

Oracle Application Server 10g (10.1.2)

MapView

*An Oracle Technical White Paper
January 2005*

Table Of Contents

1	EXECUTIVE SUMMARY	1
2	INTRODUCTION	1
3	TECHNICAL OVERVIEW	2
4	NEW AND IMPROVED FEATURES SINCE MAPVIEWER 9.0.2	3
5	ACCESSING MAPVIEWER FUNCTIONS USING THE JAVA API	5
5.1	MAPPING METADATA: STYLES, THEMES, AND BASE MAPS	6
5.2	STYLES	6
5.3	THEMES	7
5.3.1	<i>Styling Rules</i>	9
5.3.2	<i>How MapViewer Formulates Queries for Predefined Themes</i>	10
5.3.3	<i>Dynamic Themes in a Map Request</i>	10
5.3.4	<i>Thematic Mapping through Advanced Styles</i>	11
5.4	BASE MAPS	13
6	MAP GENERATION PROCESS	13
7	MAPVIEWER XML AND JAVA APPLICATION PROGRAMMING INTERFACES	14
8	MAP DEFINITION TOOL	15

1 Executive Summary

A picture, as they say, is worth a thousand words. This is particularly true when trying to capture the complexity of interactions among people, resources, products, and business processes distributed over geographic space. For many centuries people have relied on maps to capture and simplify these complex relationships, turning them into readily consumable, powerful packages of unambiguous information. Beginning with Oracle10g and Oracle Application Server 10g, the basic Oracle platform delivers this powerful, universally understood capability to every developer.

Oracle Application Server MapViewer (or simply, MapViewer) provides powerful geospatial data visualization and reporting services. Written purely in Java and run in a J2EE environment, MapViewer provides web application developers a versatile means to integrate and visualize business data with maps. MapViewer uses the basic capability included with Oracle10g (delivered via either Oracle Spatial or Locator) to manage geographic mapping data. MapViewer hides the complexity of spatial data queries and the cartographic rendering process from application developers.

The services provided by MapViewer are accessed through a flexible and powerful XML-based API over HTTP protocol. Using this API, an application developer can direct MapViewer to fetch spatial data and generate maps from any Oracle database instance. Users and developers can also customize the appearance of the map via the XML API. They can control visual map characteristics—such as the background color, the title, the symbology used to portray features such as roads, store locations and property boundaries, and so on—using extensible metadata stored in database tables. It is also possible to incorporate dynamically obtained geospatial data, such as customer locations, and plot these on top of a base map. Thematic mapping—portraying the distribution of attributes such as population density, psycho-demographic information measuring income, education, etc.—is also supported through the MapViewer API, if the user has the needed baseline data. Beginning with Oracle Application Server 10g (9.0.4), MapViewer provides a Java-based API and a set of JSP (Java Server Page) tags that hide much of the complexity in using the XML API.

MapViewer maintains a clear separation between the presentation of data and the data itself. Users control a map's appearance through mapping metadata that defines base maps, map themes, map symbols, styling rules, and other portrayal information. *The ability to manage all the portrayal data in a central repository and share such information among many users is a key benefit of MapViewer.*

2 Introduction

Geographic data has traditionally been managed in proprietary-formatted files and displayed using special GIS applications. Oracle Database 10g provides an open and standard-based geographic data management solution via either Oracle Spatial or Locator. Users can load all types of geometric data into a database, create spatial indexes, and issue spatial queries through SQL. Because of this, Oracle is becoming an industry standard for managing geospatial data.

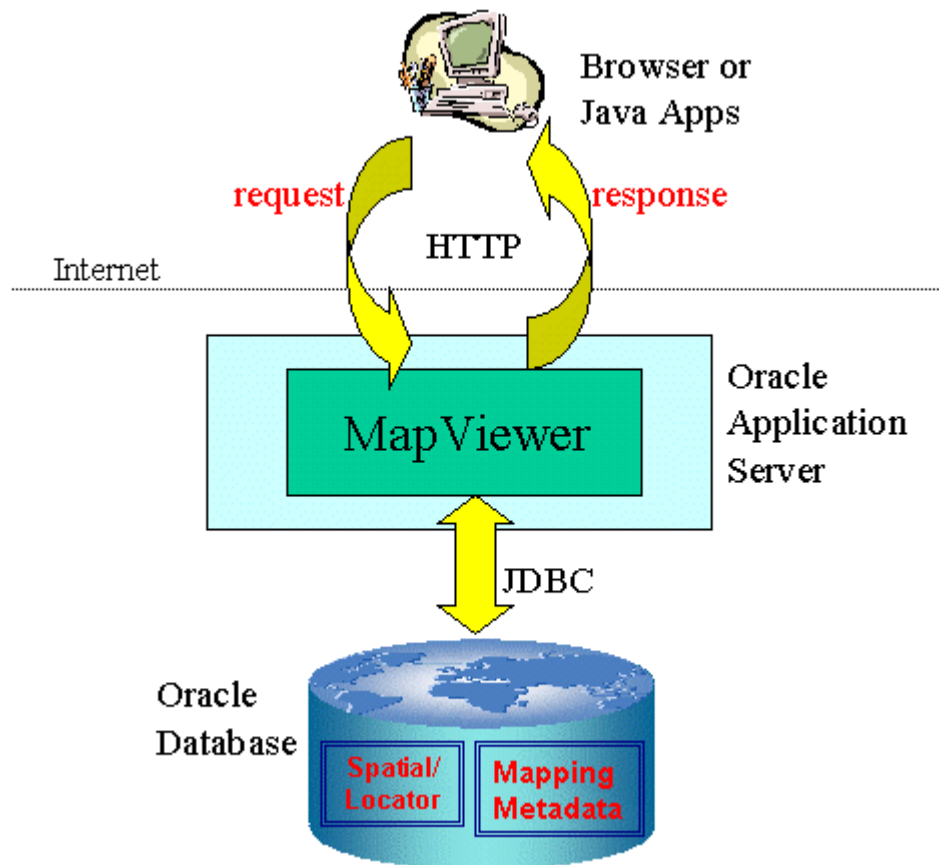
MapViewer complements the geographic data management capacity of the Oracle Database by providing a generic web-based means of delivering and viewing any geographic data in the database. This creates enormous potential for understanding and capturing the geographic component(s) of any business, by unlocking the enterprise information in many corporate warehouses and making it available to basic mapping applications. For instance, business applications such as Field Service, Transportation and Logistics, Product Lifecycle Management, Human Resources, and Real Estate can now render and visualize the massive amount of data they control *if* there is a geographic component such as an address tied to the data. Developers of location-based services, data publishers in state and local government, and architects of web services and more traditional applications can all easily integrate MapViewer into their web-based solutions.

3 Technical Overview

The MapViewer component in Oracle Application Server (available since release 9.0.2) is written in Java and runs inside the Oracle Container for J2EE (OC4J). MapViewer ships with Oracle Application Server. Or you can simply download both MapViewer and a standalone OC4J from the Oracle Technology Network (<http://otn.oracle.com/>). Once these have been obtained MapViewer must be *deployed* into OC4J. When it is up and running, MapViewer *listens* for client requests, which can range from map requests to administrative requests such as defining a data source or listing predefined maps in a data source. **All requests** will be sent using the **HTTP POST** method, **with the content** of the requests encoded **in XML** format (specifics of the MapViewer XML API are described later in this paper). If your application uses the Java API or JSP tags, then these will convert your request into XML document and send it using HTTP POST.

When a map request is received, MapViewer parses it and retrieves relevant spatial data as well as mapping metadata (symbolology, map look and feel) from the database. A map, which can be visualized in a standard browser, is then rendered and optionally saved to the local file system in a specified format. In most cases MapViewer then sends an XML-encoded reply indicating success back to the client. Figure 1 illustrates the high-level architectural overview and the generic data flow in this process.

Figure 1. MapViewer Architecture



When issuing a map request to a running instance of MapViewer, the client needs to specify a data source. A data source tells MapViewer which database schema to use to retrieve map data and mapping metadata. Data sources can be defined dynamically through administrative requests to MapViewer. For each data source, MapViewer will establish one or more JDBC connections to the specified database user, and also instantiate a specified number of *mappers* to handle map requests for that data source. The infrastructure to manage this load is provided by the connection pool feature of the Oracle Application Server.

Mapping metadata controls the *appearance* of the generated maps. This metadata includes map symbols, text fonts, area and line patterns, styling rules that associate spatial tables with map layers or themes, and base map definitions. Mapping metadata is stored inside the database schema (Figure 1 above), and individual users can either define personalized metadata for their private use or common metadata can be shared across a group of users. For example, an organization can define a set of commonly used map symbols to be shared by many departments' users. Each department can then define its own map layers and base maps using the shared map symbols.

4 New and Improved Features Since MapViewer 9.0.4

New and improved features since MapViewer v9.0.4 include:

- Support for 10g GeoRaster (Georeferenced Images)
Users define GeoRaster (image) themes that can be rendered along with the normal vector themes.
- Support for 10g Topology and Network data models
MapViewer can render spatial entities stored in the 10g Topology or Network data models. Limited network analysis functions can be performed and their results portrayed on a map.
- JPEG and Transparent PNG Format Support
MapViewer now supports indexed PNG maps with a transparent background and JPEG maps.
- Support for Open Geospatial Consortium's Web Map Server 1.1.1 specification
MapViewer supports the rendering of data delivered using the Open GIS Consortium (OGC) Web Map Service (WMS) protocol.
- Workspace Manager Support
Workspace Manager is an Oracle Database feature that lets you version-enable one or more tables in the database. You can request a map from a specific workspace, at a specific savepoint in a workspace, or at a point close to a specific date in a workspace.
- Dynamic Coordinate System Transformation in a Map Request
Users can specify a SRID (spatial reference ID, or coordinate reference system ID) in a map request, and MapViewer will transform the theme data if it is not already in the specified SRID.
- SVG map support
MapViewer now supports the output of maps in SVG Basic, SVG Compressed, and SVG Tiny formats. When generating an SVG map, you can specify attributes for a theme that will be returned with the resulting SVG map, which can then be displayed in a pop-up window that follows your cursor as you move around in the SVG map. You can also customize and control the layers in a generated SVG map through such tools as JavaScript.
- Container Data Source as the Map Data Source
MapViewer now lets you use a data source defined in the OC4J container as the map data source.
- Style Enhancements
Several enhancements have been made to existing styles. The line style supports arrows, and a line style can be used as the boundary of an area style. Text and marker style displays can be further controlled using the new support for orientation vectors.

- Temporary Styles per map request
You can now create dynamically defined styles (that is, temporary styles) for use with a map request.
- Bounding Themes Option for Restricting Displayed Data
You can use the new <bounding_themes> element in a map request to restrict the range of the user data to be plotted on a map.
- High Availability and MapViewer
MapViewer enables you to use the high availability features (e.g. load balancing, and clusters) of Oracle Application Server more effectively.

5 Accessing MapViewer Functions Using the Java API

Web application developers will benefit from MapViewer's powerful services. This section illustrates a client's interaction with MapViewer. In this case the client can be either a Java program or a Java Server Page component of a web page. The discussion here assumes that:

- MapViewer release 9.0.4 or higher is deployed within OC4J or Oracle Application Server.
- The MapViewer instance is up and running.
- A data source named *mvdemo* has been defined for MapViewer.¹

Once a MapViewer instance is running in Oracle Application Server or OC4J, it can be accessed through a service URL. The service URL <http://foo.com/mapviewer/omserver> is used in the following example.

Further suppose that in the data source *mvdemo* there is a base map named *demo_map* defined as part of its mapping metadata. To obtain a map centered at the San Francisco area (longitude/latitude: -122.40, 37.80), 400 by 360 pixels in width and height, and covering an area of 2.5 decimal degrees (from top to bottom), use the following Java code segment:

```
import oracle.lbs.mapclient.*;

String serverUrl = "http://foo.com/mapviewer/omserver";
MapViewer mapClient = new MapViewer(serverUrl);
mapClient.setDataSourceName("mvdemo");
mapClient.setBaseMapName("demo_map");
mapClient.setCenterAndSize(-122.40, 37.80, 2.5);
mapClient.setDeviceSize(new java.awt.Dimension(400,300));
mapClient.setImageFormat(MapViewer.FORMAT_PNG_URL);
boolean response = mapClient.run();
if (response)
```

¹ For information about how to install and deploy MapViewer and add a data source, see the *MapViewer User's Guide*.

```
System.out.println("URL of generated map:" +
    mapViewer.getGeneratedMapImageUrl());
else
    System.err.println("Error from MapViewer service.");
```

Note that the map image itself will be generated and saved as a PNG file in the host where the actual MapViewer service is running.

For details about the Java API, see the *MapViewer User's Guide*. That document also contains sample programs that show how to formulate and send map requests in XML directly from a Java or PL/SQL program.

The following sections describe the basic MapViewer concepts, the XML API, and the rendering process in greater detail.

5.1 Mapping Metadata: Styles, Themes, and Base Maps

In MapViewer, a map conceptually consists of one or more themes. Each theme consists of a set of individual geographic features that share certain common attributes. Each feature is rendered and (optionally) labeled with specific styles. Themes can be predefined inside a database user's schema, or can be dynamically defined as part of a map request. Predefined themes can be grouped to form a predefined base map that can also be stored in a user's schema. Styles, predefined themes, and base maps are collectively called mapping metadata for MapViewer. **This scheme provides a clear separation between the presentation of data and the spatial data itself.** For example, any mistake made while manipulating the mapping metadata will have no effect on the corresponding spatial data, and vice versa.

5.2 Styles

A **style** is a visual attribute that can be used to represent a spatial feature. The basic map symbols and legends for representing point, line, and area features are defined and stored as individual styles. Each style has a unique name and defines one or more graphical elements in an XML syntax.

Each style is of one of the following types:

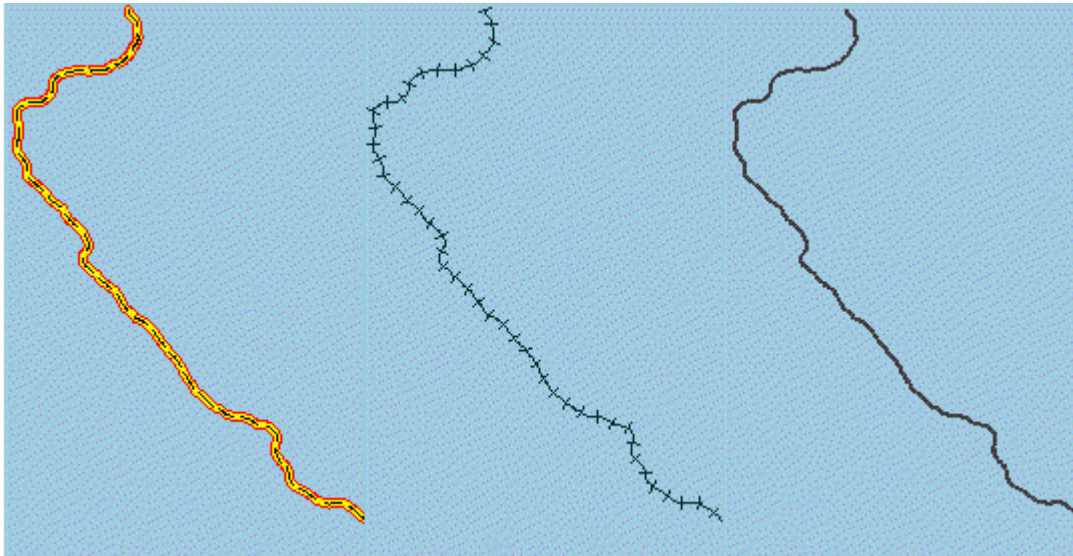
- **COLOR:** a color for the fill or the stroke (border), or both.
- **MARKER:** a shape with a specified fill and stroke color, or an image. Markers are often icons for representing point features, such as airports, ski resorts, and historical attractions. When a marker style is specified for a line feature, the rendering engine selects a suitable point on the line and applies the marker style (for example, a shield marker for a U.S. interstate highway) to that point.
- **LINE:** a line style (width, color, end style, join style) and optionally a center line, edges, and hashmark. Lines are often used for linear features such as highways, rivers, pipelines, and electrical transmission lines.
- **AREA:** a color or texture, and optionally a stroke color. Areas are often used for polygonal features such as counties and census tracts.

- **TEXT:** a font specification (size and family) and optionally highlighting (bold, italic) and a foreground color. Text is often used for feature annotation and labeling (such as names of cities and rivers).
- **ADVANCED:** a composite used primarily for thematic mapping. The core advanced style is `BucketStyle`, which defines a mapping from a set of simple styles to a set of buckets. For each feature to be plotted, a designated attribute value from that feature is used to determine which bucket it falls into, and then the style associated with that bucket is used to plot the feature. The `AdvancedStyle` class is extended by `BucketStyle`, which is in turn extended by `ColorSchemeStyle` and `VariableMarkerStyle`. (Additional advanced styles, such as for charts, are planned for a future release.)

All styles are stored in a table of the system user `MDSYS`, but are exposed to each user through its own `USER_SDO_STYLES` view.

Any geographic feature, such as a road, can be displayed differently if alternate styles are assigned or applied, even though the underlying geometric structure of the feature itself is identical. Figure 2 is an example of a single road being rendered using three different line styles.

Figure 2 Same Geometry, Different Line Styles



Note that the XML representation for each type of style is proprietary to Oracle, but these style definitions are specified in the *MapViewer User's Guide*. In the future, as a standard for such portrayal information emerges and matures, MapViewer will also move to support it.

5.3 Themes

A **theme** is a visual representation of a particular data layer. Conceptually, each theme is associated with a specific spatial geometry layer, that is, with a column of type

MDSYS.SDO_GEOMETRY in a table or view. For example, a theme named *US_States* might be associated with the GEOM column with type MDSYS.SDO_GEOMETRY in a STATES table.

There are several types of themes supported in MapViewer depending upon how the themes are created. The static ones are called **predefined themes**, whose definitions are stored in a database user's USER_SDO_THEMES view. Another type is **dynamic themes** (sometimes also called **JDBC themes**), which are simpler and defined on-the-fly within each map request.

With MapViewer release 10.1.2, basic support for geo-referenced images, stored in Oracle Spatial 10g GeoRaster, can also be rendered through **GeoRaster themes**. A GeoRaster theme definition can be either stored in the database as a predefined theme, or can be created dynamically by an application. Figure 3 is an illustration of normal vector themes overlaying an image theme. For details about how to use the GeoRaster themes, see the *MapViewer User's Guide*.

Figure 3. GeoRaster Theme Overlaid by Vector Themes



The actual definition of a predefined theme consists of the following: name of a base table or view, name of the geometry column, and one or more **styling rules** that associate styles to rows from the base table. Obviously, the styling rules are the most interesting part of a theme's definition, and they are discussed in the next section.

5.3.1 Styling Rules

A predefined theme can have one or more styling rules. Each styling rule tells MapViewer two things:

1. Which rows from the base table belong to a theme, and what **feature style** should be used to render the geometries from those rows. What this really means is that you can select only the desired rows (features) from the base table of a theme.
2. Optionally, whether the geometries in the selected rows should be annotated (labeled). If the answer is yes, then the rule must specify a column whose values will be used as the annotation text, as well as a **label style** for drawing the text. The placement of the text as relative to the geometry is, however, automatically determined by MapViewer at run time.

Each styling rule is encoded in XML following the conventions below. In this example the rule is part of a theme named *theme_us_airports*, whose base table contains airport data, and has columns such as GEOM, NAME, and RUNWAY_NUMBER.

```
<rule>
  <features style="c.black gray">
    runway_number &gt; 1
  </features>
  <label column="name" style="t.airport name">
    1
  </label>
</rule>
```

In this rule, as in each of the styling rules, there are two parts: **<features>** and **<label>**. The **<features>** element informs MapViewer which rows should be selected and the style to use when rendering the geometries from those rows. To specify the rows, you can supply any valid WHERE clause (minus the keyword WHERE) as the value of the **<features>** element. In this case, all rows that satisfy the SQL condition “runway_number > 1” will be selected, and the color style *c.black gray* will be used to depict the airport geometry for those rows.

Note that due to the restrictions of XML, the character ‘>’ is represented as ‘>’ according to XML syntax. It will be converted back to ‘>’ by MapViewer at run time in order to formulate a proper SQL query.

The second part of the preceding styling rule is the **<label>** element (optional), which can take two values: 0 or 1. When the value of this element is 1, MapViewer will attempt to label the airports when generating a map. The label text will come from the column

NAME, and the text style will be *t.airport name*. Note that if labeling is not needed, you can omit the <label> element or specify a value of 0 for the element.

Also note that when referencing a feature or label style, the name of the style can take a form of <user>:<style_name>, such as *MDSYS:t.airport name*. This directs MapViewer to apply the named style from the specified user's schema, in this case MDSYS. Thus, you can define all of an organization's basic map symbols under a common user schema (for example, MyOrgSchema), and have all other users share it without storing the same set of map symbols many times.

5.3.2 How MapViewer Formulates Queries for Predefined Themes

For each styling rule in a predefined theme, MapViewer will formulate an SQL query to retrieve the spatial data. If a theme has more than one rule, the SQL query for each of the rules will be combined using the SQL construct UNION ALL. In the following example, where *theme_us_airports* appears in a map request that specifies a map center (-122.40, 37.80) and size (5), the query formulated by MapViewer will approximate:

```
SELECT GEOM, 'c.black gray', NAME, 't.airport name', 1
FROM AIRPORTS
WHERE MDSYS.SDO_FILTER(GEOM,
    MDSYS.SDO_GEOMETRY(2003, NULL, NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1, 1003, 3),
    MDSYS.SDO_ORDINATE_ARRAY(-125.1777, 35.3, -119.6222, 40.3)),
    'querytype=WINDOW') = 'TRUE'
AND (runway_number > 1);
```

The SELECT statement is structured and is position dependent. The *first item* in the SELECT list *is always the geometry column*. The second and fourth items in the SELECT list correspond to the name of the feature and label styles as referenced in the theme's styling rule. The last SELECT item is a literal value '1', which tells MapViewer that all the rows in the result set needs to be labeled. The third SELECT item is the column that contains the actual label text, also specified in the styling rule. The SDO_FILTER operator (generated automatically by MapViewer) and the feature condition (supplied in the above styling rule sample) are combined together in the WHERE clause.

The SELECT list will have the same order and data types for each styling rule, and as such can be combined using UNION ALL when multiple styling rules are present for a theme. Multiple styling rules are required when, for example, different sets of rows are selected based on different conditions for a theme, with each set of rows having its own rendering and labeling styles. For example, if a table stores geometry for interstate highways, state roads, city streets, and suburban housing development streets, a request to MapViewer might want each of these road types to be represented differently on the map. In this case there would be four sets of styling rules referenced.

5.3.3 Dynamic Themes in a Map Request

For dynamic themes defined on a per-map request basis, there can only be one styling rule. This rule is implicitly specified by giving the **entire** SQL query and the required

feature and label styles in the theme definition in a slightly different way from the predefined themes discussed above. The following example specifies a dynamic theme as part of a map request.

```
<map_request>
...
  <theme name="sales_by_region">
    <jdbc_query spatial_column="region"
              datasource="mvdemo"
              label_column="manager"
              render_style="V.SALES COLOR"
              label_style="T.SMALL TEXT"
              > select region, manager from foo_sales_2001
    </jdbc_query>
  </theme>
...
</map_request>
```

In this case, a dynamic theme named *sales_by_region* is defined. The query that selects rows/features for this theme is `SELECT REGION, MANAGER FROM FOO_SALES_2001`. The feature and label style names are specified as *render_style* and *label_style* attributes, respectively. Note that the database connection information is explicitly listed as part of the theme definition.

Also, although the actual data for the theme may be from a different database (as indicated by the database connection information), the referenced styles will always be retrieved from the data source as indicated in the overall map request.

5.3.4 Thematic Mapping through Advanced Styles

Simple thematic mapping can be achieved through the use of advanced type styles in a theme. Assume that you want to render a theme of counties in such a way so that a county with higher population density will be rendered with a darker color. To do this, define an advanced style that has a series (buckets) of colors, and for each color assign a range of (population density) values. For example, population less than 50,000 might be yellow, population 50,000 to 2500,000 might be orange, population 250,000 to 1,000,000 might be light red, and so on.

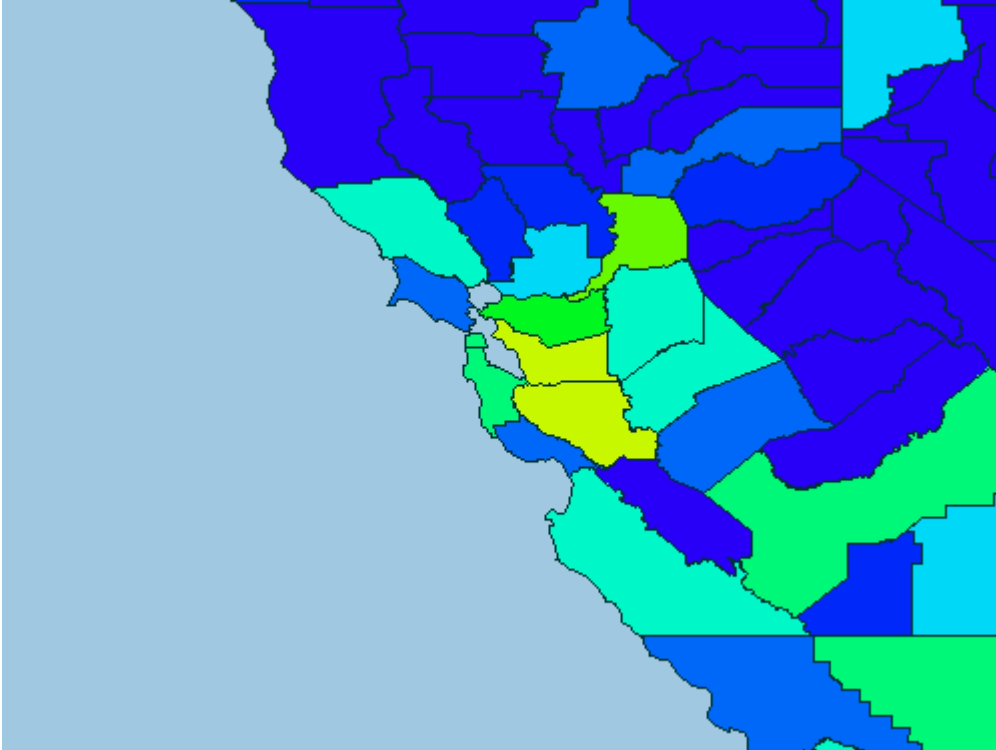
Assume that a style is named `V.POP DENSITY` will be used to represent relative population values. Once value ranges for the style have been set, define a theme that uses `V.POP DENSITY` as the feature style, just as with standard themes. However, unlike the procedure with standard themes, for this advanced theme you must also specify the column from the base table that contains the actual population density. This ensures that MapViewer will be able to map the series of colors in the `V.POP DENSITY` style to the rows (counties) selected for this theme. This is specified through a *column* attribute of the `<rule>` element in the following styling rule:

```
<rule column="pop_density">
  <features style="V.POP DENSITY">
  </features>
```

```
</rule>
```

Once this theme has been defined, it can be used like any other predefined theme. Figure 4 shows a rendered image in which each county area is scaled by color to reflect the population density value assigned to that county.

Figure 4. A Simple Thematic Map



Advanced styles can also be used with dynamic themes, as long as you specify the attribute column or columns needed by the advanced style in the SELECT list of the theme's query. For example:

```
<theme_name="sales_by_region_2">
<jdbc_query spatial_column="region"
  datasource="mvdemo"
  label_column="manager"
  render_style="V.SALES COLORS"
  label_style="T.SMALL TEXT"
>
  SELECT region, manager, sales_2003 FROM foo_sales_2003
</jdbc_query>
```

In the preceding example, the attribute column SALES_2003 is needed by the advanced style, and it is included in the SELECT list of the theme's query.

5.4 Base Maps

Predefined themes can be grouped together to form a base map. This provides a convenient way to include multiple themes in a map request. Note that only predefined themes can be included in a base map, not dynamic themes whose definitions are not retained after each processed map request. **The base map definitions are stored in a user's USER_SDO_MAPS view.**

A minimum and maximum map scale can be provided for each theme listed in a base map. This provides a powerful mechanism that is used to selectively *reveal* themes based on the current map's scale. For example, the local street network for a city like New York would be impossible to display effectively when rendering the state of New York. However, when viewing the borough of Manhattan, an application might well want the local streets portrayed. This feature in MapViewer enables this type of selective inclusion of information based on the nature of the application.

This mechanism can also be used to create generalized or simplified themes. For example, at a smaller map scale you may want to display only a simplified version of all the major roads. To achieve this you can create a table of roads whose geometry column contains simplified version of the original table, and then create a new theme that is associated with the new table. Then, you can add both the original theme and the new theme as part of a base map, but through the use of minimum and maximum scale values for each theme, only the appropriate theme will be picked and displayed at any map scale.

6 Map Generation Process

This section details the specific process of generating a map. In order for MapViewer to generate a map in response to a map request, the following conditions must be met:

- The data source indicated in the map request must have been defined or known to the MapViewer instance. When you define a data source, MapViewer establishes one or more permanent JDBC connections to the database user specified in it. The number of connections actually created is determined by the number of mappers specified in the data source definition.
All of the spatial data and mapping metadata required for a map is retrieved from the database user corresponding to the data source referenced in the map request. The only exception is dynamic themes, whose data can be retrieved from a different database schema or instance.
- If the map request references the name of a base map, the named base map must have been defined in the mapping metadata view USER_SDO_MAPS. Each database user will have this view defined to store all of that user's predefined base maps. A base map essentially defines which predefined themes should be rendered and in what order.
- For all the predefined themes of a base map, plus those predefined themes that are explicitly referenced in a map request, there must be a corresponding theme definition in the user's USER_SDO_THEMES view.

- For all the styles referenced from all the themes (predefined or dynamic), there must be a corresponding style definition in the user's USER_SDO_STYLES view. Or, if the style name is referenced in the form of <user>:<name>, the named style must exist in the specified user's USER_SDO_STYLES view.

If the conditions above are satisfied, MapViewer renders all the themes that are *implicitly or explicitly* specified in a map request. The steps in this workflow are:

1. MapViewer creates and fills a blank image based on the size and background color specified in the map request.
2. All the themes that are part of a base map (if present in the map request) are rendered to the blank image. The themes are rendered in the order they are listed in the base map definition. In particular, all the image themes will always be rendered prior to rendering any vector (regular) themes.
3. Before rendering a theme, MapViewer formulates a SQL query statement based on the styling rules of the theme. It then executes the query and fetches data from database for that theme. This process is known as “preparing a theme” to be rendered. The internal geometry cache will be used to reduce the number of repetitive fetches of the geometry data. Also, as part of preparing a theme, all the styles referenced in the theme are verified, and if they not already in an internal style cache they are retrieved from the data source.
4. Any explicitly specified themes (predefined or dynamic) in the map request are prepared and rendered on top of the same image, according to the order they are listed in the request.
5. If there are any individual GeoFeatures listed in the map request, they are plotted on top of the image.
6. For themes that have features needing to be labeled, the annotation process takes place. MapViewer automatically detects all the label collisions and determines the optimal position for each label text (if there is space to place it). The themes are labeled in the same order as they were rendered.
7. If map titles, footnotes, legend, map logo and other features were requested, they are plotted.

Once a map is rendered, MapViewer checks the map request to see what image format is requested by the client. It converts the internal raw image to the desired format, and either saves it to the local file system or sends it back directly to the client in binary form.

7 MapViewer XML and Java Application Programming Interfaces

MapViewer exposes its services through an XML API, which application developers can use when formulating XML map requests for MapViewer and parsing XML responses from it. Through the XML API, an application can:

- Customize a map's data coverage, background color, size, image format and title, and other characteristics.
- Display a map with predefined base map, plus any other predefined themes not included in the base map.

- Display a map with dynamically defined themes, with each theme's data retrieved from a user-supplied SQL query.
- Display a map with one or more individual features that the application may have obtained from other sources.
- Through the XML response, obtain the URL to the generated map image, or the actual binary image data itself, plus the minimum bounding rectangle of the data covered in the generated map.

Since MapViewer release 9.0.4, the XML API is wrapped into a Java API so that Java application developers can easily use the new API instead of manipulating XML map requests directly.

For information about the XML and Java APIs, see the *MapViewer User's Guide*.

8 Map Definition Tool

To assist with the creation and management of mapping metadata Oracle provides an unsupported utility (Map Definition Tool) on the Oracle Technology Network.

9 Summary

MapViewer provides web application developers a versatile means to integrate and visualize business data with maps. It uses the basic capability included with Oracle10g (either Oracle Spatial or Locator) to manage geographic mapping data. It hides the complexity of spatial data queries and the cartographic rendering process from application developers. They can easily integrate MapViewer into their applications. This creates enormous potential for understanding and capturing the geographic component(s) of any business, by unlocking the enterprise information in many corporate warehouses and making it available to basic mapping applications.



Copyright © 2005. Oracle Corporation
All Rights Reserved

Oracle Application Server 10g (9.0.4) MapViewer
An Oracle Technical White Paper
Authors: L.J. Qian and Jayant Sharma
Contributing Author: Chuck Murray

Oracle Corporation World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.
Phone 650.506.7000
Fax 650.506.7200

International Inquiries:
Phone 44.932.872.020
Telex 851.927444(ORACLEG)
Fax 44.932.874.625

<http://www.oracle.com>