

Listing 6.3 Output of running the RestWebAPIConsumer program

```

1 Executing request: GET http://localhost:8080/soba/restTx/txID/210225211
  HTTP/1.1
2 -----
3 HTTP/1.1 200 OK
4 Response content length: -1
5 Response content type: application/json;charset=UTF-8
6 {"transactionId":210225211,"transDate":1442097239000,"type":"regular","ini
  tiator":"self","description":"testing","amount":100.0,"balance":100.0,"acc
  ountId":"298743065","status":"complete","id":210225211}
7 Response status: HTTP/1.1 201 Created
8 All response headers ----- begin -----
9 header: name = Server, value = Apache-Coyote/1.1
10 header: name = Cache-Control, value = no-store
11 header: name = Pragma, value =
12 header: name = Expires, value = 0
13 header: name = X-XSS-Protection, value = 1; mode=block
14 header: name = X-Frame-Options, value = DENY
15 header: name = X-Content-Type-Options, value = nosniff
16 header: name = method, value = createTransaction
17 header: name = tx, value = transactionId: 818679184;accountId:
  298743065;amount: 1.23
18 header: name = Content-Type, value = text/plain;charset=ISO-8859-1
19 header: name = Content-Length, value = 83
20 header: name = Date, value = Sat, 12 Sep 2015 23:05:31 GMT
21 All response headers ----- end -----
22 Response body ----- begin -----
23 Rest createTx succeeded: transactionId: 818679184;accountId:
  298743065;amount: 1.23
24 Response body ----- end -----

```

You can also use `Spring RestTemplate` in the client in place of the `Apache HttpClient` API as shown above. Refer to the `Spring Framework` reference documentation for more information on how to access RESTful services on the client the *Spring-way*.

Next, we discuss how to use `jQuery` and its `Ajax` functions to consume `Spring RESTful Web services`.

6.5 CONSUMING RESTFUL WEB SERVICES USING JQUERY AND ITS AJAX FUNCTIONS

Ideally, a performing and scalable web application should have its backend send data to the frontend, which renders the UI to the user, rather than render a page into a HTML file and send it to the user's browser for display. In this regard, `JavaScript`, `jQuery`, `Ajax` and various `JavaScript`-based UI frameworks, such as `AngularJS`, `Backbone`, `Ember`, `Google Web Toolkit`, and so on, represent some of the mainstream technologies. Figure 6.1 shows the `Codebrief`'s view of the top 10 `JavaScript MVC` frameworks (<http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/>).

Here is a table showing all of the frameworks support for the above features. Click through the title for more detail.

Framework	UI Bindings	Composed Views	Web Presentation Layer	Plays Nicely With Others
Backbone.js	X	X	✓	✓
SproutCore 1.x	✓	✓	X	X
Sammy.js	X	X	✓	✓
Spine.js	X	X	✓	✓
Cappuccino	✓	✓	X	X
Knockout.js	✓	X	✓	✓
Javascript MVC	X	✓	✓	✓
Google Web Toolkit	X	✓	X	X
Google Closure	X	✓	✓	X
Ember.js	✓	✓	✓	✓
Angular.js	✓	X	✓	✓
Batman.js	✓	X	✓	✓

Figure 6.1 The Top 10 JavaScript MVC Frameworks Reviewed (codebrief.com)

This section shows a simple example of how to use jQuery and its Ajax functions to consume RESTful Web services. I wish to build an entirely new Ember.js-based or Backbone.js-based frontend for SOBA, but it would take a lot more time than the simple jQuery/Ajax-based demo to be presented here. In addition, the learning curve with Ember.js or Backbone.js would be much steeper than with jQuery/Ajax for a reader. So, let's settle down on jQuery/Ajax for now.

6.5.1 HTML/JSP

The first step is to have an HTML file that represents the UI. Listing 6.4 shows this file. We made it a JSP file, as we wanted to make it part of SOBA. For this purpose, note the item `${ctx}` at line 8. This item is defined in the *include.jsp* file as follows:

```
<c:set var="ctx" value="${pageContext.request.contextPath}"/>
```

The `ctx` variable in the above line represents the `pageContext` as is explicit from its value definition. Specifically, the `${ctx}` entry will be replaced at runtime with “soba” so that the `src` variable at line 8 will be translated into:

```
soba/rest/js/rest.js
```

This way, we do not have to hard-code the entry “soba” into the script. We could have made this file a `rest.html` file if we put “soba” in place of “`{ctx}`” at line 8. Either way, the `rest.js` script, which is the jQuery/Ajax script for this demo, will be triggered when the HTML or jsp file is accessed.

In addition to the above subtlety, note the following in Listing 6.4:

- Line 7 enables jQuery library by instructing downloading `jquery.min.js` from Google’s *ajax* website.
- Lines 13 and 14 define two variables named “transactionId” and “amount,” respectively. The values of these two variables will be used to display the two lines there, after the `rest.js` jQuery/Ajax script is triggered and executed.

Next, we discuss the `rest.js` jQuery/Ajax script mentioned above.

Listing 6.4 `rest.jsp`

```

1 <%@ include file = "../WEB-INF/jsp/include.jsp" %>
2 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <!DOCTYPE html>
4 <html>
5   <head>
6     <title>Hello jQuery for Spring REST</title>
7     <script
8       src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></
9     script>
10    <script src="{ctx}/rest/js/rest.js"></script>
11  </head>
12  <body>
13    <div>
14      <p class="transactionId">The tx ID is </p>
15      <p class="amount">The amount is </p>
16    </div>
17  </body>
18 </html>

```

6.5.2 JQuery/Ajax

jQuery, as introduced at <https://jquery.com>, is a fast, compact JavaScript library, which helps simplify developing JavaScript-based UIs. Its APIs work across all common browsers for common operations such as:

- Navigating and Manipulating DOM elements through jQuery Objects
- Accessing DOM elements through a concept of *selectors*
- Handling events for richly interactive Web pages
- Dynamically accessing and manipulating Web pages
- Accessing data outside the Web page using *Screen* objects, *Window* objects, browser *Location* objects and browser *History* objects, and so on
- Enhancing user interaction through animation and other special effects
- Interacting with Web forms

- Creating advanced Web page elements
- Accessing server-side data via Ajax

The other companion part of jQuery is *jQueryUI*, which is available at <https://jqueryui.com>. As is documented there, jQueryUI is divided into the following categories:

- **Interactions:**
 - Draggable
 - Droppable
 - Resizable
 - Selectable
 - Sortable
- **Widgets:**
 - Accordion
 - Autocomplete
 - Button
 - Datepicker
 - Dialog
 - Menu
 - Progressbar
 - Selectmenu
 - Slider
 - Spinner
 - Tabs
 - Tooltip
- **Effects:**
 - Add Class
 - Color Animation
 - Easing
 - Effect
 - Hide
 - Remove Class
 - Show
 - Switch Class
 - Toggle
 - Toggle Class
- **Utilities:**
 - Position
 - Widget Factory

If you have some UI programming experience, the above concepts should sound familiar and clear to you. If you don't, don't worry as we will not use them in our demo. Instead, we will use the jQuery's `$.ajax` function as described below.

The jQuery's `$.ajax` function is documented at <http://api.jquery.com/jquery.ajax/>. This function performs *asynchronous HTTP (Ajax) requests*. Its format is as simple as follows:

```
jQuery.ajax(url [, settings])
```

Here, `url` is a string representing the URL to send the request to, while `settings` is a `PlainObject` that represents a set of key/value pairs that configure the Ajax request. All settings are optional, although missing some settings may result in a non-functioning UI, as we will see with this demo. Some of the optional settings include:

- **async** (default: *true*): By default, all requests are sent asynchronously. Set it to *false* for synchronous calls.
- **cache** (default: *true*): By default, the result is cached at the browser level.
- **dataType** (default: *Intelligent Guess* (xml, json, script, or html)): The type of data expected from the server. If none is specified, jQuery will try to infer it based on the MIME type of the response (an XML MIME type will yield XML, in 1.4 JSON will yield a JavaScript object, in 1.4 script will execute the script, and anything else will be returned as a string). The available types (and the result passed as the first argument to your success callback) are:
 - "xml": Returns an XML document that can be processed via jQuery.
 - "html": Returns HTML as plain text; included script tags are evaluated when inserted in the DOM.
 - "script": Evaluates the response as JavaScript and returns it as plain text. Disables caching by appending a query string parameter, `_=[TIMESTAMP]`, to the URL unless the cache option is set to *true*.
 - "json": Evaluates the response as JSON and returns a JavaScript object. Cross-domain "json" requests are converted to "jsonp" unless the request includes `jsonp: false` in its request options. The JSON data is parsed in a strict manner; any malformed JSON is rejected and a parse error is thrown. As of jQuery 1.9, an empty response is also rejected; the server should return a response of null or `{}` instead.
 - "jsonp": Loads in a JSON block using JSONP. Adds an extra `"?callback=?"` to the end of the URL to specify the callback. Disables caching by appending a query string parameter, `_=[TIMESTAMP]`, to the URL unless the cache option is set to *true*.
 - "text": A plain text string
- **password**: A password to be used with `XMLHttpRequest` in response to an HTTP access authentication
- **username**: A username to be used with `XMLHttpRequest` in response to an HTTP access authentication

As you see, the `dataType` parameter is emphasized more than others with a more detailed description, which is because the demo we will present would not work without specifying this parameter.

Now, Listing 6.5 shows the `rest.js` script. Compare this listing with Listing 6.2. What do you think? I am asking this question, because you may notice a drastic programming paradigm shift from Java to

jQuery/Ajax! It's not well-structured, object-oriented programming style any more. In order to understand this programming paradigm shift, next, let's have a general understanding of the `$.ajax()` function first. Then we come back and explain how the jQuery/Ajax script shown in Listing 6.5 works.

Listing 6.5 The jQuery/Ajax script (rest.js)

```

1  $.support.cors = true;
2  $(document).ready(function() {
3      $.ajax({
4          url: "http://localhost:8080/soba/restTx/txID/161990136",
5          dataType: 'json'
6      }).then(function(data) {
7          alert(data.amount);
8          $('#transactionId').append(data.transactionId);
9          $('#amount').append(data.amount);
10     });
11 });

```

6.5.3 The `$.ajax()` Function

The `$.ajax()` function supports all Ajax requests sent by jQuery. At its simplest, the `$.ajax()` function can be called with no arguments as follows:

```
$.ajax();
```

The above statement, using no options, loads the contents of the current page, but does nothing with the result. To consume the result, you need to implement proper callback functions through the `jqXHR` object, included in the jQuery XMLHttpRequest (`jqXHR`) object returned by `$.ajax()`.

As of jQuery 1.5, the `jqXHR` objects returned by `$.ajax()` implement the *Promise* interface, giving the `jqXHR` objects all the properties, methods, and behavior of a Promise. Available Promise methods of the `jqXHR` object include:


- **jqXHR.done**(function(data, textStatus, jqXHR) {}): This method instructs to add handlers to be called when the Deferred object is resolved.
- **jqXHR.fail**(function(jqXHR, textStatus, errorThrown) {}): This method instructs to add handlers to be called when the Deferred object is rejected.
- **jqXHR.always**(function(data|jqXHR, textStatus, jqXHR|errorThrown) { }): This method instructs to add handlers to be called when the Deferred object is either resolved or rejected.

In response to a successful request, the function's arguments are the same as those of `.done()`: `data`, `textStatus`, and the `jqXHR` object. For failed requests the arguments are the same as those of `.fail()`: the `jqXHR` object, `textStatus`, and `errorThrown`.

In addition, the **jqXHR.then**(function(data, textStatus, jqXHR) {}, function(jqXHR, textStatus, errorThrown) {}) incorporates the functionality of the `.done()` and `.fail()` methods, allowing (as of jQuery 1.8) the underlying Promise to be manipulated. It instructs to add handlers to be called when the Deferred object is resolved, rejected, or still in progress.

The `done`, `fail` and `then` methods are implemented by the Deferred Object underneath, which is, as of jQuery 1.5, a chainable utility object created by calling the `jQuery.Deferred()` method. It can register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function. These methods take one or more function arguments that are called when the `$.ajax()` request terminates. This allows you to assign multiple callbacks on a single request, and even to assign callbacks after the request may have completed. (If the request is already complete, the callback is fired immediately.)

Both a Deferred object and a jQuery object are chainable. After creating a Deferred object, you can use any of their methods by either chaining directly from the object creation or saving the object in a variable and invoking one or more methods on that variable. See <http://api.jquery.com/category/deferred-object/> for more information on Deferred object and all relatable methods.

 **Note:** Deprecation Notice: The `jqXHR.success()`, `jqXHR.error()`, and `jqXHR.complete()` callbacks are deprecated as of jQuery 1.8. Use `jqXHR.done()`, `jqXHR.fail()`, and `jqXHR.always()`, respectively, instead.

Now, we are ready to examine the jQuery/Ajax script shown in Listing 6.5. For convenience, it is copied below. It is explained as follows:


- Line 1: This is to prevent the error of “*Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at ...*” from happening. The term *CORS* stands for *Cross-Origin Resource Sharing*.
- Line 2: This is jQuery’s `$(document).ready()` function. A page can’t be manipulated safely until the document is “*ready*.” jQuery detects the state of readiness, after which, code included inside `$(document).ready()` will be run once the page Document Object Model (DOM) is ready for JavaScript code to execute. Similarly, when the entire page including not only DOM, but also images, iframes, and so on, is ready, code included inside `$(window).load(function() { ... })` will run.
- Line 3: This is the beginning of the `$.ajax` function we introduced above.
- Line 4: This is the same URL for the RESTful Web services as we discussed in the preceding sections.
- Line 5: This is the `dataType` parameter, implying that the response should be treated as a JSON object. We’ll come back to this later.
- Line 6: This is the “*then*” method we discussed previously. It’s an all-in-one catch for all of the *done*, *fail*, and *in-progress* events. Note that `data` here represents the result of the Ajax call.
- Line 7: This is the alert function that will pop up a dialog window with the parameter object displayed on the window. Execution will be paused until the OK button is clicked, as we will see later.
- Line 8: This statement binds the value of `data.transactionId` to the `transactionId` variable on the HTML/JSP side, as shown in Listing 6.4.
- Line 9: This line is similar to line 8 except it is for the `amount` attribute.
- Lines 10 and 11: Just closing parenthesis and brackets

Next, we discuss the Restful Web Services expressed at line 4.

```

1 $.support.cors = true;
2 $(document).ready(function() {
3     $.ajax({
4         url: "http://localhost:8080/soba/restTx/txID/161990136",
5         dataType: 'json'
6     }).then(function(data) {
7         alert(data.amount);
8         $('#transactionId').append(data.transactionId);
9         $('#amount').append(data.amount);
10    });
11 });

```

 **Note:** The AJAX requests out of the `$.ajax()` function used as above are considered low-level AJAX requests. Some higher-level functions, such as `.load()`, `.get()`, and `.post()`, are available as easier-to-use equivalent wrappers.

6.5.4 RESTful Web Services

For this demo, we use the same RESTful Web services as discussed in §6.3, namely, the `RestTxController` class. It provides APIs for retrieving banking transactions by ID and inserting transactions, and so on. Since we already introduced it in §6.3, we would not spend more time here on it.

6.5.5 Spring Wiring

A question is how we can wire it into the SOBA sample that you are already familiar with at this point. Here is a list of changes I made in order to make that happen:

- Adding a servlet for handling JavaScript scripts ending in “.js:” This is done in `web.xml` by adding the following servlet-mapping. If it were not added, the servlet engine would complain that the resource for the `rest.js` script could not be found.

```

<servlet-mapping>
  <servlet-name>DefaultServlet</servlet-name>
  <url-pattern>*.js</url-pattern>
</servlet-mapping>

```

- Where to instantiate it? I modified the original dummy `Messages` tab in the `activityList.jsp` file so that after a user logs in and clicks on it, it will trigger the `rest.jsp` file:

```

<td > <a href="{ctx}/rest/rest.jsp">jQuery-ajax-test</a> </td>

```

Note that the `rest.jsp` and `rest.js` files must be dropped to a servlet engine’s `webapps` folder in order to test it. We cannot simply put them in an arbitrary folder, click on the HTML or jsp file and expect that the JavaScript/Ajax script would be triggered.

Next, we discuss how to test this demo.

6.5.6 Testing

Here is the procedure for testing how Spring RESTful Web services can be consumed by using jQuery/Ajax based clients:

1. After building SOBA successfully, deploy it to Tomcat as described in Chapter 2.
2. Start up Tomcat
3. Login as an existing user
4. Identify the last tab named *jQuery-ajax-test*. Before clicking on it, edit the *rest.js* file in the *webapps\soba\rest* folder and replace the *transactionId* with one that exists in your database. Save your change to the *rest.js* file.
5. Now click on the *jQuery-ajax-test* tab, and you should see a dialog similar to Figure 6.2. Click on OK and you should get the final result similar to Figure 6.3.

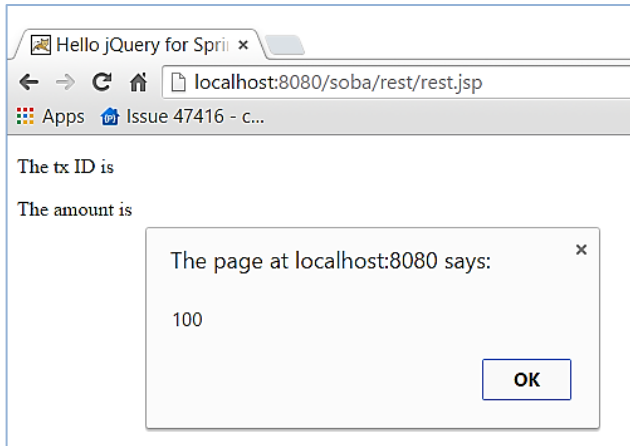


Figure 6.2 Dialog from the JavaScript/Ajax demo

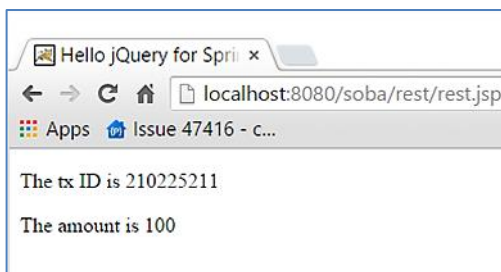


Figure 6.3 Result after the jQuery/Ajax script was executed successfully

Before concluding this demo, I'd like to ask you to do one experiment, namely, remove line 5 as shown below and retry the test (you also need to remove the comma sign at the end of the preceding line):

```
dataType: 'json'
```

You would get a dialog box as shown in Figure 6.4, which shows “undefined” for the amount attribute.

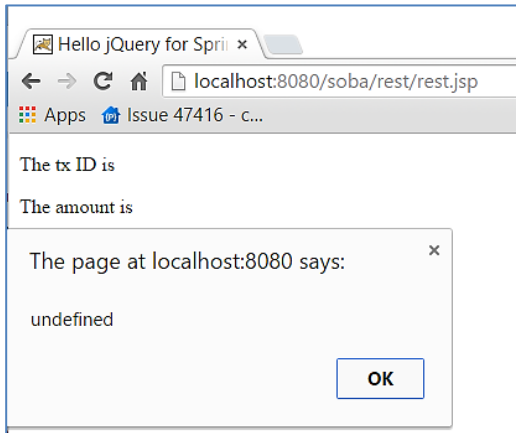


Figure 6.4 Result if the `dataType` parameter specified as ‘`json`’ were removed in the jQuery/Ajax script

Now, if you copy and paste the URL in the `rest.js` file directly in the address box of your browser, hit *Return* and you might get an XML response similar to Figure 6.5. This explains why the `dataType` parameter for the `$.ajax()` call needs to be specified if a non-XML response is expected.

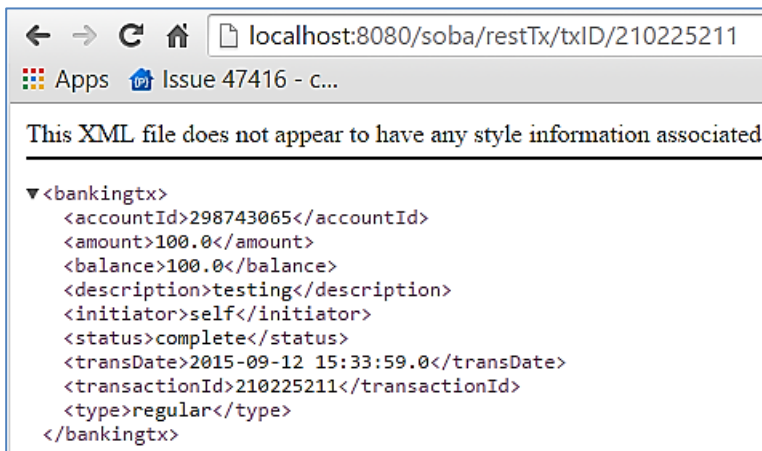


Figure 6.5 Result if the `dataType` parameter specified as ‘`json`’ were removed in the `$.ajax()` call

This concludes our demo of using jQuery/Ajax to consume Spring RESTful Web services.