Third-Party JavaScript

Ben Vinegar
Anton Kovalyov

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Third-Party JavaScript version 10

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1 Introduction to Third-party JavaScript

In this chapter:

- Explanation of third-party JavaScript
- Real world examples of third-party applications
- Walkthrough implementation of a simple embedded widget
- A discussion of challenges facing third-party development

Third-party JavaScript is a pattern of JavaScript programming that enables the creation of highly distributable web applications. Unlike regular web applications, which are accessed at a single web address (http://yourapp.com), these applications can be loaded on any arbitrary web page using simple JavaScript includes.

You've probably already encountered third-party JavaScript before. For example, consider ad scripts, which generate and display targeted ads on publisher websites. Ad scripts might not be a hit with users, but they help web publishers earn revenue and stay in business. They're visible on millions of websites, and yet nearly all of them are actually third-party scripts, served from separate ad servers.

Ad scripts are just one use case; developers look to third-party scripts to solve a number of problems. Some use them to create standalone products that serve the needs of publishers. For example, Disqus, a web startup from San Francisco—and the employers of the fine authors behind this book—develops a third-party commenting application that gives web publishers an instant commenting system. Others develop third-party scripts to extend their traditional web applications to reach audiences on other web sites. For example, Facebook and Twitter have developed dozens of social widgets that are loaded on publisher websites. These

widgets help social networks engage their users outside of their applications' normal ecosystems.

Small companies can benefit from Third-party JavaScript too. Let's say you're the owner of a B2B (Business-to-Business) web application that hosts web forms to collect information from your customers' clients. You have potential customers out there who would love to use your application, but are hesitent to redirect their users to an external website. With third-party JavaScript, you can have customers load your form application directly on their own web pages, solving their redirect concerns.

Third-party JavaScript isn't all gravy, however. Writing these applications is far from trivial. There's a mountain of pitfalls and hackery you'll need to overcome before you can ship third-party JavaScript that will hold its own in the wild. Luckily, this book will show you how by guiding you through the complete development of a full-featured third-party application.

But before we dive into the bowels of third-party JavaScript, you've got to learn the fundamentals. In this chapter, we're going to better define third-party JavaScript, look at real-world implementations from a number of companies, go over a simple implementation of a third-party application, and discuss the numerous challenges facing third-party development.

Let's start with trying to get a better handle on what Third-party JavaScript is and what we can do with it.

1.1 Defining Third-party JavaScript

In a typical software exchange, there are two parties. There is the consumer, or first party, who is operating the software. The second party is the provider or author of that software.

On the web, you might think of the first party as a user who is operating a web browser. When he or she visits a web page, the browser makes a request from a content provider. That provider, the second party, transmits the web page's HTML, images, stylesheets, and scripts from their servers back to the user's web browser.

For a particularly simple web exchange like this one, there might only be two parties. But most website providers today also include content from other sources, or third parties. As illustrated in figure 1.1, third-parties might provide anything from article content (Associated Press), to avatar hosting (Gravatar), to embedded videos (YouTube). In the strictest sense, anything served to the client that is provided by an organization that is not the website provider is considered to be third-party.

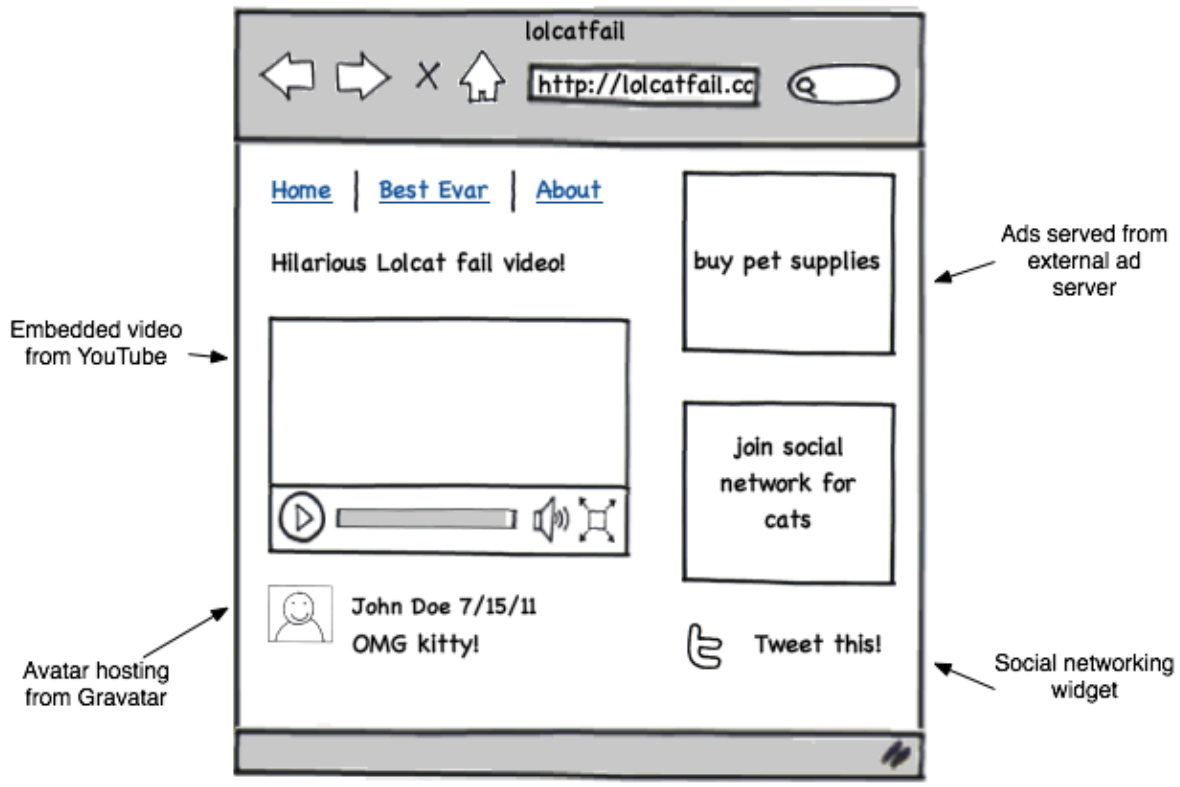

Figure 1.1 Websites today make use of a large number of third-party services

When you try to apply this definition to JavaScript, however, things become muddy. Many developers have differing opinions on what exactly constitutes third-party JavaScript. Some classify it as any JavaScript code a provider doesn't author themselves. This would include popular libraries like jQuery and Backbone.js. It would also include any code you copied and pasted from a programming solutions website like Stack Overflow. Any and all code you didn't write is fair game.

Others refer to third-party JavaScript as code that is being served from third-party servers, not under the control of the content provider. The argument is that code hosted by content providers is under their control: content providers choose when and where the code is served, they have the power to modify it, and they are ultimately responsible for its behaviour. This differs from code served from separate third-party servers, whose contents cannot be modified by the provider, and can even change without notice. Listing 1.1 shows an example content provider HTML page that loads both local and externally hosted JavaScript files.

Listing 1.1 Example content provider HTML page that loads both local and

externally-hosted scripts

```
<!DOCTYPE html>
<html>
   <head>
      <title>Example Content Provider Website</title>

      <script src="js/jquery.js"></script>                    (1)
      <script src="js/app.js"></script>
   </head>
   <body>

      ...

      <script src="http://thirdparty.com/app.js"></script>    (2)
   </body>
</html>
```

(1) Local JavaScript files hosted on the content provider's own servers
(2) JavaScript file loaded from an external (third-party) server

There's no right or wrong answer; you can make an argument for both interpretations. But for the purposes of this book, we're particularly interested in the latter definition. When we refer to third-party JavaScript, we mean code that is:

- not authored by the content provider
- served from external servers that are not controlled by the content provider
- written with the intention that it is to be executed on a content provider's website

NOTE **Where's type="text/javascript"?**
You might have noticed that the script tag declarations in this example don't specify the `type` attribute. For an "untyped" script tag, the default browser behaviour is to treat the contents as JavaScript, even in older browsers. In order to keep the examples in this book as concise as possible, we've dropped the type attribute from most of them.

So far we've been looking at third-party scripts from the context of a content provider. Let's change perspectives. As *developers* of third-party JavaScript, we are authoring scripts that we intend to execute on a content provider's website. In order to get our code onto the provider's website, we give them HTML code snippets to insert into their pages that load JavaScript files from our servers (Figure

1.x). We aren't affiliated with the website provider; we're merely loading scripts on their pages to provide them with helpful libraries or useful self-contained applications.

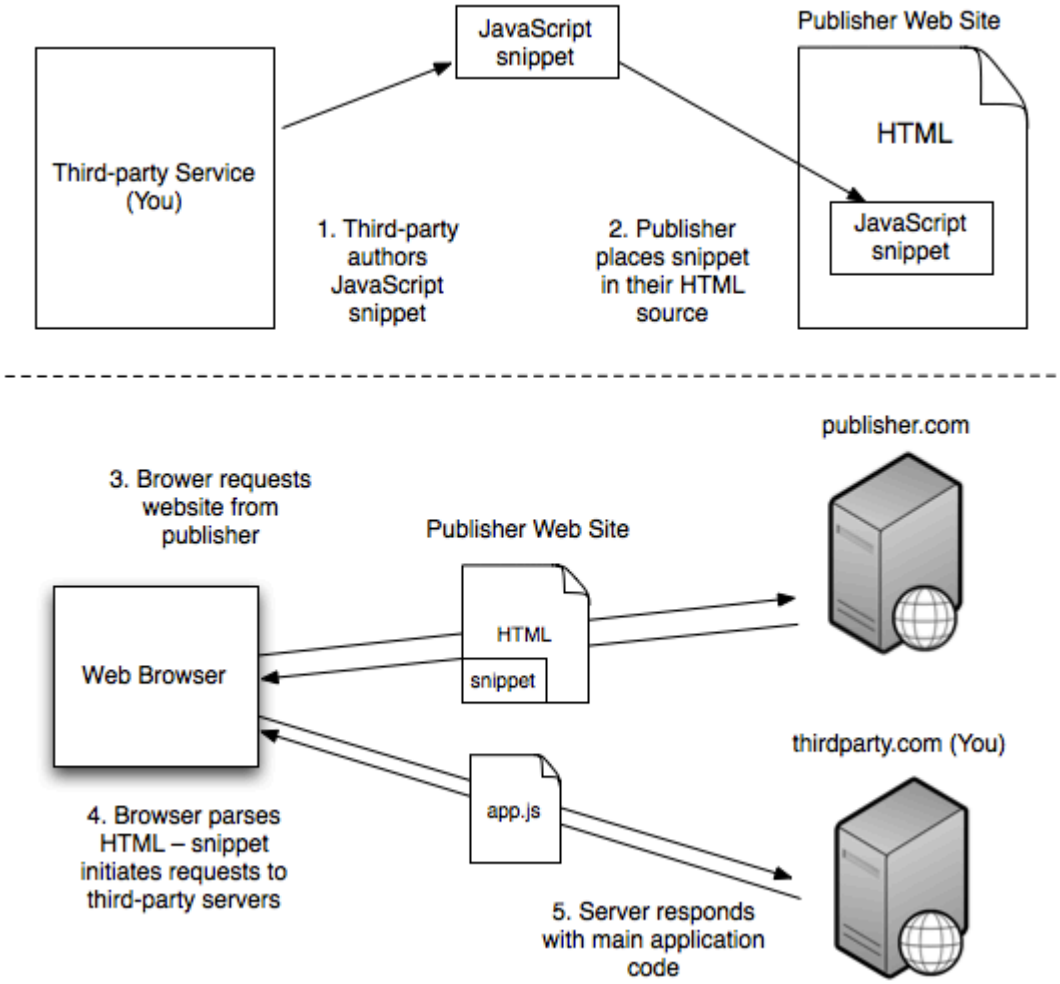

Figure 1.2 A script-loading snippet placed on the publisher's web page loads third-party JavaScript code

If you're scratching your head, don't worry. The easiest way to understand what third-party scripts are is to see how they're used in practice. In the next section, we'll go over some real world examples of third-party scripts in the wild. If you don't know what they are by the time we're finished, then our status as third-rate technical authors will be cemented. Onwards!

1.2 The many uses of third-party JavaScript

We've established that third-party JavaScript is code that is being executed on somebody else's website. This gives third-party code access to that website's HTML elements and JavaScript context. We can then manipulate that page in a number of ways, which might include creating new elements on the DOM (Document Object Model), inserting custom stylesheets, and registering browser events for capturing user actions. For the most part, third-party scripts can perform any operation you might use JavaScript for on your own website or application, but instead, on someone else's.

Armed with the power of remote web page manipulation, the question remains: what is it good for? In this section, we'll look at some real world use cases of third-party scripts:

- *Embedded widgets* – small interactive applications embedded on the publisher's web page
- *Analytics and metrics* – for gathering intelligence about visitors and how they interact with the publisher's properties
- *Web service API wrappers* – for developing client-side applications that communicate with external web services

This isn't a complete list, but should give you a solid idea of what third-party JavaScript is capable of. We'll start with an in-depth look at the first item: embedded widgets.

1.2.1 Embedded widgets

Embedded widgets (oftentimes "third-party widgets") are perhaps the most common use case of third-party scripts. These are typically small, interactive applications that are rendered and made accessible on a publisher's website, but load and submit resources to and from a separate set of servers. Widgets can vary widely in complexity; they can be as simple as a graphic that displays the weather in your geographic location, or as complex as a full-featured instant messaging client.

Widgets enable website publishers to embed applications into their web pages with very little effort. They are typically easy to install; more often than not publishers need only insert a small HTML snippet into their web page source code to get started. Since they're entirely JavaScript based, widgets don't require the

publisher to install and maintain any software that executes on their servers, which means less maintenance and upkeep.

Some businesses are built entirely on the development and distribution of embedded widgets. Earlier we mentioned Disqus, a web startup based out of San Francisco. Disqus develops a commenting widget (figure 1.2) that serves as a drop-in commenting section for blogs, online publications, and other websites. Their product is driven almost entirely by third-party JavaScript. It uses JavaScript to fetch commenting data from the server, render the comments as HTML on the page, and capture form data from other commenters. In other words—everything. It is installed on websites using a simple HTML snippet that totals 5 lines of code.

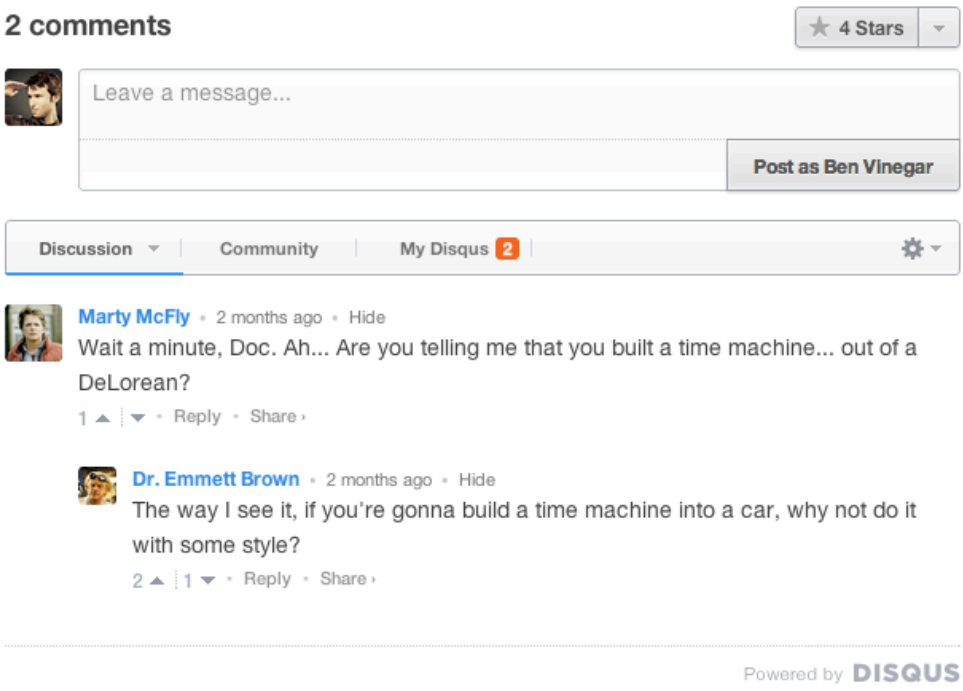

Figure 1.3 An example commenting section on a publisher's website, powered by the Disqus commenting widget

Disqus is an example of a product that is only usable in its distributed form; you'll need to visit a publisher's page to use it. But widgets aren't always standalone products like this. Oftentimes they're “portable" extensions of larger, more traditional stay-at-home web applications.

For example, consider Google Maps, arguably the web’s most popular mapping application. Users browse to maps.google.com to view interactive maps of

locations all over the world. Google Maps also provides directions by car and public transit, satellite imagery, and even street views using on-location photography.

Incredibly, all of this magic also comes in a widget flavor. Publishers can embed the maps application on their own web pages using some simple JavaScript code snippets obtained from the Google Maps website. On top of this, Google provides a set of public functions for publishers to modify the map contents.

Let's see how simple it is to embed an interactive map on our web page using Google Maps (Listing 1.2). This code example begins by first pointing to the Maps JavaScript library using a simple script include. Then, when the body's onload handler fires, we check if the current browser is compatible, and if so, initialize a new map and center it at the given coordinates[1]. We're done, and all it took was roughly 10 lines of code - powerful stuff!

Footnote 1 Not everyone knows latitude and longitude by heart. Luckily, Google has additional functions for converting street addresses to geographical coordinates. Learn more at http://code.google.com/apis/maps

Listing 1.2 Initializing the Google Maps widget

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Maps Example</title>
    <script src="http://maps.google.com/maps?file=api&v=2"></script>

    <script>

    function initialize() {
      if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map_canvas"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
        map.setUIToDefault();
      }
    }

    </script>
  </head>
  <body onload="initialize()">
    <div id="map_canvas" style="width: 500px; height: 300px"></div>
  </body>
</html>
```

We just looked at two examples of embedded widgets. But really, any application idea is fair game for embedding on a publisher's page. In our own

travels, we've come across a wide variety of widgets: content management widgets, widgets that play real-time video, widgets that let you chat in real time with a customer support person. If you can dream it, you can embed it.

1.2.2 Analytics and metrics

Third-party JavaScript isn't used exclusively in the creation of embedded widgets. There are other uses that don't necessarily involve graphical, interactive web page elements. Often they're silent scripts that are processing information on the publisher's page without the user ever knowing they're there. The most common such use case is in analytics and metrics-gathering.

One of JavaScript's most powerful features is that it enables developers to capture and respond to user events as they occur on a web page. For example, one can write JavaScript to respond to a website visitor's mouse movements and/or mouse clicks. Third-party scripts are no exception: they too can observe browser events and capture data about how the visitor interacts with the publisher's page. This might include tracking how long a visitor stays on a page before moving on, what content they saw while they were reading the page, and where they went afterwards. There are dozens of browser events your JavaScript code can hook into, from which you could derive hundreds of different insights.

PASSIVE SCRIPTS

Crazy Egg, another web startup, is one example of an organization that uses third-party scripts in this way. Their analytics product generates visualizations of user activity on your web page (see figure 1.3). To obtain this data, Crazy Egg distributes a script to publishers that captures the mouse and scroll events of web page visitors. This data is submitted back to Crazy Egg's servers, all in the same script. The visualizations Crazy Egg generates help publishers identify which parts of their website are being accessed frequently, and which are being ignored. Publishers use this information to improve their web design and optimize their content.

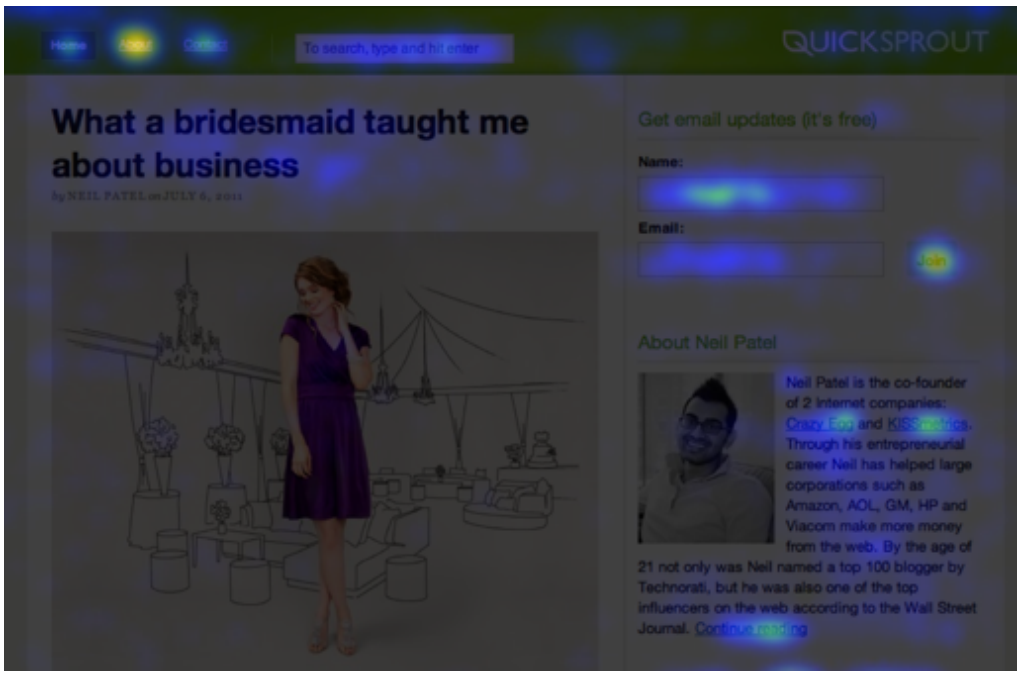

Figure 1.4 CrazyEgg's heatmap visualization highlights trafficked areas of publishers' websites

Crazy Egg's third-party script is considered a "passive" script; it records statistical data without any interaction from the publisher. The publisher is solely responsible for including the script on the page. The rest happens automatically.

ACTIVE SCRIPTS

Not all analytics scripts behave passively. MixPanel is an analytics company whose product tracks publisher-defined user actions to generate statistics about web site visitors or application users. Instead of generic web statistics, like page views or visitors, MixPanel has publishers define key application events they want to track. Some examples events might be "user clicked the signup button", or "user played a video". Publishers write simple JavaScript code (see Listing 1.3) to identify when the action takes place, then call a tracking method provided by MixPanel's third-party scripts to register the event with their service. MixPanel then assembles this data into interesting funnel statistics, to help answer questions like "what series of steps do users take before upgrading the product?

Listing 1.3 Tracking user signups with the MixPanel JS API

```
<button id="signup">Sign up!</button>

<script src="http://api.mixpanel.com/site_media/js/api/mixpanel.js">
</script>

<script>
var mpmetrics = new MixpanelLib(PUBLISHER_API_TOKEN);
```

```
                                                                    (1)
jQuery(function() {                                                 (2)
    jQuery('#signup').click(function() {
        mpmetrics.track("signup button clicked");
    });                                                             (3)
});
</script>
```

(1) Initialize MixPanel library
(2) Attach click event handler to signup button using jQuery
(3) Submit event occurrence using MixPanel library function

Unlike Crazy Egg, MixPanel's service requires some development work by the publisher to define and trigger events. The upside is that the publisher can collect custom data surrounding user actions, and answer different questions about their users' activity.

There's something else interesting about MixPanel's use of third-party scripting. In actuality, MixPanel is providing a a set of client-side functions that communicate with their web service API – a set of server HTTP endpoints that both track and report on events. This is a practical use case that can be extended to any number of different services. Let's learn more.

1.2.3 Web service API wrappers

In case you're not familiar with them, web service APIs are HTTP server endpoints that enable programmatic access to a web service. Unlike server applications that return HTML to be consumed by a web browser, these endpoints accept and respond with structured data—usually in JSON or XML formats—to be consumed by a computer program. This program could be a desktop application, an application running on a web server, or it can even be client JavaScript code hosted on a web page but executing in a user's browser.

This last use-case – JavaScript code running in the browser – is what we're most interested in. Web service API providers can give developers building on their platform – often called *integrators* – third-party scripts that simplify client-side access to their API. We like to call these scripts "web service API wrappers", since they are effectively JavaScript libraries that "wrap" the functionality of a web service API.

EXAMPLE: THE FACEBOOK GRAPH API

How is this useful? Let's look at an example. Suppose there's an independent web developer named Jill, who's tired of freelance work and looking to score a full-time job. Jill's decided that in order to better appeal to potential employers, she needs a terrific-looking online resume hosted on her personal website. This resume is for the most part static – it lists her skills, her prior work experience, and even mentions her fondness for moonlight kayaking.

Jill's decided that, in order to demonstrate her web development prowess, there ought to be a dynamic element to her resume as well. And she's got the perfect idea. What if visitors to Jill's online resume—potential employers—could see if they had any friends or acquaintances in common with Jill (Figure 1.x)? Not only would this be a clever demonstration of Jill's skills, but having a common friend could be a great way at getting her foot in the door.

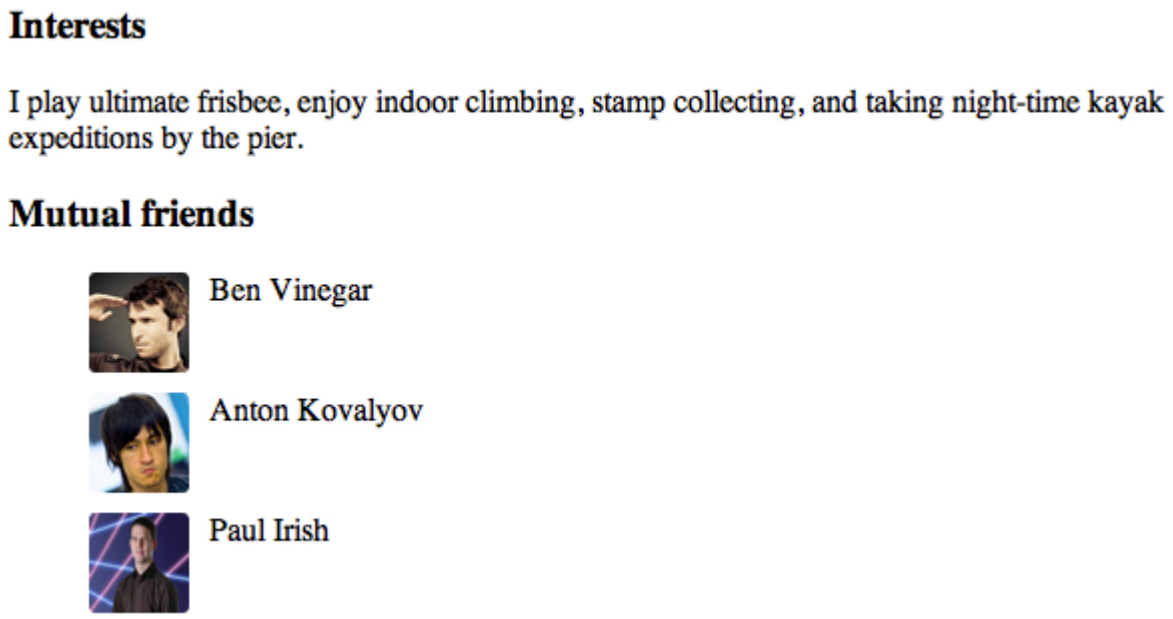

Figure 1.5 At the bottom of Jill's resume, the visitor can see mutual friends between themselves and Jill.

To implement her dynamic resume, Jill will use Facebook's Graph API. This is a web service API from Facebook that enables software applications to access or modify live Facebook user data (with permission, of course). Facebook also has a JavaScript library that provides functions for communicating with the API. Using this library, it's possible for Jill to write client-side code that can find and display common friends between herself and a visitor to her resume (Listing 1.x).

Listing 1.4 Using Facebook's Graph API to fetch and display a list of mutual friends

```
<!DOCTYPE html>
<html>
  <!-- rest of resume HTML above -->

  <a href="#" id="show-connections">Show mutual friends</a>

  <ul id="mutual-friends">
  </ul>

  <div id="fb-root"></div>

  <script src="/js/jquery.js"></script>                          (1)
  <script src="http://connect.facebook.net/en_US/all.js"></script>

  <script>
  FB.init({ appId: 'FACEBOOK_APP_ID' });                         (2)

  $('#show-connections').click(function() {
    FB.login(function(response) {                                (3)
      var userID;
      var url;
      if (response.authResponse) {                               (4)
        userID = response.authResponse.userID;
        url = '/' + userID + '/mutualfriends/jill?fields=name,picture';
        FB.api(url, showMutualFriends);
      }
    });
  });

  function showMutualFriends(response) {                         (5)
    var out = '';
    var friends = response.data;
    friends.forEach(function (friend) {
      out += '<li>';
      out += '<img src="' + friend.picture + '"/>';
      out += friend.name + '</li>';
    });
    $('#mutual-friends').html(out);
  }
  </script>
</html>
```

(1) Load both jQuery and the Facebook JavaScript SDK. jQuery is a popular DOM library that helps keep this example small. Learn more at http://jquery.com.

(2) Initialize the Facebook JavaScript SDK. You need to register your application at http://developers.facebook.com and obtain an application ID first.

(3) FB.login opens up a new Facebook window asking the website visitor to both log in, and grant the application permission to access the visitor's data.

(4) If the login was successful, request the visitor's mutual Facebook friends using the Graph API's /mutualfriends/ endpoint. When the response is ready, execute the showMutualFriends callback function.

(5)

9 Itereate through the list of friends and render them to the page.

Jill managed to embed some very powerful functionality on her resume, all using a small amount of client-side JavaScript. With this impressive piece of work, she should have no problem landing a top-flight software job.

BENEFITS OF CLIENT-SIDE API ACCESS

It's worth pointing out that this entire example could have been done without client-side JavaScript. Instead, Jill could have written a server application to query the Facebook Graph API for the necessary data, then render the result as HTML in the response to the browser. In which case the browser solely downloads the HTML from Jill's server and displays the result to the user – no JavaScript is executed.

But it's arguably better to do have the website visitor perform this work in the browser, for a few reasons:

- Code executing in the browser is code that's not executing on the integrator's servers, which can lead to bandwidth and CPU savings
- It's faster – the server implementation has to wait for the response from Facebook's API before showing any content
- Some websites are completely static, such that client-side JavaScript is their only means of accessing a web service API

NOTE **An API for every season**

This example we just covered might be regarded as a niche use-case, but this is just one possible application. Facebook is just a single web service API provider, but the reality is that there are thousands of popular APIs, all of which provide access to varying data and or functionality. Besides social networking applications like Facebook, Twitter, and LinkedIn, there are publishing platforms like Blogger and Wordpress, or search applications like Google and Bing, all of which provide varying degrees of access to their data via APIs.

Many web services—large and small—offer APIs. But not all of them have gone the extra mile of providing a JavaScript library for client-side access. This matters because JavaScript in the browser is the single largest development platform: it's supported on every website, in every browser. If you or your organization develops or maintains a web service API, and you want to reach the

largest number of possible integrators possible, you owe it to yourself to provide developers with a client-side API wrapper – a topic we'll discuss in detail later in this book.

1.3 Developing a bare-bones widget

We've just explored some popular uses for third-party JavaScript. You've seen how it can be used in the development of widgets, in analytics gathering, and as a client-side wrapper for web service APIs. Hopefully this has given you an idea of what's possible when you're designing your own third-party application.

Now that you've walked through some real world examples, it's time to develop something yourself. Let's start with something fairly simple; a bare-bones embedded widget.

Pretend for a moment that you run a website that provides up-to-the-minute local weather information. Traditionally, users visit your website directly in order to get the latest weather news. But in order to reach a wider audience, you've decided to go a step further and give users access to your data outside of your website. You'll do this by providing an embeddable widget version of your service, depicted in figure 1.4. You'll market this widget to publishers who are interested in providing their readers with local weather information, with the easy installation of a third-party script.

Figure 1.6 The weather widget as it will appear on a publisher's page

Luckily, you've already found a publisher who's interested, and they've signed on to test-drive your widget. To get them started, you'll need to provide them with an HTML snippet that will load the weather widget on their web page. The publisher will copy and paste this snippet into their HTML source code at the

location where they want the widget to appear. The snippet itself is simple: it's just a script tag pointed to a third-party JavaScript file hosted on your servers at weathernearby.com.

```
<script src="http://weathernearby.com/widget.js?zip=94105">
</script>
```

You'll notice the URL for this script element contains a single parameter, "zip". This is how you'll identify what location to render weather information for.

Now when the browser loads the publisher's web page, it will encounter this script tag and request widget.js from your servers at weathernearby.com. When widget.js is downloaded and executed, it will render the weather widget directly into the publisher's page. That's the goal, at least.

To do this, widget.js will need to have access to the company's weather data. This data could be published directly into the script file, but given that there are approximately 43,000 US zip codes, that's too much data to serve in a single request. Unless you live in Sweden or South Korea, where 100 mbps connections are the norm, it's clear that the widget will need to make separate requests for the weather data. This is typically done using AJAX, but for simplicity we're going to use a different approach: server-side script generation.

1.3.1 Server-side JavaScript generation

Instead of serving a static JavaScript file that contains your widget code, you will write a server application that generates a JavaScript file for every request (Figure 1.x). Because your server has access to your weather database, it can inject the requested weather data into the outputted JavaScript file. This means that the JavaScript file will contain all the code and data necessary to render the weather widget on the publisher's page, without having to make any additional requests.

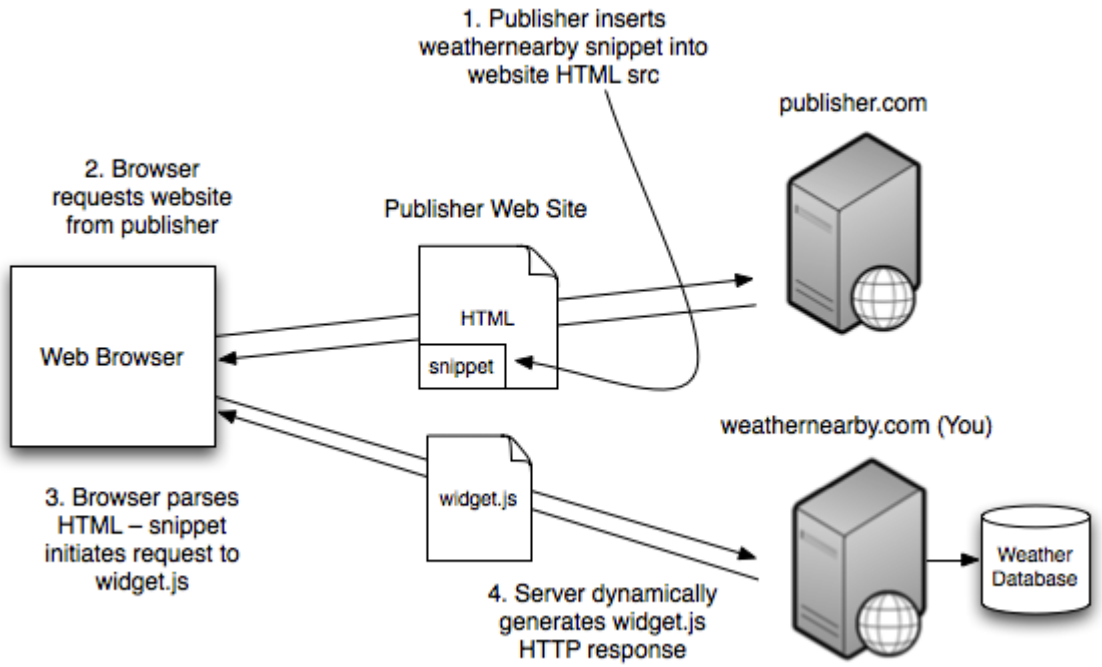

Figure 1.7 A server application dynamically generating the Weather Nearby widget code (widget.js).

This server application could be written in any programming language or platform that can execute in a server environment, like Ruby, Python, Java, ASP.NET – even server-side JavaScript. These are all fine choices, but we're going to walk you through an example written in Python, a popular scripting language. This example also uses Flask, a Python microframework for building small web applications. If you're not familiar with Python, don't sweat it – the code is easy to follow. If you'd like to try the example yourself, consult the companion source code, which also contains instructions for installing both Python and Flask.

NOTE **Why Python?**
The small handful of server-side examples in this book are written in Python. Our reason for choosing this programming language is completely biased: it's what we use every day at Disqus, and we're most familiar with it. If you don't know how to write Python, that's okay. This is first and foremost a book about JavaScript and the server-side examples could easily be re-written in any language.

Listing 1.5 server.py – Server implementation of widget.js, written in Python and Flask

```
# server.py

from flask import Flask, make_response                    (1)
```

```
from myapp import get_weather_data

app = Flask(__name__)                                                    (2)

@app.route('/widget.js')                                                 (3)
def weather_widget():

    zip = request.args.get('zip')                                        (4)
    data = get_weather_data(zip)

    out = '''                                                            (5)
        document.write(
          '<div>' +
          '    <p>%s</p>' +
          '    <img src="%s"/> ' +
          '    <p><strong>%s &deg;F</strong> &mdash; %s</p>' +
          '</div>'
        );
    ''' % (data['location'], data['image'], data['temp'], data['desc'])

    response = make_response(out)
    response.headers['Content-Type'] = \
        'application/javascript'                                         (6)

    return response
```

(1) This short example is actually a full-fledged Flask application. It starts by importing a few helper libraries, including Flask, and a utility function for querying the weather database.
(2) Initialize the Flask application.
(3) Define a single route: '/widget.js'. When the server starts, Flask will listen to all requests for /widget.js and respond by executing this function.
(4) Extract the "zip" parameter from the request's query string, and query the weather database for the corresponding weather data.
(5) Assemble a multi-line string of JavaScript code that will render the widget's contents on the publisher's page. This uses JavaScript's document.write function, which writes a string of text (in this case HTML) directly into the page.
(6) Create an HTTP response object to return the string to the browser. Set the response's Content-type header to application/javascript so that it is interpreted by the browser as JavaScript code.

Once this server endpoint is up and running, a <script> request to http://weathernearby.com/widget.js?zip=94105 should return the following JavaScript code. This renders the sample widget you saw at the beginning of this section (Figure 1.4). Also, note that the fact this code is served from a Python application is completely transparent to the requesting browser.

```
document.write(
    '<div>' +
    '  <p>San Francisco, CA</p>' +
    '  <img src="http://weathernearby.com/img/partly-cloudy.png"/>' +
```

```
    '  <p><strong>87 &deg;F</strong> &mdash; Partly Cloudy</p>' +
    '</div>'
);
```

Now, when we said this would be a bare-bones example earlier, we weren't kidding. This outputs a completely unstyled weather widget that offers absolutely no interaction with the user. It looks awful, and has probably put your fledgling weather company in jeopardy. But, it works, and illustrates the interaction between publisher web sites and third-party code.

Some of the techniques illustrated here, like using `document.write` and server-side Python, are not the only ways to generate widgets. And for reasons we'll explain later, they're even frowned upon. In future chapters, we'll explore alternate, better solutions, and tackle more complicated features like stylesheets, server communication via AJAX, and user sessions.

1.3.2 Distributing widgets as iframes

If you're fairly experienced with web development, you might be thinking to yourself, "isn't it easier to distribute this widget as an iframe?" At first blush it might seem so, but there are not-so-obvious differences that make third-party scripts the better implementation choice. In order to understand why, let's first see how one can recreate the example widget above using iframes.

In case you're not familiar with them, iframes are block-level HTML elements designed to embed external content served from a URL. One could easily recreate the weather widget example using strictly an iframe element, like so:

```
<iframe style="border:none;height:200;width:150"
 src="http://weathernearby.com/widget.html?zip=94105"/>
```

You'll notice the target `src` attribute has changed: it's no longer pointing to a JavaScript file, but instead an HTML document. This time your server endpoint will return a fully-formed HTML document containing the widget markup, completely avoiding the need for JavaScript. You'll also notice the dimensions of the widget are provided as part of the iframe's style attribute. Iframe elements don't expand to fit their contents, and need to be given explicit dimensions.

Using an iframe like this should produce the same output as the JavaScript example. So why use JavaScript over iframes, even for a simple example like this? There are many reasons, most of which revolve around a particular attribute of iframes: that external content loaded inside an iframe *cannot be accessed* by the parent page (i.e. the publisher's page), and vice versa.

- *Flexibility*. If you ever want to change the dimensions of the widget, you will be out of luck. Since the iframe dimensions are fixed on the iframe element on the parent page, and those attributes cannot be modified from content loaded inside the iframe, there is no way to dynamically resize the widget.
- *Aesthetics*. The look and feel of the widget will need to be completely independent from the parent page's styles. The widget will not be able to inherit basic styles, like font family, size, or color.
- *Interaction*. Will the widget need to read or modify the publisher's DOM? What if the publisher needs to interact with the contents of the widget? Could multiple instances of the widget communicate with each other? None of these are possible with static iframes.
- *Reporting*. Did the browser user actually view the widget? How much time did they spend on the page before viewing it? Retrieving this and other valuable statistics requires JavaScript running on the publisher's page.

These are just a few examples, but you're probably beginning to see a trend. Iframes may be the simplest mechanism for distributing the weather widget example, but in doing so you will lose many compelling abilities offered by third-party JavaScript. But don't let this sour your opinion of iframes. They are an *invaluable* tool in the third-party JavaScript developer's toolset, and we'll use them frequently for a number of different tasks over the course of this book.

1.4 Challenges of third-party development

You've learned how third-party JavaScript is a powerful way to write highly distributable applications. However, writing scripts that run on other peoples' websites carries a unique set of challenges that distinguish it from regular JavaScript programming. Namely, that it’s being executed in a DOM environment that you don’t control, on a different domain. This means you have to contend with a number of unexpected complexities, like an unknown web page context, a JavaScript environment shared with other first and third-party scripts, and even restrictions put in place by the browser. We'll take a quick look at what each of these entails.

1.4.1 Unknown context

When a publisher includes your third-party script on their web page, often you know very little about the context in which it's being placed. Your script might get included on pages that sport a variety of different doctypes, DOM layouts, and CSS rules, and ought to work correctly in all of them.

You have to consider that a publisher might include your script at the top of their page, in the `<head>` tag, or they might include it at the bottom of the `<body>`. Publishers might load your application inside of an iframe, or on a page where the head tag is entirely absent; in HTML5, head sections are optional, and not all browsers automatically generate one internally. If your script makes assumptions about these core elements when querying or appending to the DOM, it could wind up in trouble.

If you're developing an embedded widget, displaying proper styles also becomes a concern. Is the widget being placed on a web page with a light background, or a dark background? Do you want your widget to inherit styles and "blend" into the publisher's web design, or do you want your widget to look identical in every context? Solving these problems requires more than well-written CSS. We'll cover solutions to these issues in later chapters.

1.4.2 Shared environment

For a given web environment, there's only one global variable namespace, shared by every piece of code executing on the page. Not only must you take care not to pollute that namespace with your own global variables, you have to recognize that other scripts, possibly other third-party applications like yours, have the capability of modifying standard objects and prototypes that you might depend on.

For example, consider the global `JSON` object. In modern browsers, this is a native browser object that can parse and stringify JSON (JavaScript Object Notation) blazingly fast. Unfortunately, it can be trivially modified by anyone. If your application depends on this object functioning correctly, and it is altered in an incompatible way by another piece of code, your application might produce incorrect results or just plain crash.

The following sample code illustrates just how easy it is to modify the global `JSON` object by using simple variable assignment.

```
JSON.stringify = function() {
    /* custom stringify implementation */
};
```

You might think to yourself - why would anyone do such a thing? Web developers often load their own JSON methods to support older browsers that don't provide native methods. The bad news is that some of these libraries are incompatible in subtle ways. For example, older versions of the popular Prototype JavaScript library provide JSON methods that produce different output than native methods when handling undefined values.

```
// Prototype.js
JSON.stringify([1, 2, undefined])
=> "[1, 2]"

// Native
JSON.stringify([1, 2, undefined])
=> "[1, 2, null]"
```

The `JSON` object is just one example of a native browser object that can be altered by competing client code; there are hundreds of others. Over the course of this book we'll look at solutions for restoring or just plain avoiding these objects.

Similarly, the DOM is another global application namespace you have to worry about. There's only one DOM tree for a given web page, and it's shared by all applications running on the page. This means taking special care when you interact with it. Any new elements you insert into the DOM have to co-exist peacefully with existing elements, and not interfere with other scripts that are querying the DOM. Similarly, your DOM queries can inadvertently select elements that don't belong to you if they're not scoped properly. The opposite is also true; other applications might accidentally query your elements if you haven't carefully chosen unique ids and class names.

Since your code exists in the same execution environment as other scripts, security also becomes a taller order. Not only do you have to protect against improper use by users of your application, you also have to consider other scripts operating on the page, or even the publisher themselves to be a potential threat. For example, if you're writing a widget or script that ties to a larger, popular service, like a social networking website, publishers might have a vested interest in attempting to fake user interactions with their own pages.

1.4.3 Browser restrictions

If an unknown document context, multiple global namespaces, and additional security concerns weren't bad enough, web browsers actively prohibit certain actions that often directly affect third-party scripts. For example, AJAX has become a staple tool of web developers for fetching and submitting data without refreshing the page. But the web browser's Same-Origin Policy prevents `XmlHttpRequest` from reaching domains other than the one you're on (figure 1.5). If you're writing a third-party script that needs to get or send data back to an application endpoint on your own domain, you're going to have to find other ways to do it.

Figure 1.8 Failed cross-domain AJAX request from localhost to google.com in Google Chrome

In the same vein, web browsers also commonly place restrictions on the ability of applications to set or even read third-party cookies. Without them, your users won't be able to "log in" to your application, or remember actions between subsequent requests. Depending on the complexity of your application, not being able to set or read third-party cookies can be a real hindrance.

Unfortunately, this list of challenges is really just the tip of the iceberg. Third-party JavaScript development is fraught with pitfalls, because at the end of the day, the web browser wasn't built with embedded applications and third-party code in mind. Browsers are getting better, and new features are being introduced that alleviate some of the burden of doing third-party development, but it's still an uphill battle, and supporting old browsers is typically a must for any kind of distributed application.

But don't worry. You have already made the fine decision of purchasing this book, which will cover the problems facing your third-party JavaScript code. And as an added bonus, we'll even tell you how to to solve them.

1.5 Summary

Third-party JavaScript is a powerful way of building embedded and highly distributable web applications. These applications can come in many shapes and sizes, but we looked at three specific use cases: as interactive widgets, as passive scripts that collect data, and as developer libraries that communicate with third-party web APIs. But compared to developing regular stay-at-home web applications, third-party scripts face additional challenges. They require you to execute your code in an unknown, shared, and potentially hostile browser environment.

We've really only scratched the surface of what it means to write third-party scripts. In the next chapter we'll hit the ground running by covering the front-to-back creation of an embedded widget. This is one of the most common use cases for third-party JavaScript, and serves as an excellent starting place for covering third-party concepts and challenges.