THE EXPERT'S VOICE® IN SHAREPOINT

Custom SharePoint Solutions with HTML and JavaScript

For SharePoint 2013 and SharePoint Online

RAPID, RELIABLE SHAREPOINT SOLUTIONS

Brandon Atkinson

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

SharePoint has come a very long way over the years. Development in SharePoint has had just as long a journey, often much more bumpy. There are still many highly skilled developers who would rather chew their own arm off than become a SharePoint developer! However, SharePoint is not going anywhere anytime soon, and with the popularity of Office 365 and SharePoint Online growing every day, developers will find themselves being asked to create custom solutions more often. In the past this would have meant installing SharePoint on your PC, learning new APIs that were unfamiliar, getting up to speed on deployment processes, and more. You could easily look at those hurdles and become frustrated, which is common. Luckily, with SharePoint 2013 and SharePoint Online you have more options.

Starting with SharePoint 2013, Microsoft greatly enhanced what could be done in the browser with JavaScript. This included a brand-new REST API and an expanded set of JavaScript libraries utilizing the JavaScript Object Model, or JSOM for short. These were originally built for use with Apps for SharePoint, which is a new development model in SharePoint 2013. Apps allow you to write code that targets SharePoint and then deploy it somewhere else. You could write custom solutions and deploy them to the cloud, and your end users would never know the difference. This was great because it gave developers more freedom and kept custom code off the SharePoint servers.

While JSOM and the REST API are great for Apps, there usage does not need to end there. These tools can easily be consumed via JavaScript running in the browser. This opens a whole new world of development possibilities to not only seasoned professional developers, but also the power users who know HTML and JavaScript. This means that end users who are comfortable using HTML and JavaScript can begin building some of their own solutions. Even better, with a very small learning curve, developers can utilize a skillset that they already know. Gone are the days of being limited to building custom solutions in Visual Studio and C#; now you have the option to build robust solutions that run directly in the browser!

This book will explore some of the various ways that you can utilize out-of-the-box components and custom code to build solutions for SharePoint. By the time you finish, you will have created many custom web parts that you probably thought were only in the realm of professional developers using expensive and complicated tools. If you are a developer, you'll see that SharePoint development is no longer something overly complicated and to be avoided. No matter your station, I hope that you will come out the other side with the confidence and knowledge to begin building your own custom solutions in SharePoint.

CHAPTER 1

Setting Up Your Environment

Setting up your environment properly will give you all the tools needed to build your custom solutions. A lot of what you will need is already built into SharePoint and is ready to use. However, you will also want to download and install some extra tools that will make your life easier. These tools include:

- SharePoint Online Developer Site
- SharePoint Designer
- Visual Studio Express
- Various browsers and developer tools
- jQuery
- Implementing jQuery in your SharePoint environment

If this list looks daunting, don't be concerned. In this chapter we'll explore each of the items in detail to get you familiar with them and what they are used for. In addition, we'll take a look at creating your own SharePoint Online environment to use as a development area and show you how to implement jQuery in that environment.

If you've been developing SharePoint solutions already and even just doing web development, you'll most likely already have these installed. This chapter will be geared more toward the business user or those newer to front-end development. So if you're a seasoned developer, you can skim this chapter to ensure you have everything you need. No matter who you are, at the end of this chapter you will be fully prepared to start the journey of building custom solutions in SharePoint!

SharePoint Online Developer Site

It goes without saying that being able to build custom SharePoint solutions is a wasted skill without somewhere to build them! Some of you will already have an environment to use for development. This may be a SharePoint Online environment that your business uses for their intranet, or perhaps a SharePoint 2013 on-premise environment. It may be that you're looking to learn SharePoint development for the first time and you have nowhere to work. In any case, you have plenty of options available to you and we'll explore those here.

Note The options explored in this section will focus on SharePoint Online and Office 365. If you have SharePoint installed on-premises, you can easily create a new site to work in. Even if you have SharePoint at your fingertips at work, you may still want your own environment to work in so that the options here will still apply.

Microsoft has made it very easy to get started with front-end development on the latest versions of SharePoint. You can sign up for an Office 365 Developer Site at the following URL: http://msdn.microsoft.com/en-us/library/office/fp179924(v=office.15).aspx. Figure 1-1 shows the page and the various options for signing up.

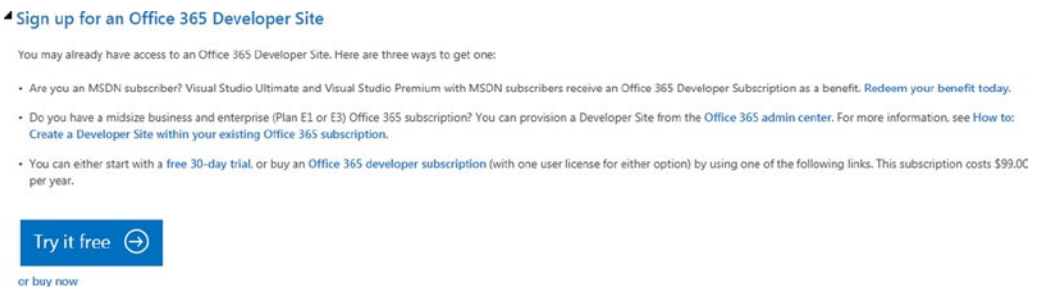

Figure 1-1. *Office 365 Developer Site sign-up options on the Microsoft site*

As you can see, there are three options for creating a developer site:

- *MSDN subscriber* - If you or your company subscribes to MSDN, you can create a developer site for free as part of that benefit. A lot of companies with dedicated developers on staff have MSDN subscriptions, or you may have one on your own. In either case, simply click the link titled "Redeem your benefit today" to get started.

  If you're a business user, it may be worth looking into this option as well. If your company has an MSDN subscription, you could either request a license or have someone in IT create the site for you and provide access.

- *Office 365 subscriber* - Businesses that already subscribe to an E3 or E4 Office 365 Plan can provision a developer site right from the Office 365 admin center. These subscription levels allow for creating a new private site collection using a special site template for developers.

  For full details and a walk-through of creating a Developer Site, visit the following URL: http://msdn.microsoft.com/en-us/library/office/jj692554(v=office.15).aspx.

- *30-day trial or developer subscription* - Microsoft also allows you a 30-day trial of Office 365 if you wanted or needed to have your own environment. This is a great option, as it gives you a good amount of time to explore Office 365 and SharePoint Online, as well as to test your development skills in an environment that is all yours.

  You can also upgrade the trial to an annual subscription, at any time, for $99.00 USD. This is a great option, even if you have SharePoint at your business. It gives you a perfect area—away from any production sites—where you can build your custom solutions. At $99 a year, it offers a low-cost option to build solutions that can be easily migrated to a production environment.

No matter your situation, you can easily obtain an environment where you can develop your custom SharePoint solutions. Now that we've covered where and how to obtain your SharePoint environment, let's explore some of the tools we'll be using in this book.

■ **Note** All of the examples and screenshots in this book will be taken from a SharePoint Online environment, which is the latest version of SharePoint available.

SharePoint Designer

If you've used SharePoint in the past, then you've probably used SharePoint Designer as well. It's a free tool provided by Microsoft and a staple workhorse when it comes to SharePoint. It allows you to perform a wide range of tasks—from branding, to connecting, to external databases, building lists, and more. For this book, Designer honestly does not play a huge role, but it will be key in helping us implement jQuery in SharePoint. This is at the end of this chapter. For now, let's just download Designer.

You can download SharePoint Designer at the following URL: `http://www.microsoft.com/en-us/download/details.aspx?id=35491`. Figure 1-2 shows the page at the Microsoft Download Center site where you can download Designer.

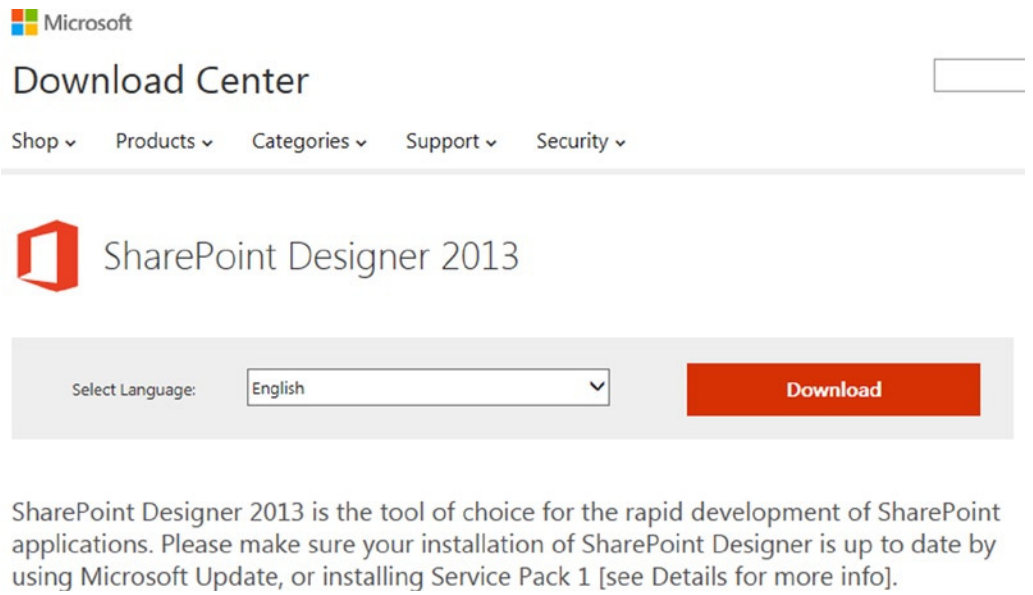

Figure 1-2. *SharePoint Designer download page on the Microsoft site*

For now, you can simply download and install Designer; we'll get to using it later on. Even though we don't dive deeply into Designer in this book, you will find it a useful tool when working with SharePoint. As I stated before, it has a wide range of capabilities and can assist you with many tasks. Designer works with both SharePoint Online and on-premises environments.

Visual Studio Express

If you're not a developer, then you may not have heard of Visual Studio Express; and if you have, it probably seems like an overwhelming program for code jockeys. In a way, it is, but it's a powerful tool that we'll be using to write our HTML and JavaScript. It's also free and provided by Microsoft.

Visual Studio is an integrated development environment, or IDE for short. This is a fancy way of saying "a program that allows you to write code." For this book, all of our examples will be in HTML and JavaScript. One of the nice things about HTML and JavaScript is that they are simply text files with special extensions: `.html` and `.js`. To that end, all you really need to create and edit these files is Notepad. However, while Notepad may seem friendly and unimposing, it is in no way your friend when it comes to writing code. Visual Studio Express provides all the tools you need—and will even let you know when it detects something is wrong. Notepad can't do that!

Visual Studio comes in a lot of different versions, depending on how technical your need is. The Express edition is a fully featured version and will more than meet the needs of this book. You can download Visual Studio Express at the following URL: `http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx`. Figure 1-3 shows the download page at the Visual Studio site.

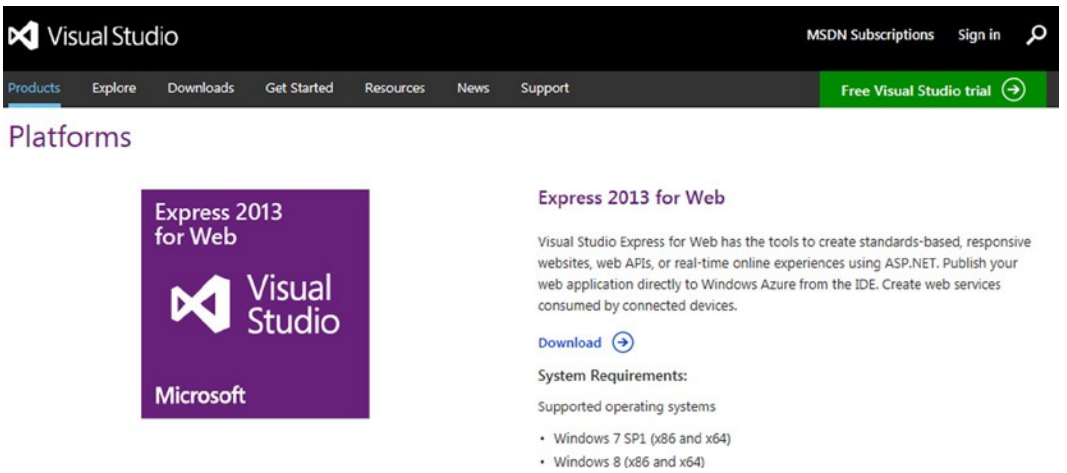

Figure 1-3. *Visual Studio Express download page on the Microsoft site*

Visual Studio Express comes in multiple editions, including Web, Windows, and more. Express for Web is a perfect version for building solutions using HTML and JavaScript.

Browsers and Developer Tools

When it comes to working with SharePoint as an end user, the obvious browser choice is Internet Explorer. While SharePoint works excellently with other browsers, such as Firefox and Chrome, Internet Explorer is clearly the Microsoft-recommended browser and what SharePoint is most likely to be friendly with. That being said, this book is all about building custom solutions using HTML and JavaScript, and you will find that most modern browsers act and perform the same. Before we talk about browsers, let's discuss developer tools.

Developer tools is a generic term, meaning the tools used to help develop, debug, and otherwise know what's going on with the HTML, JavaScript, and CSS that is in your browser. These can also be called *developer toolbars*, *browser tools*, or may go by an entirely different name altogether, like Firebug. No matter the tool, they all perform the same function, which is to help you develop your solutions. In just about every browser, all you need to do to launch the developer tools is hit the F12 key on your keyboard, as shown in Figure 1-4 in Internet Explorer.

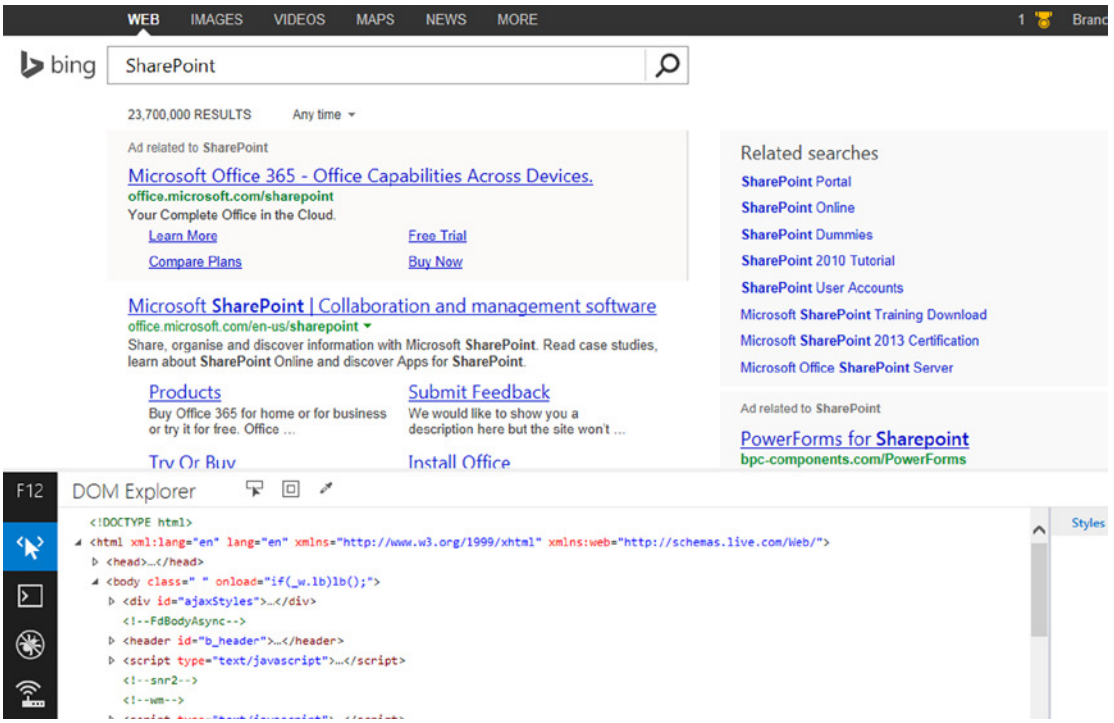

Figure 1-4. *Developer tools launched in Internet Explorer after hitting F12*

As you can see in Figure 1-4, these tools provide a lot of insight into what is going on in the browser. Immediately you can see the HTML that is generating the page, along with any CSS styling that has been applied. Using the menu on the left side of the tools, you can also look at what JavaScripts were loaded on the page, and even view the code inside them. You can take it one step further and actually set "breakpoints" in the JavaScript to pause its execution so that you can see what is happening when those scripts are working. This is an invaluable tool in your front-end development efforts and it will be used often.

Figure 1-5 shows the developer tools open in Google Chrome. Again, pressing F12 is all that is needed to open the tools. As with Internet Explorer, you immediately see the HTML and styles applied to the page.

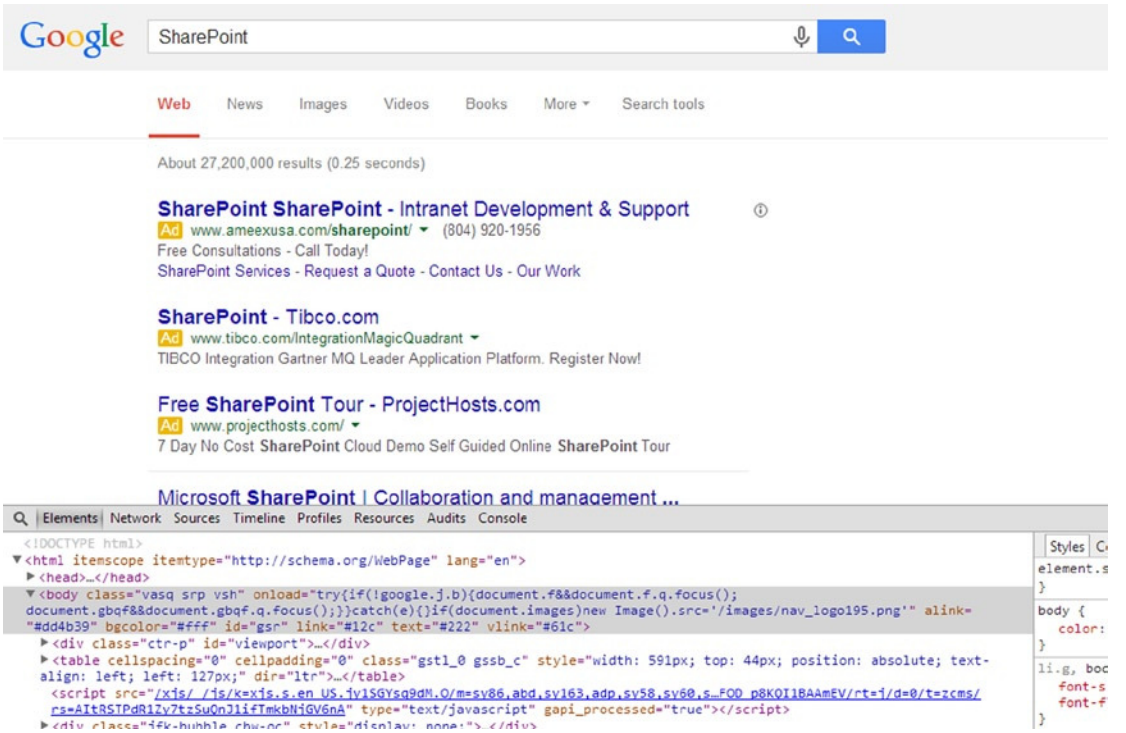

Figure 1-5. *Developer tools launched in Google Chrome after hitting F12*

In Chrome, the menu options run across the top of the tools screen and provide similar functionality. You can close developer tools the same way you open them, by pressing F12. You should download several browsers and get a feel for their tools. Every developer has his or her preferences for browsers and tools; mine are Firefox and Firebug, and we'll be using these for this book.

I'll go on record and say that I think Internet Explorer is a fantastic browser, at least the latest version is. It has come a long way over the years and is finally on par with some of the newer kids on the block. The developer tools in Internet Explorer are very good, but I prefer using Firebug in Firefox over all the rest. Firebug is an Add-on in Firefox that performs just like the tools in the other browsers, but in my opinion it organizes the content in a friendlier manner and is easier to use. You should definitely find your preference and use the browser/tools that work best for you. All the examples in this book will use Firebug, but you are not limited to those tools.

To install Firebug, download the Firefox browser at the following URL: `https://www.mozilla.org/firefox/`. Once installed, click the menu icon in the top-right corner. Figure 1-6 shows the menu open and the Add-ons option highlighted.

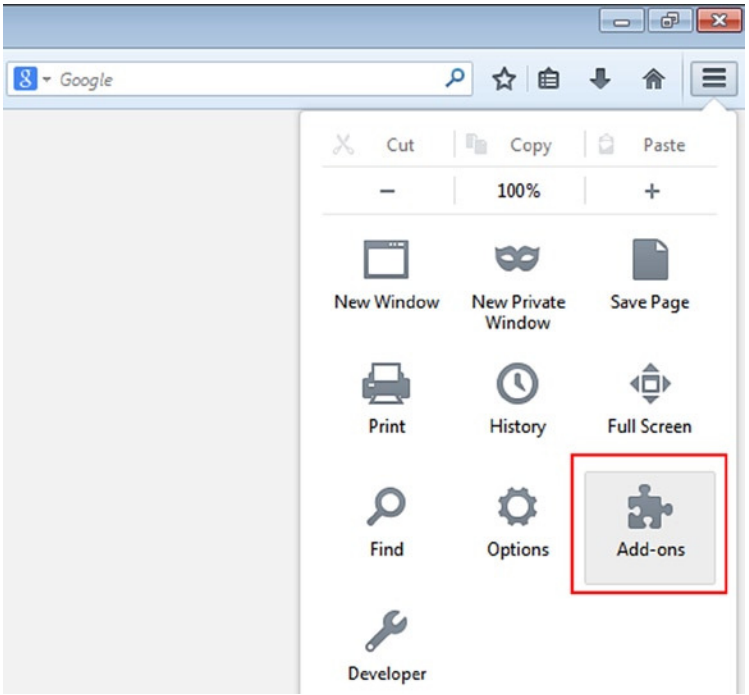

Figure 1-6. *The Add-ons menu option in the Firefox menu*

Clicking the Add-ons option will open the Add-ons Manager page in a new tab. This page allows you to manage all your Add-ons and search for new ones. Type **firebug** in the search box at the top-right of the screen, and then click the magnifying glass icon. Figure 1-7 shows the search box after typing into it.

Figure 1-7. *Search box on the Add-ons Manager page before searching*

The search results bring back any Add-ons matching your request, and Firebug should be at the top of the list. Click the Install button shown in Figure 1-8 to add Firebug to Firefox.

Figure 1-8. *Clicking the Install button will install Firebug in Firefox*

Once the download and installation are complete, Firefox will now display a "bug" icon next to the menu in the top-right corner. Figure 1-9 highlights the Firebug icon.

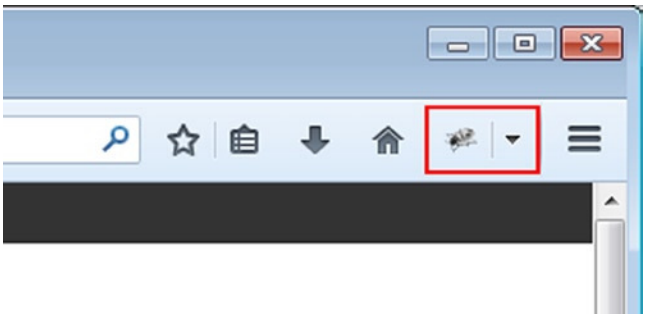

Figure 1-9. *Firebug icon is present once the installation is complete*

Now that Firebug has been installed, you can open it the same way you open developer tools in other browsers—by hitting the F12 key on your keyboard. You can also click the Firebug icon to open it as well, as shown in Figure 1-10.

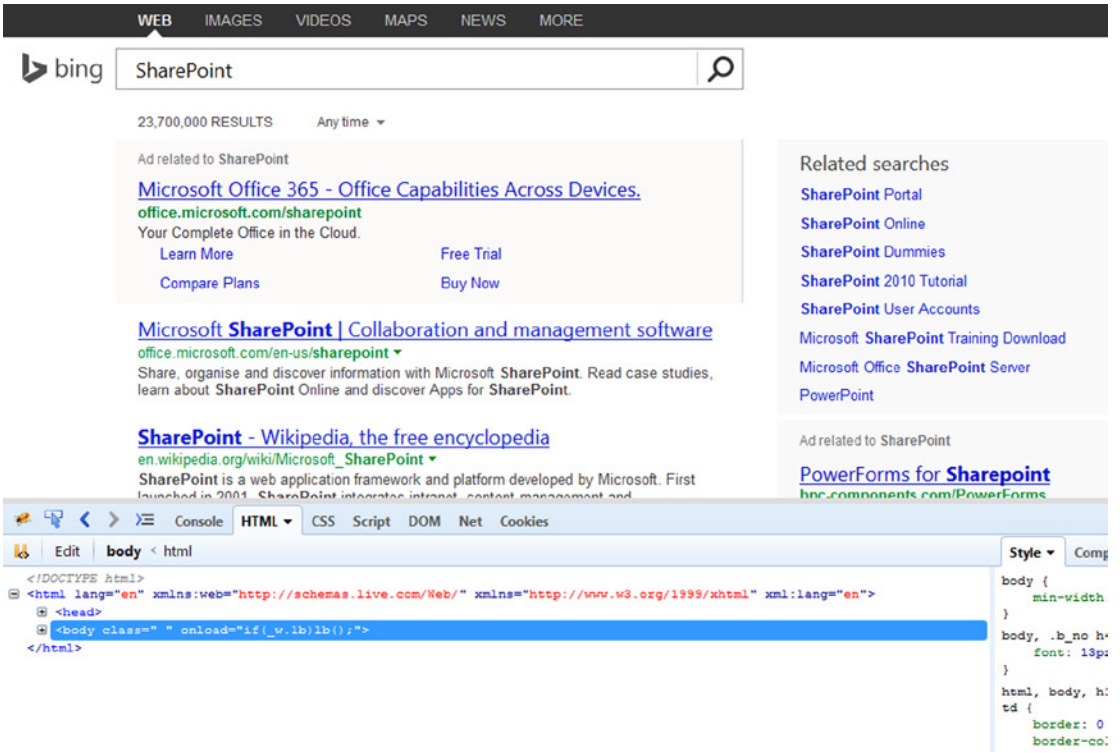

Figure 1-10. *Firebug toggled open by clicking F12*

As you can see in Figure 1-10, Firebug looks and acts a lot like the developer tools available in other browsers. Again, it's your personal preference as to which tools you ultimately decide to use. All the examples in this book will use Firefox and Firebug.

jQuery

If you've done any JavaScript work in the past, you've probably felt a bit frustrated from time to time. JavaScript is a great framework for front-end development. It's fast and you can build useful code with it in no time. But it can be difficult to work with, hard to debug at times, and appear intimidating to newcomers. jQuery came along and changed all that.

jQuery is a framework that works along with JavaScript to make writing front-end code easier and faster. jQuery bills itself as the "write less, do more" framework—and you will find that this is the case. It has quickly become a standard for JavaScript development, and a lot of the examples in this book will utilize jQuery in some way. If you've never used it before, don't worry: we'll start slow and get you up to speed.

■ **Note** If you are new to jQuery, there is a great "getting started" guide on the jQuery web site at `http://learn.jquery.com/about-jquery/how-jquery-works/`. There is also *Beginning jQuery* by Jack Franklin (Apress, 2013), a great resource for those of you who are getting started (see `www.apress.com/9781430249320`).

Now, let's do a quick jQuery exercise to get you familiar with how it works. One of the main benefits of jQuery is being able to easily select HTML elements on the page. The two main ways you can do this are by using an element's ID or its CSS class. Figure 1-11 shows the jQuery home page (`jquery.com`) in Firefox with Firebug open. We'll use jQuery's site for this exercise, since they will definitely have it loaded already!

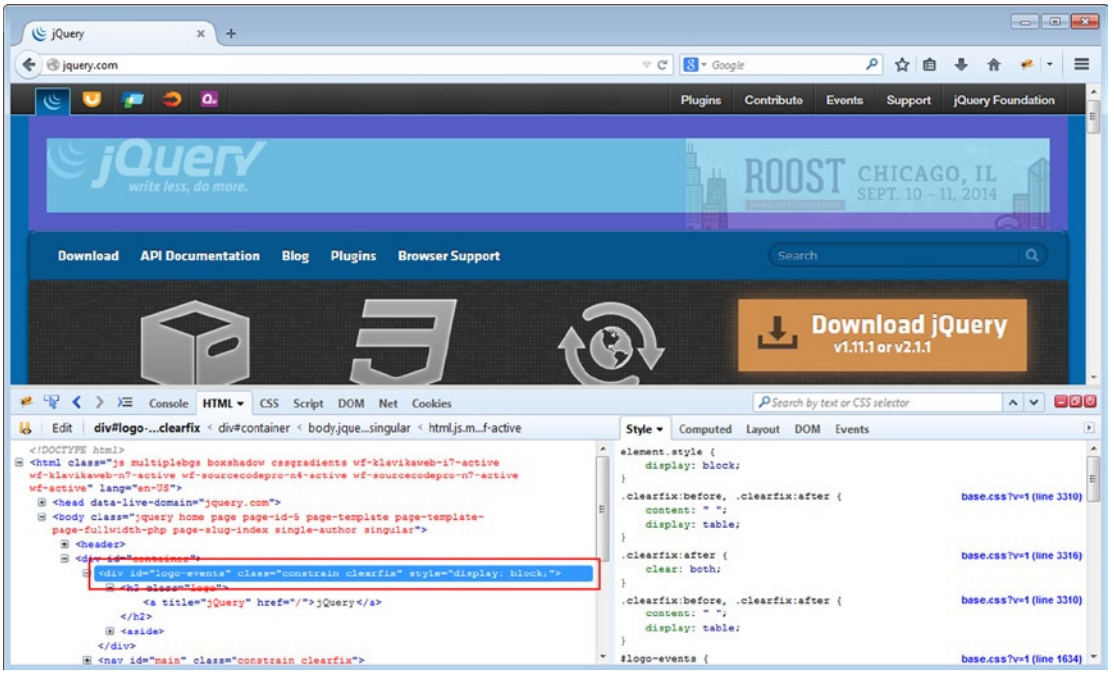

Figure 1-11. *The jQuery home page in Firefox with Firebug open. Hovering over an HTML section with the mouse will highlight it on the page*

We've expanded the HTML in Firebug and located the logo in the top left of the page. As you hover over the HTML with your mouse, Firebug will highlight the sections in the browser so that you can easily see it. On this page, the logo is inside a DIV element with an ID of "`logo-events`". Since this element has an ID, it can easily be targeted with jQuery. In Firebug, click the tab titled Script to activate it, as shown in Figure 1-12.

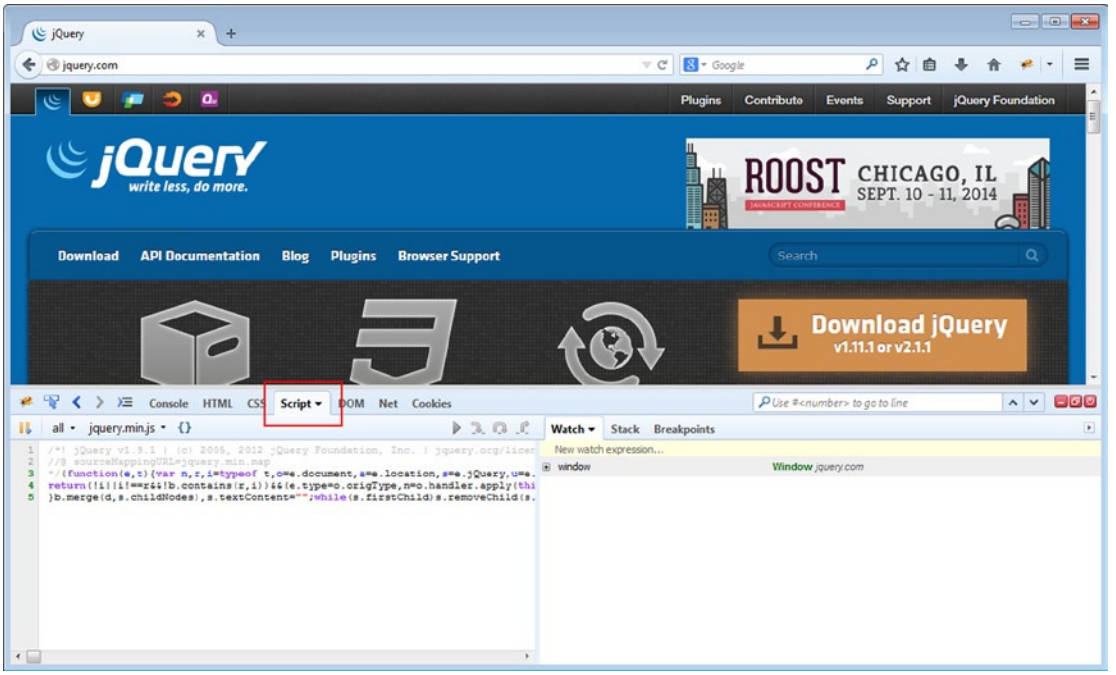

Figure 1-12. *Scripts tab activated in Firebug*

In the Scripts tab there are two main areas where you will work with your front-end code. The left panel shows the current script that you are inspecting. There is a drop-down above it, where you can see and select all the current JavaScript files that are loaded on this page. We'll explore this in more detail later on. The right panel has more tabs, but the main one is the Watch tab. In this tab we can run on-demand JavaScript commands right in the browser using the "New watch expression" textbox. This area will be extremely useful when writing front-end code.

Click inside the "New watch expression" textbox, type the following line, and then hit Enter:

```
$("#logo-events")
```

Figure 1-13 shows the output of running this line of code.

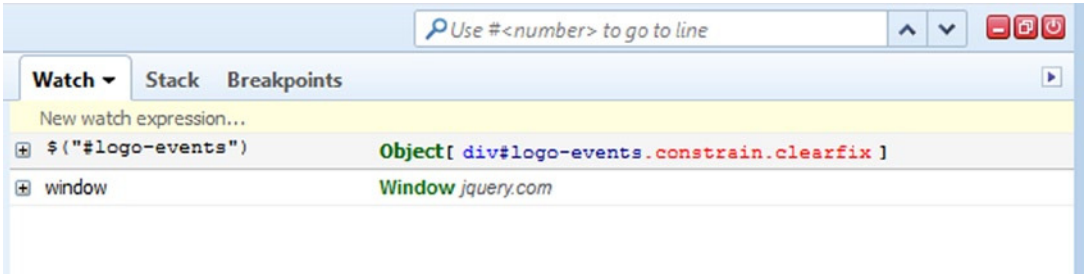

Figure 1-13. *Output of running the jQuery command in the Watch tab*

As you can see, the result is an Object, and inside the brackets you can see `"div#logo-events"`, which means that jQuery successfully found the element that you were looking for. Let's take a moment to look at what happened here and how jQuery works. We can break the structure of the jQuery command like this:

- `$` - This is a shorthand symbol for invoking jQuery. All jQuery commands will start with this symbol. You can think of it as asking jQuery to go do something for you. In this case, we are saying, "jQuery go find the element with this ID." The `$` is always followed by `()`, which enclose the element that you are looking for.
- `#` - This is a shorthand symbol for telling jQuery that you are looking for something by its ID. Any text following the `#` should be a valid ID on an HTML element and will only return one item, because HTML elements cannot share IDs.
- `.` - The period is shorthand for telling jQuery that you are looking for something by its CSS class. So if you have 10 DIVs in your HTML that all share the same CSS class of `"red-box"`, you could find them all by using the following command: `$(".red-box")`. It has the exact same structure as an ID command, just with a `.` instead of a `$`. The `.` is always followed by `()`, which enclose the element that you are looking for.

Now that we have found the DIV that contains the logo, we can start to use jQuery on it and manipulate the HTML on the page. Click inside the "New watch expression" textbox, type the following line, and then hit Enter:

```
$("#logo-events").hide("slow")
```

You will see the logo area slowly disappear from the browser window. You can bring it back by using the following command:

```
$("#logo-events").show("slow")
```

These are very simple examples, but should show the power that jQuery can easily bring to your code. We'll be using jQuery throughout this book, so you'll see more examples and get more familiar with using it as you progress. For now, let's get jQuery implemented in our SharePoint environment so that it's ready to use.

Implementing jQuery in Your SharePoint Environment Using SharePoint Designer

For some of you, jQuery is already implemented in SharePoint. This may have been done to support prior development. However, if you are starting out or just signed up for a trial of SharePoint Online, you'll need to implement jQuery for the examples in this book. Implementing jQuery is actually very easy and straightforward, and in this section we'll look at how to do this using SharePoint Designer.

■ **Note** There are a number of different ways to implement jQuery in your environment. These include custom code deployments to inject it onto all pages, placing Content Editor Webparts on the certain pages, and more. We will be using SharePoint Designer, as it offers a quick and easy way to get jQuery into SharePoint.

Obtaining jQuery

Before we can implement jQuery, we first need to either reference the jQuery code or download a copy and physically upload it to SharePoint. In this section, we'll be using a *content delivery network* (CDN) to reference jQuery. A CDN is basically a publicly available server where content resides that you can use on your sites. jQuery operates a CDN that you can reference at `https://code.jquery.com`. Many other companies also operate CDNs that host jQuery, including Microsoft and Google.

The benefits of using a CDN are many. The biggest benefit is speed, as there are many sites using the same CDN on the Web. If your users have already visited one of those sites, then jQuery is probably already loaded into their browsers, so your pages will load faster. The other big benefit is that you do not have to worry about downloading and storing the code directly. In either case, the decision to use a CDN on a production site should be discussed with your IT department. In our case, it will work out just fine and you should feel comfortable using it in a development environment.

To reference jQuery from the CDN, we will use the following line of code:

```
<script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
```

This is a basic HTML script tag that will load jQuery version 1.11 from the jQuery CDN. You'll notice that there is no HTTP or HTTPS in the URL. This is removed so that the script will work the same in either a nonsecure or secure URL. Now let's look at how to place this into SharePoint.

Team Site Template

Now that we know how we're implementing jQuery, we'll need to do it! We'll look at two options: a Team Site and a Publishing Site. They differ slightly, so let's look at a Team Site first. Launch SharePoint Designer and click the Open Site icon, as shown in Figure 1-14.

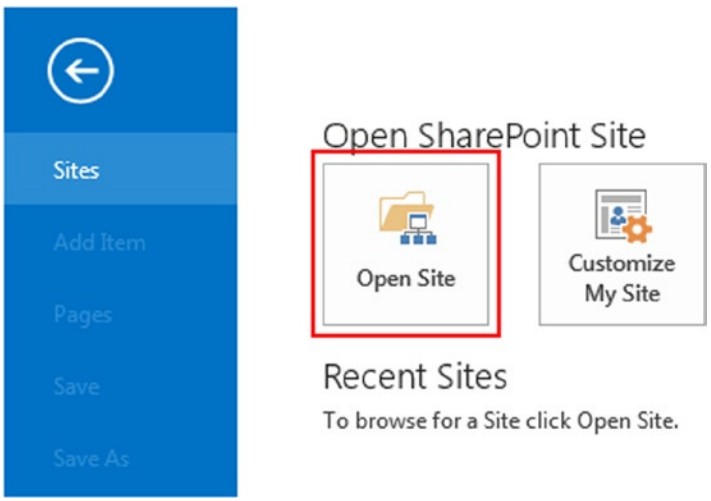

Figure 1-14. *The home screen in SharePoint Designer with an Open Site icon*

Clicking this button will launch the Open Site dialog, where you will need to specify the URL to your SharePoint site collection. You can open specific site collections or you can open the root site by typing in the base URL for your SharePoint environment. If you're using SharePoint online, the URL for your root site will look something like `https://YOURSITENAME.sharepoint.com`.

Figure 1-15 shows the Open Site dialog. You'll type the URL in the textbox labeled "Site name" and then click Open. You'll be presented with another dialog to enter your credentials before Designer opens the site.

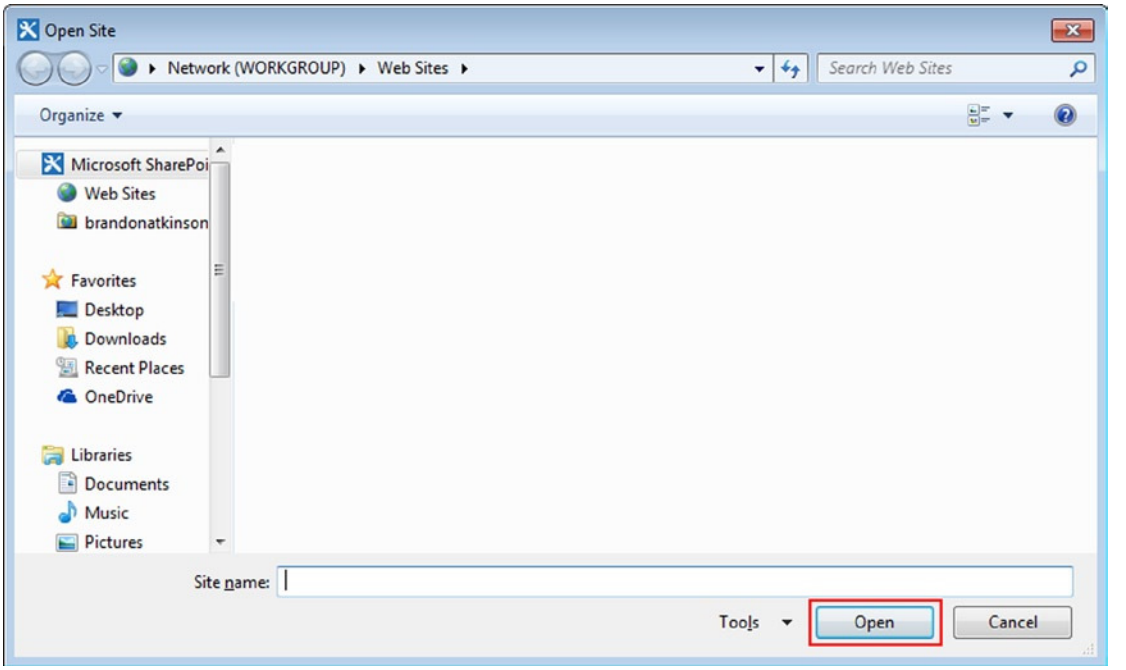

Figure 1-15. *Open Site dialog where you supply the URL to your SharePoint site*

Once you've supplied your credentials and have been authenticated, SharePoint Designer will open and give you a view into your site. If you've never used Designer before, take a moment to look around; you may be impressed by the level of information available to you. Designer gives you insights into all your lists and libraries, workflows, external content types, and a lot more. It's a very useful tool—especially in SharePoint Online, where you do not have access to a server.

■ **Note** SharePoint Designer is a very powerful tool that can very easily change your SharePoint environment. If you are new to this tool, take care when navigating around and only make changes that you know will not be harmful to your environment.

Once you've become familiar with Designer, click the Master Pages link in the left-hand navigation, as shown in Figure 1-16. This area not only displays all the master pages in SharePoint, but also other layouts, including Search Results, Display Templates, and more. We'll come back to this area later in the book, but for now, highlight the `seattle.master` file. Once this file is highlighted, click Edit File from the Ribbon.

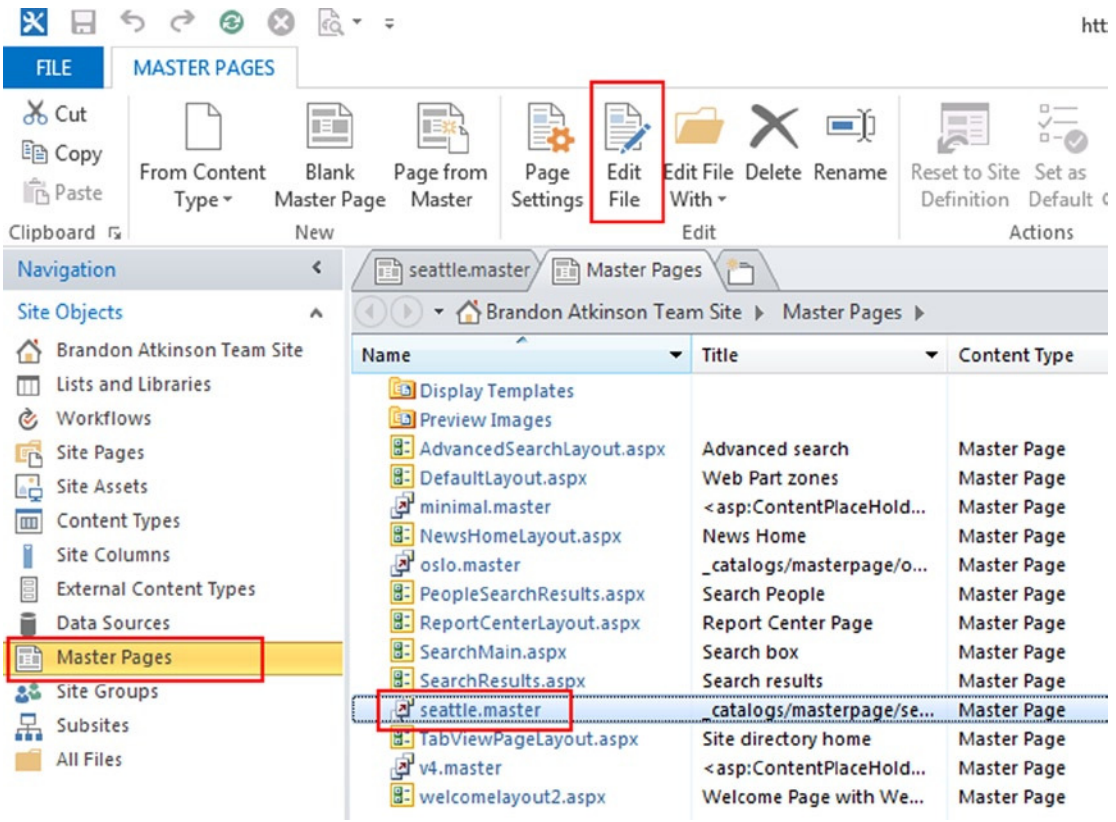

Figure 1-16. *The Master Pages tab in SharePoint Designer. Once you highlight the seattle.master file, click Edit File in the Ribbon*

If you're new to development, the next screen may look intimidating but it's not. Once you're editing the `seattle.master` file, you'll see all the ASP.NET code that renders your SharePoint pages. Well, almost. This file is responsible for the common parts of your SharePoint pages, like the logo, Ribbon, navigation, and more. Although there is a lot here, we only need to add in our jQuery reference—and then get out!

We'll place our line of code in right near the top of the page, just before the closing `</head>` tag, as shown in Figure 1-17.

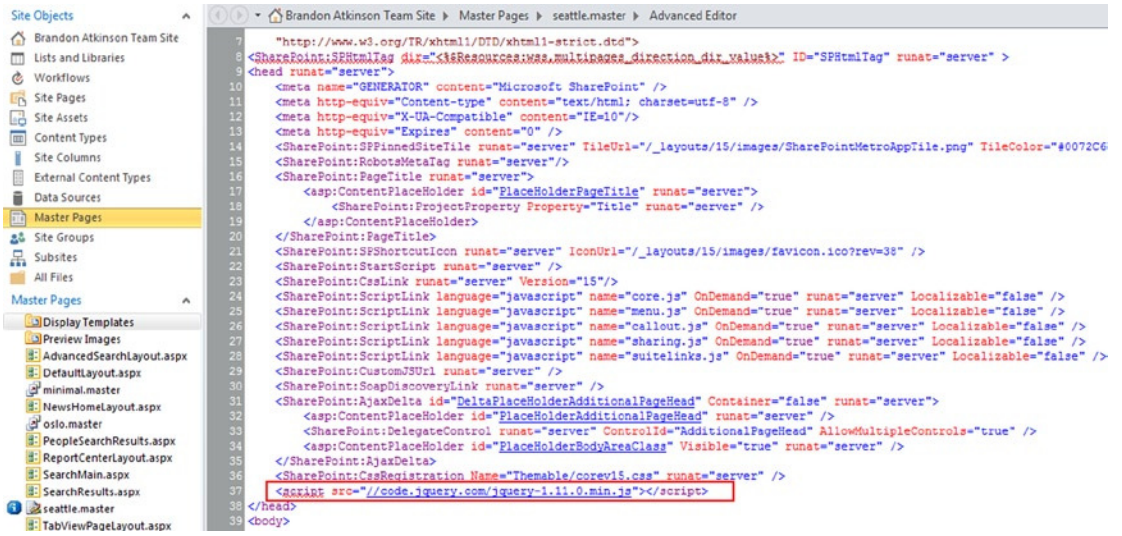

Figure 1-17. *jQuery reference added the Seattle master page*

As you can see in Figure 1-17, we place the jQuery reference right before the closing </head> tag, which is right above the opening <body> tag. It will basically be the last script referenced in the master page in the Head area. Once that is in place, click the Save icon at the top left of the screen to save the changes. This will automatically publish the change to your SharePoint environment.

Once the changes are saved, launch a browser and navigate to your site. Figure 1-18 shows the SharePoint environment with Firebug open. Inspecting the HTML, we can see that the jQuery reference is now in the master page and available to all pages in SharePoint.

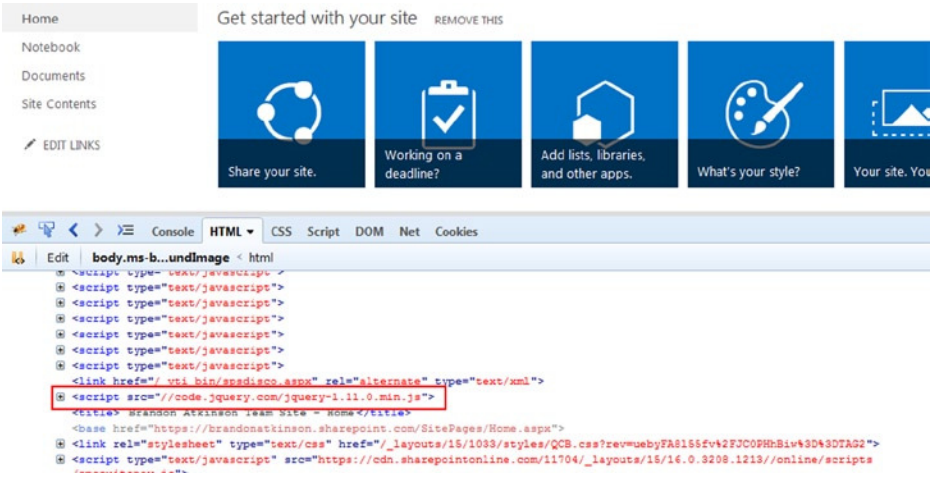

Figure 1-18. *Inspecting your SharePoint environment with Firebug to ensure the jQuery reference is in place*

■ **Note** Generally speaking, modifying the out-of-the-box master pages in SharePoint is not a good idea. The biggest reason for this is since they are out-of-the-box, if an update or new release comes along and your SharePoint Online is upgraded, you run a high risk of your changes being lost. The best practice is to make a copy of the master page you want to modify and change that one. This way, updates will not overwrite your changes.

Publishing Site Template

Publishing Sites act quite differently when it comes to modifying master pages. With SharePoint 2013 came the Design Manager, which allows you to upload an HTML file that will be converted to a master page for you. This is great for users who want to upload a custom design and totally change the look of their site. We just need to inject jQuery, however, so we'll need to complete a few extra steps.

Open SharePoint Designer and click the Master Pages link in the left-hand navigation. The first thing you'll notice is a lot more files than in a Team Site. All the extra files come along with a Publishing Site, so you can pretty much ignore them all for now. Locate the `seattle.master` file and you'll notice it has an accompanying HTML file, as shown in Figure 1-19.

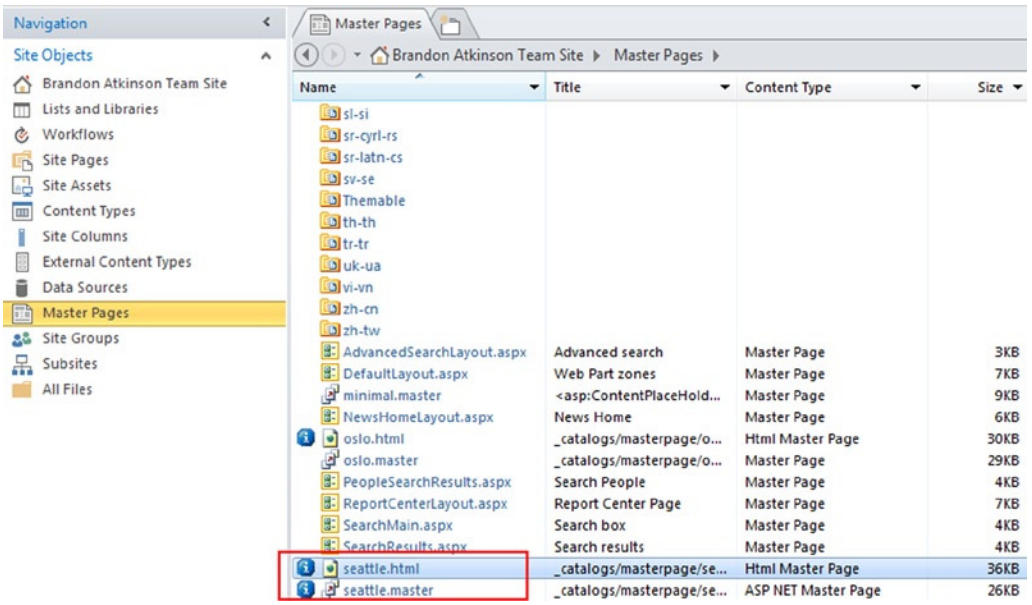

Figure 1-19. *Master Pages in SharePoint Designer on a Publishing Site template*

In this model, you can only edit the HTML file. Once you make an edit to the HTML file and save it, the changes are converted and saved into the master page file. The first thing we need to do is make a copy of the `seattle.html` file, since this is the one we can actually edit. Highlight the `seattle.html` file and from the Ribbon, click the Copy button. The Paste button will become active. Click Paste. A copy of `seattle.html` will be created, as shown in Figure 1-20.

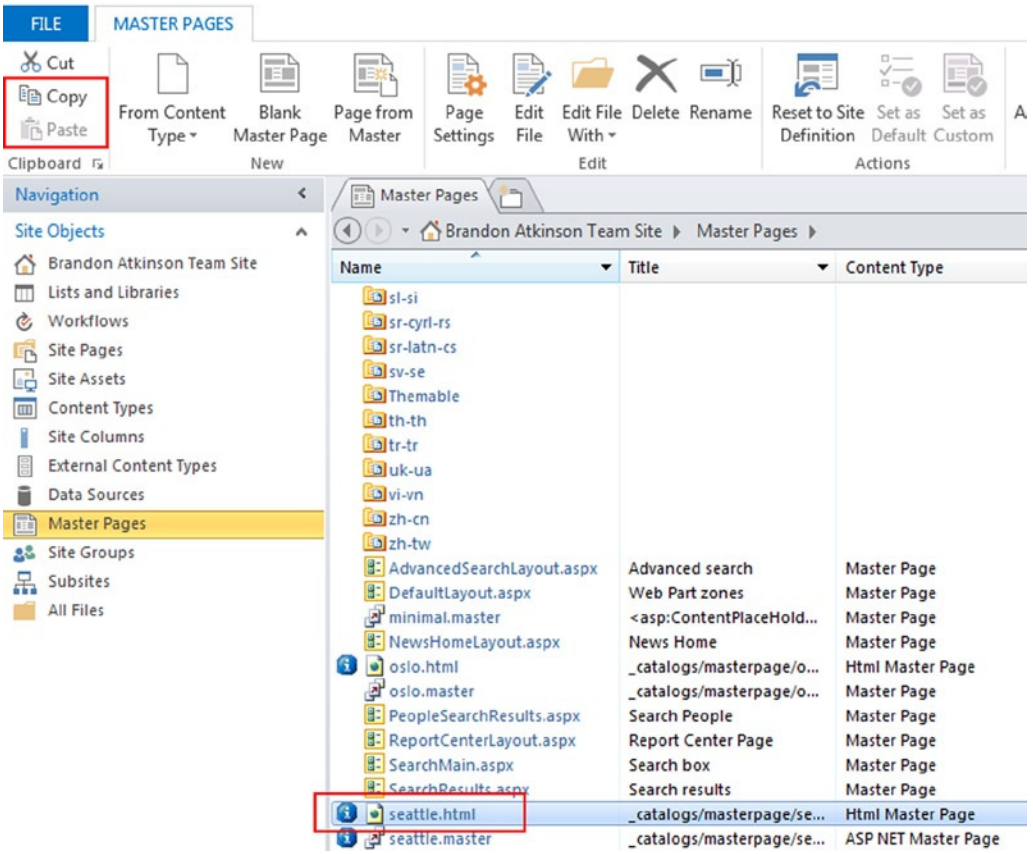

Figure 1-20. *Copy and Paste buttons in the Ribbon*

Once you make a copy of the file, it will be pasted in with a default name of `seattle_copy(1).html`. For most people, this is just not an acceptable name! You can right-click the new file and choose Rename from the context menu, as shown in Figure 1-21.

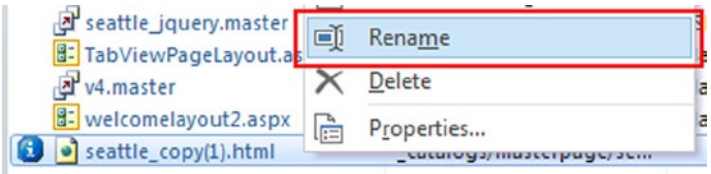

Figure 1-21. *The Rename option from the context menu*

We'll rename the file **seattle_jquery.html** and then click Enter. Since the HTML files are linked to a master page by the same name, you will be prompted by Designer before you can rename the new file. Figure 1-22 shows the prompt you will see during the rename.

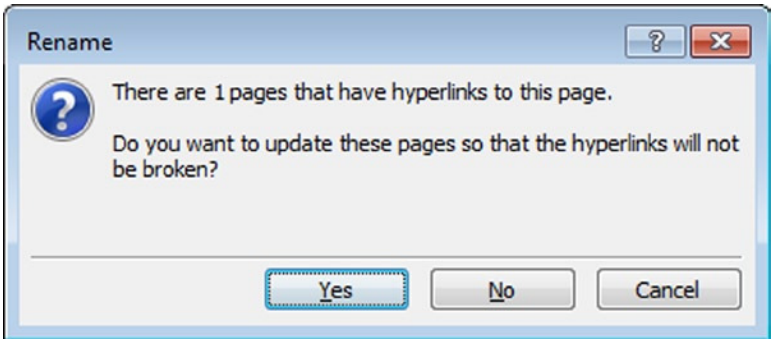

Figure 1-22. *The Rename prompt when renaming the copied HTML file*

After the rename, you will now have a `seattle_jquery.html` and a `seattle_jquery.master` file. Highlight the new `seattle_jquery.html` file and click Edit File from the Ribbon. As with the Team Site master page, we'll insert the following line to reference jQuery:

```
<script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
```

We'll place it in a different location this time, as shown in Figure 1-23. It will be added just below the META tags, right after some out-of-the-box SharePoint tags.

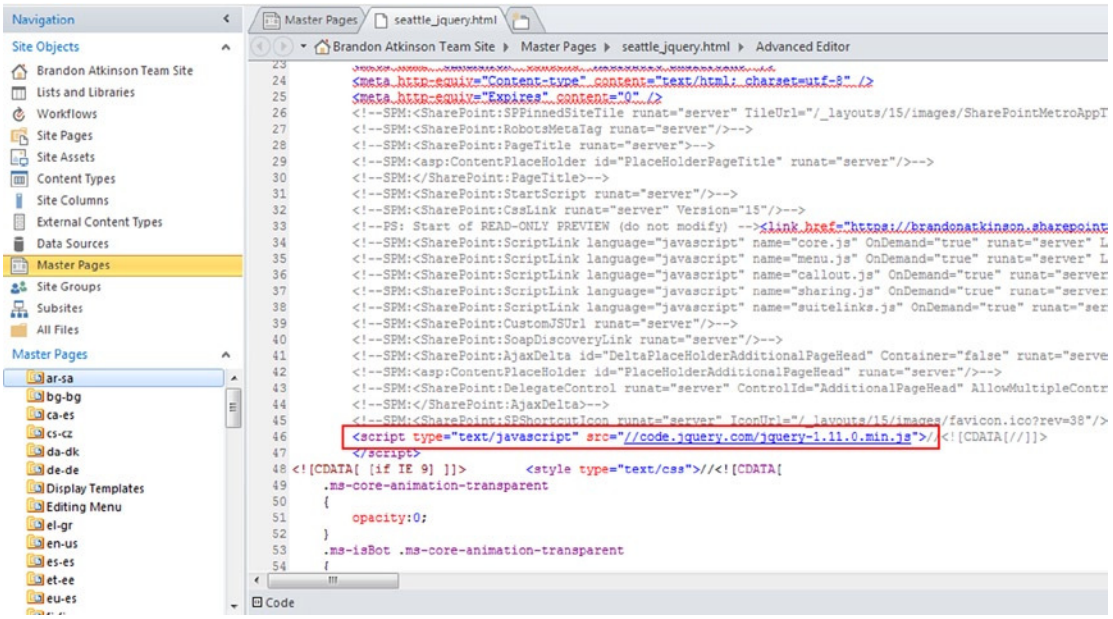

Figure 1-23. *Adding the jQuery line to the new master page file*

Once the jQuery reference has been added, click Save, which will save and publish the file. It will also generate a new `seattle_jquery.master` file with the changes. Now that we have the reference added and all the files have been updated, we need to set this new master page as the default so that our site uses it. Highlight `seattle_jquery.master` and click Set as Default in the Ribbon, as shown in Figure 1-24.

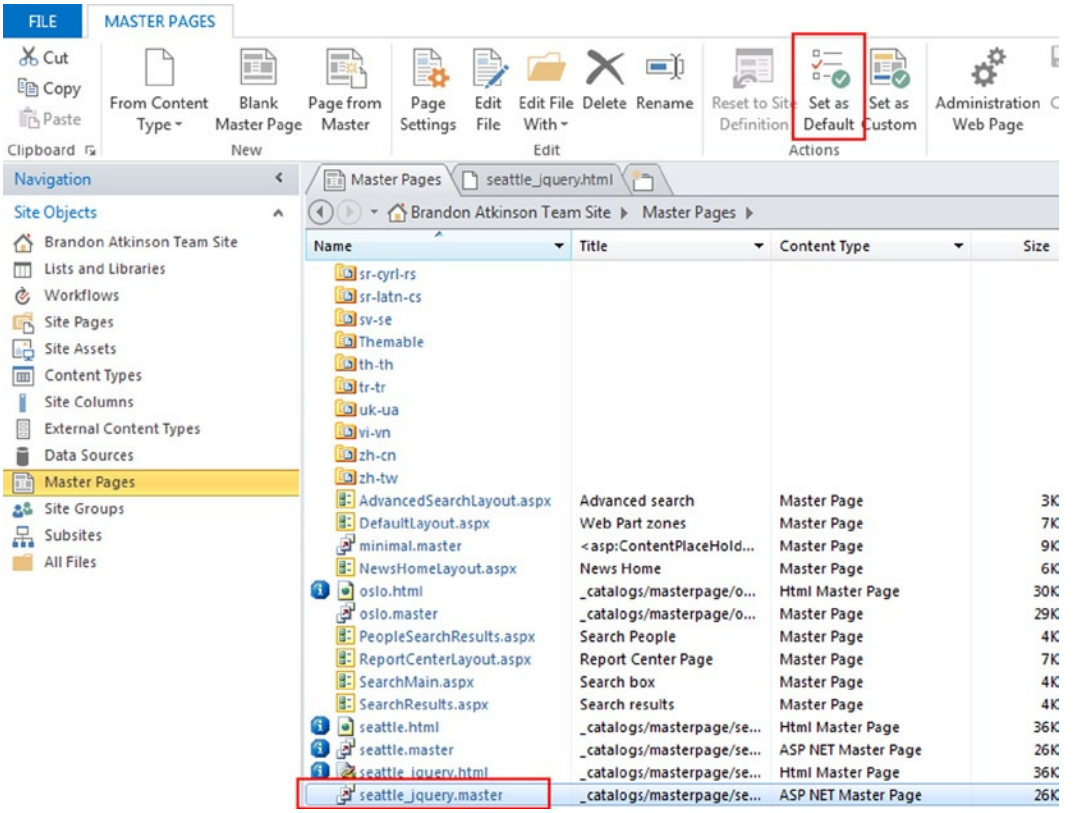

Figure 1-24. *Setting the new master page as the default*

Now that the new master page has been created and set as the default, we'll check to ensure the jQuery reference has been set correctly. Navigate to the site where the Publishing Site template is being used. Figure 1-25 shows the SharePoint environment with Firebug open. Inspecting the HTML, we can see that the jQuery reference is now in the master page and available to all pages in SharePoint.

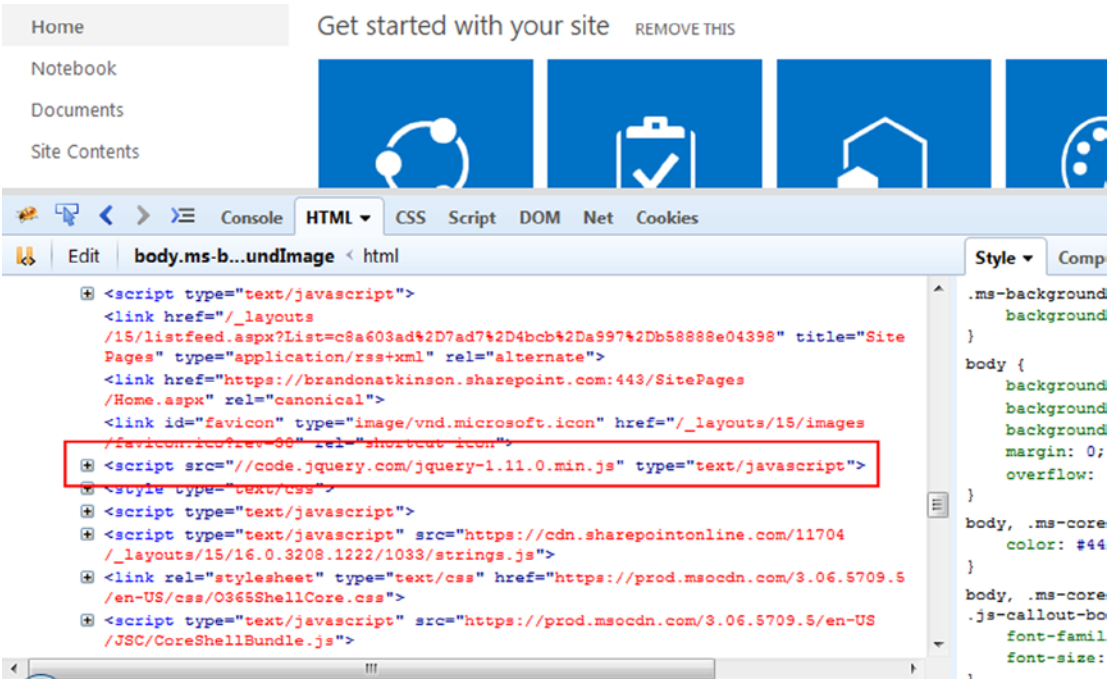

Figure 1-25. *jQuery reference in the new master page*

Any child site created from the parent site using the publishing template will automatically inherit this new master page. This is the preferred way to add a reference to jQuery in the master page. Making a copy of the file ensures that your changes will stay in place if an update is applied to SharePoint.

Summary

In this chapter, we looked at all the various components you'll need to do front-end development in this book. If you're already developing solutions, you probably had most—if not all—of these components installed before you picked up this book. If you're new to front-end development, you probably installed some new programs! By now you should have everything you need to begin creating your own custom solutions. The great thing about these components is that outside of SharePoint Online and Designer, all these can be used for many other projects. If you find that you like front-end development, you can use these tools for a wide range of projects!

In the next chapter, we'll explore the differences between server-side and front-end development. We'll also look at some of the components that SharePoint provides that allow for this type of development. You'll see some code and get prepared for further chapters!

CHAPTER 2

Introduction to SharePoint Front-End Solutions

SharePoint has come a very long way since it debuted, and the development story is no different. If you have to do any development on SharePoint 2007 or earlier, it's fairly safe to say you hated the experience. SharePoint 2010 made things much easier and introduced some new components that allowed for front-end development, like the ECMAScript Object Model, later renamed the JavaScript Object Model. With the most recent versions of SharePoint, a real focus has been put on moving code off the SharePoint server and into the browser or other servers.

This focus came in part because Microsoft was seeing a lot of support issues arise from server code that was either written poorly or was just pushing a server to its limits. Combine that with the introduction of SharePoint Online, where end users have no access to the server itself, the need for a more robust front-end development solution was needed. What came from this was an expanded JavaScript Object Model, a new REST API, and Apps for SharePoint.

We'll be digging into the JavaScript Object Model and the REST API in more detail throughout this book. We'll also be looking at new front-end components, such as JSLink and display templates. This chapter will explore the differences between server-side and front-end development, as well as some of the capabilities of this style of development. We'll also explore some of the limitations you may encounter along the way.

Note Apps for SharePoint will not be covered in this book. While Apps can be considered front-end development in some cases, it's a very large topic that involves development, networking, and domain management. Apps for SharePoint, in a lot of cases, runs server side, just not on the server running SharePoint. For more information on building Apps for SharePoint, please visit the following URL: http://msdn.microsoft.com/en-us/library/office/jj163230(v=office.15).aspx.

Server-Side Development vs. Front-End Development

SharePoint development and the building of custom solutions remained the realm of hardcore developers for a very long time. It took knowledge of a wide range of components, including Visual Studio, PowerShell, ASP.NET, Microsoft Server, and more. It was at times a daunting experience—even for those who had been doing .NET development but were new to SharePoint. Figure 2-1 shows a typical server-side development experience, which includes Visual Studio, deployment packages, and a PowerShell window to actually deploy the code to the server.

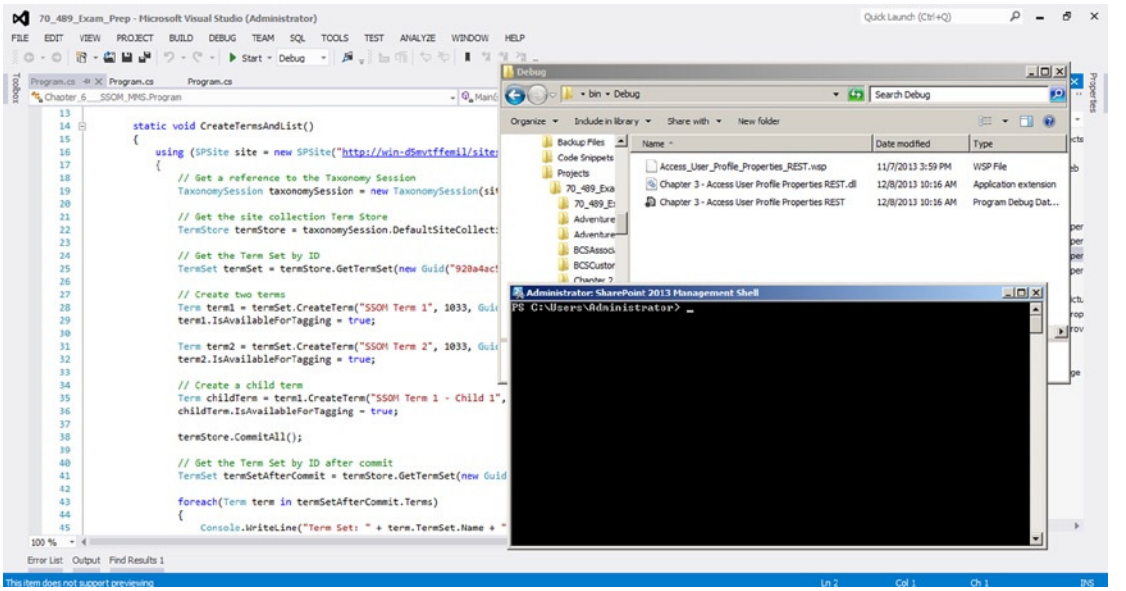

Figure 2-1. *A typical SharePoint server-side development experience*

Server-side code is written specifically for running on the SharePoint server itself. Once the code is written and packaged for deployment, it is copied to the server and installed using either PowerShell or Central Administration. In most cases it involves developers, business users, and server administrators all working together. Since the code runs directly on the server, it has a tremendous amount of power and can adversely affect the SharePoint environment quite easily.

Even writing server-side code for SharePoint can be difficult, as it requires a powerful PC that has SharePoint installed on it. It used to be that most developers would build virtual machines on their PCs, where they could write code for SharePoint in an isolated environment. It was at times difficult to learn server-side development due to the new code frameworks you had to learn, the needs of having SharePoint installed on your PC, and coding best practices required to ensure you did not harm the server with poorly written code.

Once SharePoint Online and Office 365 came along, server-side development took a hit, because with these products you have no access to the server running SharePoint. This is good from the standpoint of all the other customers using the same server. If one customer were to install poorly written or malicious code, they could potentially take down the entire server for everyone. However, if you're a customer who has the need for custom solutions, you no longer have the option for robust server-side code.

■ **Note** Sandboxed solutions offered another way to write and deploy server-side code in SharePoint. These solutions did not require server access, as they could be installed using the browser by a site collection administrator. They were called "sandboxed" because they ran in isolation on the server and were monitored by SharePoint to ensure that they did not affect performance. With SharePoint 2013, sandboxed solutions could still be installed on-premises, but were deprecated in SharePoint Online, and could only include HTML and JavaScript. For more information on sandboxed solutions, please visit the following URL: `http://technet.microsoft.com/en-us/library/ee721992(v=office.14).aspx`.

At first glance it would seem that Microsoft gutted the ability to create custom SharePoint solutions for your business, especially if you're using Office 365. Customers using SharePoint on-premises are also getting the same message from Microsoft, which has stop putting custom code on the server. Luckily, Microsoft did not leave anyone "high and dry" with this approach. The latest versions of SharePoint come with a wide range of front-end components to develop solutions.

Front-end development, or client-side code, is written specifically for running in a web browser. It uses HTML, JavaScript, and CSS, and runs in every browser. You will still add your code to SharePoint, but since it's only HTML and JavaScript, you can utilize out-of-the-box components like document libraries to upload and store your solutions. In the next section, we'll explore the various components of front-end development with SharePoint and look at what they offer you.

Components of Front-End Development in SharePoint

Now that we've looked at some differences between server-side and front-end development, let's take a look at some of the components of front-end development that this book will cover. Throughout this book we'll showcase several different ways to create custom solutions, and in this section we'll introduce you to them. These will be fairly quick, high-level introductions, as we'll dive much deeper into each of these in later chapters. We'll explore

- JavaScript Object Model (JSOM)
- REST API
- JS Link
- Display templates

Each of these items operates in conjunction with other components to comprise a total front-end solution. You may even use several of them together if needed. For now, we'll quickly look at each item and get you ready to start building some solutions in the next chapters.

JavaScript Object Model (JSOM)

The JavaScript Object Model, or JSOM for short, is part of the family of SharePoint client APIs. This is traditionally called *client-side object model*, or CSOM for short. All this is fairly confusing, as people will tend to use the term "client-side object model" for a wide range of discussions. CSOM encompasses everything from JSOM, to Silverlight libraries, down to a set of server-side libraries, and even the REST API. The point here is that if someone is talking about CSOM, you may need them to be a bit more specific about what they are doing.

The JavaScript Object Model is basically a collection of JavaScript files that come as part of SharePoint. Each file contains code that will allow you to do a wide range of tasks—from interacting with the term store, accessing user profile information, or just common core tasks like reading lists and sites. If you have access to a server, you can view all of these files at `C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\15\TEMPLATE\LAYOUTS`. Figure 2-2 shows this folder in Windows Explorer. If you are on SharePoint Online, never fear, all of these files are still available to you—you just can't view them in Windows Explorer!

All of the files are located in the Layouts folder in SharePoint. The Layouts folder is a workhorse folder in SharePoint and contains a lot of common files and code used throughout the environment. When deploying custom server-side code that uses custom pages, images, and so forth, it is all typically deployed to the Layouts folder. If you're not familiar with it, no need to worry—you will be soon.

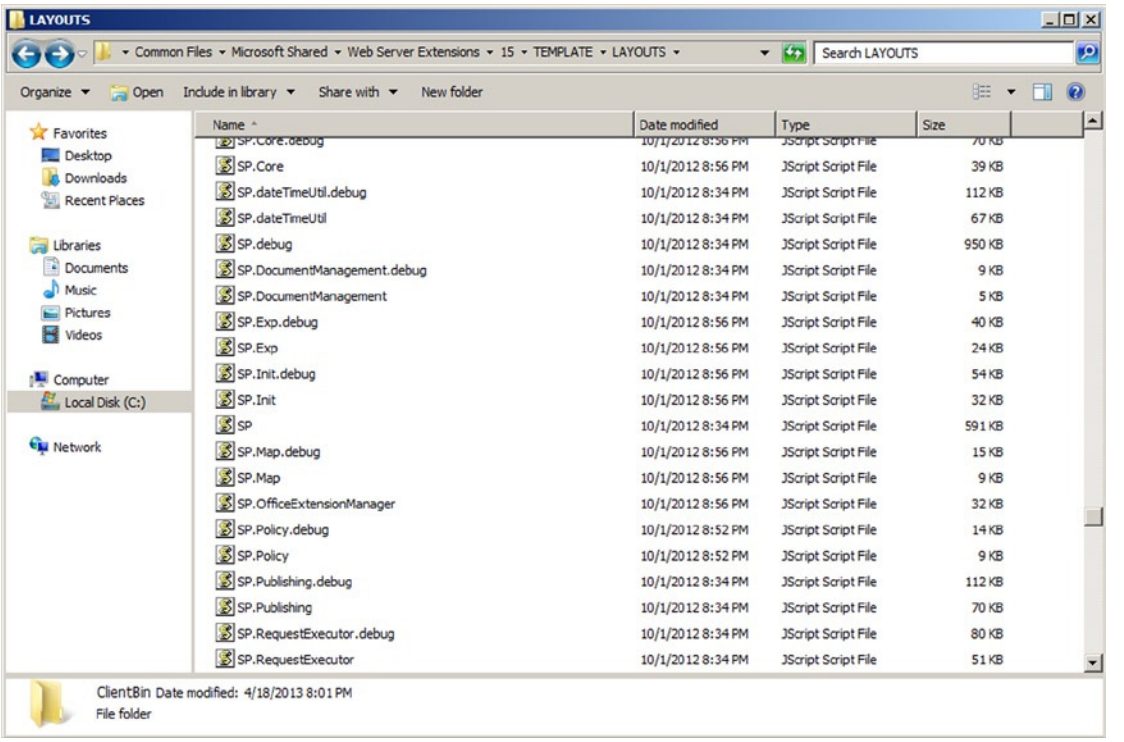

Figure 2-2. *Files that make up the JavaScript Object Model on the file system*

Figure 2-2 only shows a portion of the files available to you, but should help illustrate the power of JSOM for building custom solutions. Microsoft put a lot of time and effort into expanding the capabilities of JSOM. Since they were telling their customers to no longer use server-side code, and in many ways move to SharePoint Online, they made sure to provide a comprehensive set of tools to allow for a rich development experience.

■ **Note** For more information on JSOM, please visit the JavaScript API reference for SharePoint at the following URL: `http://msdn.microsoft.com/en-us/library/office/jj193034(v=office.15).aspx`.

Using JSOM in your solutions is fairly straightforward, as they are all JavaScript. First, you need to place a reference to a particular file, like this:

```
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
```

This is a standard SCRIPT tag in HTML, which tells the browser to load a particular file located at the location specified in the SRC attribute. In this case, the browser would load the `sp.js` file located in the Layouts folder. Once we have a reference to the SharePoint JSOM files that we need, we can write a script that looks something like this:

```
$(function () {
    retrieveListItems();
});

function retrieveListItems() {
    var clientContext = new SP.ClientContext();

    var oList = clientContext.get_web().get_lists().getByTitle('Demo List');

    var camlQuery = new SP.CamlQuery();

    this.collListItem = oList.getItems(camlQuery);

    clientContext.load(collListItem);
    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}
```

If this is the first time you're seeing JSOM code,, don't be overwhelmed! It can look intimidating, but as you work through this book, it will become easier to read and write. Let's break this example down and look at each part.

- `$(function () {});` - This is the entry point of the code and it is all jQuery. This is called a *document.ready function*, and it is called that because it's fired when the HTML page is completely loaded by the browser. You'll see this a lot in jQuery development since it's an easy and convenient way to fire code off when your pages load. In this example, all we're doing is calling another function where the JSOM code resides.
- `SP.ClientContext()` - This line obtains a reference to the client context and is used in every JSOM call to the SharePoint server. The client context contains a lot of information about the current web application, site, site collection, and more.
- `clientContext.get_web().get_lists().getByTitle('Demo List')` - Using the client context, we can obtain references to just about anything in the site. In this line we reference the current site, requesting from all the lists a particular list with the title "Demo List". The syntax of this line is very easy to read: Get Web, Get Lists, Get by Title. You'll see a similar pattern when using the REST API.
- `SP.CamlQuery()` - This line allows you to specify which items to return by using a CAML query, which is a syntax for querying SharePoint . It has been around for a while and it is used a lot when working with lists and libraries. In this example, we simply pass an empty query that tells SharePoint that we want all items in the list. We do this by calling `oList.getItems(camlQuery)` passing in the query.
- `clientContext.executeQueryAsync()` - This line will actually initiate the query to the SharePoint server. You can then process the data that is returned and display it to your users.

That was about as high-level an explanation as possible with JSOM! The main objective was to show some JSOM and briefly explain the parts. There is a lot more to JSOM, but before we jump into any demos, you will at least have had some exposure to the code.

REST API

The REST API was introduced with SharePoint 2013 and SharePoint Online. It allows you to make RESTful calls to SharePoint via a specially crafted URL. Data can be returned as either XML or JSON, and can easily be consumed by JavaScript in the browser. RESTful data was available in previous versions of SharePoint, but was limited to mostly list interactions. The new REST API has been greatly expanded, like JSOM, to include a wide range of functions.

■ **Note** For more information on the REST API, please visit the REST API reference and samples for SharePoint at the following URL: `http://msdn.microsoft.com/en-us/library/office/jj860569(v=office.15).aspx`.

A typical REST API call will look like the following URL:

```
http://SERVER/_api/web/lists/GetByTitle('Demo List')/items
```

Just like the JSOM example in the previous section, this URL will query the list titled Demo List and request all the items. The syntax should look familiar as well: Web, Lists, GetByTitle, Items. With both examples, the structure of the call is very easy to read and gives you all the context you need to know about what data to expect. Figure 2-3 shows the results of a REST call in the browser.

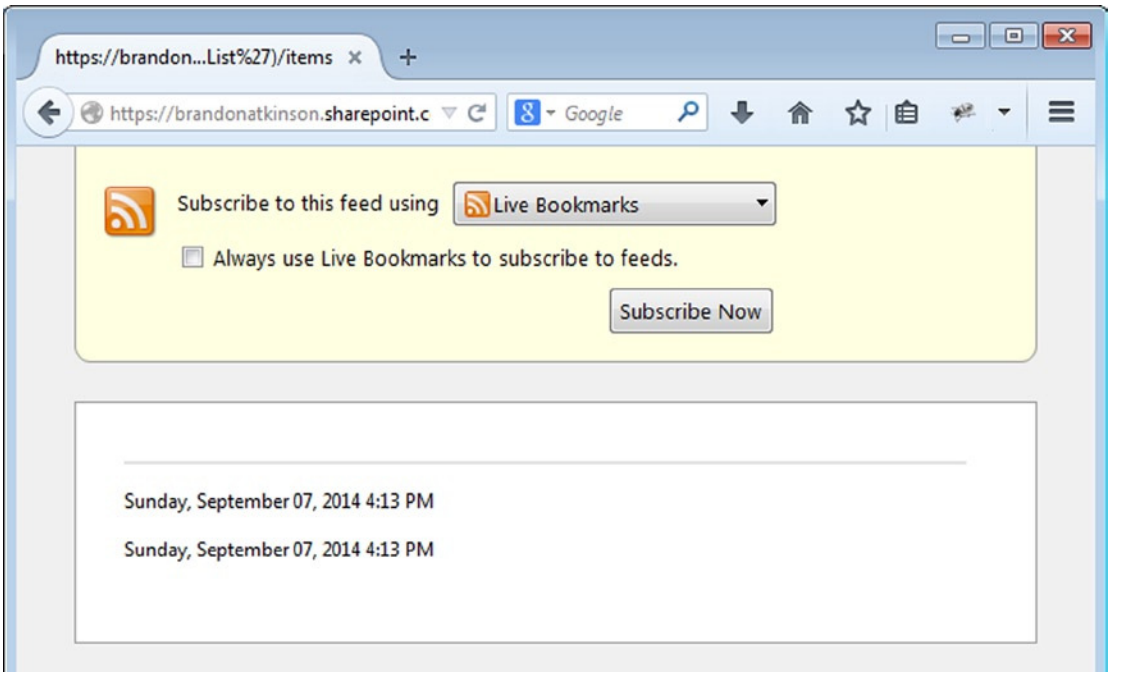

Figure 2-3. *REST API call in the browser returned as XML*

In this example, the results are returned as XML, as that is what the browser requests by default. When calling via JavaScript, we can request either XML or JSON. We'll look at that in more detail later on. Like JSOM, where we can supply a CAML query to limit the results, we can use query string operators to select, filter, and order the data we request from the API.

If we wanted to limit the data to a single row, the REST API call will look like the following URL:

```
http://SERVER/_api/web/lists/GetByTitle('Demo List')/items?$top=1
```

The query string operator `top=1` tells the API to only return the first row of data. Figure 2-4 shows the results of this call in the browser.

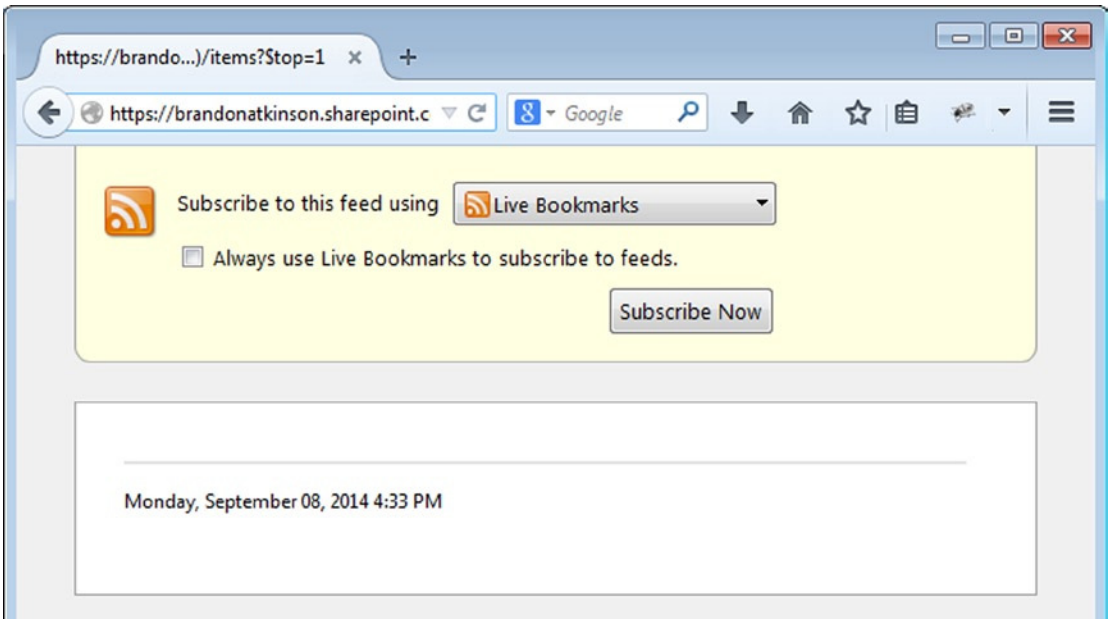

Figure 2-4. *REST API call with a query string operator limiting results*

There are many operators that you can use to craft your REST API calls. We'll be looking at more of these in later chapters, but rest assured that you'll have all the tools you need to get proper results. This example should illustrate how easy it is to request data from SharePoint using the REST API.

JSLink

JSLink is new in SharePoint 2013 and offers a very powerful and easy way to style list view web parts. Prior to JSLink you had to use XSLT, or Extensible Stylesheet Language Transformations. XSLT was a way to style, or transform, HTML and give it some life. It did a good job of allowing developers to change the way web parts were rendered, but it was quite a pain to work with. Most developers cringed at the thought of having to use XSLT and generally stayed away from it.

Along came JSLink, which allows you to style parts using JavaScript and HTML. It provides a much easier and faster way to render the web parts to meet business needs, and provide increased functionality. Not only does JSLink provide an easy way to style web parts, because it is HTML and JavaScript you can also do a lot more with it. For now, let's just take a look at how JSLink works. Figure 2-5 shows a standard SharePoint home page with a List Web Part inserted on it and showing the items from the Demo List.

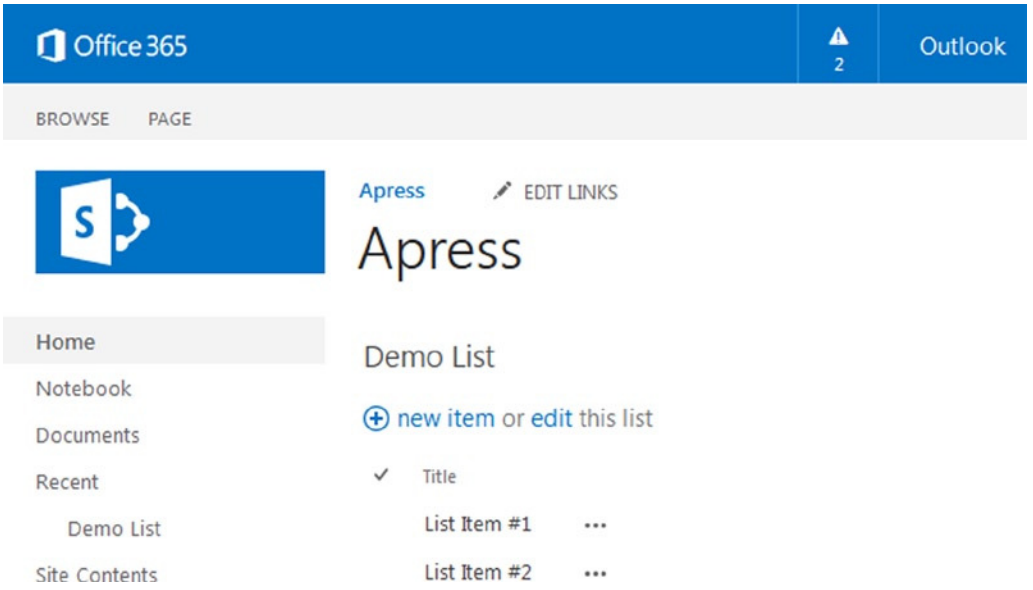

Figure 2-5. *SharePoint page with a List Web Part inserted, listing the items from the Demo List*

This is the standard List Web Part that comes with SharePoint, but now it has a new property on it. If you place the web part into edit mode and expand the Miscellaneous panel in the web part tool pane, you will see a new property called "JS Link". This is a textbox where you can place the path to a JavaScript file that will be used during rendering of the web part. Figure 2-6 shows this new property.

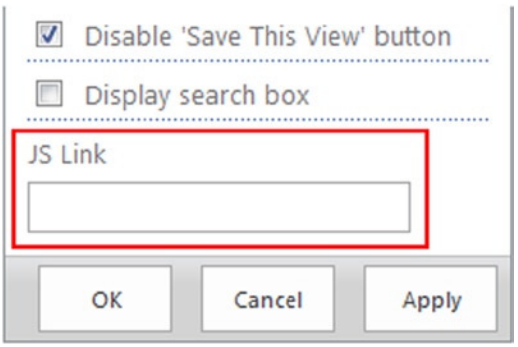

Figure 2-6. *JSLink property in the Web Part Properties pane*

Now that you know how to apply a JavaScript file via the JSLink property, let's see what the JavaScript actually looks like. The following code is a simple demonstration on how the JS Link code works.

```
(function () {

    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Item = overrideTemplate;

    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);

})();

function overrideTemplate(ctx) {
    return "<div style='font-size:32px;border:solid 1px Silver;'>" + ctx.CurrentItem.Title +
"</div>";
}
```

Let's break this example down and look at each part.

- `overrideContext`, `overrideContext.Templates`, and `overrideContext.Templates.Item` - As their names imply, these are objects that override the default rendering of the web part. The parent object, `overrideContext`, can be named anything you like. The first two lines simply create the object and a property called `Templates`. This is standard in all of your JSLink files, as it's the basis for your code. The third line defines another JavaScript function, which will fire for each individual item in the web part; this is designated by the "`Item`" in its name. You can target items, or just specific properties in an item. For instance, if we only wanted to style the `Title`, we can reference that property only.
- `SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext)` - Once you have your object built, you have to tell SharePoint that you want to use it. This line does just that by registering it with the `TemplateManager` object. Once this is done, the JavaScript function you referenced will be called when the web part is rendered.
- `function overrideTemplate(ctx)` - This is the actual code that will run when the web part is rendered. It is very simple: all it does is render an HTML DIV element with some styling and the current item's title inside.

Once this code is placed into a file, we upload it to a library in SharePoint. Then we place the path to the file in the JSLink property of the web part. Figure 2-7 shows the output once the web part is rendered using the JSLink file.

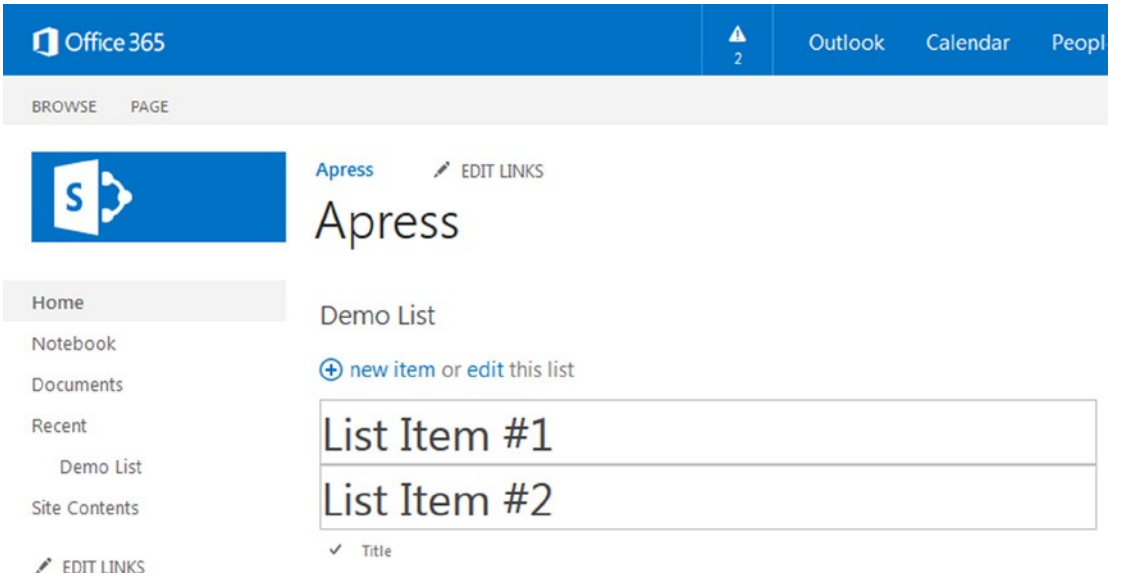

Figure 2-7. *List Web Part after applying JS Link code*

We'll get into more of the details on using JSLink in later chapters, and I'll demonstrate more ways to style content and build custom solutions. When using list web parts on your page, you'll find JSLink an extremely useful tool to achieve your business needs.

Display Templates

Like JSLink, *display templates* are new in SharePoint 2013 and SharePoint Online. They work in conjunction with Search Web Parts, such as the Content Search Web Part that comes with SharePoint. Display templates use HTML and JavaScript to render content and allow you to specify which properties to show, where to place them, and how to style them. By using these templates, you have an immense amount of control over search results and how they are displayed. Figure 2-8 shows a standard Content Search Web Part placed on a page.

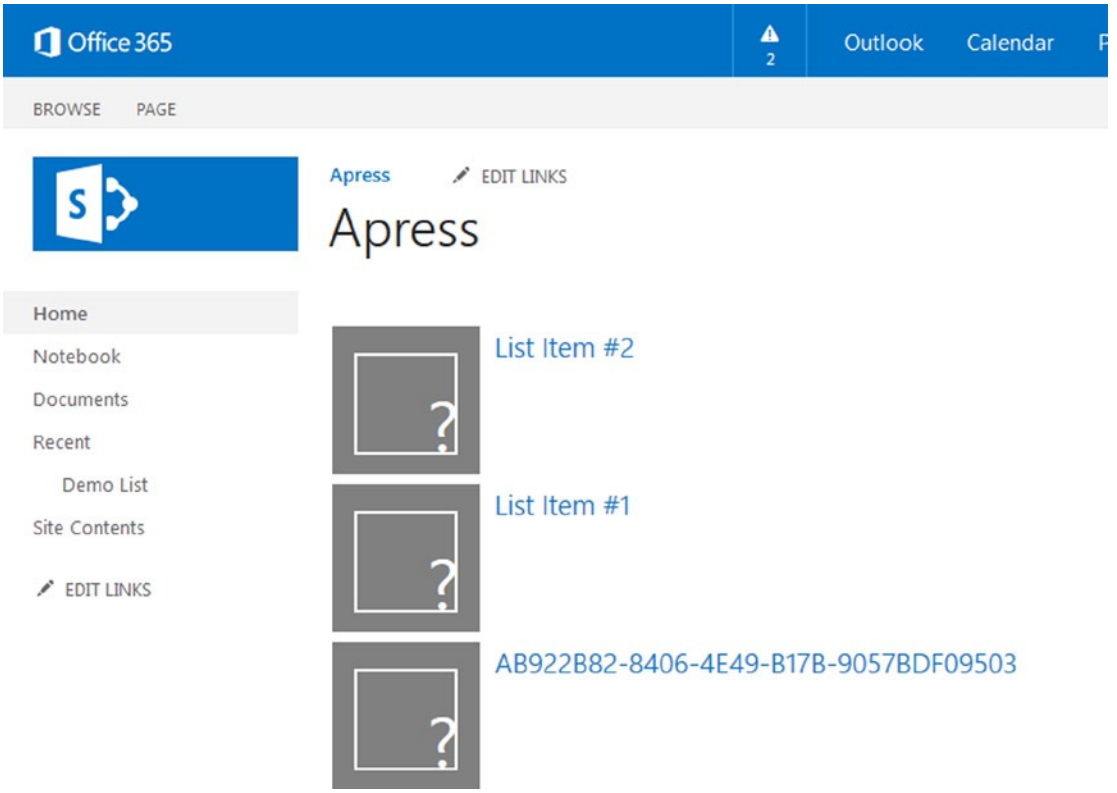

Figure 2-8. *Out-of-the-box Content Search Web Part placed on a page*

We can change the display template being used by placing the web part into edit mode and viewing the Properties panel in the web part tool pane. Directly under Search Criteria, you will find the Display Templates area. There are two drop-downs in this section that allow you to choose how the content should render: Control and Item. Figure 2-9 shows the Display Templates area and drop-downs.

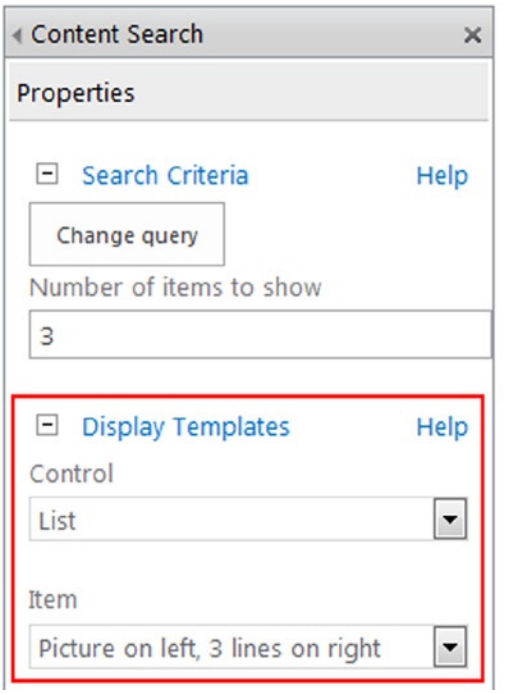

Figure 2-9. *Control and Item options in the Content Search Web Part tool pane*

Each of these items is a template, and they work together to render content. The Control template provides an overall HTML layout for your content. You can think of it as a container to hold all the search result items. It's also a good area to load any JavaScript or CSS that you would like to use in the search results. The Item template provides the individual HTML for each search result item. This template also makes use of Managed Properties on the search results, and allows you to control which properties are displayed. When used together, these templates provide a complete way to render your content.

You can view the out-of-the-box display templates by navigating to Site Settings, and then Master Pages under the Web Designer Galleries header. This will open the Master Page Gallery page, as shown in Figure 2-10.

■ **Note** In order to use display templates and the Content Search Web Part, you will need to ensure that the SharePoint Server Publishing site feature is activated. If your site is using a Publishing template, this should already be activated. This web part is also only available in the Office 365 E3 and E4 plans, or the Enterprise version of SharePoint on-premises.

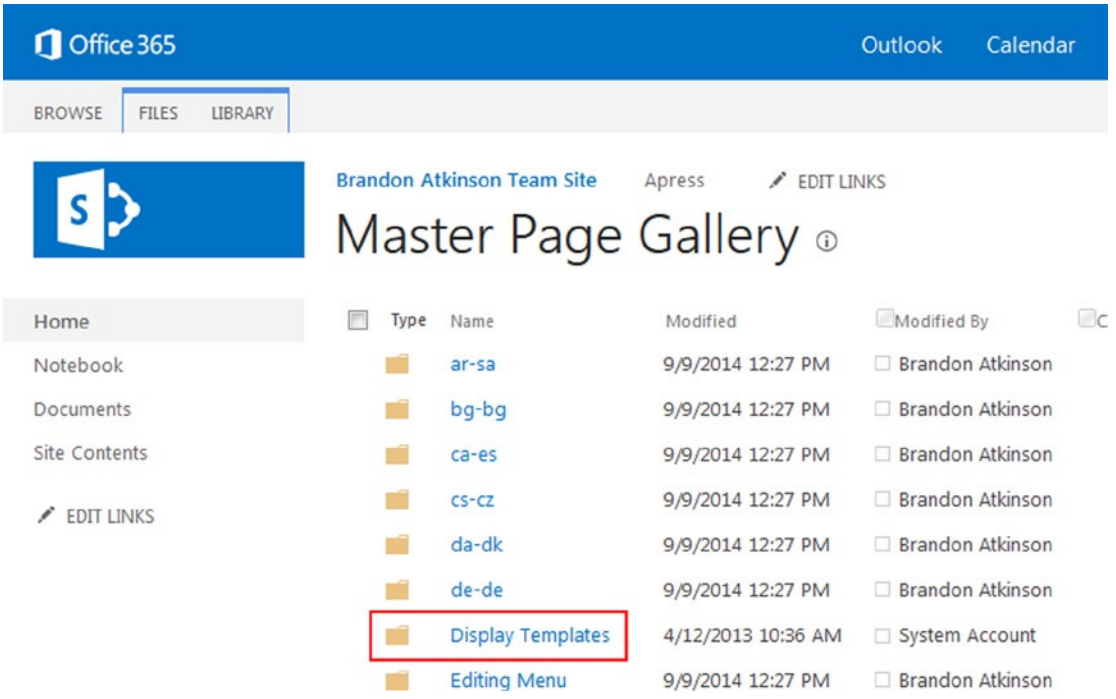

Figure 2-10. *Master Page Gallery where you can find the out-of-the-box display templates*

Clicking the Display Templates folder will display another page, with all the current display templates organized by folder. The Content Search Web Part display templates that are all located in the Content Web Parts folder, as shown in Figure 2-11.

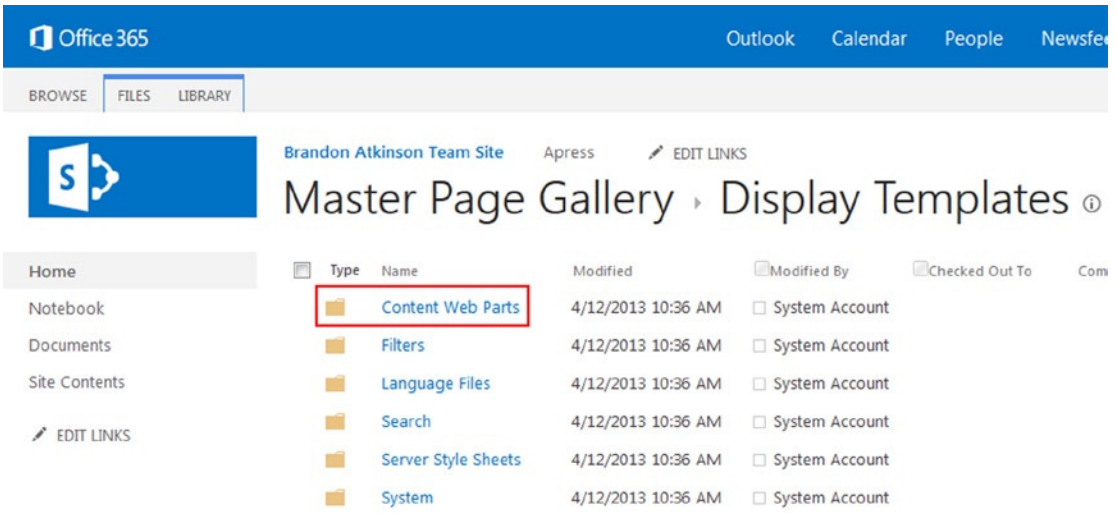

Figure 2-11. *Display Templates and the Content Web Parts folder*

When opening the Content Web Parts folder, you will find a bunch of display templates. Each template has an HTML file and a corresponding JavaScript file, as shown in Figure 2-12.

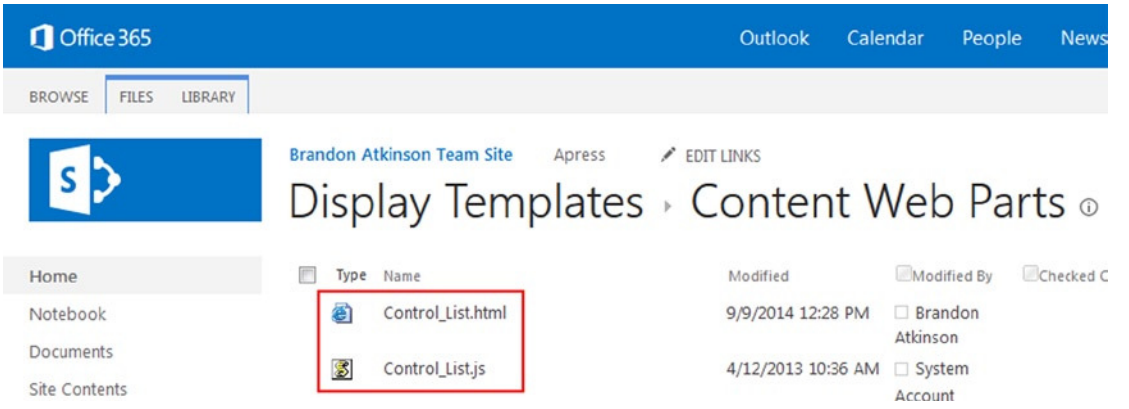

Figure 2-12. *Display template with an HTML file and corresponding JavaScript file*

When you build a new display template, you will only work with HTML files. Once you upload the HTML file, SharePoint generates the JavaScript file for you, and this is what it uses when rendering content. With this model, you don't need to worry about the scripting required; you only worry about the HTML—and SharePoint does the rest. Display templates use managed properties from the search results to render content. The template uses a token system to render the content, as you can see in the following code sample:

```
<a class="cbs-pictureImgLink" href="_#= linkURL =#_" title="_#= $htmlEncode(line1.
defaultValueRenderer(line1)) =#_" id="_#= pictureLinkId =#_">
```

The token looks like `_#= variableName =#_`. You can create your own variables to store content to be used later in the template. We'll look at these in much more detail in later chapters.

Summary

In this chapter, we looked at the differences between server-side development and front-end development and explored some of the components that you can use for your custom solutions. With the latest versions of SharePoint, Microsoft is really stressing the advantages of not using server-side code for your development efforts. When it comes to SharePoint Online, it's not available to you at all! As you have read in this chapter, they have provided a lot of options to build custom solutions that do not rely on server-side code. All of these solutions utilize HTML and JavaScript, making these solutions easily accessible to anyone who is comfortable building web pages.

In the next chapter, we'll continue expanding on what we've learned so far. We'll look at the Content Editor Web Part and how you can place HTML content directly on a page. In addition, we'll explore the new Script Editor Web Part, which can be used in conjunction with the Content Editor. We'll use some of the code you saw in this chapter and actually see it in action.

CHAPTER 3

Content Editor and Script Editor Web Parts

At this point, you should have all the tools that you need installed and a SharePoint environment to develop in. You've also been exposed to the differences between server-side development and front-end development. We've looked at some of the parts and pieces that make up front-end development as well. Now it's time to actually start using some of what we've learned and build some stuff!

In this chapter, we'll look at using the Content Editor and Script Editor Web Parts. If you've been around SharePoint for a while, then you already know about the Content Editor Web Part, and you know how it allows you to add HTML easily to any page. The latest version of SharePoint brought with it the Script Editor Web Part, which allows you to easily add JavaScript to any page. We'll look at how these web parts work and interact with each other. By the end of this chapter, you'll understand the pros and cons of using these web parts and you will have built some custom solutions using HTML and JavaScript.

Content Editor Web Part

The Content Editor Web Part (CEWP) is a workhorse in the SharePoint landscape; it's been around a long time and most users are familiar with it. The CEWP, true to its name, is used to place content on a page in SharePoint. You can easily place HTML on a page using this web part to post content to users, a list of links, and much, much more.

Using the CEWP, you can add rich content from the Ribbon using predefined styles and editing tools similar to Microsoft Word. You can write HTML directly into an editor or point it at an HTML file that holds your markup. You can find the CEWP under the Media and Content category when inserting a web part via the Ribbon, as shown in Figure 3-1.

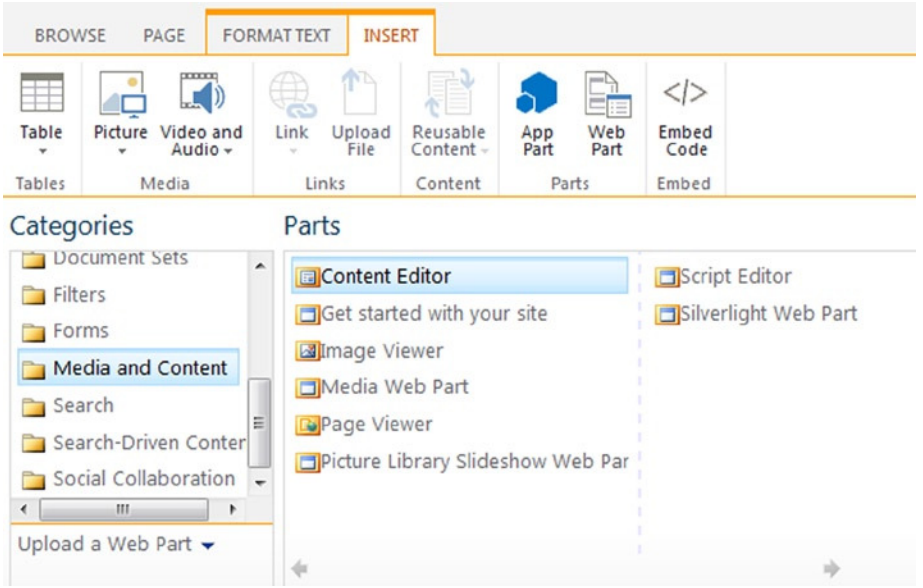

Figure 3-1. *The Media and Content category where you can find the Content Editor Web Part*

Insert a CEWP into your page and place it in edit mode. You will see that there are three ways that you can add and edit content, described as follows and as shown in Figure 3-2.

- *Rich content* - When you first add a CEWP to a page, there is obviously no content there. In this case, you will see a link titled "Click here to add new content". Clicking this link will place a cursor directly into the web part, and you can use the Ribbon to add rich content—just as you would in Microsoft Word. This is an easy and convenient way to add content to a page, but it is not a good way to build custom solutions that uses SharePoint data.
- *HTML markup* - Another way to insert content is by clicking the "Click here to add new content" link, and then clicking the Edit Source button in the Ribbon. This will open a dialog where you can directly type in HTML content into the CEWP. This is a great option for quick HTML inserts or testing your HTML markup. This is a better option for building custom solutions, but not quite the best.
- *Link to an HTML file* - You can also link directly to an HTML file to display content in the web part. In the CEWP tool pane, the first property is titled Content Link. In this textbox, simply add the URL to an HTML file, and its contents will be displayed when the web part is rendered. This is the preferred usage when building custom solutions, which we'll look at in the next chapter.

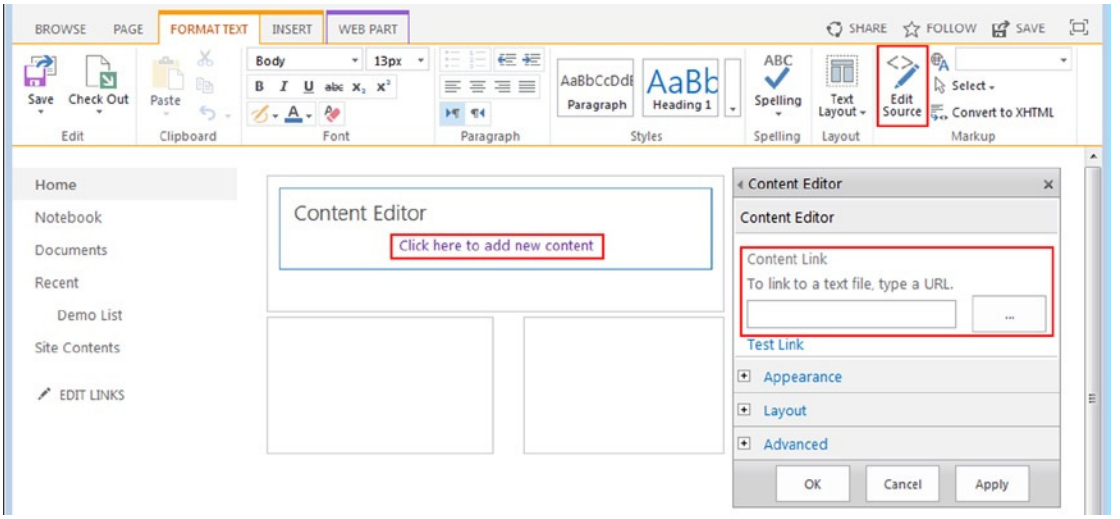

Figure 3-2. *The various ways you can add content in a Content Editor Web Part*

For now, let's place some HTML in using the Edit Source button from the Ribbon. Place the CEWP into edit mode, click the "Click here to add new content" link, and then click the Edit Source button in the Ribbon. Figure 3-3 shows the dialog that is displayed after clicking this button.

Figure 3-3. *Dialog that is presented after clicking the Edit Source button in the Ribbon*

In this dialog, we can simply type our HTML content. In this example, all we'll add is a `DIV` element that has an ID and some text inside it. The following HTML is all that is needed:

```
<div id="divHelloWorld">Hello World!</div>
```

Adding this content and saving the page will result in a nice "Hello World!" message displayed on the page, as shown in Figure 3-4.

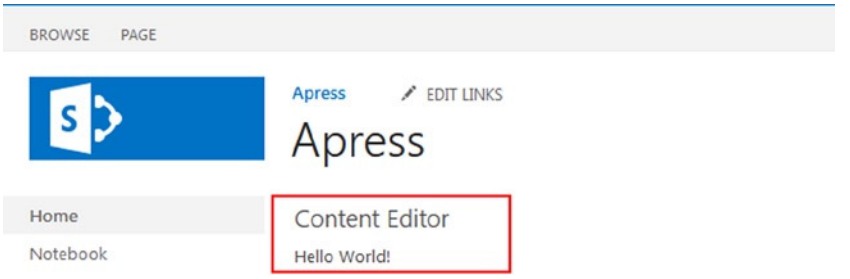

Figure 3-4. *HTML content displayed in the Content Editor Web Part*

This is a very simple and not very exciting example of adding HTML content to a page. However, if you're new to adding in custom HTML, this is a great starting point for what is to come later in this chapter and in further chapters. Now that we have some HTML on the page, let's take a look at the Script Editor Web Part and learn how we can inject custom JavaScript into the page along with our HTML.

Script Editor Web Part

The Script Editor Web Part (SEWP) is new to SharePoint 2013; it was designed just for what its name implies—adding scripts to a page in SharePoint. Prior to this web part being on the scene, if you wanted to inject a script on the page, you would use the CEWP instead. This worked great if you linked to a file that contained the script, but if you wanted to edit the source and place the script in manually, you often encountered issues. At times, scripts would not run, or would throw errors, or worse, the CEWP would actually change the code you entered. It was not a pleasant experience.

With the SEWP these are no longer issues that you need to worry about. You can now easily place JavaScript on a page using the SEWP, because it was built for this purpose. You can also place other items with this web part, including OBJECT tags, HTML, and more. You can find the SEWP under the Media and Content category when inserting a web part via the Ribbon, as shown in Figure 3-5.

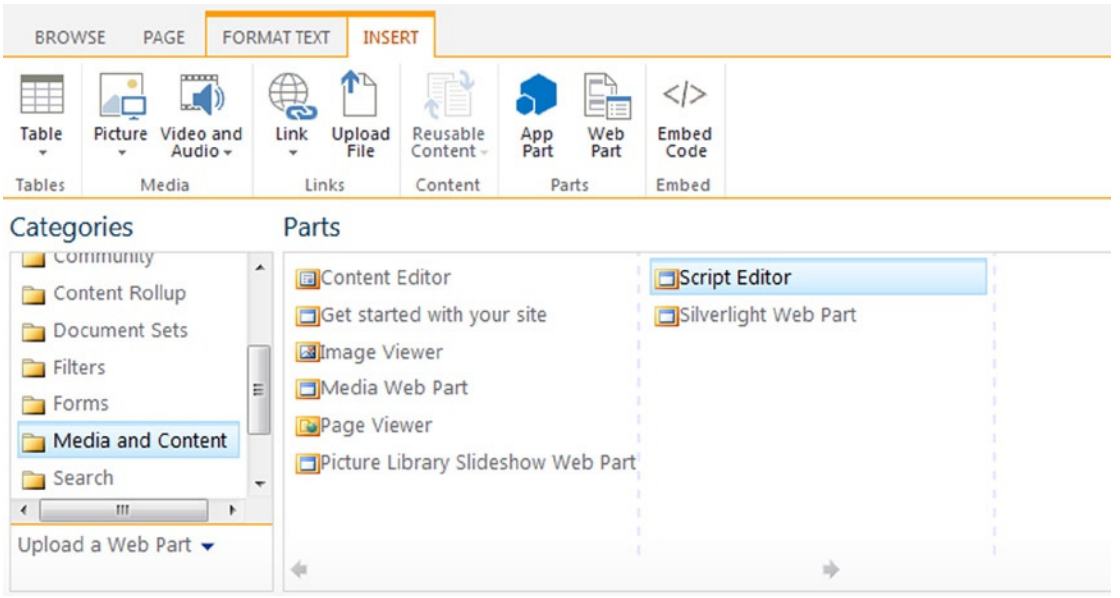

Figure 3-5. *The Media and Content category where you can find the Script Editor Web Part*

Insert a SEWP into your page and place it into edit mode. You will see that there is one way that you can add and edit scripts—by clicking the Edit Snippet link, as shown in Figure 3-6.

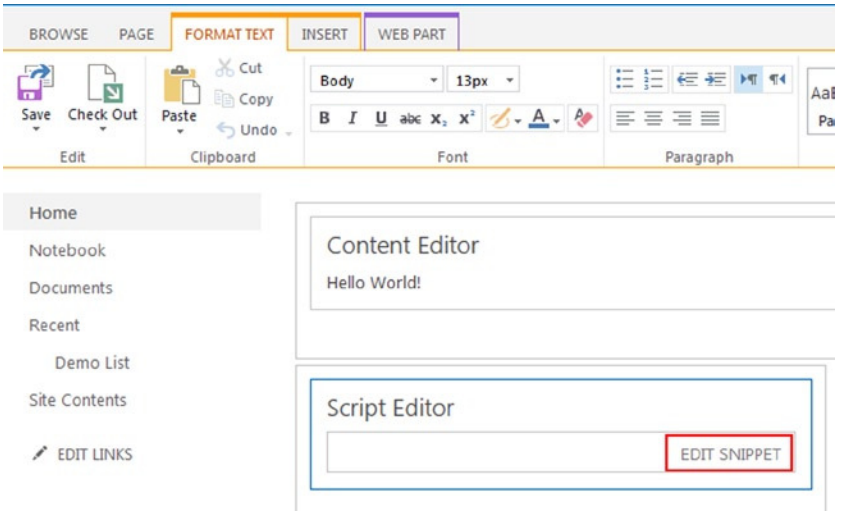

Figure 3-6. *Click Edit Snippet to add or edit your JavaScript*

Clicking the Edit Snippet link will open a dialog titled Embed, where you can write your JavaScript, as shown in Figure 3-7. JavaScript placed into this web part must be valid and include the `<script></script>` tags in order to function properly. The SEWP can even be used to place HTML on the page as well. If you are placing HTML, the dialog will show you a preview of the content below the textbox that you are working in. If you are writing JavaScript, no preview will be shown because that code is typically not visible in the page.

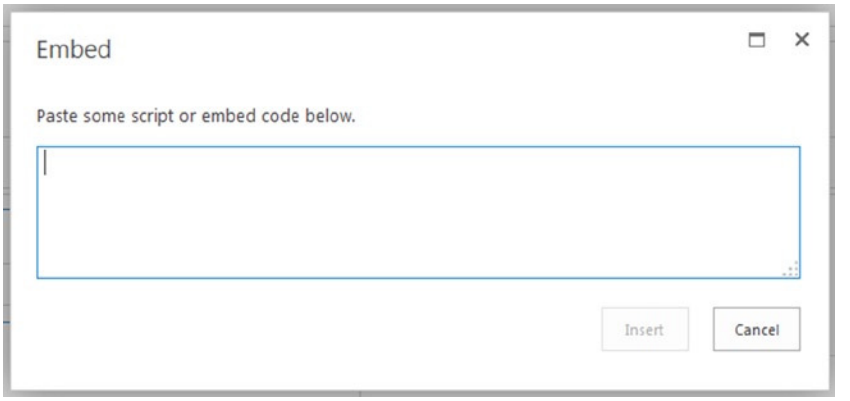

Figure 3-7. *Embed dialog for writing scripts inside the Script Editor Web Part*

■ **Note** Even though the Script Editor Web Part can be used to inject HTML, you should think of this web part as for scripts only. It's a much better practice to separate your HTML and scripts into separate web parts. That being said, it can be very useful if you just need to perform a quick test of a script.

In this example, we'll add some JavaScript using jQuery that will display an alert when the page is loaded:

```
<script type="text/javascript">
$(function() {
  alert("Hello World!");
});
</script>
```

As you can see, this code includes an opening and closing `<script>` tag with a jQuery document.ready function inside. All that this script is doing is firing an alert with the text "Hello World!" inside it when jQuery detects the page has been loaded. Place this code in the dialog and click Insert. Save the page. When the page is refreshed, you will see that the alert fires, as shown in Figure 3-8.

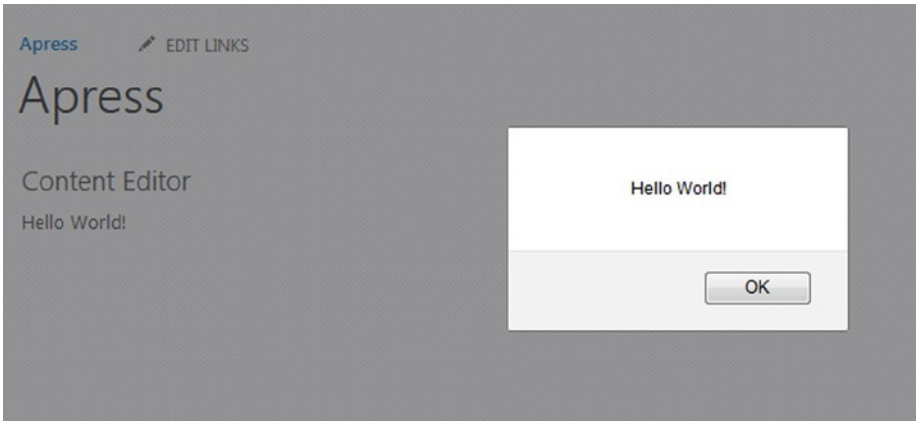

Figure 3-8. *JavaScript alert being shown via the Script Editor Web Part*

Much like the example using the Content Editor Web Part, this is quite simple and not exciting. However, at this point you should now have a CEWP and a SEWP on your page. Next, let's look at how we can wire them together for a complete solution.

Putting Them Together with the REST API

Now that we've placed a CEWP and a SEWP on our page, we can wire them up to work together. In this example, we'll use the REST API to pull some data from a list in our site. We'll use jQuery to make an AJAX call via the SEWP, and then update the CEWP after the call has been made. We'll look at how you can use these two web parts together on a page, as well as how to make an AJAX call with jQuery. First, we'll need a list to call to get data from.

Demo List

Throughout this book we'll be using JSOM and the REST API to query data from SharePoint. To help facilitate this, we'll create a new custom list that we can use in our efforts. We'll keep it fairly simple: just create a new custom list and add an additional column to it. No frills here. To start, just add a custom list using the Add an App functionality and then name this new list **Demo List**.

■ **Note** This section assumes that you know how to create a new list in SharePoint and how to create a new column. If you need assistance with creating a new column, please visit the following URL: `http://office.microsoft.com/en-us/sharepoint-server-help/create-change-or-delete-a-column-in-a-list-or-library-HA102771913.aspx`.

Once the list has been added, it will have a single column in the default view, which is Title. This is good, but let's add one more—just so that we have some extra data to work with. Go to List Settings and add a new column using the "Create column" link on that page. Choose a type of Multiple Lines of Text and title this column **Description**. In my list, I made this a Plain Text column just to keep it simple. Figure 3-9 shows the Demo List after some data has been added to it.

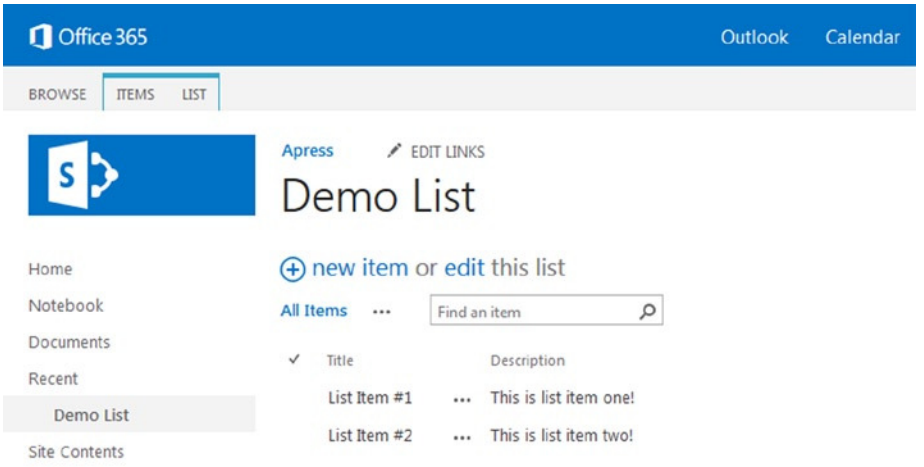

Figure 3-9. *Demo List ready to be queried for data*

Just insert a couple of rows of data for now. The actual data in the list is not important for now; it's really just there to ensure that we can query it and retrieve list items. Now that we have a list, let's look at the code needed to query it via the REST API.

jQuery, AJAX, and REST API

Querying the list will be done by making an AJAX call to the REST API using jQuery. In the previous chapter, we took a quick look at the structure of the REST API URL. A typical URL structure to query a list looks like this:

```
http://DOMAIN/SITE/_api/web/lists/GetByTitle('List Name')/items
```

All REST API URLs start with `_api` after the site where the data resides, or immediately following the domain name if querying the root site collection of SharePoint. The `_api` is an indicator to SharePoint that you are requesting data via REST. The data can be requested in either XML or JSON format, but for most JavaScript calls, it will be JSON. A nice benefit of using the REST API is that you can craft your URLs in any browser and test them out before using them in your code!

■ **Note** In my environment, I have created a subsite under the main site collection titled Apress. This site is the area where I'm writing and running all the code in this book. As a result of that, you will see URL structures that look like the following: `/apress/`. If you are working at the root site in your site collection, then your URL references would begin with `/` only. Take note of your URL and ensure that your references are correct! Later, we'll write some code that does all this for you, but for this section it's important to have the correct URL structure.

To make the call via the browser, we'll use the standard jQuery AJAX call. The following code demonstrates this call:

```
<script type="text/javascript">
$(function() {
  $.ajax({
    url: "/apress/_api/web/lists/GetByTitle('Demo List')",
    type: "GET",
    headers: {
        "accept": "application/json;odata=verbose"
    },
  }).success(function(data) {
    alert(data.d.Title + " : " + data.d.Created);
  });
});
</script>
```

In this example, we wrap the call in the now familiar document.ready function. The AJAX call is made using the `$.ajax()` function. It's very straightforward and only includes a few properties:

- `url` - This is REST API URL that has been crafted to request whatever specific data you need. In this example, all we are requesting are the list properties.
- `type` - The type represents the standard HTTP methods, IE: GET, POST, PUT, and DELETE. For this example, we are simply requesting data, so we use a GET.
- `headers` - As mentioned earlier, the REST API can return XML or JSON. If you do not specify which format you want, it will return the default, which is XML. By specifying a header of `"accept": "application/json;odata=verbose"`, we tell the REST API that we would like JSON back.

At the end of the `$.ajax` call, we attach a `success` function which processes the "data" that is returned from the REST API. jQuery makes it very easy to handle AJAX calls and perform various functions based on what happens with that call. For instance, you can attach other functions, like a `failure` and a `finally`. We'll look at these in more detail later on.

In our `success` function, we pass in the `data` object that is returned from the REST API. All we're going to do right now is pop up an alert with the list title and the data it was created. We do this with the following lines of code:

- `data.d.Title`
- `data.d.Created`

The `data` object is all the data that is returned from the query. There is a lot in this object that we won't use. The `d` property holds all the list properties that we are looking for. It may look strange, but this is how you will access all the data returned via the REST API. For instance, if you request list items in a query, you would access `data.d.results`. Placing this code into the SEWP and saving the page will result in the alert shown in Figure 3-10.

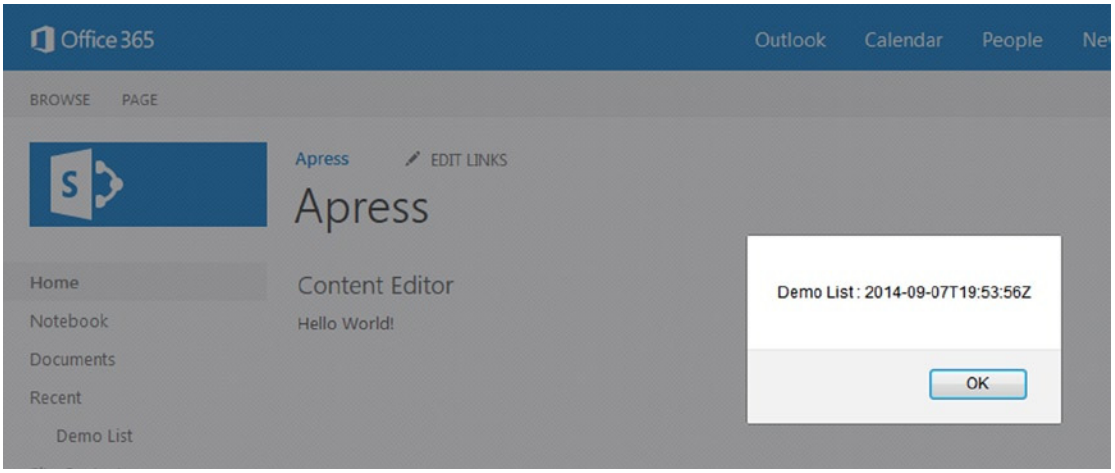

Figure 3-10. *JavaScript alert showing the Demo List title and date created*

Now that we have the AJAX call wired up, and data is being returned, you should see the annoying alert pop-up. This is great, as it lets us know that everything is working correctly. However, in real life you probably won't be making REST calls just to pop up alerts. Let's get this data into the CEWP with a little more jQuery magic.

Wiring It All Up

At this point, you should have a CEWP and a SEWP on your page. The CEWP is showing some basic HTML, and the SEWP is making a REST API call and popping up an alert. Now it's time to make them work together by having the SEWP make the call to get the list data, and then update the HTML in the CEWP with the results. After this example, you should see how easy it is to have these two web parts work together to create solutions.

Luckily, we do not need to make any changes to the CEWP! If you used the HTML from this section of this chapter, you are all set. If not, it should look like this:

```
<div id="divHelloWorld">Hello World!</div>
```

The key to this line of code and making the web parts work together is the ID attribute in the DIV tag: `"divHelloWorld"`. By giving the DIV an ID, we can very easily use jQuery to find the element in the HTML and update its content. There are other ways to do this as well; for instance, we could have place a CSS class on the DIV and searched that way. Either way, as long as we can easily find the element, we can easily update it!

Now let's make an update to the SEWP script to update the CEWP. The change is quite small, but should make a big impact. Update your script with the following code; the only change is in the `success` function, where the alert is replaced with some new jQuery code:

```
<script type="text/javascript">
$(function() {
  $.ajax({
    url: "/apress/_api/web/lists/GetByTitle('Demo List')",
    type: "GET",
    headers: {
        "accept": "application/json;odata=verbose"
    },
  }).success(function(data) {
    $("#divHelloWorld").html(data.d.Title + " : " + data.d.Created);
  });
});
</script>
```

This new line simply searches for an HTML element with the ID `"divHelloWorld"`. The # symbol is used to instruct jQuery that we are looking for an ID, rather than some other attribute. Updating the SEWP with these changes and clicking Save on the page will result in the CEWP's HTML being updated, as shown in Figure 3-11. We show the date created in the results; your date will be different from what is shown in the example.

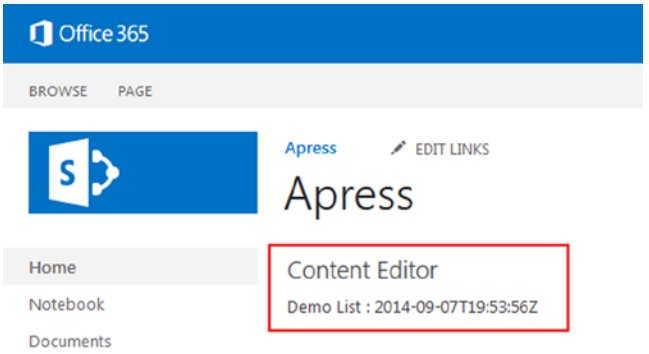

Figure 3-11. *Content Editor Web Part has been updated via the Script Editor Web Part*

If you keep your eyes on the CEWP, you can see the "Hello World" message on the screen before it is updated to the list information. This is because we are using the jQuery document.ready function, which will not fire until the page (document) is loaded. If you didn't catch it the first time, simply hit refresh in the browser to see it again. In a real-world scenario, your HTML may simply be empty to avoid this type of visible change.

Not a One-Size-Fits-All Solution

The CEWP and SEWP are quick and easy ways to inject custom HTML and scripts into your page. At this point, you may think: What else do I need? I can start building custom solutions right now! Certainly these two web parts give you a great opportunity to build solutions for your users, they are easy to use, and they work great together. However, this solution is not necessarily a great option for more complex custom development.

As you progress through your development efforts, not only in this book but in your own efforts, you may find that the solution needs become larger and more complex. You may find that you need multiple JavaScript files, custom style sheets, or maybe even multiple HTML files. Such situations would make using the CEWP and SEWP quite unruly and very difficult to manage. In addition, you may find that you need to have the same functionality used across multiple pages. Having to place multiple web parts on multiple pages and wiring them up can pose a challenge. It also opens the door for other users to modify your HTML and JavaScript. Remember, anyone who has the permission level to edit web parts can edit this code, which could be a very bad thing.

This solution can be quite effective, but it should be used sparingly. It's a great solution for simple one-off scripts that perform little work, or simple HTML to display limited amounts of data. Since it's a fast solution, it's perfect for testing some HTML and scripts to ensure that they work correctly. No matter your decision to use this method, take a moment to think about whether it's the best choice for your needs and your users.

Summary

In this chapter, we explored the Content Editor Web Part and its new counterpart, the Script Editor Web Part. We looked at how you can place custom HTML and JavaScript on your pages using these web parts, and how easy it is to do so. You learned how to wire up an AJAX call via jQuery to the REST API and retrieve data about a list. Finally, we looked at how to marry the two web parts into a single solution.

In the next chapter, we'll continue down the path of building custom solutions using what we've learned so far. We'll take a look at how to move this development into a more maintainable structure and get our hands on some more advanced scripts. We'll also expand on the REST API calls, as well as JavaScript Object Model calls.

CHAPTER 4

Building Custom Web Parts Using HTML and JavaScript

The Content Editor Web Part combined with the Script Editor Web Part allows you to quickly build solutions in your environment. While this combination certainly gives you options for developing a solution, it's not an easily scalable or reusable solution. You may find it suits your needs for single-page solutions that are simple, but once your needs become more complex, you'll find that solution limiting in many ways. Luckily, what you learned in the previous chapter can be expanded into more robust solutions.

In this chapter you'll learn how to build robust, reusable web parts using the Etherson method. This method still works with HTML and JavaScript, expanding on the lessons from the previous chapter. You'll see how to better structure your code and how this method will allow you to tackle just about any custom solution you can think of.

The Etherson Method

Over the past several years I've been fortunate enough to work as a consultant on many exciting SharePoint projects. In addition, I've been able to work with some of the most talented people—who push me every day. One of these people is Mike Etherigde. Mike and I worked on a SharePoint 2010 public-facing web site that needed to be responsive for mobile devices. It also needed to connect to external databases that were not integrated into SharePoint. Mike's solution was to build custom WCF web services that could be called via JavaScript. These services were then deployed to the server along with custom HTML and JavaScript. The HTML was consumed via an XML Viewer Web Part that was placed on each page.

This solution worked great. It was easy to develop with and fast to deploy. It allowed us to easily bring data together on the server and return it to our pages, and the HTML could be easily served from the SharePoint server. This worked great in SharePoint 2010 and would work great in SharePoint 2013, but not SharePoint Online because no server-side code can be deployed there. Luckily, since Microsoft expanded the JavaScript Object Model and REST API, the solution could be tweaked to allow for a full development life cycle outside the server—and the Etherson method was born.

The Etherson method is made up of the following components:

- Document library
- Folders
- Content Editor Web Part
- HTML, JavaScript, and CSS

As you can see, this solution only requires a few components to make a big impact. At first glance, you may be thinking how could you possibly build robust solutions with such a simple set of items? The real power of this method lies in its simplicity and the power of the JavaScript Object Model and REST API. Let's take a look at each component in detail.

Document Library

At the heart of this method is a *document library*. The library will act as the main repository for all code used in your custom web parts. All code will be stored in this library, with a folder for each web part. You can use multiple libraries if that suits your needs. For instance, you could have a library at the root of your site collection that is used for all your subsites. You can go more granular and put a library in each subsite that is only used for that site. You can be quite flexible in how you structure this, as it's simply a place to put your code.

Setting up the library will require the following steps:

1. Create a new document library.
2. Turn off search indexing.
3. Turn on versioning.
4. Place unique permissions on the library.

First, we'll simply create an out-of-the-box document library and call it **Webparts**, as shown in Figure 4-1.

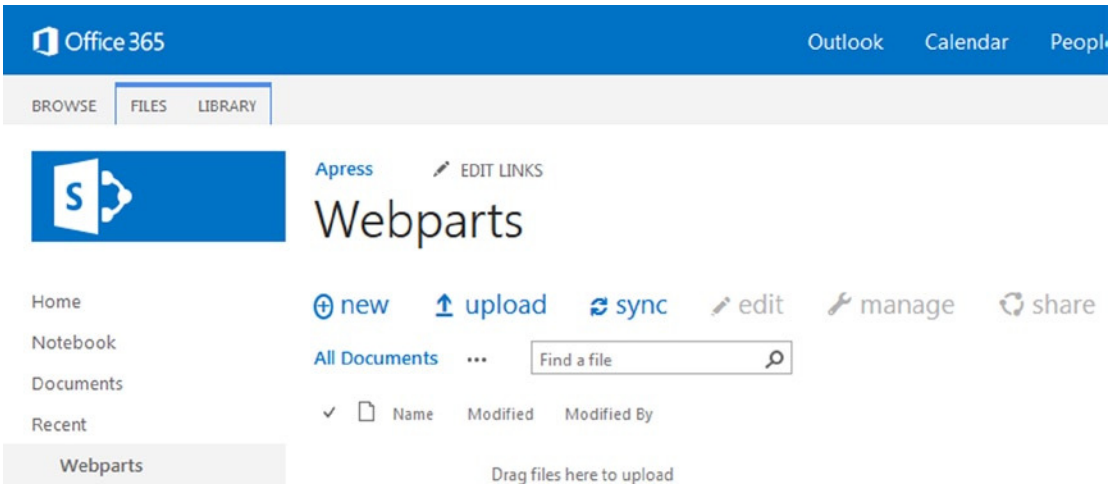

Figure 4-1. *A new out-of-the-box document library for our code*

Nothing fancy about this: we simply add it via the Settings menu using the Add an App option and choose Document Library. Next, we'll ensure that the search engine does not index this library. The reason for this is that the library will be used to store code. The average user has no need to see this code because it's used to render custom web parts; thus, having it show up in search results will muddy up the user's search experience. Turning off search indexing will ensure that the code files stay out of search results. Access the Library Settings from the Ribbon and choose Advanced Settings, as shown in Figure 4-2.

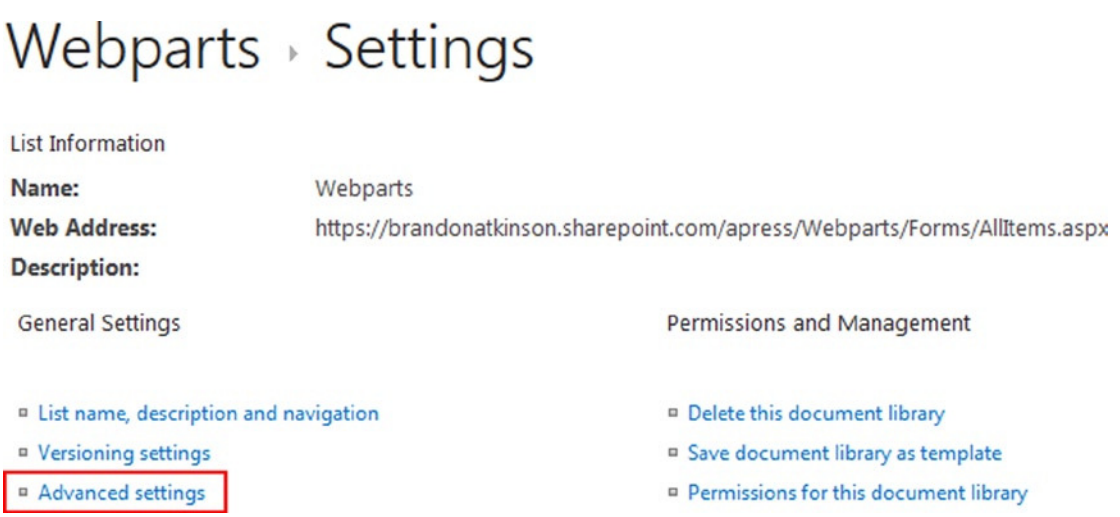

Figure 4-2. *Accessing the Advanced Settings page in the Library Settings*

On the Advanced Settings page, you'll find many options for the document library. Look about halfway down the page for the Search heading. There are only two options: allow items to appear in the search results or not. Click No, as shown in Figure 4-3.

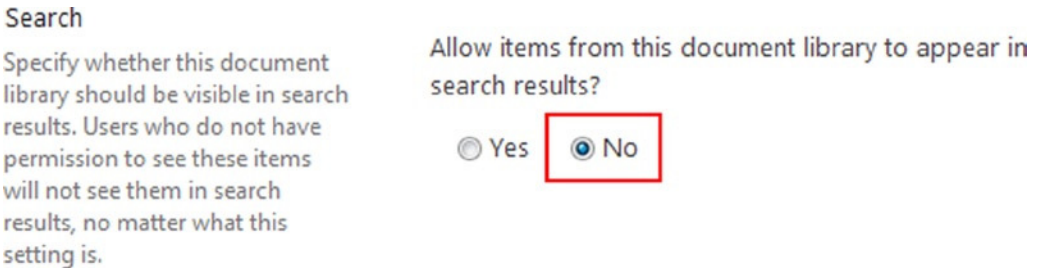

Figure 4-3. *Turning off the search indexing for the document library*

Scroll to the bottom of the page and click OK to save your changes. You will be brought back to the Settings page, where you can move on to versioning. Versioning is important because it allows for easily rolling back code. This provides you some peace of mind if you need to implement changes to existing code. For instance, imagine you have developed a solution that meets the current business needs and everything is working fine. You are asked to make a big change and develop new code to meet the requirements. You test your code and everything looks great. You upload the changes—and the users immediately start to complain that something is wrong! You forgot to test in different browsers and your code is breaking for everyone using Chrome.

If you have versioning turned on, you can simply go to the document library and restore the previous code files. Just like that, users are presented with the previous code and at least everything is working again. As you can see, having the ability to quickly and easily roll back code changes will prove very helpful at some point in your development journey.

To turn on versioning, click the Versioning Settings link, as shown in Figure 4-4.

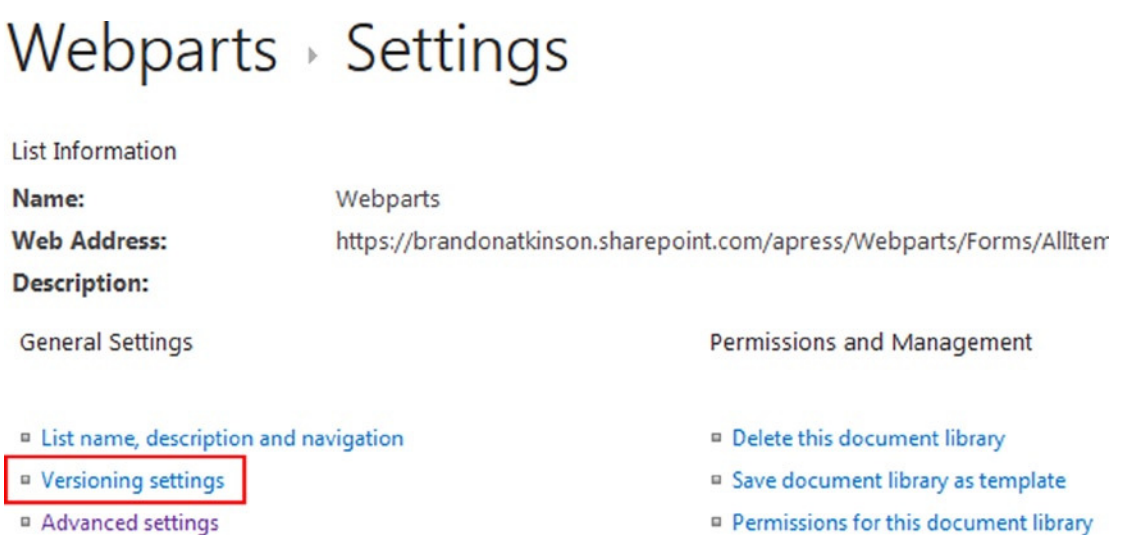

Figure 4-4. *Accessing the Versioning Settings for the document library*

Once on the Versioning Settings page, click the Create Major Versions option under the Document Version History heading, as shown in Figure 4-5. You could also create major and minor versions; however, for this book we'll keep it simple and just use major versions.

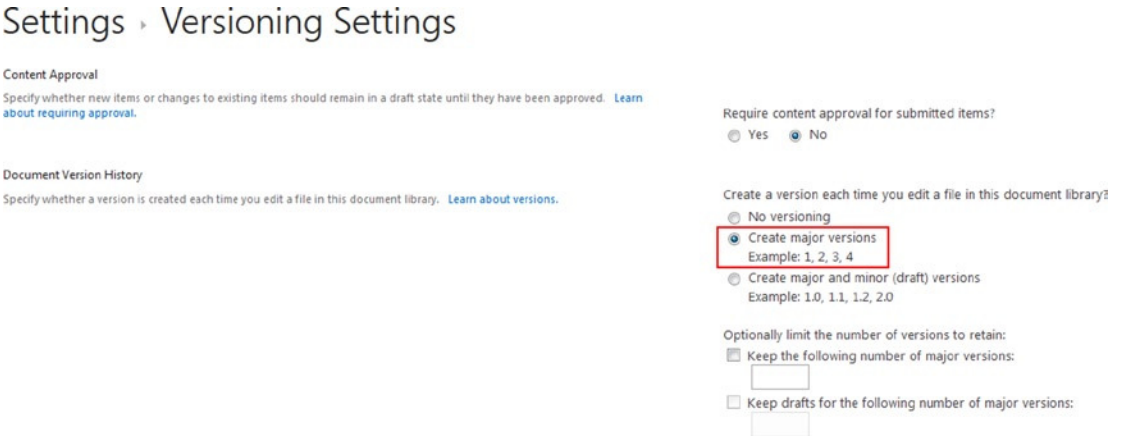

Figure 4-5. *Turning on Create Major Versions in the document library*

Scroll to the bottom of the page and click OK to save the changes. You will be brought back to the Settings page, where you can move on to permissions.

■ **Note** Depending on the type of site you have created, you may find that versioning is already turned on. For instance, a publishing portal or a team site with the Publishing infrastructure turned on will already have versioning on. In either case, open the versioning settings page to ensure that it is turned on.

Finally, you must secure the document library by applying unique permissions. We want to secure this library to control who can add or modify the code that has been placed into it. Users will need to have read access so that the code can be loaded into the web parts; but you only want developers to have contribute access. This is crucial, because you only want a small set of people with access to upload code and potentially delete code. To modify permissions, click the "Permissions for this document library" link, as shown in Figure 4-6.

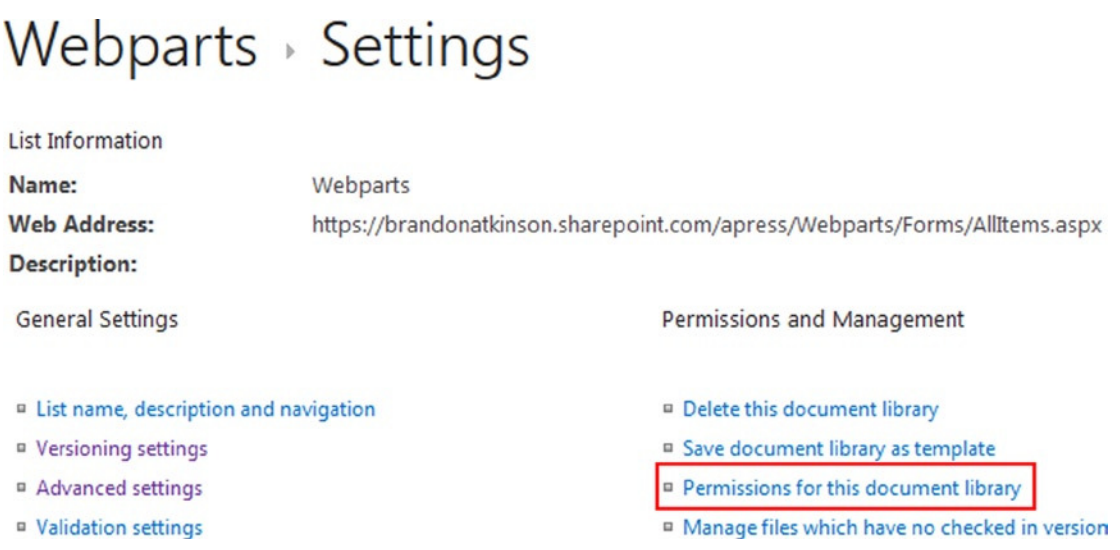

Figure 4-6. *Accessing the permissions settings for the document library*

Clicking this link brings you to the permissions page for the document library. To apply unique permissions, click the Stop Inheriting Permissions button in the Ribbon, as shown in Figure 4-7.

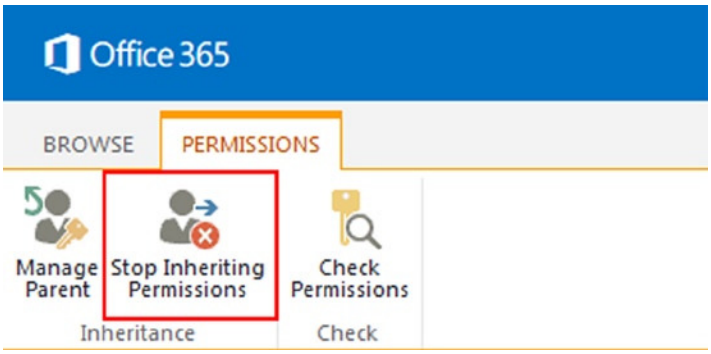

Figure 4-7. *The Permissions tab in the Ribbon for the document library*

Normally, all items in SharePoint inherit the permissions of its parent. When you create this new document library, by default it has the permissions of the site where it is located. If you choose to stop inheriting permissions, SharePoint really wants to make sure that this is your intention and prompts you with a confirmation dialog, as shown in Figure 4-8.

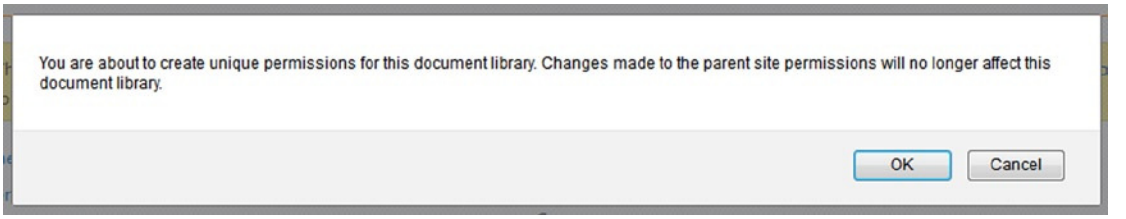

Figure 4-8. *Confirmation dialog when breaking permissions*

Click OK to confirm the change and you'll be brought back to the permissions page. You'll notice the Ribbon now has some additional options, most notably the Grant Permissions button, as shown in Figure 4-9. You'll use the Grant Permissions option to add permissions back in.

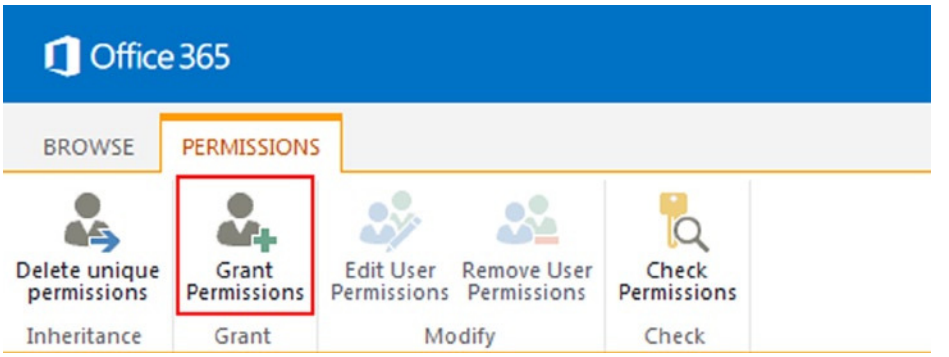

Figure 4-9. *Grant permissions using the Grant Permissions button in the Ribbon*

■ **Note** Once you stop inheriting permissions, you should remove all the users and groups currently in the list. Even though you've broken permissions, the previous users and groups will remain unless manually removed.

Clicking the Grant Permissions button launches the Share dialog, where you can choose who you want to grant permissions to, as shown in Figure 4-10. The first text box in the dialog allows you to specify which users or groups you would like to have access to the library. In this step, you want to grant everyone read access. As you type into the text box, SharePoint provides an autocomplete list of people and groups to choose from. Type **Everyone** and then choose it from the suggested list.

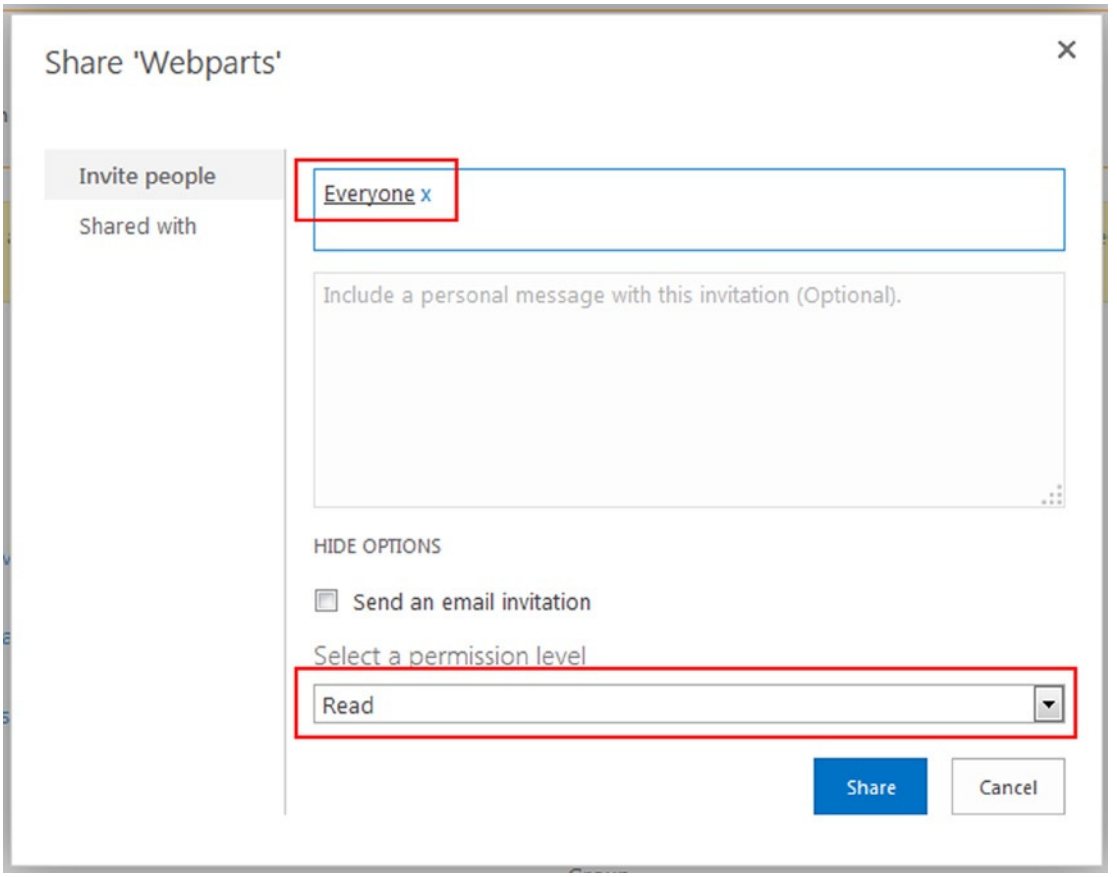

Figure 4-10. *Granting Read access to all users*

You can click the Show Options link to expand the dialog and see the additional options available to you. You may want to uncheck the Send an Email Invitation option, especially when adding large groups. In this case, it would send an email to everyone who has access to SharePoint, which is probably not a good idea. Another option is the Select Permission Level drop-down; choose Read since we want everyone to have that particular level of access.

Click the Share button to save the custom permissions. At this point, all users now have Read access to the library. Repeat these same steps for any developers and give them the permission level of Contribute. This way, those users can upload new code and modify existing code in the library.

■ **Note** If you are using a development environment or a site that is only accessed by a few people, you can skip applying unique permissions. This is crucial in a QA or production environment, but a development environment can be less stringent here.

Now that the document library has been created and configured, we can move on to the other components, like folders. Outside of writing code for your solutions, setting up the document library is the biggest task using the Etherson method. The remaining components will seem simple in comparison.

Folders

Using this method, each custom web part has a folder that contains all the files needed for that web part. You can also have a centralized folder that contains scripts that are shared across all web parts. For instance, if you wanted to store a copy of jQuery in a folder, you could avoid adding it in via the MasterPage and just have each web part reference it from the central folder. You may also have some utility scripts that perform common functions for multiple web parts. No matter the need, a central folder may come in handy.

Navigate to the Webparts document library that you created in the previous section. Using the New command in the menu, choose New Folder from the context menu, as shown in Figure 4-11.

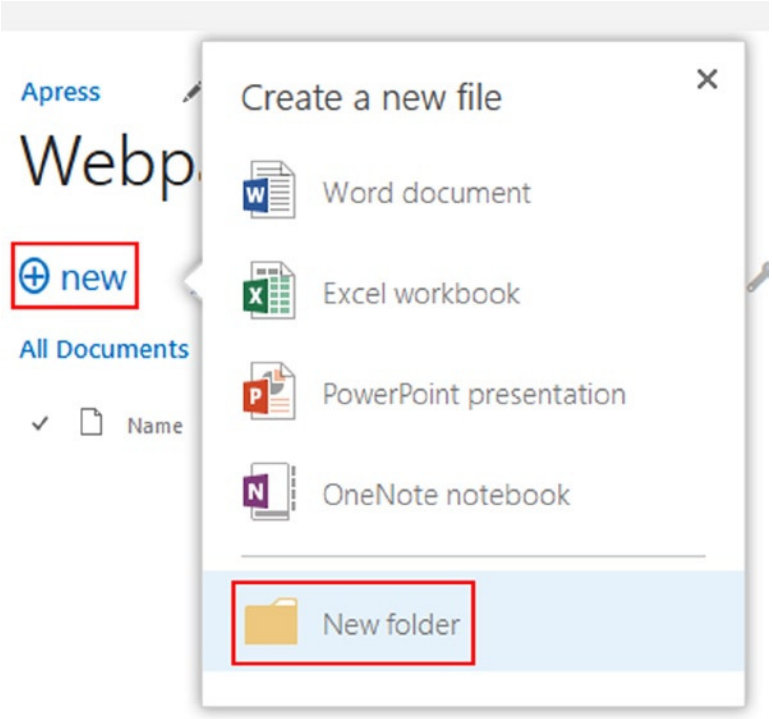

Figure 4-11. *Creating a new folder*

Create a new folder and name it **Simple Example**. We'll use this folder to host the HTML and JavaScript for our first example. At this point, you should have a single folder in the Webparts document library. You'll add more folders as you progress through this book; but for now, it's a good start.

HTML, JavaScript, and CSS

Now that we have a document library and a folder to host our code, let's put together an HTML file and JavaScript file. For this section, we'll use Visual Studio to write our code and to build our HTML and JavaScript files. If you installed Visual Studio Express in the first chapter, you're all set. You can use any code editor you like—even Notepad! Launch Visual Studio and create a new file by clicking File ➤ New ➤ File, as shown in Figure 4-12.

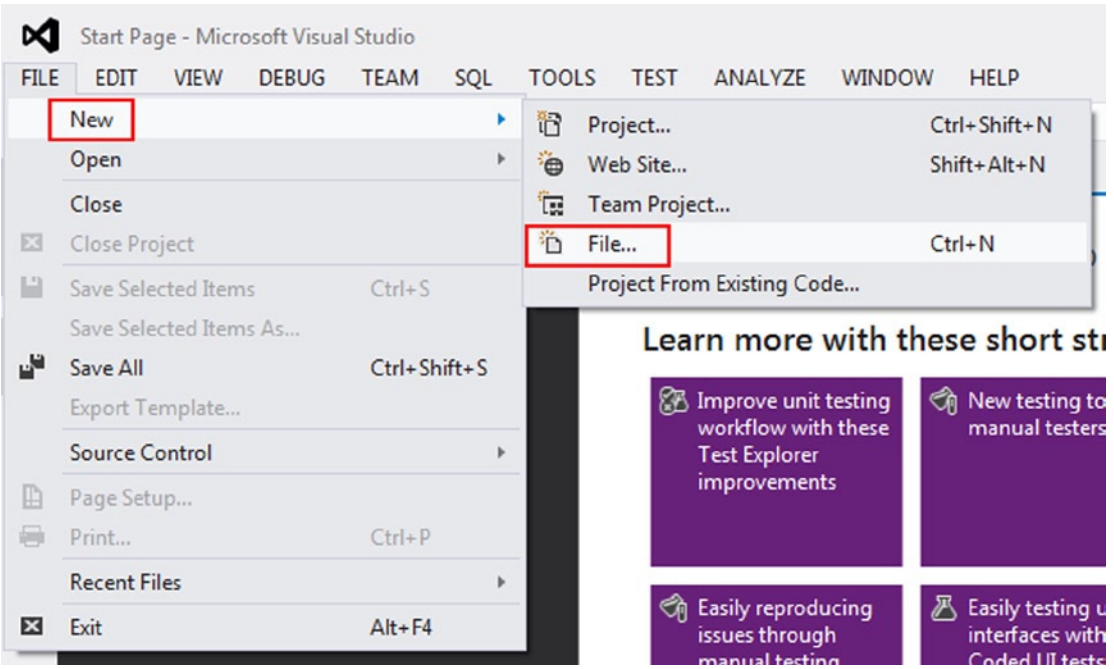

Figure 4-12. *Creating a new file in Visual Studio*

This will open a new dialog screen where you can choose the kind of file you want to create. There are a lot of options here, but the one we're looking for is right up front. Click HTML Page from the list, and then click Open, as shown in Figure 4-13.

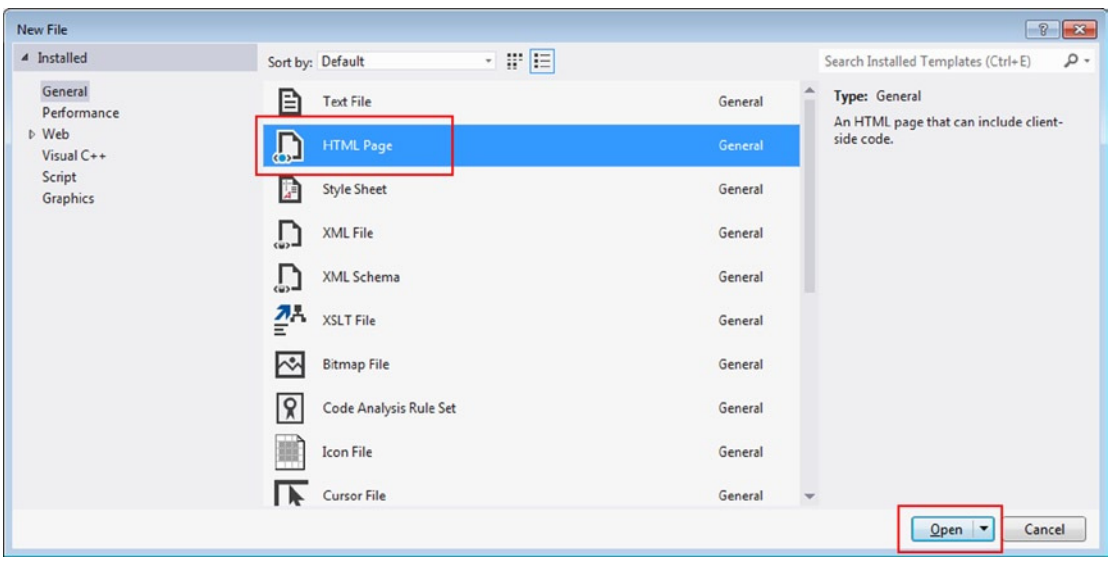

Figure 4-13. *Creating a new HTML page from the New File dialog*

Once you click Open, you'll have a new HTML file created and displayed. If you're new to Visual Studio, you'll get a very small taste of the efficiencies that a code editor will bring you. The newly created HTML file has some code already generated for you, as shown in Figure 4-14.

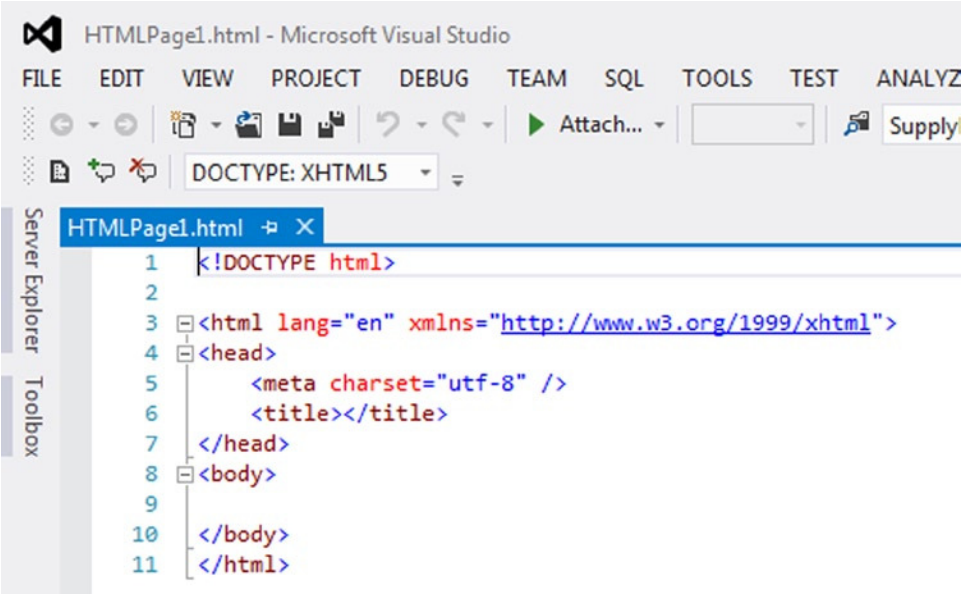

Figure 4-14. *Newly created HTML file with code*

If you were to save this file and load it into SharePoint, it wouldn't display anything because there is no content to render to the user in the browser. However, it is a perfectly valid HTML file! Although this is not overly impressive, you'll find a lot of things like this that Visual Studio helps with as you write code.

For this example, we'll start off using the code from Chapter 3, except we'll split the code into an HTML file and a JavaScript file. Start by adding the following line to your new HTML page:

```
<div id="divHelloWorld">Hello World!</div>
```

Place this line between the two `<body>` tags, as shown in Figure 4-15.

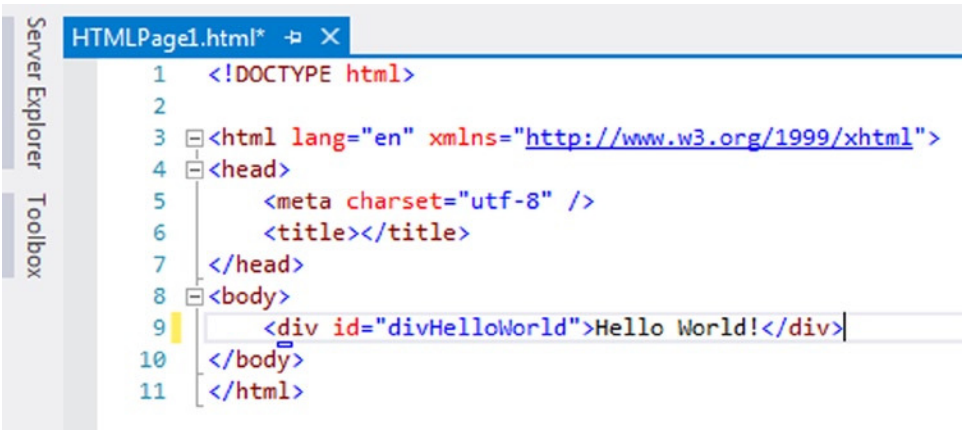

Figure 4-15. *New HTML file with code from the previous chapter*

Now let's create a JavaScript file to host our script. As before, choose File ➤ New ➤ File from the menu. When the New File dialog opens, you'll notice that there is no option listed for creating a new JavaScript file. For this we'll need to choose Web from the left navigation to view all the file templates for web sites. Scroll to the bottom of the templates listed, and you'll find the JavaScript File template, as shown in Figure 4-16.

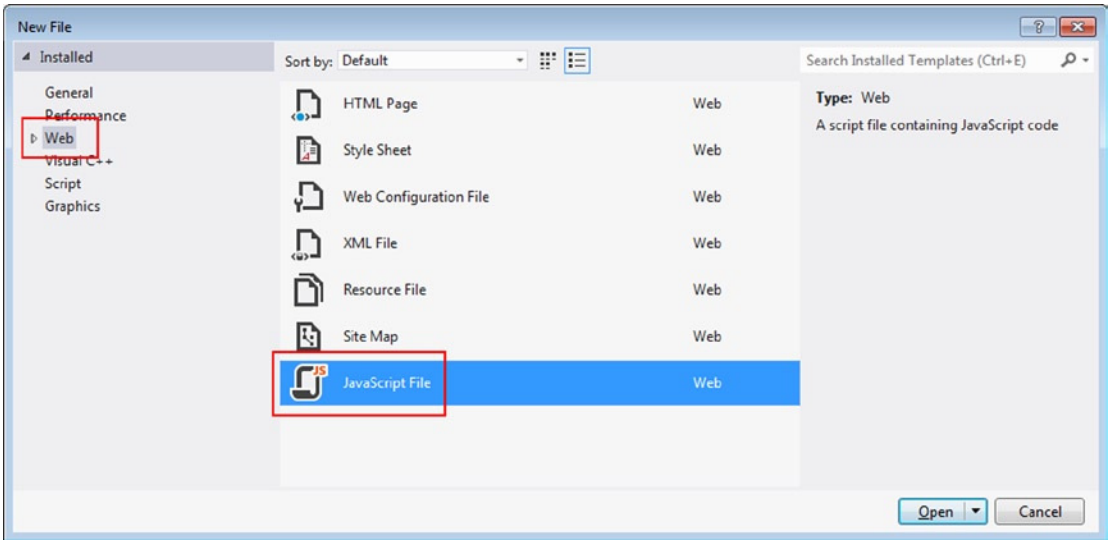

Figure 4-16. *JavaScript file option in the New File dialog*

Once you create the new JavaScript file, you'll notice that it looks very different from the HTML file you created before. In fact, it's completely blank, as shown in Figure 4-17.

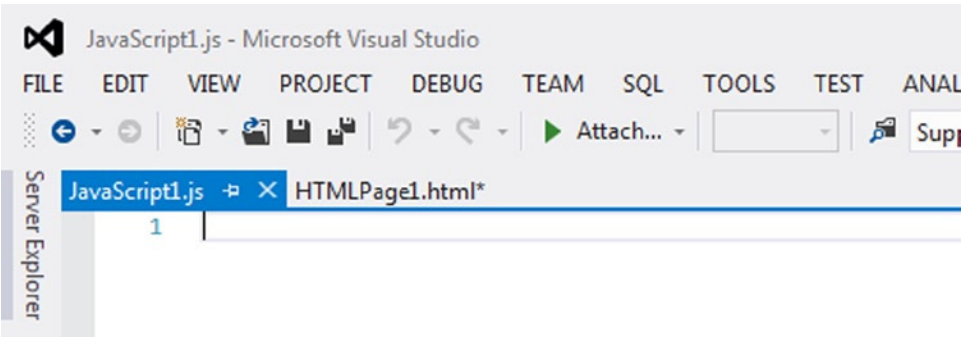

Figure 4-17. *New JavaScript file created in Visual Studio*

As with the HTML file, you'll use the same code from the Chapter 3 example. Place the following code into the new JavaScript file you just created:

```
$(function() {
  $.ajax({
    url: "/apress/_api/web/lists/GetByTitle('Demo List')",
    type: "GET",
    headers: {
        "accept": "application/json;odata=verbose"
    },
  }).success(function(data) {
    $("#divHelloWorld").html(data.d.Title + " : " + data.d.Created);
  });
});
```

■ **Note** In all the code examples in this book, the URLs are for my environment, which includes a root site collection and a subsite titled **Apress**, where all of my code resides. Thus, all of my URLs will appear as "/apress/….." You should update your URLs accordingly where your code is deployed, as well as the structure of your sites.

You'll notice that unlike in the Chapter 3 example, you do not have the opening and closing `<script></script>` tags wrapping your code. Since the code is being served from a JavaScript file with a `.js` extension, the browser does not need to be told it's a script with these tags, as shown in Figure 4-18.

Figure 4-18. *JavaScript code in Visual Studio*

Now that both files are created, you'll save them. Save these files wherever you like on your computer. Name the HTML file **SimpleExample.html** and the JavaScript file **SimpleExample.js**. With the files saved, we now need to wire up the JavaScript file to the HTML file so that the code will fire appropriately. To do this, we need to add a JavaScript reference to the HTML file, similar to the following line:

```
<script type="text/javascript" src="/apress/webparts/Simple Example/SimpleExample.js"></script>
```

In my environment, I have a subsite titled Apress, which is where my Webparts document library is located. This script reference begins with a /, which means use the root URL of the site. Next, we place the subsite name, the document library name, the folder name, and finally, the script file name. Depending on how you've set up your environment, you may not need the subsite reference, or it may be titled something different.

Place the script reference in the <head> of the HTML file, as shown in Figure 4-19.

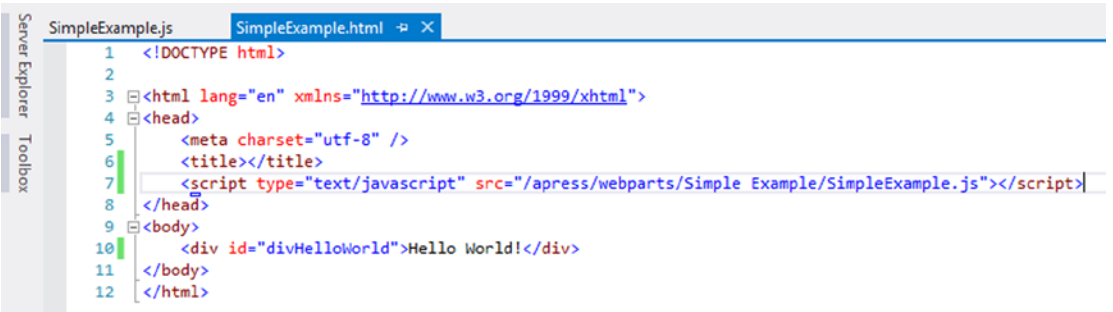

Figure 4-19. *The HTML file with the script reference added in*

Save the SimpleExample.html file with the new script reference. Next, you'll upload both files into the Simple Example folder that you created in the document library, as shown in Figure 4-20.

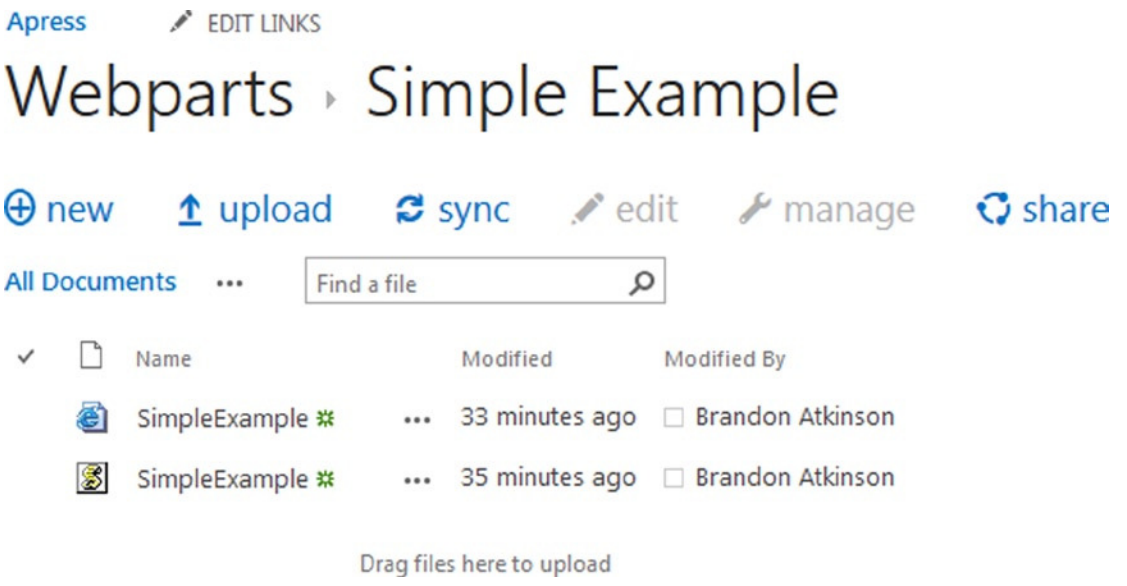

Figure 4-20. *HTML and JavaScript files uploaded to the Simple Example folder*

At this point, using the Etherson method, you've created and uploaded the two files required to re-create the example in Chapter 3. If you had a more complex solution, you can imagine this folder holding several JavaScript files, CSS files, and maybe multiple HTML files. It all depends on how complex or simple your needs are. Next, let's look at using the Content Editor Web Part to consume these files.

Content Editor Web Part

Now that we have the files needed to render some content, let's place a CEWP on the page and consume them. We'll use the same site and page from the Chapter 3 example. If you still have the CEWP and SEWPs on your page, go ahead and delete them so that the page is empty. Place the page into edit mode and then insert a new CEWP on the page. Save the page—we'll come back to this web part in a moment.

Navigate back to the Webparts document library and open the Simple Example folder. Right-click the `SimpleExample.html` file and choose Copy Link Location from the menu, as shown in Figure 4-21.

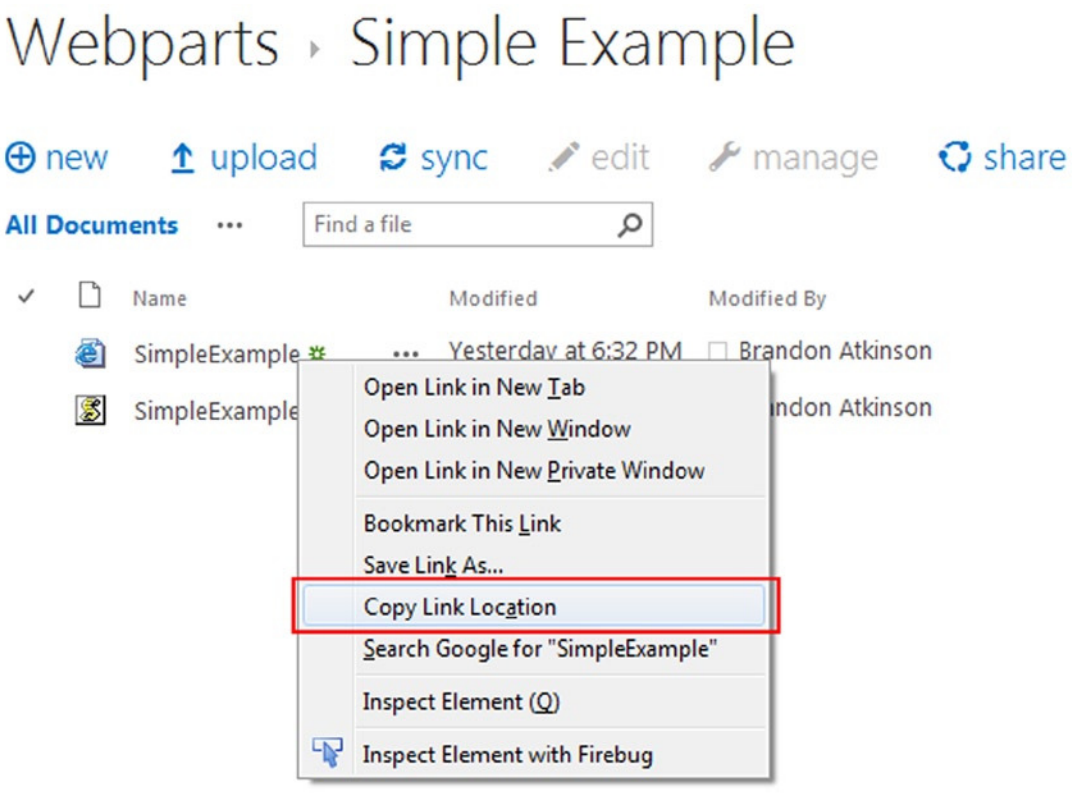

Figure 4-21. *Copying the location of the HTML file*

Depending on which browser you are using, this option may be titled differently. For instance, if using Internet Explorer, you would choose Copy Shortcut. No matter which browser you use, right-clicking exposes an option to copy the URL to the file. Once you have it copied, you can navigate back to the page with the CEWP.

Place the page into edit mode, and then place the CEWP into edit mode. Just like in Chapter 3, in the CEWP tool pane, the first property is title Content Link. Paste the URL that you copied into this text box, as shown in Figure 4-22.

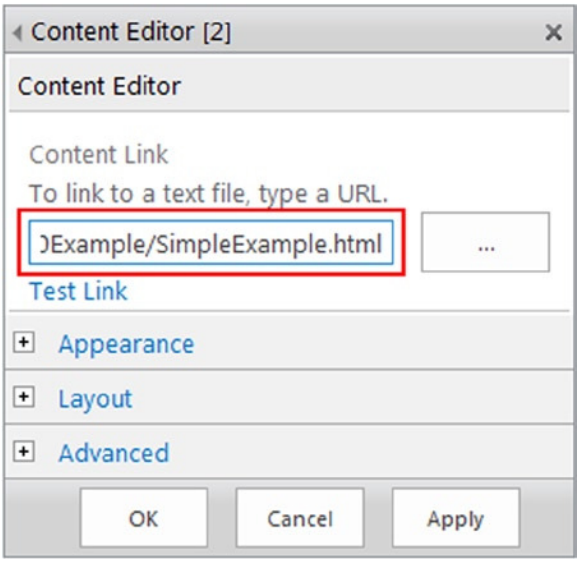

Figure 4-22. *The URL to the SimpleExample.html file in the CEWP Content Link property*

Unlike the example in Chapter 3, since this HTML contains a reference to the JavaScript file, all we need to do is put the file's URL into the Content Link property. When the HTML file loads, the JavaScript file will load as well. There is no need for a Script Editor Web Part in this example. Click OK and then save the page. When it reloads, the HTML file will render as shown in Figure 4-23.

Figure 4-23. *SimpleExample.html file rendered in the Content Editor Web Part*

At this point, you may be wondering why you would perform so much setup and configuration, since using a CEWP and SEWP can produce quicker results. The benefits of using the Etherson method become more evident with more real-world solutions. For instance, imagine that you need to develop a custom solution that aggregates tasks and documents the user is following, and places them into a list of links for the user to click on to view the item. It will be deployed on the home page for each department subsite in a site collection. Using the CEWP/SEWP solution, not only do you need to paste that code onto each page, if a change to the solution is needed later, you will need to update all instances of that code. That can be quite a challenge. With the Etherson method, you still add a CEWP in each site, but only reference one set of code files. If a change is needed, you only need to update code in one spot. This is much more manageable.

Another benefit of this method is that the CEWP can perform some error handling for you. If for any reason the HTML file is no longer available—perhaps it was moved or deleted—the CEWP can display a friendly message to users. To demonstrate this, let's remove the Content Link from the CEWP, and then click the Apply button. The content in the web part will be removed and you'll be left with a blank CEWP, as shown in Figure 4-24.

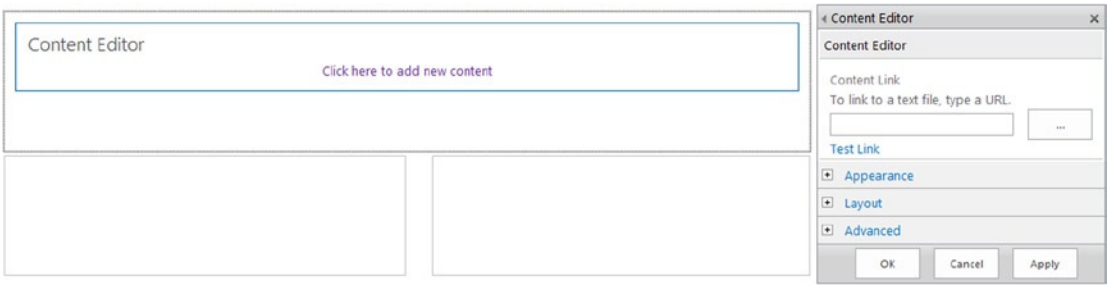

Figure 4-24. *Content Editor Web Part after the Content Link has been removed*

At this point, you've pretty much reverted the CEWP back to its initial state when it was added. Click the "Click here to add new content" link in the middle of the web part. You'll notice a cursor appear in the web part, in which you can type text directly. Here you'll add the text **Sorry, we've encountered an error**. Next, you'll add the link to the `SimpleExample.html` back into the Content Link property, as shown in Figure 4-25.

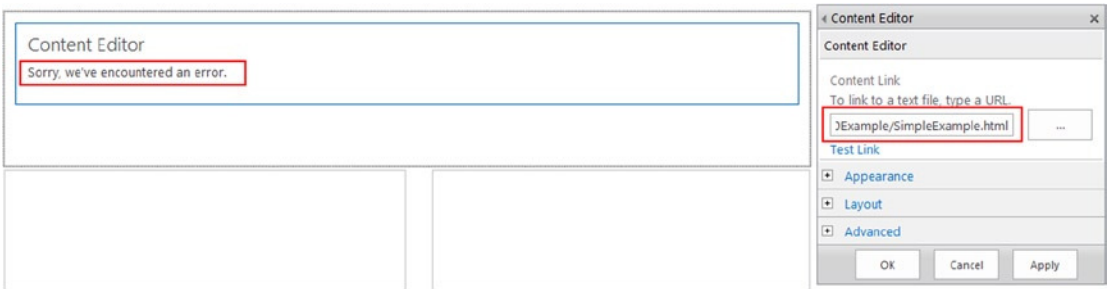

Figure 4-25. *Plain text and Content Link URL placed in the Content Editor Web Part*

Click OK and then save the page. You'll notice that the web part looks exactly the same way as it did when you only used the Content Link property. The plain-text message is not displayed and the content from the HTML page is rendered. The CEWP will always try to use any URL added to the Content Link property over displaying any plain text it may have. Now, let's delete the HTML and JavaScript files from the Simple Example folder, as shown in Figure 4-26.

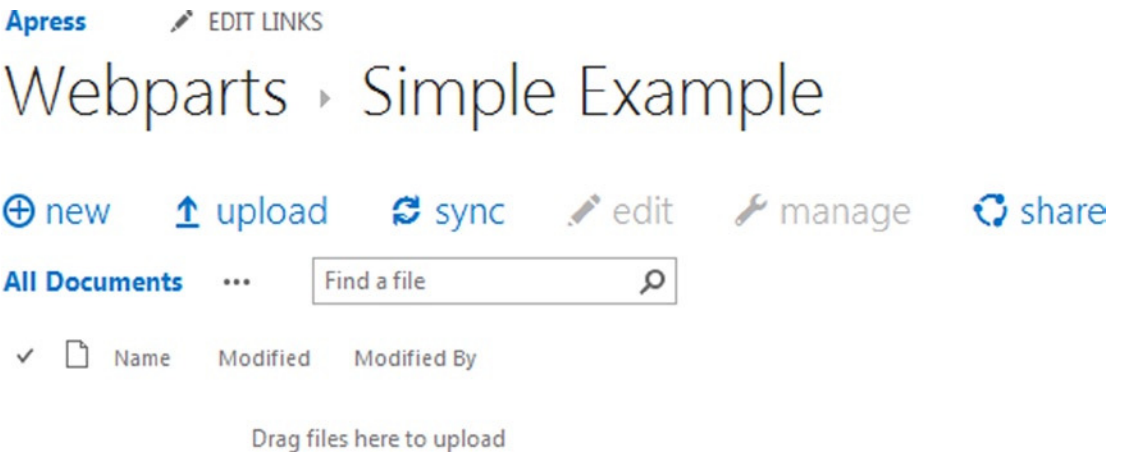

Figure 4-26. *HTML and JavaScript files deleted from the Simple Example folder*

Once the HTML file has been deleted, the CEWP will no longer be able to load it; it's gone. By providing the plain text in the web part, it will now display the error message that you loaded, as shown in Figure 4-27.

Figure 4-27. *The plain-text message displayed when the web part could not load the HTML file*

You will want to code for errors in your development efforts, but this fallback that is built into the CEWP is a great option to have in case something else goes wrong. You should always provide some content in this manner. There are many scenarios where the CEWP cannot access your HTML file, such as the following:

- The files get deleted from the folder (like we just did)
- A network error occurs
- You manually type the URL into the Content Link property and misspell the file name

It's definitely better to utilize this functionality just in case. Now let's update the code in the previous example to do more than just display a list name.

Working with List Data

The previous example introduced you to the Etherson method. At this point, you have an HTML file and a JavaScript file that pulls data about a list and displays it. Not too bad, but clearly this is not something you would do in a real-world solution. So let's expand on this code and actually pull some data from the list and display it to the user. If you haven't already, put some data into your Demo List, as shown in Figure 4-28.

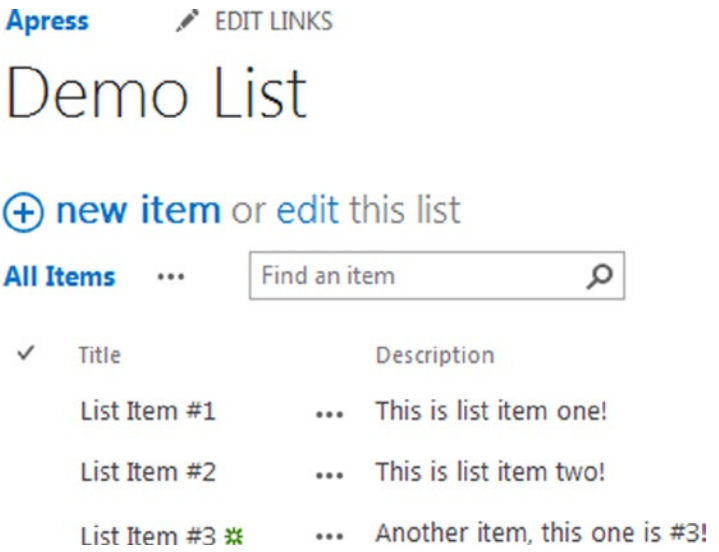

Figure 4-28. *The Demo List with a few items populated in it*

The items in this list don't really matter at this point; we just need some data to pull and display. Place as many items as you like; we're going to grab them all. Now that we have more data in the list, we'll update the `SimpleExample.js` file from earlier so that we request the list items rather than data about the list. Update your code with the following:

```
$(function () {
    $.ajax({
        url: "/apress/_api/web/lists/GetByTitle('Demo List')/items",
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose"
        },
    }).success(function (data) {
        var listItemInfo = '';

        $.each(data.d.results, function (key, value) {
            listItemInfo += "<strong>Title: </strong>" + value.Title + "
            <strong>Description: </strong>" + value.Description + "<br />";
        });

        $("#divHelloWorld").html(listItemInfo);
    });
});
```

It looks similar to the previous code, but with some key things added. First, the URL in the AJAX call has been updated to include `/items` at the end:

```
url: "/apress/_api/web/lists/GetByTitle('Demo List')/items"
```

We are still calling the same list, but we need to let the REST API know that we want the items from the list this time. By providing `/items` and no other information, we are telling the API to return all items in the list.

■ **Note** Unless you are confident that your list will not have much data in it, you should never request all the items in any call for data. You can provide ways to limit the data coming back by providing limitors or a CAML query that will return only the items you are looking for. We'll dig deeper into these options later in the book.

In the success function of our JavaScript, we will loop through all the data that is returned:

```
var listItemInfo = '';

        $.each(data.d.results, function (key, value) {
            listItemInfo += "<strong>Title: </strong>" + value.Title + "
            <strong>Description: </strong>" + value.Description + "<br />";
        });
```

We start with a new empty variable called `listItemInfo`. This variable is used to hold all the HTML that we build as we loop through the results of the API call. Next, we use a built-in jQuery function, `each`, to loop through the results. By using `$.each`, we're saying that for each item in the `data.d.results` object, perform the following function. You'll use this function a lot in your custom solutions, as looping through list items is very common.

Like the example where we simply displayed properties and data about the list, we see the familiar `data.d` object. However, this time we have a new property called `results`. When we queried the list for its data, all the information we needed was found under the `d` property; for instance, `d.Title`. When you request list items, all the data will be found in the `results` property.

Inside the `each` function, we set the `listItemInfo` variable to some HTML grabbing the current list item Title and Description. Using the `+=` syntax, we are setting the variable to the current value, plus the new values. This is a simple way of adding onto the HTML if there is any from the previous loop. After the loop is complete, we set the HTML to the Hello World DIV:

```
$("#divHelloWorld").html(listItemInfo);
```

Figure 4-29 shows the complete file in Visual Studio.

Figure 4-29. *The updated SimpleExample.js file in Visual Studio*

Save the file with the updated code. Before uploading it to the document library, let's quickly change the title of the CEWP so that this example is slightly more real world. Navigate back to the page where your web part is placed and put it into edit mode. Expand the Appearance accordion and update the Title property, as shown in Figure 4-30.

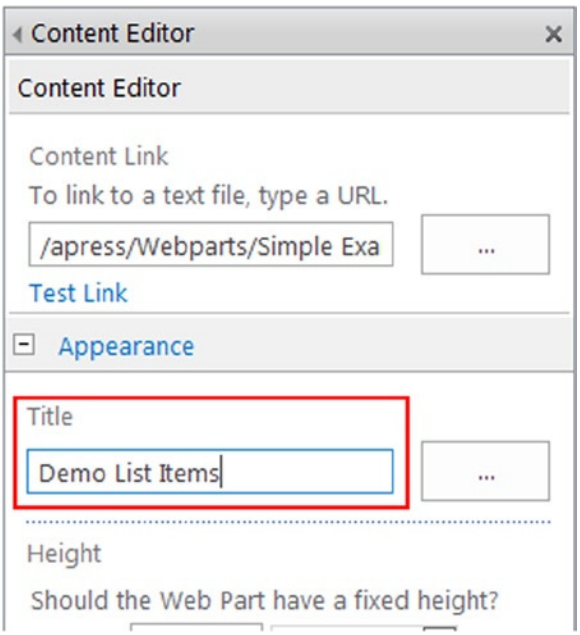

Figure 4-30. *Updating the title of the Content Editor Web Part*

Click OK, and then save the page. Upload your updated `SimpleExample.js` file to the Simple Example folder in the Webparts document library. You will be prompted with a dialog asking if you would like to replace the current document; click OK. The beauty of the Etherson method is still in effect; because you have turned on versioning in the document library, you can replace files without any stress. If you make a mistake, you can restore the previous version! Navigate back to your page and you will see the updated code grabbing list items and displaying them, as shown in Figure 4-31.

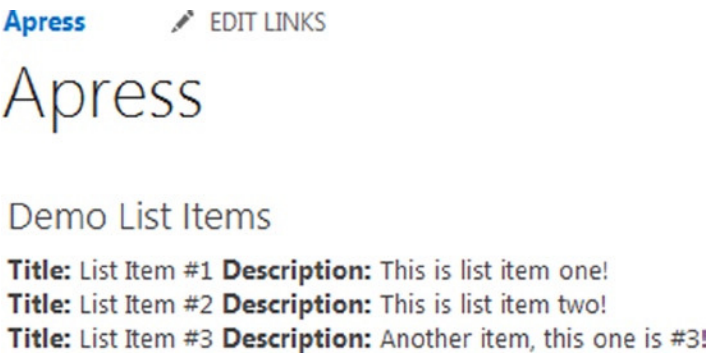

Figure 4-31. *Updated JavaScript fetching list items*

Combined with the updated title, this CEWP now resembles more of a real-world solution! Of course, all we've really done here is re-create an out-of-the-box List View Web Part, but it should have shown how easy it is to begin working with SharePoint data using HTML and JavaScript. Next, let's continue to look at some of the benefits of the Etherson method.

Rolling Changes Back with Versioning

Earlier in this chapter, when you created the document library to hold custom code, you enabled versioning. At some point during your development efforts you will make a mistake when writing your code, and it will get past testing. This happens to all of us and it is inevitable. Luckily, with versioning turned on you can quickly roll back changes if and when this happens. You may find that you need to roll back simply because users are not happy with the latest version of your code. No matter the reason, it's quick and easy.

When you uploaded the new version of the `SimpleExample.js` file, you were asked if you wanted to replace the existing version. By clicking OK, you created a new version of the file in the library. Since versioning is on, you can restore the previous version with a few clicks. Navigate to the Simple Example folder and click the menu for the `SimpleExample.js` file, as shown in Figure 4-32.

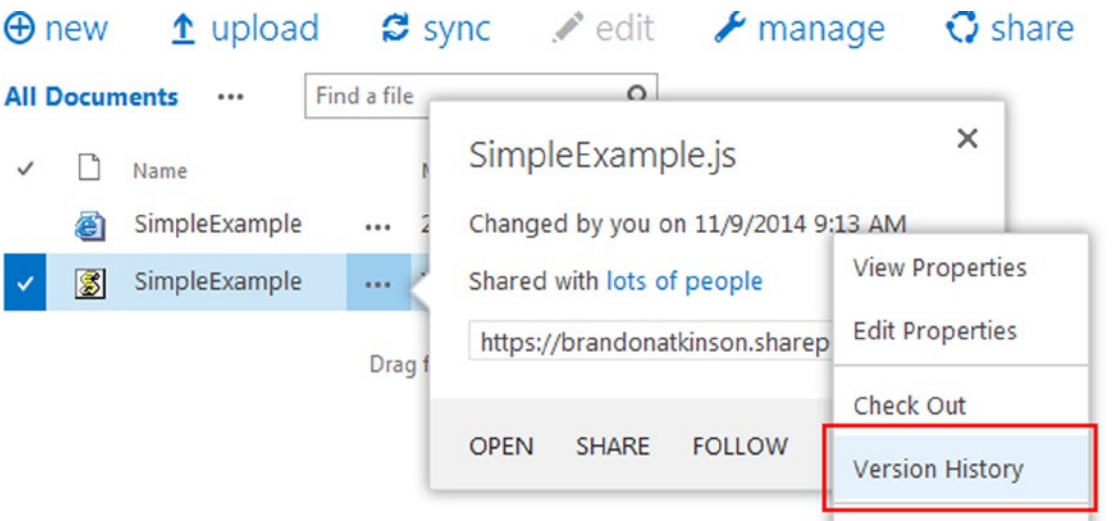

Figure 4-32. *Accessing the Version History on a file*

Once the menu is open, you'll need to access the secondary menu, where you can find the Version History options. Click this option and you'll be presented with a dialog showing all the versions of the file. The number of versions shown here depends on how you set up your library. For instance, if you limited the number of versions to five, you would only see five previous versions here. Choose the version you want; in this case, it would be the second version in the list. Versions are listed in ascending order from the upload date. Using the menu options, choose Restore from the menu, as shown in Figure 4-33.

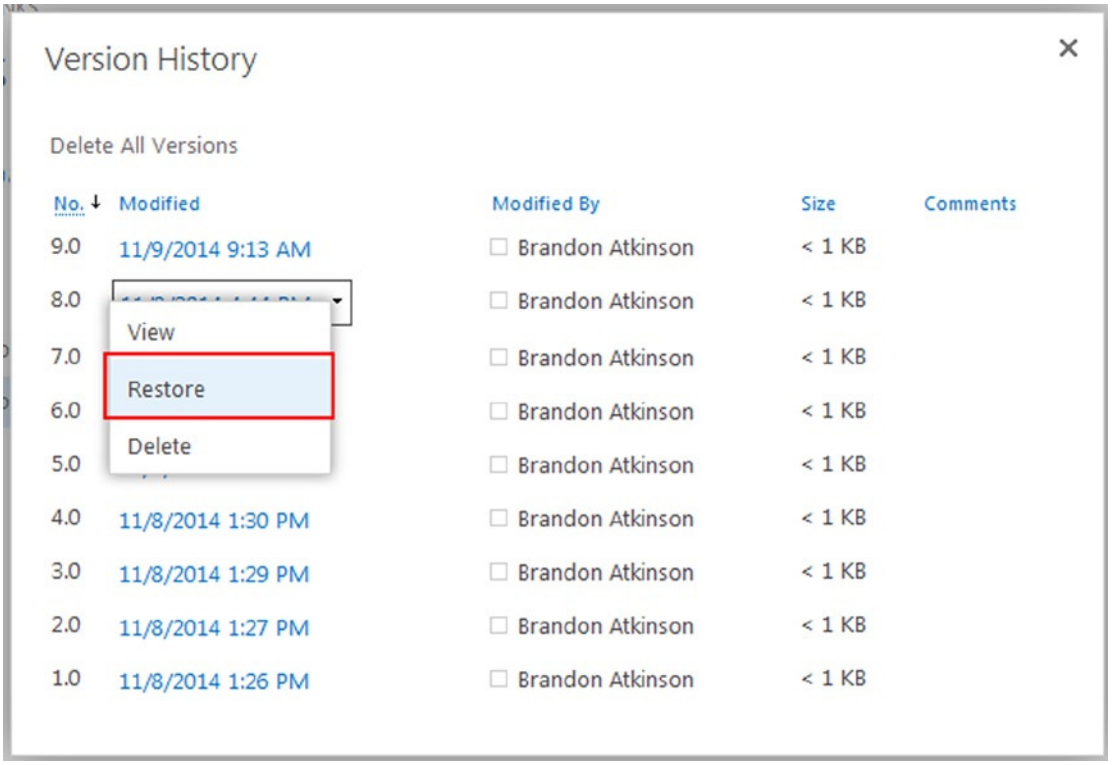

Figure 4-33. *Restoring a previous version on the file*

When you click the Restore option, you are presented with a confirmation dialog stating, "You are about to replace the current version with the selected version." Click OK to confirm the restoration, and the screen will refresh. A new version is added to the list. Navigate back to your page with the CEWP and refresh it. You may need to perform a hard refresh by pressing Ctrl+F5 on your keyboard; this will instruct the browser to load fresh copies of JavaScript files from the server. The result should be the previous script that loads data about the list, as shown in Figure 4-34.

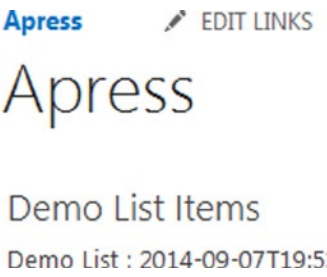

Figure 4-34. *Restored file being displayed in the Content Editor Web Part*

One thing to note here is that while the script content has been updated, the title of the web part remains the same. Since these are two separate pieces of the puzzle, restoring a script file will do nothing to the web part itself!

Debugging with Firebug (Developer Tools)

When you're writing code for anything, no matter the tools, language, or environment, you need to be able to debug it. In simple terms, stop the code from running and step through it line by line to see what data is coming in and how your code is processing it. Since the Etherson method utilizes HTML and JavaScript, you can easily debug code in your browser using Firebug or developer tools. As mentioned in Chapter 1, we'll be using Firebug a plug-in for Firefox. You can use the built-in developer tools in any browser. To launch these tools, just press F12 on your keyboard and the tools will launch at the bottom of your screen, as shown in Figure 4-35.

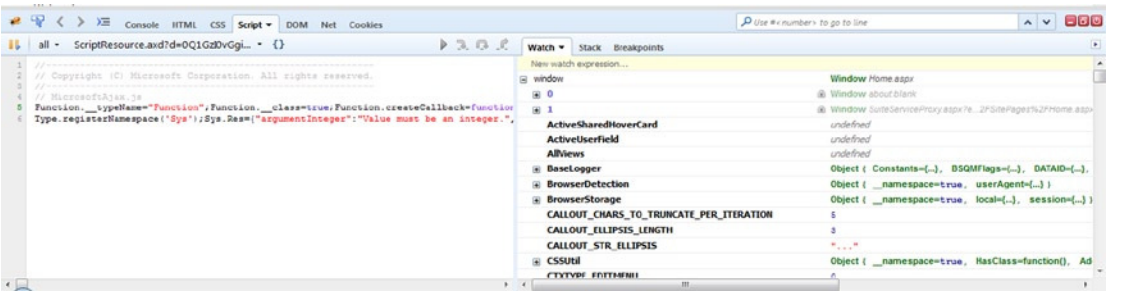

Figure 4-35. *Firebug console window in Firefox*

If it's not already open, select the Script tab in Firebug. If this is the first time you have launched Firebug, you may need to enable script debugging by pulling down the arrow on the tab and choosing Enabled in the options. This tab allows you to inspect all the scripts that are being loaded for the page. The left-hand pane shows the current script that you are looking at, and the right-hand pane shows any JavaScript objects that are loaded. These objects are found under the Watch tab in the right pane. Let's open our `SimpleExample.js` script by using the drop-down list right above the left pane, as shown in Figure 4-36.

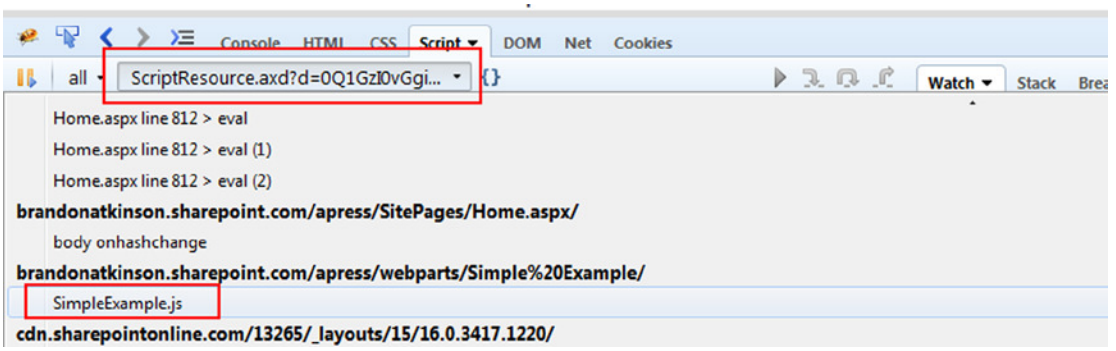

Figure 4-36. *Locating the SimpleExample.js file in Firebug*

This list contains all the JavaScript files that were loaded for the page. Scroll down until you find `SimpleExample.js`, and then click it. You'll notice that Firebug provides the path from where the script is loaded, which is very convenient should you need to locate it in your environment. Clicking the file loads it into the left pane, as shown in Figure 4-37.

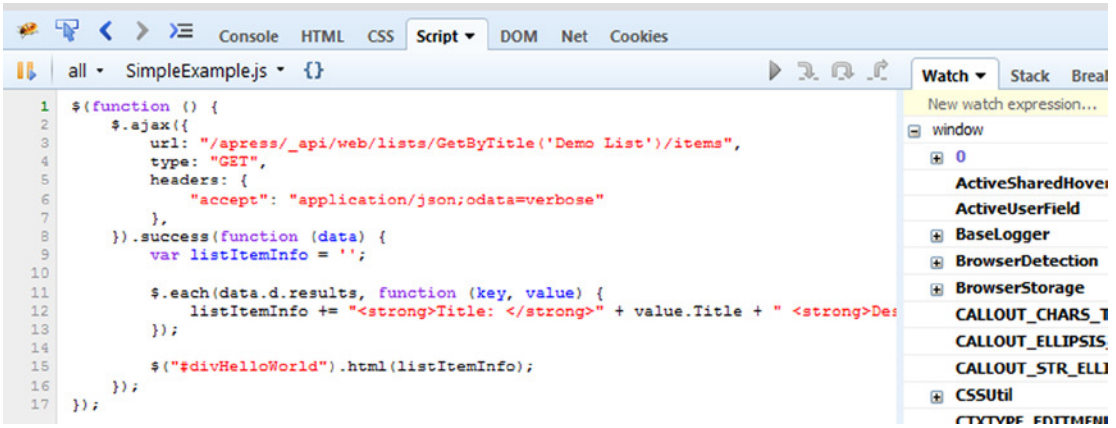

Figure 4-37. *SimpleExample.js file loaded in Firebug*

You'll notice that line numbers are provided on the left of the script file. These come in very handy if an error is thrown, as the browser will let you know which line an error occurred on. You can set a *breakpoint* by clicking one of the line numbers. A breakpoint is simply a location where your code will stop running so that you can inspect what is going on. You can set multiple breakpoints; in fact, you could set one on each line. When you set a breakpoint, a red dot appears next to the line number, as shown in Figure 4-38.

Figure 4-38. *Placing a breakpoint in the JavaScript file*

Once a breakpoint is set, you can reload the page, and when the script is loaded, it will stop running once the breakpoint is hit. You can see this when the red dot has a yellow arrow inside of it and the line is highlighted, as shown in Figure 4-39.

Figure 4-39. *Breakpoint hit while debugging SimpleExample.js in Firebug*

Now that the script has stopped, you can look in the right pane in the Watch tab and see the objects that your script is consuming. For instance, in this `success` function we declared a variable called `data`, which is where the results are located. The Watch tab now has a `data` object that we can inspect. Click the + icon next to `data`, and you can expand it to view its properties. Continue to click the + icons until you see the results, as shown in Figure 4-40.

Figure 4-40. *Data object that is sent back from the REST API*

We know the number of items that are coming back from our list and we can see these in the `results` object. They are labeled by their index, or position, in the list and indexing always starts at 0. You can further expand the items and view their properties as well. Click the + icon next to the item at the 0 index. You can see that SharePoint returns a lot of data about list items. This does not only apply to list items—just about anything you request data for in SharePoint comes with a lot of information tacked on. Scroll down to explore all the properties in the item and you'll find the Title and Description properties that we display via the script, as shown in Figure 4-41.

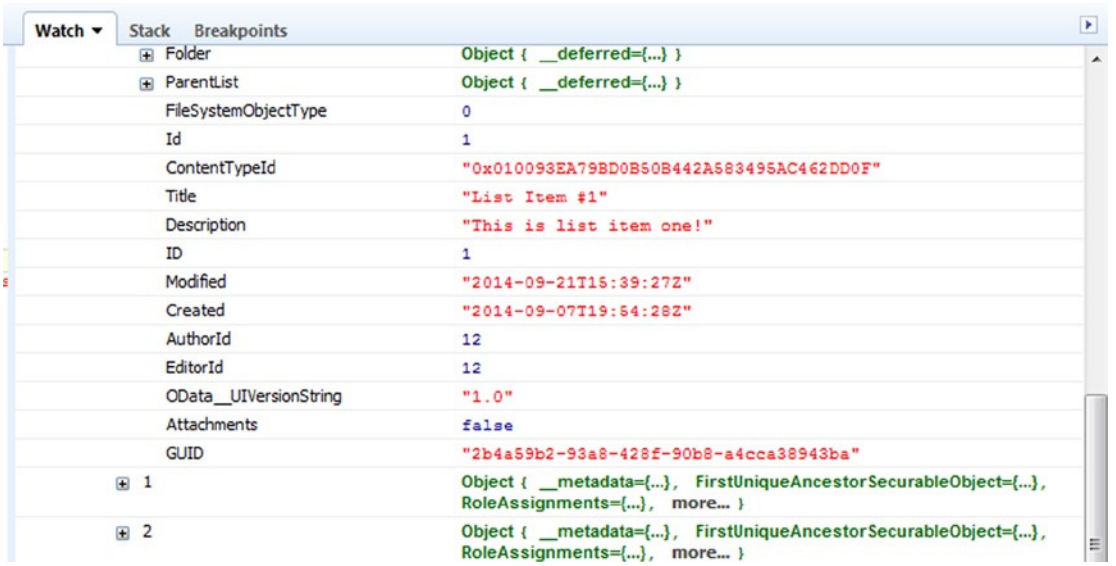

Figure 4-41. *Expanding the first list item to inspect its properties*

Debugging will prove extremely useful to you as you develop custom solutions, because you can inspect all the data coming back to ensure that it's exactly what you need. It's very handy as well, because sometimes data is not in the places that you thought. Since the script is stopped at this point, you have a couple of ways to start it back up again. Pressing the F10 key on your keyboard will make the script move forward one line at a time. This is very helpful because you can see what happens as each line is executed. Some blocks of code will execute together; for instance, all the code inside the `$.each` loop will act as a single line if you are using F10 to step through the code. You can easily place another breakpoint inside the loop to stop it there as well.

The other option is to use the arrow icon, as shown in Figure 4-42.

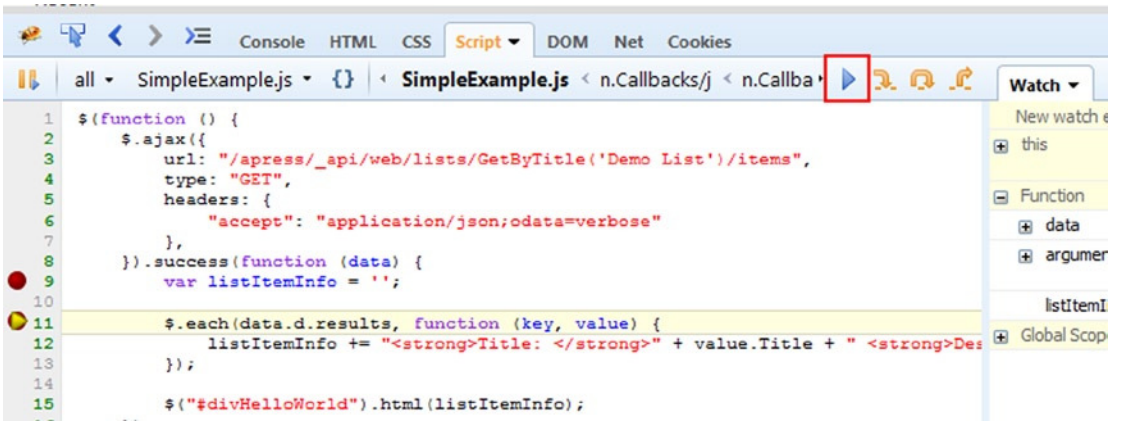

Figure 4-42. *Continue to run the script by clicking the Play icon*

This icon simply tells the debugger to continue on and resume processing the script. If no other breakpoints are present, it will finish processing the script completely. The buttons and file locations can vary between developer tools a great deal. You can see Chrome and Internet Explorer developer tools in Figures 4-43 and 4-44.

Figure 4-43. *Developer tools in Chrome*

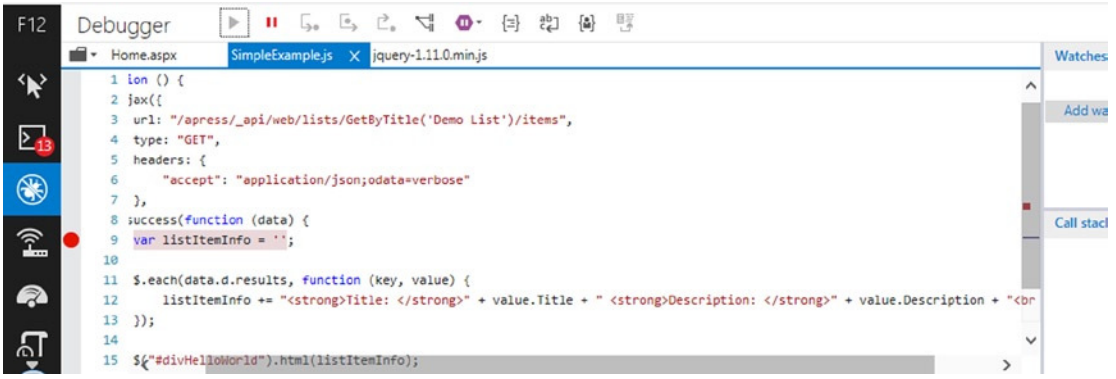

Figure 4-44. *Developer tools in Internet Explorer 11*

No matter which browser you use or how strange the developer tools may appear, rest assured that they all perform the same functions. You can inspect the HTML, review and change the CSS, and most importantly, debug your JavaScript. As you progress down the path of custom solutions using HTML and JavaScript, you'll find these tools as important as ever.

Summary

In this chapter, you were introduced to the Etherson method for building custom solutions using HTML and JavaScript. We covered the various components and you saw how to configure each and learned how they all worked together as a single unit. You built a couple of sample solutions and saw how you could easily restore code after uploading new files. As you work through this book, we'll continue to use this method for many examples, and you'll learn how to expand it using CSS.

In the next chapter, we'll explore using JSLink to style and enhance List View Web Parts. JSLink is a new and exciting way to bring more functionality to out-of-the-box SharePoint. It can also be combined with JavaScript to create very unique solutions to meet your business needs.

CHAPTER 5

Using JSLink to Style List Web Parts

In SharePoint, you have many different out-of-the-box web parts that you can place on your pages. These offer views into lists, searching, calendars, external data, and more. One of the most popular and widely used of these is the List View Web Part. Creating custom lists and then displaying their content to users via these web parts is quite common. These web parts allow you to quickly and easily expose data to users, and most users are very familiar with them.

The List View Web Part offers a number of different ways to customize the data that is shown. Some of these options include:

- Choosing the columns to display
- Sorting
- Filtering
- Grouping

This is a good amount of functionality that is provided for you out of the box. In a lot of scenarios, these options provide everything you need to customize the data for your needs; however, if you needed something really custom, these options just aren't enough. Imagine a scenario where the business requirement is to show an image based on the status of an item in the list. Perhaps there is a column with statuses like "On Time" and "Late". For each status, you need to display a smiley face or a frowny face based on this status. With the out-of-the-box List View Web Part, you would have had to use a rendering technology called XSLT. Now, you can utilize JSLink to perform this kind of data manipulation in the List View Web Part. In this chapter, we'll explore how JSLink works, as well as what you can accomplish with it.

JSLink and a Simple Example

Prior to SharePoint 2013, if you wanted to perform any type of styling on a List View Web Part you had to use XSLT. XSLT stands for *EXtensible Stylesheet Language Transformation* and it was used to style and render XML content. Not only is it hard to say, it was even more difficult to work with. You were hard-pressed to find a developer that not only knew how to effectively write XSLT, but who wanted to even touch it. It was difficult to write and hard to understand. A typical XSLT file would look something like the following:

```
<?xml version="1.0"?>
 <xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Description</th>
      </tr>
      <xsl:for-each select="catalog/cd">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="description"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Microsoft knew the challenges of working with XSLT and needed to provide developers with a better way of developing these types of solutions. JSLink was born from this need and it is a perfect alternative. The beauty of JSLink lies in the fact that it is basically JavaScript! Anyone who is comfortable writing JavaScript can quickly write JSLink code. In addition, JSLink can be combined with other JavaScript files to create very robust solutions.

JSLink is basically a property on the web part that provides a location(s) for JavaScript files. If this property is set with a valid URL to a file, when the web part is rendered, it picks up the file and uses it to render the contents to the browser. Since it's all JavaScript, all the rendering is handled by the browser, thus taking a load off the SharePoint server, which only has the job of returning data. This provides for easy customizations that do not impact the server performance.

Let's take a look at a quick example of using JSLink to style a web part. We'll expand on what we've built in the previous chapters by utilizing the Webparts document library and the Demo List. Navigate to the page where you placed the Content Editor Web Part and then delete it, so that you have a clean page. Insert a List View Web Part for the Demo List using the Ribbon, as shown in Figure 5-1.

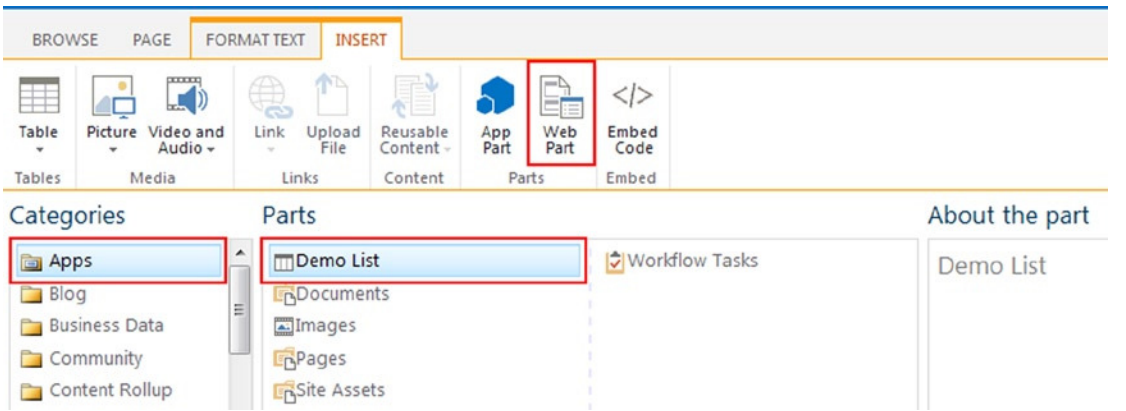

Figure 5-1. *Adding the List View Web Part for the Demo List*

If it's not already highlighted, click the Apps category. The Demo List should be at the top of the list, depending on how many lists you have in your site. Insert the web part onto the page and save it. The resulting view will simply display the list items, as shown in Figure 5-2.

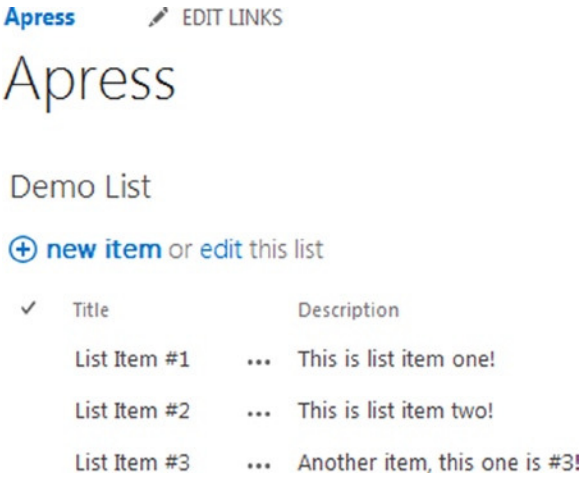

Figure 5-2. *The out-of-the-box List View Web Part for the Demo List*

We'll need to create a folder to hold our scripts for this example. Navigate back to the Webparts document library and create a new folder called **JSLink**, as shown in Figure 5-3. For this chapter, we'll place all of our scripts into this folder; however, in a production environment, you would want to follow the Etherson method and have a folder for each web part you are styling using JSLink.

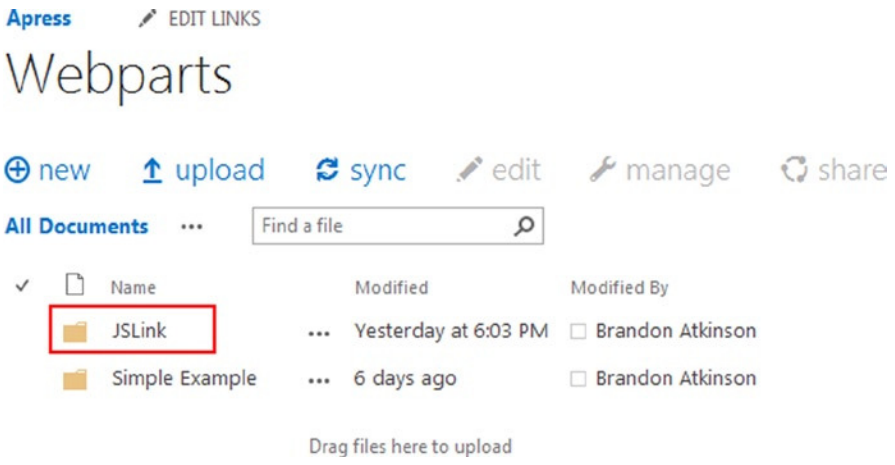

Figure 5-3. *Adding a new folder for JSLink in the Webparts document library*

Open Visual Studio and create a new file by going to File ➤ New ➤ File. Choose the Web category and then choose "JavaScript file" from the options. We'll start with a very basic script that will take each item in the web part and style it. Place the following code into the file:

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Item = overrideTemplate;
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();

function overrideTemplate(ctx) {
    return "<div style='font-size:18px;border:solid 1px Silver;margin-bottom:6px;
    padding:4px;width:200px;'>"
        + ctx.CurrentItem.Title
        + "</div>";
}
```

This script is about as streamlined as we can make a JSLink JavaScript file; however, it will provide an excellent introduction to how these scripts function. First, you'll notice that we declare a function that has no name. This function serves as the entry point for the script and where all the setup takes place. Next, we declare a new variable with the following line:

```
var overrideContext = {};
```

This is a simple JavaScript variable that we named `overrideContext` and set to an empty object using the {} syntax. What's important to note here is that there is no magic to what this is named. We could call it "`myObject`" or "`SomeCoolNewThing`", or anything for that matter. We're calling it `overrideContext` because we are providing an override of the default styling. Next, we declare another object called `Templates` with the following line:

```
overrideContext.Templates = {};
```

This object is used to hold a reference to all the different templates that we want to override the styling on. For instance, you can add `Item` to override styling on the entire list item, or `Field` to override styling on an individual column in the list. We do this with the following line:

```
overrideContext.Templates.Item = overrideTemplate;
```

This line indicates that we want to override the `Item` template with another function, called `overrideTemplate`. Again, there is nothing magical about the name of this function; it can be anything. Finally, we register the override with the following line:

```
SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
```

`SPClientTemplates.TemplateManager.RegisterTemplateOverrides` is a built-in SharePoint function that will register your code for rendering. You pass in the base object that you created at the beginning. Once the override is registered, the last piece is the function to actually perform the rendering:

```
function overrideTemplate(ctx) {
    return "<div style='font-size:18px;border:solid 1px Silver;margin-bottom:6px;
    padding:4px;width:200px;'>"
        + ctx.CurrentItem.Title
        + "</div>";
}
```

This function takes in a single argument named `ctx`. As with the other items in this script, the name means nothing. `ctx` is the list context and will be used to render the list items based on what is inside this function. We'll keep it simple, and for each item in the list we'll return a DIV element with the list item title in it. We can access the title using `ctx.CurrentItem.Title`. Since this function will fire for each item in the list, `CurrentItem` is just that—the current list item in the loop. The DIV has some inline CSS so that the custom rendering really stands out.

Save the file and name it **JSLinkDivExample.js**, and then upload it to the JSLink folder in the Webparts document library. The complete file in Visual Studio is shown in Figure 5-4.

Figure 5-4. *JSLinkDivExample.js file in Visual Studio*

Once the file is uploaded, we need to grab the URL of the file. In the previous chapter, we saw how you could right-click the file and choose Copy Link Location from the context menu. You can also obtain the URL by opening the file's menu and copying it from the text menu in the pop-out, as shown in Figure 5-5.

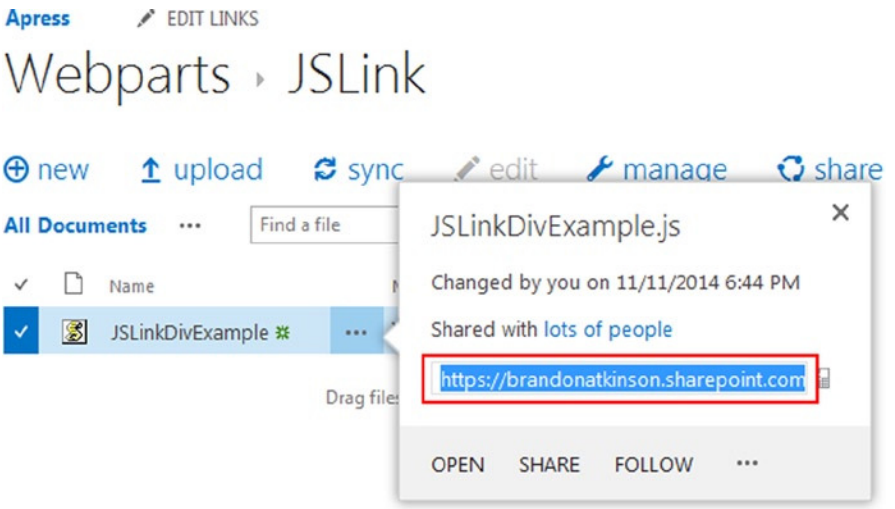

Figure 5-5. *Copying the URL of the JavaScript file from the menu*

Navigate back to the page where the List View Web Part was placed. Place the page into edit mode, and then place the List View Web Part into Edit mode as well. In the tool pane of the web part, scroll to the bottom and expand the Miscellaneous accordion panel. At the bottom of this panel, you will find the JSLink property, as shown in Figure 5-6.

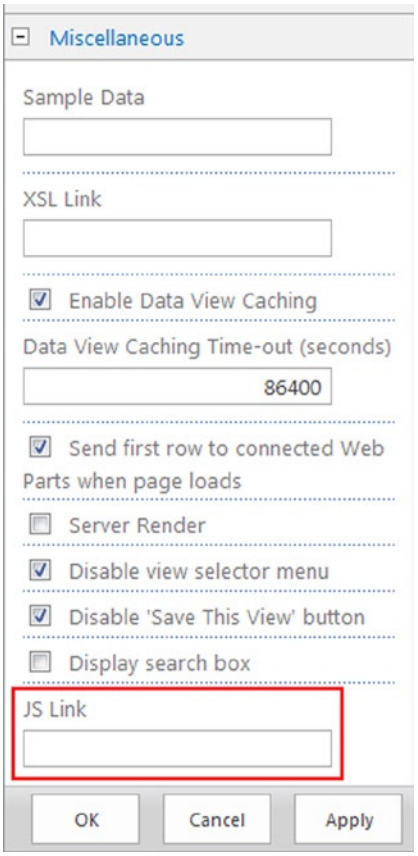

Figure 5-6. *JSLink property in the web part tool pane*

When using a URL with the Content Editor Web Part, you can simply paste in the URL and SharePoint will automatically convert it to a relative path URL. Meaning, SharePoint will strip off the domain from the URL and leave only the parts it needs to locate the file. For instance, the `JSLinkDivExample.js` file path in my environment looks like the following:

```
https://brandonatkinson.sharepoint.com/apress/Webparts/JSLink/JSLinkDivExample.js
```

If you paste this into the JSLink property and save the web part, SharePoint will convert the URL to the following:

```
/apress/Webparts/JSLink/JSLinkDivExample.js
```

Normally, SharePoint prefers its URLs, due to the fact that SharePoint can be configured to serve up content from multiple domains. By utilizing relative path URLs (those without a domain), SharePoint can easily figure out where content is located, no matter which domain it's served from. This works great with the Content Editor Web Part, but it will fail when used with JSLink.

Luckily, SharePoint provides an easy way to remedy this by using URL tokens. Tokens provide a way to pass in a placeholder for the parts of the URL that SharePoint wants to remove. The following list shows some of the more common tokens:

- `~site` - The URL of the current site
- `~sitecollection` - The URL of the site collection of the current site
- `~layouts` - The URL of the `_layouts/15` folder

Using these tokens, the URL can be updated to the following:

```
~sitecollection/apress/Webparts/JSLink/JSLinkDivExample.js
```

or

```
~site/Webparts/JSLink/JSLinkDivExample.js
```

Either of these new URLs will work because they both point to the same location. Since our Webparts document library is located in the same site as the page where our web part is placed, we can use the shorter of the two. Type this URL into the JSLink property, as shown in Figure 5-7.

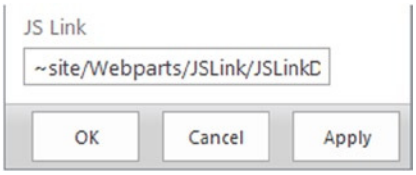

Figure 5-7. *URL to the JavaScript file using the URL token*

Click OK and save the page. When the page reloads, the `JSLinkDivExample.js` file will be loaded and the list items will be rendered based on the `overrideTemplate` function inside it. The results are shown in Figure 5-8.

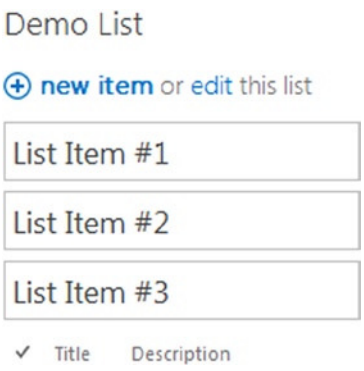

Figure 5-8. *JSLink rendering of the Demo List items*

As you can see, each list item was rendered inside a DIV with a border and larger text, as per the inline CSS in the JavaScript file. In our function, the only piece of data that we rendered was the `Title` property, which is what is shown inside the DIV. This simple example should begin to illustrate the power that JSLink provides for styling the List View Web Part, as well as a way to begin building robust custom solutions.

Working with the Item Template

The prior example was a quick sample of what can be done with JSLink. However, it's far from a realistic solution and it has some issues. For instance, the sorting links for the Title and Description columns are still present and shown under the list items. Clearly, that is not the way you would want this to appear to end users. In this section, we'll dig a bit deeper into the Item template and how you can work with it.

When using the Item template, you will usually style all aspects of the List View Web Part. By this I mean that it usually won't look like a List View anymore. JSLink provides a lot of ways to help you achieve this goal. Let's start by providing a template for the header and footer of the List View using the following code:

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Item = overrideTemplate;
    overrideContext.Templates.Header = "<h2>Custom Header</h2>";
    overrideContext.Templates.Footer = "<h3>Custom Footer</h3>";
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();
```

This sample is exactly the same as before, only we added two additional lines:

```
overrideContext.Templates.Header = "<h2>Custom Header</h2>";
overrideContext.Templates.Footer = "<h3>Custom Footer</h3>";
```

By using these lines we are telling SharePoint that we want to override the default header and footer using the supplied HTML. The HTML is very simple—it just returns some text wrapped inside an H2 and H3 tag. However, the result quickly shows the power of these overrides, as seen in Figure 5-9.

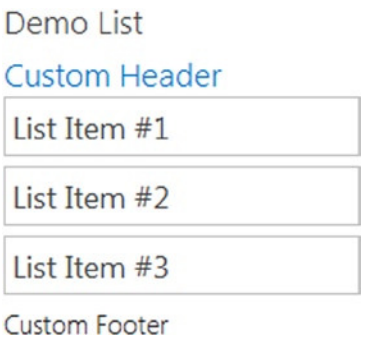

Figure 5-9. *Custom header and footer being rendered on the List View Web Part*

You can further extend this by providing a function to return the header and footer, rather than having HTML declared in the same line, like the following:

```
overrideContext.Templates.Header = overrideHeader;
overrideContext.Templates.Footer = overrideFooter;
```

The resulting functions would look like the following:

```
function overrideHeader() {
    return "<h2>Custom Header</h2>";
}

function overrideFooter() {
    return "<h3>Custom Footer</h3>";
}
```

Placing this code inside its own function opens the door for increased functionality. You can imagine a scenario where you need to make a web service call to retrieve the data that you want to display in the header. This allows for very robust code to fire off and perform any task that is available to you. You can begin to create very rich user experiences in this manner. The complete code in Visual Studio is shown in Figure 5-10.

JSLinkItemTemplate.js

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Item = overrideTemplate;
    overrideContext.Templates.Header = overrideHeader;
    overrideContext.Templates.Footer = overrideFooter;
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();

function overrideTemplate(ctx) {
    return "<div style='font-size:18px;border:solid 1px Silver;margin-bottom:6px;padding:4px;width:200px;'>"
        + ctx.CurrentItem.Title
        + "</div>";
}

function overrideHeader() {
    return "<h2>Custom Header</h2>";
}

function overrideFooter() {
    return "<h3>Custom Footer</h3>";
}
```

Figure 5-10. *Providing header and footer overrides via JavaScript functions*

Save this file under a new name using File ➤ Save As. Call it **JSLinkItemTemplate.js**. Upload it to the JSLink folder in the Webparts document library. Navigate back to the List View Web Part, place it in Edit mode, and update the JSLink property to the following:

```
~site/Webparts/JSLink/JSLinkItemTemplate.js
```

The web part should be using the new JSLink file. This example is still far from a real-world experience. Let's update the Demo List with some data that could represent a task list or some other type of list that would have a status. Navigate to the Demo List and access the List Settings from the Ribbon. On the List Settings page, create a new Choice column called **Status**. Provide the following choices (see Figure 5-11):

- On-time
- Delayed
- Late

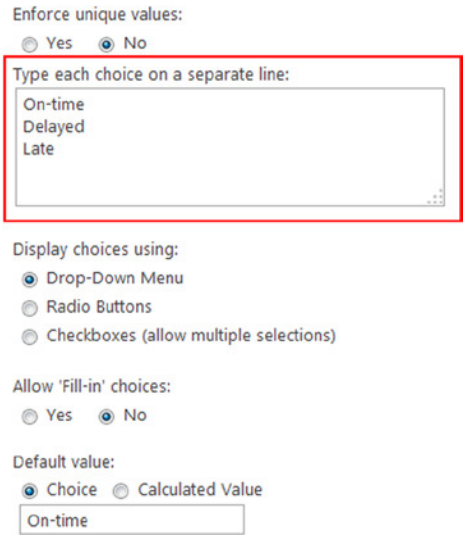

Figure 5-11. *Adding a Choice field to the Demo List called Status*

We'll use this new status column to display an image based on the status that is chosen. However, before we do that, let's populate the Demo List with some data that would make more sense for a business. Figure 5-12 shows the list with some more appropriate data to work with.

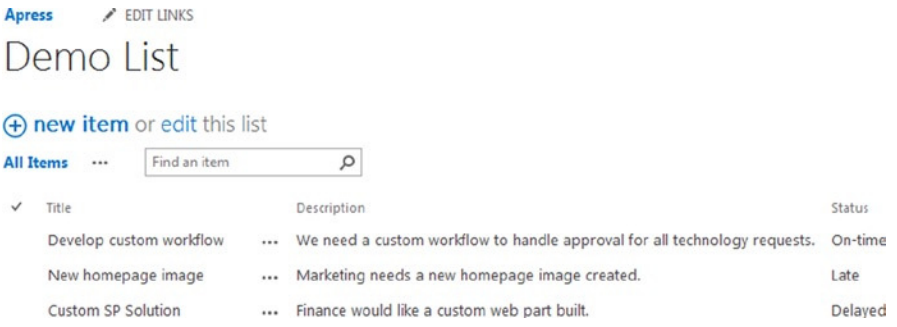

Figure 5-12. *Demo List with more real-world data populated*

To make this example a little more realistic, we'll update our script so that the following requirements will be met:

- When the web part is rendered, each item will display only the title.
- A custom header and footer will be displayed with instructions for the user.
- When the user clicks the item title, the status and description will be displayed underneath.
- The status will be shown as an image.

Let's begin with the header and footer, since these functions already exist and we only need to update what is returned. The custom code can be updated with the following:

```
function overrideHeader() {
    return "<h3>These items are tasks from the Demo List:</h3>";
}

function overrideFooter() {
    return "<h4>Click an item to view it's details.</h4>";
}
```

We actually haven't changed the code much at all. We've changed the H2/H3 to an H3/H4, respectively, and updated the text that gets returned. We'll use the customer header as our title in the web part, and the custom footer now provides instructions to the user. Next, we'll update the `overrideTemplate` function; this is where the major changes will be implemented. Take the following code:

```
function overrideTemplate(ctx) {
    var status = ctx.CurrentItem.Status;
    var image = "";

    if (status == "Delayed")
        image = "/apress/Webparts/JSLink/Status-Delayed.png";
    if (status == "On-time")
        image = "/apress/Webparts/JSLink/Status-Ontime.png";
    if (status == "Late")
        image = "/apress/Webparts/JSLink/Status-Late.png";

    return "<div style='font-size:18px;margin-bottom:6px;padding:4px;cursor:pointer;'
    class='list-item-div'>"
        + ctx.CurrentItem.Title
        + "<div style='font-size:14px;display:none;'>"
        + "<span style='display:inline-block;'>" + ctx.CurrentItem.Status + "<br />
          <img src='" + image + "' /></span>"
        + "<span style='display:inline-block;vertical-align:top;'>" +
          ctx.CurrentItem.Description + "</span>"
        + "</div>"
        + "</div>";
}
```

The overrideTemplate function has been updated quite a bit. Although it may look like a lot more functionality at first glance, there isn't a whole lot more going on here. There is additional logic as well as more HTML. Let's break down each piece individually to better understand what is going on.

```
var status = ctx.CurrentItem.Status;
var image = "";
```

These two lines simply declare a couple of variables to be used in the rest of the code. First, we declare a variable called status and assign it the value of the current list item's Status property. This just simplifies the code and makes it more readable. Next, we declare an empty string called image. This will hold the URL of an image that we want to show based on the status of the item.

```
if (status == "Delayed")
    image = "/apress/Webparts/JSLink/Status-Delayed.png";
if (status == "On-time")
    image = "/apress/Webparts/JSLink/Status-Ontime.png";
if (status == "Late")
    image = "/apress/Webparts/JSLink/Status-Late.png";
```

This section is a simple grouping of if statements. We check the value of the status variable, checking it against the known values of the list column. Once a status match has been found, we set the image variable to the appropriate URL for the image we want to show.

■ **Note** The images used in this example can be downloaded from the Apress site. Visit the Apress site and search for this book's title to locate the image files and source code. For this example, I've uploaded the images into the same folder with the JavaScript files for simplicity.

```
return "<div style='font-size:18px;margin-bottom:6px;padding:4px;cursor:pointer;'
class='list-item-div'>"
        + ctx.CurrentItem.Title
        + "<div style='font-size:14px;border-top:solid 1px Silver;display:none;'>"
        + "<span style='display:inline-block;'>" + ctx.CurrentItem.Status + "<br />
          <img src='" + image + "' /></span>"
        + "<span style='display:inline-block;vertical-align:top;padding-left:16px;'>"
        + "<strong>Description</strong><br />"
        + ctx.CurrentItem.Description + "</span>"
        + "</div>"
        + "</div>"
```

Finally, we've updated the HTML that gets returned so that we can dramatically change the results. First, we create a DIV element with some additional inline CSS, as well as a CSS class list-item-div. This CSS class will be used later when we target the element using jQuery. Next, we display the current list item title, and then create a child DIV element to hold the status and description. This DIV also has some inline CSS that includes display:none;. This will keep the DIV hidden until we want to view it. We continue to create HTML to hold the item status, the status image, and the description, finally closing off all the DIV elements.

At this point, the page would render and you would see the Title for each list item on the page, but the status and description information would be hidden. We need to provide a way to view this information easily. The following function will handle this:

```
function listItemClick() {
    $(".list-item-div").on("click", function () {
        var childDiv = $(this).children()[0];
        $(childDiv).toggle("slow");
    });
}
```

When we created each child DIV in the previous function, we hid those using CSS and applied a CSS class. The class is there only to provide a means to target it via jQuery. We use the `on` function that is built into jQuery to attach a click event to each DIV. So by using `$(".list-item-div").on("click", function ()...`, we are telling jQuery to find all the elements with the `list-item-div` CSS class and attach an `on click` event to each one. The function provided is fired when the element is clicked. Inside this function, we ask jQuery to give us the first child element in the DIV. `$(this).children()[0]` tells jQuery that for the item that was just clicked ("this"), give us the first child object in its "children" collection. Because we crafted our HTML to have a parent DIV with one child DIV, we know that this will return the DIV element that holds the status and description information.

Next, we use a jQuery function called `toggle` to show the element. The `toggle` will actually toggle the visibility of the element. So if the element is currently hidden, `toggle` will show it—and vice versa. By using this function we can easily supply open and close functionality with a single line of code.

Once we have the function used to attach a click event to the DIVs, we need a way to fire it. Simply writing this function and having it in the script will do nothing until it's called. Normally, we would include a jQuery `Document.ready` call to ensure that the events are wired up properly using a `$(function()...` call. In JSLink, however, this does not work and it will produce an error. The reason is that JSLink onload functions will fire before jQuery is loaded. This means that the reliable `Document.ready` function from jQuery is useless in JSLink. However, SharePoint provides its own built-in function that does the same thing:

```
_spBodyOnLoadFunctions.push(listItemClick);
```

The `_spBodyOnLoadFunctions` allows you to register your own JavaScript functions to fire when the page is loaded. It's very similar to jQuery's `Document.ready`, but it plays nicely with JSLink, so it's the method we need to use when wiring up jQuery events. The full script in Visual Studio is shown in Figure 5-13.

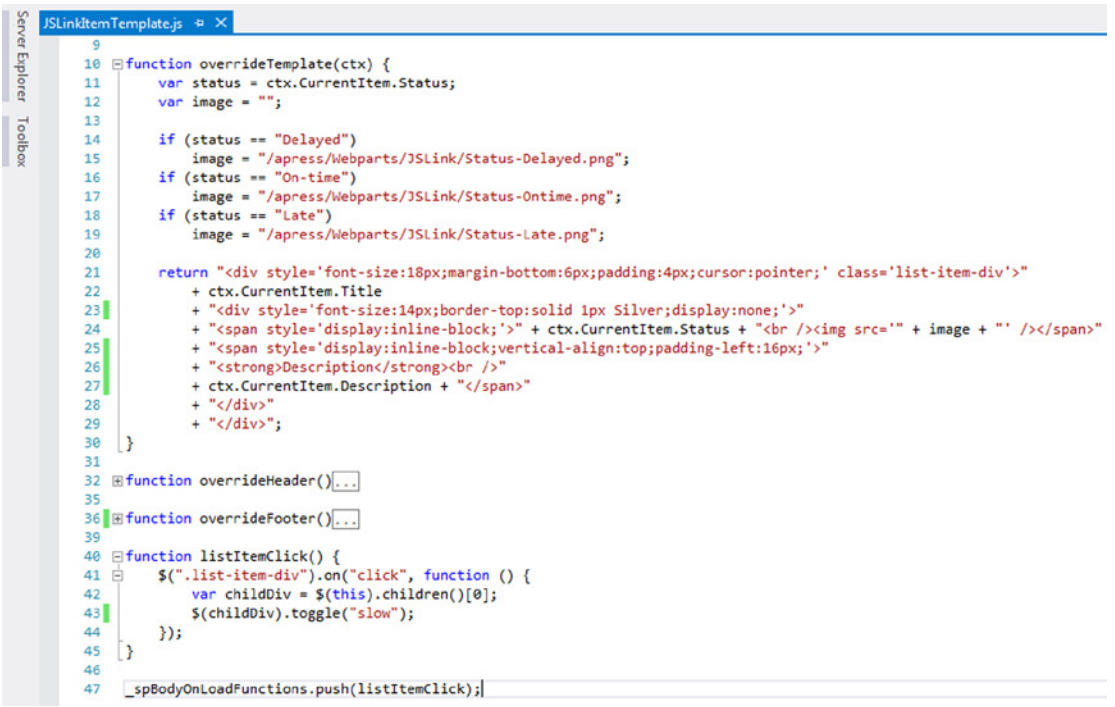

Figure 5-13. *The updated JSLinkItemTemplate.js script in Visual Studio. The custom header and footer functions have been collapsed for readability*

Save the updated file and upload it to the JSLink folder in the Webparts library. Before this script will work properly, you need to update some properties on the List View Web Part. When you first place the List View Web Part on the page, the data that gets returned is the bare minimum for the list type. For instance, the Demo List is a custom list, so only the Title and Description are returned by default. This is the same for a Task list; for instance, only the columns that are present at the time of creation get returned when the List View Web Part is added to the page. Of course, you can update this easily. Navigate to the page where the List View Web Part is placed, and place it and the web part into Edit mode. In the tool pane of the web part, the first property is called `Selected` View. Using the drop-down choose the All Items view, as shown in Figure 5-14.

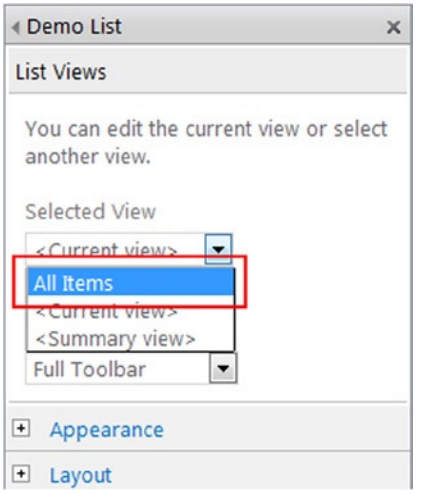

Figure 5-14. *Changing the Selected View on the List View Web Part*

Changing this view ensures that the new `Status` property will be returned and available to our JSLink code. Next, since we are using the custom header in our code to render a title, we'll remove the Chrome from the web part. Expand the Appearance accordion in the tool pane. At the very bottom of this section, you'll find the Chrome Type property. From the drop-down choose None, as shown in Figure 5-15.

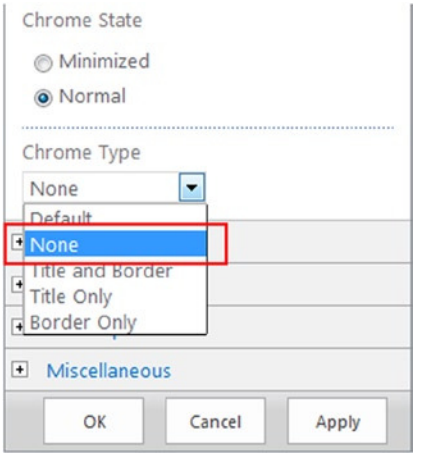

Figure 5-15. *Changing the Chrome Type on the List View Web Part*

Click OK and then save the page. The List View Web Part should now display properly using the JSLink file for rendering. The results are shown in Figure 5-16. The custom header and footer are displayed, which provide a web part title as well as instructions to the user on how to use the web part. The Chrome is no longer displayed and each item only displays the item title. Hover over the item and the mouse cursor will change to a pointer, indicating that it can be clicked. Clicking the item causes the jQuery `toggle` function to fire, opening and closing the item.

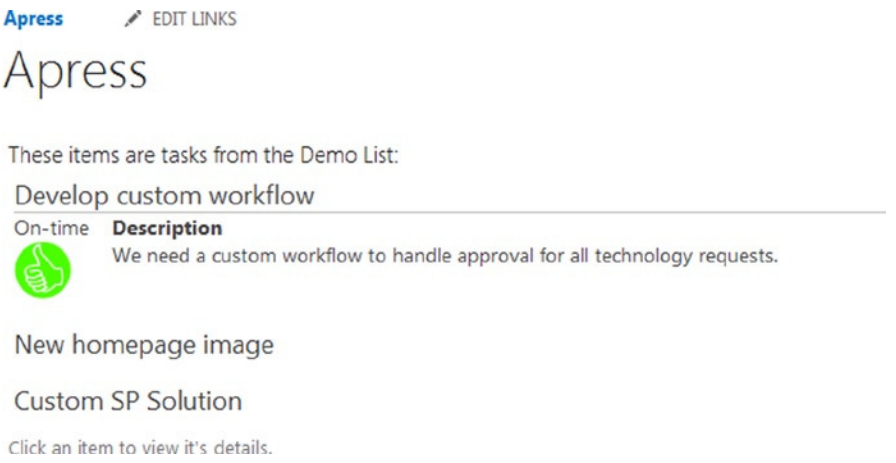

Figure 5-16. *The Demo List items displayed using the JSLink file*

As you can see, this is a drastic change from the out-of-the-box List View Web Part you saw earlier. In fact, it looks and acts like what you would expect—a custom web part built to the needs of the business! In the past, this type of solution would have required server-side coding and a custom web part that would have been deployed to the SharePoint server. Now all this is possible in the browser with JavaScript! Next, let's look at how we can manipulate individual columns rather than all the data at once.

Working with the Field Template

The Field template in JSLink allows you to target specific fields or columns in the list for custom rendering. You can use the Field template when your custom rendering needs are not as complex. You can certainly end up with very complex custom rendering using this template. However, unlike the Item template, where you usually end up with something that looks nothing like a List View Web Part, the Item template is much more targeted. The end result is something that looks like a List View Web Part on steroids.

The code looks very similar to the Item template code. We'll continue to work with the Status column. For this solution, we'll inspect the Status column and return the corresponding image, but this time it will be in the List View Web Part as column data. Create a new file in Visual Studio and begin with the following code:

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Fields =
    {
        'Status': { 'View': overrideTemplate }
    };
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();
```

This code looks very similar to the Item template code, with a couple major changes. First, we are no longer using a custom header and footer. The reason is that this template will not override the standard functions of the web part. So we still want the regular column headers, sorting, and so forth. Next, we specify that this is a Fields template with the following lines:

```
overrideContext.Templates.Fields =
{
    'Status': { 'View': overrideTemplate }
};
```

`'Status'` is telling JSLink that we are targeting the Status field on the list. `'View'` specifies which function to fire when the Status column is rendered. So we need to declare the function called `overrideTemplate` to render the column by using the following code:

```
function overrideTemplate(ctx) {
    var status = ctx.CurrentItem.Status;
    var image = "";

    if (status == "Delayed")
        image = "/apress/Webparts/JSLink/Status-Delayed.png";
    if (status == "On-time")
        image = "/apress/Webparts/JSLink/Status-Ontime.png";
    if (status == "Late")
        image = "/apress/Webparts/JSLink/Status-Late.png";

    return "<img class='status-image' src='" + image + "' data-val='" + status + "' />";
}
```

This function is nearly identical to the previous example. We still declare a couple of variables to hold the current item's Status and the image URL. By using the `if` statements, we are setting the image URL based on the status, just as before. What's different is the return, which is an image HTML element. We set the SRC attribute to the image URL we determined from the `if` statements. We also set a CSS class of `status-image` and an attribute called `data-val` that holds the text value of the status. These last two attributes will be used with a jQuery click event. This function simply returns an image for the Status rather than the text of the status.

Next, with the following code we'll set up the function to attach the click event to each image:

```
function listItemClick() {
    $(".status-image").on("click", function () {
        var status = $(this).attr("data-val");
        alert(status);
    });
}
```

This function also looks very familiar to the previous example. We set up a jQuery click event using the CSS class "`status-image`". Next, we grab the text of the status using `$(this).attr("data-val")`. jQuery's "`attr`" function returns the value of any attribute on an HTML element, so we can easily access this value. Then, we fire a standard alert with the status text in it. Finally, we wire up the function using the `_spBodyOnLoadFunctions.push` function:

```
_spBodyOnLoadFunctions.push(listItemClick);
```

This example is much smaller than the Item template example. This is because the override is targeting only a single field rather than all the columns, as well as the header and footer. The complete code sample in Visual Studio is shown in Figure 5-17.

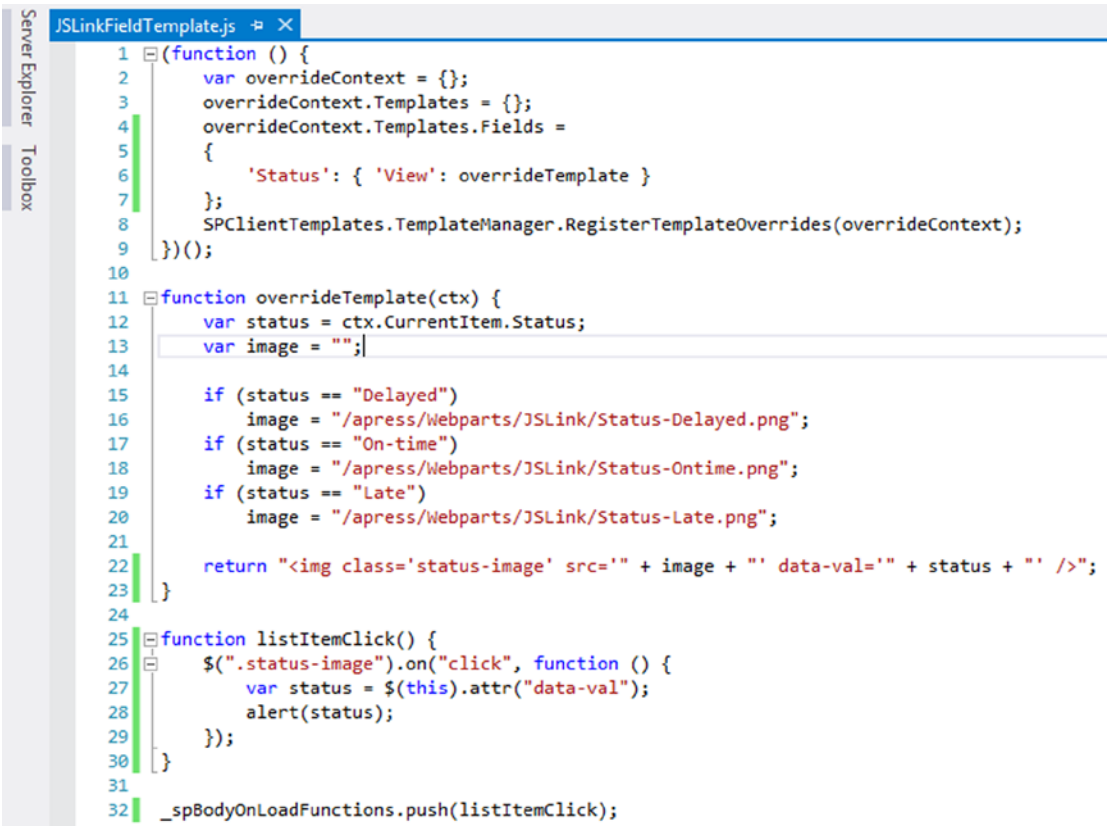

Figure 5-17. *JSLinkFieldTemplate.js file in Visual Studio*

Save this file and name it `JSLinkFieldTemplate.js`. Upload it to the JSLink folder in the Webparts document library where the other JSLink files are located. Navigate back to the List View Web Part, place it in Edit mode, and update the JSLink property to the following:

```
~site/Webparts/JSLink/JSLinkFieldTemplate.js
```

As with the Item template, you may need to set the appropriate View using the Selected View property. You can also set the Chrome type back to Default so that the web part title is displayed again. The new JSLink file rendered in the List View Web Part is shown in Figure 5-18.

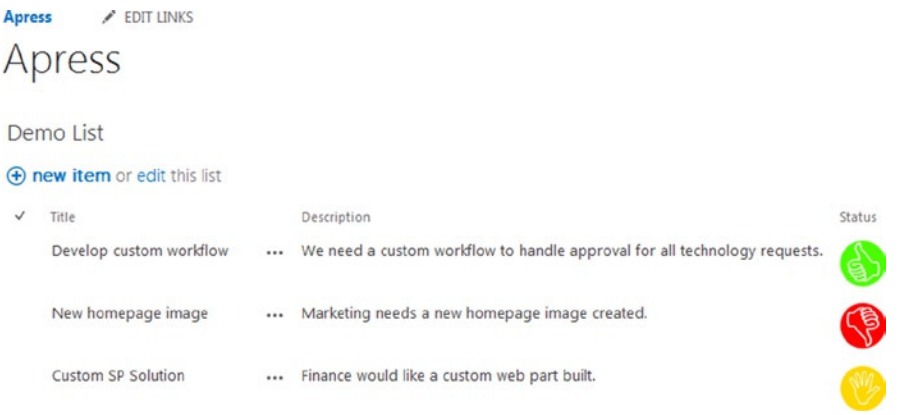

Figure 5-18. *Field template rendering using JSLink*

You can also click the image and see the status text in a JavaScript alert, as shown in Figure 5-19.

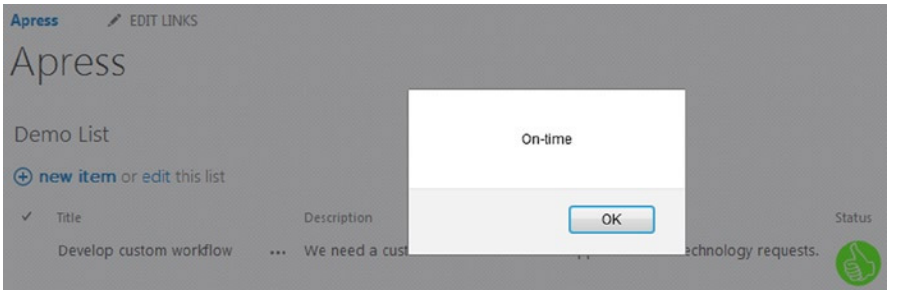

Figure 5-19. *Click event on the status image*

You can override multiple fields by supplying multiple override functions. For instance, if we wanted to apply a custom render to the Description field as well, we could update the code to the following:

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Fields =
    {
        'Status': { 'View': overrideTemplate },
        'Description': { 'View': overrideDescTemplate }
    };
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();

function overrideDescTemplate(ctx) {
    return "<span style='font-weight:bold;'>" + ctx.CurrentItem.Description + "</span>";
}
```

The Fields property will accept multiple overrides by simply placing a comma between each one. In this example, we've created another function, called `overrideDescTemplate`, which simply returns the current item's Description field wrapped in a SPAN element that provides a bold font. The complete script in Visual Studio is shown in Figure 5-20.

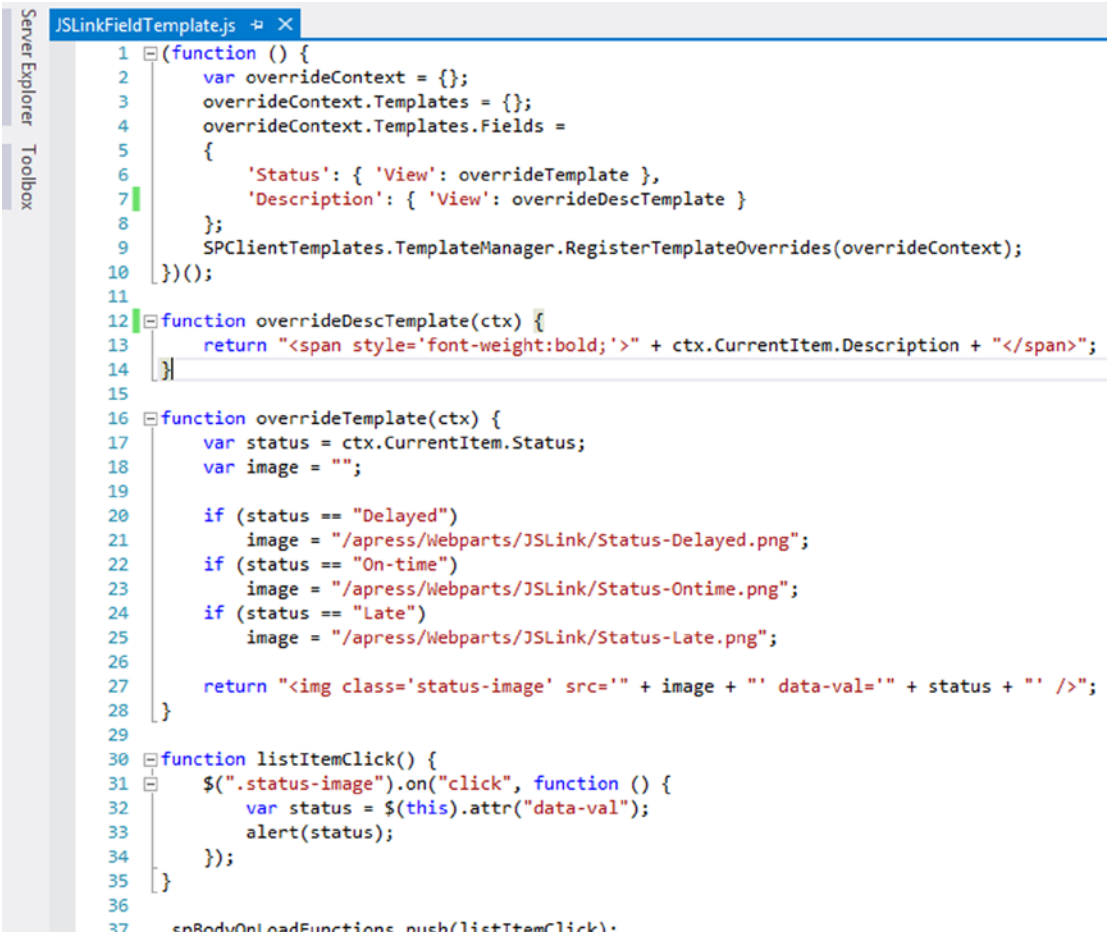

Figure 5-20. *The updated JSLinkFieldTemplate.js file in Visual Studio*

■ **Note** Some fields cannot be overridden using JSLink. A Taxonomy field, for instance, does not allow a different rendering. You can imagine the headaches this would cause if you could override the rendering of a field as complex as a managed metadata column. If you try to override any field that does not allow this, it will simply result in SharePoint ignoring your custom rendering function. This is nice because it means your script will not break the page; however, it can be frustrating because nothing seems to happen! If you notice this, take a step back and remember that some fields can't be overridden.

As you can see in this updated code sample, JSLink provides all the tools needed to override one or more fields. Simply provide a function to handle the rendering and define the override. Save this file and upload it to the document library. Figure 5-21 shows the List View Web Part rendered with the new JSLink file.

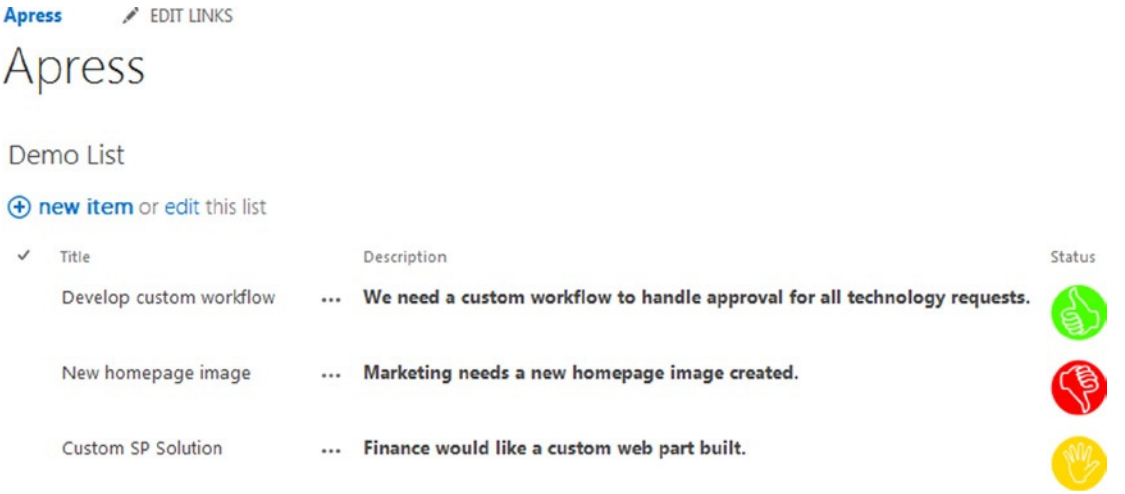

Figure 5-21. *Updated JSLink file rendering multiple fields on the web part*

Now you've seen multiple examples of using JSLink to render your List View Web Parts. Whether you need to override specific fields or completely change the way that the web part looks, you have all the tools you need to get a custom solution. You can take JSLink even further by loading multiple scripts to help you create robust solutions.

Loading Multiple Scripts

In your development efforts with JSLink, you may find that your file is getting large, or perhaps you have some other scripts that you would like to load as well. Luckily, this is easily done with JSLink. JSLink allows multiple files to be added by simply using a pipe, |, placing it between each file path in the JSLink property. For instance, if we had multiple files in our folder, we could use the following path:

```
~site/Webparts/JSLink/File1.js|~site/Webparts/JSLink/File2.js
```

The combined paths with the | separator would be placed in the JSLink property of the List View Web Part. We can update the previous code example to illustrate this. First, we'll go back to a simplified version of the sample with the following code:

```
(function () {
    var overrideContext = {};
    overrideContext.Templates = {};
    overrideContext.Templates.Fields =
    {
        'Status': { 'View': overrideTemplate }
    };
    SPClientTemplates.TemplateManager.RegisterTemplateOverrides(overrideContext);
})();
```

```
function overrideTemplate(ctx) {
    var status = ctx.CurrentItem.Status;
    var image = "";

    if (status == "Delayed")
        image = "/apress/Webparts/JSLink/Status-Delayed.png";
    if (status == "On-time")
        image = "/apress/Webparts/JSLink/Status-Ontime.png";
    if (status == "Late")
        image = "/apress/Webparts/JSLink/Status-Late.png";

    return "<img class='status-image' src='" + image + "' data-val='" + status + "' />";
}
```

This sample is basically back to the first Field code sample where we are only overriding the Status field with an image. The custom jQuery code that rendered an alert has been removed, so we'll move this to another file. The script in Visual Studio is shown in Figure 5-22.

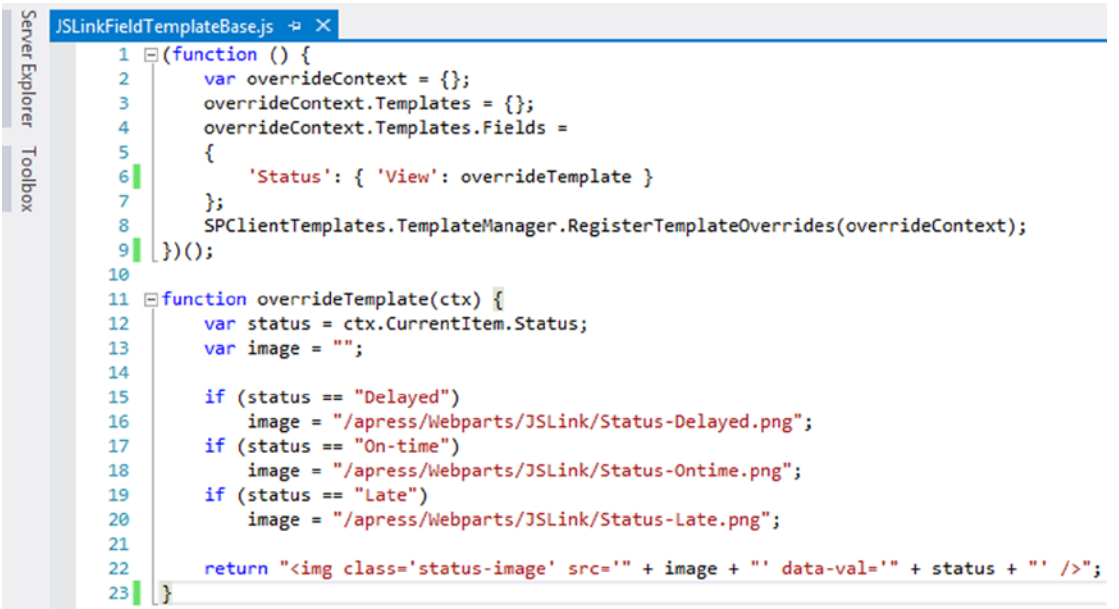

Figure 5-22. *JSLink file that only renders the Status field on the list*

Save this file and name it **JSLinkFieldTemplateBase.js**. Next, we'll create the companion file to handle the jQuery click event functionality. Create a new file using File ➤ New ➤ File. Choose the Web category, and then choose "JavaScript file" from the options. We'll use the code from the previous section, which added the click event to the DIV using jQuery with the following:

```
function listItemClick() {
    $(".status-image").on("click", function () {
        var status = $(this).attr("data-val");
        alert("STATUS: " + status);
    });
}

_spBodyOnLoadFunctions.push(listItemClick);
```

This is the same code as before, with the addition of the text "STATUS:" in the alert. This is mainly to show that we are using a new file and that JSLink is loading both files. Save this file and name it **JSLinkFieldTemplateExt.js**. The script in Visual Studio is shown in Figure 5-23.

Figure 5-23. *JSLinkFieldTemplateExt.js file in Visual Studio*

Now that both files have been created, upload these to the JSLink folder in the Webparts document library. Once the files are uploaded, you can craft the combined paths, and it should look like the following:

```
~site/Webparts/JSLink/JSLinkFieldTemplateBase.js|~site/Webparts/JSLink/
JSLinkFieldTemplateExt.js
```

Place the List View Web Part into Edit mode and update the JSLink property with the combined path. Click Apply and then save the page. At first glance, the custom rendering has not changed much. Click a status image to trigger the click event, as shown in Figure 5-24.

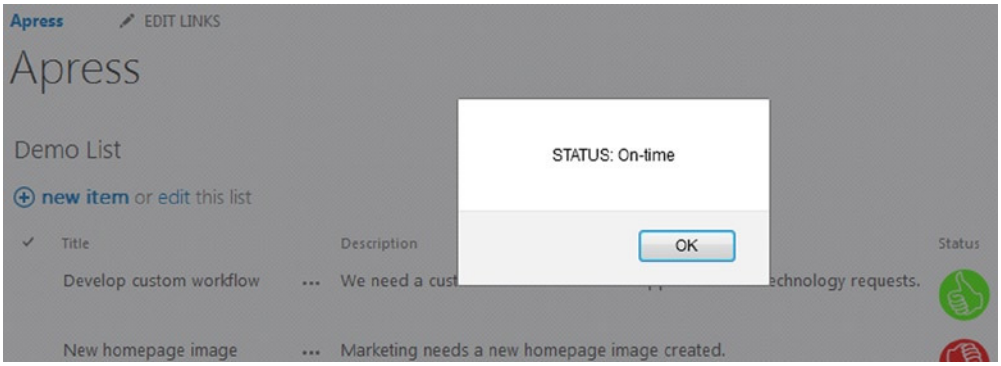

Figure 5-24. *Updated click event from the second JavaScript file*

In the alert, you can see the status with the additional "STATUS:" text displayed. This is confirmation that the two JavaScript files have been loaded via JSLink. Although this is a very simple example, it should illustrate the power of loading multiple files. As your scripts get larger and larger, you now have the flexibility to split your code into multiple files and reference them all via JSLink.

Summary

In this chapter, we explored the power of JSLink as a way to build custom solutions in SharePoint. We saw how you can use this method to override the default rendering of the List View Web Part. You can use either the "Item" template to completely customize the overall rendering, or use the "Fields" template to override single or multiple fields in the list. We also saw how to break up your code into multiple files. Hopefully, this chapter demonstrated what is possible with JSLink. In the next chapter, we'll continue to look at new functionality in SharePoint using display templates and result sources to create custom search solutions.

CHAPTER 6

Content Search Web Part and Display Templates

Since SharePoint began, search has always played a major role in its usage. Whether it's a site search, an Enterprise search, or a custom solution, one of the key benefits to SharePoint is being able to easily find things. One of the ways to facilitate search in SharePoint is by way of the Content Search Web Part. This can be easily placed on a page and configured to return specific results for your users. It's a tried-and-true web part that has been around a long time. In the latest version of SharePoint, the Content Search Web Part utilizes new functionality in the form of *result sources* and *display templates*.

Result sources allow you to craft specific search criteria to quickly target the content that you are looking for. Display templates allow you to style and render those search results to your business needs. This new functionality offers developers another avenue for creating rich, custom solutions in SharePoint. In this chapter, we'll look at these areas and create some custom search solutions.

Adding Documents

Before you can begin building solutions using search in SharePoint, you'll need some documents to actually search for. If you're using an existing SharePoint environment, you probably already have some documents to search for. If you've set up your environment for this book, you'll need to upload some documents to perform some searches. In this example, you'll have a variety of document types to search for, including Word, Excel, and PDF. Figure 6-1 shows some samples files in a document library.

Note The documents used in this example can be downloaded from Apress. Visit Apress (www.apress.com) and search for this book title to locate the documents and source code.

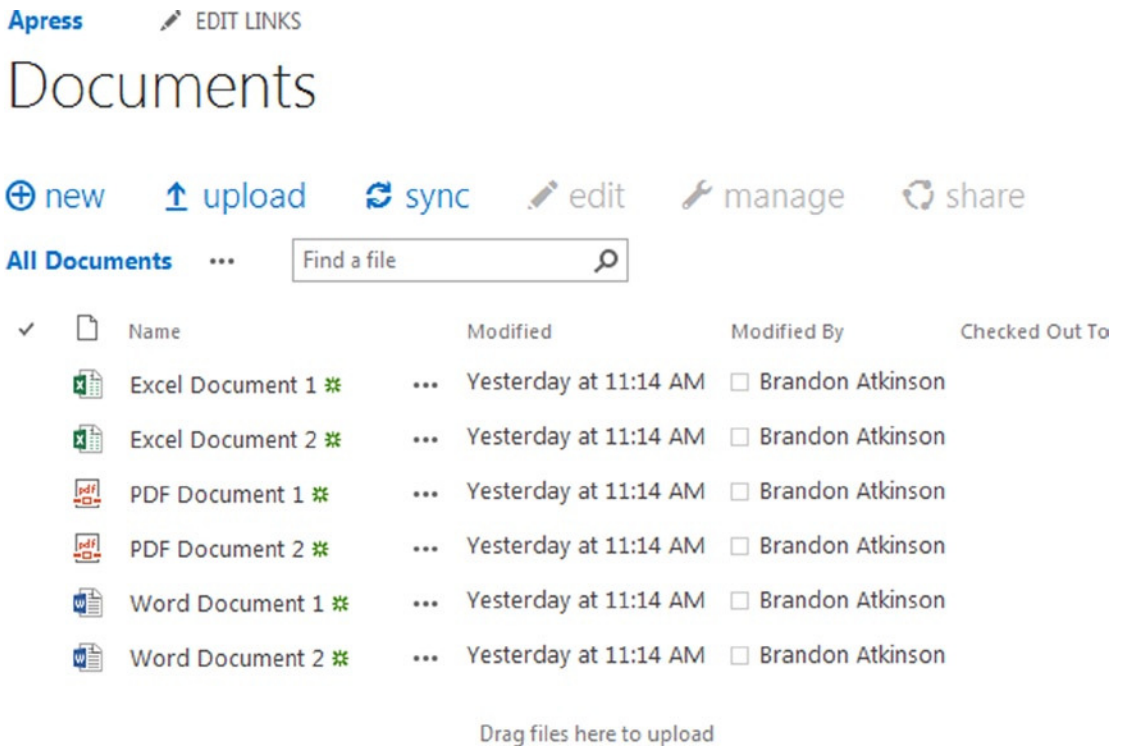

Figure 6-1. *Excel, PDF, and Word documents in the Documents library*

I've used the out-of-the-box "Documents" document library that was created when I created my team site called Apress. No matter where you store your documents, the main takeaway here is to just upload some files into SharePoint so that you can search for them. Now that some files are uploaded and will be eventually indexed, you can create a custom result source to facilitate searches.

Result Sources

In SharePoint 2010, you had the concept of custom search scopes. They allowed you to limit search results to specific content enabling more targeted searches for your end users without them having to know how to craft complex search queries. In SharePoint 2013, you can accomplish the exact same thing with custom result sources.

When a search is performed in SharePoint 2013, the system matches the query with a result source to provide results. A result source is a definition that specifies the following:

- *Protocol* - This can be the local search service index, a remote SharePoint installation, Exchange, or even another search engine that uses the OpenSearch protocol.
- *Source URL* - If you are using a protocol other than Local SharePoint, you will need to provide a URL for the service returning the search results.

- *Type* - When using Local or Remote SharePoint, you can specify either SharePoint Search or People Search.
- *Query transform* - Using a query transform, you can narrow results for the given search using KQL property restrictions.
- *Credentials* - You can specify credentials to use for the search.

SharePoint comes with many preconfigured result sources for you to use, including Conversations, Documents, and Pictures, to name a few. Custom result sources can also be created at various levels throughout SharePoint, and where you create the result source determines the level of permissions that you'll need. The following lists the levels at which result sources may be created and the permissions required when creating them:

- *Site Collection* - You must be a site collection administrator, and the result source can be used by any site within the site collection.
- *Site* - You must be a member of the Owners group for the site, and the result source can only be used by that site.

You can also specify a result source at the Search Service Application level, or the SharePoint Admin Center for Office 365, and the result source is available to any site collection that consumes the Search Service Application. This level won't be covered in this book, however, the steps to create the result source at this level is also exactly the same.

The first thing you need to decide when creating a custom result source is the level to create it at. This depends on how you intend to use the source; for instance, if it's to highlight items in a specific site, then the Site level would make sense. But if you want a more encompassing result source, you may want to create it at the Site Collection level. The steps are exactly the same for both. If you plan to create a new result source at the Site Collection or Site level, you can simply navigate to the Site Settings page and choose Result Sources under the Search heading for Site level, or Search Result Sources under the Site Collection Administration heading for Site Collection level. Figure 6-2 shows the Site Settings page for the site collection.

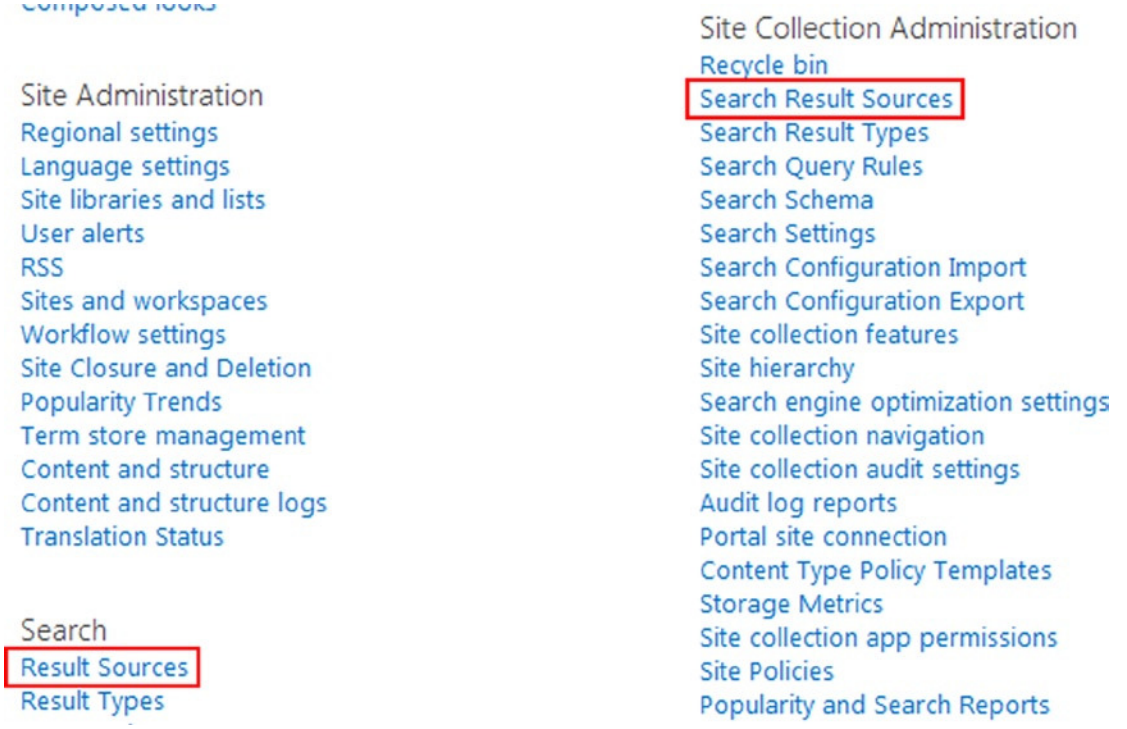

Figure 6-2. *Accessing Result Sources from the Site Collection site settings page*

As you can see at the site collection level, you can create a result source for the entire site collection or just the site at the root level. For this example, we'll navigate to the Apress subsite where our other code samples live, and create a new result source for only this site, as shown in Figure 6-3.

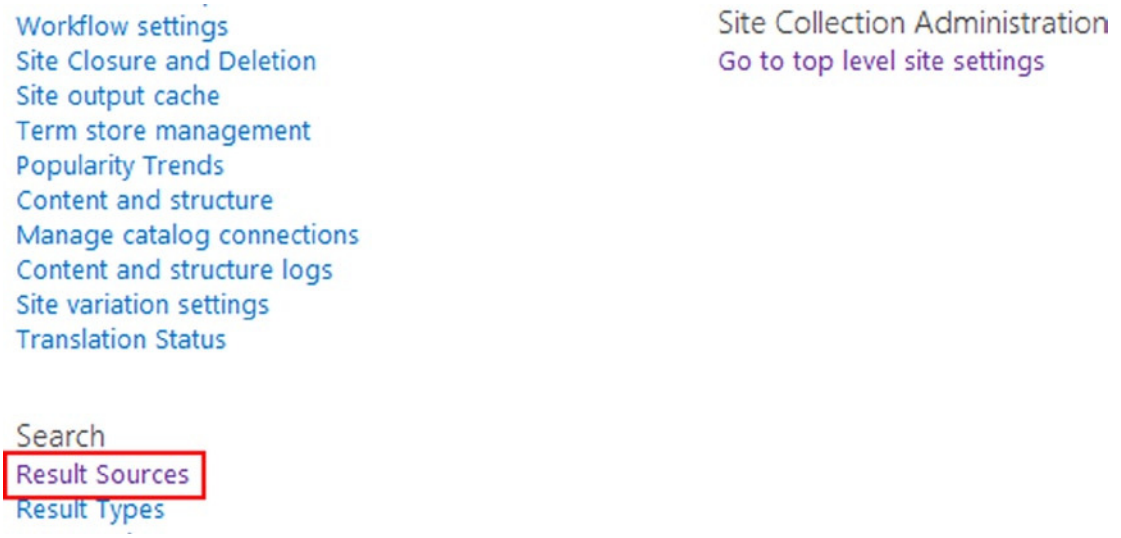

Figure 6-3. *Accessing Result Sources from the Site's site settings page*

Clicking the Result Sources link will bring you to the Manage Result Sources page. This page displays all the current result sources that are set up. It also includes all the out-of-the-box result sources that SharePoint comes with. You cannot change any of the out-of-the-box result sources; however, you can view their settings. To begin creating a custom result source, click the New Result Source link at the top of the page, as shown in Figure 6-4.

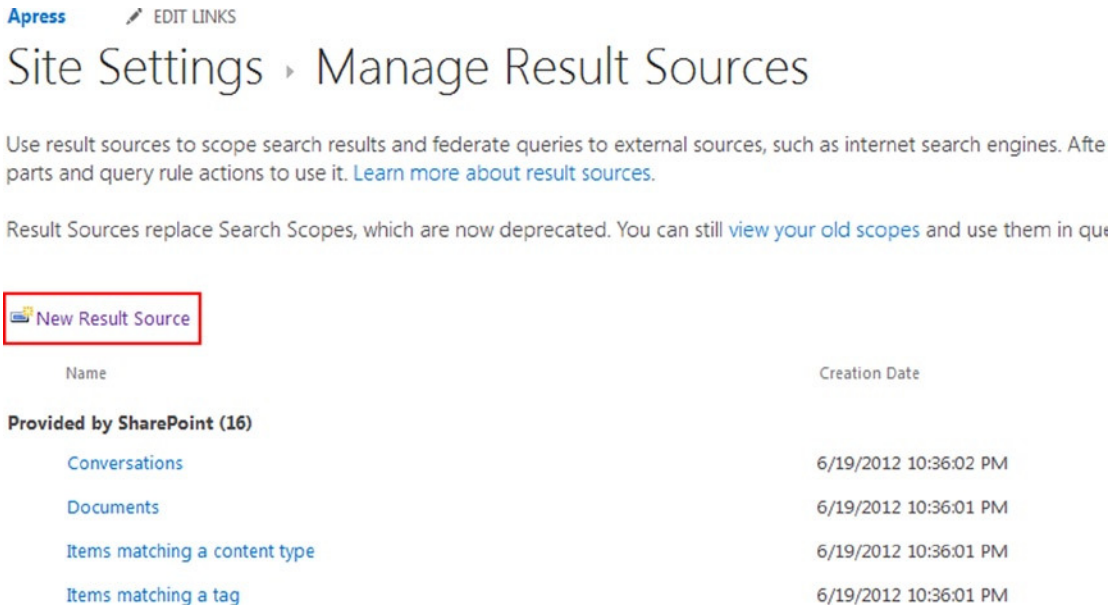

Figure 6-4. *Manage Result Sources page in site settings*

Once you are on the Add Result Source page, the first thing you'll need to supply is a name and description. The description is optional, but you should provide one for other users. Figure 6-5 shows the initial properties required for a new result source.

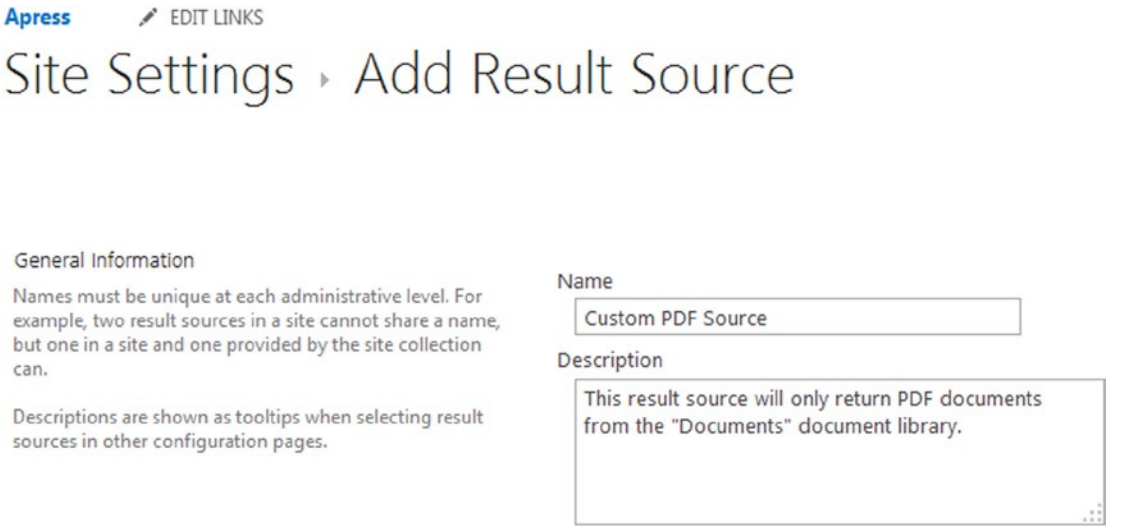

Figure 6-5. *Providing a name and description for the new result source*

In this example, you'll create a result source that will only return PDF documents from the Documents library in the site. To that end, we'll provide the title **Custom PDF Source** and a proper description so that users know exactly what to expect when using this result source. Next, you need to supply a protocol and type; in this case, it the protocol will be Local SharePoint and the type will be SharePoint Search Results, as shown in Figure 6-6. Depending on your needs, you may need to select a different protocol and type and supply additional information. In this scenario, the page will update to show the additional fields required for the selected options.

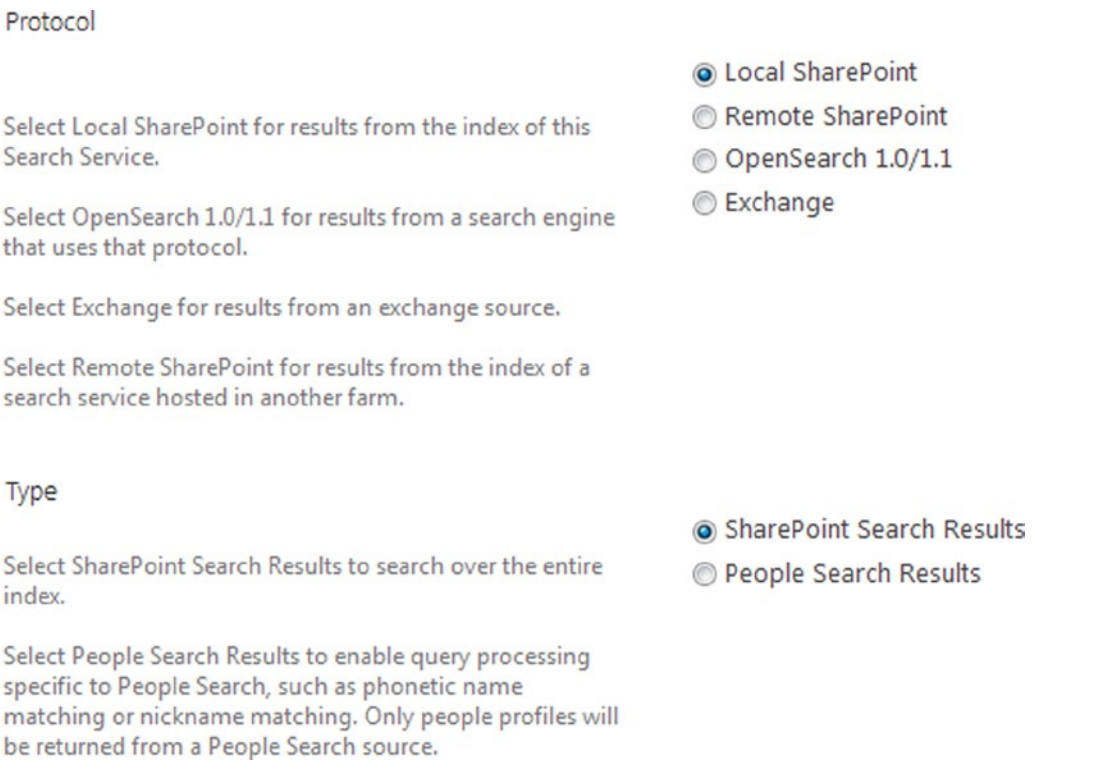

Figure 6-6. *Protocol and Type options for the new result source*

Next, you'll need to provide a query transform that will actually do the limiting of the search results. This is done using Keyword Query Language, which is a set of search options, property restrictions, and operators. The query transform will always start off with the token {searchTerms}; this will be replaced with whatever the user enters as their search term. After that, you can provide any valid property restrictions and/or operators. We'll dig deeper into this in a moment. For now, you can see the {searchTerms} token in the textbox, as well as the Launch Query Builder button, as shown in Figure 6-7.

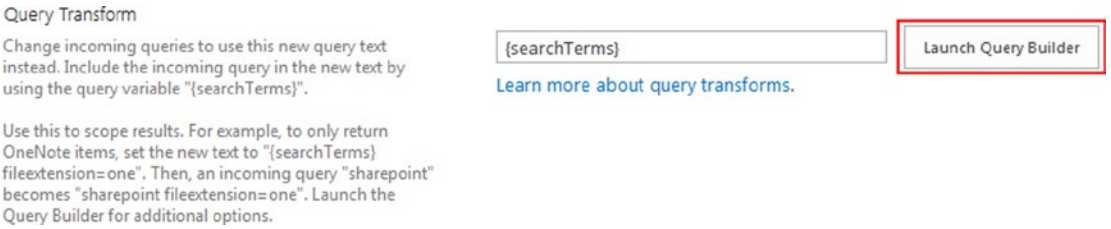

Figure 6-7. *Query Transform options and the Launch Query Builder option*

Finally, you can specify credentials to be used with this result source. You can specify the Default Authentication to use the user's authentication when performing a search. This is usually the desired result. The authentication types are shown in Figure 6-8.

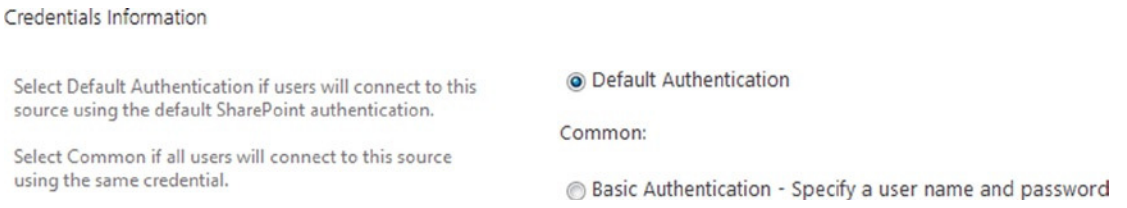

Figure 6-8. *Credentials Information for the result source*

Click Save to create the new result source. You will be taken back to the Manage Result Sources page where the custom result source will be listed, as shown in Figure 6-9. The new result source is listed under the heading "Defined for this site" at the top of the page.

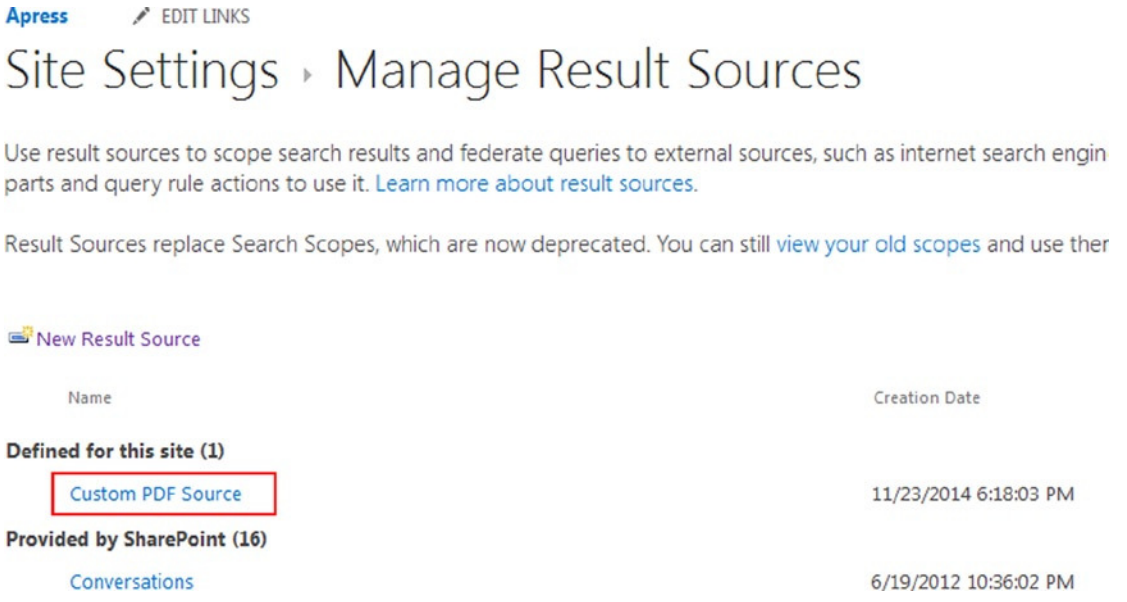

Figure 6-9. *The new result source in the Manage Result Sources page*

Since we did not provide any details in the Query Transform section, this new result source will simply take the user's input and perform a search. We definitely did not need to create a new result to achieve that result. Let's update it so that it will only return PDFs from the Documents folder. Open the result source context menu and click Edit, as shown in Figure 6-10.

Figure 6-10. *Editing the result source we just created*

Editing the result source takes you to the Edit Result Source page and you will see all the previous settings that you just entered. Scroll down to the Query Transform section and click the Launch Query Builder button, as shown in Figure 6-11.

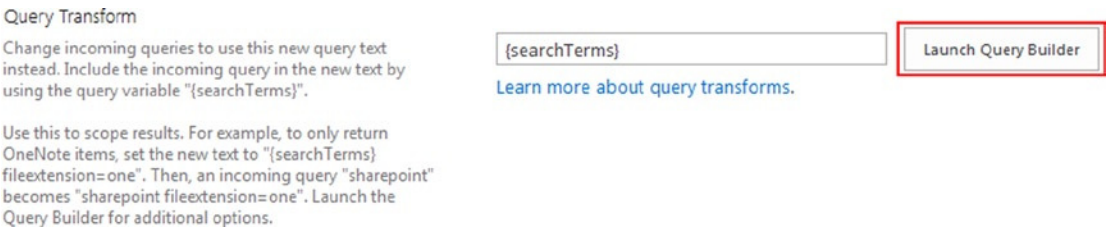

Figure 6-11. *Query Transform section with the Launch Query Builder button*

The Build Your Query modal window opens, which will help you build your query transform. There are many out-of-the-box filters that you can use in this window by accessing the drop-down lists that are provided. You can apply sorting and test your queries here as well. You can also opt to just type in your query, as shown in Figure 6-12.

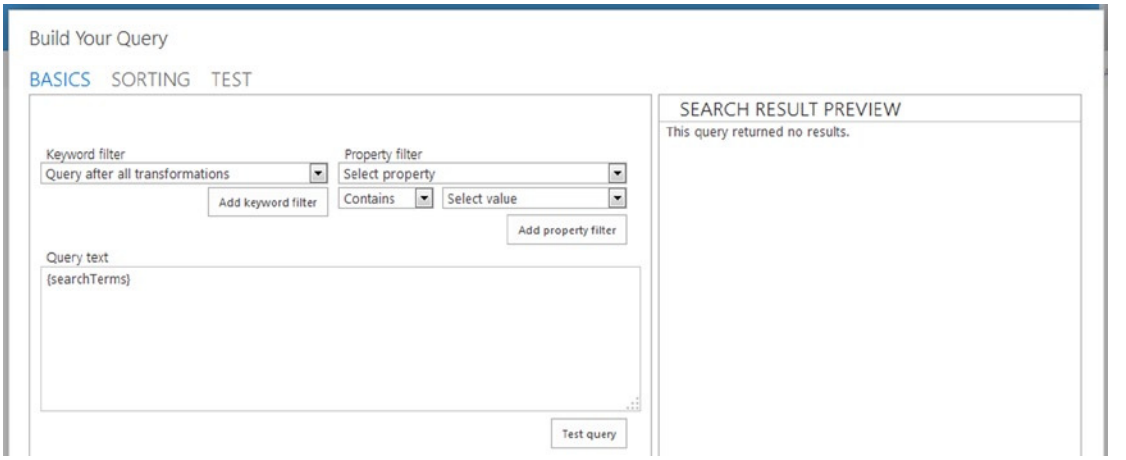

Figure 6-12. *The Build Your Query modal window*

Queries are built in SharePoint using Keyword Query Language, or KQL for short. Using KQL you can specify the search terms or property restrictions that are passed to the SharePoint search service. A KQL query is not overly complicated and consists of one or more of the following elements:

- *Free-text keywords* - These can be words or phrases, such as whitepaper, event, Bob Smith, and so forth.
- *Property restrictions* - These can be the file type, author, file name, and so forth.

Property restrictions do pretty much what their name implies: they restrict the search results. For instance, you could add a FileType restrictor and only return a certain document type by using `FileType:docx` to only return Word files. There a lot of property restrictions available, making KQL very robust in your search needs.

■ **Note** Keyword Query Language, while easy to implement, has a very rich and deep syntax. It is great for crafting queries that exactly meet your needs, but it is way too large a subject for this book. We'll simply scratch the surface on what you can accomplish. For detailed information on KQL, property restrictions, and more, visit `http://msdn.microsoft.com/en-us/library/office/ee558911(v=office.15).aspx`.

The KQL query must contain one free-text keyword, or one property restriction, or a combination of the two to be valid. To begin, we'll remove the {searchTerms} part of the query. We won't use this since our Content Search Web Part will simply return results without any interaction from the user. Next, we'll add a property restriction using the `path` property:

```
path:"https://brandonatkinson.sharepoint.com/apress/Shared Documents"
```

The `path` property instructs the query that we only want results from the specified path and anything below it, which includes folders, subsites, and so forth. The path must be a fully qualified URL, and if it includes any spaces, the URL must be wrapped in quotes. In my environment, I'm using the out-of-the-box Documents folder, which is actually the Shared Documents folder. You can always double-check the actual name of the folder by clicking it in the navigation panel and inspecting the URL in the address bar of the browser. Placing this property restriction in the Query Text textbox and clicking the Test Query button will produce the results shown in Figure 6-13.

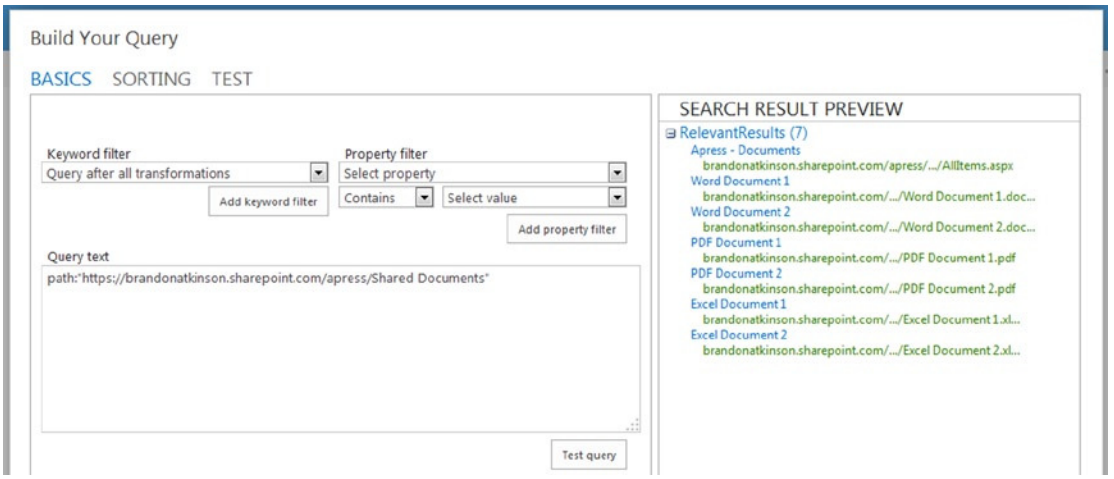

Figure 6-13. *Path property restriction returning only files from the specific document library*

The property restriction is limiting the search results to the Documents library, and only seven documents are being returned. We can now limit it once again by using the `FileType` property:

```
FileType:pdf
```

This property instructs the query to only return files that have a `.pdf` extension. Adding this to the query and clicking the Test Query button will further limit the results, as shown in Figure 6-14.

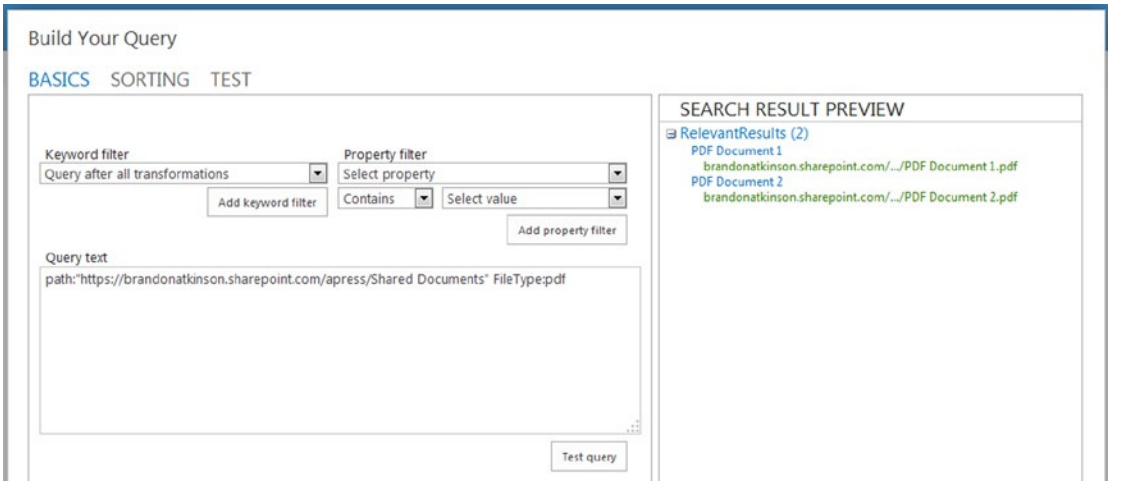

Figure 6-14. *FileType property restriction limiting results to only PDF files*

Now only two files are being returned because we only have two PDF files in the Documents library. Although this is a very simple example, it should illustrate the power that KQL provides when constructing your queries. With only two property restrictions, we've been able to limit the search results quite a bit. You can imagine that if you had a requirement to display only Office files located in a particular site, no matter which folder they were in, your KQL query would look something like the following:

```
path:"http://SERVER/SITE" FileType:docx FileType:xlsx FileType:ppt
```

As you can see, you can combine the same type of property restrictors to craft exactly what you need. KQL is very flexible; you should have no problems finding the right combination of search terms and property restrictions to meet your needs.

Now that we the query set up the way we want it, click OK in the Build Your Query window. Then click Save on the Edit Result Source page. The custom result source is now set up and ready to use. Next, let's place a Content Search Web Part on the page and consume the result source.

Content Search Web Part

The Content Search Web Part is another workhorse that is new to SharePoint 2013. When the Content Search Web Part is placed on a page, it automatically issues a search query when a user loads the page. Depending on how the web part is configured, the user will most likely not even know that a search is taking place. This makes the Content Search Web Part a powerful way to create custom solutions that are based on search. It's an easy web part to use and configure, and as you saw in the previous section, it's very easy to set up specific search queries for your needs.

■ **Note** The Content Search Web Part is available as part of the Search Web Parts feature. This is available only to the SharePoint Online E3 and E4 plans, and on-premises for Enterprise licensing. If you do not have these, you will find that the Content Search Web Part is not available for adding a new web part to the page. You can find more information about the features available for SharePoint Online plans at http://technet.microsoft.com/en-us/library/sharepoint-online-service-description.aspx.

To begin, navigate to the page where the other web parts have been placed for this book, and then remove any existing web parts from previous chapters. With the page still in Edit mode, place a new Content Search Web Part by selecting the Insert tab from the Ribbon. Choose Content Rollup from the Categories section, and then choose Content Search, as shown in Figure 6-15.

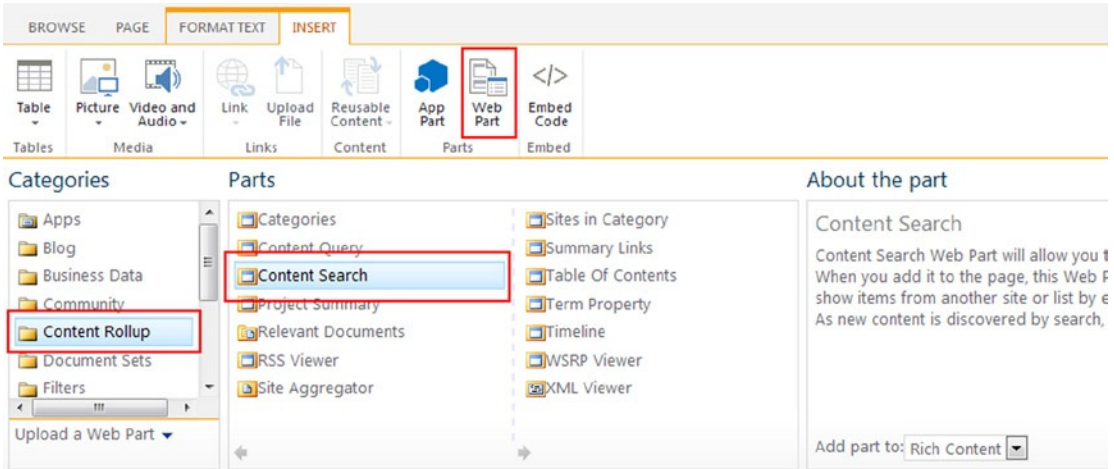

Figure 6-15. *Inserting the Content Search Web Part onto the page*

Click Add to place the web part on the page. The web part will display search results immediately. The default search for the Content Search Web Part is called Recently Changed Items and it displays just that—a list of recently changed items. In this case, the list is comprised of the documents that were recently added to the Documents library, as shown in Figure 6-16.

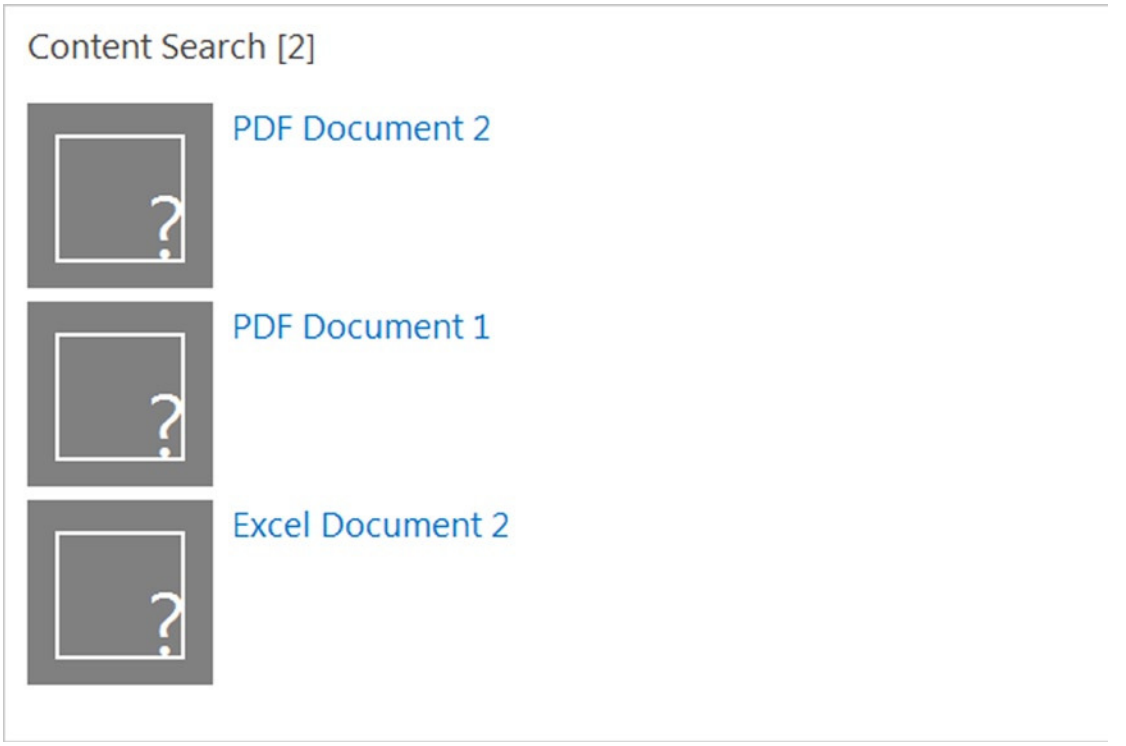

Figure 6-16. *The Content Search Web Part using the Recently Changed Items search*

At first glance, this web part is doing almost what we wanted from the beginning, which is to show only PDF documents. The fact that the two PDF documents are shown in this web part is only by the luck of the draw. For instance, depending on the order in which you uploaded your documents, you may see two Word documents at the top. We'll update the web part to use the custom result source that we created in the previous section to ensure that we only get PDF documents in the results. Place the Content Search Web Part into Edit mode, and in the Properties accordion pane, click the Change Query button, as shown in Figure 6-17.

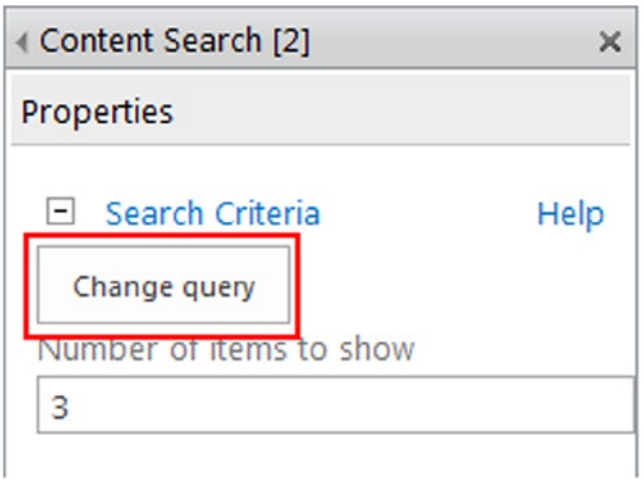

Figure 6-17. *Accessing the query settings of the Content Search Web Part*

Clicking this button opens the Build Your Query modal window. Under the Basics tab, you'll see the Select a Query header and the selected option, Recently changed items (System), as shown in Figure 6-18.

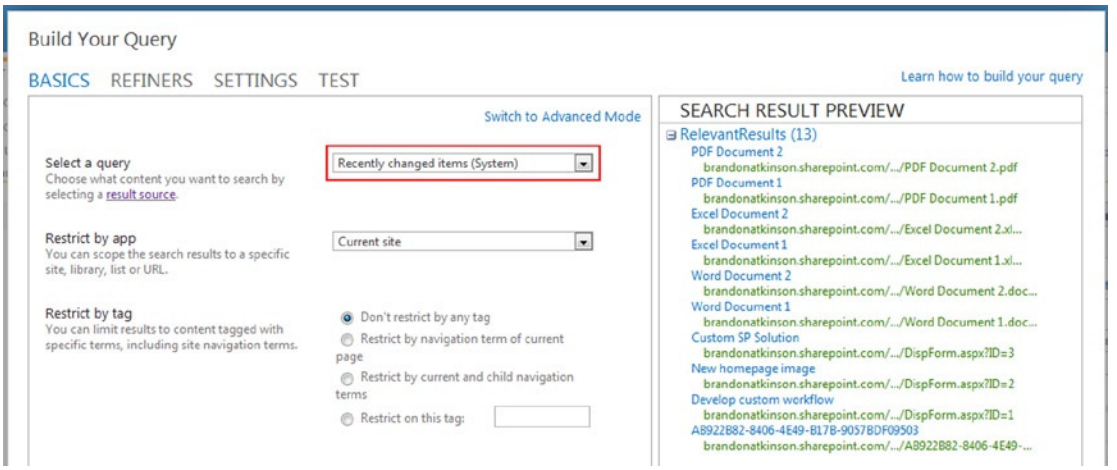

Figure 6-18. *The Build Your Query modal window with the "Recently changed items" result source selected*

As you can see in this window, the Content Search Web Part has a lot of settings—and these only concern the search query! This web part can be configured to give you very exact results and it is able to meet a wide variety of business needs. For now, let's change the result source to the custom PDF source. Click the link in the top-right corner of the window titled Switch to Advanced Mode, as shown in Figure 6-19.

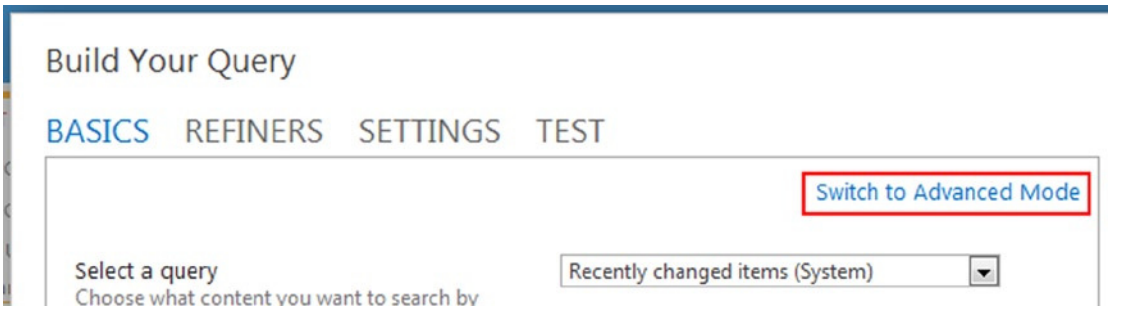

Figure 6-19. *Switch to Advanced Mode link in the Build Your Query modal window*

Clicking this link brings up a window that looks a lot like the Build Your Query window that you saw when building a custom result source. You can type in a query, use keyword filters and property filters, and even see a preview of the search query in the right pane. All of these settings were available when we created the custom result source; however, a new drop-down is here, called Select a Query. Opening this will display a list of result sources that you can choose to use for the query. Open the drop-down list and choose Custom PDF Source, as shown in Figure 6-20.

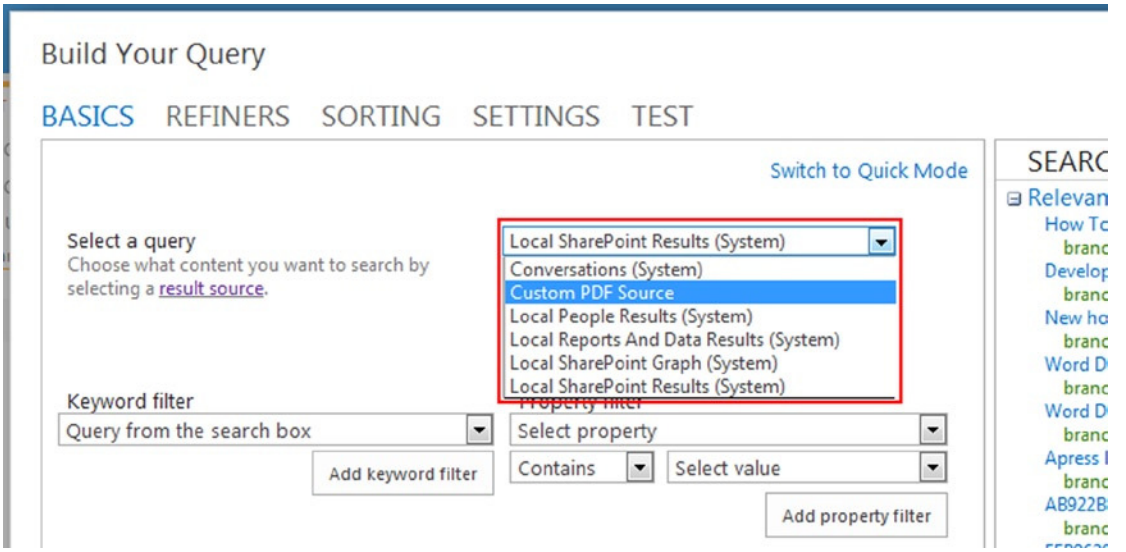

Figure 6-20. *Choosing the custom result source Custom PDF Source from the drop-down menu*

Once you have chosen a result source from the drop-down, click the Test Query button at the bottom of the modal window to preview the custom result source query. As shown in Figure 6-21, it's performing the proper search of only returning PDF documents from the Documents folder.

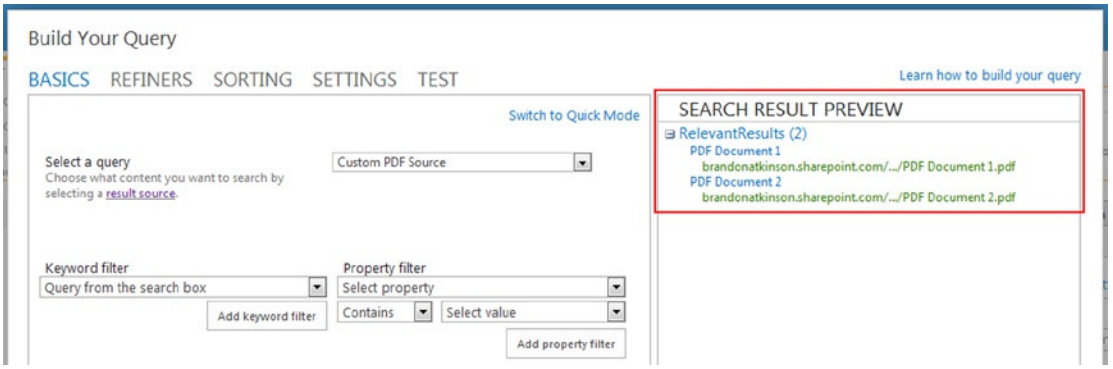

Figure 6-21. *The Custom PDF Source result source search shown in the preview window*

Click OK in the Build Your Query modal window. Next, click OK in the Content Search Web Part tool pane to save the changes. Save the page. The custom result source will be active, showing only PDF documents, as shown in Figure 6-22.

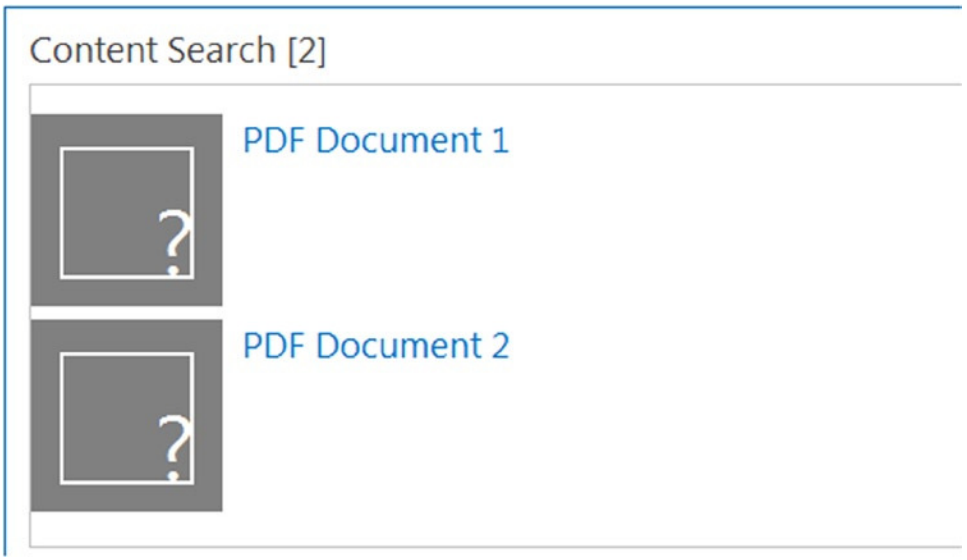

Figure 6-22. *Custom result source displaying only PDF documents*

At this point, we've created a custom result source and applied it to a Content Search Web Part with success. Only the documents specified by the result source are returned and displayed to the user. However, the display is nothing to be desired. Whereas the content is correct, it does not look very nice. Next, let's explore display templates and see how you can apply custom styling to match the custom results.

■ **Note** The Content Search Web Part is quite complex. It has many settings that can be configured to help you achieve very precise results for your solutions. For more detailed information on these settings, please visit https://support.office.com/en-us/article/Configure-a-Content-Search-Web-Part-in-SharePoint-0dc16de1-dbe4-462b-babb-bf8338c36c9a.

Display Templates

SharePoint allows you full control over your search results, down to how they appear on the page. This is accomplished with display templates. You can use display templates to style your search results however you like using HTML. There are two types of display templates that you can create:

- *Control* - These templates handle the overall structure of how search results are displayed, and can include things like headers and footers. This template will only be rendered once per web part.
- *Item* - These templates handle how each individual search result item is displayed. This template will be rendered once for every search result that is returned.

You can see these display templates by placing the Content Search Web Part into Edit mode, and in the tool pane you will find the Display Templates header in the Properties accordion, as shown in Figure 6-23.

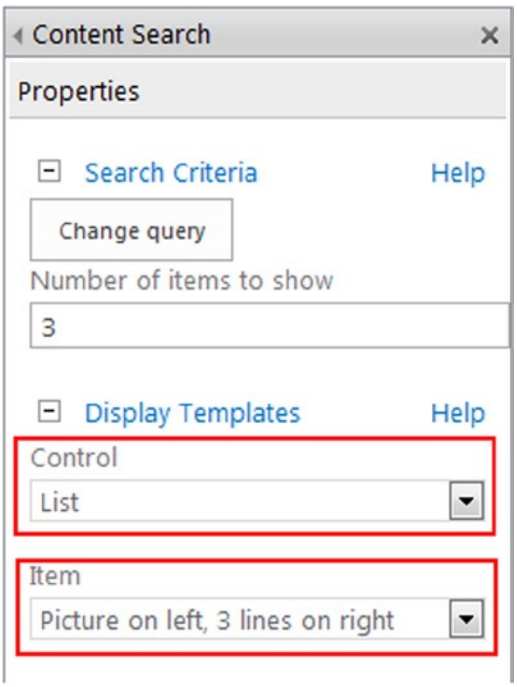

Figure 6-23. *Control and Item display templates in the Content Search Web Part tool pane*

There are many out-of-the-box display templates available and you can view them by clicking the drop-down list for each type. For instance, the Control templates include List, List with Paging, and Slideshow. The Item templates have even more options to choose from. To begin creating a new display template, the easiest place to start is with an existing template. To view all the current search display templates, navigate to the Master Page Gallery, which is found under Site Settings and the Web Designer Galleries. Click the "Master pages and page layouts" link, as shown in Figure 6-24.

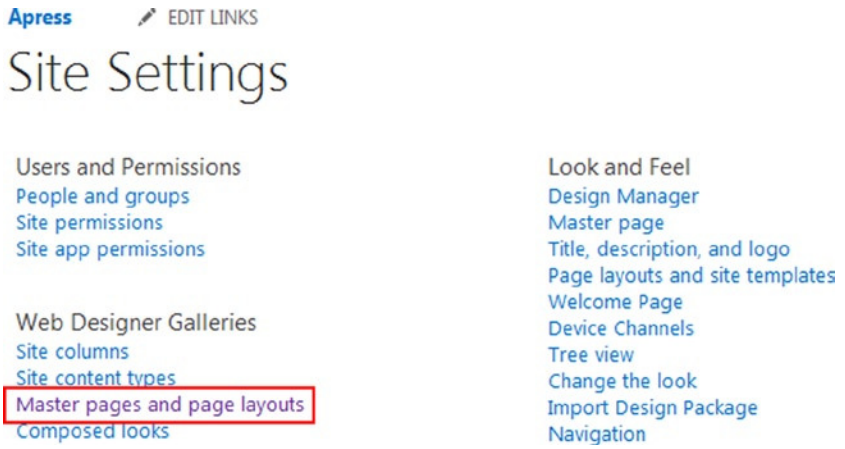

Figure 6-24. *The "Master pages and page layouts" link in Site Settings*

Once you are in the Master Page Gallery, you'll notice it has a lot more in it than previous versions of SharePoint. Near the top there is a folder called Display Templates. By opening this folder you'll see several more folders that correspond to different areas in SharePoint; one of these is Content Web Parts, as shown in Figure 6-25.

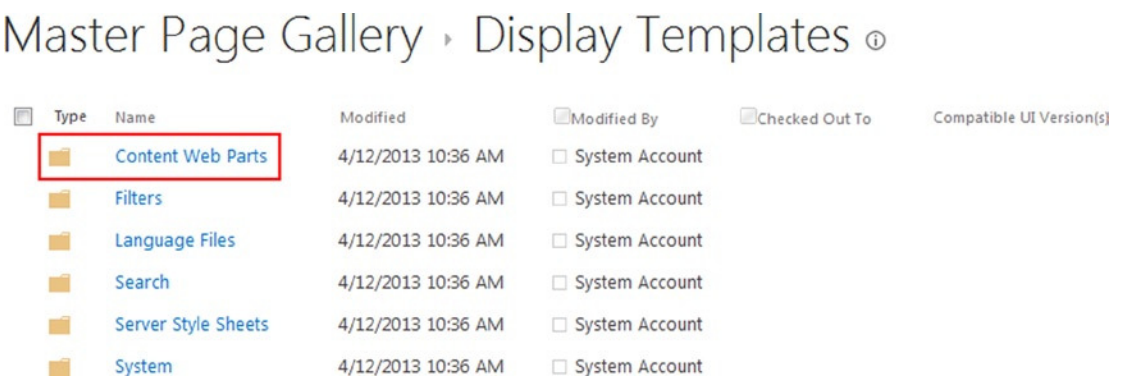

Figure 6-25. *Content Web Parts folder inside the Display Templates folder*

Open the Content Web Parts folder to view all the available display templates that come out of the box with SharePoint. As you can see in Figure 6-26, there are quite a few in here. At the top of the page you can see the Control templates that you saw in the Content Search Web Part tool pane.

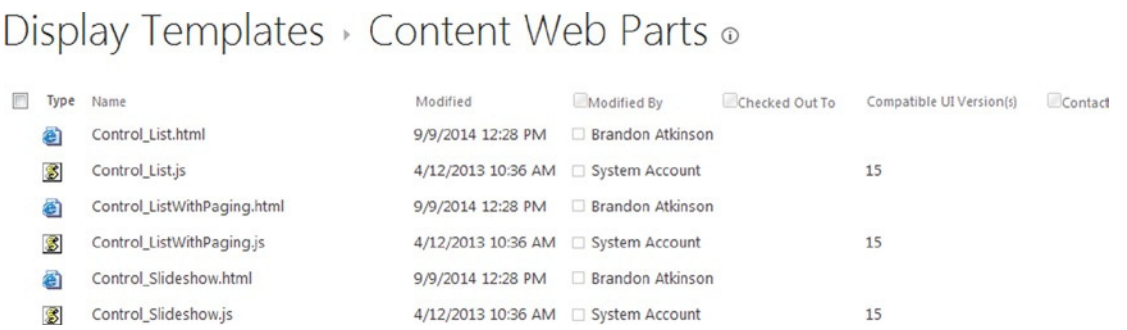

Figure 6-26. *Out-of-the-box display templates in the Content Web Parts folder*

Each display template has a corresponding JavaScript file associated with it. Behind the scenes, SharePoint is using this JavaScript file to render the display templates. You do not need to create or modify this file; when you upload a new display template to this folder, a corresponding JavaScript file is created for you, which we'll see later in this chapter.

■ **Note** Display templates can be very simple or very complex. You have a good deal of control over what properties are available and how the content is laid out, and you can even add additional JavaScript to your templates. The overall topic of display templates is quite large and too complex to completely cover in this chapter. For more information, please visit `http://msdn.microsoft.com/en-us/library/office/jj945138%28v=office.15%29.aspx`.

To start building a new display template, we'll download a copy of another template and use it as a base. Display templates can be quite complex, and building one from scratch can be a painstaking task. Using an existing template to start with will dramatically cut down on the time it takes to create a new template. For this example, we'll focus on an item template and use the default "Picture on Left, 3 Lines on Right" template. In the Content Web Parts folder, this template is titled `Item_Picture3Lines.html`. Check this template in the folder, and from the Ribbon, click Download a Copy, as shown in Figure 6-27.

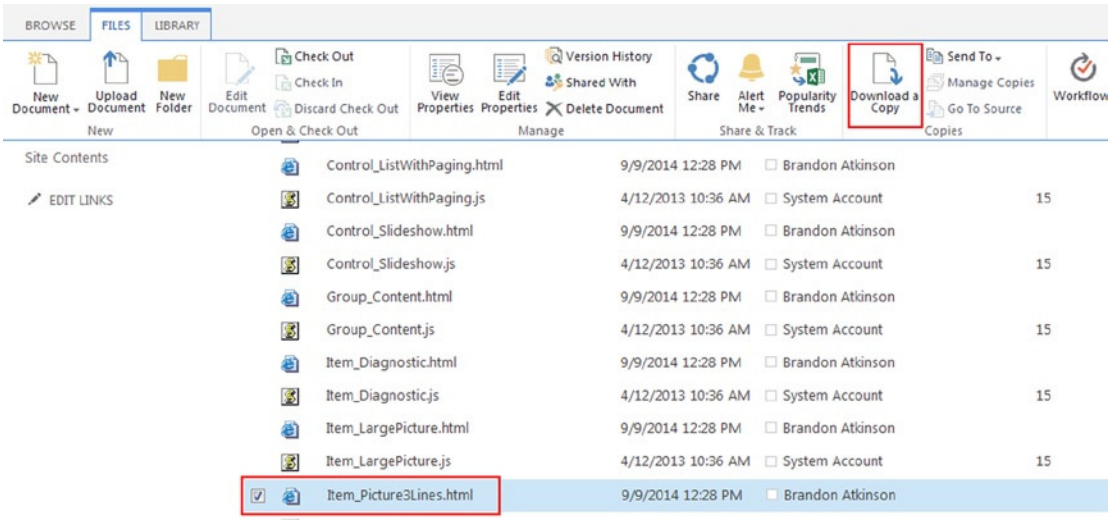

Figure 6-27. *Downloading a copy of an existing display template*

Once you have downloaded a copy, you can open it in Visual Studio. Examine this document and you'll see that it's a little complicated. It has a HEAD section, a TITLE, and a BODY—just as you'd expect an HTML document to have—along with a lot of rendering code. The actual rendering takes place in the BODY. You'll notice inside the BODY there are a couple of DIVs that contain all the logic. The file in Visual Studio is shown in Figure 6-28.

Figure 6-28. *Item_Picture3Lines.html file open in Visual Studio*

Let's break this file down by each section to get a grasp on what's going on. First, go ahead and save the file and name it **Item_CustomPdf.html**. At the top of the file are standard HTML and HEAD tags; there is nothing special about these tags. The TITLE tag is important, as it will be display in the drop-down list when choosing an Item template in the Content Search Web Part. Change the title to **Custom PDF Template**.

```
<html xmlns:mso="urn:schemas-microsoft-com:office:office" xmlns:msdt="uuid:C2F41010-65B3-
11d1-A29F-00AA00C14882">
<head>
<title>Custom PDF Template</title>
```

The next section is a collection of properties in a section titled CustomDocumentProperties. These properties provide additional information about the display template to SharePoint. Some of these are populated automatically for you, and others you will want to update. Here are some of the more important ones:

- `ManagedPropertyMapping` - This property takes the managed properties used by search and maps them to values that can be used by the display template.
- `MasterPageDescription` - This property provides a friendly description for the display template and is shown to users.
- `TargetControlType` - This property specifies the context in which the display template is used; for instance, Content Web Parts.

The `ManagedPropertyMapping` property allows you to use the managed properties in searching and to map them to values that can be used by the display template. It's a comma-delimited list of values that uses the following format:

```
'property display name'{property name}:'managed property'
```

In the `ManagedPropertyMapping` in this display template, for example, you can find the following: `'Link URL'{Link URL}:'Path'`. This takes the managed property `Path` and maps it to a variable called `Link URL`. The `ManagedPropertyMapping` has several managed properties mapped:

```
<mso:ManagedPropertyMapping msdt:dt="string">'Picture URL'{Picture URL}:'PublishingImage;
PictureURL;PictureThumbnailURL','Link URL'{Link URL}:'Path','Line 1'{Line 1}:'Title',
'Line 2'{Line 2}:'Description','Line 3'{Line 3}:'','SecondaryFileExtension','ContentTypeId'
</mso:ManagedPropertyMapping>
```

Once a property has been mapped you can obtain its value using the following JavaScript:

```
var linkURL = $getItemValue(ctx, "Link URL");
```

The `getItemValue()` function takes two parameters. The first is called `ctx` and represents the current context. You'll notice that it's not defined anywhere; it is just available to use. All you need to remember is that you must pass this in and it will always be there to use. The second parameter is a property display name in single quotes used in the `ManagedPropertyMapping` element. In this example, `Link URL` is the property name.

The only other property that we'll update is the `MasterPageDescription` property, just to update the description so that it's not confusing.

```
<mso:MasterPageDescription msdt:dt="string">This Item Display Template will display custom
PDF results.</mso:MasterPageDescription>
```

Immediately after the <body> tag, you'll find a <script> section:

```
<script>
        $includeLanguageScript(this.url, "~sitecollection/_catalogs/masterpage/Display
                                Templates/Language Files/{Locale}/CustomStrings.js");
</script>
```

In this section, you can add your own scripts or CSS references. By default it includes a reference to a file called CustomStrings.js, which is in all the display templates. You can add additional references by using some of the following lines:

- Script located in the same site: $includeScript(this.url, "~site/Webparts/CustomScript.js");
- Script located on an external site: $includeScript(this.url, "http://www.contoso.com/CustomScript.js");
- CSS file located in the same site: $includeCSS(this.url, "~site/Webparts/CustomCSS.css");

Immediately after the <script> tag, you'll find a element that contains all the HTML and code that will be included in the display template. This DIV has an ID that must match the name of the display template file. For instance, the DIV in this example would look like the following:

```
<div id="Item_CustomPdf">
```

After the DIV element, you'll see code inside comment blocks that begin with <!--#_ and end with _#-->. Inside this block you can use any JavaScript code that you would like and this is where you will also obtain all the values for any managed properties. We'll keep this display template simple and only use two managed properties with the following code:

```
<!--#_

var linkURL = $getItemValue(ctx, "Link URL");
var line1 = $getItemValue(ctx, "Line 1");

 _#-->
```

As you can see, we simply declare two variables to hold the values from the managed properties that are defined in the ManagedPropertyMapping. We'll use these two variables in our HTML code. We'll keep the HTML simple as well with the following code:

```
<div style="border:solid 1px Silver;padding:10px;margin-bottom:20px;">
            <strong>_#= line1 =#_</strong>
            <br />
            <a href="_#= linkURL =#_">
                _#= linkURL =#_
            </a>
        </div>
```

In this code, we simply have a DIV wrapping all the content. We've applied a simple border and padding via inline CSS. Next, we'll display the `line1` variable using the following syntax: `_#= line1 =#_`. You can use JavaScript variables in your display template by using this syntax. Next, we'll show the link to the PDF file by using an `<a>` element. The complete file is shown in Figure 6-29.

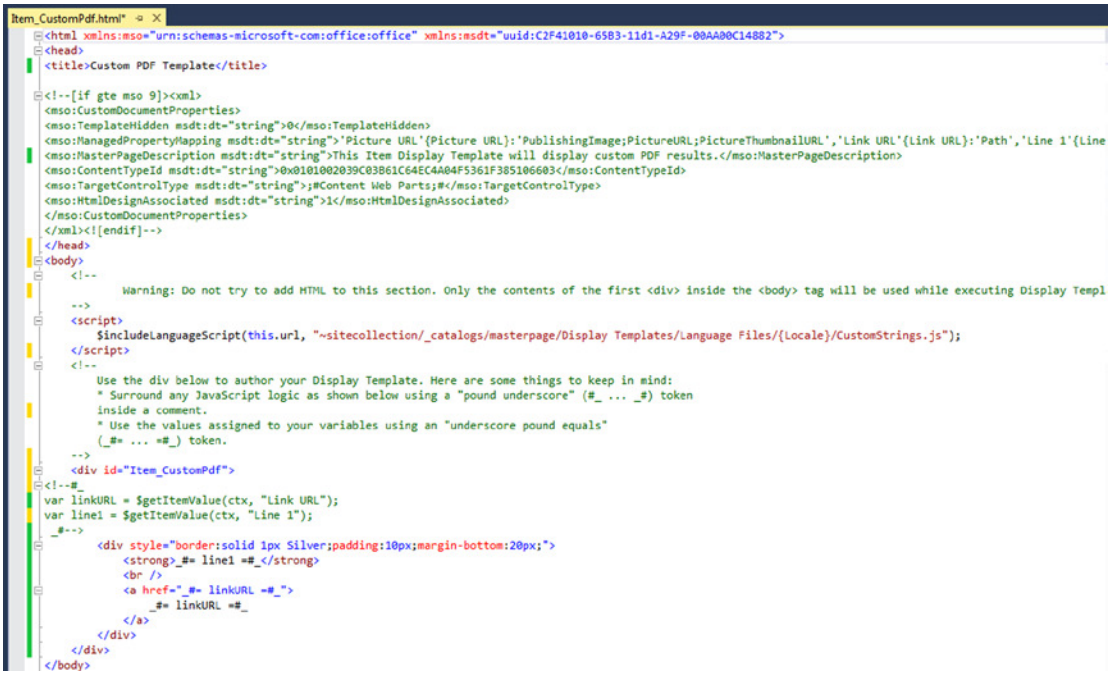

Figure 6-29. *Display template in Visual Studio*

Navigate back to the Content Web Parts folder in the Master Page Gallery, and from the Ribbon upload the `Item_CustomPdf.html` file to the library. Once the file has been uploaded, you will be prompted with a modal window in which you can set properties for the file, as shown in Figure 6-30.

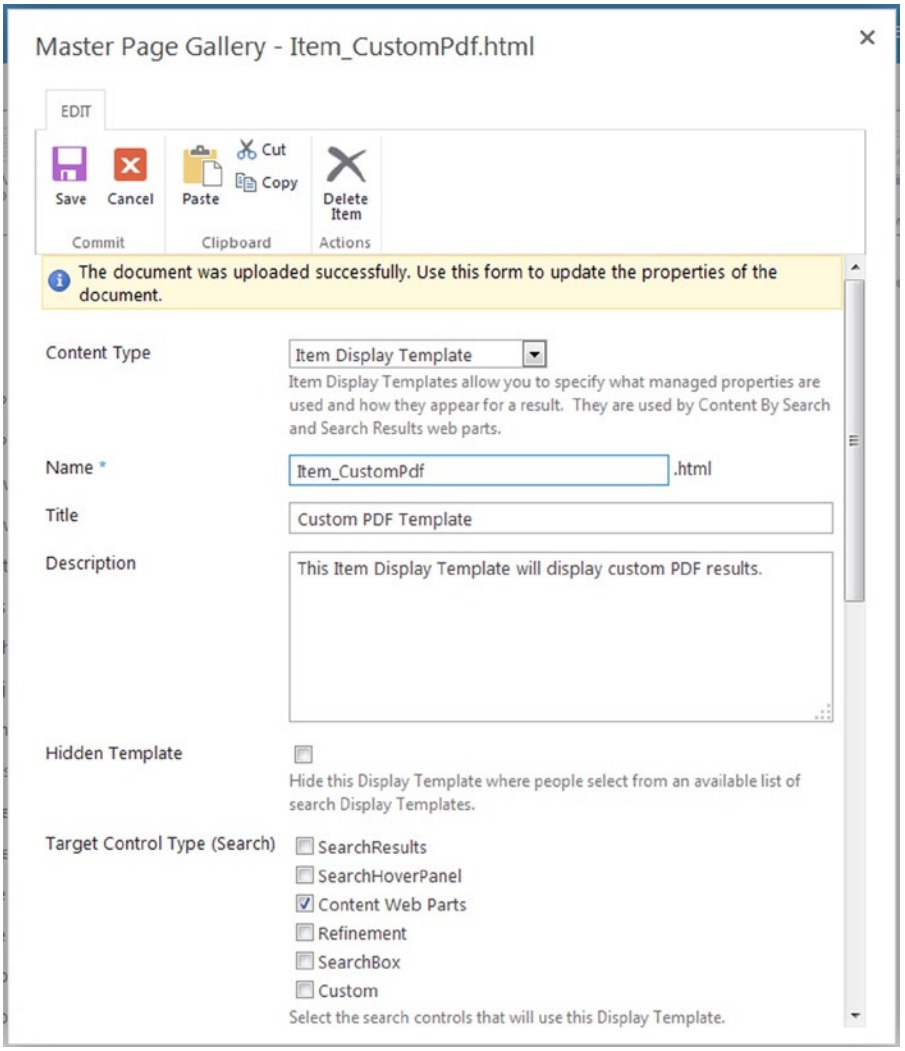

Figure 6-30. *Properties dialog for the display template*

This dialog has many of the properties that you saw in the HTML file for the display template. All the properties that are in the file are pulled and populated in this window for you. As such, you can pretty much just leave all the properties as is and click Save. You'll see that the display template has been uploaded to the library, as shown in Figure 6-31.

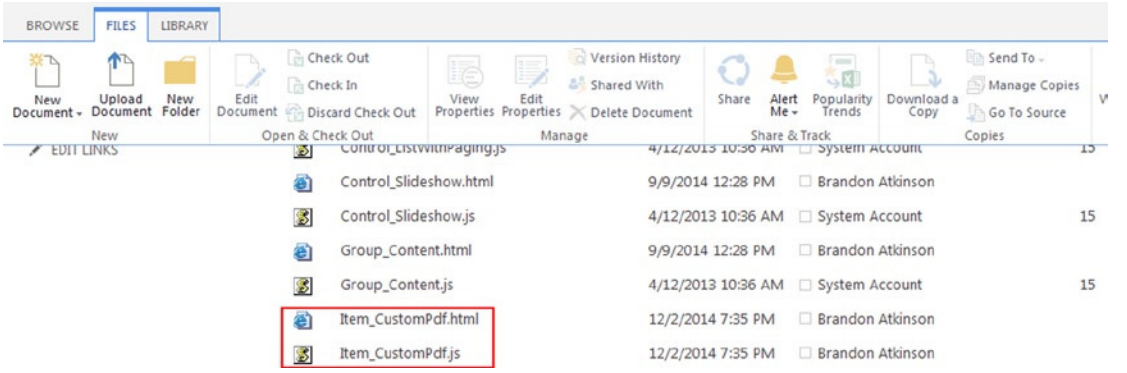

Figure 6-31. *The newly uploaded display template and generated JavaScript file*

You will also see that a companion JavaScript file has been created with the same name. SharePoint generates this file based on the HTML file that you upload. This is what it uses when it renders the content. You don't need to worry about this file and you should never try to modify it. If you need to make changes, simply change the HTML file and upload it again. A new JavaScript file is generated each time you upload a new copy of the HTML.

Navigate back to the page with the Content Search Web Part and place the web part into Edit mode. If you click the Item drop-down in the Display Templates section, you will now see the Custom PDF Template as an option, as shown in Figure 6-32. Choose this option, click OK, and then save the page.

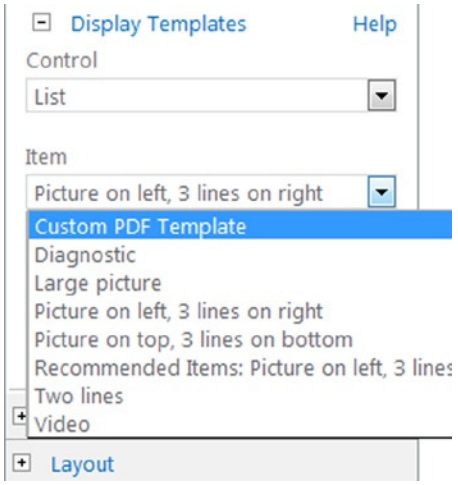

Figure 6-32. *The new display template is available for use*

The page reloads and the new display template is rendered, as shown in Figure 6-33.

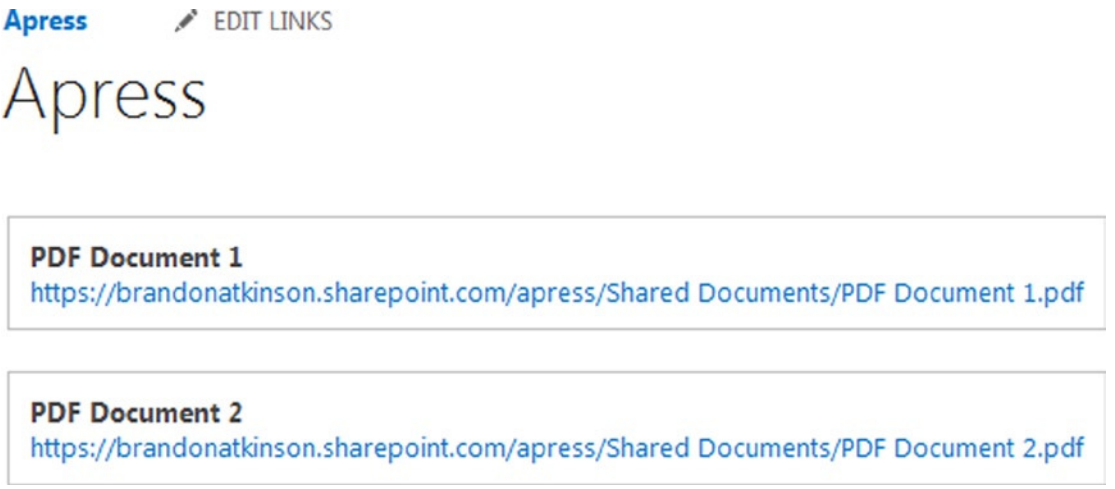

Figure 6-33. *The new display template being used to render the Content Search Web Part*

This example is about as basic as you can get, but hopefully it demonstrates the ease and power of display templates and search web parts. Using these templates, you can build custom search solutions that utilize managed properties, JavaScript, and CSS. All the tools you need to create a rich experience are available.

■ **Note** Spend some time downloading and exploring the out-of-the-box display templates that come with SharePoint. You can find a lot of useful information by simply browsing these files and seeing the code used in each one.

Summary

In this chapter, we explored how to build custom search solutions using the Content Search Web Part, custom result sources, and custom display templates. You learned about result sources and saw how to build a new source and use it with the Content Search Web Part. We quickly explored display templates and created a simple example of how to apply custom rendering. Search will always play a big role in SharePoint and now you have some tools to build custom search solutions for your business needs.

In the next chapter, we'll continue to explore what SharePoint has to offer when crafting custom solutions. We'll look at some of the built-in JavaScript functionality that is at your fingertips when writing your code. We've already seen some of this functionality, and we'll dive much deeper into what's possible.

CHAPTER 7

Working with SharePoint's Built-in JavaScript Functions and Properties

When building custom solutions using JavaScript, very often you need help from other libraries to accomplish your efforts. We've already seen the power of the jQuery library and how it can help locate HTML elements on our page, update their values, and even make AJAX calls for data. When working with SharePoint, you often need to build URLs, show notifications to users, and more. Luckily, Microsoft provides an enormous amount of JavaScript to assist you with these tasks. I'm not simply talking about the JavaScript Object Model or the REST API. In this chapter, we'll explore some of the built-in JavaScript functions and properties that are readily available to help you build your custom solutions.

Exploring the JavaScript

SharePoint comes with a long list of built-in JavaScript methods and properties that you can utilize in your code. In fact, the list is so long, it seems quite overwhelming once you dive into it. However, you'll quickly get used to a handful of them in your day-to-day coding efforts and you can feel at ease knowing there is much more help when it's needed. Let's first explore some of the files using Internet Explorer 11's developer tools. As mentioned earlier in this book, each browser has its own set of developer tools, and each one is slightly different from the other. You can absolutely perform the same tasks in each, but IE11 (and Chrome) show a handy list of available functions once you start typing in the JavaScript console, as shown in Figure 7-1.

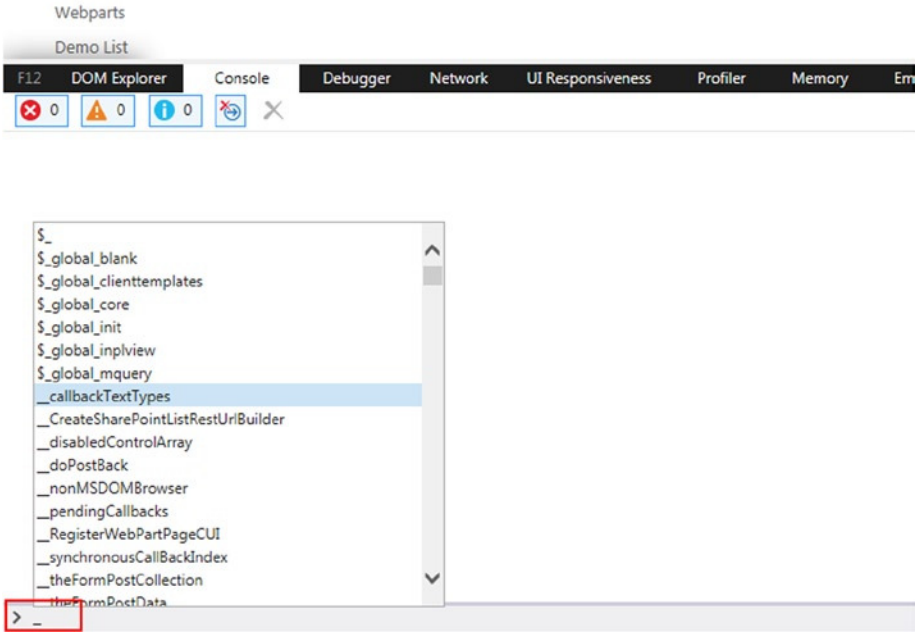

Figure 7-1. *Developer tools in IE11. Typing an underscore in the console window will display a list of available JavaScript functions*

Open the developer tools in IE11 by pressing F12 on your keyboard. You can also access them via the Settings button in the browser. Click the Console tab, and in the textbox at the bottom, type an underscore. A list will open displaying all the JavaScript functions that begin with an underscore. You can use the arrow keys to move up and down through the list.

Take a few minutes to look through everything you see here. It should give you a nice indication of what is possible using JavaScript with SharePoint. You've already seen some of these functions in action; for instance, when using JSLink we utilized the `SPClientTemplates` functionality to override the web part rendering.

■ **Note** The JavaScript methods and properties available to you are extensive. So much so that there is no way to properly cover it in a single chapter; it would take an entire book. We'll look at some of the functionality and use it in our example code. You should take some time to fully explore the functions in developer tools and use Google or Bing to track down more information.

_spBodyOnLoadFunctions

The `_spBodyOnLoadFunctions` is an array that holds a collection of JavaScript functions that should be fired when the page loads. We saw this in Chapter 5 when we needed to wire up a function to fire on page load. The array contains a method named `push`, where you can pass in the name of a function that should be fired:

```
_spBodyOnLoadFunctions.push(<FUNCTION_NAME>);
```

This is an extremely useful function when you do not have access to the jQuery `Document.ready()` function. There is also an equivalent function called `ExecuteOrDelayUntilBodyLoaded`, which performs the same task and even has the same signature:

```
ExecuteOrDelayUntilBodyLoaded(<FUNCTION_NAME>);
```

_spPageContextInfo

When writing any kind of code, whether JavaScript or something like C#, there are always times when you need to know things about the page, location, or some other environment variable. The `_spPageContextInfo` object can provide a lot of useful information with ease. In most of the examples in this book, it's used to provide location information in order to build URLs to request data. At first, this may not seem like terribly important information, but when making REST API calls, or targeting another site via the JavaScript Object Model, it becomes extremely important.

■ **Note** All the properties on the `_spPageContextInfo` object can be accessed easily in Firebug or in developer tools. Simply open the JavaScript console window and type in each one to see the results for your environment.

_spPageContextInfo.siteAbsoluteUrl

This property provides the absolute URL for the current site collection that the user is in. An absolute URL is a fully qualified URL, which contains the protocol (`http://` or `https://`) and the domain (`www.yoursite.com`). Since SharePoint can utilize Alternate Access Mappings, which allows you to utilize different domains for the same content, this function provides a very useful way to build URLs without worrying about where the user is accessing the page from.

Since this is the path to the site collection, the URL produced could look anything like the following:

- `http://www.<DOMAIN>.com`
- `http://www.<DOMAIN>.com/sites/<SITE_NAME>`

_spPageContextInfo.siteServerRelativeUrl

This property provides the server-relative URL for the current site collection that the user is in. A relative URL simply omits the protocol and domain from the URL. This is a handy tool, as there will be many times when you need the relative URL for the site collection. As mentioned, since SharePoint content has the ability to be accessed on many different domains, targeting the relative URL allows you to publish the same code across multiple domains without worrying where the code will be run.

If you were to call this property in the root site collection, for instance, the resulting URL would simply be `/`.

_spPageContextInfo.webAbsoluteUrl

This property provides the absolute URL for the current web, or site, that the user is in. This is the fully qualified URL that includes the protocol and domain. The URL-produced code looks like the following:

- `http://www.<DOMAIN>.com`
- `http://www.<DOMAIN>.com/<SUBSITE>`
- `http://www.<DOMAIN>.com/sites/<SITE_COLLECTION>/<SUBSITE>`

_spPageContextInfo.webServerRelativeUrl

This property provides the server-relative URL for the current web, or site, that the user is in. As with the site-relative URL, it omits the protocol and domain from the URL. The URL-produced code looks like the following:

- `/`
- `/<SUBSITE>`
- `/sites/<SITE_COLLECTION>/<SUBSITE>`

ExecuteOrDelayUntilScriptLoaded

This function allows you to specify a JavaScript function to delay until another script has been completely loaded. In essence, we're telling SharePoint: "I want to run this particular function, but I need to wait until this other script has been loaded." It has the following signature:

```
ExecuteOrDelayUntilScriptLoaded(<FUNCTION_NAME>, <SCRIPT_FILE>);
```

SharePoint first checks to see if the specified JavaScript file is loaded; if so, it immediately executes the specified function. If the file has not been loaded, it places the function call into a queue and executes it later. We'll use this in the next chapter when querying user data using the JavaScript Object Model:

```
$(function () {
    ExecuteOrDelayUntilScriptLoaded(getUserProfileProperties, "sp.userprofiles.js");
});

function getUserProfileProperties() {
    var clientContext = new SP.ClientContext();
    ...
```

As you can see in this code snippet, we have a jQuery `Document.ready()` function where we have placed the `ExecuteOrDelayUntilScriptLoaded` function. We pass in the function `getUserProfileProperties` and the JavaScript file `sp.userprofiles.js`. We need to implement this function here, as the code contained in our function can only operate if the `sp.userprofiles.js` file is loaded. SharePoint provides an easy way to ensure functionality in your code!

■ **Note** The `ExecuteOrDelayUntilScriptLoaded` functionality is a carryover from SharePoint 2010, where it was located under the `SP.SOD` object. There are scenarios in SharePoint 2013 where the newer version will not work properly, usually on Publishing Portal pages. If you find that this function is not working, you can revert back to the 2010 implementation. More information can be found at `http://msdn.microsoft.com/en-us/library/office/ff408081%28v=office.14%29.aspx`.

SP.UI.Notify

The `SP.UI.Notify` object provides a number of ways to notify the user that something is happening. This functionality provides a way to tap into the out-of-the-box SharePoint notifications and allows your custom solutions to blend seamlessly into the environment. While the examples in this book do not utilize this notification framework, they could easily be refactored to include notifications when a Create or Delete operation is taking place. It's always a good practice to notify your users of events taking place when you can.

SP.UI.Notify.showLoadingNotification

The `SP.UI.Notify.showLoadingNotification` function is a quick and easy way to display the familiar "Working on it..." notification. If you've used SharePoint for any amount of time, you've surely seen one of these messages, which pop out from the right side of the screen. It includes a "spinner" icon, as shown in Figure 7-2.

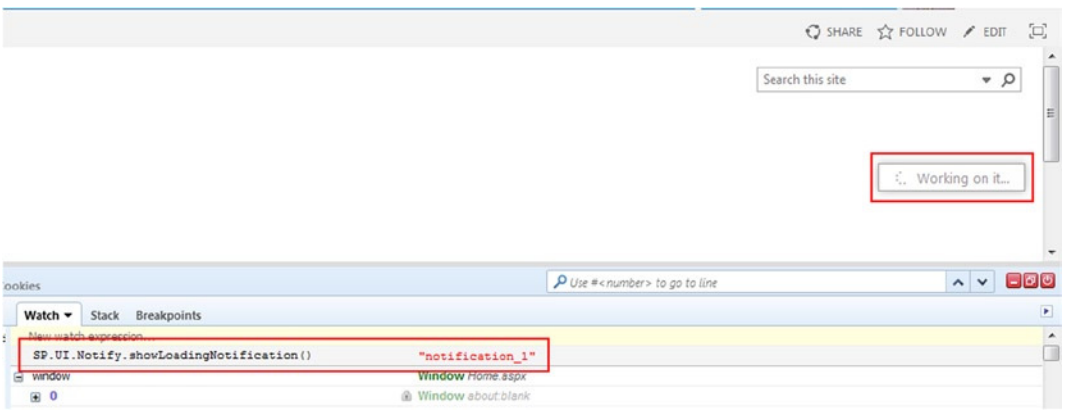

Figure 7-2. *Loading notification launched via Firebug*

You can launch this notification at any time in your code with the following line:

```
SP.UI.Notify.showLoadingNotification();
```

In fact, you can also launch Firebug or developer tools and invoke it directly in the browser. Figure 7-2 shows the notification being triggered via Firebug from the console.

This is a quick and handy way to notify the user that something is happening via your code. While this generic notification can certainly suffice in some situations, you may need to provide more explanation about what is going on. For that you can use the `SP.UI.Notify.addNotification` method.

SP.UI.Notify.addNotification

Much like the previous function, `SP.UI.Notify.addNotification` is used to display a notification to the user in the same location, but with a custom message. This is perfect when you need to notify the user of an action taking place, but need to be more descriptive than a simple "Working on it..." message. Figure 7-3 shows a custom message being displayed.

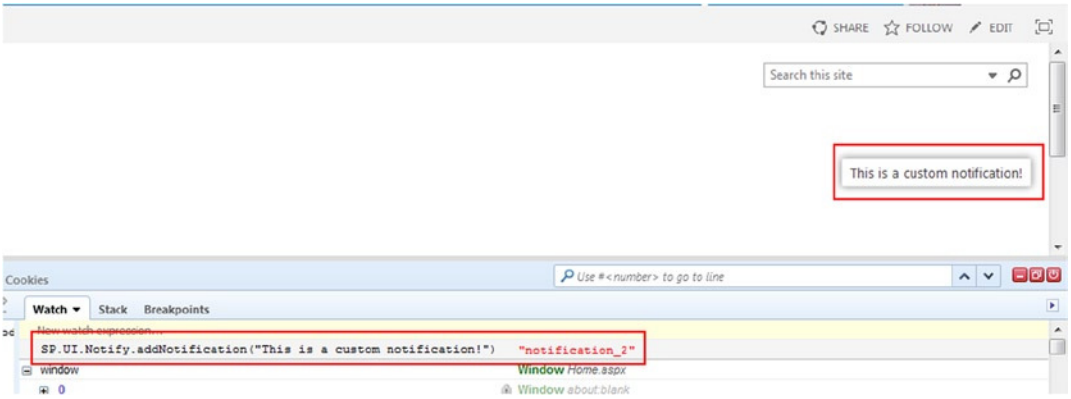

Figure 7-3. *A custom message being displayed to notify the user*

You can launch this notification with the following line:

```
SP.UI.Notify.addNotification("Your Custom Message");
```

As you can see, adding a custom notification couldn't be easier via JavaScript. There is no maximum length for the message you display. However, as the message gets longer, it eventually hits the other side of the screen and the text begins to wrap. This may not always layout correctly, so it's best to keep your messages short. Figure 7-4 shows a longer message being displayed.

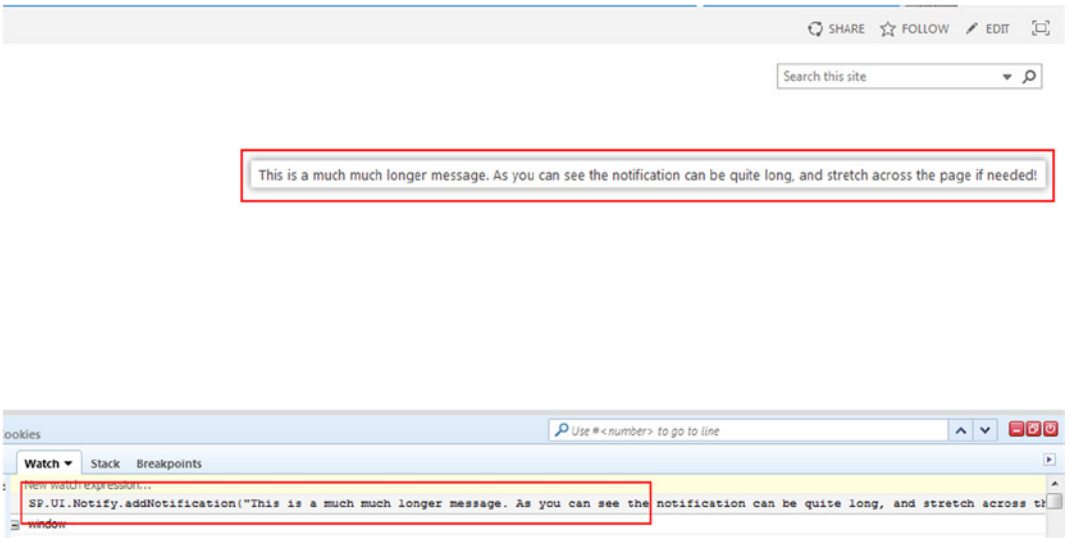

Figure 7-4. *A longer message can be displayed via a notification*

Notifications only remain on the page for about 6 seconds. If you just need to notify the user that an action took place but you do not need to wait for a response, this works just fine. However, sometimes you need to kick off an action and wait for a result. In this case, you may need your notification to stick around a little longer. You can include a Boolean to indicate that the notification should remain by using the following line:

```
SP.UI.Notify.addNotification("Your Custom Message", true);
```

By using the preceding line, the notification will remain until it's manually closed. You can do this by calling the `SP.UI.Notify.removeNotification` function, passing in the Notification ID from the `addNotification` function call. When you call the `addNotification` function, you get back the ID of the notification that was just created, as shown in Figure 7-5.

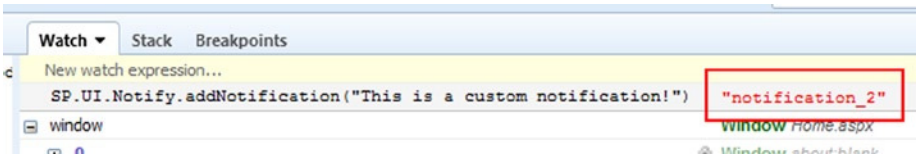

Figure 7-5. *The Notification ID that is returned when you create a new notification*

If we were to use this in code, the line would look like the following:

```
var notifyId = SP.UI.Notify.addNotification("Hello World", true);
```

When we execute this line, the `notifyId` variable now has the ID of the newly created notification. We could later close the notification by using the following line:

```
SP.UI.Notify.removeNotification(notifyId);
```

The `SP.UI.Notify.removeNotification` simply takes the ID of the notification and removes it from the screen. You can also use the Boolean flag when calling the `showLoadingNotification` function:

```
SP.UI.Notify.showLoadingNotification(true);
```

Passing in the Boolean triggers the same return of a notification ID, as shown in Figure 7-6.

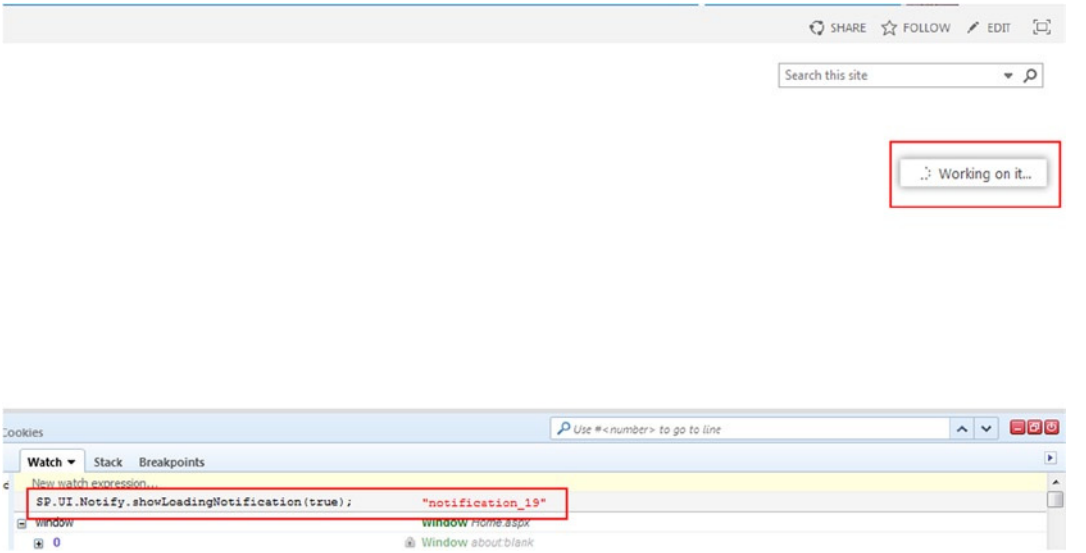

Figure 7-6. *A Notification ID is returned when using the Boolean flag*

These functions provide an easy way to notify your user of what is going on via JavaScript. If you need to perform a long-running AJAX call, for instance, you can let the user know that this is taking place so that they do not lose patience with your web part.

Summary

Microsoft has gone a long way to not only provide developers with ways to access data via JavaScript, but to also tap into core SharePoint functionality. In this chapter, we explored a small cross section of the built-in JavaScript functions and properties that are available to you. We saw how to access location data to more easily build URLs for REST API calls and hyperlinks. We saw how to load function calls into the body onload event inside SharePoint, as well as how to ensure that functionality is available to our code before it is executed. Finally, we looked at some of the various ways you can notify your users of what your code is doing to provide a better user experience.

CHAPTER 8

Working with the JavaScript Object Model

So far, you've been introduced to several different ways to create custom solutions in SharePoint using HTML and JavaScript. In this chapter, we'll continue to explore using the Etherson method to create custom web parts along with the JavaScript Object Model. We looked briefly at the JavaScript Object Model, or JSOM, in Chapter 2. In this chapter, we'll look at how to work with lists, libraries, files, sites, and social data using JSOM. You'll be introduced to a variety of ways of working with SharePoint data and begin to fully see the power of this technology.

JSOM has the power to provide you with a wide range of options for your development needs. You can create new lists and sites; you can read files and interact with social data. By the end of this chapter, you'll have a solid grasp on how to write JSOM code and know what it is capable of doing.

Note Code that utilizes the JavaScript Object Model runs under the context of the user who is logged in. This means that the code only has the ability to perform actions that the user's permission level will allow. For instance, if you have code to create a new subsite in the site collection, but the user who is logged in does not have that permission, it will run but produce an error. Keep this in mind as you build your solutions and ensure that you do not show users options that they cannot utilize.

Getting Set Up

Before getting started, we need to add some additional data to the Demo List we've been using throughout this book. Adding some additional data will allow us to query for more data and apply filters. Navigate to the Demo List and add a few more list items with various statuses. The updated list is shown in Figure 8-1.

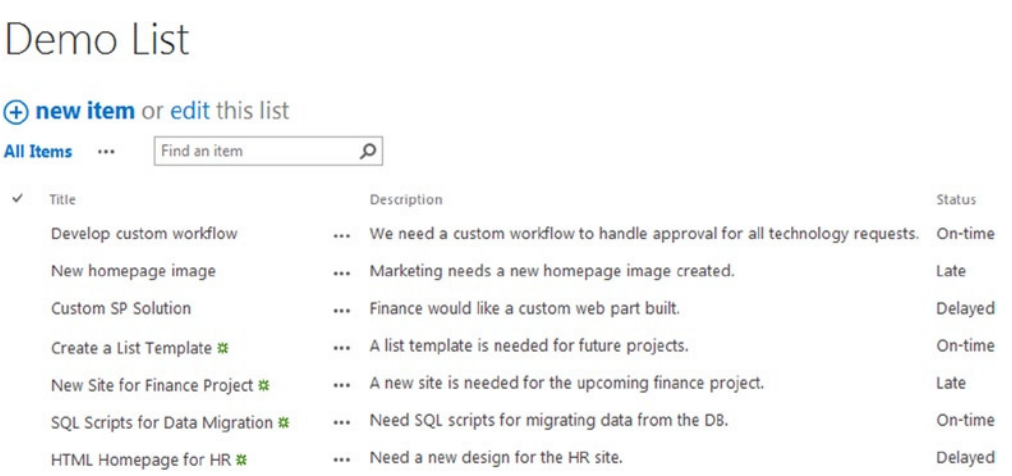

Figure 8-1. *Demo List with more items added to it*

Now that we have some additional data in the list, let's begin to look at using JSOM to query for data. The examples in this chapter will utilize the Etherson method introduced in Chapter 4. Navigate to the Webparts folder in your site and create a new folder named **Chapter 8**, as shown in Figure 8-2.

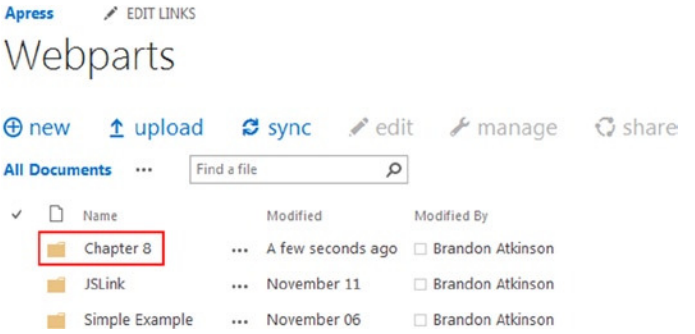

Figure 8-2. *Chapter 8 folder in the Webparts document library*

Each example in this chapter will follow the same pattern. We'll look at the HTML file rendered in the Content Editor Web Part, the JavaScript file used to retrieve and render data, and finally, the results. Along the way, we'll dive into each file and explain what is happening.

Working with Lists

Lists are everywhere in SharePoint! The large majority of things you will interact with in SharePoint as a user is a list. The following examples will demonstrate some various ways to interact with lists via code.

Get List Data

In this example, we'll simply retrieve some list data to show the user.

HTML

Create a new HTML file in Visual Studio and name it **JSOMGetListData.html**. The HTML for this example is very simple, and only consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/webparts/chapter 8/JSOMGetListData.js"></script>

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

First, you'll notice that we have to load a couple of SharePoint JavaScript files: `sp.runtime.js` and `sp.js`. These two files are very important because they provide the necessary code needed to properly run JSOM code. In SharePoint 2010, all the JavaScript needed was always loaded on every page. This made JavaScript development easier because you never had to worry about whether core SharePoint scripts were loaded or not. In SharePoint 2013, only the bare-bones code is loaded for what is needed on the page. This helps SharePoint load pages much faster; but it also means that if you need a specific library, you need to ensure that it's loaded. In our case, we'll need both of these files.

Next, we load in our custom JavaScript file called `JSOMGetListData`. This file contains all the code to load data into the DIV on the page, which has an ID of `divGetListData`. All in all, there is not much on this page.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMGetListData.js**. The JavaScript file is more complex in comparison and consists of the following lines:

```
$(function () {
    getListData();
});

function getListData() {
    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();
    this.collList = oWebsite.get_lists();

    clientContext.load(collList);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    var listInfo = '';
    var listEnumerator = collList.getEnumerator();
```

```
    while (listEnumerator.moveNext()) {
        var oList = listEnumerator.get_current();
        listInfo += 'Title: ' + oList.get_title() + ' - Created: ' +
            oList.get_created().toString() + '<br />';
    }
    $("#divGetListData").html(listInfo);
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

Most of the JSOM examples in this book follow the same pattern as this example. Let's dive into each line in detail to fully understand what is going on. The script begins with a jQuery `Document.ready` method, which simply calls a custom function called `getListData` that makes the JSOM call and is where the substantial code resides.

First, we create a new variable called `clientContext` and set it using the following line:

```
var clientContext = new SP.ClientContext();
```

The client context is a SharePoint object, which is really the entry point for all JSOM calls. It holds a lot of information about the context of the current site, user, web, and so forth. In this example, we simply ask for the context to be returned without passing any parameters. This will automatically return the context for the current site. You could also ask for the context of other sites by passing in a URL; for instance, `SP.ClientContext('http://www.site.com')`.

Next, we create another variable called `oWebsite` and set it using the following line:

```
var oWebsite = clientContext.get_web();
```

This will give us the current site, or web, in the site collection that the user is executing the code from. We need this web in order to grab all the lists. Next, we create one more variable called `collList` by using the following line:

```
this.collList = oWebsite.get_lists();
```

This variable is populated with all the lists from the `oWebsite` variable. The `collList` variable is declared a little differently from the others, as there is no `var` keyword here, rather `this`. The reason for this is that the `this` keyword will make this variable global so that it can be accessed later by the functions needed to process the data.

The next line is often a source of headaches when you first start using JSOM:

```
clientContext.load(collList);
```

This line is "loading" the query into the client context to send to SharePoint. What this means is that at this point you have not actually sent any requests to SharePoint for data. In most JavaScript development, you would think that a line like `this.collList = oWebsite.get_lists();` would return all the lists

immediately and you could start using them. In JSOM, this is not the case. You have to load any requests you would like to make and then submit them later using the following line:

```
clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
```

Not until you call the `executeQueryAsync` function does a request to SharePoint actually take a place. This is very important to understand when using JSOM and it trips up a lot of developers when they are first starting out. The reason for having to load all your requests before submitting them is a feature called batch processing. Basically, SharePoint allows you to load multiple requests for data and then only make one execute call to get all the data you need. This allows for better performance on the server where your JSOM call is actually processed. None of the examples in this book actually utilizes batch processing, but it's an important concept to know about.

The `executeQueryAsync` method allows you to specify a `success` function and a `failure` function, which are called after SharePoint returns the results of the query. These are declared using the `Function.createDelegate` method, as shown earlier. Once you have specified these, you need to create the function to actually process the results.

```
function onQuerySucceeded() {
    var listInfo = '';
    var listEnumerator = collList.getEnumerator();

    while (listEnumerator.moveNext()) {
        var oList = listEnumerator.get_current();
        listInfo += 'Title: ' + oList.get_title() + ' - Created: ' +
            oList.get_created().toString() + '<br />';
    }
    $("#divGetListData").html(listInfo);
}
```

This function begins with a new variable called `listInfo`, which will be used to hold all the results of the query before displaying them on the page. Next, we get an enumerator from the `collList` variable using the `collList.getEnumerator()` method and we put it in a variable called `listEnumerator`. We need to loop through all the results that are returned, and the enumerator allows us to do this. Using a `while` loop, we loop through each item that was returned using the `listEnumerator.get_current()` method. During each loop, we grab the title and the created date from the current item. We include more HTML and add the content to the `listInfo` variable using `+=`. This allows us to keep adding content to the `listInfo` variable rather than overwriting it each time.

Finally, we output the contents of the `listInfo` variable to the DIV in our HTML page using `$("#divGetListData").html(listInfo);`.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The results will display all the lists in the current site, as shown in Figure 8-3.

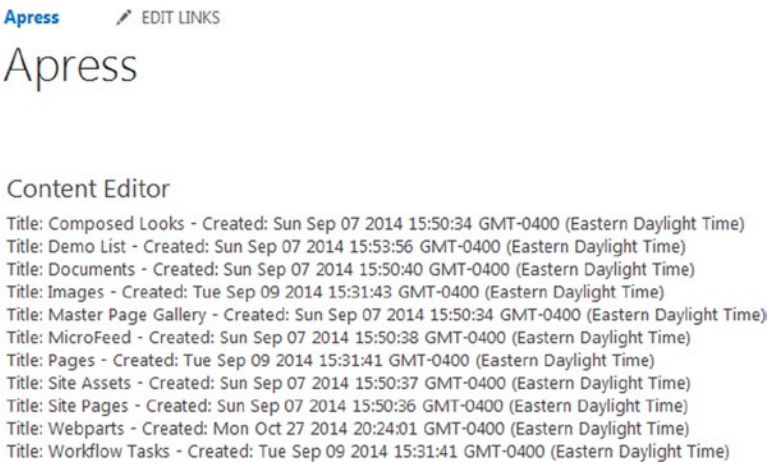

Figure 8-3. *Get List Data results in the Content Editor Web Part*

Create a New List

In this example, we'll create a new list. We won't do anything fancy, just create a new list and set its title.

HTML

Create a new HTML file in Visual Studio and name it **JSOMCreateList.html**. The HTML for this example is a little more complex than the previous example, and it consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/webparts/chapter 8/JSOMCreateList.js"></script>

<div>
    <strong>Enter a name for the list:</strong>
    <input type="text" id="txtListName" />
    <input type="button" id="btnSubmitListName" value="Submit" />
</div>
<div id="divCreateListResults"></div>
```

Just as before, we need to load in the `sp.runtime.js` and `sp.js` files, as well as a reference to our custom JavaScript file. The HTML is a little more complex, with several elements on the page. First, there is some simple text wrapped in a `<strong>` tag to make it bold on the page. Next, there is an INPUT of type Text that is just a simple textbox for the user to input the name of the list that they wish to create. There is a button for the user to submit the new list request and, finally, a DIV where we can output the results of the list creation. Each element has an ID so that we can easily target it with jQuery.

■ **Note** In order to create a new list via JSOM, the user executing the code must have at least Manage Lists permissions in SharePoint. The page will be rendered no matter the permission level; however, SharePoint will throw an error if the user does not have adequate permissions.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMCreateList.js**. In comparison, the JavaScript file is more complex, and it consists of the following lines:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        createList(listName);
    });
}

function createList(listName) {
    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();

    var listCreationInfo = new SP.ListCreationInformation();
    listCreationInfo.set_title(listName);
    listCreationInfo.set_templateType(SP.ListTemplateType.genericList);
    this.oList = oWebsite.get_lists().add(listCreationInfo);

    clientContext.load(oList);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    var results = oList.get_title() + ' successfully created!';
    $("#divCreateListResults").html(results);
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

In this example, we don't need to fire any actions on the page load; however, we do need to wire up a button click event when the page loads. In the `Document.ready` function, we fire the `bindButtonClick` function. In this function, we use the jQuery `on` operator to wire a click event on the `btnSubmitListName` button in the HTML. When the user clicks the button, we grab the text from the textbox on the page and pass it to the `createList()` function using the following lines:

```
function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        createList(listName);
    });
}
```

The `createList()` function takes a variable called `listName`, which is the text the user inputs on the page. Next, we grab a reference to the current client context and get the current web. Then, we create the list using the following code:

```
var listCreationInfo = new SP.ListCreationInformation();
listCreationInfo.set_title(listName);
listCreationInfo.set_templateType(SP.ListTemplateType.genericList);
this.oList = oWebsite.get_lists().add(listCreationInfo);
clientContext.load(oList);
```

In order to create a new list, we use an `SP.ListCreationInformation()` object. We can instantiate a new `ListCreationInformation` object using the following line:

```
var listCreationInfo = new SP.ListCreationInformation();
```

All this line does is create a new object and assign it a new variable called `listCreationInfo`. Next, we set the title and template using the following lines:

```
listCreationInfo.set_title(listName);
listCreationInfo.set_templateType(SP.ListTemplateType.genericList);
```

The `set_title` method takes the `listName` that was passed into the function. The `set_templateType` method takes a built-in value of `SP.ListTemplateType`, which, in this case, we pass in `genericList`. The available values here would be anything you would expect to see if creating the list in the browser using the SharePoint UI, IE: Announcements, Calendar, Task, and so forth. Next, we add the list to the web using the following line:

```
this.oList = oWebsite.get_lists().add(listCreationInfo);
clientContext.load(oList);
```

Just like in the previous example, after we add the list to the site, we then "load" the list to the client context. Finally, we call the `executeQueryAsync` function, which will actually make the call to SharePoint to create the list. Once the call is completed, we process the results with the following function:

```
function onQuerySucceeded() {
    var results = oList.get_title() + ' successfully created!';
    $("#divCreateListResults").html(results);
}
```

In this function, using the `oList.get_title()` method, we can get the title of the list that was just created. This is a nice check to ensure that the list was indeed created, since we get the title from the list itself rather than using the text that the user inputted. Once we have the list title, we use jQuery to add the success message to the `divCreateListResults` DIV on the page.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show the textbox and button for the user to input a new list name. Type **My Custom List** and then click the Submit button. The results will display the success message, as shown in Figure 8-4.

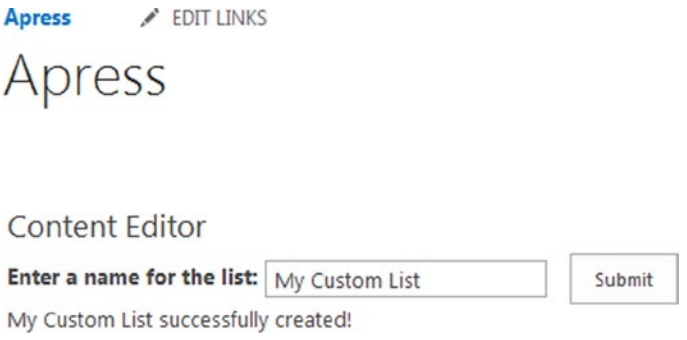

Figure 8-4. *Create new list code after it has been run*

Since this is all done using JavaScript and AJAX the list will get created and the user will be notified as expected. However, the left navigation on the page will not be updated since the page has not been refreshed. You could absolutely change this code to refresh the page on success, but for this example we'll simply refresh the page in the browser. Figure 8-5 shows the page after a refresh where the new list is displayed in the left navigation.

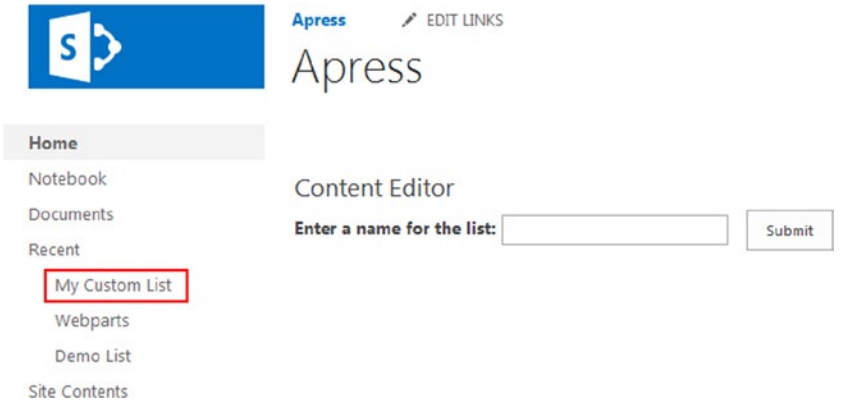

Figure 8-5. *The new list as displayed in the left navigation pane*

Delete a List

In this example, we'll simply delete the list we just created in the previous section.

HTML

Create a new HTML file in Visual Studio and name it **JSOMDeleteList.html**. The HTML for this example is almost identical to the previous example, and it consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/webparts/chapter 8/JSOMDeleteList.js"></script>

<div>
    <strong>Enter the name of the list to delete:</strong>
    <input type="text" id="txtListName" />
    <input type="button" id="btnSubmitListName" value="Submit" />
</div>
<div id="divDeleteListResults"></div>
```

The only difference here is that we updated the text on the page to indicate that this is web part will delete a list, and we changed the ID of the results DIV to `divDeleteListResults`.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMDeleteList.js**. The JavaScript file consists of the following lines:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        deleteList(listName);
    });
}

function deleteList(listName) {
    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();

    this.oList = oWebsite.get_lists().getByTitle(listName);
    oList.deleteObject();

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}
```

```
function onQuerySucceeded() {
    $("#divDeleteListResults").html("List successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

The script starts with the familiar jQuery `Document.ready` function that binds a button click to a function:

```
function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        deleteList(listName);
    });
}
```

This function is nearly identical to the previous example—with the exception of `deleteList`, the new function name to delete the list. This function is smaller and more to the point, with only a couple of relevant lines:

```
this.oList = oWebsite.get_lists().getByTitle(listName);
oList.deleteObject();
```

After obtaining a reference to the client context and current web, we use the `getByTitle()`method on the web object to locate the list that we wish to delete, and then assign it to a variable called `oList`. Next, we call the `oList.deleteObject()` method to actually perform the delete. You'll notice that there is no "load" method here on the client context object. When deleting a list, you do not need to perform this step. After the `deleteObject()`line, you can simply call the `executeQueryAsync()`method to perform the delete operation.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show the textbox and button for the user to input a list name to delete. Type **My Custom List** and then click the Submit button. The results will display the success message, as shown in Figure 8-6.

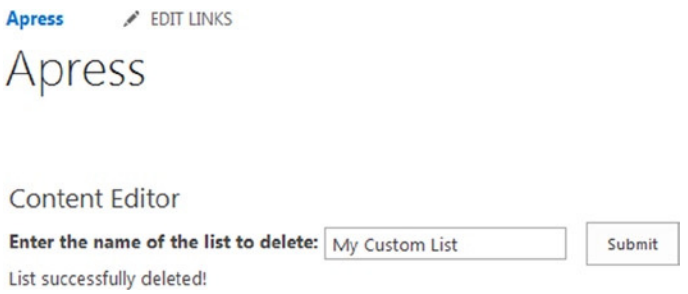

Figure 8-6. *Delete List code after it has successfully run*

Since this is all done using JavaScript and AJAX, the list will get deleted and the user will be notified as expected. However, the left navigation on the page will not be updated because the page has not been refreshed. As before, we could update this code to refresh the page on success, but for this example, we'll simply refresh the page in the browser. Figure 8-7 shows the page after a refresh where the list is no longer displayed in the left navigation.

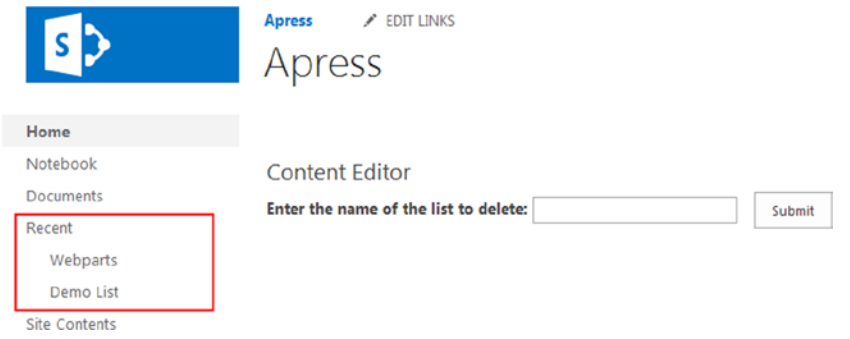

Figure 8-7. *The deleted list has been removed from the left navigation*

Working with List Items

The only thing more plentiful in SharePoint than lists, are list items. You'll find that a lot of your interactions in SharePoint are with list items, and thus a lot of your code will involve them as well. In this section we'll explore some of the ways to work with list items via code.

Get List Items

In this example, we'll simply retrieve some list items and display them to the user.

HTML

Create a new HTML file in Visual Studio and name it **JSOMGetListItems.html**. The HTML for this example continues to follow the minimalist pattern of the previous examples, and it consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/webparts/chapter 8/JSOMGetListItems.js"></script>

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

For this example, we'll simply grab all the list items from the Demo List and display them in the `divListItems` DIV on the page.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMGetListItems.js**. The script to retrieve list items should look very familiar to the other scripts that we've seen so far, and it consists of the following lines:

```
$(function () {
    retrieveListItems();
});

function retrieveListItems() {
    var clientContext = new SP.ClientContext();
    var oList = clientContext.get_web().get_lists().getByTitle('Demo List');

    var camlQuery = new SP.CamlQuery();
    this.collListItem = oList.getItems(camlQuery);

    clientContext.load(collListItem);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded(sender, args) {
    var listItemInfo = '';
    var listItemEnumerator = collListItem.getEnumerator();

    while (listItemEnumerator.moveNext()) {
        var oListItem = listItemEnumerator.get_current();
        listItemInfo += '<strong>ID: </strong> ' + oListItem.get_id() +
            ' <strong>Title:</strong> ' + oListItem.get_item('Title') +
            '<br />';
    }

    $("#divListItems").html(listItemInfo);
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

For this example, since we'll just be fetching all the items from the list, we'll call the `retrieveListItems()`function using the jQuery `Document.ready` function. After obtaining a reference to the client context, we get a reference to the Demo List with the following line:

```
var oList = clientContext.get_web().get_lists().getByTitle('Demo List');
```

This line may at first glance appear to be intimidating because it's quite long. However, it's actually quite easy to read. First, we get the current web, and then we get all the lists in the web, and, finally, we get the list by its title. JSOM and REST both follow this type of pattern when querying for objects, making the code easy to understand. Next, we need to create a CAML Query to let SharePoint know which items we are looking for:

```
var camlQuery = new SP.CamlQuery();
this.collListItem = oList.getItems(camlQuery);
```

CAML (Collaborative Application Markup Language) Query is a query language used in SharePoint to define the query for list items. You can specify which items to return based on criteria much like a SQL query, although they look nothing alike. In this example, we simply create an empty query using the new `SP.CamlQuery()` method. When you query a list, you must provide a CAML query, even if no parameters are defined. If you pass an empty query, as in this example, you are in effect asking for all items to be returned. In practice, this is not a great idea, since a lot of lists contain hundreds or thousands of items. After we've created the query, we pass it to the `oList.getItems()` method to perform the query. Just as in the other examples, we use a global variable called `collListItems` using the `this` keyword. Finally, we "load" the query and execute it.

■ **Note** CAML Query is quite a complex and deep topic. We'll only scratch the surface of CAML in this book; it's that large. We'll dig a bit deeper into CAML in the next section, but you can learn more about how to build these queries at http://msdn.microsoft.com/en-us/library/ms467521(v=office.15).aspx.

Once we retrieve the list items, we process them in the `onQuerySucceeded` function:

```
function onQuerySucceeded(sender, args) {
    var listItemInfo = '';
    var listItemEnumerator = collListItem.getEnumerator();

    while (listItemEnumerator.moveNext()) {
        var oListItem = listItemEnumerator.get_current();
        listItemInfo += '<strong>ID: </strong> ' + oListItem.get_id() +
            ' <strong>Title:</strong> ' + oListItem.get_item('Title') +
            '<br />';
    }

    $("#divListItems").html(listItemInfo);
}
```

This function starts off with declaring an empty variable called `listItemInfo`. This variable will hold the entire HTML that will be displayed on the page. Next, we get the enumerator for the `collListItem` variable, which you will remember is available on any collection of items that gets returned from SharePoint. We loop through the items using the `listItemEnumerator.moveNext()` method. Inside the loop, we grab each item using the `listItemEnumerator.get_current()` method. We then build some HTML for each item in the result set displaying the ID and Title, and stuff it in the `listItemInfo` variable. Finally, we set the HTML to the `divListItems` DIV on the page with jQuery.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show all the items in the Demo List, as shown in Figure 8-8.

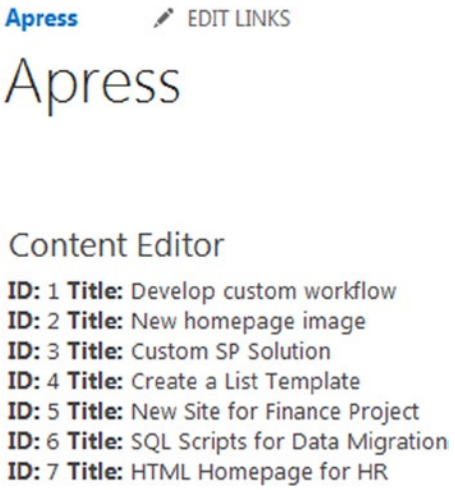

Figure 8-8. *List items from the Demo List displayed in the Content Editor Web Part*

Limit Results with CAML Query

As I mentioned, it's usually not a good idea to query for all items in a list. There are certainly times where this is needed or makes sense; for instance, you may have a list with very few items, or you have an administrator task where you need to view all the items. If you've worked with SharePoint in the past, you already know that SharePoint automatically limits list results, which default at 2,000 results in a query. In either case, most users will never need to see all the items in a list and the query will be limited to only show the items the user needs to see.

CAML allows you to limit the results based on a wide variety of parameters, which gives you complete control over what is displayed to the user. In the previous example, we passed in an empty CAML query by creating a new `SP.CamlQuery` object, which returned all the list items. You can pass in a CAML query by using the `set_viewXml()` method on the `CamlQuery` object, as seen in the following "JavaScript" section.

JavaScript

Update the `JSOMGetListItems.js` from the previous example and update the section where you declare the `SP.CamlQuery()` object with the following lines:

```
var camlQuery = new SP.CamlQuery();
    camlQuery.set_viewXml(
        '<View><Query><Where><Geq><FieldRef Name=\'ID\'/>' +
        '<Value Type=\'Number\'>5</Value></Geq></Where></Query>' +
        '<RowLimit>10</RowLimit></View>'
    );
    this.collListItem = oList.getItems(camlQuery);
```

In this example, we limit the results by specifying that we only want items with an ID greater than or equal to 5. This is done using the `GEQ` element, which stands for "greater than or equal to." Before the `GEQ` element, we use a `Where` element and specify the `FieldRef` of ID, which is the ID of the list item. If all this seems like Latin, don't worry, there are tools you can download to help you write CAML. We won't go deeply into the finer details of writing CAML queries; for this section, you should take away that you can fine-tune your queries with great detail.

■ **Note** At first glance, CAML may appear quite confusing and overly complex. After you have been working with CAML for a while, you may still be thinking the same thing! Luckily, there are several great free tools available to help you craft your CAML statements. I personally like SharePoint CAML Query Helper, which is free and can be downloaded on CodePlex. Visit `https://spcamlqueryhelper.codeplex.com` to download a copy.

Results

Update the JavaScript file and upload it, overwriting the previous version. Refresh the page with the Content Editor Web Part and you will see that the results now only show items with an ID greater than or equal to 5, as shown in Figure 8-9.

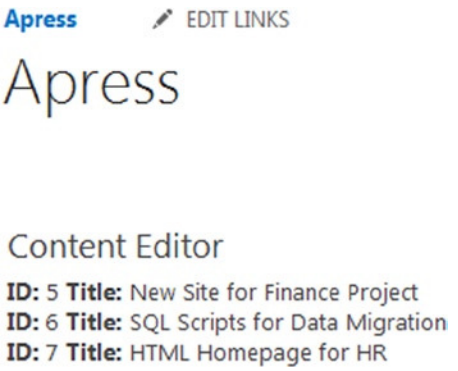

Figure 8-9. *The results of the Demo List query limited by a CAML query*

Add List Items

In this example, we'll add a new item to the demo list.

HTML

Create a new HTML file in Visual Studio and name it `JSOMAddListItems.html`. The markup for this page will allow a user to add an item to the Demo List, providing a Title and a Description. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMAddListItems.js"></script>
```

```
<div id="AddListData">
    <div>
        Title:
        <br />
        <input type="text" id="txtTitle" />
    </div>
    <div>
        Description:
        <br />
        <textarea cols="20" id="txtDesc"></textarea>
    </div>
    <br />
    <div>
        <input id="btnSubmit" type="button" value="Submit" />
    </div>
</div>
<div id="divResult"></div>
```

We provide an INPUT of type TEXT for the list item Title, and a TEXTAREA for the Description. Each element on the page has an ID so that we can easily target it with jQuery.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMADDListItems.js**. The script to add list items looks similar to the script for creating a new List. It consists of the following lines:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        addListItem();
    });
}

function addListItem() {
    var title = $("#txtTitle").val();
    var desc = $("#txtDesc").val();

    var clientContext = new SP.ClientContext();
    var oList = clientContext.get_web().get_lists().getByTitle('Demo List');

    var itemCreateInfo = new SP.ListItemCreationInformation();
    this.oListItem = oList.addItem(itemCreateInfo);
    oListItem.set_item('Title', title);
    oListItem.set_item('Description', desc);
    oListItem.set_item('Status', 'On-time');
    oListItem.update();
```

```
    clientContext.load(oListItem);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onAddSucceeded),
        Function.createDelegate(this, this.onAddFailed)
    );

}

function onAddSucceeded(sender, args) {
    $("#divResult").html("Item successfully added!");
}

function onAddFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

After we attach a button click event with jQuery, the first thing we do on the click event is grab the values that the user is providing with the following lines:

```
var title = $("#txtTitle").val();
var desc = $("#txtDesc").val();
```

Next, we get the reference to the current client context, as well as the Demo List. Then, in order to create a new list item, we have to instantiate an `SP.ListItemCreationInformation` object. This object allows you to create a new list item for a list and set its properties. This is done with the following code:

```
var itemCreateInfo = new SP.ListItemCreationInformation();
this.oListItem = oList.addItem(itemCreateInfo);
oListItem.set_item('Title', title);
oListItem.set_item('Description', desc);
oListItem.set_item('Status', 'On-time');
oListItem.update();
```

Once we create a new `SP.ListItemCreationInformation` object, we add it to the list using the `addItem()` method on the list object. Next, we set individual properties using the `set_item()` method on the list item object. This method takes two parameters. The first parameter is the field or column for which you want to set the value, and the second parameter is the value itself. For the Title and Description, we simply pass in the values we retrieved earlier. For the Status column, we simply hard-code a value of **On-time**. Finally, we update the item using the `update()` method, and then load it into the client context and execute the query. In the `success` function, we simply output to the screen that the list item creation was successful.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textboxes for the user to input the details, and will then display a success message when an item is added, as shown in Figure 8-10.

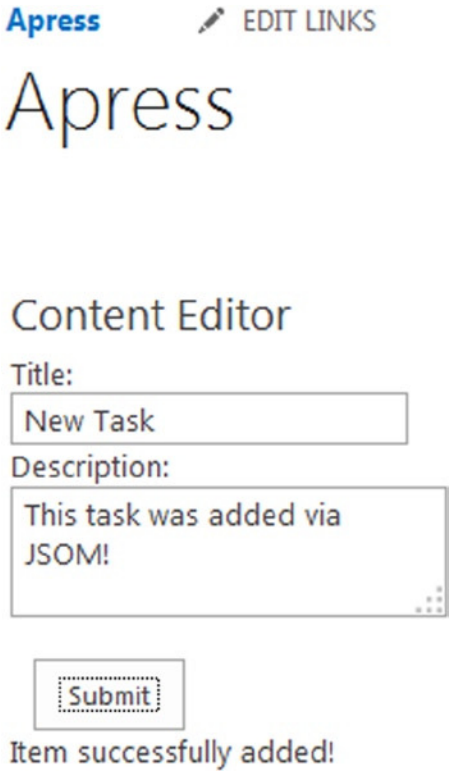

Figure 8-10. *Adding a new item to the Demo List*

We can then navigate to the Demo List to confirm that a new item was added. Figure 8-11 shows the Demo List with a new list item added and with the values supplied by the HTML page.

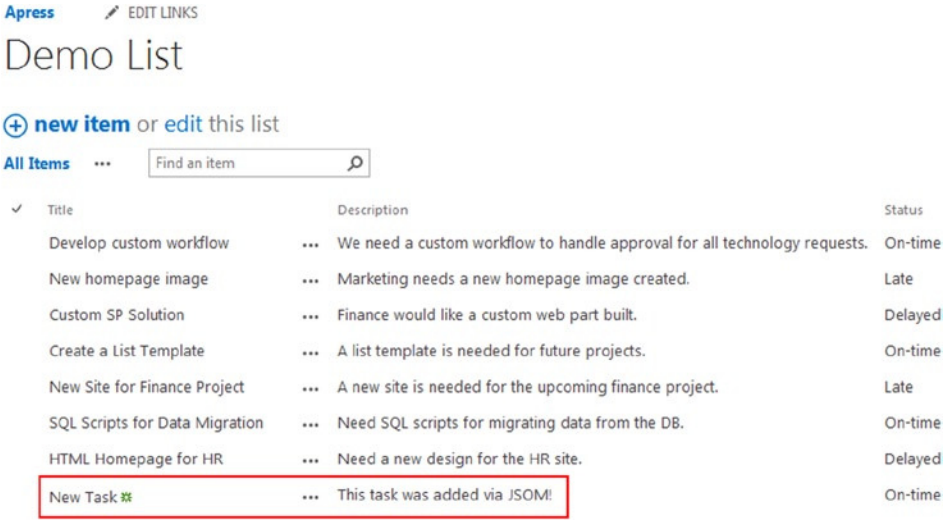

Figure 8-11. *A new list item added to the Demo List*

Delete List Items

This example will show you how to delete list items, in this case we'll delete the item we added in the previous example.

HTML

Create a new HTML file in Visual Studio and name it **JSOMDeleteListItems.html**. The HTML for this file is much smaller, as all we need to capture from the user is an ID corresponding to the list item that they wish to delete. This is accomplished with the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMDeleteListItems.js">
</script>

<div>
    Enter ID to Delete: <input type="text" id="txtId" />
</div>
<div>
    <input id="btnSubmit" type="button" value="Submit" />
</div>
<div id="divResult"></div>
```

This example is overly simplified, in that a user will most likely not know the ID of a certain list item to delete. A more real-world example would most likely show the user a list of items and allow them to click one to delete it. In such a scenario, you would have the list item IDs on the page and could easily access it. We'll keep a simple textbox to keep the example streamlined.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMDeleteListItems.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteListItem();
    });
}

function deleteListItem() {
    var id = $("#txtId").val();

    var clientContext = new SP.ClientContext();
    var oList = clientContext.get_web().get_lists().getByTitle('Demo List');
```

```
    this.oListItem = oList.getItemById(id);
    oListItem.deleteObject();

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );

}

function onQuerySucceeded(sender, args) {
    $("#divResult").html("Item successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

The `deleteListItem()` function will be called on the button click event on the page. Using the following line, the first thing that we do is grab the ID that the user entered on the page:

```
var id = $("#txtId").val();
```

Next, we get the current client context and a reference to the Demo List. Deleting the item is accomplished with the following lines:

```
this.oListItem = oList.getItemById(id);
oListItem.deleteObject();
```

We use the `getItemById()` method on the list object passing in the ID that the user submitted. Then, we call the `deleteObject()` method to actually perform the deletion. Similar to deleting a list earlier in the chapter, this is all that is required to delete the list item. We do not need to load the delete call into the client context; we simply need to execute the query. The `success` function will simply notify the user that the delete was successful.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textbox for the user to input the ID, and will then display a success message when an item is deleted, as shown in Figure 8-12.

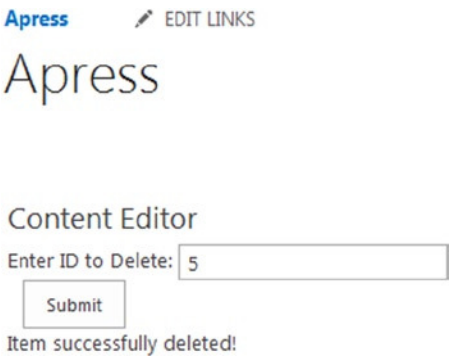

Figure 8-12. *Deleting a list item using JSOM*

Working with Document Libraries

Since one of SharePoint's strengths lies in document management, you need to know how to work with document libraries via code. Since document libraries are really just lists in SharePoint, you already have some experience here!

Create a New Folder

In this example, we'll demonstrate how to create a new folder in an existing document library.

HTML

Create a new HTML file in Visual Studio and name it **JSOMCreateFolder.html**. The markup for this page is quite simple, allowing the user to enter a name for the folder they wish to create. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMCreateFolder.js"></script>

<div>
    <strong>Enter a name for the folder:</strong>
    <input type="text" id="txtFolderName" />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMCreateFolder.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createFolder();
    });
}

function createFolder() {
    var folderName = $("#txtFolderName").val();

    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();
    var oList = oWebsite.get_lists().getByTitle("Documents");

    var folderCreateInfo = new SP.ListItemCreationInformation();
    folderCreateInfo.set_underlyingObjectType(SP.FileSystemObjectType.folder);
    folderCreateInfo.set_leafName(folderName);
    this.oListItem = oList.addItem(folderCreateInfo);
    this.oListItem.update();

    clientContext.load(oList);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    $("#divResults").html("Folder successfully created!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

After using jQuery to wire up the button click, the `createFolder()` function begins with grabbing the name of the folder that the user entered on the page:

```
var folderName = $("#txtFolderName").val();
```

Next, we get a reference to the current client context and the current web, and then we reference the Shared Documents folder with the following line:

```
var oList = oWebsite.get_lists().getByTitle("Documents");
```

This example will create a new folder in the Shared Documents library. Since all document libraries are really lists in SharePoint, we can use the `getLists().getByTitle()` method to retrieve the library. Next, we create a new `SP.ListItemCreationInformation` object for the folder:

```
var folderCreateInfo = new SP.ListItemCreationInformation();
```

At first glance, this may seem strange, as we are creating a new folder and not a list item. In SharePoint, folders are list items even though they look and act just like you would expect a folder to. For this reason, the same `SP.ListItemCreationInformation` object is used. Now that the object is created, we need to set some properties using the following lines:

```
folderCreateInfo.set_underlyingObjectType(SP.FileSystemObjectType.folder);
folderCreateInfo.set_leafName(folderName);
this.oListItem = oList.addItem(folderCreateInfo);
this.oListItem.update();
```

First, since this is not a standard list item, we need to specify the object type using the `set_underlyingObjectType()`method, passing in `SP.FileSystemObjectType.folder` as the parameter. Next, we set the folder name using the `set_leafName()` method, passing in the name that the user is submitting from the page. Finally, we add the new folder to the list and update it, and then load the query on the client context. The `success` function will simply output to the screen a message informing the user that the folder was successfully created.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textbox for the user to input the name of the folder to be created, and will then display a success message when completed, as shown in Figure 8-13.

Figure 8-13. *Creating a new folder with JSOM*

We can then navigate to the Documents library to visually verify that the new folder was created, as shown in Figure 8-14.

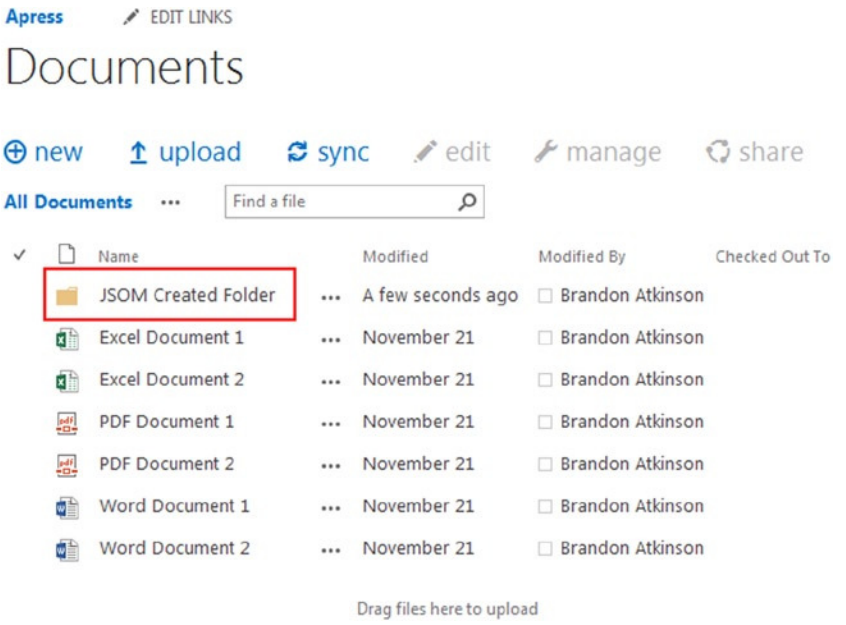

Figure 8-14. *New folder created via JSOM*

Working with Files

Lists and libraries are not the only areas where your coding skills will be used. In this section, we'll look at working with files via code.

Create a New File

In this example, we'll create a new text file and add it to a document library.

HTML

Create a new HTML file in Visual Studio and name it **JSOMCreateDocument.html**. The markup for this page will allow the user to enter a name for a new text file, as well as the content for the file itself. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMCreateDocument.js">
</script>
```

```
<div id="CreateFile">
    <div>
        <strong>Enter a title for the document:</strong>
        <br />
        <input type="text" id="txtDocumentTitle" />
    </div>
    <div>
        <strong>Enter content for the document:</strong>
        <br />
        <textarea cols="20" id="txtDocumentContent"></textarea>
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMCreateDocument.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createDocument();
    });
}

function createDocument() {
    var docTitle = $("#txtDocumentTitle").val() + ".txt";
    var docContent = $("#txtDocumentContent").val();

    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();
    var oList = oWebsite.get_lists().getByTitle("Documents");

    var fileCreateInfo = new SP.FileCreationInformation();
    fileCreateInfo.set_url(docTitle);
    fileCreateInfo.set_content(new SP.Base64EncodedByteArray());

    for (var i = 0; i < docContent.length; i++) {
        fileCreateInfo.get_content().append(docContent.charCodeAt(i));
    }

    this.newFile = oList.get_rootFolder().get_files().add(fileCreateInfo);

    clientContext.load(this.newFile);
```

```
    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    $("#divResults").html("Document successfully created!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

The first thing that we need to do is grab the document title and content that the user entered on the page. This is accomplished with the following lines:

```
var docTitle = $("#txtDocumentTitle").val() + ".txt";
var docContent = $("#txtDocumentContent").val();
```

Notice that we append `.txt` to the end of the document title. This will let SharePoint know that we intend to create a text file, as it will use the extension in the name to determine the file type. Next, we get a reference to the current client context, as well as the web and document library where we would like to save the new document. Now we need to create a new `SP.FileCreationInformation` object and set its properties:

```
var fileCreateInfo = new SP.FileCreationInformation();
fileCreateInfo.set_url(docTitle);
fileCreateInfo.set_content(new SP.Base64EncodedByteArray());
```

You should now begin to see a pattern with creating objects in JavaScript with SharePoint. Just about anything you create has a `CreationInformation` object associated with it. Once we have the object created, we set the URL of the file using the `set_url()`method. Finally, we set the content of the file to an empty byte array using the `SP.Base64EncodedByteArray()`object. You don't need to worry too much about the details of this object for now; just know that this is what the JSOM code expects when setting the content of a new file. Now that we have the empty array created, we need to populate it using the following code:

```
for (var i = 0; i < docContent.length; i++) {
        fileCreateInfo.get_content().append(docContent.charCodeAt(i));
}
```

This is a simple loop that will take each character in the text that the user submitted and append it to the empty array in the file content. Once the loop is complete and the text has been entered into the document, we add it to the library:

```
this.newFile = oList.get_rootFolder().get_files().add(fileCreateInfo);
```

We then load the query on the client context and execute it. The `success` function will simply output to the screen that the file creation was successful.

■ **Note** In this example, we are creating a simple text file. In most real-world scenarios, a text file would not be the intended result, but perhaps a Word file or an Excel file instead. You could absolutely change this example to generate a Word document by simply changing the `.txt` to `.docx`. This simple change would, in fact, create a new Word document with the content in it. However, it could only be opened in Word on the client machine, not Office Web Apps. Luckily, there are Office JavaScript files available via Microsoft to allow you to create and manipulate Office files from the browser. You can find more information at `http://msdn.microsoft.com/en-us/library/office/fp160953%28v=office.15%29.aspx`.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show textboxes for the user to input the title of the document and the content, and will then display a success message when the new document has been created, as shown in Figure 8-15.

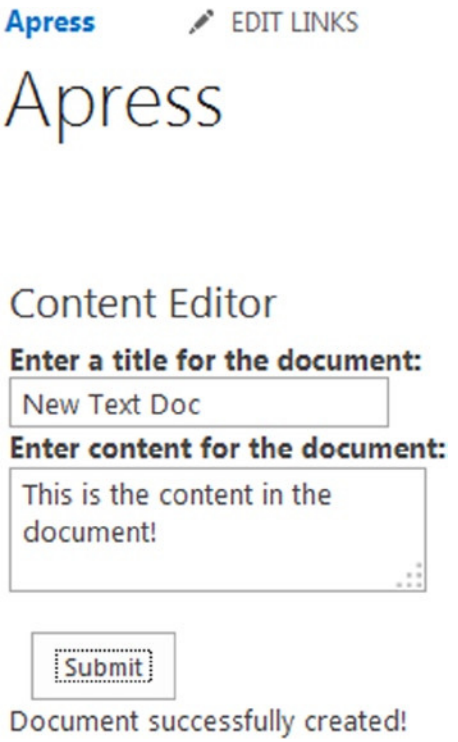

Figure 8-15. *Creating a new document from a Content Editor Web Part*

We can navigate to the Documents folder and find that the new text document has been created. You can click the document and it will open in the browser and display the contents the user entered from the page, as shown in Figure 8-16.

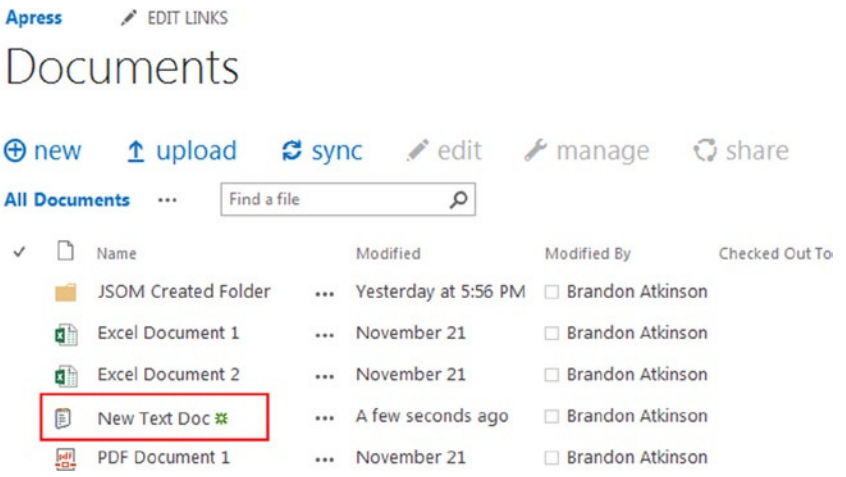

Figure 8-16. *New text document created in the Documents folder*

Read a File

In this section, we'll see how to read the file we created in the previous example.

HTML

Create a new HTML file in Visual Studio and name it **JSOMReadDocument.html**. The markup for this page is extremely simple, with only a reference to the custom JavaScript file and a DIV to output the document contents. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMReadDocument.js"></script>

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

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMReadDocument.js**; include the following code:

```
$(function () {
    readDocument();
});

function readDocument() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var documentLibrary = "/Shared Documents/"
    var fileUrl = "New Text Doc.txt";
    var fullUrl = siteUrl + documentLibrary + fileUrl;
```

```
    $.ajax({
        url: fullUrl,
        type: "GET"
    })
    .done(function (data) {
        $("#divReadDocument").html(data);
    })
    .fail(function () {
        alert("error");
    });
}
```

At first glance, this code should seem concise and you'll notice there is no JSOM code here at all. The beauty of reading a text file from SharePoint is that all you need is jQuery. Since this is a chapter about JSOM, this code should probably be somewhere else, but since we created a document, you should also know how to read it. There is a simple jQuery `Document.ready` function, which will fire the code to read the file. We'll intentionally keep this code simple and hard-code the file name for the document we just created:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var documentLibrary = "/Shared Documents/"
var fileUrl = "New Text Doc.txt";
var fullUrl = siteUrl + documentLibrary + fileUrl;
```

The first thing we do is get the current web's full URL by using `_spPageContextInfo.webAbsoluteUrl`. Next, we hard-code the document library portion of the URL and the file name. Finally, we put them all together into a new variable called `fullUrl`. You could absolutely have done all of that in one line, but you'll find it's easier to debug your code later if you break it up into smaller lines. Once we have the URL for the file, we use a jQuery AJAX call to get the document:

```
$.ajax({
        url: fullUrl,
        type: "GET"
    })
    .done(function (data) {
        $("#divReadDocument").html(data);
    })
    .fail(function () {
        alert("error");
    });
```

When retrieving a text file via AJAX in jQuery, the content will be returned as text and we can simply output that to the page without any modifications. We use the jQuery `.done` function, setting the content of the file to a new variable called `data`. Then we set the data to the HTML of the results DIV on the page.

■ **Note** Just as before, when creating a new file, we kept things simple by utilizing a text file. This is also true for this section and reading a file. You cannot read an Office file using the method just described. You need to utilize the Office JavaScript files to properly read a Word file or an Excel file in the browser. You can find more information at http://msdn.microsoft.com/en-us/library/office/fp160953%28v=office.15%29.aspx.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display the contents of the text file, as shown in Figure 8-17.

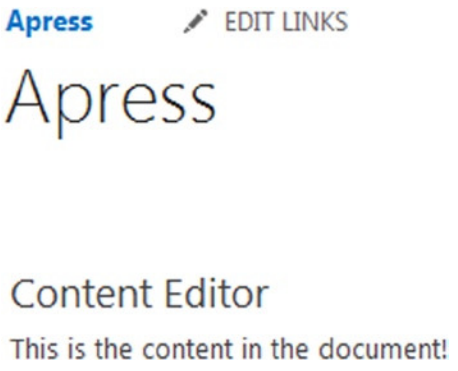

Figure 8-17. *Contents of the text file displayed on the page*

Updating a File

You can also easily update a file using JSOM. In fact, updating the file is essentially the same code as creating the file, with the addition of a single line:

```
fileCreateInfo.set_overwrite(true);
```

This line simply tells SharePoint to overwrite a previous file of the same name. Using this, you could easily read the contents of the text file into a textbox on the page and allow the user to modify the content. When the user wishes to save the file, use the same code as creating the file and include the overwrite command.

Delete a File

In this example, we'll see how to delete a file.

HTML

Create a new HTML file in Visual Studio and name it **JSOMDeleteDocument.html**. The markup for this page has a textbox for the user to enter the name of the file that they wish to delete. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMDeleteDocument.js">
</script>
```

```
<div id="DeleteFile">
    <div>
        <strong>Enter the name of document to delete:</strong>
        <br />
        <input type="text" id="txtDocumentTitle" />
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMDeleteDocument.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteDocument();
    });
}

function deleteDocument() {
    var docTitle = $("#txtDocumentTitle").val() + ".txt";

    var clientContext = new SP.ClientContext();
    var oWebsite = clientContext.get_web();

    var fileUrl = _spPageContextInfo.webServerRelativeUrl +
            "/Shared Documents/" + docTitle;
    this.fileToDelete = oWebsite.getFileByServerRelativeUrl(fileUrl);
    this.fileToDelete.deleteObject();

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    $("#divResults").html("Document successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

After wiring up the button click event, the first thing that we do is grab the title of the document that the user wishes to delete:

```
var docTitle = $("#txtDocumentTitle").val() + ".txt";
```

We'll keep this example simple and delete the text file that was created earlier, and thus hard-code the `.txt` extension into the code. Next, we perform the usual ritual of referencing the current client context and web, and then we construct the path to the document:

```
var fileUrl = _spPageContextInfo.webServerRelativeUrl +
"/Shared Documents/" + docTitle;
```

For this path, we only need the relative path of the document, and not the full URL. We can obtain the relative path of the current web by using `_spPageContextInfo.webServerRelativeUrl`. Since we know that this document resides in the Documents folder, we'll hard-code that and add in the document title that the user submitted. Just like with other delete operations in SharePoint, we do not need to load this request on the client context and can simply call the `deleteObject` method and execute the query. The `success` function will output a message to the user informing them that the document has been deleted.

Unlike with the create, read, and update document functions we saw earlier, you can delete any file that you wish with JSOM, assuming that the user has delete permissions. SharePoint is quite agnostic when it comes to deleting documents. You can easily use the same code to delete Office documents without any issues or extra JavaScript libraries.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a textbox for the user to enter the document title and display a result message, as shown in Figure 8-18.

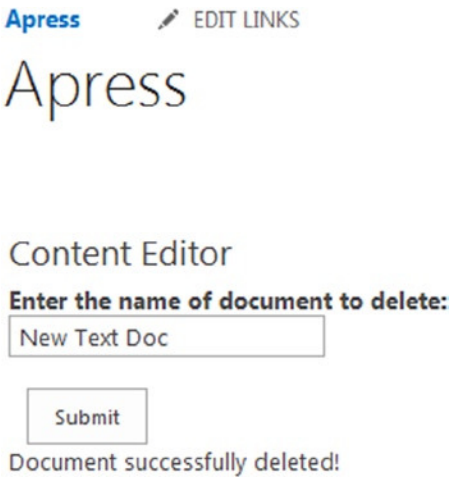

Figure 8-18. *Deleting a file via JSOM*

Working with Sites

You may be thinking at this point that JSOM is pretty powerful! And in fact it is, and to prove it we'll create a new site with it! Working with lists and libraries is pretty cool, but being able to create a new site via JavaScript will really demonstrate the power of JSOM.

Create a Site

In this example, we'll create a new team site. In addition to creating the new site, we'll also set a number of its properties including Title, Description, and Site Template.

HTML

Create a new HTML file in Visual Studio and name it **JSOMCreateSite.html**. The markup for this page has a textbox for the user to enter the name of the new site, as well as a textbox for a site description. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMCreateSite.js"></script>

<div id="DeleteFile">
    <div>
        <strong>Enter the name of the site:</strong>
        <br />
        <input type="text" id="txtSiteTitle" />
    </div>
    <br />
    <div>
        <strong>Enter site description:</strong>
        <br />
        <textarea cols="20" id="txtSiteDescription"></textarea>
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMCreateSite.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createSite();
    });
}
```

```
function createSite() {
    var siteTitle = $("#txtSiteTitle").val();
    var siteDesc = $("#txtSiteDescription").val();
    var siteUrl = siteTitle.replace(/\s/g, "");

    var clientContext = new SP.ClientContext();
    var collWeb = clientContext.get_web().get_webs();

    var webCreationInfo = new SP.WebCreationInformation();
    webCreationInfo.set_title(siteTitle);
    webCreationInfo.set_description(siteDesc);
    webCreationInfo.set_language(1033);
    webCreationInfo.set_url(siteUrl);
    webCreationInfo.set_useSamePermissionsAsParentSite(true);
    webCreationInfo.set_webTemplate('STS#0');

    var oNewWebsite = collWeb.add(webCreationInfo);

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    $("#divResults").html("Site successfully created!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

Creating a new site takes a bit more code, but it should all be very familiar by now. First, we grab the Title and Description that the user submitted by using the following lines:

```
var siteTitle = $("#txtSiteTitle").val();
var siteDesc = $("#txtSiteDescription").val();
var siteUrl = siteTitle.replace(/\s/g, "");
```

When you create a site, you must provide a URL for the site in addition to the Title. For this example, we'll simply take the text that the user entered for the Title and remove all the spaces to make the URL. This is done using the `replace()` method in JavaScript, with the first parameter being what we are searching for, and the second parameter being what it should be replaced with. The `/\s/g` text denotes a space and must be used for this function to work properly.

Once we have the variables needed for site creation, we reference the current client context and then obtain the collection of web sites in the current web:

```
var collWeb = clientContext.get_web().get_webs();
```

The `get_web().get_webs()` method returns a collection of all the subsites in the current web. It's this collection that we'll ultimately add the new site to. Next, we create an `SP.WebCreationInformation` object:

```
var webCreationInfo = new SP.WebCreationInformation();
webCreationInfo.set_title(siteTitle);
webCreationInfo.set_description(siteDesc);
webCreationInfo.set_language(1033);
webCreationInfo.set_url(siteUrl);
webCreationInfo.set_useSamePermissionsAsParentSite(true);
webCreationInfo.set_webTemplate('STS#0');
```

Just like the other `CreationInformation` objects that we've seen before, we create a new `WebCreationInformation` object and set its various properties. Luckily, just about all the methods needed to configure properties on this object are very self-explanatory; for instance, `set_useSamePermissionsAsParentSite()` determines if this new site should inherit the permissions of the parent site. There are some tricky parameters that you may not be familiar with, such as `1033` (which means "English" when setting the language for the site) and `STS#0` (which means a team site template when setting the site template). All the other properties should look familiar.

Once the properties have been set, we simply add the new site to the collection:

```
var oNewWebsite = collWeb.add(webCreationInfo);
```

Creating a new site is also another area where you do not need to load the query on the client context. You can simply execute the query to have the new site created. The `success` function will simply notify the user that the site has been created.

■ **Note** Again, the code in these examples run under the permission level of the current user. Since a lot of these examples perform higher-level functions, like creating and deleting SharePoint objects, the user will need to have appropriate permissions to perform these functions. It's worth pointing this out again, as site creation will require the user to have Full Control permissions for the site where the new web is being created.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display textboxes for the user to enter the new site title and description, and display a result message, as shown in Figure 8-19.

Figure 8-19. *Creating a new site via JSOM*

We can then open the Site Contents page where the new site was created, and scroll to the bottom. In the Subsites section, we can see that the new site has been created, as shown in Figure 8-20.

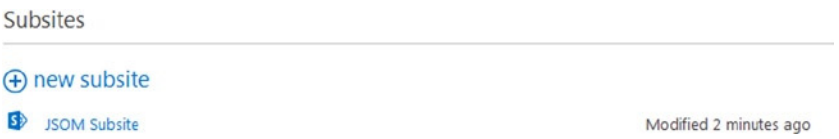

Figure 8-20. *The newly created subsite in the Site Contents page*

Delete a Site

In this section, we'll delete the site we created in the previous example.

HTML

Create a new HTML file in Visual Studio and name it **JSOMDeleteSite.html**. The markup for this page has a textbox for the user to enter the name of the site they would like to delete. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMDeleteSite.js"></script>
```

```
<div id="DeleteFile">
    <div>
        <strong>Enter the name of the site to delete:</strong>
        <br />
        <input type="text" id="txtSiteTitle" />
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMDeleteSite.js**; include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteSite();
    });
}

function deleteSite() {
    var siteTitle = $("#txtSiteTitle").val();
    var siteTitleNoSpaces = siteTitle.replace(/\s/g, "");
    var siteUrl = _spPageContextInfo.webAbsoluteUrl + "/" + siteTitleNoSpaces;

    var clientContext = new SP.ClientContext(siteUrl);
    var oWebsite = clientContext.get_web();

    oWebsite.deleteObject();

    clientContext.executeQueryAsync(
        Function.createDelegate(this, this.onQuerySucceeded),
        Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
    $("#divResults").html("Site successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

Just like in the previous example, we grab the title of the site that the user entered, and then we remove all the spaces:

```
var siteTitle = $("#txtSiteTitle").val();
var siteTitleNoSpaces = siteTitle.replace(/\s/g, "");
var siteUrl = _spPageContextInfo.webAbsoluteUrl + "/" + siteTitleNoSpaces;
```

Once we have the title and have removed the spaces, we need to construct a full URL to the site. We can get the full URL for the current site by using `_spPageContextInfo.webAbsoluteUrl` and simply appending a /and the site title without spaces on the end. We need to construct a full URL so that we can request the client context for the site that we wish to delete, and not the current site that the user is on:

```
var clientContext = new SP.ClientContext(siteUrl);
var oWebsite = clientContext.get_web();
```

Up until now, we have always used the client context for the current site that the user is viewing the page on. However, you can request the context for another site by passing in a URL as a parameter to the `SP.ClientContext()` method, as we have done in this example. Once we have the context for the site, we get a reference to the root web in that site by using `clientContext.get_web()`. Just like with all the other delete operations, we simply call the `deleteObject()` method to request the delete operation:

```
oWebsite.deleteObject();
```

The `success` function will display a message to the user, informing them the delete has taken place.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a textbox for the user to enter the site title, and display a result message, as shown in Figure 8-21.

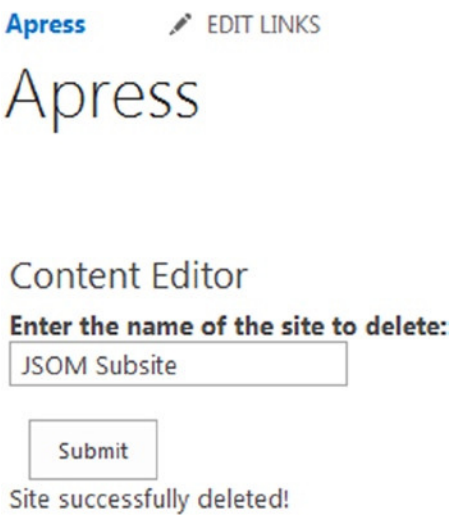

Figure 8-21. *Deleting a subsite via JSOM*

In this example, we'll delete the site that we just recently created. Navigating back to the Site Contents page will show that the subsite was successfully deleted, as shown in Figure 8-22.

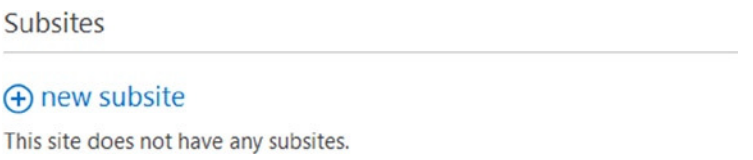

Figure 8-22. *The site has been removed from the root site*

Working with Social Data

One of the big new areas of SharePoint is social data. If you're going to be writing custom solutions for SharePoint you will absolutely encounter the need to access social data at some point. This section will demonstrate how to pull some of this data for use in your solutions.

HTML

Create a new HTML file in Visual Studio and name it **JSOMGetSocialData.html**. The markup for this page will include a couple of DIVs and a SPAN to output data to. It consists of the following lines:

```
<script type="text/javascript" src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.js"></script>
<script type="text/javascript" src="/_layouts/15/sp.userprofiles.js"></script>
<script type="text/javascript" src="/apress/Webparts/chapter 8/JSOMGetSocialData.js"></script>
<link rel="stylesheet" type="text/css" href="/apress/Webparts/chapter 8/
JSOMGetSocialData.css" />

<div id="WelcomeMessage">
    Welcome back <span id="WelcomeMessageUserName"></span>
</div>
<div id="UserFollows" />
```

This example has more references than any other example so far. You'll notice that in addition to the references to `sp.js` and `sp.runtime.js`, we've also included a reference to `sp.userprofiles.js`. As mentioned earlier in this chapter, JSOM functionality is included throughout many different files that serve specific needs. In this example, since we want to get social data for the user, we have to include a reference to the `sp.userprofiles.js` file in order to access that functionality.

We'll also include a custom CSS file titled `JSOMGetSocialData.css`. This will be used to style the output from the script. All of the other examples could have easily included CSS for styling. You'll find in your custom development efforts that styling will play a big role in a lot of your web parts; so this example will show you how easy it is to include.

JavaScript

Create a new JavaScript file in Visual Studio and name it **JSOMGetSocialData.js**; include the following code:

```
$(function () {
    ExecuteOrDelayUntilScriptLoaded(getUserProfileProperties, "sp.userprofiles.js");
});

function getUserProfileProperties() {
    var clientContext = new SP.ClientContext();

    var peopleManager = new SP.UserProfiles.PeopleManager(clientContext);
    this.userProperties = peopleManager.getMyProperties();
    clientContext.load(userProperties);

    var followingManager = new SP.Social.SocialFollowingManager(clientContext);
    this.following = followingManager.getFollowed(15);

    clientContext.executeQueryAsync(
    Function.createDelegate(this, this.onQuerySucceeded),
    Function.createDelegate(this, this.onQueryFailed)
    );
}

function onQuerySucceeded() {
        $("#WelcomeMessageUserName").text(userProperties.get_displayName());

        var followedItems = "Items you are following:<br />";
        $.each(following, function( index, value ) {
                followedItems += "<a href='" + value.get_uri() + "'>" + value.get_name() +
                                 "</a><br />";
        });
        $("#UserFollows").append(followedItems);
}

function onQueryFailed(sender, args) {
    alert('Request failed. ' + args.get_message() +
        '\n' + args.get_stackTrace());
}
```

In the jQuery `Document.ready` function, we include the `ExecuteOrDelayUntilScriptLoaded` built-in function. We need to use this because our JavaScript code relies on the functionality found in `sp.userprofiles.js` and would cause errors if it ran prior to that file being loaded:

```
ExecuteOrDelayUntilScriptLoaded(getUserProfileProperties, "sp.userprofiles.js");
```

Once the getUserProfileProperties function is fired, we obtain a reference to the current client context, and then create a new PeopleManager object with the following code:

```
var peopleManager = new SP.UserProfiles.PeopleManager(clientContext);
this.userProperties = peopleManager.getMyProperties();
clientContext.load(userProperties);
```

We can create a new PeopleManager object by using the SP.UserProfiles.PeopleManager() method, passing in the client context as its parameter. The PeopleManager object provides methods for accessing data about users. In this case, we can obtain all the properties for the current user by using the peopleManager.getMyProperties() method, and loading the userProperties variable on the client context.

Next, we'll get all the items the user is following by using the SocialFollowingManager() object:

```
var followingManager = new SP.Social.SocialFollowingManager(clientContext);
this.following = followingManager.getFollowed(15);
```

We can create a new SocialFollowingManager object using the SP.Social.SocialFollowingManager() method, passing in the client context as its parameter. We can then obtain all the "actors" the user is following by using the getFollowed() method. An "actor" is really just anything that the user follows, which can be users, documents, sites, or tags. When you call the getFollowed() method, you need to pass in a number, which corresponds to the actors that you want returned. In this example, we use 15, which indicates all the actors. You can see the full list of values at http://msdn.microsoft.com/en-us/library/microsoft.sharepoint.client.social.socialactortypes(v=office.15).aspx.

The success method in this example is the most complex so far, as we need to show more data than in other examples. First, we'll get the user name of the current user to display on the page:

```
$("#WelcomeMessageUserName").text(userProperties.get_displayName());
```

The userProperties object has many methods to retrieve data; to get the username, we can call the get_displayName() method. Next, we'll loop through all the followed items to build a list to show the user:

```
var followedItems = "Items you are following:<br />";
$.each(following, function( index, value ) {
       followedItems += "<a href='" + value.get_uri() + "'>" + value.get_name() + "</a><br />";
});
```

First, we create a new variable to hold all the followed items. Next, we loop through each item using a jQuery each loop. For each item, we call the get_uri() and get_name() methods to construct a link so that the user can jump directly to the item from the page. Finally, we append all the HTML to the page with the following line:

```
$("#UserFollows").append(followedItems);
```

CSS

Create a new CSS file in Visual Studio and name it **JSOMGetSocialData.css**; include the following code:

```
#WelcomeMessage {
    margin-bottom: 20px;
}

#WelcomeMessageUserName {
    font-weight: bold;
}

#UserFollows {
    border:dashed 1px #C0C0C0;
    width:240px;
    padding: 6px;
}
```

This CSS is very basic and only intended to provide a simple demonstration of styling. We'll apply a bottom margin on the welcome message to the user, and put the user name in bold. The followed items will be wrapped in a dashed border for emphasis.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a "Welcome back" message to the user, along with their name in bold. Directly underneath is a list of all the items that they are following, displayed as hyperlinks, as shown in Figure 8-23.

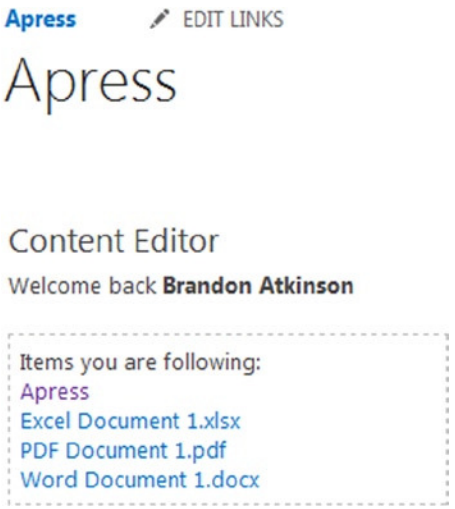

Figure 8-23. *Social data being retrieved via JSOM*

Summary

In this chapter, we took a deep dive into the JavaScript Object Model (JSOM) and looked at how it can be used to create custom web parts using the Etherson method. We looked at a wide variety of examples: querying for list data, creating new lists and sites, and querying for social data about users. Even though this was a long chapter in comparison to others in this book, it only scratched the surface on what is possible with JSOM in SharePoint. However, after working through all the examples presented here, you should have a very firm grasp on working with JSOM and on how HTML, JavaScript, and CSS all work together in a custom web part. You should now be comfortable enough to begin crafting your own custom solutions using JSOM. In the next chapter, we'll look at a lot of the same examples and learn how to perform tasks using the REST API.

CHAPTER 9

Working with the REST API

In the previous chapter, we saw how to perform a great range of tasks using the JavaScript Object Model (JSOM) and the Etherson method. In this chapter, we'll look at the same tasks, except how to perform them using the REST API. To that end, a lot of the HTML from the previous chapter will be similar and the results will be the same. The JavaScript, however, will be completely different. If you skipped the JSOM chapter, don't worry; all the explanations of the code are present, so you won't be left in the dark on any example.

REST stands for *Representational State Transfer*, which is a fancy way of saying, "Web services that allow anything to connect to them via the Web." REST services can be called from any device that is capable of making an HTTP request over the Internet. This type of service has gained a lot of traction lately, as it's a simple means for making data accessible to any system. SharePoint 2013 introduced a new REST API for exposing data for apps. Luckily, the API can be easily consumed via HTML and JavaScript as well.

Note Code that utilizes the REST API runs under the context of the user that is logged in. What this means is that the code only has the ability to perform actions that the user's permission level allows. For instance, if you have code to create a new subsite in the site collection, but the user that is logged in does not have that permission, the code will run but will produce an error. Keep this in mind as you build your solutions, and ensure that you do not show users options that they cannot utilize.

Getting Set Up

The examples in this chapter continue to utilize the Etherson method introduced in Chapter 4. Navigate to the Webparts folder in your site and create a new folder named **Chapter 9**, as shown in Figure 9-1.

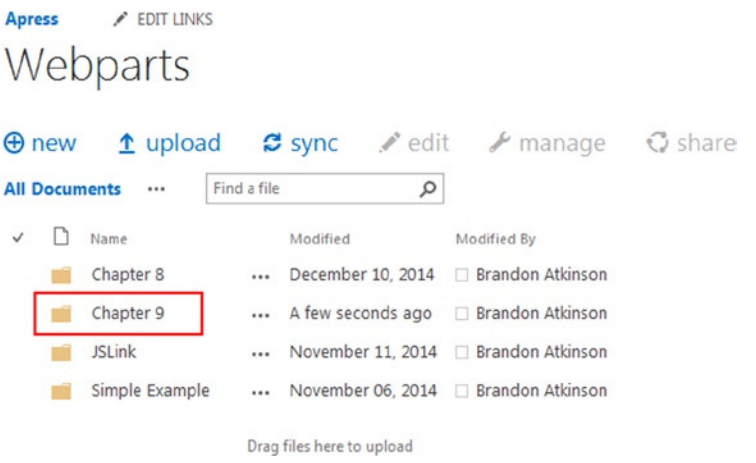

Figure 9-1. *Chapter 9 folder in the Webparts document library*

Each example in this chapter will follow the same pattern. We'll look at the HTML file rendered in the Content Editor Web Part, the JavaScript file used to retrieve and render data, and, finally, the results. We'll dive into each file and along the way I'll explain what is happening in each one.

Working with Lists

As stated in the previous chapter, lists are everywhere in SharePoint! You won't go very far in your development efforts without working with a list. In this section, we'll explore some of the ways you can use the REST API to interact with lists.

Get List Data

In this example, we'll simply retrieve some list data to display to the user.

HTML

Create a new HTML file in Visual Studio and name it **RESTGetListData.html**. The HTML in this example is very simple. It only consists of the following lines:

```
<script type="text/javascript" src="/apress/webparts/chapter 9/RESTGetListData.js"></script>

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

If you worked through the JSOM examples, you'll notice that this HTML is much more concise with fewer JavaScript references. The main reason for this is that when working with the REST API, you do not need to load any of the SharePoint JavaScript files, like JSOM. All the calls to the REST API are performed with JavaScript AJAX calls. We'll be using jQuery's AJAX methods in our examples.

Next, we load in our custom JavaScript file called `RESTGetListData`. This file contains all the code to load data into the DIV on the page, which has an ID of `divGetListData`. All in all, there is not much on this page.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTGetListData.js**. The JavaScript file is more complex in comparison to the HTML. It consists of the following lines:

```
$(function () {
    getListData();
});

function getListData() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists";

    $.ajax({
        url: fullUrl,
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded(data) {
    var listItemInfo = '';

    $.each(data.d.results, function (key, value) {
        listItemInfo += 'Title: ' + value.Title + ' - Created: ' +
            value.Created + '<br />';
    });

    $("#divGetListData").html(listItemInfo);
}

function onQueryFailed() {
    alert('Error!');
}
```

The first thing you'll notice with this file is that there is very little SharePoint code here. The overwhelming majority of the code in this file is JavaScript and jQuery, with only the `_spPageContextInfo.webAbsoluteUrl` call referencing any SharePoint JavaScript. For those new to front-end development with SharePoint, but who have jQuery experience, this can seem the way to go! It certainly does provide a faster way to begin creating custom solutions if you are familiar with jQuery and making AJAX calls.

Most of the REST examples in this book follow the same pattern as this example. Let's dive into each line in detail to fully understand what is going on. The script begins with a jQuery `Document.ready` method, which simply calls a custom function called `getListData`, where we first build the URL to the REST API:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists";
```

We use the `_spPageContextInfo.webAbsoluteUrl` method to obtain the full URL that the user is currently on. Next, we append the path to the REST API for the lists in this web by using `/_api/web/lists`. This produces a URL that looks similar to the following:

```
http://<DOMAIN>/<SITE>/_api/web/lists
```

This is the basic structure for all the REST URLs. The `_api` denotes that this is a call to the REST API. When SharePoint sees this as part of the URL in the AJAX call, it knows to route the request to the REST API for processing. For this particular example, we're asking the list collection on the current web. If you compare this to the JSOM request, you can see some similarities:

```
var collList = clientContext.get_web().get_lists();
```

As you can see, the REST API follows a similar pattern of "Get the web, and then get the lists." This pattern repeats itself throughout the chapter. Next, we make a standard jQuery AJAX call to get the lists in the web:

```
$.ajax({
        url: fullUrl,
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
```

For this AJAX call, the first thing we pass in is the URL of the call. This is the `fullUrl` variable we created earlier that holds the full URL to the REST API. Next, we specify the "type" of call, in this case a `GET`. We have to supply some "headers" in the call so that SharePoint knows we are anticipating JSON back as the result. To do this we pass in `application/json;odata=verbose` for both the `accept` and `content-type` headers. These two headers will be present in just about all your REST calls. Finally, we specify the `success` and `failure` functions that should fire once the call is complete. The `success` function simply displays each list and its created timestamp:

```
function onQuerySucceeded(data) {
    var listItemInfo = '';

    $.each(data.d.results, function (key, value) {
        listItemInfo += 'Title: ' + value.Title + ' - Created: ' +
            value.Created + '<br />';
    });

    $("#divGetListData").html(listItemInfo);
}
```

The structure of the data coming back from a REST call can be confusing at first. If you were to open Firebug and look at the object, you would find that a huge amount of data is being returned. More than you would ever need to show a user. Furthermore, the data you would want to show a user is nested in a property called `d`. For instance, in this example the list collection is in a property called `data.d.results`. If you were to query for a single list, there would be no `results` property, and you could just access data directly off the `d` property, like `data.d.Title`. You'll see many more instances of this throughout the chapter.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The results will display all the lists in the current site, as shown in Figure 9-2.

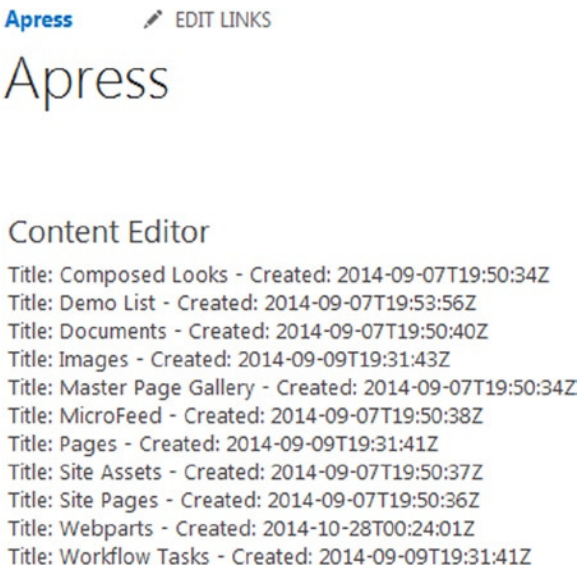

Figure 9-2. *Get List Data results in the Content Editor Web Part*

Create a New List

We can easily create new lists using REST, as you'll see in this example.

HTML

Create a new HTML file in Visual Studio and name it **RESTCreateList.html**. The HTML for this example is a little more complex than the previous example. It consists of the following lines:

```
<script type="text/javascript" src="/apress/webparts/chapter 9/RESTCreateList.js"></script>

<div>
    <strong>Enter a name for the list:</strong>
    <input type="text" id="txtListName" />
    <input type="button" id="btnSubmitListName" value="Submit" />
</div>
<div id="divCreateListResults"></div>
```

The HTML is a little more complex, with several elements on the page. First, there is some simple text wrapped in a `<strong>` tag to make it bold on the page. Next, there is an INPUT of type Text that is just a simple textbox for the user to input the name of the list they wish to create. There is a button for the user to submit the new list request and, finally, a DIV where we can output the results of the list creation. Each element has an ID so that we can easily target it with jQuery.

■ **Note** In order to create a new list via REST, the user executing the code must have at least Manage Lists permissions in SharePoint. The page will be rendered no matter the permission level; however, SharePoint will throw an error if the user does not have adequate permissions.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTCreateList.js**. The JavaScript file is more complex in comparison. It consists of the following lines:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        createList(listName);
    });
}

function createList(listName) {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists";

    $.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            '__metadata': { 'type': 'SP.List' },
            'BaseTemplate': 100,
            'Title': listName
        }),
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded(data) {
    $("#divCreateListResults").html(data.d.Title + " successfully created!");
}

function onQueryFailed() {
    alert('Error!');
}
```

In this example, we don't need to fire any actions on the page load; however, we do need to wire up a button click event when the page loads. In the `Document.ready` function, we fire the `bindButtonClick` function, which we use the jQuery `on` operator to wire a click event on the `btnSubmitListName` button in the HTML. When the user clicks the button, we grab the text from the textbox on the page and pass it to the `createList()` function. This is accomplished using the following lines:

```
function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        createList(listName);
    });
}
```

The `createList()` function takes a variable called `listName`, which is the text the user inputs on the page. Next, we build the URL for the REST call using the following code:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists";
```

You'll notice that this is the same URL as in the previous example, as we are targeting the list collection in the web. The AJAX, however, looks quite different:

```
$.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            '__metadata': { 'type': 'SP.List' },
            'BaseTemplate': 100,
            'Title': listName
        }),
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
```

Just as in the previous example, we pass the `fullUrl` variable in as the URL to call, and the type is now a `POST`. Since we are creating a new list in this example, we use the `POST` to indicate to SharePoint that this is a creation operation. Next, we pass in the data for the new list using the following lines:

```
data: JSON.stringify({
            '__metadata': { 'type': 'SP.List' },
            'BaseTemplate': 100,
            'Title': listName
        })
```

When we perform any type of operation using the REST API other than GET, we are passing data to SharePoint. This data is in the JSON format. The JSON.stringify method is a standard JavaScript method that converts JavaScript data structures into JSON text. This helps to ensure that the data we pass to SharePoint is in the proper format. The first thing we do is set the "type" as "SP.List" and set the "BaseTemplate" to "100", which is a generic list. Finally, we pass in the listName variable for the Title. Next, we add the headers and include a new header called X-RequestDigest:

```
"X-RequestDigest": $("#__REQUESTDIGEST").val()
```

This header allows us to pass in the value for the FormDigest control on the page. This control is used as a security measure in SharePoint to ensure that a POST request is in fact coming from a valid SharePoint page. You don't need to know all the details of this control, other than you can obtain its value using the jQuery line $("#__REQUESTDIGEST").val(). Once the headers are in place, we specify the success and failure functions. The success function simply displays the list name along with a success message:

```
function onQuerySucceeded(data) {
    $("#divGetListData").html(data.d.Title + " successfully created!");
}
```

This function takes in a parameter called data. This variable will be populated with the data from the list that was just created. There is nothing special about the naming of this variable; we could call it "list", or "myCoolData". You could even not specify a parameter if you wish and just display a success message!

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show the textbox and button for the user to input a new list name. Type **My Custom REST List** and then click the Submit button. The results will display the success message, as shown in Figure 9-3.

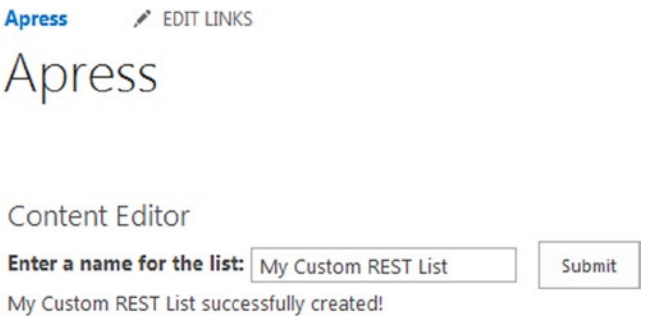

Figure 9-3. *Create new list code after it has been run*

Since this is all done using JavaScript and AJAX, the list will get created and the user will be notified as expected. However, the left navigation on the page will not be updated because the page has not been refreshed. You could absolutely change this code to refresh the page on success, but for this example, we'll simply refresh the page in the browser. Figure 9-4 shows the page after a refresh, where the new list is displayed in the left navigation.

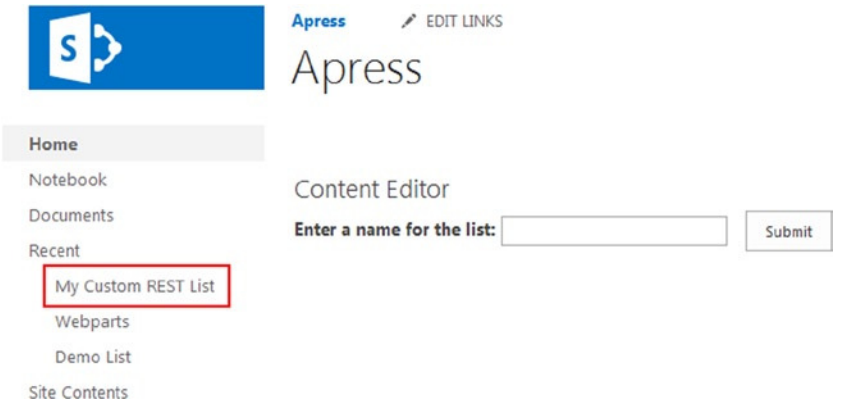

Figure 9-4. *The new list as displayed in the left navigation pane*

Delete a List

In this example, we'll simply delete the list we just created.

HTML

Create a new HTML file in Visual Studio and name it **RESTDeleteList.html**. The HTML for this example is almost identical to the previous example. It consists of the following lines:

```
<script type="text/javascript" src="/apress/webparts/chapter 9/RESTDeleteList.js"></script>

<div>
    <strong>Enter the name of the list to delete:</strong>
    <input type="text" id="txtListName" />
    <input type="button" id="btnSubmitListName" value="Submit" />
</div>
<div id="divDeleteListResults"></div>
```

The only difference here is that we updated the text on the page to indicate that this web part will delete a list, and changed the ID of the results DIV to `divDeleteListResults`.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTDeleteList.js**. The JavaScript file consists of the following lines:

```
$(function () {
    bindButtonClick();
});
```

```
function bindButtonClick() {
    $("#btnSubmitListName").on("click", function () {
        var listName = $("#txtListName").val();
        deleteList(listName);
    });
}

function deleteList(listName) {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('" + listName + "')";

    $.ajax({
        url: fullUrl,
        type: "POST",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val(),
            "X-HTTP-Method": "DELETE",
            "IF-MATCH": "*"
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded() {
    $("#divDeleteListResults").html("List successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Error!');
}
```

The jQuery button wiring is the same as in the previous example. Next, we need to craft the URL for the REST call, this time requesting a specific list:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('" + listName + "')";
```

This URL adds the additional `GetByTitle` command, passing in the name of the list we are looking for. The final URL will look like the following:

```
http://<DOMAIN>/<SITE>/_api/web/lists/GetByTitle('<LISTNAME>')
```

Next, we need to add in a couple of additional headers:

```
"X-HTTP-Method": "DELETE",
"IF-MATCH": "*"
```

You'll notice that the type of request is a `POST`, just like when we created a new list. However, here we want to initiate a "delete" request, so we need to include the `X-HTTP-Method` header to let SharePoint know that we want to delete the list, and not take some other action on it. Next, we include the `IF-MATCH` header. This header is required when making an update via REST and when deleting. This header helps prevent concurrent updates from happening. For instance, if you were trying to update a list item, you could pass in an `etag` property that would look like the following:

```
"If-Match": data.d.__metadata.etag
```

Passing the * in this header basically tells SharePoint not to worry about concurrency and to just perform the update. In this example, we're deleting the list, so we don't need to worry about this check. The `success` function simply notifies the user that the deletion was successful.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show the textbox and button for the user to input a list name to delete. Type **My Custom REST List** and then click the Submit button. The results will display the success message, as shown in Figure 9-5.

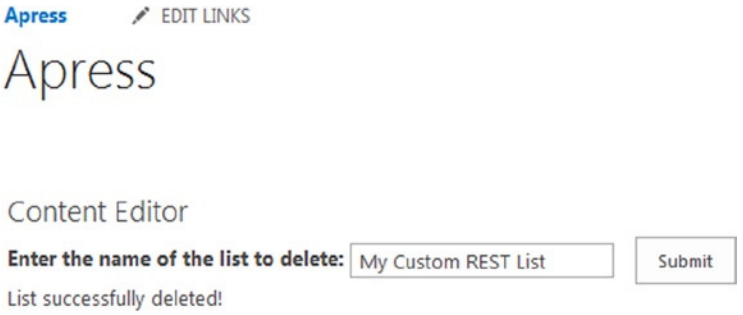

Figure 9-5. *Delete List code after it has successfully run*

Since this is all done using JavaScript and AJAX, the list will be deleted and the user will be notified as expected. However, the left navigation on the page will not be updated because the page has not been refreshed. As before, you could update this code to refresh the page on success, but for this example, we'll simply refresh the page in the browser. Figure 9-6 shows the page after a refresh where the list is no longer displayed in the left navigation.

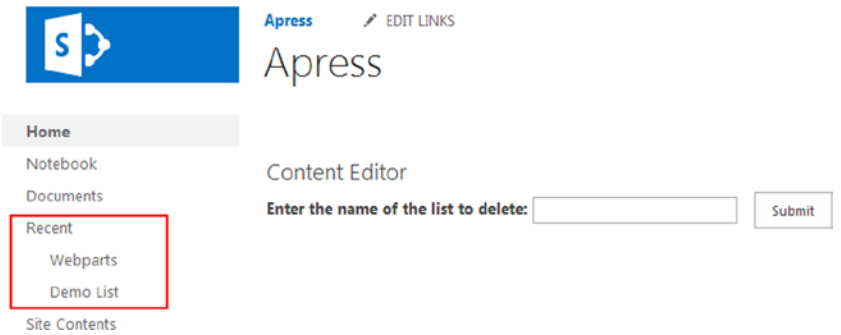

Figure 9-6. *Deleted list has been removed from the left navigation*

Working with List Items

Since working with lists is a very common task in SharePoint development, working with list items is even more common. In this section we'll explore some ways you can interact with list items using the REST API.

Get List Items

This example will show you how to retrieve list items using REST.

HTML

Create a new HTML file in Visual Studio and name it **RESTGetListItems.html**. The HTML for this example continues to follow the minimalist pattern of the previous examples. It consists of the following lines:

```
<script type="text/javascript" src="/apress/webparts/chapter 9/RESTGetListItems.js"></script>

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

For this example, we'll simply grab all the list items from the Demo List and display them in the `divListItems` DIV on the page.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTGetListItems.js**. The script to retrieve list items should look very familiar to the other scripts that we've seen so far. It consists of the following lines:

```
$(function () {
    retrieveListItems();
});

function retrieveListItems() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items";

    $.ajax({
        url: fullUrl,
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}
```

```
function onQuerySucceeded(data) {
    var listItemInfo = '';

    $.each(data.d.results, function (key, value) {
        listItemInfo += '<strong>ID: </strong> ' + value.Id +
            ' <strong>Title:</strong> ' + value.Title +
            '<br />';
    });

    $("#divListItems").html(listItemInfo);
}

function onQueryFailed(sender, args) {
    alert('Error!');
}
```

The URL for the REST call to get list items should look very familiar at this point:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items";
```

As with the delete URL, we get a list's title by using the `GetByTitle` command, and then add the `items` command, which tells SharePoint to return all the items in the list. The AJAX call is a simple `GET` with the familiar JSON headers included. The `success` function uses jQuery to loop through the results:

```
$.each(data.d.results, function (key, value) {
        listItemInfo += '<strong>ID: </strong> ' + value.Id +
            ' <strong>Title:</strong> ' + value.Title +
            '<br />';
    });
```

Just like when we access the lists in the site, we use the `data.d.results` collection to access the query results. In the jQuery `each` loop, we pull out the ID and Title for each list item and display these results on the page.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will now show all the items in the Demo List, as shown in Figure 9-7.

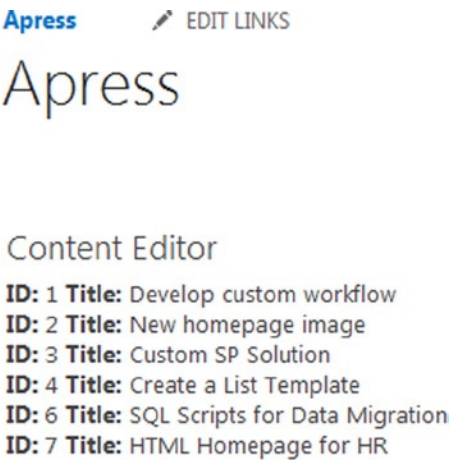

Figure 9-7. *List items from the Demo List displayed in the Content Editor Web Part*

Limit Results with Query Operators

In the previous chapter, we discussed the fact that in most cases you should not be requesting all items from a list; whether using JSOM or REST, it's not a good idea. We saw how you can limit results in JSOM using the CAML query. When using the REST API, you can use query operators to limit the results that are returned. Query operators are basically query string parameters that are added on to the end of the REST URL in your AJAX call.

JavaScript

Update the `RESTGetListItems.js` from the previous example and update the section where you declare the REST URL with the following lines:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items?$filter=Id ge 4";
```

The only addition here is on the end of the URL, with the following query string: `$filter=Id ge 4`. We are using the `filter` operator, which filters the results. We specify that we want to filter on the ID that is greater than or equal to (`ge`) 4. This is very simple addition, but it is a good introduction to the query operators.

■ **Note** Query operators are a large topic—way too big for this book. These operators allow you to select, filter, sort, order, and more. You can even tell SharePoint to only return specific data about items in SharePoint by using these operators. For more in-depth information on using query operators, please visit `http://msdn.microsoft.com/en-us/library/office/fp142385%28v=office.15%29.aspx`.

Results

Update the JavaScript file and upload it, overwriting the previous version. Refresh the page with the Content Editor Web Part and you will see that the results now only show items with an ID greater than or equal to 4, as shown in Figure 9-8.

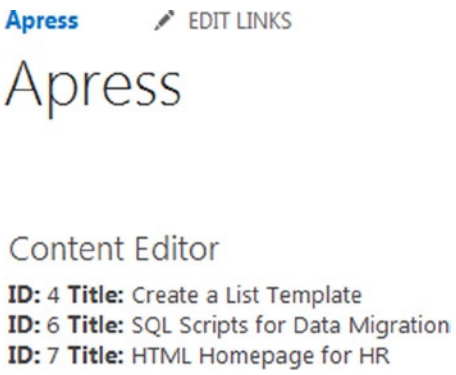

Figure 9-8. *The results of the Demo List query limited by a query operator*

Add List Items

In this example, we'll look at how you can add new list items to a list.

HTML

Create a new HTML file in Visual Studio and name it **RESTAddListItems.html**. The markup for this page will allow a user to add an item to the Demo List, providing a Title and a Description. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTAddListItems.js"></script>

<div id="AddListData">
    <div>
        Title:
        <br />
        <input type="text" id="txtTitle" />
    </div>
    <div>
        Description:
        <br />
        <textarea cols="20" id="txtDesc"></textarea>
    </div>
    <br />
    <div>
        <input id="btnSubmit" type="button" value="Submit" />
    </div>
</div>
<div id="divResult"></div>
```

We provide an INPUT of type TEXT for the list item Title and a TEXTAREA for the Description. Each element on the page has an ID, so we can easily target it with jQuery.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTADDListItems.js**. The script to add list items will look similar to the script for creating a new list. It consists of the following lines:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        addListItem();
    });
}

function addListItem() {
    var title = $("#txtTitle").val();
    var desc = $("#txtDesc").val();

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items";

    $.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            '__metadata': { 'type': 'SP.Data.Demo_x0020_ListListItem' },
            'Title': title,
            'Description': desc,
            'Status':'On-time'
        }),
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });

}

function onQuerySucceeded(sender, args) {
    $("#divResult").html("Item successfully added!");
}

function onQueryFailed() {
    alert('Error!');
}
```

After we attach a button click event with jQuery, the first thing we do on the click event is grab the values that the user is providing. This is accomplished with the following lines:

```
var title = $("#txtTitle").val();
var desc = $("#txtDesc").val();
```

Next, we craft the REST URL, targeting the Demo List and the list items in the list:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items";
```

We then create a jQuery AJAX `POST` call, passing in the data for the new list item:

```
data: JSON.stringify({
            '__metadata': { 'type': 'SP.Data.Demo_x0020_ListListItem' },
            'Title': title,
            'Description': desc,
            'Status':'On-time'
        })
```

This code looks very familiar to the Create List code seen earlier in the chapter. The first thing we do is specify the type of item being created; in this case, an `SP.Data.Demo_x0020_ListListItem`. The type of the item always includes the name of the list from which it is being created, and it follows this pattern:

```
SP.Data.<LISTNAME>ListItem
```

In this example, since our list is titled Demo List, we have to include the `_x0020_`, which SharePoint uses to represent a space. For the Title and Description, we simply pass in the values that we retrieved earlier. For the Status column, we simply hard-code a value of **On-time**. Since we are performing a `POST`, we include the `X-RequestDigest` header. The `success` function will simply display a message to the user that the item was added to the list.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textboxes for the user to input the details. It will display a success message when an item is added, as shown in Figure 9-9.

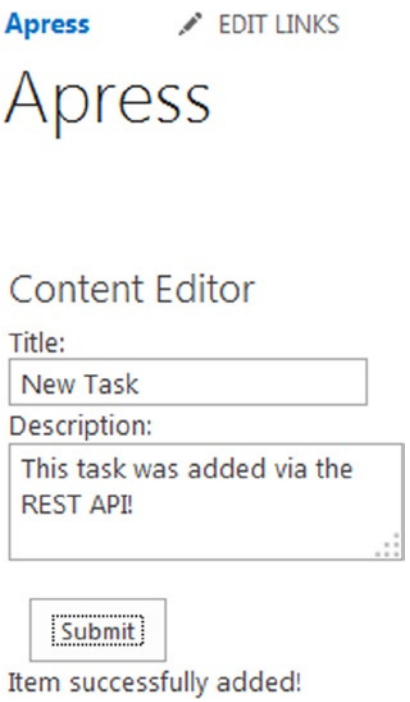

Figure 9-9. *Adding a new item to the Demo List*

We can then navigate to the Demo List to confirm that a new item was added. Figure 9-10 shows the Demo List with a new list item added and with the values supplied by the HTML page.

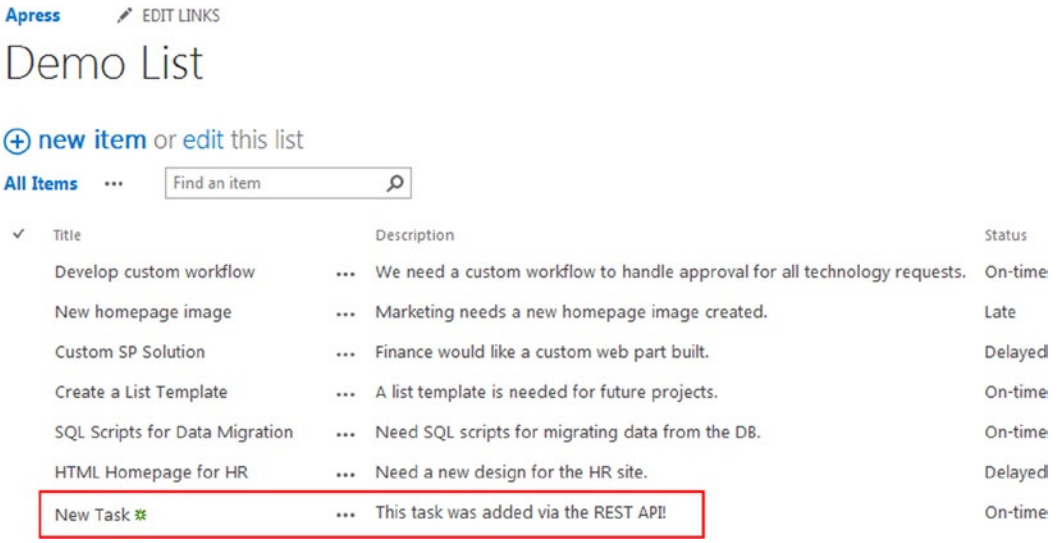

Figure 9-10. *A new list item added to the Demo List*

Delete List Items

For this example we'll simply delete the new list item we created in the previous section.

HTML

Create a new HTML file in Visual Studio and name it **RESTDeleteListItems.html**. The HTML for this file will be much smaller, as all we need to capture from the user is an ID corresponding to the list item that they wish to delete. This is accomplished with the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTDeleteListItems.js"></script>

<div>
    Enter ID to Delete: <input type="text" id="txtId" />
</div>
<div>
    <input id="btnSubmit" type="button" value="Submit" />
</div>
<div id="divResult"></div>
```

This example is overly simplified, in that a user will most likely not know the ID of a certain list item to delete. A more real-world example would most likely show the user a list of items and allow them to click one to delete it. In such a scenario, you would have the list item IDs on the page and you could easily access it. We'll keep a simple textbox to keep the example streamlined.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTDeleteListItems.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteListItem();
    });
}

function deleteListItem() {
    var id = $("#txtId").val();

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items(" + id + ")";
```

```
    $.ajax({
        url: fullUrl,
        type: "POST",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val(),
            "X-HTTP-Method": "DELETE",
            "IF-MATCH": "*"
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded(sender, args) {
    $("#divResult").html("Item successfully deleted!");
}

function onQueryFailed() {
    alert('Error!');
}
```

The `deleteListItem()` function will be called on the button click event on the page. The first thing we do is grab the ID that the user entered on the page. This is accomplished using the following line:

```
var id = $("#txtId").val();
```

Next, we craft the REST URL targeting the specific list item by its ID:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/lists/GetByTitle('Demo List')/items(" + id + ")";
```

You can request or target a single list item by including its ID in parentheses at the end of the URL. The full URL would look like the following:

```
http://<DOMAIN>/<SITE>/_api/web/lists/GetByTitle('<LISTNAME>')/items(<ID>)
```

Just as before, when deleting a list, we include the `"X-HTTP-Method": "DELETE"` and `"IF-MATCH": "*"` headers when making the AJAX call. The `success` method will simply display a message to the user, informing them that the delete has been completed.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textbox for the user to input the ID and will then display a success message when an item is deleted, as shown in Figure 9-11.

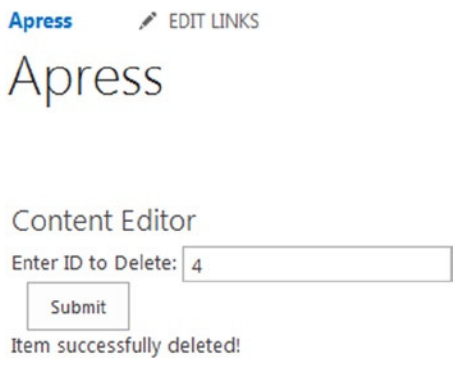

Figure 9-11. *Deleting a list item using REST*

Working with Document Libraries

One of SharePoint's main strengths lies in document management and it's document libraries. You will certainly find requirements where you will need to work with libraries via code. In this section, we'll look at some of the ways you can interact with document libraries using the REST API.

Create a New Folder

Folders are something SharePoint users love to create, and in this example we'll see how to create one via code.

HTML

Create a new HTML file in Visual Studio and name it **RESTCreateFolder.html**. The markup for this page is quite simple, allowing the user to enter a name for the folder that they wish to create. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTCreateFolder.js"></script>

<div>
    <strong>Enter a name for the folder:</strong>
    <input type="text" id="txtFolderName" />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTCreateFolder.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});
```

```
function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createFolder();
    });
}

function createFolder() {
    var folderName = $("#txtFolderName").val();

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/folders";

    $.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            '__metadata': { 'type': 'SP.Folder' },
            'ServerRelativeUrl': 'Shared Documents/' + folderName
        }),
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded() {
    $("#divResults").html("Folder successfully created!");
}

function onQueryFailed() {
    alert('Error!');
}
```

After using jQuery to wire up the button click, the `createFolder()` function begins with grabbing the name of the folder that the user entered on the page:

```
var folderName = $("#txtFolderName").val();
```

Next, we create the REST URL, but this time we target the Folders collection for the site:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/folders";
```

This breaks from the familiar pattern that we've seen up to this point, in that we're requesting a URL that is not specifically where we want to create the folder. We specify where the folder should reside in the `data` property:

```
data: JSON.stringify({
            '__metadata': { 'type': 'SP.Folder' },
            'ServerRelativeUrl': 'Shared Documents/' + folderName
        })
```

The `ServerRelativeUrl` property is used to tell SharePoint where the folder should be created. In this example, we'll create it in the Shared Documents folder and append the folder name that the user entered on the page. As with the previous examples, we specify a "type" for the item, which in this case is an `SP.Folder`. We include the familiar headers and notify the user of the folder creation in the `success` method.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show the textbox for the user to input the name of the folder to be created, and will then display a success message when completed, as shown in Figure 9-12.

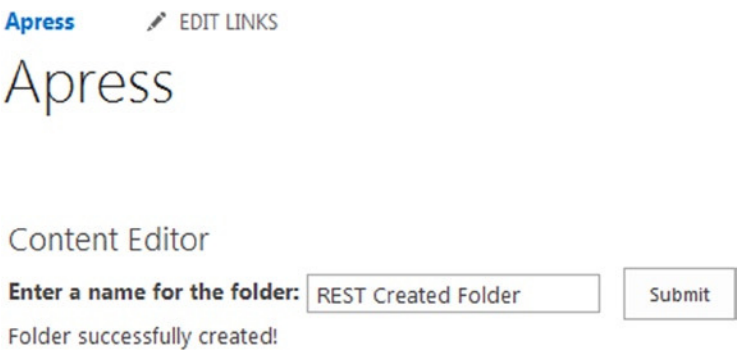

Figure 9-12. *Creating a new folder with REST*

We can then navigate to the Documents library to visually verify that the new folder was created, as shown in Figure 9-13.

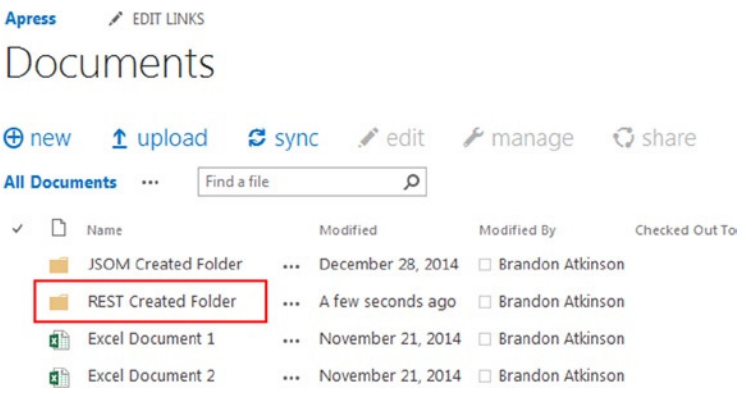

Figure 9-13. *New folder created via REST*

Working with Files

There will be times you will need to work with files in your custom solutions. In this section, we'll see how to create, read, and delete text files using REST.

Create a New File

In this example, we'll see how you can create a new text file via code.

HTML

Create a new HTML file in Visual Studio and name it **RESTCreateDocument.html**. The markup for this page will allow the user to enter a name for a new text file, as well as the content for the file itself. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTCreateDocument.js"></script>

<div id="CreateFile">
    <div>
        <strong>Enter a title for the document:</strong>
        <br />
        <input type="text" id="txtDocumentTitle" />
    </div>
    <div>
        <strong>Enter content for the document:</strong>
        <br />
        <textarea cols="20" id="txtDocumentContent"></textarea>
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTCreateDocument.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createDocument();
    });
}

function createDocument() {
    var docTitle = $("#txtDocumentTitle").val() + ".txt";
    var docContent = $("#txtDocumentContent").val();

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/GetFolderByServerRelativeUrl('Shared Documents')/
    Files/add(url='" + docTitle + "',overwrite=true)";

    $.ajax({
        url: fullUrl,
        type: "POST",
        data: docContent,
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded() {
    $("#divResults").html("Document successfully created!");
}

function onQueryFailed() {
    alert('Error!');
}
```

The first thing that we need to do is grab the document title and the content that the user entered on the page. This is accomplished with the following lines:

```
var docTitle = $("#txtDocumentTitle").val() + ".txt";
var docContent = $("#txtDocumentContent").val();
```

Notice that we append `.txt` to the end of the document title. This will let SharePoint know that we intend to create a text file, as it will use the extension in the name to determine the file type. Next, we build the REST URL:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/GetFolderByServerRelativeUrl('Shared Documents')
/Files/add(url='" + docTitle + "',overwrite=true)";
```

This is by far the most complex REST URL that we've crafted so far. In order to reference the folder where we want to create the file, we need to include the `GetFolderByServerRelativeUrl` command, passing in the folder name; in this case, Shared Documents. We then target the Files collection in the folder and use the `add` command to actually create the file. We can pass in properties for the file; in this case, the URL and the Overwrite flag. The content of the file is passed in via the `data` property in the AJAX call:

```
$.ajax({
        url: fullUrl,
        type: "POST",
        data: docContent,
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        }
```

The `success` function displays a message to the user that the file has been created.

■ **Note** In this example, we are creating a simple text file. In most real-world scenarios, a text file would not be the intended result, but perhaps a Word file or an Excel file instead. You could absolutely change this example to generate a Word document by simply changing the `.txt` to `.docx`. This simple change would in fact create a new Word document with the content in it. However, it could only be opened in Word on the client machine, not Office Web Apps. Luckily, there are Office JavaScript files available via Microsoft to allow you to create and manipulate Office files from the browser. You can find more information at http://msdn.microsoft.com/en-us/library/office/fp160953%28v=office.15%29.aspx.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will show textboxes for the user to input the title of the document and the content, and will then display a success message when the new document has been created, as shown in Figure 9-14.

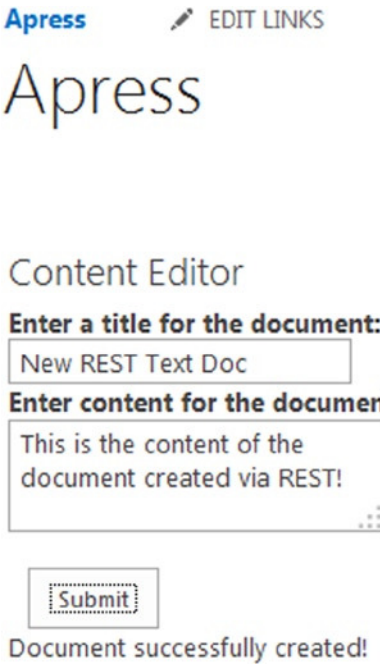

Figure 9-14. *Creating a new document from a Content Editor Web Part*

We can navigate to the Documents folder and find that the new text document has been created, as shown in Figure 9-15. You can click the document and it will open in the browser and display the contents that the user entered from the page.

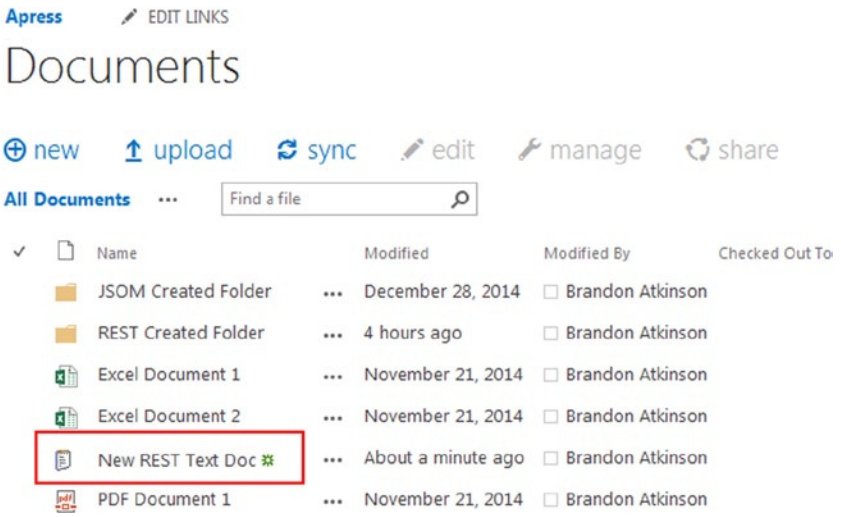

Figure 9-15. *New text document created in the Documents folder*

Read a File

In this example, we'll see how to read the text file we just created and display the contents to the user.

HTML

Create a new HTML file in Visual Studio and name it **RESTReadDocument.html**. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTReadDocument.js"></script>

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

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTReadDocument.js**. Include the following code:

```
$(function () {
    readDocument();
});

function readDocument() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var documentLibrary = "/Shared Documents/"
    var fileUrl = "New Text Doc.txt";
    var fullUrl = siteUrl + documentLibrary + fileUrl;

    $.ajax({
        url: fullUrl,
        type: "GET"
    })
    .done(function (data) {
        $("#divReadDocument").html(data);
    })
    .fail(function () {
        alert("error");
    });
}
```

This is the same code example from the previous chapter, as it does not require JSOM or REST! If you skipped the JSOM chapter, read on; otherwise, the only thing that has changed are the file references.

The beauty of reading a text file from SharePoint is all that you need is jQuery. There is a simple jQuery `Document.ready` function that will fire the code to read the file. We'll intentionally keep this code simple and hard-code the file name for the document we just created:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var documentLibrary = "/Shared Documents/"
var fileUrl = "New REST Text Doc.txt";
var fullUrl = siteUrl + documentLibrary + fileUrl;
```

The first thing we do is get the current web's full URL using `_spPageContextInfo.webAbsoluteUrl`. Next, we hard-code the document library portion of the URL and the file name. Finally, we put them all together into a new variable called `fullUrl`. You could absolutely have done all of that in one line, but you'll find it's easier to debug your code later if you break it up into smaller lines. Once we have the URL for the file, we use a jQuery AJAX call to get the document:

```
$.ajax({
        url: fullUrl,
        type: "GET"
    })
    .done(function (data) {
        $("#divReadDocument").html(data);
    })
    .fail(function () {
        alert("error");
    });
```

When retrieving a text file via AJAX in jQuery, the content will be returned as text and we can simply output that to the page without any modifications. We use the jQuery `.done` function, setting the content of the file to a new variable called `data`. Then, we set the data to the HTML of the results DIV on the page.

■ **Note** As before, when creating a new file, we kept things simple by utilizing a text file. This is also true for this section and reading a file. You cannot read an Office file using the method described earlier. You will need to utilize the Office JavaScript files to properly read a Word file or an Excel file in the browser. You can find more information at http://msdn.microsoft.com/en-us/library/office/fp160953%28v=office.15%29.aspx.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display the contents of the text file, as shown in Figure 9-16.

Figure 9-16. *Contents of the text file displayed on the page*

Delete a File

In this example, we'll simply delete the newly created text file.

HTML

Create a new HTML file in Visual Studio and name it **RESTDeleteDocument.html**. The markup for this page has a textbox for the user to enter the name of the file that they wish to delete. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTDeleteDocument.js"></script>

<div id="DeleteFile">
    <div>
        <strong>Enter the name of document to delete:</strong>
        <br />
        <input type="text" id="txtDocumentTitle" />
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTDeleteDocument.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteDocument();
    });
}

function deleteDocument() {
    var docTitle = $("#txtDocumentTitle").val() + ".txt";

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var webRelUrl = _spPageContextInfo.webServerRelativeUrl;
    var fullUrl = siteUrl + "/_api/web/GetFileByServerRelativeUrl('" + webRelUrl +
    "/Shared Documents/" + docTitle + "')";
```

```
    $.ajax({
        url: fullUrl,
        type: "POST",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val(),
            "X-HTTP-Method": "DELETE",
            "IF-MATCH": "*"
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded() {
    $("#divResults").html("Document successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Error!');
}
```

After wiring up the button click event, the first that we do is grab the title of the document that the user wishes to delete:

```
var docTitle = $("#txtDocumentTitle").val() + ".txt";
```

We'll keep this example simple and delete the text file that was created earlier, and thus hard-code the `.txt` extension into the code. Next, we build the REST URL as we've done in all the previous examples:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var webRelUrl = _spPageContextInfo.webServerRelativeUrl;
var fullUrl = siteUrl + "/_api/web/GetFileByServerRelativeUrl('" + webRelUrl +
"/Shared Documents/" + docTitle + "')";
```

For this example, in addition to getting the web's absolute URL, `webAbsoluteUrl`, we also need to the get the server-relative URL, `webServerRelativeUrl`. Where the absolute URL looks like `http://<DOMAIN>/<SITE>`, the server-relative URL of the site looks like `/<SITE>`. We need this value to properly target the file in the REST URL. In order to delete the file, we need to utilize the `GetFileByServerRelativeUrl` command, passing in the relative URL to the file. In this case, we combine the server relative URL, `/Shared Documents/` and the document title that the user passed in.

As with the other delete operations, we include the `"X-HTTP-Method": "DELETE"` and `"IF-MATCH": "*"` headers. The `success` function will simply notify the user that the delete took place.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a textbox for the user to enter the document title and display a result message, as shown in Figure 9-17.

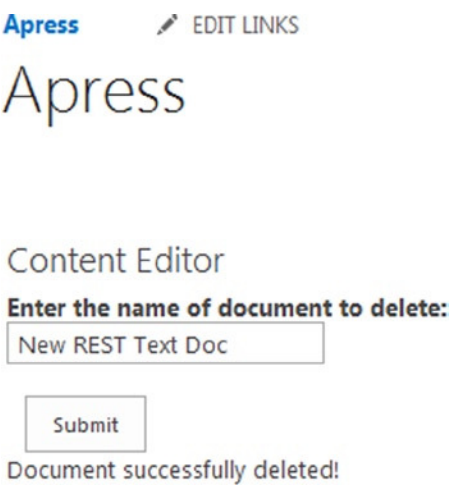

Figure 9-17. *Deleting a file via REST*

Working with Sites

Creating new sites in SharePoint is a common task. In this section, we'll see how to create and delete a site using the REST API. The previous examples definitely illustrated the power of the REST API, and this section should drive home just how much power it offers you in your development efforts.

Create a Site

In this example, we'll create a new site and set several of it's properties.

HTML

Create a new HTML file in Visual Studio and name it **RESTCreateSite.html**. The markup for this page has a textbox for the user to enter the name of the new site, as well as a textbox for a site description.
It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTCreateSite.js"></script>

<div id="DeleteFile">
    <div>
        <strong>Enter the name of the site:</strong>
        <br />
        <input type="text" id="txtSiteTitle" />
    </div>
    <br />
    <div>
        <strong>Enter site description:</strong>
        <br />
        <textarea cols="20" id="txtSiteDescription"></textarea>
    </div>
    <br />
```

```
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTCreateSite.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        createSite();
    });
}

function createSite() {
    var newSiteTitle = $("#txtSiteTitle").val();
    var newSiteDesc = $("#txtSiteDescription").val();
    var newSiteUrl = newSiteTitle.replace(/\s/g, "");

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/web/webinfos/add";

    $.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            'parameters': {
                '__metadata':  {'type': 'SP.WebInfoCreationInformation' },
                'Url': newSiteUrl,
                'Title': newSiteTitle,
                'Description': newSiteDesc,
                'Language':1033,
                'WebTemplate':'sts#0',
                'UseUniquePermissions': false
            }
        }),
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}
```

```
function onQuerySucceeded() {
    $("#divResults").html("Site successfully created!");
}

function onQueryFailed(sender, args) {
    alert('Error!');
}
```

Creating a new site takes a bit more code, but it should all be very familiar by now. First, we grab the Title and Description that the user submitted. This is accomplished using the following lines:

```
var newSiteTitle = $("#txtSiteTitle").val();
var newSiteDesc = $("#txtSiteDescription").val();
var newSiteUrl = newSiteTitle.replace(/\s/g, "");
```

When you create a site, you must provide a URL for the site in addition to the Title. For this example, we'll simply take the text the user entered for the Title and remove all the spaces to make the URL. This is done using the `replace()` method in JavaScript, with the first parameter being what we are searching for, and the second being what it should be replaced with. The `/\s/g` text denotes a space and must be used for this function to work properly.

Once we have the variables needed for site creation, we create the REST URL, but this time we target the Webinfos collection for the site:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/web/webinfos/add";
```

Similar to when we created a new folder, we're targeting a location that is different from where the site will ultimately reside. We also include the `add` command in the URL; this is different from the other creation scripts so far. Next, we create the AJAX call in jQuery:

```
$.ajax({
        url: fullUrl,
        type: "POST",
        data: JSON.stringify({
            'parameters': {
                '__metadata':  {'type': 'SP.WebInfoCreationInformation' },
                'Url': newSiteUrl,
                'Title': newSiteTitle,
                'Description': newSiteDesc,
                'Language':1033,
                'WebTemplate':'sts#0',
                'UseUniquePermissions': false
            }
        })
```

This example differs from previous ones when passing in the data using the `JSON.strigify()` method. You'll notice that we have to create a `parameters` object that then contains all the information for creating the new site. This was not required for the other examples. There is nothing of note about this object, as it simply serves as a wrapper for the data inside; however, SharePoint expects it and will not create a site without this structure in place.

First, we specify the "type" and in this case it's an `SP.WebInfoCreationInformation` type. If you worked through the JSOM examples in the previous chapter, you'll recognize this as the same object used to create sites using JSOM. The remainder of the properties are fairly straightforward: we pass in the site title, URL, and description variables that were created earlier. We also pass in a Language setting (`1033` for English), as well as the WebTemplate (`sts#0`, which is a Team Site Template). The `success` function will display a success message once the site has been created.

■ **Note** Again, the code in these examples run under the permission level of the current user. Since a lot of these examples perform higher-level functions like creating and deleting SharePoint objects, the user needs to have appropriate permissions to perform these functions. It's worth pointing this out again, as site creation requires the user to have Full Control permissions for the site where the new web is being created.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display textboxes for the user to enter the new site title and description, and display a result message, as shown in Figure 9-18.

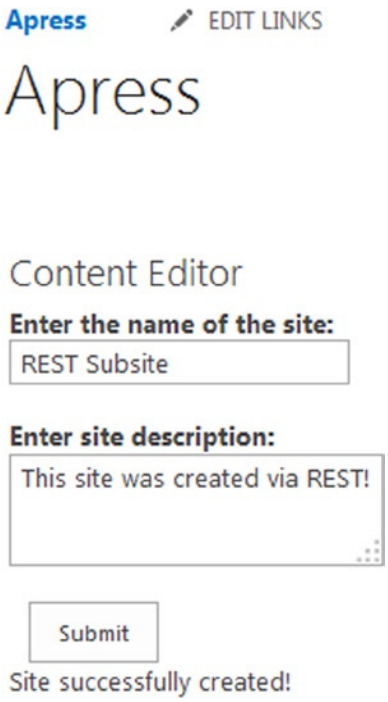

Figure 9-18. *Creating a new site via REST*

We can then open the Site Contents page where the new site was created, and scroll to the bottom. In the Subsites section, we can see that the new site has been created, as shown in Figure 9-19.

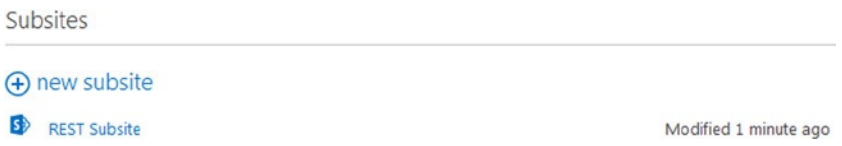

Figure 9-19. *The newly created sub-site in the Site Contents page*

Delete a Site

In this example, we'll delete the site we created in the previous section.

HTML

Create a new HTML file in Visual Studio and name it **RESTDeleteSite.html**. The markup for this page has a textbox for the user to enter the name of the site that they would like to delete. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTDeleteSite.js"></script>

<div id="DeleteFile">
    <div>
        <strong>Enter the name of the site to delete:</strong>
        <br />
        <input type="text" id="txtSiteTitle" />
    </div>
    <br />
    <input type="button" id="btnSubmit" value="Submit" />
</div>
<div id="divResults"></div>
```

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTDeleteSite.js**. Include the following code:

```
$(function () {
    bindButtonClick();
});

function bindButtonClick() {
    $("#btnSubmit").on("click", function () {
        deleteSite();
    });
}
```

```
function deleteSite() {
    var siteTitle = $("#txtSiteTitle").val();
    var siteTitleNoSpaces = siteTitle.replace(/\s/g, "");

    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/" + siteTitleNoSpaces + "/_api/web";

    $.ajax({
        url: fullUrl,
        type: "POST",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val(),
            "X-HTTP-Method": "DELETE",
            "IF-MATCH": "*"
        },
        success: onQuerySucceeded,
        error: onQueryFailed
    });
}

function onQuerySucceeded() {
    $("#divResults").html("Site successfully deleted!");
}

function onQueryFailed(sender, args) {
    alert('Error!');
}
```

Just like in the previous example, we grab the title of the site that the user entered and then remove all the spaces:

```
var siteTitle = $("#txtSiteTitle").val();
var siteTitleNoSpaces = siteTitle.replace(/\s/g, "");
```

Once we have the title and have removed the spaces, we need to construct the REST URL to the site:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/" + siteTitleNoSpaces + "/_api/web";
```

Deleting a site is no different from any of the other delete operations performed so far. All we need to do is include the `"X-HTTP-Method": "DELETE"` and `"IF-MATCH": "*"` headers. The `success` function will display a message to the user, informing them that the delete has taken place.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a textbox for the user to enter the site title, and display a result message, as shown in Figure 9-20.

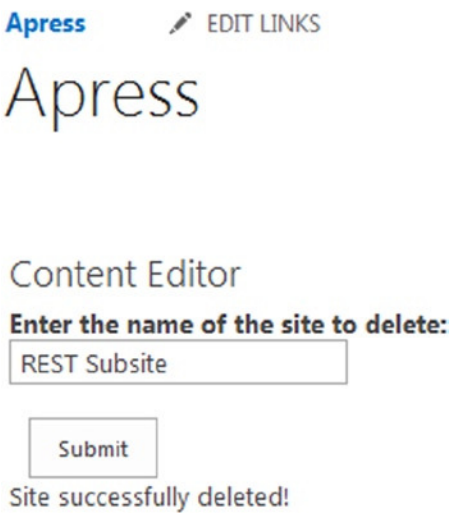

Figure 9-20. *Deleting a subsite via REST*

In this example, we'll delete the site that we just recently created. Navigating back to the Site Contents page will show that the subsite was successfully deleted, as shown in Figure 9-21.

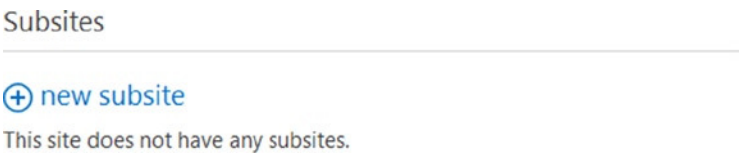

Figure 9-21. *Site has been removed from the root site*

Working with Social Data

Social data is a hot topic in SharePoint 2013 and something you should become familiar with as you begin creating custom solutions. In this section, we'll see how to retrieve some social data and display this to the currently logged-in user.

HTML

Create a new HTML file in Visual Studio and name it **RESTGetSocialData.html**. The markup for this page will include a couple of DIVs and a SPAN to output data to. It consists of the following lines:

```
<script type="text/javascript" src="/apress/Webparts/chapter 9/RESTGetSocialData.js"></script>
<link rel="stylesheet" type="text/css" href="/apress/Webparts/chapter 9/RESTGetSocialData.css" />

<div id="WelcomeMessage">
    Welcome back <span id="WelcomeMessageUserName"></span>
</div>
<div id="UserFollows" />
```

We'll include a custom CSS file titled `RESTGetSocialData.css`. It will be used to style the output from the script. All of the other examples could have easily included CSS for styling. You'll find in your custom development efforts that styling will play a big role in a lot of your web parts, so this example will show you how easy it is to include.

JavaScript

Create a new JavaScript file in Visual Studio and name it **RESTGetSocialData.js**. Include the following code:

```
$(function () {
    getUserProfileInfo();
    getUserFollows();
});

function getUserProfileInfo() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/social.feed/my";

    $.ajax({
        url: fullUrl,
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
        },
        success: onUserInfoQuerySucceeded,
        error: onQueryFailed
    });
}

function getUserFollows() {
    var siteUrl = _spPageContextInfo.webAbsoluteUrl;
    var fullUrl = siteUrl + "/_api/social.following/my/followed(types=15)";

    $.ajax({
        url: fullUrl,
        type: "GET",
        headers: {
            "accept": "application/json;odata=verbose",
            "content-type": "application/json;odata=verbose",
        },
        success: onFollowQuerySucceeded,
        error: onQueryFailed
    });
}

function onUserInfoQuerySucceeded(data) {
        $("#WelcomeMessageUserName").text(data.d.Me.Name);
}
```

```
function onFollowQuerySucceeded(data) {
    var following = data.d.Followed.results;
    var followedItems = "Items you are following:<br />";
    $.each(following, function (index, value) {
        followedItems += "<a href='" + value.Uri + "'>" + value.Name + "</a><br />";

    });
    $("#UserFollows").append(followedItems);
}

function onQueryFailed(sender, args) {
    alert("Error");
}
```

The first thing that should strike you about this example is the introduction of a second function to retrieve data. We have one function to retrieve the user profile information, and another to retrieve the "following" information. This is because the REST endpoints for this data are different, and thus two calls are required to retrieve everything that we wish to show on the page.

■ **Note** In the previous chapter, we discussed how you could "batch" JSOM requests using batch processing. This allows you wrap all your different requests for data into a single call, which improves performance. SharePoint Online and Office 365 supports batch processing when using the REST API via the `$batch` query operator. Using this, you could optimize the performance of this example. However, batch processing using the REST API is an in-depth topic and outside the scope of this book. It's more important that you know that it's available. You can read more about it at http://msdn.microsoft.com/EN-US/library/office/dn903506%28v=office.15%29.aspx.

The first function, `getUserProfileInfo`, is used to retrieve the display name for the user. To get this information, we'll utilize the social feed REST API:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/social.feed/my";
```

This URL looks a bit different from all the other endpoints in the REST API in that is has a `social.feed` as part of the URL. The social feed has two components, Feed and Following, and this format is how we target each component. User information is found under the `my` endpoint and you can find additional information like News, Likes, and so forth, for each user.

■ **Note** The social feed exposes a lot of data about users. In this example, all we'll show is the display name for user, but you have access to much more. You can find more information about what is available at http://msdn.microsoft.com/EN-US/library/office/dn155789%28v=office.15%29.aspx.

The `success` function pulls the display name from the results:

```
$("#WelcomeMessageUserName").text(data.d.Me.Name);
```

The results from the social feed are also a little different than in previous examples. As you can see, we have the familiar `data.d` object, but the user data is further nested under an object named `Me`, from which we can get the `Name` property.

The second function, `getUserFollows`, is used to retrieve all the items that the user is following. To get this information, we'll utilize the `social.following` feed:

```
var siteUrl = _spPageContextInfo.webAbsoluteUrl;
var fullUrl = siteUrl + "/_api/social.following/my/followed(types=15)";
```

We can then obtain all the actors that the user is following using the `followed` endpoint. An *actor* is really just anything the user follows, which can be users, documents, sites, or tags. When you call the `followed` endpoint, you need to pass in a number that corresponds to the actors that you want returned. In this example, we use 15, which indicates all actors.

You can see the full list of values at http://msdn.microsoft.com/en-us/library/microsoft.sharepoint.client.social.socialactortypes(v=office.15).aspx.

■ **Note** The following feed also exposes a lot of data about users. You can find more information at http://msdn.microsoft.com/EN-US/library/office/dn194080%28v=office.15%29.aspx.

The `success` function contains a jQuery loop where we build the HTML to display all the items that the user is following:

```
var following = data.d.Followed.results;
    var followedItems = "Items you are following:<br />";
    $.each(following, function (index, value) {
        followedItems += "<a href='" + value.Uri + "'>" + value.Name + "</a><br />";

    });
    $("#UserFollows").append(followedItems);
```

Much like the user info, the following data is nested inside the `data.d` object in an object called `Followed`. This contains the familiar `results` object, which holds all the items. We create a new variable to hold all the followed items. Next, we loop through each item using a jQuery `each` loop. For each item, we get the Uri, the URL to the item, and the Name, in order to construct a link so that the user can jump directly to the item from the page. Finally, we append all the HTML to the page with the following line:

```
$("#UserFollows").append(followedItems);
```

CSS

Create a new CSS file in Visual Studio and name it **RESTGetSocialData.css**. Include the following code:

```
#WelcomeMessage {
    margin-bottom: 20px;
}
```

```
#WelcomeMessageUserName {
    font-weight: bold;
}

#UserFollows {
    border:dashed 1px #C0C0C0;
    width:240px;
    padding: 6px;
}
```

This CSS is very basic and it is only intended to provide a simple demonstration of styling. We'll apply a bottom margin on the welcome message to the user, and put the user name in bold. The followed items will be wrapped in a dashed border for emphasis.

Results

Set the Content Link property of the Content Editor Web Part to the HTML file and save the page. The page will display a "Welcome back" message to the user, along with their name in bold. Directly underneath is a list of all the items that they are following, displayed as hyperlinks, as shown in Figure 9-22.

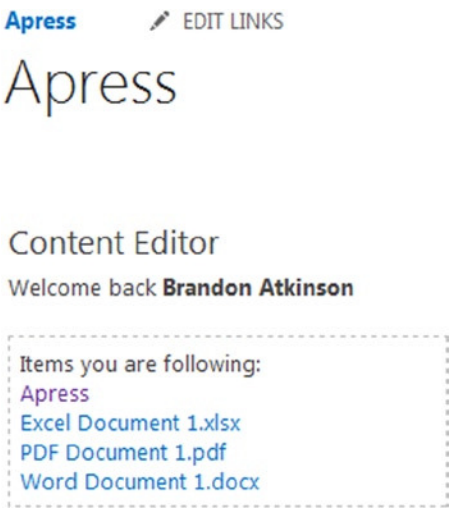

Figure 9-22. *Social data being retrieved via REST*

Summary

In this chapter, we took a deep dive into the REST API and looked at how it can be used to create custom web parts using the Etherson method. We looked at wide variety of examples: querying for list data, creating new lists and sites, and querying for social data about users. Even though this was a long chapter in comparison to others in this book, it only scratched the surface on what is possible with REST in SharePoint. However, after working through all the examples presented here, you should have a very firm grasp on working with the REST API and on how HTML, JavaScript, and CSS all work together in a custom web part. You should now be comfortable enough to begin crafting your own custom solutions using REST.

Index

D

E

F, G, H

I

R

S, T, U

V, W

X, Y, Z

Custom SharePoint Solutions with HTML and JavaScript

For SharePoint 2013 and SharePoint Online

Brandon Atkinson

Apress®

Custom SharePoint Solutions with HTML and JavaScript: For SharePoint 2013 and SharePoint Online

ISBN-13 (pbk): 978-1-4842-0545-7

ISBN-13 (electronic): 978-1-4842-0544-0

Managing Director: Welmoed Spahr
Lead Editor: Gwenan Spearing
Development Editor: Douglas Pundick
Technical Reviewer: Adam Preston
Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Susan McDermott, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Melissa Maldonado
Copy Editor: Kimberly Burton
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is dedicated to the love of my life, Jennie.
You give me more strength and courage than you realize.

Contents

About the Author

Brandon Atkinson is an architect and developer for CapTech Consulting in Richmond, VA. He specializes in Microsoft technologies, including SharePoint and .NET. Brandon has worked in all areas of SharePoint, including architecture, development, installations, configuration, and administration. He is a regular contributor to the SharePoint forums on TechNet, as well as providing code samples to the TechNet Gallery and Codeplex. He has also received the Microsoft MVP award for SharePoint Server. Brandon has authored books and provided technical reviews for other SharePoint titles.

About the Technical Reviewer

Adam Preston is a manager at CapTech Consulting in Richmond, VA. Adam specializes in aligning technology solutions with business goals and strategies. Skilled in IT kstrategy, project management, business consulting, as well as architecting and implementing enterprise solutions, Adam has vast experience across diverse industries, from startups to Fortune 500 companies. He is active in the SharePoint community through having been the president of the Richmond SharePoint User Group, a top SharePoint blogger, and a speaker at numerous SharePoint conferences and the Microsoft Mid-Atlantic CIO Summit.