Chapter 1
brief contents

**PART 1   INTRODUCING SPAs** .................................................................1

1 ■ Our first single page application   3  
2 ■ Reintroducing JavaScript   23

**PART 2   THE SPA CLIENT** .................................................................59

3 ■ Develop the Shell   61  
4 ■ Add feature modules   95  
5 ■ Build the Model   139  
6 ■ Finish the Model and Data modules   178

**PART 3   THE SPA SERVER** .................................................................227

7 ■ The web server   229  
8 ■ The server database   265  
9 ■ Readying our SPA for production   313
Introducing SPAs

In the time it takes to read this page, 35 million person minutes will be spent waiting for traditional website pages to load. That’s enough spinning icon time for the Curiosity Lander to fly to Mars and back 96 times. The productivity cost of traditional websites is astonishing, and the cost to a business can be devastating. A slow website can drive users away from your site—and into the welcoming wallets of smiling competitors.

One reason traditional websites are slow is because popular MVC server frameworks are focused on serving page after page of static content to an essentially dumb client. When we click a link in a traditional website slideshow, for example, the screen flashes white and everything reloads over several seconds: the navigation, ads, headlines, text, and footer are all rendered again. Yet the only thing that changes is the slideshow image and perhaps the description text. Worse, there’s no indicator when some element of the page becomes functional. For example, sometimes a link can be clicked as soon as it appears on a web page; other times we have to wait until the redrawing is 100% complete plus five seconds. This slow, inconsistent, and clunky experience is becoming unacceptable for an increasingly sophisticated web consumer.

Prepare to learn about another—and dare we say better—approach to developing web applications, the single page web application (SPA). An SPA delivers a desktop application in the browser. The result is a highly responsive experience that surprises and delights users instead of confusing and annoying them. In part 1 we learn:

- What an SPA is and the advantages it provides over traditional websites
- How an SPA approach can make our web application a great deal more responsive and compelling
- How to improve our JavaScript skills for SPA development
- How to build an example SPA

Product design is increasingly seen as the decisive factor in the success of commercial and enterprise web applications. SPAs are often the best choice to provide the optimal user experience. As a result, we expect the demand for user-focused design to drive SPA adoption and sophistication.
Our first single page application

This chapter covers

- Defining single page web applications
- Comparing the most popular single page application platforms—Java, Flash, and Javascript
- Writing our first JavaScript single page application
- Inspecting the application using Chrome Developer Tools
- Exploring the user benefits of single page applications

This book is intended for web developers, architects, and product managers with at least a smattering of JavaScript, HTML, and CSS experience. If you’ve never even dabbled in web development, this book is not for you, although you’re welcome to buy it anyway (go ahead, daddy needs a new car). Many books are available that do a great job teaching beginner website development and design, but this isn’t one of them.

This book does aspire to be a great guide to designing and building large-scale single page web applications [SPAs] using JavaScript end to end. In fact, as figure 1.1
CHAPTER 1  Our first single page application

depicts, we use JavaScript as the language of the database, the web server, and the browser application.

We’ve spent the last six years leading the development of numerous large-scale commercial and enterprise SPAs. During that time we’ve constantly updated our practices to meet the challenges we’ve found. We share these practices in this book as they have helped us develop faster, provide a better user experience, ensure quality, and improve team communication.

1.1 Definition, a little history, and some focus

An SPA is an application delivered to the browser that doesn’t reload the page during use. Like all applications, it’s intended to help the user complete a task, such as “write a document” or “administer a web server.” We can think of an SPA as a fat client that’s loaded from a web server.

1.1.1 A little history

SPAs have been around for a long time. Let’s look at some early examples:

- **Tic-Tac-Toe**—http://rintintin.colorado.edu/~epperson/Java/TicTacToe.html. Hey, we didn’t say this would be pretty. This application challenges us to beat a formidable and ruthless computer nemesis in a game of Tic-Tac-Toe. The Java plugin is required—see http://www.java.com/en/download/index.jsp. You may have to grant permission for your browser to run this applet.

- **Flash Spacelander**—http://games.whomwah.com/spacelander.html. This is one of the earlier Flash games, written by Duncan Robertson circa 2001. The Flash plugin is required—see http://get.adobe.com/flashplayer/.

- **JavaScript mortgage calculator**—http://www.mcfedries.com/creatingawebpage/mortgage.htm. This calculator seems almost as old as JavaScript itself, but it works nicely. No plugin is required.

The astute reader—and even a few slovenly ones—will notice that we’ve provided examples of three of the most popular SPA platforms: Java applets, Flash/Flex, and

---

1 If you’re reading this chapter as you eat potato chips off your chest, you’re slovenly.
JavaScript. And those same readers may have noticed that **only the JavaScript SPA works without the overhead or security concerns of a third-party plugin.**

Today, JavaScript SPAs are often the best choice of the three. But JavaScript took a while to become competitive, or even possible, for most SPA uses. Let’s take a look at why.

### 1.1.2 What took JavaScript SPAs so long?

Flash and Java applets had evolved nicely by the year 2000. Java was being used to deliver complex applications and even a complete office suite via the browser.² Flash had become the platform of choice for delivering rich browser games and, later, video. On the other hand, JavaScript was still mostly relegated to little more than mortgage calculators, form validation, rollover effects, and pop-up windows. The problem was that we couldn’t rely on JavaScript (or the rendering methods it used) to provide critical capabilities consistently on popular browsers. Even so, JavaScript SPAs promised a number of enticing advantages over Flash and Java:

- **No plugin required**—Users access the application without concern for plugin installation, maintenance, and OS compatibility. Developers also don’t need to worry about a separate security model, which reduces development and maintenance headaches.³
- **Less bloat**—An SPA using JavaScript and HTML should use significantly fewer resources than a plugin that requires an additional run-time environment.
- **One client language**—Web architects and most developers have to know many languages and data formats—HTML, CSS, JSON, XML, JavaScript, SQL, PHP/Java/Ruby/Perl, and so on. Why write applets in Java, or Flash applications in ActionScript, when we’re already using JavaScript elsewhere on our pages? Using a single programming language for everything on the client is a great way to reduce complexity.
- **A more fluid and interactive page**—We’ve all seen a Flash or Java application on a web page. Often the application is displayed in a box somewhere and many details are different than the HTML elements that surround it: the graphical widgets are different, the right-click is different, the sounds are different, and interaction with the rest of the page is limited. With a JavaScript SPA, the entire browser window is the application interface.

As JavaScript has matured, most of its weaknesses have been either fixed or mitigated and its advantages have increased in value:

- **The web browser is the world’s most widely used application**—Many people have a browser window always open and use it throughout the day. Access to a JavaScript application is one more bookmark click away.

---

² Applix (VistaSource) Anywhere Office
³ Can you say “same origin policy”? If you’ve ever developed in Flash or Java, you almost certainly are familiar with this challenge.
CHAPTER 1  Our first single page application

- **JavaScript in the browser is one of the world’s most widely distributed execution environments**—By December 2011, nearly one million Android and iOS mobile devices were being activated every day. Each of these devices has a robust JavaScript execution environment built into the OS. More than one billion robust JavaScript implementations have shipped in the last three years on phone, tablet, laptop, and desktop computers around the world.

- **Deployment of JavaScript is trivial**—A JavaScript application can be made available to more than a billion web users by hosting it on an HTTP server.

- **JavaScript is useful for cross-platform development**—Now we can create SPAs using Windows, Mac OS X, or Linux, and we can deploy a single application not only to all desktop machines but also to tablets and smart phones. We can thank converging implementations of standards across browsers, and mature libraries such as jQuery and PhoneGap that smooth over inconsistencies.

- **JavaScript has become surprisingly fast and can, at times, rival compiled languages**—Its speedup is thanks to ongoing and heated competition between Mozilla Firefox, Google Chrome, Opera, and Microsoft. Modern JavaScript implementations enjoy advanced optimizations such as JIT compilation to native machine code, branch prediction, type-inference, and multi-threading.4

- **JavaScript has evolved to include advanced features**—These features include the JSON native object, native jQuery-style selectors, and more consistent AJAX capabilities. Push messaging has become far easier with mature libraries like Strophe and Socket.IO.

- **HTML5, SVG, and CSS3 standards and support have advanced**—These advancements allow for the rendering of pixel-perfect graphics that can rival the speed and quality produced by Java or Flash.

- **JavaScript can be used throughout a web project**—Now we can use the excellent Node.js web server and data stores such as CouchDB or MongoDB, both of which communicate in JSON, a JavaScript data format. We can even share libraries between the server and the browser.

- **Desktop, laptop, and even mobile devices have become more powerful**—The ubiquity of multi-core processors and gigabytes of RAM means processing that used to be accomplished on the server can now be distributed to the client browsers.

JavaScript SPAs are becoming increasingly popular due to these advantages, and the demand for experienced JavaScript developers and architects has blossomed. Applications that were once developed for many operating systems (or for Java or Flash) are now delivered as a single JavaScript SPA. Startups have embraced Node.js as the web server of choice, and mobile application developers are using JavaScript and PhoneGap to create “native” applications for multiple mobile platforms using a single code base.

---

JavaScript isn’t perfect, and we don’t have to look far to find omissions, inconsistencies, and other aspects to dislike. But this is true of all languages. Once you become comfortable with its core concepts, employ best practices, and learn what parts to avoid, JavaScript development can be pleasant and productive.

Generated JavaScript: One destination, two paths

We’ve found it easier to develop SPAs using JavaScript directly. We call these native JavaScript SPAs. Another surprisingly popular approach is to use generated JavaScript, where developers write code in another language which is then converted to JavaScript. This conversion occurs either at runtime or during a separate generation stage. Notable JavaScript generators include:

- **Cappuccino**—See [http://cappuccino.org/](http://cappuccino.org/). Cappuccino uses Objective-J, a clone of the Objective-C language from Mac OS X. Cappuccino itself is a port of the Cocoa application framework, again from OS X.

Given that Google uses GWT for Blogger, Google Groups, and many other sites, we can safely say that generated JavaScript SPAs are widely used. This raises the question: **why bother writing in one high-level language and then converting it to another?**

Here are a number of reasons generated JavaScript remains popular, and why these reasons aren’t as compelling as they once were:

- **Familiarity**—The developers can use a more familiar or simpler language. The generator and framework allows them to develop without having to learn the vagaries of JavaScript. The problem is that something eventually gets lost in translation. When this happens, the developers have to inspect the generated JavaScript and understand it to get things to work right. We feel we’re more effective when we work directly in JavaScript instead of working through a language abstraction layer.
- **Framework**—The developers appreciate that GWT provides the cohesive system of matching libraries built for server and client. This is a persuasive argument, particularly if the team already has a lot of expertise and products that are in production.
- **Multiple targets**—The developers can have the generator write for multiple targets, such as one file for Internet Explorer and one for the rest of the world’s browsers. Although generating code for different targets sounds nice, we think it’s even more effective to deploy a single JavaScript source for all browsers. Thanks to converging browser implementations and mature cross-browser libraries like jQuery, it’s now much easier to write a sophisticated SPA that runs across all major browsers without modification.
1.1.3 Our focus

This book shows how to develop engaging, robust, scalable, and maintainable SPAs using JavaScript end to end. Unless otherwise noted, when we refer to an SPA from this point forward, we mean a native JavaScript SPA, where the business and presentation logic is written directly in JavaScript and executed by the browser. This JavaScript renders the interface using browser technologies such as HTML5, CSS3, Canvas, or SVG.

SPAs can use any number of server technologies. Because so much of the web application moves to the browser, the server requirements are often significantly reduced. Figure 1.2 illustrates how the business logic and generation of HTML migrates from the server to the client.

We believe native JavaScript SPAs are usually the better choice today. And that’s what we design and build in this book.

Figure 1.2 Responsibilities of the database, server, and client

---

Another title for this book might have been Building Single Page Web Applications Using Best Practices. But that seemed too wordy.
We focus on the backend in chapters 7 and 8, where we use a web server and database with JavaScript as their control languages. You may not have this choice or may prefer a different backend. That’s okay—most of the SPA concepts and techniques we use in this book work well regardless of what backend technologies you use. But if you want to use JavaScript end-to-end, we’ve got you covered.

Our client libraries include jQuery for DOM manipulation with plugins for history management and event handling. We use TaffyDB2 to provide high-performance, data-centric models. Socket.IO provides seamless near-real-time messaging between the web server and the client. On the server, we use Node.js for our event-based web server. Node.js uses the Google V8 JavaScript engine and excels at handling tens of thousands of concurrent connections. We also use Socket.IO on the web server. Our database is MongoDB, a noSQL database that uses the JavaScript native data format, JSON, to store data and also has a JavaScript API and command-line interface. All of these are proven and popular solutions.

SPA development requires JavaScript coding at a scale at least an order of magnitude greater than a traditional website, as much of the application logic moves from the server to the browser. The development of a single SPA may require many developers to code concurrently and may result in well over 100,000 lines of code. Conventions and discipline previously reserved for server-side development become a must for working at this scale. On the other hand, the server software is simplified and relegated to authentication, validation, and data services. Keep this in mind as we proceed through our examples.

1.2 **Build our first SPA**

It’s now time to develop an SPA. We’ll use best practices and explain them as we go.

1.2.1 **Define the goal**

Our first SPA will have the modest goal of providing a chat slider at the bottom right of the browser window, similar to one you might see on Gmail or Facebook. When we load the application, the slider will be retracted; when we click on the slider, it’ll extend, as shown in figure 1.3. Clicking again will retract it.
SPAs usually do many other things besides opening and closing a chat slider—like sending and receiving chat messages. We’ll omit such pesky details to keep this introduction relatively simple and brief. To pervert a famous saying, one can’t conquer SPAs in a day. Fear not, we’ll return to sending and retrieving messages in chapters 6 and 8.

In the next few sections, we’ll set up a file for SPA development, introduce some of our favorite tools, develop the code for the chat slider, and highlight some best practices. We’ve given you a lot to absorb here, and you’re not expected to understand everything right now—particularly some of the JavaScript tricks we’re using. We’ll have a lot more to say about each of these topics in the next few chapters, but for now, relax, don’t sweat the small stuff, and take in the lay of the land.

### 1.2.2 Start the file structure

We’ll create our application in a single file, spa.html, using only jQuery as our one external library. Usually, it’s better to have separate files for CSS and JavaScript, but starting with a single file is handy for development and examples. We start by defining where we’ll place our styles and our JavaScript. We’ll also add a `<div>` container where our application will write HTML entities, as shown in listing 1.1:

```html
<!doctype html>
<html>
<head>
    <title>SPA Chapter 1 section 1.2.2</title>
    <style type="text/css"></style>
    <script type="text/javascript"></script>
</head>
<body>
    <div id="spa"></div>
</body>
</html>
```

Now that we have the file ready, let’s set up Chrome Developer Tools to inspect the application in its current state.

### 1.2.3 Set up Chrome Developer Tools

Let’s use Google Chrome to open our listing—spa.html. We should see a blank browser window, because we haven’t added any content. But activities are going on under the hood. Let’s use Chrome Developer Tools to inspect them.

We can open Chrome Developer Tools by clicking on the wrench in the upper-right corner of Chrome, selecting Tools, and then Developer Tools (Menu > Tools > Developer Tools). This will display the Developer Tools, as shown in figure 1.4. If we don’t see the JavaScript console, we can display it by clicking on the Activate console button at the bottom left. The console should be blank, which means we have no
Build our first SPA

JavaScript warnings or errors. This is good, because currently we have no JavaScript. The Elements section above the console shows the HTML and structure of our page.

Although we use Chrome Developer Tools here and throughout the book, other browsers have similar capabilities. Firefox, for example, has Firebug, and both IE and Safari provide their own version of Developer Tools.

When we present listings in this book, we’ll often use the Chrome Developer Tools to ensure our HTML, CSS, and JavaScript all play nicely together. Now let’s create our HTML and CSS.

1.2.4 Develop the HTML and CSS

We’ll need to add a single chat slider container to our HTML. Let’s begin by styling the containers in the <style> section in the spa.html file. The adjustments to the <style> section are shown in the following listing:

```
<!doctype html>
<html>
<head>
<title>SPA Chapter 1 section 1.2.4</title>
<style type="text/css">
body {
  width   : 100%;
  height   : 100%;
  overflow : hidden;
  background-color : #777;
}
</style>
</head>
<body>
</body>
</html>
```

Figure 1.4 Google Chrome Developer Tools

Define the <body> tag to fill the entire browser window and hide any overflow. Set the background color to mid-gray.
When we open spa.html in our browser, we should see the slider retracted, as shown in figure 1.5. We’re using a liquid layout where the interface adapts to the display size and the slider always stays anchored at the bottom-right corner. We didn’t add any borders to our containers because they add to container width and can impede development, as we have to resize containers to accommodate those borders. It’s handy to add borders after the basic layout is created and verified, as we do in later chapters.

Now that we have the visual elements in place, it’s time to use JavaScript to make the page interactive.

1.2.5 Add the JavaScript

We want to employ best practices with our JavaScript. One tool that will help is JSLint, written by Douglas Crockford. JSLint is a JavaScript validator that ensures that our code doesn’t break many sensible JavaScript best practices. And we also want to use jQuery, a Document Object Model (DOM) toolkit written by John Resig. jQuery provides simple cross-browser tools to easily implement the slider animation.
Before we get into writing the JavaScript, let’s outline what we want to do. Our first script tag will load the jQuery library. Our second script tag will contain our JavaScript which we’ll break into three parts:

1. A header that declares our JSLint settings.
2. A function called `spa` that creates and manages the chat slider.
3. A line to start the `spa` function once the browser’s Document Object Model (DOM) is ready.

Let’s take a closer look at what we need the `spa` function to do. We know from experience that we’ll want a section where we declare our module variables and include configuration constants. We’ll need a function that toggles the chat slider. And we’ll need a function that receives the user click event and calls the toggle function. Finally, we’ll need a function that initializes the application state. Let’s sketch an outline in more detail:

Listing 1.3 JavaScript development, first pass—spa.html

```plaintext
/* jslint settings */
// Module /spa/
// Provides chat slider capability
// Module scope variables
// Set constants
// Declare all other module scope variables
// DOM method /toggleSlider/
// alternates slider height
// Event handler /onClickSlider/
// receives click event and calls toggleSlider
// Public method /initModule/
// sets initial state and provides feature
// render HTML
// initialize slider height and title
// bind the user click event to the event handler

// Start spa once DOM is ready
```

This is a good start! Let’s keep the comments just as they are and add our code. We have kept the comments in bold for clarity.

Listing 1.4 Javascript development, second pass—spa.html

```javascript
/* jslint settings */

var spa = (function ( $ ) {

    // Module scope variables
    var configMap = { },
    // Declare all other module scope variables

    // Set constants
    configMap = { },
    // Declare all other module scope variables
```
CHAPTER 1  Our first single page application

```javascript
chapSlider, toggleSlider, onClickSlider, initModule;

// DOM method /toggleSlider/
// alternates slider height
//
toggleSlider = function () {};

// Event handler /onClickSlider/
// receives click event and calls toggleSlider
//
onClickSlider = function ( event ) {};

// Public method /initModule/
// sets initial state and provides feature
//
initModule = function ( $container ) {
  render HTML
  // initialize slider height and title
  // bind the user click event to the event handler
};

// Start spa once DOM is ready

Now let’s make a final pass at spa.html as shown in listing 1.5. We load the jQuery library and then we include our own JavaScript, which has our JSLint settings, our spa module, and a line to start the module once the DOM is ready. The spa module is now fully functional. Don’t worry if you don’t “get” everything right away—there’s lots to take in here, and we’ll be covering everything in more detail in upcoming chapters. This is just an example to show you what can be done:

Listing 1.5  JavaScript development, third pass—spa.html

```
<!doctype html>
<html>
<head>
  <title>SPA Chapter 1 section 1.2.5</title>
  <style type="text/css">
...
  </style>

  <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
  </script>

  <script type="text/javascript">
  /*jslint
  browser : true, continue : true,
  devel : true, indent : 2, maxerr : 50,
  newcap : true, nomen : true, plusplus : true,
  regexp : true, sloppy : true, vars : true,
  white : true
  */
  /*global jQuery */

  // Module /spa/
  ```
// Provides chat slider capability

// Module scope variables
var

// Set constants
configMap = {
  extended_height : 434,
  extended_title : 'Click to retract',
  retracted_height : 16,
  retracted_title : 'Click to extend',
  template_html : '<div class="spa-slider">\</div>'
},

// Declare all other module scope variables
$chatSlider,
toggleSlider, onClickSlider, initModule;

// DOM method /toggleSlider/
// alternates slider height
//
toggleSlider = function () {
  var
    slider_height = $chatSlider.height();

  // extend slider if fully retracted
  if ( slider_height === configMap.retracted_height ) {
    $chatSlider
      .animate({ height : configMap.extended_height })
      .attr( 'title', configMap.extended_title );
    return true;
  }

  // retract slider if fully extended
  else if ( slider_height === configMap.extended_height ) {
    $chatSlider
      .animate({ height : configMap.retracted_height })
      .attr( 'title', configMap.retracted_title );
    return true;
  }

  // do not take action if slider is in transition
  return false;
}

// Event handler /onClickSlider/
// receives click event and calls toggleSlider
//
onClickSlider = function ( event ) {
  toggleSlider();
  return false;
};

// Public method /initModule/
// sets initial state and provides feature
//
initModule = function ( $container ) {
  // render HTML

Package our code into the spa namespace. More details on this practice are provided in chapter 2.

Declare all variables before they are used. Store module configuration values in configMap and state values in stateMap.

Add the code to extend the chat slider. It inspects the slider height to determine if it’s fully retracted. If so, it uses a jQuery animation to extend it.

Add the code to retract the chat slider. It inspects the slider height to determine if it’s fully extended. If so, it uses a jQuery animation to retract it.

Group all event handler methods in a section. It is good practice to keep the handlers small and focused. They should call other methods to update the display or adjust business logic.


Group all public methods in a section.
Don’t worry too much about JSLint validation, as we’ll detail its use in coming chapters. But we’ll cover a few noteworthy concepts now. First, the comments at the top of the script set our preferences for validation. Second, this script and settings pass validation without any errors or warning. Finally, JSLint requires that functions be declared before they’re used, and therefore the script reads “bottom up” with the highest level functions at the end.

We use jQuery because it provides optimized, cross-browser utilities for fundamental JavaScript features: DOM selection, traversal, and manipulation; AJAX methods; and events. The jQuery $(selector).animate(...) method, for example, provides a simple way to do something that’s otherwise quite complex: animate the height of the chat slider from retracted to extended (and vice versa) within a specified time period. The motion starts slowly, accelerates, and then slows to a stop. This type of motion—called easing—requires knowledge of frame-rate calculations, trigonometric functions, and the vagaries of implementation across popular browsers. If we wrote it ourselves, it would require dozens of additional lines.

The $(document).ready(function) also saves us a lot of work. It runs the function only after the DOM is ready for manipulation. The traditional way to do this was to use the window.onload event. For a variety of reasons, window.onload isn’t an efficient solution for more demanding SPAs—although it makes little
difference here. But writing the correct code to use across all browsers is painfully tedious and verbose.  

jQuery’s benefits, as the previous example shows, usually significantly outweigh its costs. In this case, it shortened our development time, reduced the length of our script, and provided robust cross-browser compatibility. The cost of using it is somewhere between low and negligible, as its library is small when minimized and users likely have it already cached on their devices anyway. Figure 1.6 shows the completed chat slider.

Now that we’ve completed the first implementation of our chat slider, let’s look at how the application actually works using the Chrome Developer Tools.

1.2.6 Inspect our application using Chrome Developer Tools

If you’re comfortable using Chrome Developer Tools, you may skip this section. If not, we highly encourage you to play along at home.

Let’s open our file, spa.html, in Chrome. After it loads, let’s immediately open up the Developer Tools (Menu > Tools > Developer Tools).

The first thing you may notice is how the DOM has been changed by our module to include the `<div class="spa-slider" ... >` element, as shown in figure 1.7. As we continue, our application will be adding a lot more dynamic elements like this one.

Figure 1.6 The completed chat slider in action—spa.html

Figure 1.7 Inspecting the elements—spa.html

---

6 See [www.javascriptkit.com/dhtmltutors/domready.shtml](http://www.javascriptkit.com/dhtmltutors/domready.shtml) to get a taste of the pain.
We can explore the JavaScript execution by clicking on the Sources button in the top menu of the Developer Tools. Then select the file that contains the JavaScript, as shown in figure 1.8.

In later chapters we’ll be placing our JavaScript into separate files. But for this example it’s in our HTML file as shown in figure 1.9. We’ll need to scroll down to find the JavaScript we want to inspect.

When we navigate to line 76, we should see an `if` statement, as shown in figure 1.10. We should like to inspect the code before this statement is executed, so we click on the left margin to add a breakpoint. Whenever the JavaScript interpreter reaches this line in the script, it’ll pause so we can inspect elements and variables to better understand what’s happening.
Now let’s go back to the browser and click on the slider. We’ll see that the JavaScript has paused at the red arrow at line 76, as in figure 1.11. While the application is paused, we can inspect variables and elements. We can open the console section and type in various variables and press Return to see their values in this paused state. We see that the if statement condition is true (slider_height is 16, and configMap.retracted_height is 16), and we can even inspect complex variables like the configMap object, as shown at the bottom of the console. When we’re done inspecting, we can remove the breakpoint by clicking on the left margin of line 76, and then clicking the Resume button at the top right (above Watch Expressions).

Once we click Resume, the script will continue from line 76 and finish toggling the slider. Let’s return to the Elements tab and look at how the DOM has changed, as shown in figure 1.12. In this figure we can see that the CSS height property, which was provided by the spa-slider class (see Matched CSS Rules on the lower right), has been overridden by an element style (element styles have higher priority over styles that come from classes or IDs). If we click on the slider again, we can watch the height change in real-time as the slider retracts.
Our short introduction to Chrome Developer Tools shows only a small portion of their ability to help us understand and change what’s occurring “under the hood” of our application. We’ll continue to use these tools as we develop this application, and we recommend you spend some quality time with the online manual at http://mng.bz/PzIJ. It’s time well spent.

1.3 The user benefits of a well-written SPA

Now that we’ve built our first SPA, let’s consider the primary benefit of an SPA over a traditional website: it provides a substantially more engaging user experience. An SPA can deliver the best of both worlds: the immediacy of a desktop application and the portability and accessibility of a website.

- **An SPA can render like a desktop application**—The SPA redraws the parts of the interface that need to change only as needed. A traditional website, in comparison, redraws the entire page on many user actions, resulting in a pause and a “flash” while the browser retrieves from the server and then redraws everything on the page. If the page is large, the server is busy, or the internet connection is slow, this flash can take several seconds or more, and the user has to guess when the page is ready to use again. This is a horrible experience when compared to the rapid rendering and immediate feedback of an SPA.

- **An SPA can respond like a desktop application**—The SPA minimizes response time by moving working (transient) data and processing from the server to the browser as much as possible. The SPA has the data and business logic needed to make most decisions locally and therefore quickly. Only data validation, authentication, and permanent storage must remain on the server, for reasons we discuss in chapters 6-8. A traditional website has most of the application logic on the server and the user must wait for a request/response/redraw cycle in response to much of their input. This can take several seconds, compared to the near immediate response of the SPA.

- **An SPA can notify users of its state like a desktop application**—When an SPA does have to wait on a server, it can dynamically render a progress bar or busy indicator so the user isn’t befuddled by a delay. Compare this to a traditional website, where the user actually has to guess when the page is loaded and usable.

- **An SPA is nearly universally accessible like a website**—Unlike most desktop applications, users can access an SPA from any web connection and a decent browser. Today, the list includes smart phones, tablets, televisions, laptops, and desktop computers.

- **An SPA can be instantly updated and distributed like a website**—The user doesn’t have to do anything to realize the benefits—when they reload the browser it works. The hassle of maintaining multiple concurrent versions of software is largely eliminated. But not completely: what happens if the server-client data exchange format changes, yet many users have the prior version of software loaded in their browser? This can be accommodated with some forethought.
updated multiple times in a single day. Desktop applications often require a download and administrative access to install a new version, and the interval between versions can be many months or years.

- **An SPA is cross-platform like a website**—Unlike most desktop applications, a well-written SPA can work on any operating system that provides a modern HTML5 browser. Though usually this is considered a developer benefit, it’s extremely useful for many users who have a combination of devices—say Windows at work, a Mac at home, a Linux server, an Android phone, and an Amazon tablet.

All of these benefits mean that you may want to make your next application an SPA. Clunky websites that re-render an entire page after each click tend to alienate increasingly sophisticated users. The communicative and responsive interface of a well-written SPA, along with the accessibility of the internet, helps keep our customers where they belong—using our product.

### 1.4 Summary

The single page application has been around for some time. Flash and Java have, until recently, been the most widely used SPA client platforms because their capability, speed, and consistency exceeded those of JavaScript and browser rendering. But recently, JavaScript and browser rendering have reached a tipping point where they’ve overcome their most troublesome deficiencies while providing significant advantages over other client platforms.

We focus on creating SPAs using native JavaScript and browser rendering, and when we refer to an SPA we mean a native JavaScript SPA unless otherwise noted. Our SPA tool chain includes jQuery, TaffyDB2, Node.js, Socket.IO, and MongoDB. All of these are proven, popular solutions. You may choose to employ alternatives to these technologies, but the fundamental structure of an SPA would remain regardless of specific technology decisions.

The simple chat slider application we developed demonstrates many features of a JavaScript SPA. It responds immediately to user input, and it uses data stored in the client instead of the server to make decisions. We used JSLint to ensure that our application didn’t contain common JavaScript mistakes. And we used jQuery to select and animate the DOM and to handle the event when a user clicks on the slider. We explored using the Chrome Developer Tools to help us understand how our application was working.

An SPA can provide the best of both worlds—the immediacy of a desktop application and the portability and accessibility of a website. The JavaScript SPA is available on over a billion devices that support a modern web browser and that don’t require any proprietary plugins. With a little effort, it can support desktops, tablets, and smart phones running many different operating systems. SPAs are easily updated and distributed, usually without requiring any action from the user. All of these benefits explain why you may want to make your next application an SPA.
In the next chapter, we’ll explore some key JavaScript concepts that are needed for SPA development, but are frequently ignored or misunderstood. We’ll then build on this foundation to improve and extend the example SPA we developed in this chapter.
If your website is a jumpy collection of linked pages, you are behind. Single page web applications are your next step: pushing UI rendering and business logic to the browser and communicating with the server only to synchronize data, they provide a smooth user experience, much like a native application. But, SPAs can be hard to develop, manage, and test.

**Single Page Web Applications** shows how your team can easily design, test, maintain, and extend sophisticated SPAs using JavaScript end-to-end, without getting locked into a framework. Along the way, you’ll develop advanced HTML5, CSS3, and JavaScript skills, and use JavaScript as the language of the web server and the database.

**What’s Inside**

- Design, build, and test a full-stack SPA
- Best-in-class tools like jQuery, TaffyDB, Node.js, and MongoDB
- Real-time web with web sockets and Socket.IO
- Touch controls for tablets and smartphones
- Common SPA design mistakes

This book assumes basic knowledge of web development. No experience with SPAs is required.

The authors are architects and engineering managers. **Michael Mikowski** has worked on many commercial SPAs and a platform that processes over 100 billion requests per year. **Josh Powell** has built some of the most heavily trafficked sites on the web.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SinglePageWebApplications

"Insights from generations of SPA refinement."
—From the Foreword by Gregory D. Benson

"Thorough, comprehensive, and methodical."
—Mark Ryall, ThoughtWorks

"Essential reading, even if you’re using a framework."
—Ken Rimple
Author of *Spring Roo in Action*

"I highly recommend the techniques outlined here."
—Jason Kaczor, SharePoint MVP

"An excellent guide."
—Mike Greenhalgh, NHS Wales