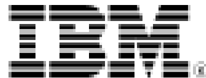


developerWorks.

Mastering Grails: Grails services and Google Maps

Mix external technologies into a Grails application

Skill Level: Introductory

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

20 May 2008

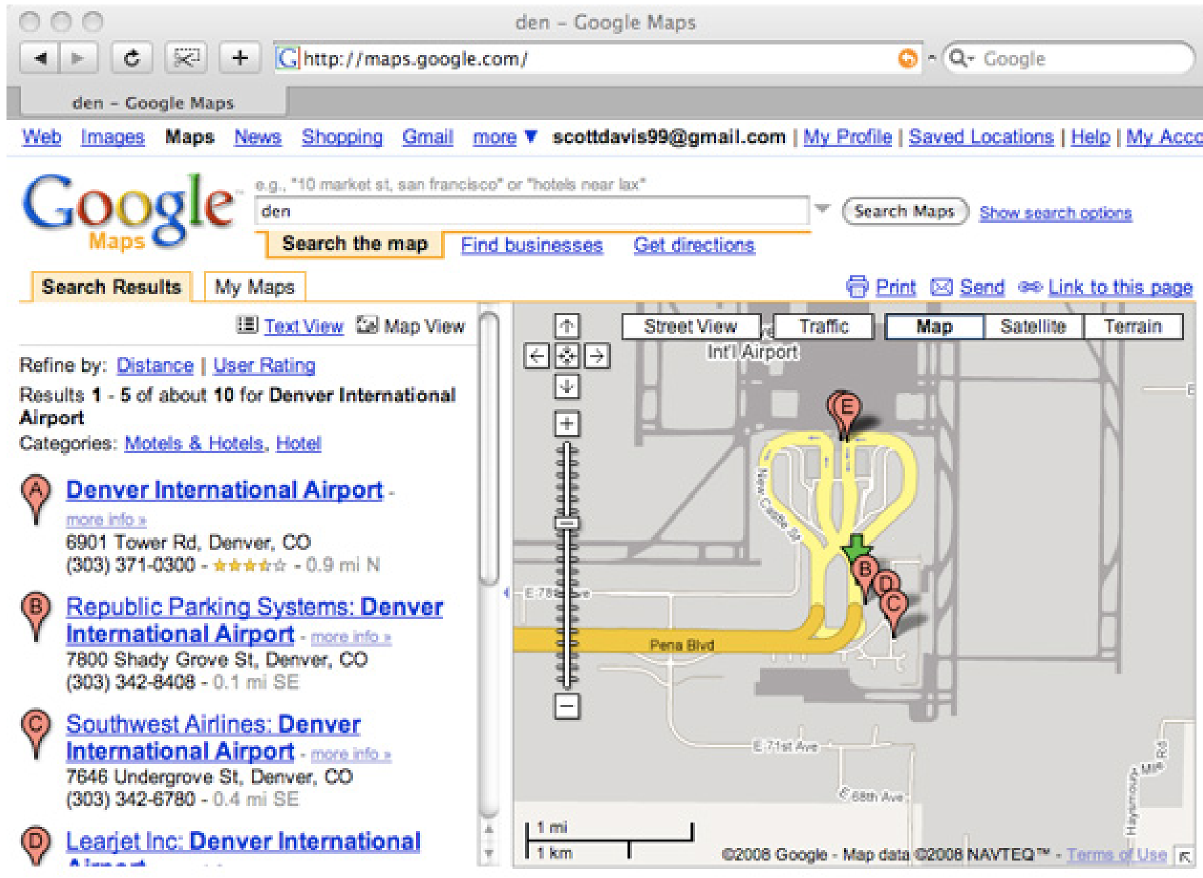
Scott Davis shows you how you can add maps to a Grails application using freely available APIs and Web services in this latest installment of *Mastering Grails*. He uses the trip-planner sample application from previous installments and takes it to the next level with geocoding, Google Maps, and Grails services.

I've been building a trip-planner application since the first article in this series. Now that the basic Model-View-Controller (MVC) framework is in place, I'm ready to mix in some external technologies. Specifically, I'm going to add a map. I can say, "I'm taking a trip from Denver to Raleigh, with stops in San Jose and Seattle along the way," but a map would help describe the trip better. You probably know that Seattle and Raleigh are on opposite sides of the United States, but a map helps you visualize the distance between the two cities.

To give you a rough idea of what the application will do by the end of this article, go to http://maps.google.com and type the IATA code `DEN` in the search box. You should end up at Denver International Airport, as shown in Figure 1. (For more on IATA codes, see last month's article.)

Figure 1. Denver Airport, courtesy of Google Maps

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



In addition to displaying the U.S. airports you create in an HTML table, the trip planner will plot the airports on a map as well. I'll use the free Google Maps API in this article. I could use the free Yahoo! Maps API or any number of others (see Resources). Once you understand the basics of online Web mapping, you'll see that the different APIs are reasonably interchangeable. Before I can get to the mapping part of the solution, you need to understand how a simple three-letter string like DEN gets converted to a point on a map.

Geocoding

When you entered DEN into Google Maps, the application did a bit of transformation behind the scenes. You may think of locations in terms of street addresses like 123 Main Street, but Google Maps needs a latitude/longitude point in order to display it on the map. Rather than forcing you to provide the latitude/longitude point yourself, it translates human-readable addresses into latitude/longitude points on your behalf. This transformation is called *geocoding* (see Resources).

> **About this series**
> Grails is a modern Web development framework that mixes familiar Java™ technologies like Spring and Hibernate with contemporary

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

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.

A similar transformation happens when you surf the Web. Technically, the only way to contact a remote Web server is by providing the server's IP address. Luckily, you don't need to enter the IP address yourself. You type a friendly URL into your Web browser, and it makes a call to a Domain Name System (DNS) server. The DNS server converts the URL to the appropriate IP address, and the browser makes the HTTP connection to the remote server. All of this is transparent to the user. DNS makes the Web infinitely easier to use. Geocoders do the same for Web-based mapping applications.

A quick Web search on *free geocoder* yields a number of potential candidates for the trip planner's geocoding needs. Both Google and Yahoo! offer geocoding services as a standard part of their APIs, but for this application, I'll use the free geocoding service provided by geonames.org (see Resources). Its RESTful API allows me to indicate that I am supplying an IATA code instead of a generic text-search term. I have nothing against the residents of Ord, Neb., but the ORD I'm most interested in is the Chicago O'Hare International Airport.

Enter the URL `http://ws.geonames.org/search?name_equals=den&fcode=airp&style=full` into your Web browser. You should see the XML response shown in Listing 1:

Listing 1. XML results from geocoding request


```
<geonames style="FULL">
  <totalResultsCount>1</totalResultsCount>
  <geoname>
    <name>Denver International Airport</name>
    <lat>39.8583188</lat>
    <lng>-104.6674674</lng>
    <geonameId>5419401</geonameId>
    <countryCode>US</countryCode>
    <countryName>United States</countryName>
    <fcl>S</fcl>
    <fcode>AIRP</fcode>
    <fclName>spot, building, farm</fclName>
    <fcodeName>airport</fcodeName>
    <population/>
    <alternateNames>DEN,KDEN</alternateNames>
    <elevation>1655</elevation>
    <continentCode>NA</continentCode>
    <adminCode1>CO</adminCode1>
    <adminName1>Colorado</adminName1>
    <adminCode2>031</adminCode2>
    <adminName2>Denver County</adminName2>
    <alternateName lang="iata">DEN</alternateName>
    <alternateName lang="icao">KDEN</alternateName>
    <timezone dstOffset="-6.0" gmtOffset="-7.0">America/Denver</timezone>
  </geoname>
```


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

```
</geonames>
```

The `name_equals` parameter in the URL you entered is the IATA code for the airport. It's the only part of the URL that needs to be changed for each query. `fcode=airp` indicates that the feature code you are searching on is an airport. The `style` parameter — `short`, `medium`, `long`, or `full` — specifies the verbosity of the XML response.

Now that you have a geocoder in place, the next step is to integrate it with your Grails application. To do so, you need a *service*.

Grails services

By this point in the *Mastering Grails* series, you should have a fairly good idea how domain classes, controllers, and Groovy Server Pages (GSPs) all work together in a coordinated manner. They facilitate basic Create/Retrieve/Update/Delete (CRUD) operations on a single datatype. This geocoding service seems to go a bit beyond the scope of simple Grails Object Relational Mapping (GORM) transformations from relational database records to POGOs (plain old Groovy objects). Also, the service will most likely be used by more than one method. Both `save` and `update` will need to geocode the IATA code, as you'll see in just a moment. Grails gives you a place to store commonly used methods that transcend any single domain class: services.

To create a Grails service, type `grails create-service Geocoder` at the command line. Look at grails-app/services/GeocoderService.groovy, shown in Listing 2, in a text editor:

Listing 2. A stubbed-out Grails service

```
class GeocoderService {
    boolean transactional = true
    def serviceMethod() {

    }
}
```

The `transactional` field is of interest if you are making multiple database queries in the same method. It wraps everything in a single database transaction that rolls back if any individual query fails. Because in this example you are making a remote Web service call, you can safely set it to `false`.

The name `serviceMethod` is a placeholder that can be changed to something more descriptive. (Services can contain as many methods as you'd like.) In Listing 3, I've changed the name to `geocodeAirport`:

Listing 3. The geocodeAirport() geocoder service method

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

```
class GeocoderService {
    boolean transactional = false

    // http://ws.geonames.org/search?name_equals=den&fcode=airp&style=full
    def geocodeAirport(String iata) {
      def base = "http://ws.geonames.org/search?"
      def qs = []
      qs << "name_equals=" + URLEncoder.encode(iata)
      qs << "fcode=airp"
      qs << "style=full"
      def url = new URL(base + qs.join("&"))
      def connection = url.openConnection()

      def result = [:]
      if(connection.responseCode == 200){
        def xml = connection.content.text
        def geonames = new XmlSlurper().parseText(xml)
        result.name = geonames.geoname.name as String
        result.lat = geonames.geoname.lat as String
        result.lng = geonames.geoname.lng as String
        result.state = geonames.geoname.adminCode1 as String
        result.country = geonames.geoname.countryCode as String
      }
      else{
        log.error("GeocoderService.geocodeAirport FAILED")
        log.error(url)
        log.error(connection.responseCode)
        log.error(connection.responseMessage)
      }
      return result
    }
}
```

The first part of the `geocodeAirport` method builds up the URL and makes the connection. The query-string elements are gathered up in an `ArrayList` and then joined together with an ampersand. The last part of the method parses the XML result using a Groovy `XmlSlurper` and stores the results in a hashmap.

Groovy services aren't directly accessible from a URL. If you want to test this new service method in your Web browser, add a simple closure to the `AirportController`, as shown in Listing 4:

Listing 4. Providing an URL to a service in a controller

```
import grails.converters.*

class AirportController {
  def geocoderService
  def scaffold = Airport

  def geocode = {
    def result = geocoderService.geocodeAirport(params.iata)
    render result as JSON
  }

  ...
}
```


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

Spring injects the service into the controller automatically if you define a member variable with the same name as the service. (For this trick to work, you must change the first letter of the service name from uppercase to lowercase to follow Java-style variable-naming conventions.)

To test the service, enter the URL `http://localhost:9090/trip/airport/geocode?iata=den` in your Web browser. You should see the result shown in Listing 5:

Listing 5. Results of the geocoder request

```
{"name":"Denver International Airport",
"lat":"39.8583188",
"lng":"-104.6674674",
"state":"CO",
"country":"US"}
```

The `geocode` closure in `AirportController` is there just for you to sanity check the service. You can remove it at this point, or you can leave it in place for possible future Ajax calls. The next step is to refactor the `Airport` infrastructure to take advantage of this new geocoding service.

Mixing in the service

To start, add the new `lat` and `lng` fields to grails-app/domain/Airport.groovy, as shown in Listing 6:

Listing 6. Adding lat and lng fields to the Airport POGO

```
class Airport{
  static constraints = {
    name()
    iata(maxSize:3)
    city()
    state(maxSize:2)
    country()
  }

  String name
  String iata
  String city
  String state
  String country = "US"
  String lat
  String lng

  String toString(){
    "${iata} - ${name}"
  }
}
```

Type `grails generate-views Airport` at the command prompt to create the

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

GSP files. They've been dynamically scaffolded at run time up to this point, courtesy of the `def scaffold = Airport` line in AirportController.groovy. Now that I want to make some changes to the views, I need to get my hands on the code.

When creating a new `Airport`, I'm going to limit the user-editable fields to `iata` and `city`. The `iata` field is necessary for the geocoding query to work. I'm leaving `city` in place because I'd prefer to supply that information myself. DEN is truly in Denver, but ORD (Chicago O'Hare) is in Rosemont, Ill., and CVG (the Cincinnati, Ohio airport) is technically in Florence, Ky. Leave these two fields in create.gsp and delete the rest, so that create.gsp now looks like Listing 7:

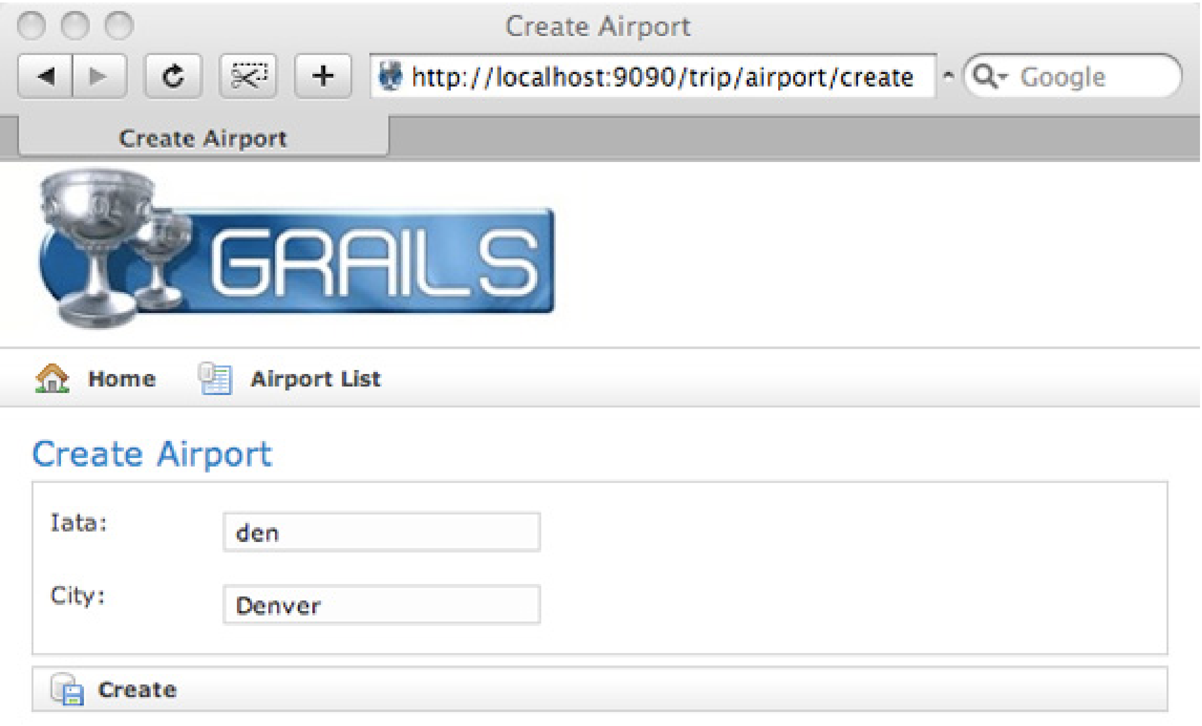
Listing 7. Modifying create.gsp

```
<g:form action="save" method="post" >
  <div class="dialog">
    <table>
      <tbody>
        <tr class="prop">
          <td valign="top" class="name"><label for="iata">Iata:</label></td>
          <td valign="top"
              class="value ${hasErrors(bean:airport,field:'iata','errors')}">
              <input type="text"
                     maxlength="3"
                     id="iata"
                     name="iata"
                     value="${fieldValue(bean:airport,field:'iata')}"/>
          </td>
        </tr>
        <tr class="prop">
          <td valign="top" class="name"><label for="city">City:</label></td>
          <td valign="top"
              class="value ${hasErrors(bean:airport,field:'city','errors')}">
              <input type="text"
                     id="city"
                     name="city"
                     value="${fieldValue(bean:airport,field:'city')}"/>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
  <div class="buttons">
    <span class="button"><input class="save" type="submit" value="Create" /></span>
  </div>
</g:form>
```

Figure 2 shows the resulting form:

Figure 2. Create Airport form

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



This form is submitted to the `save` closure in `AirportController`. Add the code in Listing 8 to the controller to make a call to `geocodeAirport` before the new `Airport` is saved:

Listing 8. Modifying the save closure

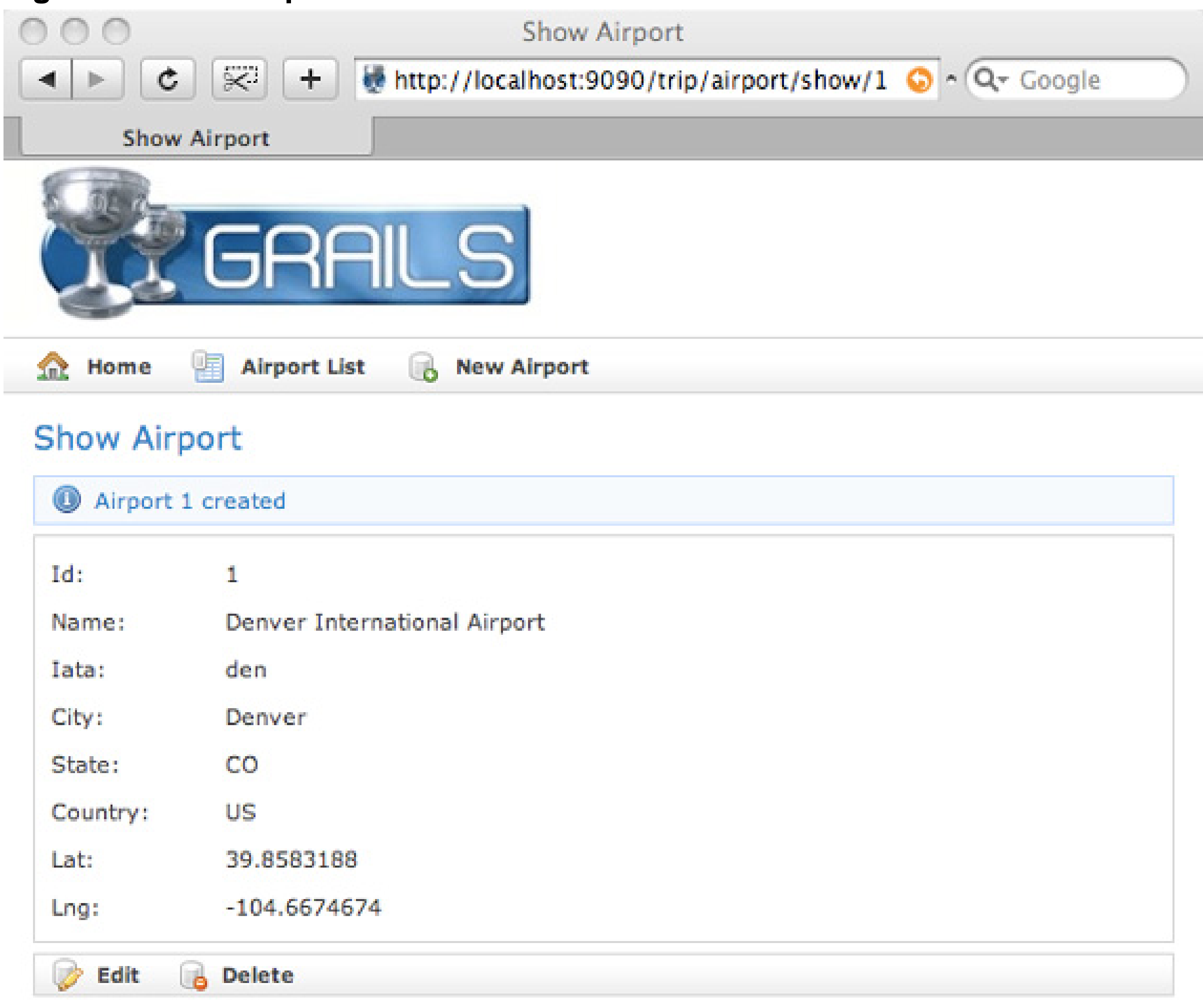
```
def save = {
    def results = geocoderService.geocodeAirport(params.iata)
    def airport = new Airport(params + results)
    if(!airport.hasErrors() && airport.save()) {
        flash.message = "Airport ${airport.id} created"
        redirect(action:show,id:airport.id)
    }
    else {
        render(view:'create',model:[airport:airport])
    }
}
```

The bulk of the method is identical to what you'd see if you typed `grails generate-controller Airport` at the command prompt. The only difference from the default-generated closure is the first two lines. The first line gets a `HashMap` from the geocoder service. The second line combines the `results HashMap` with the `params HashMap`. (Yes, merging two `HashMap`s in Groovy is as simple as adding them together.)

If the database save is successful, you are redirected to the show action. Thankfully, no changes are required to show.gsp, as shown in Figure 3:

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

Figure 3. Show Airport form



To support editing an `Airport`, leave the `iata` and `city` fields in place in edit.gsp. You can copy and paste the rest of the fields from show.gsp to make them read-only. (Or if you took my "copy and paste is the lowest form of object-oriented programming" rant to heart from an earlier article, you can extract the common fields into a partial template and render it in both show.gsp and edit.gsp.) Listing 9 shows edit.gsp, as modified:

Listing 9. Modifying edit.gsp

```
<g:form method="post" >
  <input type="hidden" name="id" value="${airport?.id}" />
  <div class="dialog">
    <table>
      <tbody>
        <tr class="prop">
          <td valign="top" class="name"><label for="iata">Iata:</label></td>
          <td valign="top"
```

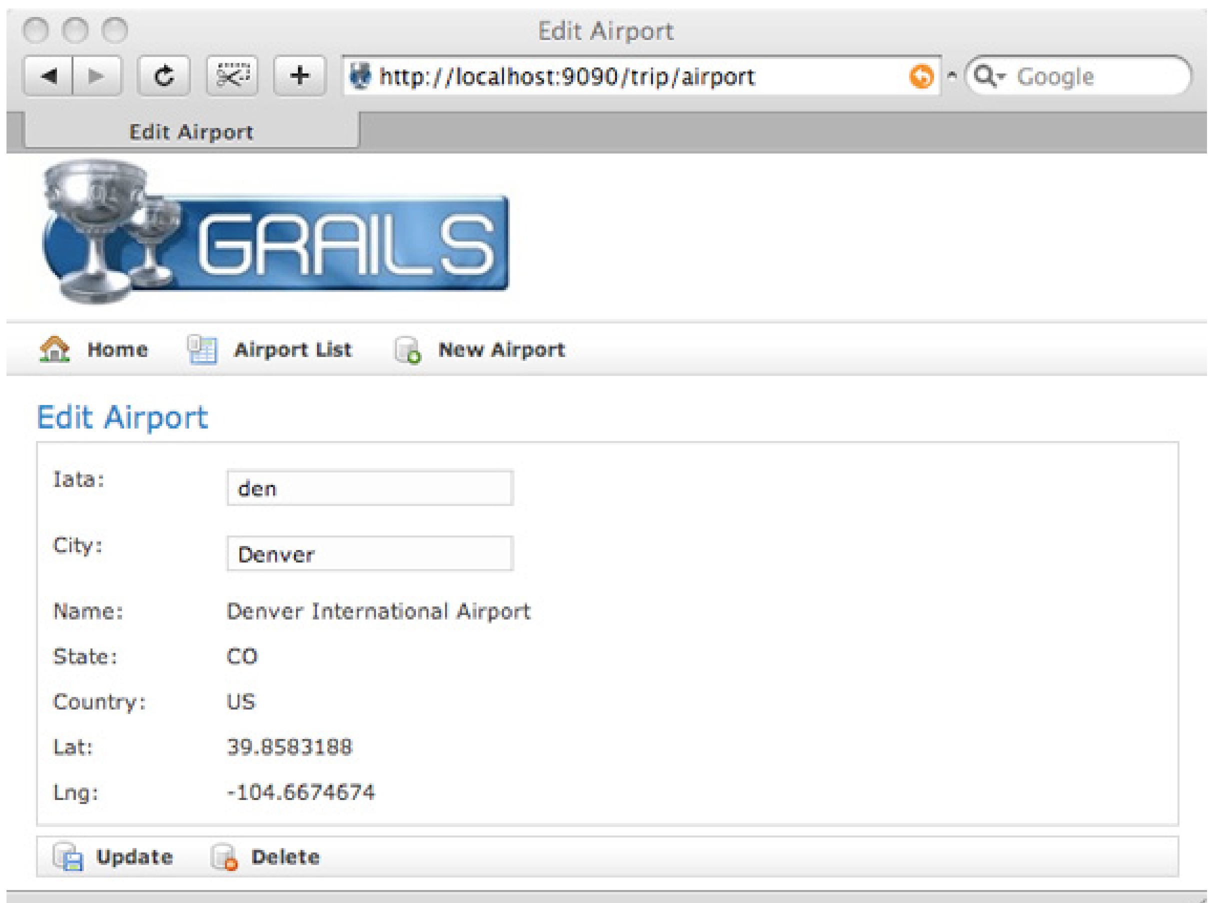

© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
                class="value ${hasErrors(bean:airport,field:'iata','errors')}">
                <input type="text"
                       maxlength="3"
                       id="iata"
                       name="iata"
                       value="${fieldValue(bean:airport,field:'iata')}"/>
            </td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name"><label for="city">City:</label></td>
            <td valign="top"
                class="value ${hasErrors(bean:airport,field:'city','errors')}">
                <input type="text"
                       id="city"
                       name="city"
                       value="${fieldValue(bean:airport,field:'city')}"/>
            </td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name">Name:</td>
            <td valign="top" class="value">${airport.name}</td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name">State:</td>
            <td valign="top" class="value">${airport.state}</td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name">Country:</td>
            <td valign="top" class="value">${airport.country}</td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name">Lat:</td>
            <td valign="top" class="value">${airport.lat}</td>
          </tr>
          <tr class="prop">
            <td valign="top" class="name">Lng:</td>
            <td valign="top" class="value">${airport.lng}</td>
          </tr>
        </tbody>
      </table>
    </div>
    <div class="buttons">
      <span class="button"><g:actionSubmit class="save" value="Update" /></span>
      <span class="button">
        <g:actionSubmit class="delete"
                        onclick="return confirm('Are you sure?');"
                        value="Delete" />
      </span>
    </div>
  </g:form>
```

The resulting form is shown in Figure 4:

Figure 4. Edit Airport form

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



Clicking the **Update** button sends the form values to the `update` closure. Add the service call and the hashmap merge to the default code, as shown in Listing 10:

Listing 10. Modifying the update closure

```
def update = {
    def airport = Airport.get( params.id )
    if(airport) {
        def results = geocoderService.geocodeAirport(params.iata)
        airport.properties = params + results
        if(!airport.hasErrors() && airport.save()) {
            flash.message = "Airport ${params.id} updated"
            redirect(action:show,id:airport.id)
        }
        else {
            render(view:'edit',model:[airport:airport])
        }
    }
    else {
        flash.message = "Airport not found with id ${params.id}"
        redirect(action:edit,id:params.id)
    }
}
```

At this point, you've seamlessly integrated geocoding into your application just as

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

Google Maps does. Take a moment to consider all of the places in your applications where you are capturing addresses — customers, employees, remote offices, warehouses, retail locations, and so on. By simply adding a couple of fields to store the latitude/longitude coordinates and mixing in a geocoding service, you are setting yourself up for some easy maps to display the objects — which is exactly what I am going to do next.

Google Maps

Most people agree that Google Maps set the standard for ease of use when it comes to Web mapping. Few realize that this ease of use extends to embedding a Google Map into your own Web page. Getting the latitude/longitude coordinates for the data points is the hardest part of the exercise, and we already have that problem solved.

To embed a Google Map in your Grails application, the first thing you need to do is get a free API key. The sign-up page details the terms of use. Essentially, Google provides the API to you for free as long as your application is free as well. This means that you cannot password protect your Google Maps application, charge for access to it, or host it behind a firewall. (Shameful plug: My book *GIS for Web Developers* gives you a step-by-step set of instructions for building a Google Maps-like application using free data and open source software; see Resources. This frees you from even the slight use restrictions Google places on its API.)

The API key is tied to a specific URL and directory. Type `http://localhost:9090/trip` in the form and click the **Generate API Key** button. The confirmation page shows your newly generated API key, the URL it's associated with, and a sample Web page to, as they say, "get you started on your way to mapping glory."

To incorporate this sample page into your Grails application, create a file named map.gsp in the grails-app/views/airport directory. Copy the sample page from Google into map.gsp. Listing 11 shows the contents of map.gsp:

Listing 11. A simple Google Maps Web page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>Google Maps JavaScript API Example</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=ABCDE"
            type="text/javascript"></script>
    <script type="text/javascript">
    //<![CDATA[
    function load() {
      if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
```


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

```
        }
      }
      //]]>
      </script>
    </head>
    <body onload="load()" onunload="GUnload()">
      <div id="map" style="width: 500px; height: 300px"></div>
    </body>
  </html>
```

Notice that your API key is embedded in the script URL at the top of the page. In the `load` method, you are instantiating a new `GMap2` object. This is the map that appears in the <div /> with the ID of `map` at the bottom of the page. If you'd like the map to be bigger, adjust the width and height in the Cascading Style Sheets (CSS) `style` attribute. Currently the map is centered over Palo Alto, Calif. at zoom level 13. (Level 0 is zoomed all the way out. As the numbers increase, you get closer to street-level views.) You'll adjust these values in just a moment. In the meantime, add an empty `map` closure to `AirlineController`, as shown in Listing 12:

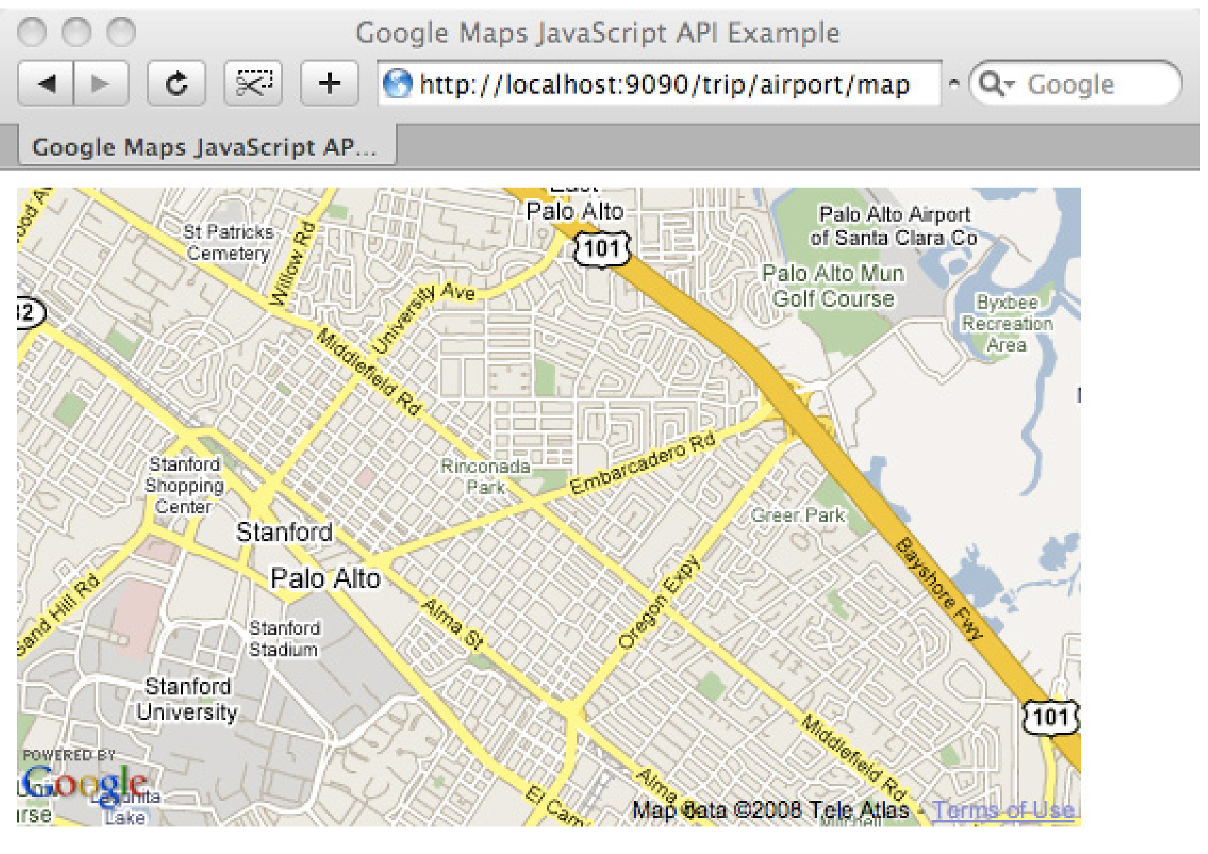
Listing 12. Adding the map closure

```
class AirportController {
  def map = {}

  ...
}
```

Now visit http://localhost:9090/trip/airport/map in your browser. You should see your embedded Google Map, which looks like Figure 5:

Figure 5. Simple Google Map

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



Now go back to map.gsp and tweak the values, as shown in Listing 13:

Listing 13. Adjusting the basic map

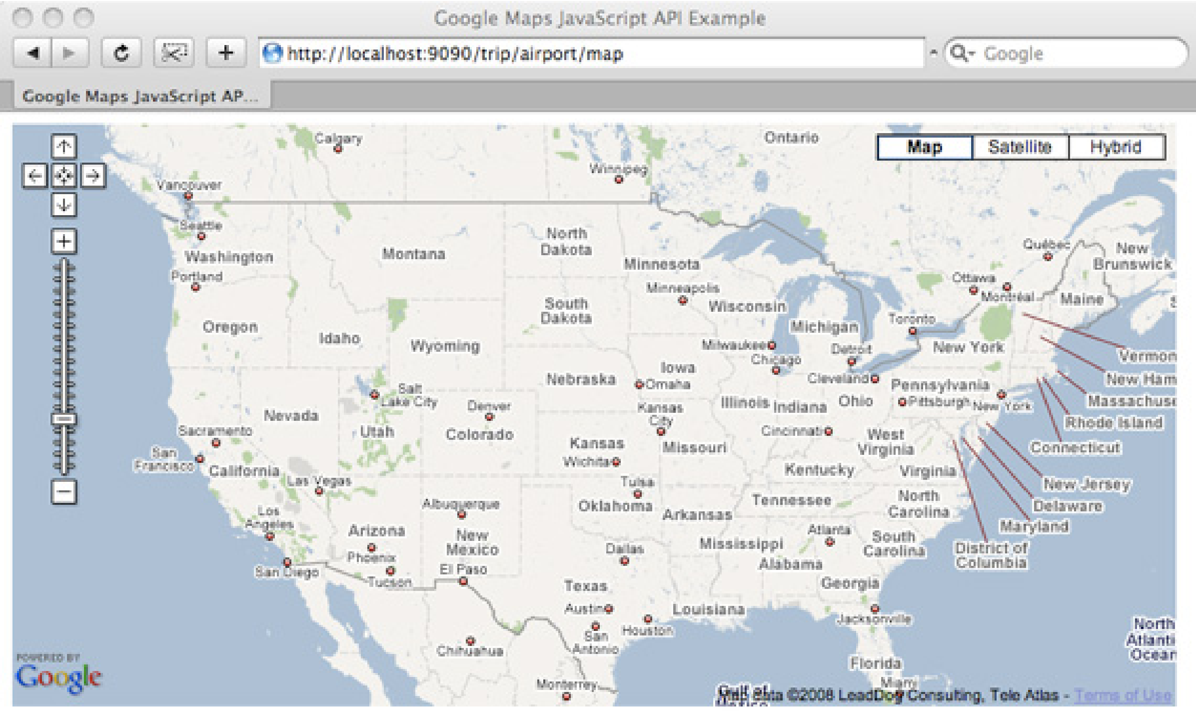
```
    <script type="text/javascript">
    var usCenterPoint = new GLatLng(39.833333, -98.583333)
    var usZoom = 4

    function load() {
      if (GBrowserIsCompatible()) {
        var 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 id="map" style="width: 800px; height: 400px"></div>
  </body>
```

800 x 400 pixels is a good set of dimensions for seeing the entire United States. Listing 13 adjusts the center point and the zoom level so that you can see the full map. You can add a number of different map controls. `GLargeMapControl` and `GMapTypeControl` in Listing 13 give you the usual controls along the left and

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

upper-right corner of the map. Periodically hit your browser's Refresh button to see your changes take effect as you play around. Figure 6 reflects the adjustments made in Listing 13:

Figure 6. Adjusted map



Now that your base map is in place, you are ready to begin adding markers — push-pins for each of your airports. Before I automate the process, Listing 14 manually adds a single marker:

Listing 14. Adding a marker to the map

```
<script type="text/javascript">
var usCenterPoint = new GLatLng(39.833333, -98.583333)
var usZoom = 4

function load() {
  if (GBrowserIsCompatible()) {
    var map = new GMap2(document.getElementById("map"))
    map.setCenter(usCenterPoint, usZoom)
    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());

    var marker = new GMarker(new GLatLng(39.8583188, -104.6674674))
    marker.bindInfoWindowHtml("DEN<br/>Denver International Airport")
      map.addOverlay(marker)
  }
}
</script>
```

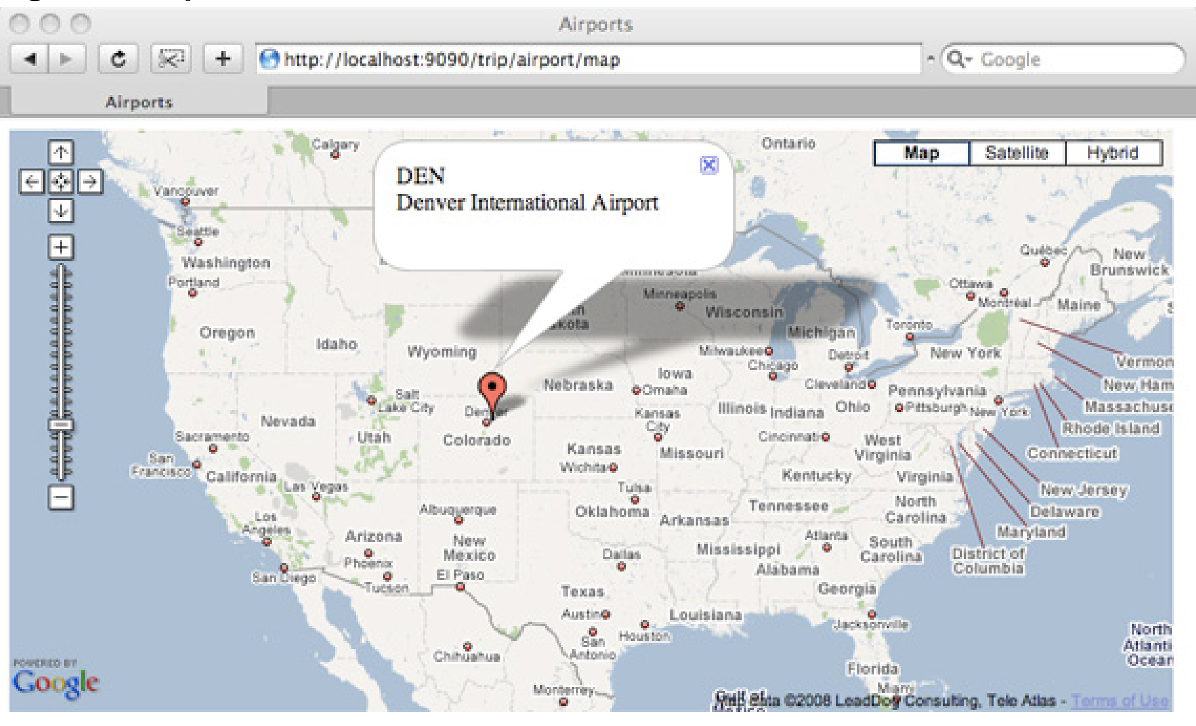
The `GMarker` constructor takes a `GLatLng` point. The `bindInfoWindowHtml`

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

method provides the HTML snippet to be displayed in the Info window when the user clicks the marker. The last thing Listing 14 does is add the marker to the map using the `addOverlay` method.

Figure 7 shows the map with the added marker:

Figure 7. Map with marker



Now that you see how to add a single point, automatically adding all points from the database requires two small changes. The first change to make the `map` closure in `AirportController` return a list of `Airport`s, as shown in Listing 15:

Listing 15. Returning a List of Airports

```
def map = {
  [airportList: Airport.list()]
}
```

Next, you need to iterate through the list of `Airport`s and create markers for each. Previously in this series, you've seen the `<g:each>` tag used to add rows to an HTML table. Listing 16 uses it to create the necessary lines of JavaScript to display the `Airport`s on the map:

Listing 16. Dynamically adding markers to the map

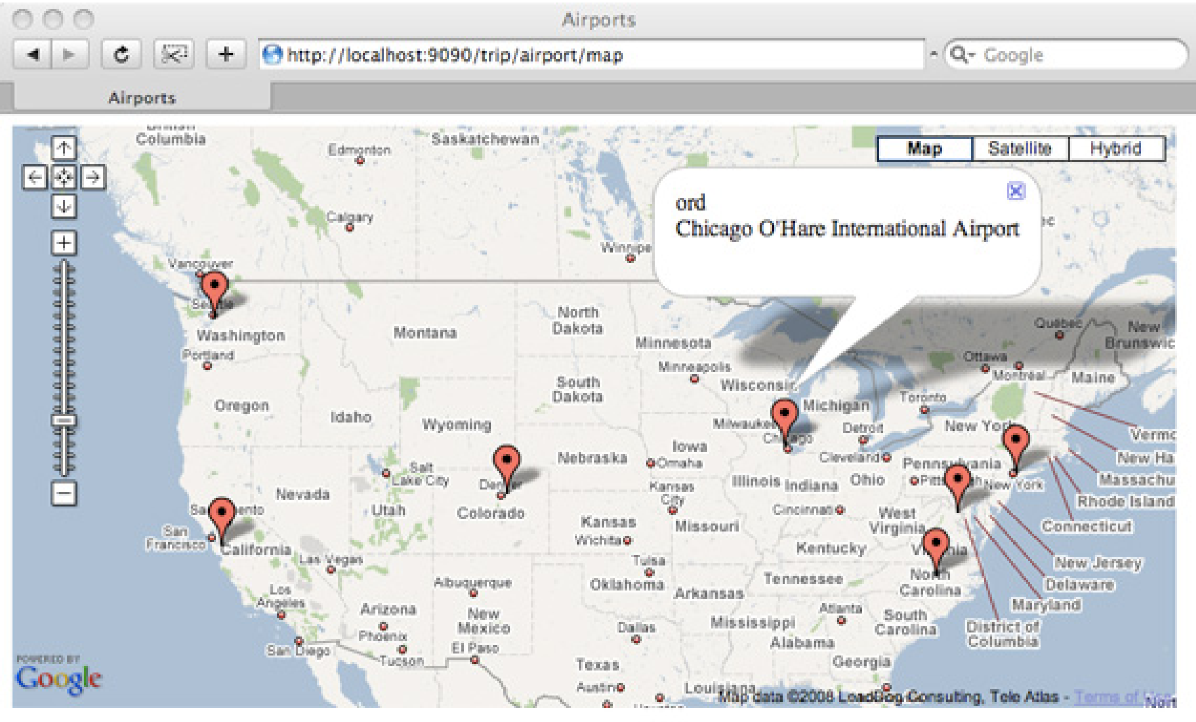
```
<script type="text/javascript">
```


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

```
var usCenterPoint = new GLatLng(39.833333, -98.583333)
var usZoom = 4

function load() {
  if (GBrowserIsCompatible()) {
    var map = new GMap2(document.getElementById("map"))
    map.setCenter(usCenterPoint, usZoom)
    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());

      <g:each in="${airportList}" status="i" var="airport">
         var point${airport.id} = new GLatLng(${airport.lat}, ${airport.lng})
      var marker${airport.id} = new GMarker(point${airport.id})
      marker${airport.id}.bindInfoWindowHtml("${airport.iata}<br/>${airport.name}")
         map.addOverlay(marker${airport.id})
      </g:each>
  }
}
</script>
```

Listing 8 shows the map with all the automatically added markers in place:

Figure 8. Map with multiple markers



This whirlwind tour of the Google Maps API just barely scratched the surface of things you can do. You might decide to tap into the event model to make Ajax calls that return JavaScript Object Notation (JSON) data when the markers are clicked. You can use `GPolyline`s to plot the individual legs of a trip on a map. The possibilities are endless. For more information, Google's online documentation is excellent. You can also get a gentle introduction to the API from my PDF book *Google Maps API* (see Resources).

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

Conclusion

Adding maps to your Grails application requires three elements to be in place.

The first piece of the puzzle is geocoding your data. There are plenty of free geocoders out there that convert human-readable locations into latitude/longitude points. Nearly anything can be geocoded: street addresses, cities, counties, states, countries, ZIP codes, phone numbers, IP addresses, and yes, even IATA codes for airports.

Once you find the right geocoder for the job, create a Grails service to encapsulate the remote Web services call in a reusable method call. Services are for methods that go beyond simple CRUD operations on a single domain object. Services aren't associated with an URL by default, but you can easily create a closure in a controller to make them Web addressable.

Finally, take advantage of a free Web mapping API such as Google Maps to plot your latitude/longitude points on a map. These free services generally require your application to be freely accessible as well. If you'd prefer to keep your map private, consider an open source API such as OpenLayers, which provides the same user experience as a Google Map without the corresponding use restriction (see Resources). You'll need to provide your own mapping layers, but you'll be able to host the entire application in the privacy of your own server.

In the next article, I'll talk about ways to make your Grails application mobile-phone friendly. You'll see how to optimize the views for an iPhone display. I'll also demonstrate sending information from Grails via e-mail that appears as an SMS message on your phone. 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: Takes you through getting started with Grails and building Web applications with the Grails framework.
- GeoNames: The GeoNames geocoding service is based on Representational State Transfer (REST) architecture.
- Google Maps API: Embed Google Maps in your Web pages.
- Yahoo! Maps API: Embed Yahoo! Maps into your Web and desktop applications.
- Geocoding: Wikipedia's entry on geocoding.
- *GIS for Web Developers* (Scott Davis, Pragmatic Programmers, 2007): Scott Davis' book introduces Geographic Information Systems (GIS) in simple terms and demonstrates hands-on uses.
- *The Google Maps API* (Scott Davis, Pragmatic Programmers, 2006): Learn how to draw maps, add annotations and routes, and geocode your data.
- OpenLayers API: OpenLayers is a pure JavaScript library for displaying map data in most Web browsers.
- *Groovy Recipes* (Scott Davis, Pragmatic Programmers, 2007): 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

- Grails: Download the latest Grails release.
- OpenLayers: Download OpenLayers.

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

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.

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