SAMPLE CHAPTER

Dave Crane, Bear Bibeault, and Jord Sonneveld

with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Ajax IN PRACTICE

- GET GOING
- GET SAVVY
- 60 PROBLEMS SOLVED

MANNING

Ajax in Practice

by Dave Crane
Bear Bibeault
Jord Sonneveld
with Ted Goddard, Chris Gray,
Ram Venkataraman and Joe Walker

Sample Chapter 2

brief contents

2 How to talk Ajax

This chapter covers

- Identifying the main dialects of Ajax
- Using Ajax with JavaScript and JSON
- Working with XML, XPath, and web services

Ajax is a melting pot, with many different approaches to application design and a myriad of dialects. In chapter 1 we looked at the mechanics of the XHR object that lies at the heart of Ajax, and you saw how to wrap those details up in a helper object. Without the distraction of having to write all that plumbing code by hand, we can focus on the more interesting issues of structuring the communication between the server and the client. We can now look at the different categories of Ajax communication, and teach you how to talk Ajax in a range of dialects.

We'll continue to work with the Hello World example that we introduced in chapter 1 and the problem/solution format. We'll also continue to use frameworks to handle the low-level Ajax concerns for us and free us up to look at the interesting issues. Many of the examples will continue to use Prototype's Ajax. Request object, but we'll also take a look at Sarissa and a web services client toolkit. Let's begin by looking at JavaScript as a medium of communication.

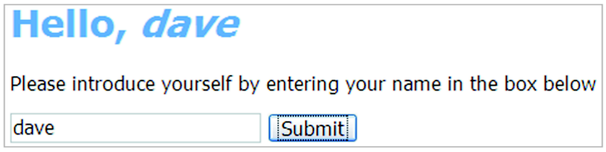
2.1 Generating server-side JavaScript

When the server returns HTML from an Ajax request, we can generate complex user interfaces on the fly, but they remain largely static. Any significant interaction with the application will require further communication with the server. In many cases, this isn't a problem, but in others, it is necessary to deliver behavior as well as content. All client-side behavior is driven by JavaScript, so one way forward is for the server to generate JavaScript for us.

2.1.1 Evaluating server-generated code

When handling server-generated HTML, we can go a long way using only `innerHTML`. When handling server-generated JavaScript, we can make similar use of the `eval()` method. JavaScript is an interpreted language, and any snippet of text is a candidate for evaluating as code. In the next example, we'll see how to use `eval()` as part of the Ajax-processing pipeline.

We'll stick with our Hello World app through this chapter. In this first example, we'll use the response to modify the title element again, as shown in figure 2.1.


Hello, *dave*

Please introduce yourself by entering your name in the box below

dave

Figure 2.1
Result of evaluating server-generated JavaScript

Problem

The server is returning JavaScript code from an Ajax request. We need to run the code when we receive it.

Solution

Using `eval()` is almost as simple as using `innerHTML`. Listing 2.1 presents the third incarnation of our Hello World application.

Listing 2.1 hello3.html

```
<html>
<head>
<title>Hello Ajax version 3</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick=function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello3.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          eval(xhr.responseText);       <-- 1 Evaluates the response
        }
      }
    );
  };
};
</script>
</head>
<body>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</body>
</html>
```

Comparing our earlier solution with listing 2.1, you can see that we've had to change very little. We can still read the response body using the `responseText` property of the XHR object, which we then pass straight to `eval()` ❶.

We want to modify the title block of the page when the response comes in. To do this, we need to perform a bit of DOM manipulation. The method calls can be generated directly on the server, as shown in listing 2.2.

Listing 2.2 hello3.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
%>
document.getElementById('helloTitle').innerHTML =
  "<h1>Hello, <b><i>"+name+"</i></b></h1>";
```

Generally, it is good manners to set the MIME type, but we've switched it off here because Prototype is clever enough to recognize the `text/javascript` MIME type and would evaluate it automatically for us. Here we want to do the evaluation manually in order to demonstrate some general principles, not show off Prototype's power-user features!

Discussion

In this example, we've demonstrated the principle of passing JavaScript from the server to the client, but in the process we've raised a few interesting problems. We'll fix these up in the next example, but first let's examine them.

The first problem is that we've created a very tight coupling between the client and server code. The JSP needs to know the `id` attribute of the DOM element that it is going to populate. If we change the HTML on the user interface, we need to alter the server code. In a small example like this one, that's not too great a burden, but it will quickly become unscalable.

Second, we've created a solution looking for a problem. We aren't doing anything here that we couldn't do more elegantly and simply using `innerHTML`. As long as we're using the response to update a single element on the page, this approach is overkill.

In the next example, we're going to address both these points and see how to reduce the coupling across the tiers as well as update several elements at once.

2.1.2 Utilizing good code-generation practices

When we generate JavaScript on the server, we are practicing code generation. Code generation is an interesting topic in its own right, with a well-established set of conventions and ground rules. A cardinal rule of code generation is to always generate code at the highest level possible.

In the next example, we're going to increase the complexity of our Hello World application a little and demonstrate how to tighten up our code generation to cope with it.

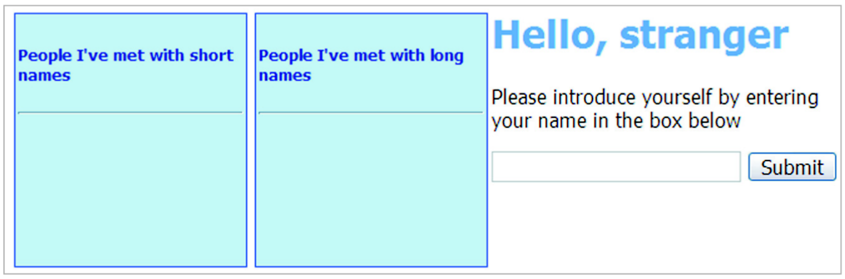
Problem

Writing low-level JavaScript on the server leads to unacceptable tight coupling between the server and client codebases. This will give our application severe growing pains and lead to increased brittleness.

At the same time, we want to maintain a list of previous visitors to our page, as well as display the name of the current visitor. The server is going to classify visitors' names as either long or short, and we'll provide a separate list for each (once again, this is a surrogate for real business logic, because we want to keep the server code simple in this chapter).

We'll also pop up an alert message when the data comes in. The revised UI for the Hello World app is shown in figure 2.2.

Every time the form is submitted to the server, the most recent name will be displayed in the title element, as before. We'll also keep a running list of visitors in the box elements on the left. Figure 2.3 shows our version 4 Hello World in action, after several interesting visitors have passed by!

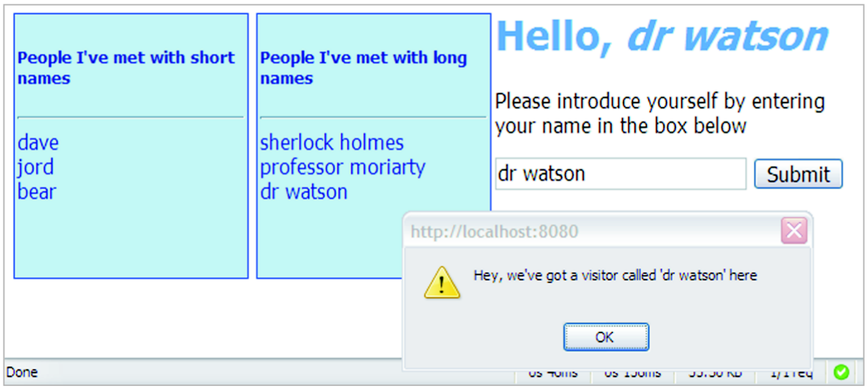

People I've met with short names

People I've met with long names

Hello, stranger

Please introduce yourself by entering your name in the box below

Figure 2.2 Expanded UI for version 4 of Hello World, with a list of previous visitors alongside the form


Hello, *dr watson*

People I've met with short names

dave
jord
bear

People I've met with long names

sherlock holmes
professor moriarty
dr watson

Please introduce yourself by entering your name in the box below

dr watson

Hey, we've got a visitor called 'dr watson' here

Figure 2.3 Hello World version 4 after several visits. Here we see a modified title, an updated list to the left, and an alert message, all from a single server-generated call.

Solution

When the response comes back from the server, we want to update the client with the new information. The code that the server is sending us is simply a carrier for some data, so we will simplify the server-generated JavaScript to call a single `updateName()` function, passing in the data as arguments.

On the client side, we need to define that `updateName()` function as handwritten JavaScript, as shown in listing 2.3. `updateName()` will handle all of our expanded requirements.

Listing 2.3 hello4.html

```
<html>
<head>
<title>Hello Ajax version 4</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
```

```
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello4.jsp?name = "+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          eval(xhr.responseText);      <-- Evaluates response
        }
      }
    );
  };
};

function updateName(name,isLong){      <-- Defines API
  $('helloTitle').innerHTML=
    "<h1>Hello, <b><i>"+name+"</i></b></h1>";
  var listDivId=(isLong)
    ? 'longNames' : 'shortNames';
  $(listDivId).innerHTML+=name+"<br/>";
  alert("Hey, we've got a visitor called '"
    +name+"' here");
}

</script>
</head>
<body>

<div id='shortNames' class='sidebar'>
<h5>People I've met with short names</h5><hr/>
</div>
<div id='longNames' class='sidebar'>
<h5>People I've met with long names</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

In spite of the increased complexity of our requirements, the server-side code has become simpler. Listing 2.4 shows the JSP used to serve data to version 4 of our app.

Listing 2.4 hello4.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
boolean isLong = (name.length() > 8);
%>
updateName("<%= name %>",<%= isLong %>);
```

The JSP is simpler in terms of length of code, but also in the number of concepts. It is only concerned with the business logic appropriate to the server and talks to the client via a high-level API.

Discussion

With this example, we've crossed an important threshold and need to update multiple regions of the UI from a single Ajax call. At this point, the simple `innerHTML` approach that we used in example 2 can no longer suffice. In this case, the requirements were somewhat artificial, but in many real-world applications, multiple update requirements exist. For example:

- In a shopping cart, adding a new item will result in adding a row to the cart body, and updating the total price, and possibly shipping costs, estimated shipping date, and so on.
- Updating a row in a data grid may require updates to totals, paging information, and so on.
- A two-pane layout in which a summary list is shown on the left and drill-down details of the selected item on the right will have a tight interdependency between the two panes.

We solved the multiple-update issue in this case by defining a JavaScript API and generating calls against that API. In this case, we defined one API method and called it only once, but a more sophisticated application might offer a handful of API calls and generate scripts consisting of several lines. As long as we stick to the principle of talking in conceptual terms, not in the details of DOM element IDs and methods, that strategy should be able to work for us as the application grows.

An alternative to generating API calls on the server is to generate raw data and pass it to the client for parsing. This opens up a rich field, which we'll spend the remainder of this chapter exploring.

2.2 Introducing JSON

We began our exploration of Ajax techniques in chapter 1 by doing all the processing on the server and sending prerendered HTML content to the browser. In section 2.1, we looked at JavaScript as an alternative payload in the HTTP response. The crucial win here was that we were able to update several parts of the screen at once. At the same time, we were able to maintain a low degree of coupling between the client-side and server-side code.

If we follow this progression further, we can divide the responsibilities between the tiers, such that only business logic is processed server-side and only application workflow logic on the client. This design resembles a thick-client architecture, but without the downside of installing and maintaining the client on client PCs.

In this type of design, the server would send data to the client—potentially complex structured data. As we noted at the beginning of this chapter, we have a great deal of freedom as to what form this data can take. There are two strong contenders at the moment: JavaScript Object Notation (JSON) and XML. We'll begin to explore data-centric Ajax in this section with a look at JSON.

A one-minute JSON primer

Before we dive into any examples, let's quickly introduce JSON. JSON is a lightweight data-interchange format that can be easily generated and parsed in many different server-side technologies and in JavaScript. A complete data-interchange format will provide two-way translation between the interchange format and live objects, as illustrated in figure 2.4.

Half of JSON is provided for free as part of the JavaScript language specification, and the other half is available as a third-party library. That sounds like an unusual state of affairs, so let's explain what we mean by it.

First, let's look at what a JSON definition looks like. The following example defines a variable `customers` and stores in it an array attribute called `details`. Each array element is a collection of attributes of each customer object. Each customer object has three attributes: `num`, `name`, and `city`.

```
var customers = { "details": [
                  {"num": "1","name":"JBoss","city":"Atlanta"},
                  {"num": "2","name":"Red Hat","city":"Raleigh"},
                  {"num": "3","name":"Sun","city":"Santa Clara"},
                  {"num": "4","name":"Microsoft","city":"Redmond"}
              ]
          } ;
```

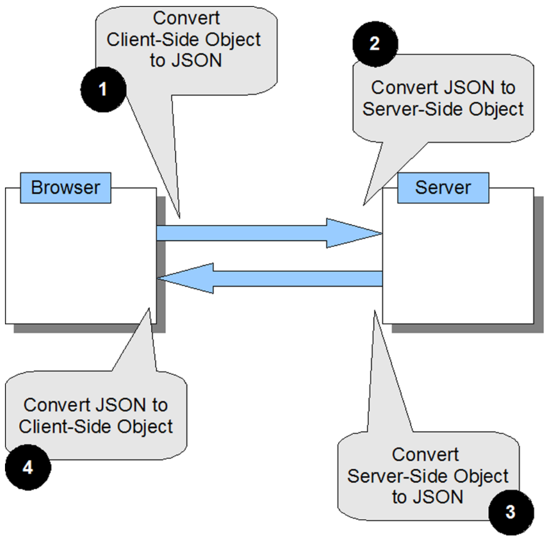

Convert Client-Side Object to JSON
1
2
Convert JSON to Server-Side Object
Browser
Server
Convert JSON to Client-Side Object
4
Convert Server-Side Object to JSON
3

Figure 2.4
JSON as a round-trip data-interchange format

We've defined this rather complex variable using JSON syntax. At the same time, all we've written is a standard piece of JavaScript. Curly braces are used to delimit JavaScript objects (which behave kind of like associative arrays), and square braces delimit JavaScript Array objects. If you want to brush up on your core JavaScript language skills, we cover these things in greater depth in chapter 4.

Once we've defined the variable, we can easily read its values using standard JavaScript syntax:

```
alert (customers.details[2].name);
```

This would display the string "Sun" in an alert box. So far, all we've done is take a standard piece of JavaScript syntax and called it JSON.

We can also create a string variable and then evaluate it using `eval()` to generate our variable:

```
var customerTxt = "{ 'details': [ "  +
              "{'num': '1','name':'JBoss','city':'Atlanta'}, " +
              " {'num': '2','name':'Red Hat','city':'Raleigh'}, " +
              " {'num': '3','name':'Sun','city':'Santa Clara'}, " +
              " {'num': '4','name':'Microsoft','city':'Redmond'}" +
            "] }" ;
var cust = eval ('(' + customerTxt + ')');
alert (cust.details[0].city); //shows 'Atlanta'
```

There's no good reason to write code like this when we're declaring the string ourselves, but if we're retrieving the string in a different way—say, as the response to an Ajax request—then suddenly we have a nifty data format that can express complex data structures easily and that can be unpacked with extreme ease by the JavaScript interpreter.

At this point, we have half a data-interchange format. Standard JavaScript doesn't provide any way of converting a JavaScript object into a JSON string. However, third-party libraries can be found at http://www.json.org, which allow us to serialize client-side objects as JSON, using a function called `stringify()`. The JSON library also provides a `parse()` method that wraps up the use of `eval()` nicely.

You'll also find libraries at json.org for creating and consuming JSON in a number of server-side languages. With these tools, it's possible for the client and server to send structured data back and forth as JSON over the entire course of a web application's user session.

Let's return to our Hello World example for now, and see how the client handles a JSON response.

2.2.1 Generating JSON on the server

We can go quite a long way with JSON, so let's break it up into two parts. First, we're going to look at how far we can get simply by using the browser's built-in ability to parse JSON data, and replace the generic JavaScript response from the previous example with a JSON object definition.

Problem

We want the server to respond to our request with rich structured data, and let the client decide how to render the data.

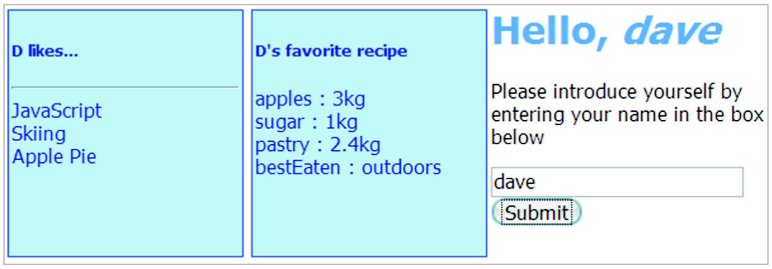
Solution

Sticking with the Hello World theme, this example is going to return a fuller description of the individual than just their name:

- The person's initial, calculated on the server using string manipulation
- A list of things that the person likes
- Their favorite recipe, encoded as an associative array

Figure 2.5 shows the application after receiving a response.

In keeping with previous examples, the back end is going to be pretty dumb and will, in fact, return the same data (apart from the initial) for every name. It's a simple step from dummy data to a real database, but we don't want to confuse things by introducing too many Java-specific back-end features, as the client-side code can talk to any server-side technology.


D likes...

JavaScript
Skiing
Apple Pie

D's favorite recipe

apples : 3kg
sugar : 1kg
pastry : 2.4kg
bestEaten : outdoors

Hello, *dave*

Please introduce yourself by entering your name in the box below

dave

Figure 2.5 JSON-powered Hello World application displaying rich data

So, first we're going to do things the simple way and just make use of JavaScript's built-in JSON-parsing capabilities. Our client-side code appears in listing 2.5.

Listing 2.5 hello5.html

```
<html>
<head>
<title>Hello Ajax version 5</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello5.jsp?name = "+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          var responseObj = eval("("+xhr.responseText+")");   ❶ Parses JSON response
          update(responseObj);
        }
      }
    );
  };
};

function update(obj){
  $('helloTitle').innerHTML = "<h1>Hello, <b><i>"
    +obj.name            <-
    +"</i></b></h1>";        ❷ Uses parsed object
  var likesHTML = "<h5>"
    +obj.initial
    +"likes...</h5><hr/>";
  for (var i=0;i<obj.likes.length;i++){
    likesHTML += obj.likes[i]+"<br/>";
  }
  $('likesList').innerHTML = likesHTML;
  var recipeHTML = "<h5>"
    +obj.initial
    +"'s favorite recipe</h5>";
  for (key in obj.ingredients){
    recipeHTML += key
     +" : "
     +obj.ingredients[key]
     +"<br/>";
  }
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

Working with JSON in this way is pretty simple. We use `eval()` to parse the JSON response ❶, remembering to add parentheses around the string before we parse it. Using the parsed object in our `update()` method ❷ is then entirely natural, because it's just another JavaScript object.

Let's look briefly at the server-side code required to get us this far. Listing 2.6 shows iteration 5 of our JSP file.

Listing 2.6 hello5.jsp

```
<jsp:directive.page contentType="application/javascript"/>
<%
String name=request.getParameter("name");
%>
{
  name: "<%=name%>",
  initial: "<%=name.substring(0,1).toUpperCase()%>",
  likes: [ "JavaScript", "Skiing", "Apple Pie" ],
  ingredients: {
    apples: "3kg",
    sugar: "1kg",
    pastry: "2.4kg",
    bestEaten: "outdoors"
  }
}
```

As we said earlier, most of the data that we've generated here is dummy data. What's interesting to us here is the creation of the JSON string, which we've simply written out by hand, inserting variable values where appropriate.

Discussion

We've demonstrated in this example that parsing JSON on the client is extremely easy, and that alone makes it a compelling possibility. However, looking back at figure 2.4, you can see that we've only covered one of the four stops in the full round-trip between client and server: the conversion of JSON to client-side objects. For a small app like this one, what we've done so far is good enough, but in larger apps, or those handling more complex data, we would want to automatically handle all aspects of serialization and deserialization, and be free to concentrate on business logic on the server and rendering code on the client. Before we leave JSON, let's run through one more example, in which we execute a full round-trip between the client and server.

2.2.2 Round-tripping data using JSON

When we're writing the client callback, we love JSON, because it makes everything so simple. However, we passed the request data down to the server using a standard HTTP query string, and then constructed the JSON response by hand. If we could manage all communication between the browser and server using JSON, we might save ourselves a lot of extra coding.

To get to that happy place, we're going to have to employ a few third-party libraries. So, let's get coding, and see how happy we are when we've got there.

Problem

We want to apply JSON at all the interfaces between our application tiers and HTTP, so that the client code can be written purely as JavaScript objects and the server purely as Java (or PHP, .NET, or whatever) objects.

Solution

We can use figure 2.4 as a crib sheet, to see where the gaps in our design are. On the browser, we've already handled step 4, the conversion of the response text into a JavaScript object. We still need to consider the conversion of the object into JSON on the client, though. To do this, we'll need to use the json.js library from www.json.org. Listing 2.7 shows how it works.

Listing 2.7 hello5a.html

```
<html>
<head>
<title>Hello Ajax version 5a</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript' src='json.js'> </script>      ◁ ❶ Includes JSON library
<script type='text/javascript'>
window.onload = function(){
  $('helloBtn').onclick = function(){
    var name = $('helloTxt').value;
    new Ajax.Request(
      "hello5a.jsp",
      {
        postBody:JSON.stringify({name:name}),      ❷ Converts object to JSON
        onComplete:function(xhr){
          var responseObj = JSON.parse(xhr.responseText);      ❸ Converts JSON to object
          update(responseObj);
        }
      }
    );
  };
};

function update(obj){
  $('helloTitle').innerHTML = "<h1>Hello, <b><i>"+obj.name+"</i></b></h1>";
  var likesHTML = "<h5>"+obj.initial+" likes...</h5><hr/>";
  for (var i=0;i<obj.likes.length;i++){
    likesHTML+=obj.likes[i]+"<br/>";
  }
  $('likesList').innerHTML=likesHTML;
  var recipeHTML="<h5>"+obj.initial+"'s favorite recipe</h5>";
  for (key in obj.ingredients){
    recipeHTML+=key+" : "+obj.ingredients[key]+"<br/>";
  }
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

The first thing that we need to do is include the json.js library ❶. Once we've done that, we can simplify the response-handling code by using the `JSON.parse()` method ❸. More importantly, though, we can reconsider the way we put together the request.

So far, we've been sending GET requests to the server, passing in data on the query string. This is fine for requesting data, but when we want to update information or send a more complex request to the server, we'd be better off using a POST request. POST requests have a body as well as a set of headers, and we can populate that body with any text that we want. Here, we're going to use JSON.

We're still using Prototype to send the request, and we now pass in a `postBody` property with the options to the Ajax.Request constructor. The value of this is the result of calling `JSON.stringify()` ❷. `stringify()` takes a JavaScript object as an argument and recurses through it, writing it out as JSON. Thus, our POST body will not contain URL-encoded key-value pairs, as it would if sent from an HTML form, but a JSON string, something like this:

```
{ name: 'dave' }
```

For such a simple piece of structured data, this might be considered overkill, but we could potentially pass very complex data in this way.

Now that we've figured out the client side of the solution, let's turn to the server. We happen to be using Java on the server for these examples, and Java knows nothing about JSON whatsoever. Neither do most server-side languages. So, to make sense of the response we've just been sending, we'll need to bring in a third-party library.

Whatever your server-side technology, you're likely to find a JSON library to fit it at www.json.org (scroll down to the bottom of the page). We selected Json-lib, which is based on Doug Crockford's original JSON for Java libraries.

Json-lib has quite a bit of work to do. JSON encodes structured data in a very fluid way, and Java is a strongly typed language, so the two don't sit together naturally. Nonetheless, we managed to get the job done without too much trouble. Listing 2.8 shows the not-so-gory details.

Listing 2.8 hello5a.jsp

```
<jsp:directive.page
  contentType="application/javascript"
  import="java.util.*,net.sf.json.*"      <-┐
/>                                          ❶ Imports JSON classes
<%
String json=request.getReader().readLine();   <-❷ Reads POST body
```

```
JSONObject jsonObj=new JSONObject(json);          ❸ Parses JSON string
String name=(String)(jsonObj.get("name"));
                                                   ❹ Reads parsed object
jsonObj.put("initial",
  name.substring(0,1).toUpperCase());
                                                ❺ Adds new values
String[] likes=new String[]
  { "JavaScript", "Skiing", "Apple Pie" };
jsonObj.put("likes",likes);

Map ingredients=new HashMap();
ingredients.put("apples","3kg");
ingredients.put("sugar","1kg");
ingredients.put("pastry","2.4kg");
ingredients.put("bestEaten","outdoors");
jsonObj.put("ingredients",ingredients);         ❻ Writes object as JSON
%><%=jsonObj%>
```

In order to use the Json-lib classes in our project, we need to import the net.sf.json package, which we do in the `<jsp:directive.page>` tag ❶. Now, on to the code.

The first challenge that we face is decoding the POST body. The Java Servlet API, like many web technologies, has been designed to make it easy to work with POST requests sent from HTML forms. With a JSON request body, we can't use `HttpServletRequest.getParameter()`, but need to read the JSON string in the request via a `java.io.Reader` ❷. Similar capabilities are available for other technologies. If you're using PHP, use the `$HTTP_RAW_POST_DATA` variable. If you're using the .NET libraries, you'll need to get an `InputStream` from the HttpRequest object, much as we've done here with our Java classes.

Back to the Java now. Once we've got the JSON string, we parse it as an object ❸. Because of the fundamental disjoint between loosely typed JSON and strictly typed Java, the Json-lib library has defined a JSONObject class to represent a parsed JSON object. We can read from it using the `get()` method ❹ and extract the name from the request.

Now that we've deserialized the incoming JSON object, we want to manipulate it, and then send it back to the client again. The JSONObject class is able to consume simple variable types such as strings, arrays, and Java Maps (that is, associative arrays) ❺, to add extra data to the object. Once we've modified the object, we serialize it again ❻, sending it back to the browser in the response.

And that's it! We've now sent an object from the client, modified it on the server, and returned it back to the client again.

Discussion

This has been the most complex example so far, demonstrating a way of communicating structured objects back and forth over the network on top of the text-based HTTP protocol. Because both the client and server can understand the JSON syntax, with a little help from some libraries, we haven't had to write any parsing code ourselves. However, as we noted, JSON is suited for use with loosely typed scripting languages, and so there was still some translation required. The goal of a system like this is to be able to serialize and deserialize our domain objects over the network. If our domain objects graph is written in Java (or C#, say), then we still need to manually translate them into generic hashes and arrays before they can be passed to the JSON serializer. The clumsiest piece of coding in our round-trip was in the JSP, where we assembled the Maps and lists of data for the JSONObject. This problem is strongly emphasized in the case of Java, which lacks a concise syntax for defining associative arrays in particular, compared with Ruby or PHP, for example.

From square brackets to angle brackets

There is another text-based format that can be understood by both client and server: XML. Most server-side languages have good support for XML, so we might find that we have an easier time working with XML than with JSON on the server. In the next section, we'll look at XML and Ajax, and see whether that is the case.

2.3 Using XML and XSLT with Ajax

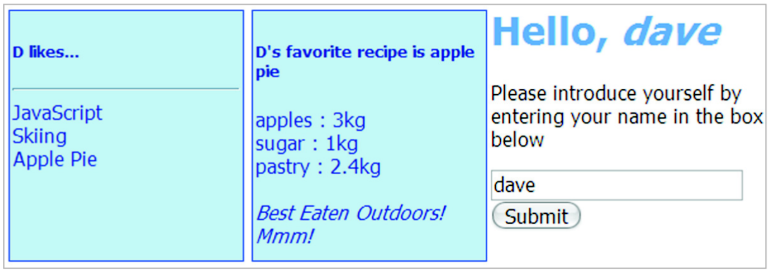
XML is a mature technology for representing structured data, supported by most programming languages either as a core part of the language or through well-tested libraries or extensions. In the remainder of this chapter, we'll look at how various XML technologies work with XML, and complete our survey of basic Ajax communication techniques.

The XMLHttpRequest object that we've been using for our Ajax requests has special support for XML built into it. So far, we've been extracting the body of the HTTP response as text and parsing it from there. JavaScript in the web browser doesn't have a standard XML parser available to it, but the XHR object can parse XML responses for us, as we'll see in the next example.

2.3.1 Parsing server-generated XML

So far, we've had the server generate HTML, JavaScript, and JSON responses for us in various versions of our Hello World application. All of these formats are designed to appeal to the browser rather than the server. XML, in contrast, is often used to communicate between server processes, in a variety of technologies ranging from RSS syndication feeds to web service protocols, such as XML-RPC and SOAP. If we're transmitting information from our domain objects up to the client, then many server-side technologies provide support for serializing and deserializing objects as XML.

In any of these scenarios, we may find that it's easy to transmit data as XML, from the perspective of the server. If this is going to be a useful way forward, then we'll also need to handle the XML on the client. We'll start by looking at the built-in support for XML offered by the XHR object. XML, like JSON, is a format for exchanging structured data. The best way to compare the two is to set them the same task, so we'll follow the lead from the previous section and supply a list of likes and a favorite recipe, as shown in figure 2.6.


D likes...

JavaScript
Skiing
Apple Pie

D's favorite recipe is apple pie

apples : 3kg
sugar : 1kg
pastry : 2.4kg

Best Eaten Outdoors! Mmm!

Hello, *dave*

Please introduce yourself by entering your name in the box below

dave

Figure 2.6 Hello World example version 6 after parsing XML response

Problem

The server is sending structured data as XML. We need to parse this data on the client.

Solution

The first step in handling XML on the client side is to use the XHR's ability to parse the response into a structured XML document. The second step is to read (and potentially write) the parsed XML document using the W3C standard known as the Document Object Model (DOM). In JavaScript, we already have an implementation of the DOM for working with HTML web pages programmatically. The good news, then, is that we can leverage these existing skills to work with XML documents delivered by Ajax. Listing 2.9 shows the full code for version 6 of our Hello World application.

Listing 2.9 hello6.html

```
<html>
<head>
<title>Hello Ajax version 6</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload = function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello6.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          var responseDoc = xhr.responseXML;        ❶ Reads response as XML
          update(responseDoc);
        }
      }
    );
  };
};

function update(doc){
  var personNode = doc
    .getElementsByTagName('person')[0];
  var initial = personNode
    .getAttribute('initial');
  var nameNode = personNode
    .getElementsByTagName('name')[0];             ❷ Extracts data
  var name = nameNode.firstChild.data;              using DOM
  var likesNode = personNode
    .getElementsByTagName('likes')[0];
  var likesList = likesNode
    .getElementsByTagName('item');
  var likes = [];
  for (var i=0;i<likesList.length;i++){
    var itemNode = likesList[i];
    likes[i] = itemNode
      .firstChild.data;
  }
  var recipeNode = personNode
    .getElementsByTagName('recipe')[0];
  var recipeNameNode = recipeNode
     .getElementsByTagName('name')[0];
  var recipeName = recipeNameNode.firstChild.data;
  var recipeSuggestNode = recipeNode
    .getElementsByTagName('serving-suggestion')[0];
  var recipeSuggest = recipeSuggestNode.firstChild.data;
  var ingredientsList = recipeNode
    .getElementsByTagName('ingredient');
  var ingredients = {};
  for(var i=0;i<ingredientsList.length;i++){
    var ingredientNode = ingredientsList[i];
    var qty = ingredientNode.getAttribute("qty");
    var iname = ingredientNode.firstChild.data;
    ingredients[iname] = qty;
  }

  $('helloTitle').innerHTML =
    "<h1>Hello, <b><i>"
    +name
    +"</i></b></h1>";
  var likesHTML = '<h5>'
    +initial
    +' likes...</h5><hr/>';
  for (var i=0;i<likes.length;i++){
    likesHTML += likes[i]+"<br/>";
  }
  $('likesList').innerHTML = likesHTML;
  var recipeHTML = "<h5>"
    +initial
    +"'s favorite recipe is "
    +recipeName
    +"</h5>";
  for (key in ingredients){
    recipeHTML += key+" : "
      +ingredients[key]
      +"<br/>";
  }
  recipeHTML+="<br/><i>"
    +recipeSuggest
    +"</i>";
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
```

❷ Extracts data using DOM

❸ Assembles HTML

```
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

The first step here is by far the easiest. We can retrieve the response as an XML document object simply by reading the `responseXML` property ❶ rather than `responseText`. We've then rewritten our `update()` function to accept the XML object. All we need to do now is read the individual data elements from the XML object ❷ and render the data as HTML content ❸.

In practice, neither of these steps is difficult, but they are rather lengthy. Using the DOM methods and properties such as `getElementsByTagName()`, `getAttribute()`, and `firstChild`, we can drill down to the data we want, but we need to do it step by step. These properties and methods are identical to the ones that we use when working with HTML documents, so we won't deal with them individually here. If you've used the DOM to manipulate HTML, everything should look familiar. If you haven't, then there is plenty of information on these methods online.

Once we have extracted the data that we need, then we simply assemble the HTML content necessary to update the UI.

We've already discussed the many scenarios under which it might make sense for the server to generate XML. To keep things simple in this example and avoid in-depth coverage of technologies that only apply to a single programming language, we've simply generated the XML by hand in our JSP. Listing 2.10 presents the JSP for the sake of completeness.

Listing 2.10 hello6.jsp

```
<jsp:directive.page contentType="text/xml"/>
<%
String name=request.getParameter("name");
%>
<person initial="<%=name.substring(0,1).toUpperCase()%>">
 <name><![CDATA[<%=name%>]]></name>
 <likes>
  <item>JavaScript</item>
  <item>Skiing</item>
  <item>Apple Pie</item>
 </likes>
 <recipe>
  <name>apple pie</name>
  <ingredient qty="3kg">apples</ingredient>
  <ingredient qty="1kg">sugar</ingredient>
  <ingredient qty="2.4kg">pastry</ingredient>
  <serving-suggestion>
   <![CDATA[Best Eaten Outdoors! Mmm!]]>
  </serving-suggestion>
 </recipe>
</person>
```

Note that we've set the `contentType` of our response as `text/xml` in this case. We've been doing this throughout our examples, largely as a show of good habits. In this case, though, we have a strong practical reason for doing so. If we don't set the MIME type to some type of `xml` (either `text/xml` or `application/xml` will do), then the `responseXML` property of the XHR object won't be populated correctly.

The rest of the JSP is unremarkable. To a seasoned Java and XML coder, it might also look overly simplistic, with the XML being handwritten as text. A more robust solution would be to use a library like JDOM to generate the XML document programmatically, and we encourage the reader to do that in practice. However, we've left it simple here—maybe painfully simple—to show the intent of what we're doing without getting too deeply into Java-specific libraries. After all, our main aim in this book is to teach client-side techniques, and our choice of Java rather than PHP, Ruby, or .NET was essentially arbitrary.

So, getting back to the code, we've simply created a template of the XML document, most of which contains dummy data, and added in a few dynamic values along the way. Let's move on to evaluate our experience with this example.

Discussion

Our first encounter with XML and Ajax has been rather mixed. Initially, things looked pretty good, given the special support for XML baked into the XHR object. However, manually walking through the XML response using the DOM was rather lengthy and uninspiring. Experience of this sort of coding has been sufficient to put a lot of developers off XML in favor of JSON.

The DOM is a language-independent standard, with implementations in Java, PHP, C++, and .NET, as well as the JavaScript/web browser version that we're familiar with. When we look at the use of XML outside of the web browser, we find that the DOM is not very widely used and that other XML technologies exist that make working with XML much more palatable. Thankfully, these technologies are available within the browser too, and we'll see in the next section how we can use them to make Ajax and XML work together in a much happier way.

2.3.2 Better XML handling with XSLT and XPath

The XML techniques that we saw in the previous example represent the core functionality that is available free of charge via all implementations of the XHR object. Working directly with the DOM is not pleasant, especially if you're used to more modern XML-handling technologies in other languages. The most common of these tools are XPath queries and Extensible Stylesheet Language Transformations (XSLT) transforms.

XPath is a language for extracting data out of XML documents. In listing 2.9, we had to drill down through the document one node at a time. Using XPath, we can traverse many nodes in a single line. XSLT is an XML-based templating language that will allow us to generate any kind of content, such as, for instance, HTML, from our XML document more easily, and also separate out the logic from the presentation rather better than we've been doing so far. XSLT style sheets (as the templates are known—no relation to Cascading Style Sheets) use XPath internally to bind data to the presentation.

The good news is that XSLT transforms and XPath queries are available on many browsers, specifically on Firefox and Internet Explorer. Even better, these are native objects exposed to the JavaScript engine, so performance is good. Safari hasn't yet provided a native XSLT processor, so this isn't a good option if support for a (non-Firefox) Mac audience is important.

In the following example, we'll let Prototype have a well-earned rest, and use the Sarissa library to demonstrate simple cross-browser XSLT and XPath as a way of simplifying our XML-based Hello World example.

Problem

Working with the DOM on our Ajax XML responses is slow and cumbersome. We want to use modern XML technologies to make it easy to develop with Ajax and XML.

Solution

Use XPath and XSLT to simplify things for you. Both Internet Explorer and Firefox support these technologies, but in quite different ways. As with most cross-browser incompatibilities, the best strategy is to use a third-party library to present a unified front to our code. For this example, we've chosen Sarissa (http://sarissa.sf.net), which provides cross-browser wrappers for many aspects of working with XML in the browser. Listing 2.11 shows the client-side code for our XSLT and XPath-powered app.

Listing 2.11 hello7.html

```
<html>
<head>
<title>Hello Ajax version 7</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript'
  src='sarissa.js'> </script>
<script type='text/javascript'
  src='sarissa_ieemu_xpath.js'> </script>           ❶ Imports Sarissa libraries
<script type='text/javascript'
  src='sarissa_dhtml.js'> </script>
<script type='text/javascript'>

var xslDoc=null;

window.onload=function(){

  xslDoc=Sarissa.getDomDocument();                 ❷ Loads XSL style sheet
  xslDoc.load("recipe.xsl");

  document.getElementById('helloBtn')
```

```
   .onclick = function(){
     var name = document.getElementById('helloTxt').value;
     var xhr = new XMLHttpRequest();
     xhr.open("GET",                                ❸ Creates XHR
       "hello7.jsp?name="                             object
       +encodeURI(name),true);
     xhr.onreadystatechange = function(){               ❹ Assigns callback
       if (xhr.readyState == 4){                          function
         update(xhr.responseXML);
       }
     };
     xhr.send("");
   };
 };

 function update(doc){
   var initial = doc.selectSingleNode(
     '/person/@initial'
   ).value;                                          ❺ Selects individual
   var name = doc.selectSingleNode(                    nodes
     '/person/name/text()'
   ).nodeValue;
   document.getElementById('helloTitle')
    .innerHTML = "<h1>Hello, <b><i>"
       +name+"</i></b></h1>";

   var likesList = doc                                ❻ Selects multiple
     .selectNodes('/person/likes/item');                 nodes
   var likes = [];
   for (var i=0;i<likesList.length;i++){
     var itemNode = likesList[i];
     likes[i]=itemNode
       .firstChild.data;
   }
   var likesHTML='<h5>'
     +initial+' likes...</h5><hr/>';
   for (var i=0;i<likes.length;i++){
     likesHTML += likes[i]+"<br/>";
   }
   document.getElementById('likesList')
     .innerHTML = likesHTML;

   var personNode = doc.selectSingleNode('/person');

   var xsltproc = new XSLTProcessor();
   xsltproc.importStylesheet(xslDoc);
   Sarissa.updateContentFromNode(                     ❼ Invokes XSLT
     personNode,                                         transform
     document.getElementById('ingrList'),
     xsltproc
   );
 }
```

```
</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

The first thing that we need to do is to import the Sarissa libraries ❶. As well as importing the core library, we import a support library that provides IE-style XPath under Firefox, and a helper library that offers some convenience methods for inserting XSLT-generated content into web pages.

Generating content using XSLT requires two XML documents from the server: the style sheet (that is, the template) and the data. We'll fetch the data on demand, as before, but can load the style sheet up front when we load the app ❷. We do this using a DomDocument object rather than the XHR. Once again, Sarissa provides us with a cross-browser wrapper.

To load the XML data, we will use an XHR object. Because we've put Prototype aside for this example, we need to create the XHR object by hand ❸ and assign the callback ❹. Nonetheless, the code is simpler than we saw in chapter 1, because we can access a native XHR object, even on Internet Explorer. Internally, Sarissa does a bit of object detection, and if no native XHR object can be found, it will create one for us that secretly creates an ActiveX control and uses it.

So, once we've got our XHR object, we can pass a DOM object to our `update()` function. This was where our troubles started when using the DOM. Using XPath, we can drill down through several layers of DOM node in a single line of code. For example, the XPath query

```
/person/name/text()
```

selects the internal text of a `<name>` tag nested directly under a `<person>` tag at the top of the document. XPath is too big a subject for us to tackle in depth here. We suggest http://zvon.org as a good starting place for newcomers to XPath and XSLT. The DOM Node methods `selectSingleNode()` ❺ and `selectNodes()` ❻ are normally only found in Internet Explorer, but the second Sarissa library that we loaded has provided implementations for Firefox/Mozilla. We're using XPath to extract the name data and the list of likes, and constructing the HTML content for those regions of the screen manually, as they're relatively straightforward. The recipe section is more complex, so we'll use that to showcase XSLT.

The final step is to perform the XSLT transform ❼. The XSLTProcessor object is native to Mozilla, and provided under IE by Sarissa. We pass it a reference to the style sheet, and then call a method `updateContentFromNode()`. This helper method, provided by the third Sarissa library that we loaded, will pass the data (i.e., `personNode`) through the XSLT processor and write the resulting HTML into the specified DOM node (i.e., `ingrList`).

To make this work, of course, we also need to provide an XSL style sheet. That's shown in listing 2.12.

Listing 2.12 recipe.xsl

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

 <xsl:template match="/person">
  <div>
  <h5><xsl:value-of select='@initial'/>'s
   favorite recipe is
  <xsl:value-of select='recipe/name'/></h5>
  <p><xsl:apply-templates select="recipe/ingredient" /></p>
  <p><i><xsl:value-of select='recipe/serving-suggestion'/></i></p>
  </div>
 </xsl:template>

 <xsl:template match="ingredient">
  <xsl:value-of select='@qty'/> : <xsl:value-of select='.'/><br/>
 </xsl:template>

</xsl:stylesheet>
```

Our XSL style sheet is quite straightforward. It's a mixture of ordinary XHTML markup, and special tags prefixed with `xsl`, indicating the XSL namespace. These are treated as processing instructions. `<xsl:template>` tags specify chunks of content that will be output when a node matching the XPath query in the match attribute is encountered. `<xsl:value-of>` prints out data from the matched nodes, again using XPath expressions. The `<xsl:apply-templates>` tag routes nodes to other template tags for further processing. In this case, each ingredient node will be passed to the second template from the first, generating a list.

Again, we don't have space for a full exposition of XSLT style sheet rules here. If you wish to know more, we recommend you visit http://zvon.org.

Finally, let's turn briefly to the server side. The JSP used in this example is identical to that from the previous example, as presented in listing 2.10. The only changes that we've introduced have been in the client code.

Discussion

Using XSLT and XPath has certainly simplified our client-side XML-handling code. In a more complex application, these technologies will scale more easily than the DOM in terms of coding effort. We recommend that anyone considering using Ajax with XML investigate these technologies.

In our section on JSON, we discussed the notion of round-tripping structured data between the client and the server. Sarissa promotes this approach, using XML as the interchange format, as it also supports cross-browser serialization of XML objects. As we stated earlier, almost any server-side technology will provide support for serializing and deserializing XML, too. We won't explore a full example here, but the principle is similar to the JSON case. When using JSON with Java, we noted that a fair amount of manual work was required to construct the JSON response because of the mismatch between loosely typed JSON and strictly typed Java. The same issues exist when converting between Java and XML, but the problem space is better understood, and out-of-the-box solutions such as Castor and Apache XMLBeans are available.

We've presented Sarissa as a one-stop shop for these technologies. It isn't the only game in town. If you only want XPath queries, then the mozXPath.js library (http://km0ti0n.blunted.co.uk/mozXPath.xap) provides a lightweight alternative, with support for the Opera browser as well. And if you like the look of XSLT but need it to work on Safari, then you can try Google's AJAXSLT, a 100 percent JavaScript XSLT engine (http://goog-ajaxslt.sf.net). Be warned, though, that AJAXSLT is slow compared to the native engines in IE and Mozilla and won't support the full XSL namespace, so you'll need to write your style sheets with the limitations of the library in mind and keep them reasonably small.

We're nearly done with our review of Ajax technologies. In the final section, we'll explore another Internet technology that makes use of XML, SOAP web services, and see how Ajax can interface with that.

2.4 Using Ajax with web services

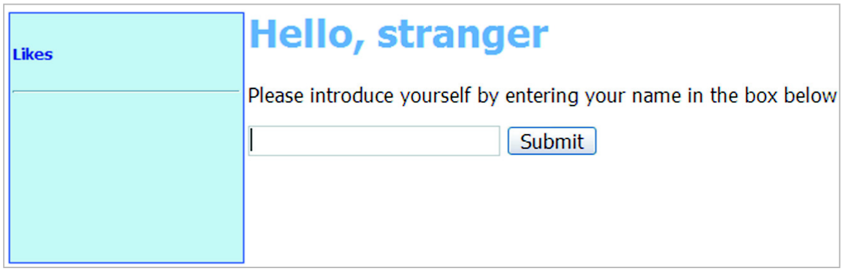
In this section, we will see how to call web services running on a remote server over SOAP. After all, what is a web service but XML data being passed back and forth? The XHR object is ideally suited for such a task and makes invoking remote methods over SOAP less of a daunting task than it may seem.

Internet Explorer and the Mozilla versions of browsers all have native objects that can be used to invoke web services. Sadly, these objects are not portable between browsers; the developer is left to write a custom framework that can choose the proper objects to invoke. Microsoft maintains several pages dedicated to its version of browser-side SOAP at http://msdn.microsoft.com/workshop/author/webservice/overview.asp. Microsoft's implementation is based on both JavaScript and VBScript. Mozilla explains their version at www.mozilla.org/projects/webservices/; more information can also be found at http://developer.mozilla.org/en/docs/SOAP_in_Gecko-based_Browsers. Their version of browser-side SOAP is accessible through native objects that can be constructed on the browser side.

Fortunately, there is another way. Instead of writing a high-level API that can make use of either Internet Explorer or Mozilla objects, we can create our own library that uses XMLHttpRequest to exchange XML, and that can parse and generate the SOAP messages. Such a library would also allow us to run our code on browsers that do not supply either the Microsoft or Mozilla SOAP APIs but that do have the XHR object. The kind people at IBM have created just such a library and have named it ws-wsajax. It can be found at www.ibm.com/developerworks/webservices/library/ws-wsajax/. We will be using this library for the remainder of this section.

We've simplified the UI for this example, removing the recipe section. Passing in the name will return a map with three entries: the name, the initial, and the list of likes. Figure 2.7 shows the UI for this example.

This section assumes some familiarity with SOAP and SOAP-RPC. Once again, there are several books available, as well as many good tutorials online, that cover this topic in depth.



Hello, stranger

Please introduce yourself by entering your name in the box below

Figure 2.7 Hello World version 8. We've simplified the presentation here, removing the recipe element from the UI.

Problem

You need to perform SOAP-RPC from a web browser. You need to display the resulting SOAP response as HTML.

Solution

In this section, we will write a small client using IBM's SOAP toolkit to access our own Hello World SOAP service, written using Apache's Axis framework (http://ws.apache.org/axis/). Let's begin by defining our web service. Axis makes it very easy to prototype web services by writing Java classes in files with a special filename extension: `.jws`. Like JSPs, `.jws` files will be compiled on demand by a special servlet, in this case the AxisServlet, and, while not robust enough for production use, serve the purposes of our simple demonstration admirably. Listing 2.13 shows a simple `.jws` file for our Hello World service.

Listing 2.13 HelloWorld.jws

```
import java.util.Map;
import java.util.HashMap;

/**
 * class to list headers sent in request as a string array
 */
public class HelloWorld {

    public Map getInfo(String name) {
      String initial=name.substring(0,1).toUpperCase();
      String[] likes=new String[]
  { "JavaScript", "Skiing", "Apple Pie" };
      Map result=new HashMap();
      result.put("name",name);
      result.put("initial",initial);
```

```
        result.put("likes",likes);
        return result;
      }

  }
```

The class contains a single method, which will be mapped to a SOAP-RPC function. The function takes one argument, of type String, and returns an associative array (referred to in Java as a Map).

Pointing our browser at `HelloWorld.jws` will return a Web Service Description Language (WSDL) file, which the SOAP client, such as the IBM library, can interrogate in order to build up client-side stubs, allowing us to call the service. Listing 2.14 shows the WSDL generated by this class.

Listing 2.14 WSDL for HelloWorld.jws

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:intf="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema
    targetNamespace="http://xml.apache.org/xml-soap"
    xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType name="mapItem">
    <sequence>
     <element name="key" nillable="true" type="xsd:anyType"/>
     <element name="value" nillable="true" type="xsd:anyType"/>
  </sequence>
  </complexType>
   <complexType name="Map">
    <sequence>
     <element maxOccurs="unbounded" minOccurs="0"
       name="item" type="apachesoap:mapItem"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>
```

```
    <wsdl:message name="getInfoResponse">
       <wsdl:part name="getInfoReturn" type="apachesoap:Map"/>
    </wsdl:message>
    <wsdl:message name="getInfoRequest">
       <wsdl:part name="name" type="xsd:string"/>
   </wsdl:message>
   <wsdl:portType name="HelloWorld">
       <wsdl:operation name="getInfo" parameterOrder="name">
          <wsdl:input message="impl:getInfoRequest"
             name="getInfoRequest"/>
          <wsdl:output message="impl:getInfoResponse"
             name="getInfoResponse"/>
       </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="impl:HelloWorld">
       <wsdlsoap:binding style="rpc"
          transport="http://schemas.xmlsoap.org/soap/http"/>
       <wsdl:operation name="getInfo">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input name="getInfoRequest">
             <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://DefaultNamespace" use="encoded"/>
          </wsdl:input>
          <wsdl:output name="getInfoResponse">
             <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://localhost:8080/AiP2/HelloWorld.jws"
               use="encoded"/>
          </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
    <wsdl:service name="HelloWorldService">
       <wsdl:port binding="impl:HelloWorldSoapBinding"
         name="HelloWorld">
          <wsdlsoap:address
            location="http://localhost:8080/AiP2/HelloWorld.jws"/>
       </wsdl:port>
    </wsdl:service>
 </wsdl:definitions>
```

The WSDL includes details on the argument types and return types of each RPC call, bindings to the functions, and other details needed by the client and server to specify the nature of the interchange. Fortunately, the WSDL is generated for us automatically by Axis and is consumed by the IBM toolkit, so we don't need to understand every line in it.

Let's turn now to our client-side code. Listing 2.15 shows the full listing for version 8 of our Hello World app.

Listing 2.15 hello8.html

```
<html>
<head>
<title>Hello Ajax version 8</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript'                 ❶ Imports Prototype
  src='prototype_v131.js'> </script>
<script type='text/javascript' src='ws.js'> </script>    ❷ Imports IBM WS library
<script type='text/javascript'>

window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    var wsNamespace = '../axis/HelloWorld.jws';     ❸ Creates client
    var wsCall = new WS.Call(wsNamespace);             from WSDL
    var rpcFunction = new
      WS.QName('getInfo',wsNamespace);    ❹ References RPC function
    wsCall.invoke_rpc(
      rpcFunction,
      [{name:'name',value:name}],    ❺ Passes RPC arguments
      null,
      function(call,envelope){
        var soapBody = envelope.get_body();      ❻ Defines callback
        var soapMap = soapBody
          .get_all_children()[1].asElement();
        var itemNodes = soapMap
          .getElementsByTagName('item');
        var initial = "";
        var likes = [];
        for (var i=0;i<itemNodes.length;i++){
          var itemNode = itemNodes[i];
          var key = itemNode
            .getElementsByTagName('key')[0]
```

```
              .firstChild.data;
            if (key == 'initial'){
              initial = itemNode
                .getElementsByTagName('value')[0]
                .firstChild.data;
            }else if (key == 'likes'){
              var likeNodes = itemNode
                .getElementsByTagName('value')[0]
                .getElementsByTagName('value');
              for (var j=0;j<likeNodes.length;j++){
                likes[likes.length] = likeNodes[j]
                  .firstChild.data;
              }
            }
          }
          update(initial,likes);
        }
      );
    };
  };

function update(initial,likes){    <-7 Updates UI
  var content = "<h5>"+initial
    +" likes...</h5><hr/>";
  for (var i=0;i<likes.length;i++){
    content += likes[i]+"<br/>";
  }
  $('likesList').innerHTML = content;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
&nbsp;
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

There's a lot going on here, so let's take it line by line. First, we need to import the IBM library ❷. Because this library is built on top of Prototype, we include that too. It relies on an older version of Prototype (v1.3.1), so we've renamed it to avoid confusion with the rest of our examples ❶.

To consume the Web Service, the first thing that we need to do is reference the WSDL and feed it to a `WS.Call` object ❸. We then extract a reference to the specific function, as a `WS.QName` object ❹. We can call this object, providing the input parameters as a JavaScript object (which we've defined inline here using JSON) ❺, and a callback function to parse the response ❻. Parsing the response requires a lot of node traversal. We're working with SOAP nodes rather than DOM nodes here, but the SOAP nodes can be converted to DOM nodes at any point. We've omitted any use of XPath here to keep the example simple, but wading through a larger SOAP response would certainly merit investigating use of XPath. Once we have extracted the data from the response, we pass it to our `update()` function ❼, as usual. Again, we've opted for simplicity here, but there's nothing to stop you from using XSLT transforms on the SOAP response once you've got ahold of it.

Discussion

We've shown that it's possible to use SOAP with Ajax, provided of course that the SOAP service is coming from the same server as the Ajax client, and therefore honoring the browser's same-origin security restrictions. If your back-end system already generates SOAP, then this is a valid way of reusing existing resources. However, we'd be tempted to say that SOAP, as an architecture for a green-field development, is unnecessarily complex if interoperability with external entities is not also a requirement.

The IBM SOAP toolkit made it very easy to call the service, but somewhat less easy to parse the response. SOAP-RPC responses typically involve several namespaces and are complex to decode. Document/literal-style SOAP bindings generally provide simpler responses, which might be a better fit for this toolkit in production.

As always, caveat programmer. If you need a quick solution, SOAP may not be the way to go. However, if you are creating a large application that you foresee will require many updates and extensions, as well as integration with many aspects of your organization, and you have the time and skills to do it, browser-side SOAP may benefit you.

2.5 Summary

By the end of chapter 1, we'd figured out how to make an Ajax request, and looked at ways of simplifying the process by using third-party libraries. We've covered a lot of ground since then and shifted our focus from simply being able to make a request, to looking at how we want to structure the conversation between client and server over the lifetime of the application.

We've looked at several techniques in this chapter and evaluated the strengths and weaknesses of each. We began by looking at generating JavaScript code on the server and saw the benefits of writing generated code against a high-level API in order to prevent excessive tangling between the client and server codebases.

We moved on from there to look at ways of passing structured data between the client and server, starting with JSON and then continuing on to XML. In each case, we began by simply looking at how to parse the data when it arrived from the server, and then moved on to consider the full round-trip of data between client and server. By round-tripping the data, and having library code to serialize and deserialize at both ends, we can free ourselves up to write business code rather than low-level plumbing.

In contrast to JSON and XML, JSON has a closer affinity with the client side. We struggled with our client-side XML initially but made significant advances when we picked up XPath and XSLT. There is no clear winner between the two technologies, and the decision remains a matter of personal taste, and depends on whether you are integrating with legacy systems that naturally fit better with either JSON or XML.

In the next chapter, we'll look at JavaScript as the programmatic glue that binds the entire Ajax app together. We'll discuss recent advances in thinking about JavaScript, and how they can help you to write better-structured code for your Ajax app. We'll conclude with a discussion of some of the popular Ajax frameworks.

WEB PROGRAMMING

Ajax IN PRACTICE

Dave Crane, Bear Bibeault, and Jord Sonneveld
with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Collectively, web developers have learned a huge amount about Ajax. But it's difficult for any one of them to access and distill all that knowledge. Fortunately, it's now unnecessary: this book collects the most valuable techniques developed by the Ajax community and puts them in your hands.

Ajax in Practice gives you 60 best-practice Ajax techniques illustrated with crisp examples and tons of well-explained code you can reuse. All this is presented in an easy-to-follow, repeating format. The book starts by covering the prerequisites—key Ajax frameworks and object-oriented JavaScript (something you'll need if you want to write scalable Ajax code). Then, it helps you master practical methods for event handling, validation, and state management. A thorough discussion makes each example clear and shows how individual techniques can be combined and extended.

You'll learn how to

- Implement drag and drop the right way (Chapter 9)
- Control the propagation of an event through the DOM tree (Chapter 5)
- Add back-button and undo support (Chapter 8)
- Implement effective navigation strategies (Chapter 7)
- Prefetch data to improve performance (Chapter 11)
- Build a Yahoo! Maps + Flickr mashup (Chapter 13)

Ajax in Practice brings together a team of experts including **Dave Crane**, leading Ajax authority and best-selling author of Manning's *Ajax in Action*, **Bear Bibeault** of Works.com and JavaRanch, **Jord Sonneveld** of Google, **Chris Gray** of Infor, **Ram Venkataraman** of JBoss, **Ted Goddard** of IceFaces, and **Joe Walker**, creator of DWR.

For more information, code samples, and to purchase an ebook visit www.manning.com/AjaxinPractice

"A 'second-generation' book that distills experience-based practices. Confident and balanced!"

—Ernest J. Friedman-Hill
Sandia National Laboratory
Author of *Jess in Action*

"Any Ajax coder will benefit. [This book] will be useful for years to come."

—Curt Christianson
Microsoft MVP

ISBN-10: 1-932394-99-0
ISBN-13: 978-1-932394-99-3

54499

9 781932 394993

MANNING $44.99/Can$ 58.99