ORACLE
TECHNOLOGY NETWORK
DEVELOPER: MapViewer

As Published In
ORACLE
MAGAZINE
May/June 2005

Fact-Finding with MapViewer

By Liujian Qian and Jayant Sharma

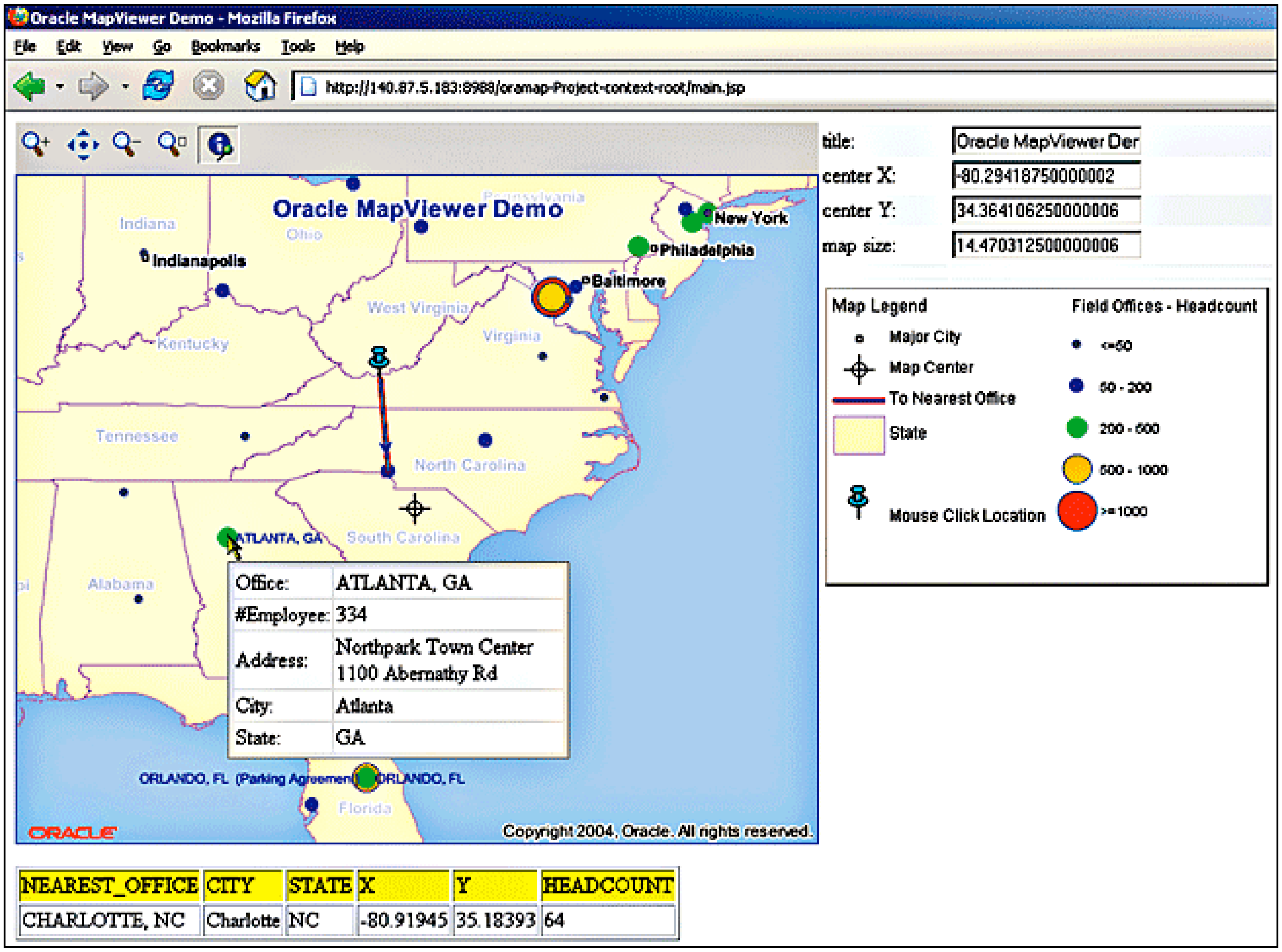
Map answers with Oracle Application Server 10*g*

For years people have been able to enter an address into a Web site and get a map of that location. Adding a map to an address gives that location additional context beyond the street name and address number.

Maps of business information can add value to names and numbers as well and are now finding their way into analysis, planning, and user interfaces for many kinds of business and reporting applications.

This article presents an overview of Oracle Application Server 10*g* MapViewer and a sample application—including a description of the datasets, database, and Oracle Application Server Containers for J2EE (OC4J) setup—that uses MapViewer to display active location-based information.

The sample application is a pure HTML Web application that displays active location-based information, including a company's field offices and its head count, on a map, as shown in Figure 1. Each field office is represented by a graduated circle whose color and size are determined by the office's head count. As the user moves the cursor over a field office location, an info-tip window displays selected attributes of the office. Additional possible operations include drilling down on a field office for more information and finding the nearest field office for any given location (such as a customer address) on the map.



title:	Oracle MapViewer Der
center X:	-80.29418750000002
center Y:	34.364106250000006
map size:	14.470312500000006

Office:	ATLANTA, GA
#Employee:	334
Address:	Northpark Town Center 1100 Abernathy Rd
City:	Atlanta
State:	GA

NEAREST_OFFICE	CITY	STATE	X	Y	HEADCOUNT
CHARLOTTE, NC	Charlotte	NC	-80.91945	35.18393	64

Figure 1: Client accessing the MapViewer sample application

Overview of MapViewer

MapViewer is a component of Oracle Application Server 10*g*. It is a lightweight yet powerful servlet for visualizing geospatial data managed by Oracle Spatial. This includes thematic mapping of business data by associated geographies.

A map is made up of a set of themes (or layers). Each theme is a query containing a spatial column (of type `SDO_GEOMETRY`) and other columns in its `SELECT` list and has a styling rule specifying its presentation. MapViewer receives a map request and renders a map image, using the styles and styling rules stored in the database.

You can deploy MapViewer on Oracle Application Server 10*g* or in a standalone OC4J instance.

MapViewer Application Architecture

A MapViewer application consists of

- Content managed by Oracle Spatial, such as the street network, administrative boundaries, and office locations
- Map metadata in database views containing a list of named maps, the set of themes that constitutes each map, and the styling rules and cartographic symbols used for rendering the themes
- A configured, deployed, and running MapViewer instance listening for requests
- A Web client that submits XML-based map requests, processes the XML response, and displays the resulting map image

The Web client may optionally use the MapViewer client Java library or JSP tag library to simplify the process of submitting map requests, handling the response, managing user interaction with the map image, and maintaining state—such as the current map center point and scale—between requests. This process is demonstrated in the following sample Web application.

Install MapViewer

The easiest way to get started with MapViewer is to download the Quick Start kit for Oracle Application Server 10*g* 10.1.2 MapViewer from "http://www.oracle.com/technology/products/mapviewer/index.html" on the Oracle Technology Network (OTN). It contains a preconfigured standalone OC4J with MapViewer.

Load the Data

The following are the MapViewer themes for the sample application:

- Base themes, including state boundaries and major cities
- The field offices theme

The definitions of such themes are usually stored in each database user's dictionary view (`USER_SDO_THEMES`). You can also create dynamic themes based on queries constructed at runtime and have the results of such queries displayed on a map. This process is illustrated in the jview.jsp demonstration that is part of the MapViewer Quick Start kit.

The base themes are all in the sample demonstration dataset (mvdemo.dmp), available at the MapViewer site on OTN. This dataset includes spatial information for U.S. states, counties, highways, and cities.

The data for this article's sample application is in the MapViewer.zip archive. MapViewer.zip contains the field_offices.dmp file—an export of the `FIELD_OFFICES` table from the user `MVDEMO`. MapViewer.zip also contains the oramap.zip file, which includes the application's source code, scripts, libraries, and an Oracle JDeveloper 10*g* workspace. Finally, MapViewer.zip includes a readme file.

After you import the mvdemo.dmp and field_offices .dmp files, run the style_theme.sql script to create the required styles and themes.

How to Run the Application

Open the workspace file, oramap.jws, in Oracle JDeveloper (9.0.5 or later), and browse the source code. With a MapViewer service running (with mapviewer.ear deployed in a standalone OC4J and loaded on startup) locally, run the JSP file main.jsp under the project named Project. You need to add MapViewer's Java client library, $OC4J_HOME/j2ee/home/applications/mapviewer/web/WEB-INF/lib/mvclient.jar, to your project's library path.

Alternatively, to run the application without Oracle JDeveloper, copy everything in the oramap/Project/public_html directory to a new directory in your deployed MapViewer location (such as $OC4J_HOME/j2ee/home/applications/mapviewer/web/oramap) and, with the MapViewer service running, point your browser to http://localhost:8888/mapviewer/oramap/main.jsp.

Code Organization

All the application code is in files contained in the oramap/Project/public_html directory. The main files are

- main.jsp
- toolbar.js
- toolbar.html
- infotip.js
- style_theme.sql
- myicons (directory)
- legend.xml

Because this sample application's purpose is to illustrate the client API usage, both presentation logic and core application code reside in main.jsp. When developing larger-scale Web applications, you should use a mature application framework, such as Oracle Application Development Framework (ADF).

The toolbar.js and toolbar.html files implement the toolbar, with its rollover buttons, using standard JavaScript code.

The infotip.js file contains JavaScript code for the info-tip window that displays detailed attributes of a field office as you mouse over it. This code complements MapViewer's built-in support for generating an HTML image map for any map image. An HTML image map is a set of "clickable" areas defined on an image that is viewed in a browser. A map image created by MapViewer does not have any HTML image map associated with it.

The myicons directory contains all the icons used by the toolbar and map legend, and the legend.xml file contains the map request for the map legend.

What's in main.jsp

The main.jsp file contains the most important code in the sample application. The primary code sections of the main.jsp file are (in order)

- **Environment setup.** Imports necessary Java classes and packages, such as MapViewer Client API, and includes JavaScript code files implementing the toolbar and info-tip features on the client side.

- **Preparing map parameters.** Initializes or obtains essential parameters from the current Web session. These include the map center, map size, map title, user-specified map action (pan, zoom in, zoom out, or identify a feature), and where the user has clicked on the previous map.

- **Rendering a new map.** Constructs and sends a map request, based on the user action and map parameters.

- **Presenting the result page.** Presents the resulting map to a client browser.

This code flow generally applies to any MapViewer-based Web application. The application sends a map request to MapViewer and presents the result map. An end user performs certain actions on the map, and the application sends a new request to the MapViewer server and presents a new result map.

The complete main.jsp file text is included at "http://www.oracle.com/technology/oramag/oracle/05-may/main_jsp.zip".

Environment Setup

The environment setup section of main.jsp imports the required classes and JavaScript. The following imports a class, contained in `mvclient.jar`, that represents your client handle when you are working with MapViewer:

```
<%@ page import=
"oracle.lbs.mapclient.MapViewer" %>
```

It sends map requests and processes responses from a MapViewer server.

The following statements import the JavaScript that implements the client-side toolbar and info-tip mechanisms:

```
<script type="text/javascript"
  src="toolbar.js">    </script>
<script type="text/javascript"
  src="infotip.js">    </script>
```

The following is the customizable function `customMapClicked()`:

```
function customMapClicked(
                         action, x, y, w, h)
{
  document.omv_mapform.map_action
.value = action;
  document.omv_mapform.map_click_x
.value = x;
  document.omv_mapform.map_click_y
.value = y;
  document.omv_mapform.map_box_w
.value = w;
  document.omv_mapform.map_box_h
.value = h;
  document.omv_mapform.submit();
  return false;
}
```

It ensures that a new request is submitted, with the proper parameters, when a user clicks anywhere on the map (except on HTML image map areas). Mouse-click event handlers defined in toolbar.js call this function.

Clicking on an HTML image map area on the map, such as on a field office marker, calls the following function on the client side:

```
function areaClicked(event, info)
{
  alert(info);
}
```

The info parameter contains the essential attributes of the field office. The FIELD OFFICE theme definition specifies these parameters. Customizing the `areaClicked` function allows for more-sophisticated processing, such as opening a new report or creating charts about the identified map feature.

Next in the environment setup section of main.jsp, you declare two DIV objects named tbar_rect and infotip_window in the HTML header. You use tbar_rect to support zooming to an area within a box and infotip_window to display the info-tip window.

Preparing Parameters

The preparing parameters section of main.jsp declares variables and obtains parameter values used in constructing a map request. The key input parameter, action, holds the current user-selected map action. Its value (`pan, zoomin, zoomout, zoombox`, or `id`) indicates which action MapViewer should perform.

The variable `mvurl` specifies the location of the MapViewer server. Assume that MapViewer is deployed to a local standalone OC4J instance, and hence use the URL http://localhost:8888/mapviewer/omserver.

Other parameters in the preparing parameters section of main.jsp include the center point of the map and the size (in decimal degrees from the top to the bottom of the map). If the user has previously clicked on the map, the location of the mouse click (in the device coordinate system relative to the image) is also obtained from the input HTTP request. With this information, you request a new MapViewer map.

Rendering a New Map

The code in the "rendering a new map" section of main.jsp sets up a map request and sends it to the MapViewer server. Most of the map-related logic, as well as your custom geospatial queries, are performed here with the client Java API.

To render a new map, you first obtain two references to MapViewer client instances from the current user session. Normally only one such client is required to send map requests and receive responses. The sample application, however, uses the new HTML image map support, so you need a separate MapViewer client to send a slightly different map request.

Creating new MapViewer clients. For a new browser session or if you cannot obtain an existing MapViewer client instance—because, for example, the session expires—you create new client instances, using the code in Listing 1.

Code Listing 1: Creating client instances

```
if (mv == null || newSession)             // new session
    {
      mv = new MapViewer(mvURL);      // one for the main map request
      session.setAttribute("oramap", mv);  // keep client handle in the session

      mv.setDataSourceName(dataSrc);     // specify the data source (database)
      mv.setImageFormat(MapViewer.FORMAT_PNG_URL); // PNG Image
      mv.setMapTitle(title);                      // set map title
      // specify marker symbol denoting map center
      mv.setDefaultStyleForCenter("M.IMAGE89_BW", null, null, null);
      mv.setAntiAliasing(true);                     //make map look nicer
      mv.setCenterAndSize(cx, cy, size);      // initial center & size
      mv.setDeviceSize(new Dimension(width, height));  // window size
     // Specify themes to display. States, Cities, and field office locations
      mv.addPredefinedTheme("THEME_DEMO_STATES");
      mv.addPredefinedTheme("THEME_DEMO_BIGCITIES");
      mv.addPredefinedTheme("FIELD OFFICE");

      // now create MapViewer instance for handling HTML image maps
      clkmv = new MapViewer(mvURL);  // for "FIELD OFFICE CLK" theme.
      clkmv.setDataSourceName(dataSrc); // same data source
      clkmv.setCenterAndSize(cx, cy, size);    // and center and size
      // but different image format. We use SVG to construct the image map
     clkmv.setSVGFragmentType(MapViewer.SVG_LAYERS_WITH_LABELS) ;
      clkmv.setSVGFragmentInDeviceCoord(true);

      clkmv.setDeviceSize(new Dimension(width, height));
      // specify the theme. FIELD OFFICE CLK lists the attributes
      // that show up in an info-tip
      clkmv.addPredefinedTheme("FIELD OFFICE CLK");
      session.setAttribute("oramap_clk", clkmv);

      // submit the two map requests
      mv.run();
      clkmv.run();
    }
```

The two MapViewer client instances (also called*handles* or *beans*) are `mv` and `lkmv`. The first, `mv`, serves as the main client for constructing and sending regular map requests, whereas `clkmv` sends HTML image map-related requests.

Themes and styles: How field offices are displayed. The client `mv` map request includes three predefined themes. The key theme is FIELD OFFICE, which is defined in `USER_SDO_THEMES:`

```
SQL> select base_table,
geometry_column, styling_rules
from user_sdo_themes
where name='FIELD OFFICE';
```

The result of the query is

```
FIELD_OFFICES
LOCATION
<?xml version="1.0" standalone="yes"?>
<styling_rules>
  <rule column="HEADCOUNT"
    order_by="HEADCOUNT"
    sort_order="DESC">
    <features style=
    "OFFICE_STYLE">  </features>
    <label column="NAME"
     style="T.STREET NAME">
     headcount - 250
    </label>
  </rule>
</styling_rules>
```

So `FIELD_OFFICES` is the base table for this theme, and the `LOCATION` column contains the location of each office.

The `STYLING_RULES` definition specifies that the `HEADCOUNT, NAME`, and `LOCATION` columns are queried from `FIELD_OFFICES`. It also stipulates that the result is sorted in descending order of `HEADCOUNT` value. This is important, because you represent field offices by using circles of varying sizes determined by the `HEADCOUNT` value. If two office locations are very close to each other, you want MapViewer to render the larger circle first and then the smaller one on top.

A feature is labeled only when the numeric value supplied in the `LABEL` column is greater than 0. The <label> element above has a condition "headcount - 250," so MapViewer labels an office only if its head count is greater than 250.

The `clkmv` client's sole purpose is to send a map request containing the FIELD OFFICE CLK theme. It is very similar to the FIELD OFFICE theme used by the `mv` client, except that this one sorts the offices in ascending order of head count, so that the larger HTML image map areas generated for the field offices come after smaller ones in the area list. The FIELD OFFICE CLK theme also has a <hidden_info> element:

```
<styling_rules >
  <hidden_info>
      <field column="NAME"
              name="Office" />
      <field column="HEADCOUNT"
              name="#Employee" />
      <field column="ADDRESS"
              name="Address" />
      <field column="CITY"
              name="City" />
      <field column="STATE"
              name="State" />
  </hidden_info>
  ...
</rule>
```

MapViewer retrieves the columns queried in the <hidden_info> element as part of the query and includes them in the map response.

The `mv` client asks the server to render the map into a PNG file, save it on the server host, and return a URL for the image file that is presented in the browser. The `clkmv` client, however, uses a Scalable Vector Graphics (SVG) file format as follows:

```
clkmv.setSVGFragmentType(
MapViewer.SVG_STYLED_LAYERS_WITH_LABELS) ;
clkmv.setSVGFragmentInDeviceCoord(true);
```

These two methods tell MapViewer to create an SVG map for the request and return an SVG document containing map data in the device coordinate system. The coordinates are used for generating the HTML image map areas.

Having created the two clients, you call the `run()` method, which sends the request to the server. The clients wait for the response and extract relevant information, such as the generated map image URL, using accessor methods.

Now that you know how the map content and format are specified, it's time to consider how the field offices' locations are rendered. The two field office themes' styling rules reference a style named OFFICE_STYLE, which is a bucket-based—or advanced—style. OFFICE_STYLE contains a set of buckets, each corresponding to a value range, such as 250 < headcount < 500, with a primitive style, such as a red circle. For the FIELD OFFICE themes, each office's head count value determines its bucket and style.

Processing user actions. The action parameter in main.jsp contains the user interaction information. If the value of the action is `pan, zoomin, zoomout`, or `zoombox`, you reissue the map request, by calling the corresponding methods, such as `pan()`, with the new center and size parameters. You always call the methods on both clients, so the HTML image map generated by `clkmv` is in sync with the displayed map image.

Processing the ID function. The ID action is initiated when an end user clicks on the ID button on the toolbar and then clicks anywhere on the map. The actual processing takes place on the server side.

Listing 2 shows the relevant code segment for processing an ID action.

Code Listing 2: Processing an ID action

```
else if("id".equals(action))
      {
        String[] columns = new String[]{"NAME Nearest_Office",
                                        "City", "State",
                                        "a.location.sdo_point.x X",
                                        "a.location.sdo_point.y Y",
                                        "Headcount"};

        //find out the office nearest to where user clicked on the map
        officeInfo = mv.identify(dataSrc,
                     "field_offices a",  //the table name
                     columns,            // columns in SELECT clause
                     "location",         // geometry column name
                     srid,               // spatial reference system id
                     mapClickX, mapClickY //mouse click position
                     );
        Point2D officeLoc = null;
        if(officeInfo!=null)
        {
          // identify() returns a String[][] of row, column values
          // row 0 is the column name list, row 1 on are values
          // columns are named in the "columns" parameter
          // so here column 0 is Name, 1 is City, 2 is State,
          // 3 is X, and 4 is Y
          String x = officeInfo[1][3];
          String y = officeInfo[1][4];
          officeLoc = new Point2D.Double(Double.parseDouble(x),
                                         Double.parseDouble(y));
        }
        // mark user click on the map with a PIN marker
        Point2D p2 = mv.getUserPoint(mapClickX, mapClickY);
        mv.addPointFeature(p2.getX(), p2.getY(), srid,
                           "M.CYAN PIN",  //a PIN marker style
                           null, null, null);

        //add a leader line from user click to nearest office
        if(officeLoc!=null)
          mv.addLinearFeature(new double[]{p2.getX(), p2.getY(),
                              officeLoc.getX(), officeLoc.getY()},
                              srid, "NEAREST_LINE_STY", null,
                              null, false);

        /* For identify: use previously generated map as backdrop
           and avoid rerendering all base themes.
         */
        if(mv.getBackgroundImageURL()==null)
          mv.setBackgroundImageURL(mv.getGeneratedMapImageURL());
        String[] enabledThemes = mv.getEnabledThemes();
        mv.setAllThemesEnabled(false); //temporarily disable themes

        mv.run();  // reissue map request to draw a PIN marker
        // reenable all themes
        mv.enableThemes(enabledThemes);

        mv.removeAllPointFeatures();  // clean up PIN marker
        mv.removeAllLinearFeatures(); // clean up leader line as well
      } // end id action
```

The code in Listing 2 first finds the field office nearest to the user-specified location, using the `identify()` method. The input parameters are the name of the target table (`FIELD_OFFICES`) to search against, the list of columns to return, and the mouse click location in the screen coordinate system.

Next, the code determines the ground coordinates corresponding to the mouse click location:

```
Point2D p2 = mv.getUserPoint(
                mapClickX, mapClickY);
```

Then the code adds a PIN marker symbol at that location:

```
mv.addPointFeature(
  p2.getX(), p2.getY(), srid,
  "M.CYAN PIN",  //a PIN marker style
  null, null, null);
```

Next, the code adds a line connecting these two locations to the new map:

```
mv.addLinearFeature(
  new double[]{p2.getX(), p2.getY(),
  officeLoc.getX(), officeLoc.getY()},
  srid, "NEAREST_LINE_STY", null,
  null, false);
```

Because you are adding only the PIN and line features to the new map, it's wasteful to regenerate the entire thing. So you simply set the previous map as the backdrop, temporarily disable all other themes, and issue the map request as follows:

```
if(mv.getBackgroundImageURL()==null)
  mv.setBackgroundImageURL(
    mv.getGeneratedMapImageURL());
String[] enabledThemes =
  mv.getEnabledThemes();
mv.setAllThemesEnabled(false);
  //temporarily disable themes
mv.run();
  // reissue map request to
  // draw PIN and line
```

Once you have generated the new map, you reenable the themes and clean up the PIN and line features:

```
mv.enableThemes(enabledThemes);
mv.removeAllPointFeatures();
// clean up the PIN marker
mv.removeAllLinearFeatures();
// clean up leader line as well
```

At this point, you have a new map, have fetched the attributes for the nearest office, and are ready to present the information.

Next Steps

READ	More about MapViewer
DOWNLOAD	MapViewer
	This article's sample application
	main.jsp

Presenting the Result Page

The presenting-the-result-page section of main.jsp contains simple HTML tags. The toolbar is shown in Listing 3.

Code Listing 3: Presenting the result page toolbar

```
<!-- Left column : toolbar and map image -->
    <TD width="<%=width%>" bgcolor="#d4d0c8">
       <%
            // pick up the correct icon file for the toolbar buttons
            String[] toolbarNames = new String[]{"zoomin", "pan",
                                     "zoomout", "zoombox",  "id"};
            String[] toolbarImgs = new String[toolbarNames.length];
            for(int i=0; i<toolbarImgs.length; i++)
            {
              if(toolbarNames[i].equals(action))
                toolbarImgs[i] = "myicons/" + toolbarNames[i]
                                 + "_dn.png";
              else
                toolbarImgs[i] = "myicons/"+toolbarNames[i]+".png";
            }
       %>
        <%@  include file="toolbar.html" %>
    </TD>
```

The statement `<%@ include file="toolbar.html" %>` lays out the toolbar buttons. A valid button icon file must be selected for each button and reflect its current status (that is, whether it is clicked on). The JSP code in Listing 3 loops through the list of button names and chooses the proper icon image file from the myicons directory. If a button, such as "pan," is clicked on, you use the <button_name>_dn.png file (in this case, pan_dn.png). These filenames are picked up by the toolbar.html file.

Presenting the Generated Map Image

The code in Listing 4 presents the generated map image.

Code Listing 4: Presenting the map

```
<%
        // Get the HTML AREA definition of selected theme's Image Map.
        String areas = clkmv.getThemeAsHTMLAreas(
                              "FIELD OFFICE CLK", true) ;
      %>
      <MAP NAME="omv_infomap">
        <%= areas==null?"":areas %>
      </MAP>
      <div id="infotip_window"></div>
      <div id="display" style="position:relative">
        <!— now for the actual map image -->
        <img id="oramap"  src="<%=mv.getGeneratedMapImageURL()%>"
             border="1"
             usemap="#omv_infomap"
             onload="changeActionButton('<%=action%>')" />
      </div>
      <div id="tbar_rect"></div>
```

This code is essentially an HTML <img> tag, with the image src obtained from `mv` through the `getGeneratedMapImageURL()` method. It uses an HTML image map named omv_infomap, whose contents are returned from the method call `getThemeAsHTMLAreas()` on `clkmv` and placed between the <MAP> tags before the <img> tag. Finally the <img> tag is surrounded by various <DIV> elements, including infotip_window and tbar_rect, for info-tip window and box-based zoom, respectively.

When MapViewer presents a map image, a user can click on map navigation buttons ("pan" or "zoom") or on the map to identify a displayed feature. Because certain state information, or parameter values, must be maintained between requests, an HTML FORM contains the current parameter values. When a user clicks on the map, a JavaScript function uses these parameters to submit a new map request.

Conclusion

MapViewer gives Web application developers a versatile means of integrating and visualizing business data with maps. It uses the capability included with Oracle 10*g* to manage geographic mapping data and hides the complexity of spatial data queries and the cartographic rendering process from application developers.

Liujian Qian (lj.qian@oracle.com) is the lead developer for MapViewer. **Jayant Sharma** (Jayant.Sharma@oracle.com) is technical director for Oracle Spatial.