Mastering Grails: Asynchronous Grails with JSON and Ajax

Google Maps mashups in Grails

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

18 Nov 2008

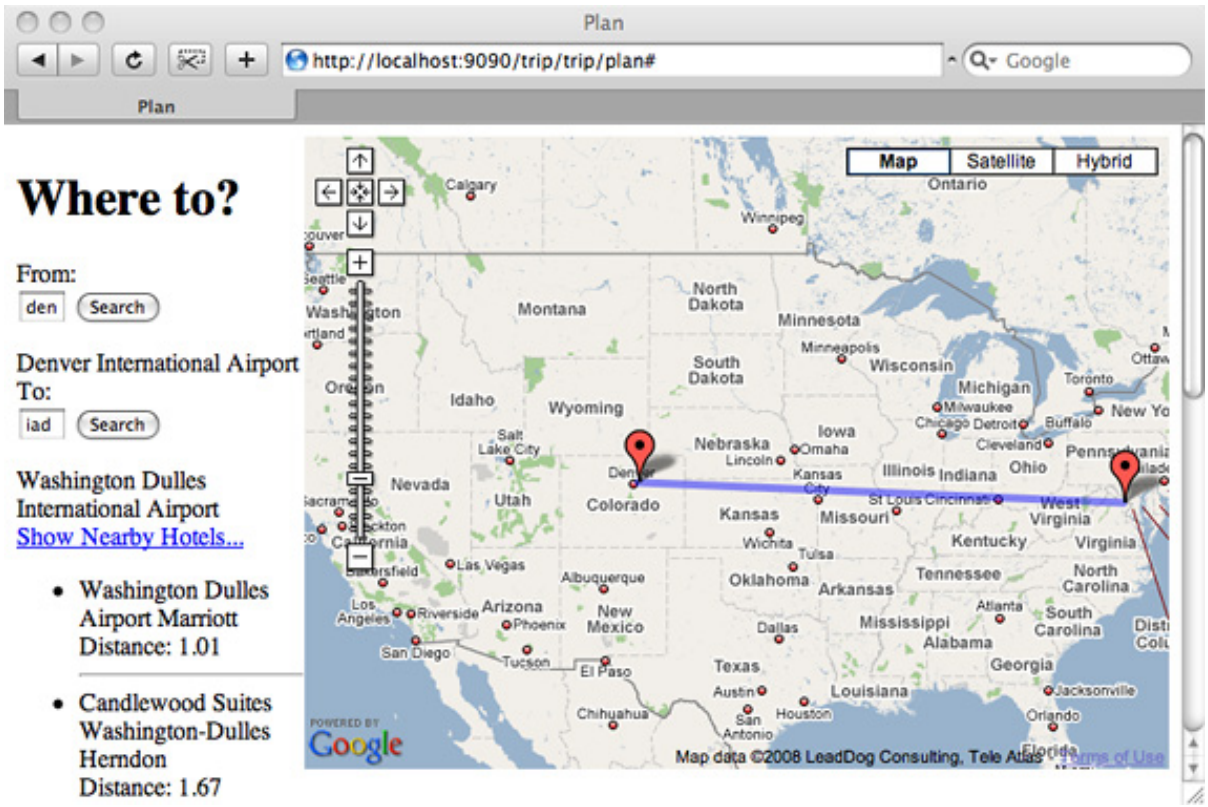
JavaScript Object Notation (JSON) and Asynchronous JavaScript + XML (Ajax) are staples of Web 2.0 development. In this installment of the *Mastering Grails* series, author Scott Davis demonstrates the native JSON and Ajax capabilities baked into the Web framework.

This article discusses Grails support for the complementary technologies JSON and Ajax. After playing supporting roles in previous *Mastering Grails* articles, this time they take center stage. You will make an Ajax request using the included Prototype library as well as the Grails `<formRemote>` tag. You'll also see examples of both serving up local JSON and dynamically pulling in remote JSON from across the Web.

To see all this in action, you'll put together a trip-planning page in which the user can type in a source and destination airport. After the airports are displayed on a Google Map, a link lets them search for hotels near the destination airport. Figure 1 shows this page in use:

Figure 1. Trip-planning page

© Copyright IBM Corporation 1994, 2008. All rights reserved.



You can achieve all of this functionality in about 150 lines of code spread across a single GSP file and three controllers.

A brief history of Ajax and JSON

About this series

Grails is a modern Web development framework that mixes familiar Java™ technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails give you seamless integration with your legacy Java code while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

When the Web first rose to popularity in the mid-1990s, browsers allowed only coarse-grained HTTP requests. Clicking on a hyperlink or a form-submit button caused the entire page to be erased and replaced by the new results. This was fine for page-centric navigation, but individual components on the page couldn't update themselves independently.

Microsoft® introduced the `XMLHTTP` object with the release of Internet Explorer 5.0 in 1999. This new object gave developers the ability to make "micro" HTTP requests, leaving the surrounding HTML page in place. Although this feature wasn't based on

© Copyright IBM Corporation 1994, 2008. All rights reserved.

a World Wide Web Consortium (W3C) standard, the Mozilla team recognized its potential and added an `XMLHttpRequest` (XHR) object to the 2002 release of Mozilla 1.0. It has since become a de facto standard, present in every major Web browser.

In 2005, Google Maps was released to the general public. Its extensive use of asynchronous HTTP requests put it in stark contrast with the other Web mapping sites of the day. Instead of clicking and waiting for the entire page to reload as you pan a Google Map, you seamlessly scroll around the map with your mouse. Jesse James Garrett used the catchy mnemonic Ajax in a blog post describing the collection of technologies used in Google Maps, and the name has stuck ever since (see Resources).

In recent years, Ajax has come to be more of a loose umbrella term for "Web 2.0" applications than a specific list of technologies. The requests are usually asynchronous and made with JavaScript, but the response isn't always XML. The problem with XML in browser-based application development is the lack of a native, easy-to-use JavaScript parser. It's certainly possible to parse XML using the JavaScript DOM API, but it isn't easy for the novice. Consequently, Ajax Web services frequently return results in plain text, HTML snippets, or JSON.

In July 2006, Douglas Crockford submitted RFC 4627 to the Internet Engineering Task Force (IETF) describing JSON. By the end of that year, major service providers such as Yahoo! and Google were offering JSON output as an alternative to XML (see Resources). (You'll take advantage of Yahoo!'s JSON Web services later in this article.)

The benefits of JSON

JSON offers two major advantages over XML when it comes to Web development. First of all, it is less verbose. A JSON object is simply a series of comma-separated *`name`*`:`*`value`* pairs wrapped in curly braces. In contrast, XML uses duplicate start and end tags to wrap data values. This yields twice the metadata overhead than the corresponding JSON, inspiring Crockford to call JSON "the fat-free alternative to XML" (see Resources). When you are dealing with the "thin pipe" of Web development, every reduction in bytes pays real performance dividends.

Listing 1 shows how JSON and XML organize the same information:

Listing 1. Comparing JSON and XML

```
{"city":"Denver", "state":"CO", "country":"US"}

<result>
  <city>Denver</city>
  <state>CO</state>
```


© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
  <country>US</country>
</result>
```

JSON objects should look familiar to Groovy programmers: if you replace the curly braces with square brackets, you'd be defining a `HashMap` in Groovy. And speaking of square brackets, an array of JSON objects is defined in exactly the same way as an array of Groovy objects. A JSON array is simply a comma-separated series wrapped in square brackets, as shown in Listing 2:

Listing 2. A list of JSON objects

```
[{"city":"Denver", "state":"CO", "country":"US"},
 {"city":"Chicago", "state":"IL", "country":"US"}]
```

JSON's second benefit becomes evident when you parse it and work with it. Loading JSON into memory is one `eval()` call away. Once it is loaded, you can directly access any field by name, as shown in Listing 3:

Listing 3. Loading JSON and calling fields

```
var json = '{"city":"Denver", state:"CO", country:"US"}'
var result = eval( '(' + json + ')' )
alert(result.city)
```

Groovy's `XmlSlurper` gives you the same direct access to XML elements. (You worked with `XmlSlurper` in "Grails services and Google Maps.") If modern Web browsers supported client-side Groovy, I'd be far less interested in JSON. Sadly, Groovy is strictly a server-side solution. JavaScript is the only game in town when it comes to client-side development. So I prefer working with XML in Groovy on the server side and JSON in JavaScript on the client side. In both cases, I can get my hands on the data with a minimal effort.

Now that you've gotten a glimpse of JSON, it's time to have your Grails application produce some JSON of your own.

Rendering JSON in a Grails controller

You first returned JSON from a Grails controller in "Many-to-many relationships with a dollop of Ajax." The closure in Listing 4 is similar to the one that you created back then. The difference is that this one is accessed via a friendly Uniform Resource Identifier (URI), as discussed in "RESTful Grails." It also uses the Elvis operator you first saw in "Testing your Grails application."

Add a closure called `iata` to the grails-app/controllers/AirportMappingController.groovy class you created in "Grails

© Copyright IBM Corporation 1994, 2008. All rights reserved.

and legacy databases," remembering to import the `grails.converters` package at the top of the file, as shown in Listing 4:

Listing 4. Converting Groovy objects to JSON

```
import grails.converters.*
class AirportMappingController {
    def iata = {
      def iata = params.id?.toUpperCase() ?: "NO IATA"
      def airport = AirportMapping.findByIata(iata)
      if(!airport){
        airport = new AirportMapping(iata:iata, name:"Not found")
      }
      render airport as JSON
    }
}
```

Try it out by typing `http://localhost:9090/trip/airportMapping/iata/den` in your browser. You should see the JSON results shown in Listing 5:

Listing 5. A valid AirportMapping object in JSON

```
{"id":328,
"class":"AirportMapping",
"iata":"DEN",
"lat":"39.858409881591797",
"lng":"-104.666999816894531",
"name":"Denver International",
"state":"CO"}
```

You can also type `http://localhost:9090/trip/airportMapping/iata` and `http://localhost:9090/trip/airportMapping/iata/foo` to make sure that "Not Found" is returned. Listing 6 shows the resulting invalid JSON object:

Listing 6. An invalid AirportMapping object in JSON

```
{"id":null,
"class":"AirportMapping",
"iata":"FOO",
"lat":null,
"lng":null,
"name":"Not found",
"state":null}
```

Of course, this "gut check" is no replacement for a real set of tests.

Testing the controller

Create AirportMappingControllerTests.groovy in test/integration. Add the two tests in

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Listing 7:

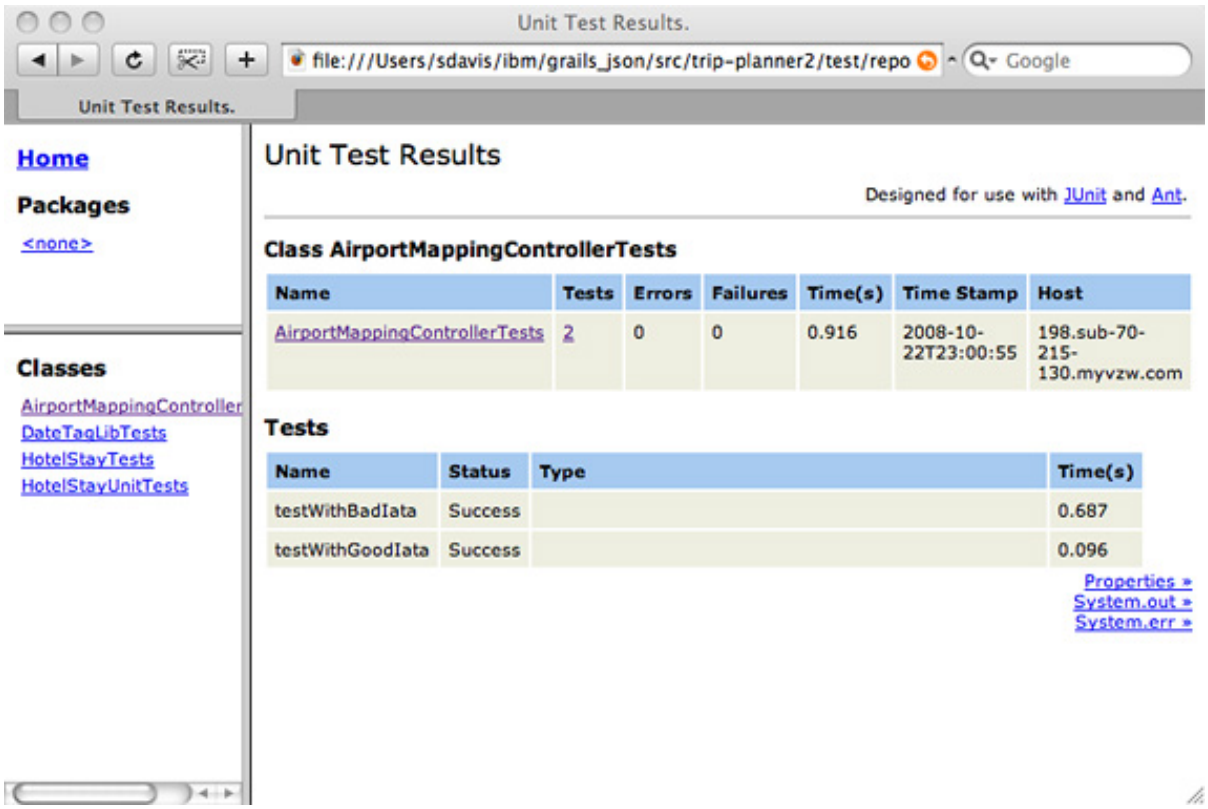
Listing 7. Testing a Grails controller

```
class AirportMappingControllerTests extends GroovyTestCase{
  void testWithBadIata(){
    def controller = new AirportMappingController()
    controller.metaClass.getParams = {->
      return ["id":"foo"]
    }
    controller.iata()
    def response = controller.response.contentAsString
    assertTrue response.contains("\"name\":\"Not found\"")
    println "Response for airport/iata/foo: ${response}"
  }
  void testWithGoodIata(){
    def controller = new AirportMappingController()
    controller.metaClass.getParams = {->
      return ["id":"den"]
    }
    controller.iata()
    def response = controller.response.contentAsString
    assertTrue response.contains("Denver")
    println "Response for airport/iata/den: ${response}"
  }
}
```

Type `$grails test-app` to run the tests. You should see the successes in the JUnit HTML reports, as in Figure 2. (For a refresher on testing Grails applications, see "Testing your Grails application.")

Figure 2. Passing tests in JUnit

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Here's what's going on in `testWithBadIata()` in Listing 7. The first line (obviously) creates an instance of the `AirportMappingController`. You do this so that later you can call `controller.iata()` and write an assertion against the resulting JSON. To make the call fail (in this case) or succeed (in the case of `testWithGoodIata()`), you need to seed the `params` hashmap with an `id` entry. Normally the query string gets parsed and stored in `params`. In this case, however, there's no HTTP request to get parsed. Instead, I use Groovy metaprogramming to override the `getParams` method directly, forcing the expected values to be present in the `HashMap` that's returned. (For more on metaprogramming in Groovy, see Resources.)

Now that the JSON producer is working and tested, it's time to focus on consuming the JSON from a Web page.

Setting up the initial Google Map

I want the planning page to be available at http://localhost:9090/trip/trip/plan. This means adding a `plan` closure to grails-app/controllers/TripController.groovy, as shown in Listing 8:

Listing 8. Setting up the controller

© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
class TripController {
  def scaffold = Trip
  def plan = {}
}
```

Because `plan()` doesn't end with a `render()` or a `redirect()`, convention over configuration dictates that grails-app/views/trip/plan.gsp will be displayed. Create the file using the HTML code in Listing 9. (To review the basics behind this Google Map, see "Grails services and Google Maps.")

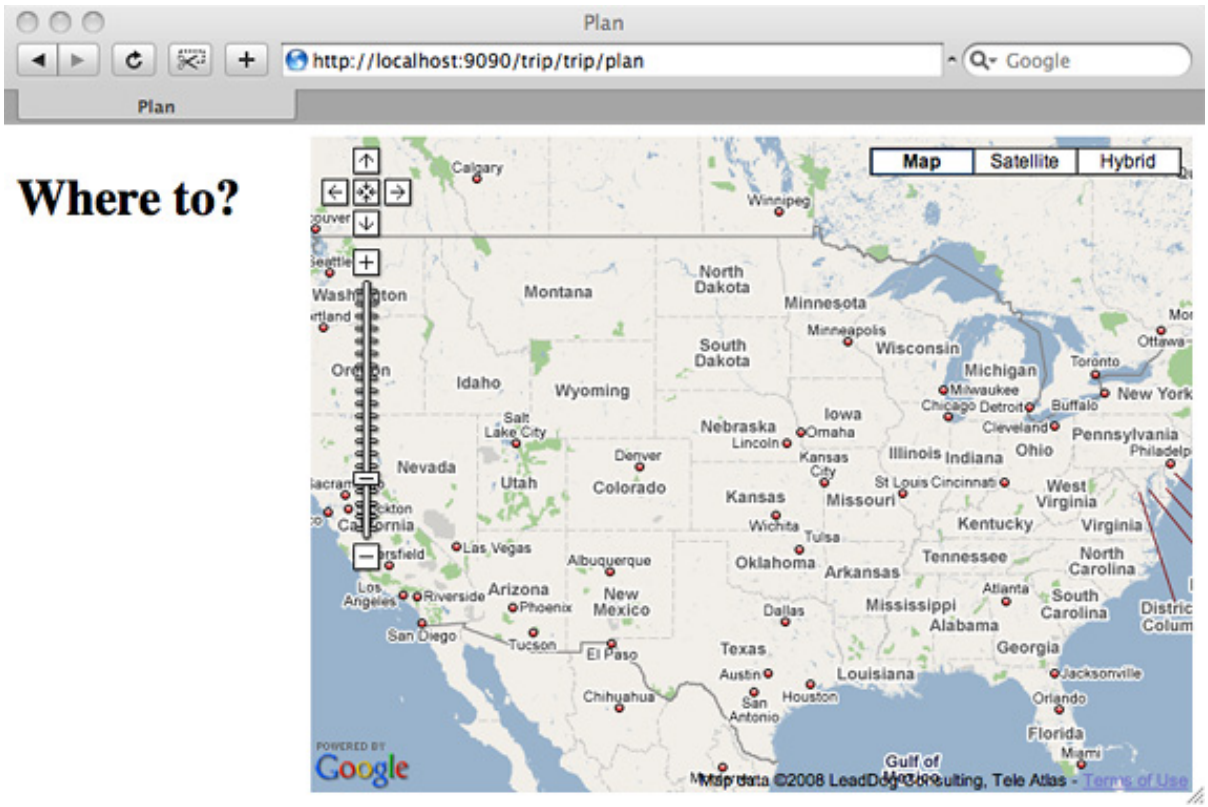
Listing 9. Setting up the initial Google Map

```
<html>
  <head>
    <title>Plan</title>
    <script src="http://maps.google.com/maps?file=api&v=2&key=YourKeyHere"
      type="text/javascript"></script>
    <script type="text/javascript">
    var map
    var usCenterPoint = new GLatLng(39.833333, -98.583333)
    var usZoom = 4
    function load() {
      if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"))
        map.setCenter(usCenterPoint, usZoom)
        map.addControl(new GLargeMapControl());
        map.addControl(new GMapTypeControl());
      }
    }
    </script>
  </head>
  <body onload="load()" onunload="GUnload()">
    <div class="body">
      <div id="search" style="width:25%; float:left">
      <h1>Where to?</h1>
      </div>
      <div id="map" style="width:75%; height:100%; float:right"></div>
    </div>
  </body>
</html>
```

If all goes well, visiting http://localhost:9090/trip/trip/plan in your browser should give you something that looks like Figure 3:

Figure 3. A plain Google Map

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Now that the basic map is in place, you should add a couple of fields for the source and destination airports.

Adding the form fields

In "Many-to-many relationships with a dollop of Ajax," you used Prototype's `Ajax.Request` object. You'll use it again later in this article when you get some JSON from a remote source. In the meantime, you'll take advantage of the `<g:formRemote>` tag. Add the HTML in Listing 10 to grails-app/views/trip/plan.gsp:

Listing 10. Using <g:formRemote>

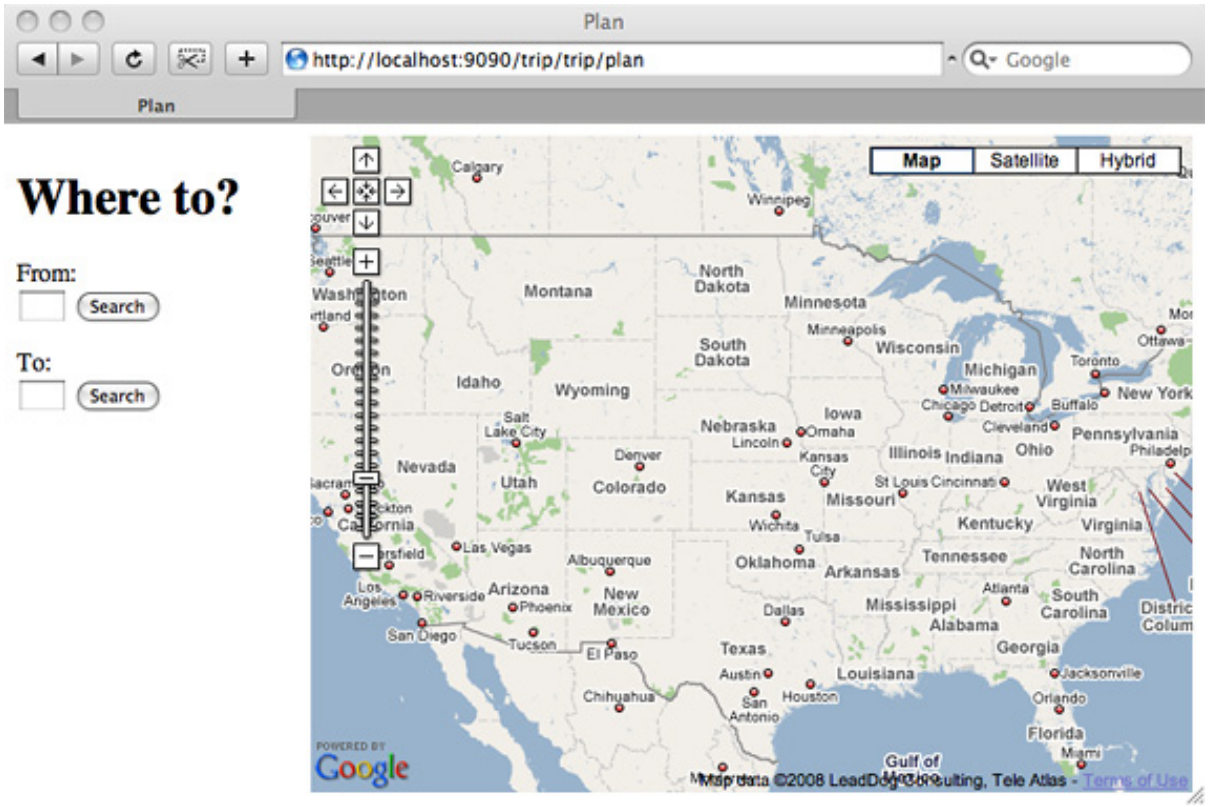
```
<div id="search" style="width:25%; float:left">
<h1>Where to?</h1>
<g:formRemote name="from_form"
              url="[controller:'airportMapping', action:'iata']"
              onSuccess="addAirport(e, 0)">
  From:<br/>
  <input type="text" name="id" size="3"/>
  <input type="submit" value="Search" />
</g:formRemote>
<div id="airport_0"></div>
<g:formRemote name="to_form"
              url="[controller:'airportMapping', action:'iata']"
              onSuccess="addAirport(e, 1)">
  To: <br/>
  <input type="text" name="id" size="3"/>
```


© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
  <input type="submit" value="Search" />
</g:formRemote>
<div id="airport_1"></div>
</div>
```

Click the **Refresh** button in your Web browser to see the new changes, shown in Figure 4:

Figure 4. Adding the form fields



Using a normal `<g:form>` would cause the entire page to refresh when the user submits the form. By choosing `<g:formRemote>`, you make an `Ajax.Request` perform the form submission asynchronously behind the scenes. The input text field is named `id`, ensuring that `params.id` will be populated in the controller. The `url` attribute on `<g:formRemote>` clearly shows you that `AirportMappingController.iata()` will be called when the user clicks the submit button.

You couldn't take advantage of `<g:formRemote>` in "Many-to-many relationships with a dollop of Ajax" because you can't nest one HTML form inside another HTML form. Here, though, you can create two separate forms and not worry about having to write the Prototype code yourself. The results of the asynchronous JSON request will be passed to the `addAirport()` JavaScript function.

Your next task is to create `addAirport()`.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Adding the JavaScript to handle the JSON

The `addAirport()` function that you are about to create does two simple things: it loads the JSON object into memory and then uses the fields for various purposes. In this case, you use the latitude and longitude values to create a `GMarker` and add it to the map.

For `<g:formRemote>` to work, be sure that you include the Prototype library at the top of the head section, as shown in Listing 11:

Listing 11. Including Prototype in a GSP

```
<g:javascript library="prototype" />
```

Next, add the JavaScript in Listing 12 after the `init()` function:

Listing 12. Implementing addAirport and drawLine

```
<script type="text/javascript">
var airportMarkers = []
var line
function addAirport(response, position) {
  var airport = eval('(' + response.responseText + ')')
  var label = airport.iata + " -- " + airport.name
  var marker = new GMarker(new GLatLng(airport.lat, airport.lng), {title:label})
  marker.bindInfoWindowHtml(label)
  if(airportMarkers[position] != null){
    map.removeOverlay(airportMarkers[position])
  }
  if(airport.name != "Not found"){
    airportMarkers[position] = marker
    map.addOverlay(marker)
  }
  document.getElementById("airport_" + position).innerHTML = airport.name
  drawLine()
}
function drawLine(){
  if(line != null){
    map.removeOverlay(line)
  }

  if(airportMarkers.length == 2){
    line = new GPolyline([airportMarkers[0].getLatLng(), airportMarkers[1].getLatLng()])
    map.addOverlay(line)
  }
}
</script>
```

The first thing the code in Listing 12 does is declare a couple of new variables: one to hold the line and an array to hold the two airport markers. After you `eval()` the incoming JSON, you call the fields such as `airport.iata`, `airport.name`, `airport.lat`, and `airport.lng` directly. (For a reminder of what the JSON object looks like, see Listing 5.)

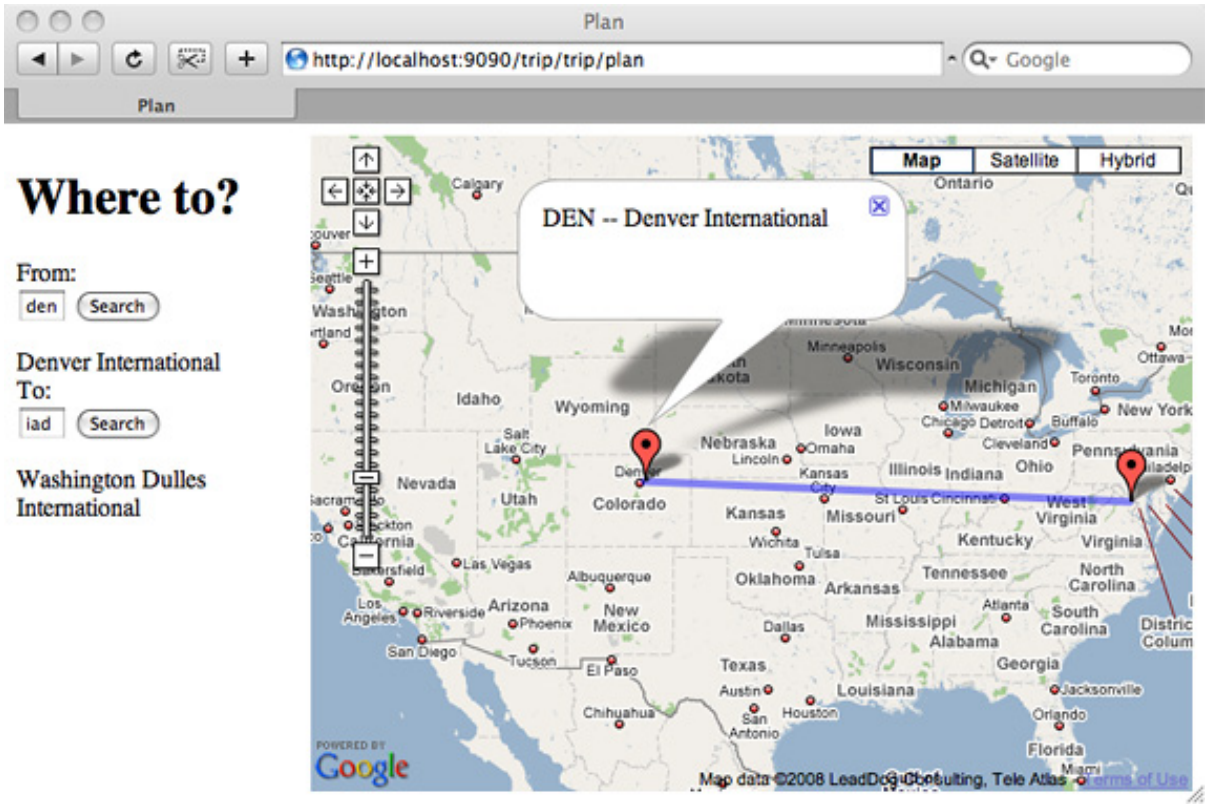
© Copyright IBM Corporation 1994, 2008. All rights reserved.

Once you have a handle to the `airport` object, you create a new `GMarker`. This is the familiar "red push pin" you are used to seeing on Google Maps. The `title` attribute tells the API what to display as a tooltip when the user's mouse hovers over the marker. The `bindInfoWindowHtml()` method tells the API what to display when the user clicks on the marker. Once the marker is added to the map as an overlay, the `drawLine()` function is called. As the name suggests, it draws a line between the two airport markers if they both exist.

For more information on the Google Maps API objects such as `GMarker`, `GLatLng`, and `GPolyline`, see the online documentation (see Resources).

Entering a couple of airports should make the page look like Figure 5:

Figure 5. Displaying two airports with a line drawn between them



Don't forget to refresh the Web browser each time you make changes to the GSP file.

Now that you have an example of using JSON returned locally from your Grails application, it's time to expand your horizons a bit. In the next section, you'll dynamically get JSON from a remote Web service. Of course once you have it, you work with the same way you just did in this example: you load it into memory and directly access the various attributes.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Remote or local JSON?

Your next task is to display the 10 closest hotels to the destination airport. This will almost certainly require you to get the data remotely.

There is no standard answer to the question of whether you should host the data locally or fetch it remotely per request. In the case of the airport dataset, I feel reasonably confident hosting it locally. The data is freely available and easy enough to ingest. (The United States has only 901 airports, and the number of major airports is fairly static; the list probably won't be out of date any time soon.)

If the airport dataset were more volatile, too large to reasonably store locally, or simply not available as a single download, I'd be far more inclined to request it remotely. The geonames.org geocoding service you used in "Grails services and Google Maps" offers JSON output as well as XML (see Resources). Type `http://ws.geonames.org/search?name_equals=den&fcode=airp&style=full&type=` in your Web browser. You should see the JSON results shown in Listing 13:

Listing 13. JSON results from GeoNames

```
{"totalResultsCount":1,
"geonames":[
  {"alternateNames":[
    {"name":"DEN","lang":"iata"},
    {"name":"KDEN","lang":"icao"}],
  "adminCode2":"031",
  "countryName":"United States",
  "adminCode1":"CO",
  "fclName":"spot, building, farm",
  "elevation":1655,
  "countryCode":"US",
  "lng":-104.6674674,
  "adminName2":"Denver County",
  "adminName3":"",
  "fcodeName":"airport",
  "adminName4":"",
  "timezone":{
    "dstOffset":-6,
    "gmtOffset":-7,
    "timeZoneId":"America/Denver"},
  "fcl":"S",
  "name":"Denver International Airport",
  "fcode":"AIRP",
  "geonameId":5419401,
  "lat":39.8583188,
  "population":0,
  "adminName1":"Colorado"}]
}
```

As you can see, the GeoNames service offers more information about the airport than the dataset from the USGS that you imported in "Grails and legacy databases." If new user stories emerge, such as needing to know the airport's time zone or the elevation in meters, GeoNames offers a compelling alternative. It also includes

© Copyright IBM Corporation 1994, 2008. All rights reserved.

international airports like London Heathrow (LHR) and Frankfort (FRA). I'll leave it as an extra-credit exercise for you to convert `AirportMapping.iata()` to use GeoNames under the covers.

In the meantime, your only real option for displaying a list of hotels in close proximity to the destination airport is to take advantage of a remote Web service. There are thousands of hotels in an ever-changing list, so letting someone else take responsibility for managing this list is pretty compelling.

Yahoo! offers a local search service that allows you to search for businesses near a street address, ZIP code, or even a latitude/longitude point (see Resources). If you registered for a developer key in "RESTful Grails," you can reuse it here. Not surprisingly, the format of the generic search URI you used then and the local search you are about to use now are very similar. Last time, you allowed the Web service to return XML by default. By adding one more *name=value* pair (`output=json`), you can get JSON instead.

Type this in your browser (without the line breaks) to see a JSON list of hotels near Denver International Airport:

```
http://local.yahooapis.com/LocalSearchService/V3/localSearch?appid=
   YahooDemo&query=hotel&latitude=39.858409881591797&longitude=
   -104.666999816894531&sort=distance
```

Listing 14 shows the (truncated) JSON results:

Listing 14. JSON results from Yahoo!

```
{"ResultSet":
  {"totalResultsAvailable":"803",
  "totalResultsReturned":"10",
  "firstResultPosition":"1",
  "ResultSetMapUrl":"http:\/\/maps.yahoo.com\/broadband\/?tt=hotel&tp=1",
  "Result":[
    {"id":"42712564",
    "Title":"Springhill Suites-Denver Arprt",
    "Address":"18350 E 68th Ave",
    "City":"Denver",
    "State":"CO",
    "Phone":"(303) 371-9400",
    "Latitude":"39.82076",
    "Longitude":"-104.673719",
    "Distance":"2.63",
    [SNIP]
```

Now that you have a viable list of hotels, you need to create a controller method as you did for `AirportMapping.iata()`.

Creating the controller method to make the remote JSON

© Copyright IBM Corporation 1994, 2008. All rights reserved.

request

You should already have a `HotelController` in place from a previous article. Add the `near` closure in Listing 15 to it. (You saw similar code in "Grails services and Google Maps.")

Listing 15. The HotelController

```
class HotelController {
  def scaffold = Hotel
  def near = {
    def addr = "http://local.yahooapis.com/LocalSearchService/V3/localSearch?"
    def qs = []
    qs << "appid=YahooDemo"
    qs << "query=hotel"
    qs << "sort=distance"
    qs << "output=json"
    qs << "latitude=${params.lat}"
    qs << "longitude=${params.lng}"
    def url = new URL(addr + qs.join("&"))
    render(contentType:"application/json", text:"${url.text}")
  }
}
```

All of the query-string parameters are hardcoded except the last two: `latitude` and `longitude`. The next-to-last line instantiates a new `java.net.URL`. The last line calls the service (`url.text`) and renders the results. Because you aren't using the JSON converter, you must explicitly set the MIME-type to `application/json`. `render` returns `text/plain` unless you tell it otherwise.

Type this (without the line break) in your browser:

```
http://localhost:9090/trip/hotel/near?lat=
   39.858409881591797&lng=-104.666999816894531
```

Compare the results with the direct call that you made earlier to http://local.yahooapis.com — they should be identical.

Why can't I call remote Web services directly from the browser?

If you plugged the `local.yahooapis.com` URL into an `Ajax.Request`, it would fail silently. It works when you type it into your browser's address bar, yet it fails when you call it programmatically from JavaScript. Trust me — this is a feature, not a bug.

Specifically, Ajax requests are bound by the *same source* or *same origin* rule. This means that Ajax requests can only go back to the same domain that the source HTML page came from. In your case, that means that you can make all the calls you'd like to http://localhost, but http://local.yahooapis.com or anywhere else is

© Copyright IBM Corporation 1994, 2008. All rights reserved.

strictly off-limits.

This is done for security reasons. When you are typing your credit card number into http://amazon.com, you want to make sure that those digits aren't silently sent off to http://hackers.r.us behind the scenes as well. (This is known more formally as XSS, or cross-site scripting.)

The same-origin rule only applies to client-side JavaScript, not server-side Groovy. That's why I'm having you proxy the call to http://local.yahooapis.com from a controller and transparently pass it back to the browser.

If you absolutely want to call a Yahoo! or Google Web service directly from the browser, both provide a semi-sneaky way around the same-source rule by providing a callback option. For more information on JSON callbacks, see the documentation links Resources.

Having a controller method make the remote JSON request offers two benefits: it provides a workaround for the same-source Ajax limitation (see the Why can't I call remote Web services directly from the browser? sidebar), but — more important — it provides some encapsulation. The controller effectively becomes something akin to a Data Access Object (DAO).

You would no more want to have raw SQL in your view than you would a hardcoded URL to a remote Web service. By making a call to a local controller, you protect your downstream clients from implementation changes. A table- or field-name change would break an embedded SQL statement, and a URL change would break an embedded Ajax call. By calling `AirportMapping.iata()`, you are free to change the datasource from a local table to the remote GeoNames service while preserving the client-side interface. Long term, you might even decide to cache the calls to the remote service in a local database for performance reasons, building up your local cache one request at a time.

Now that the service is working in isolation, you can call it from the Web page.

Adding the ShowHotels link

There is no sense in displaying the Show Nearby Hotels hyperlink until the user provides the destination airport. Similarly, there is no sense in making the remote request until you are sure that the user really wants to see a list of hotels. So to start, add the `showHotelsLink()` function to the script block in plan.gsp. Also, add a call to `showHotelsLink()` to the last line of `addAirport()`, as shown in Listing 16:

Listing 16. Implementing showHotelsLink()

© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
function addAirport(response, position) {
  ...
  drawLine()
  showHotelsLink()
}
function showHotelsLink(){
  if(airportMarkers[1] != null){
    var hotels_link = document.getElementById("hotels_link")
    hotels_link.innerHTML = "<a href='#' onClick='loadHotels()'>Show Nearby Hotels...</a>"
  }
}
```

Grails provides a `<g:remoteLink>` tag that creates asynchronous hyperlinks (much as `<g:formRemote>` provides asynchronous form submissions), but life-cycle issues make them unusable here. The `g:` tags are rendered on the server. Because this link is being added dynamically on the client side, you need to rely on a pure JavaScript solution.

You probably noticed the call to `document.getElementById("hotels_link")`. Add a new `<div>` to the bottom of the `search <div>`, as shown in Listing 17:

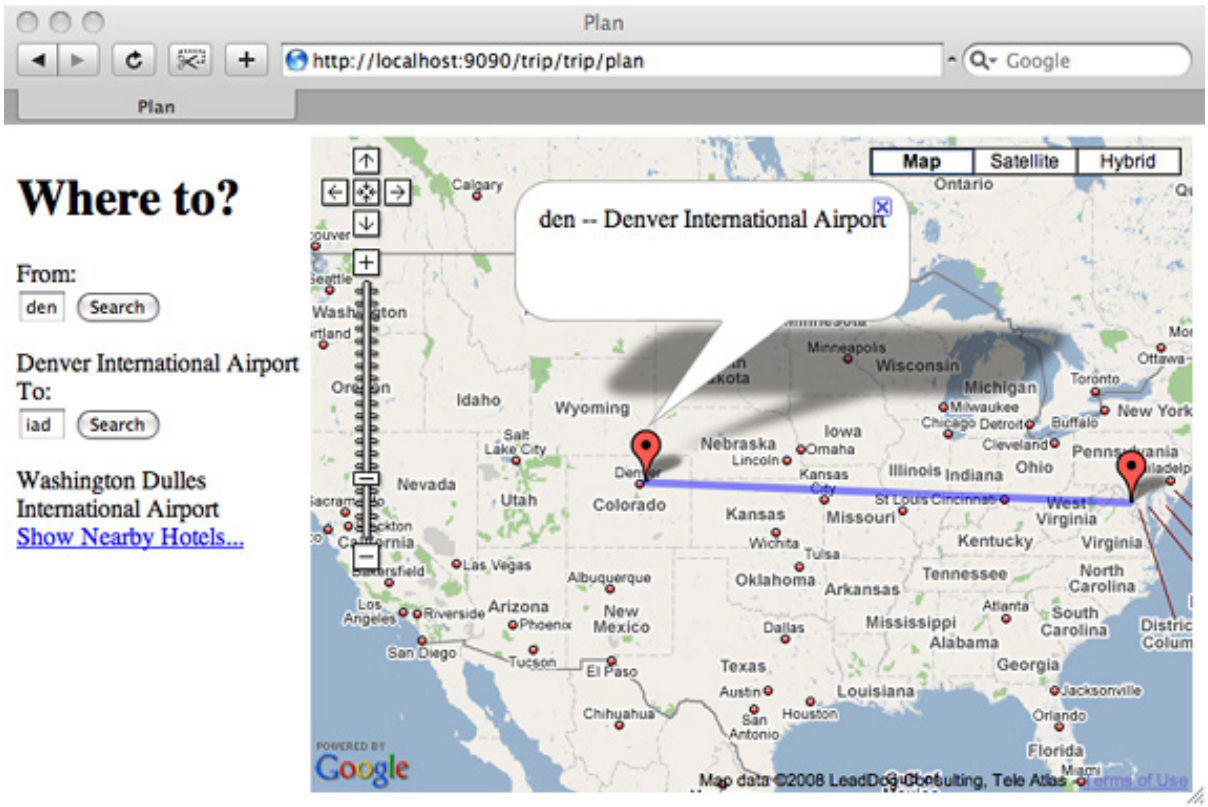
**Listing 17. Adding the hotels_link **

```
<div id="search" style="width:25%; float:left">
<h1>Where to?</h1>
<g:formRemote name="from_form" ... >
<g:formRemote name="to_form" ...>
<div id="hotels_link"></div>
</div>
```

Refresh your browser and confirm that the hyperlink appears after you provide a destination airport, as in Figure 6:

Figure 6. Displaying the Show Nearby Hotels hyperlink

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Now you need to create the `loadHotels()` function.

Making the Ajax.Remote call

Add a new function to the script block in plan.gsp, as shown in Listing 18:

Listing 18. Implementing loadHotels()

```
function loadHotels(){
  var url = "${createLink(controller:'hotel', action:'near')}"
  url += "?lat=" + airportMarkers[1].getLatLng().lat()
  url += "&lng=" + airportMarkers[1].getLatLng().lng()
  new Ajax.Request(url,{
    onSuccess: function(req) { showHotels(req) },
    onFailure: function(req) { displayError(req) }
  })
}
```

It is safe to use the Grails `createLink` method here, because the base part of the URL to `Hotel.near()` won't change when the page is rendered server-side. You append the dynamic parts of the URL using client-side JavaScript and then make the Ajax request using the by-now familiar Prototype call.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Handling errors

I ignored error handling in the `<g:formRemote>` call for brevity's sake. Now that you are making a call to a remote service (albeit via a local controller proxy), it might be more prudent to provide some feedback instead of just failing silently. Add the `displayError()` function to the script block in plan.gsp, as shown in Listing 19:

Listing 19. Implementing displayError()

```
function displayError(response){
  var html = "response.status=" + response.status + "<br />"
  html += "response.responseText=" + response.responseText + "<br />"
  var hotels = document.getElementById("hotels")
  hotels.innerHTML = html
}
```

This admittedly isn't doing much more than displaying the error to the user in the `hotels <div>` below the Show Nearby Hotels link where the results will normally appear. You're encapsulating the remote call in a server-side controller, so you might choose to do some more extensive error correction there.

Add a `hotels <div>` below the `hotels_link <div>` you added earlier, as shown in Listing 20:

**Listing 20. Adding the hotels **

```
<div id="search" style="width:25%; float:left">
<h1>Where to?</h1>
<g:formRemote name="from_form" ... >
<g:formRemote name="to_form" ...>
<div id="hotels_link"></div>
<div id="hotels"></div>
</div>
```

You have just one more thing to do: add a function to load the successful JSON request and populate the `hotels <div>`.

Handling success

This last function, shown in Listing 21, takes the JSON response from the local Yahoo! service, builds up an HTML list, and writes it to the `hotels <div>`:

Listing 21. Implementing showHotels()

```
function showHotels(response){
  var results = eval( '(' + response.responseText + ')')
```


© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
  var resultCount = 1 * results.ResultSet.totalResultsReturned
  var html = "<ul>"
  for(var i=0; i < resultCount; i++){
    html += "<li>" + results.ResultSet.Result[i].Title + "<br />"
    html += "Distance: " + results.ResultSet.Result[i].Distance + "<br />"
    html += "<hr />"
    html += "</li>"
  }
  html += "</ul>"
  var hotels = document.getElementById("hotels")
  hotels.innerHTML = html
}
```

Refresh your browser one last time and type in a couple of airports. Your screen should look like Figure 1.

I'll end the example here with the hope that you'll continue to play with it on your own. You might decide to plot the hotels on the map using another array of `GMarkers`. You could add in additional fields from the Yahoo! results such as phone number and street address. The possibilities are endless.

Conclusion

Not bad for roughly 150 lines of code, is it? In this article, you saw how JSON can be a viable alternative to XML when making Ajax requests. You saw how easy it is to return JSON locally from a Grails application, and that it's not much more difficult to return JSON from a remote Web service. You can use Grails tags like `<g:formRemote>` and `<g:linkRemote>` when the HTML is being rendered server-side, but knowing how to use the underlying `Ajax.Request` call provided by Prototype is critical for truly dynamic Web 2.0 applications.

Next time you'll see the native Java Management Extensions (JMX) capabilities of Grails in action. Until then, have fun mastering Grails.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Resources

Learn

- *Mastering Grails*: Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- Grails Framework Reference Documentation: The Grails bible.
- "Ajax: A New Approach to Web Applications" (Jesse James Garrett, Adaptive Path, February 2005): Garrett, who coined the term "Ajax," describes its basic architectural and user experience properties.
- Using JSON (JavaScript Object Notation) with Yahoo! Web Services and Using JSON with Google Data APIs: Yahoo! and Google both offer Web services output in JSON format.
- JSON: Visit the JSON site and check out "JSON: The Fat-Free Alternative to XML."
- Metaprogramming in Groovy: Learn more about Groovy's `ExpandoMetaClass`.
- Google Maps API Reference: Find out more about Google Maps API objects such as `GMarker`, `GLatLng`, and `GPolyline`.
- GeoNames Search Webservice: The GeoNames search Web service taps into a database of more than 8 million geographical names.
- Local Search Web Services: Yahoo!'s local search service lets you search for businesses near a street address, ZIP code, or latitude/longitude point.
- *Groovy Recipes*: Learn more about Groovy and Grails in Scott Davis' latest book.
- *Practically Groovy*: This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

© Copyright IBM Corporation 1994, 2008. All rights reserved.

- Grails: Download the latest Grails release.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

© Copyright IBM Corporation 1994, 2008. All rights reserved.