dataset and the Selection.
The concept of data joins enable the developer to append new data to a graphic when new data is available, to update existing data and to remove data from the graphic when it is not available anymore. Instead of redrawing the complete image, the elements of the graphic are transformed.

Finding resources
Michael Bostock provides an extensive source of detailed information on D3.js, helpful posts, and lots of examples. Once you are stuck or need to find particular information on specific topics or examples, I recommend you to read through the following links:

- Michael Bostock's web page at http://bost.ocks.org/mike/
- Infinite amount of examples and demos at http://bl.ocks.org/mbostock
- Stack Overflow questions at http://stackoverflow.com/questions/tagged/d3.js

If you Google D3.js, you will find a lot of additional resources; however, most of them are just dealing with the basics. To get a good and deeper understanding of D3.js, I would rather advise you to look up the relevant chapters in the book Mastering D3.js, Pablo Navarro Castillo, Packt Publishing, or look directly into the source code of D3.js on GitHub.

D3.js meets AngularJS
AngularJS is a JavaScript framework that modernizes development of web applications in multiple ways; it introduces client-side templates, MVC/MVVM pattern, scoping, two-way data binding, dependency injection, and so on. Therefore, it's our JavaScript application framework of choice. At this point, I assume that you are already familiar with the main concepts of AngularJS and you know when and how to apply them. If there are still problems, I would recommend you to read the relevant chapters in the book Mastering Web Application Development with AngularJS by Pawel Kozlowski and Peter Bacon Darwin, published by Packt Publishing.

Theoretically, we can simply add a D3.js visualization library to the same application that also uses AngularJS without caring about modules, isolation, dependency injection, and so on without any extra effort.
However, once we know how awesome AngularJS is, we want to fully exploit every single advantage of this framework. Having said that, we want every component of the application being injectable, maintainable, and testable. We want to extend HTML syntax and add custom directives to templates. We want proper scope isolation. We want to put common tasks into reusable services. We want to use dependency injection on every single component of the application. We want to integrate D3.js into an application the Angular way.

Testable and maintainable components

AngularJS strongly focuses on testability and maintainability of the components of an application. Once we use plain D3.js to modify the DOM in order to load data and create graphical content, it will become very complex and uncomfortable to test single components or the whole application. We will use the full power of AngularJS, the concepts of dependency injection, modularization, isolation, and directives to create testable components.

Custom directives

AngularJS lets you develop your own directives that extend the HTML syntax to create reusable components for HTML. This is exactly what we want: a reusable component for each different type of visualization that we are going to build. We aim to declare the different elements of a visualization like in the following example:

```html
<html>
  <head>
    <script>
      ...
      app.directive('d3Map', function() { ... });
      app.directive('d3LineChart', function() { ... });
      app.directive('d3ScatterPlot', function() { ... });
      app.directive('d3ChordDiagram', function() { ... });
    </script>
  </head>
  <body>
    <d3-map></d3-map>
    <d3-line-chart data="data"></d3-line-chart>
    <d3-scatter-plot data="data"></d3-scatter-plot>
    <d3-chord-diagram data="data"></d3-chord-diagram>
  </body>
</html>
```
We can immediately see that this is a very clean and elegant way to embed your visualization components in the HTML document.

**Custom filters**

AngularJS introduces filters in frontend templates that allow you to modify variables and filter arrays directly inside the template. For our visualization component, we want to create custom filters (for example, to clamp the dataset to a specific range) that can be applied to all graphics at once. Additionally, we want these filters to be autoupdated whenever data is selected in one graphic as follows:

```html
<html>
<head>
  <script>
    ...
    app.filter('startDate', function(){ ... });
  </script>
</head>
<body>
  <d3-line-chart data="timeData | startDate: '01.01.2015'"></d3-line-chart>
  <d3-scatter-plot data="timeData | startDate: '01.01.2015'"></d3-scatter-plot>
</body>
</html>
```

**Custom loading and parsing service**

AngularJS emphasizes the concepts of services to implement common functionalities. We want to implement a data loading and parsing service that uses AngularJS' Promises and the capabilities of D3.js parsing functions at the same time. The service should be used like this:

```javascript
<script type="text/javascript">
  app.controller('MainCtrl', ['$scope', 'myService',

    function($scope, myService) {
      myService.get('data.json').then(function(data){
        $scope.data = data;
      });
    }]);
</script>
```
A brief overview of visualization tools for the Web

Now, let me explain why we are using D3.js and not any other similar visualization library to manipulate vector graphics.

When I started with web development in 2004, vector graphics and interactive applications were mostly embedded as Flash objects into web pages. The main reason for this was the lack of web standards for vector graphics or good and powerful JavaScript libraries for image manipulation. It was difficult to create interactive visualizations that integrate into the underlying page because functionalities (such as layouts and fonts) defined in CSS and user interactions written in JavaScript were not available in embedded objects. Thus, these visualization objects often felt like a strange add-on to the underlying application, where fonts, size, and colors were not completely matching with the application.

A typical code example for an embedded visualization looks like this:

```html
<html>
  ...
  <body>
    <object id="vis" width="50" height="30">
      <param name="movie" value="vis.swf">
      <embed src="vis.swf" type="application/x-shockwave-flash">
    </object>
  </body>
</html>
```

We can see in the preceding example that the compiled and embedded `vis.swf` Flash object is completely isolated from the scope of the host application. In addition to the web browser, we would also need a plugin that can interpret the Flash binary object. While the application and the embedded visualization have the same task—displaying data on a web page—they are not sharing common styles or a common scope for user interactions.

Java and Flash

One of the first consistent toolsets for interactive data visualization for the Web was the Java library Prefuse, which was published by Jeffrey Heer in 2007, who, at this time, is a member at the Berkley Visualization Lab. Prefuse provided rich tools for data modeling and interactive graphics. The visualization could be embedded as Java applets into a web page, but this required the Java runtime environment to be installed on every browser that wants to display the visualization.
Later in 2008, Heer released the first version of Flare, a port of the Prefuse library to ActionScript, which could compile the visualization to a more common Flash object. A simple application that shows three circles with different x coordinates in an image with the size of 50 x 30 pixel looks like this with Flare:

```ActionScript
[SWF(width="50", height="30")]
public class Vis extends Sprite
{
    public function Vis()
    {
        // Define the dataset
        var data:Array = [15, 25, 35];
        for each (var d:int in data) {
            var sprite:Sprite = new Sprite();
            // Draw and color the circles
            sprite.graphics.beginFill(0xff0000, 1.0);
            sprite.graphics.drawCircle(0, 0, 5);
            this.addChild(sprite);
            // Set the coordinates of the circle
            sprite.x = d;
            sprite.y = 15;
        }
    }
}
```

Looking at the preceding code, we can see that in each loop, we create a drawing container (sprite) and a circle element. We also color it and with `sprite.x = d`, we set the x coordinate of the sprite container to the current value d of the data array. Don't worry if this code looks quite complicated to you because this is exactly the reason why I am showing it. It's complicated and not intuitive to create containers, add elements, and move containers to the position where we want to display the circle. Furthermore, to run this example in the browser, we have a very similar problem as before with Prefuse: each browser needs the Flash runtime installed. The resulting image generated by the previous code will look like the following figure:

Image of three circles
Raphaël (JavaScript – SVG/VML)

By 2008, most of the major browsers provided native support for SVG and signified the certain end of Flash in the upcoming years. However, Internet Explorer implemented a different markup language for vector graphics than SVG; it used the similar, but different Vector Markup Language (VML).

In 2009, Dmitry Baranovskiy announced the first release of Raphaël, a JavaScript library that aims to simplify the manipulation of vector graphics by providing a JavaScript API for SVG with a compatibility layer for VML for Internet Explorer. The representation of graphics inside the DOM not only enabled the use of JavaScript event handlers for user interactions on elements of the image, but it also enabled the use of CSS to style these elements. This was a huge step towards open web standards, accessibility, and acceptance of SVG. An example of drawing the same three circles looks like this:

```html
<script type="text/javascript">
    // Define the dataset
    var data = [15, 25, 35];
    // Draw the canvas
    var paper = Raphael(0, 0, 50, 30);
    // Draw and color the circles
    for (var i = 0; i < 3; i++) {
        var circle = paper.circle(data[i], 15, 5);
        circle.attr('fill', 'red');
    }
</script>
```

Again, we generate a circle with the $x$ coordinate of the data array each time we loop over the array. In a modern browser, the preceding code produces an SVG image that looks exactly like the previous example, but additionally outputs directly to the DOM of the HTML page. It adds three circle elements with different $x$ coordinates of the center point to the DOM tree inside the SVG node. The generated SVG code embedded in the web page will look like this:

```html
<html>
    ...
    <body>
        <svg width="50" height="30">
            <circle cx="15" cy="15" r="5" style="fill:red;"/>
            <circle cx="25" cy="15" r="5" style="fill:red;"/>
            <circle cx="35" cy="15" r="5" style="fill:red;"/>
        </svg>
    </body>
</html>
```
Protovis (JavaScript – SVG)

In the same year at Stanford, Michael Bostock and Jeffrey Heer published the visualization library Protovis for SVG manipulation in JavaScript. Its new data-driven approach allowed the developers to declare the representation of data, rather than looping and drawing multiple elements explicitly. The following code uses Protovis to generate the exact same SVG graphic of the three circles shown in the previous figure:

```javascript
<script type="text/javascript">
// Define the dataset
var data = [15, 25, 35];
// Define the canvas
var vis = new pv.Panel()
  .width(50)
  .height(30);
// Define the circles and their color
vis.add(pv.Dot)
  .data(data)
  .left(function(d) { return d; })
  .bottom(15)
  .radius(5)
  .fillStyle('red');
// Draw the canvas and the circles
vis.render();
</script>
```

The main difference in the previous example is that the explicit loop over the data array in Raphaël is replaced by the implicit `data()` function in Protovis, where the x coordinate of each circle is called as a function that returns the current element of the data array.

D3.js (JavaScript – HTML/SVG)

In 2011, when SVG was finally supported in all major browsers and Internet Explorer, the same authors of Protovis—Michael Bostock and further members of the Stanford Visualization Group—published D3.js, a more generalized version of Protovis with built-in support for animations. The goal was not to restrict the library anymore on just the SVG object, but to access the complete DOM tree and use all of its features and underlying standards. Therefore, all updates and extensions for HTML and CSS (for example, new attributes, and so on) are immediately available in D3.js. To support dynamic visualizations, D3.js also introduced the concepts of data joins, which let the developer add, update, and remove elements depending on data that was added, updated, or removed from a Selection (this will be discussed in more detail in the next chapter).
The same graphic as previously generated with Raphael and Protovis can be created with D3.js as follows:

```javascript
<script type="text/javascript">
  // Define the dataset
  var data = [15, 25, 35];

  // Draw the canvas
  var vis = d3.select('body').
    append('svg')
    .attr('width', 50)
    .attr('height', 30);

  // Draw and color the circles
  vis.selectAll('circle')
    .data(data)
    .enter().append('circle')
    .attr('cx', function(d) { return d; })
    .attr('cy', 15)
    .attr('r', 5)
    .style('fill', 'red');
</script>
```

We remark that D3.js implements more general methods, for example, `attr('r')` that uses the underlying SVG attribute `r` for the radius explicitly instead of the `radius()` function, which is an abstraction of the SVG attribute `r` in Protovis.

The `selectAll().data().enter()` construct in this static example looks more complicated than the simple `data()` function of Protovis, but it implements data joins to create new elements for new data in the Selection. Therefore, it will be very useful to update dynamic graphics in the following chapters of this book.

D3.js forces developers to use the underlying standards HTML, CSS, and SVG instead of providing an abstraction of these attributes.

**Canvas API (JavaScript – Canvas)**

Since the introduction of HTML5, we can also use the Canvas element and its JavaScript API to draw the exact same circles as in the previous examples. The performance of Canvas is much better than SVG when drawing large amount of objects/elements. However, the content of the Canvas will be drawn as a pixel graphic and no element will be appended to the DOM or the Canvas node, which is a huge drawback.
Three.js (JavaScript – WebGL)

With WebGL that was introduced in 2011, we can also draw the example of the three circles using the Three.js JavaScript library. WebGL has access to hardware acceleration of the operating system and is mainly used for 3D graphics. The resulting image is a pixel graphic. However, it’s worth mentioning that it’s not available in all modern browsers.

Summary

In this chapter, we discovered the benefits of the powerful combination of SVG, D3.js, and AngularJS. SVG are lossless and rescalable vector graphics that are supported in all modern browsers. All the elements of an image are directly appended to the DOM tree of the HTML page, which makes debugging with the browser’s developer tools very comfortable. D3.js is a versatile low-level library for generating interactive graphical content based on the underlying standards (HTML, CSS, and SVG).

In the first section of this chapter, we outlined the sample application that we will develop during this book: a dashboard for interactive real-time visualizations of web server logs and system status.

You learned about the concept of data joins, which differentiates D3.js from other DOM-transformation libraries like jQuery. Developers can not only bind data to a Selection, but also retrieve the enter, update, and exit sets (the intersection of the dataset with a current Selection of DOM elements).

D3.js and AngularJS can live side by side in an application, but we want to integrate the D3.js visualization into the AngularJS application the Angular way. This enhances maintainability, testability, and reusability of all components of an application.

In the next chapter, we will start to develop our first visualization with D3.js. In the first step, you will learn about Selections and transformations and use them to draw circles in SVG. Then, we will apply the concepts of data binding and data joins to draw a simple scatter chart based on an array of data samples.
Where to buy this book

You can buy Data Visualization with D3 and AngularJS from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.