ORACLE
FUSION MIDDLEWARE

An Oracle White Paper
July 2016

Oracle Fusion Middleware Mapviewer 12c Technical Overview

ORACLE

Executive Overview

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 Oracle10*g* and Oracle Application Server 10*g*, the basic Oracle platform delivers this powerful, universally understood capability to every developer. Oracle Fusion Middleware MapViewer (or simply, MapViewer) provides powerful geospatial data visualization and reporting services. Written purely in Java and run in a Java EE environment, MapViewer provides web application developers a versatile means to integrate and visualize business data with maps. MapViewer uses the basic capability included with the Oracle database (delivered via either Oracle Spatial and Graph 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, or the AJAX based JavaScript API included since version 10g. 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 these 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.

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*.

Introduction

Geographic data has traditionally been managed in proprietary formatted files and displayed using special GIS applications. The Oracle Database provides an open and standard-based geographic data management solution via either Oracle Spatial and Graph 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, Asset 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.

Technical Overview

The MapViewer component in Oracle Fusion Middleware is written in Java and runs inside a Java EE container. MapViewer is licensed with Oracle Weblogic Server, Oracle Application Server, or the ADF/Toplink Runtime bundle. It is available for download from Oracle's Software Delivery Cloud or from the Oracle Technology Network (http://www.oracle.com/technology/products/mapviewer). Once download MapViewer must be deployed into Weblogic or a supported Java EE server such as Glassfish. 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 (symbology, 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 XMLencoded reply indicating success back to the client. Figure 1 illustrates the high-level architectural overview and the generic data flow in this process.

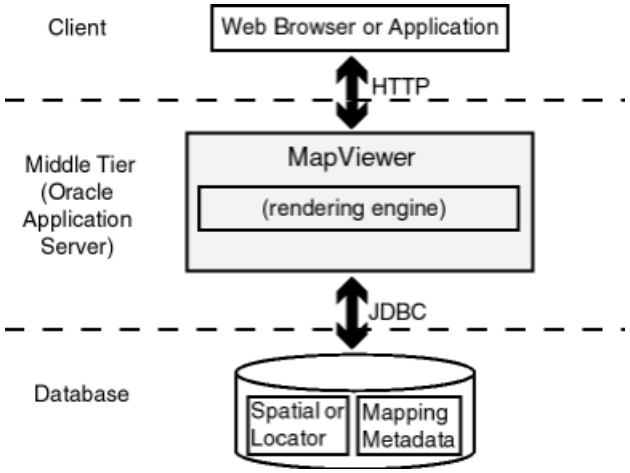

Figure 1: Mapviewer Architecture

The current release of MapViewer includes a new suite of technologies, called Oracle Maps, which consists of a map cache server and a Javascript library for developing web mapping client applications. The Oracle Maps architecture and functionality is described in a separate section later in this paper.

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 Weblogic 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.

New and Improved Features in MapViewer 12c

New and improved features in Mapviewer 12c include:

Oracle Maps

- A new version (V2) of the Javascript API: The new version takes advantage of the capabilities of modern browsers that support HTML5 Canvas and SVG. The new API will not work with older browsers that do not support Canvas or SVG. It is not backward compatible with V1 of the API. Both versions, V1 and V2, will continue to co-exist. Some features of the new API include:
    - Rich client side rendering of geospatial data with on-the-fly application of rendering styles and effects such as gradients, animation, and drop-shadows.
    - Autoclustering of large number of points and client side heat map generation.
    - Built-in support of various third party map tile services, such as maps.oracle.com, Nokia Maps, Bing Maps, OpenStreet Maps, and other mapping service providers.
    - Client side feature filtering based on attribute values and spatial predicates (query windows).
- Multi touch mobile device support: The Javascript APIs support multi-touch gestures (pinch, swipe) on iOS and late-version Android devices. No code modifications are needed in the client application.
- External map tile layer support: The Oracle Maps client library can now access and display map tiles directly from an external provider. See the documentation for MVCustomMapTileLayer for details.

- Improved Client-Side support for accessing cross-domain map tile and FOI servers: The Oracle Maps client can now communicate with cross-domain map cache tile and FOI servers without relying on a proxy server, which was previously required. For more information, see the JavaScript API documentation for MVMapView.enableXMLHTTP.
- Improved Info-Window handling: The positioning, styling, and sizing of the information window have been improved. Previously, the Oracle Maps client always displays the information window at a fixed position relative to the specified map location. The Oracle Maps client now can place the information window at the optimal variable position relative to the specified map location. As the result, the map does not to be panned in order to make the information window visible inside the map. In addition, you can specify tabs for the information window. For more information, see the JavaScript API documentation for MVMapView's displayInfoWindow and displayTabbedInfoWindow. The Tabbed info window demo on the Oracle Maps tutorial page shows how to display a tabbed information window.
- Enhanced Map Decoration: The client now supports multiple collapsible map decoration components that can be positioned at arbitrary positions inside the map container. Map decoration can now be dragged inside the map container. For more information, see the JavaScript API documentation for MVMapDecoration.
- Flexible placement of navigation panel and scale bar: The navigation panel and the scale bar can now be placed inside a map decoration component, which can be displayed or hidden and can be placed at a position of your choice inside the map container.
- Improved error reporting: Previously, all error messages thrown by the Oracle Maps client were displayed as browser alerts. Now applications can customize how the error messages are handled by using a custom error handler. For more information, see the JavaScript API documentation for MVMapView.setErrorHandler.
- Individual theme feature highlighting: Applications can enable the user to select and highlight individual theme features (FOIs) by clicking the mouse on the features. For more information, see the JavaScript API documentation for MVThemeBasedFOI.enableHighlight and the Highlighting individual features of a theme based FOI layer demo on the Oracle Maps tutorial page.
- Wraparound map display: Applications can now display a map in the wraparound manner. When the map is displayed in this manner, the map wraps around at the map coordinate system boundary horizontally and therefore can be scrolled endlessly. For more information, see the JavaScript API documentation for MVMapView.enableMapWrapAround.
- Enhanced redline tool: The redline line tool can now be used to create polyline, polygon, and point geometries. The redline line tool also supports an editing mode, in which you can move an existing redline point or line segment, remove a redline point or line segment, or add a redline point or line segment programmatically.

- Automatic determination of whole image theme for FOI display: Displaying a theme-based FOI layer as a whole image may greatly improve the application performance, but it may be difficult for application developers to determine when to display a theme as a whole image theme. However, you can now choose to let MapViewer make the determination automatically. For more information, see the JavaScript API documentation for MVThemeBasedFOI.enableAutoWholeImage.
- Automatic recovery of long running tile generation administrative request: Long running tile admin requests that are interrupted due to Fusion Middleware or MapViewer shutdown will be able to resume automatically after MapViewer is restarted. (You do not need to do anything to enable this feature, other than creating the new database view USER_SDO_TILE_ADMIN_TASKS if it does not already exist).
- Built-in toolbar and distance measurement: Applications can now use a built-in distance measurement tool to measure distance on the map. The built-in toolbar provides an easy graphic user interface for accessing utilities such as the redline tool, rectangle tool, circle tool, distance measurement tool, and any user-defined capabilities. For more information, see the JavaScript API documentation for MVToolBar and the Tool bar demo on the Oracle Maps tutorial page.
- Annotation Text: MapViewer supports the OpenGIS Consortium"s annotation text standard. Oracle Spatial and Graph in Oracle Database Release 11g supports storage of annotation text objects in the database, and MapViewer displays such annotation texts on a map. For information about annotation text themes, see Section 2.3.11 of the user guide.

Other Enhancements

- MapViewer editor: A web-based (Java applet) for editing 2-D spatial data stored in Oracle sdo_geometry format. It supports multi-user, multi-session editing and optional use of Oracle Workspace Manager for versioning.
- Expanded third party spatial data source support: OGR/GDAL is now suported as a built-in spatial data provider. So MapViewer can access, and render, content from any OGR/GDAL supported spatial data source.
- Theme wide transparency: Any theme can now have a specified transparency when rendered.
- Embedded geometries and label text: MapViewer can now render geometry or label text that are instances of an attribute defined within a user-defined Oracle SQL object. For example, the Location attribute of a Warehouse object containing the geometry instance, and the Name attribute the label text.
- WMS Service enhancements: WMS 1.3 is now supported. Additionally a MapViewer now uses customizable configuration file to determine Oracle Spatial srid (SDO) to EPSG mappings for use in WMS requests and responses. The configuration file can also contain other GetCapabilities response elements (such as Contact Information) for the service.

- WMTS support: MapViewer now supports the OGC Web Map Tile Service (1.0) interface. A pre-defined theme can be based on an external WMTS compliant service (similar to a WMS or WFS theme). It can be used as a standalone theme. Or in a basemap and hence tile layer. Further details are provided in the MapViewer User Guide.
- Image processing operations with GeoRaster themes: Image processing operations, such as normalization or equalization, on GeoRaster themes can now be included in a map request.

Oracle MapBuilder tool

- WMTS, WMS, WFS, and Annotation themes: MapBuilder supports the definition, and use, of OGC's WMTS (web map tile service 1.0.0), WMS (web map service 1.1, 1.3), WFS (web feature service 1.0), and Annotation text themes.
- Automatic reduction of repetitive labels: Previously, repetitive street labels or highway shields on linear features were displayed when such features consisted of many small segments. Specifying 'No Repetitive Labels' option in the base map properties causes features (such as road segments) with same name to be labeled only once. For information about specific options in Map Builder, see the online help for that tool.
- Scale ranges for theme labeling: In the context of a base map, you can now assign scale limits to its themes' labels. These scale limits control when a theme's features will display their label texts.
- Heatmap support: MapViewer now supports heat maps, which are two-dimensional color maps of point data sets.
- Scalable styles: MapViewer now supports scalable styles. A scalable style (such as a Marker or Line style) uses real-world units such as meter or mile to specify its size and other dimensional attributes; however, at run time MapViewer automatically scales the style so that the features rendered by the style always show the correct size, regardless of the current map display scale. See the user guide for information about using scalable styles.
- Text Style Enhancements: The Text style has been improved to support customizable spacing between letters. It also supports additional (vertical) alignment options when labeling linear features.
- User-specified JDBC fetch size for predefined themes: You can now specify a nondefault row fetch size on a theme, by setting the Fetch Size base map property with the Map Builder tool. MapViewer can use this value when fetching theme features from the database. Specifying an appropriate value can increase performance in certain situations.
- Custom Spatial Data Provider Support: MapViewer now supports rendering of geospatial data stored in non-Oracle Spatial repositories. This is achieved through a Custom Spatial Data Provider API, where you can implement an Interface that feeds your own (proprietary) spatial data to MapViewer for rendering. Note that you will still need an Oracle Database to manage the mapping metadata, such as styles and themes definitions.

Developing applications with MapViewer

MapViewer is a developer's toolkit for interactive web mapping; that is, a set of programmable Java components for rendering maps from spatial data that is stored and managed in Oracle Database. It queries and renders the content based on metadata stored in the database, and on current map context (e.g. scale and extent). So both the data and metadata are stored in an Oracle database and securely shared across an enterprise.

MapViewer includes three sets of APIs. It has an XML request/response protocol for embedding static maps in web pages, and a Java API for embedding maps in an application. Both APIs support interactive querying of maps for features within and near an area of interest, and appropriate rendering of features depending on the map scale. The third is a JavaScript API in Oracle Maps, a component of MapViewer, that enhances interactivity to the next level by adding "slippy" capabilities; that is, the ability to pan, zoom and slide the map image in any direction.

All API depend on mapping metadata, described in further detail below, which specifies the data the be queried and the way it is to be rendered.

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.

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.
- Test: 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. Release 10g added support for Pie/Bar chart and dot density map styles, and a recent one included the heatmap style.

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 linear feature being rendered using three different line styles.

Figure 2: Same geometry rendered with 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.

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.

MapViewer supports several types of themes depending upon how they 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.

MapViewer 10g added support for Oracle Spatial GeoRaster and rendering geo-referenced images. 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. The MapViewer User Guide provides more details on using GeoRaster themes.

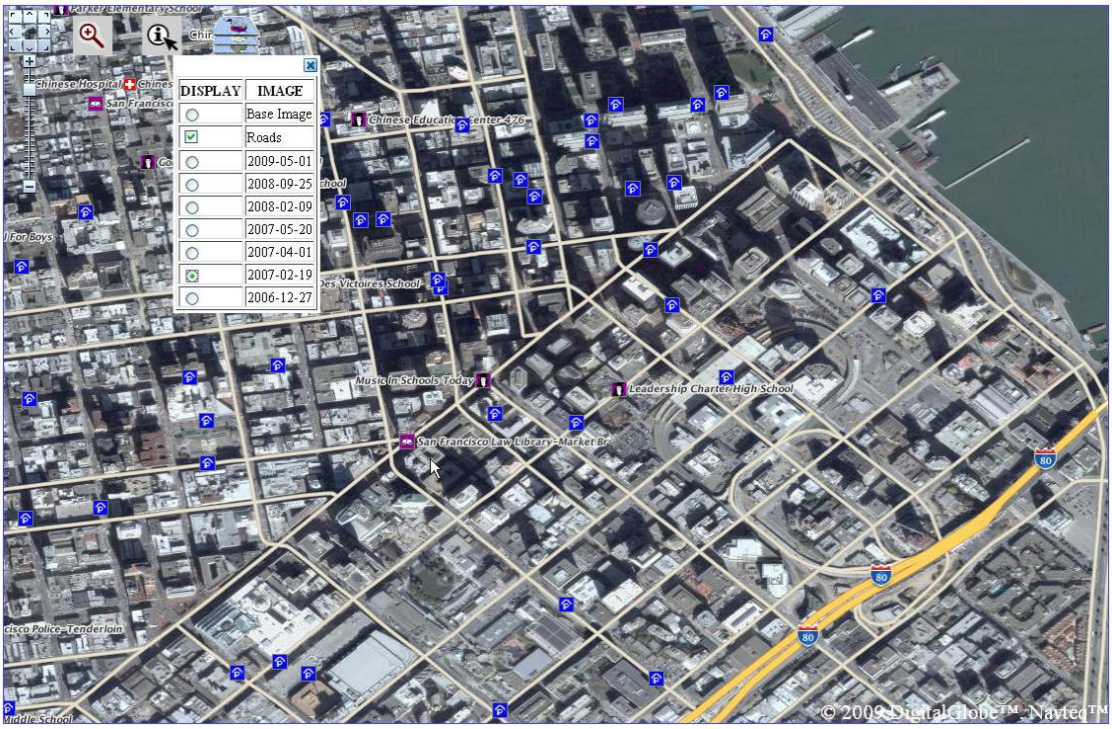

Figure 3. GeoRaster theme with overlaid vector themes.

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. Styling rules are a critical part of a theme's definition, and are discussed in the next section.

Styling Rules

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

- Which rows from the base table belong to a theme, and what **feature style** should be used to render the geometries from those rows. This means you can select only the desired rows (features) from the base table of a theme.
- 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 without storing the same set of map symbols many times.

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.

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.

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 state.

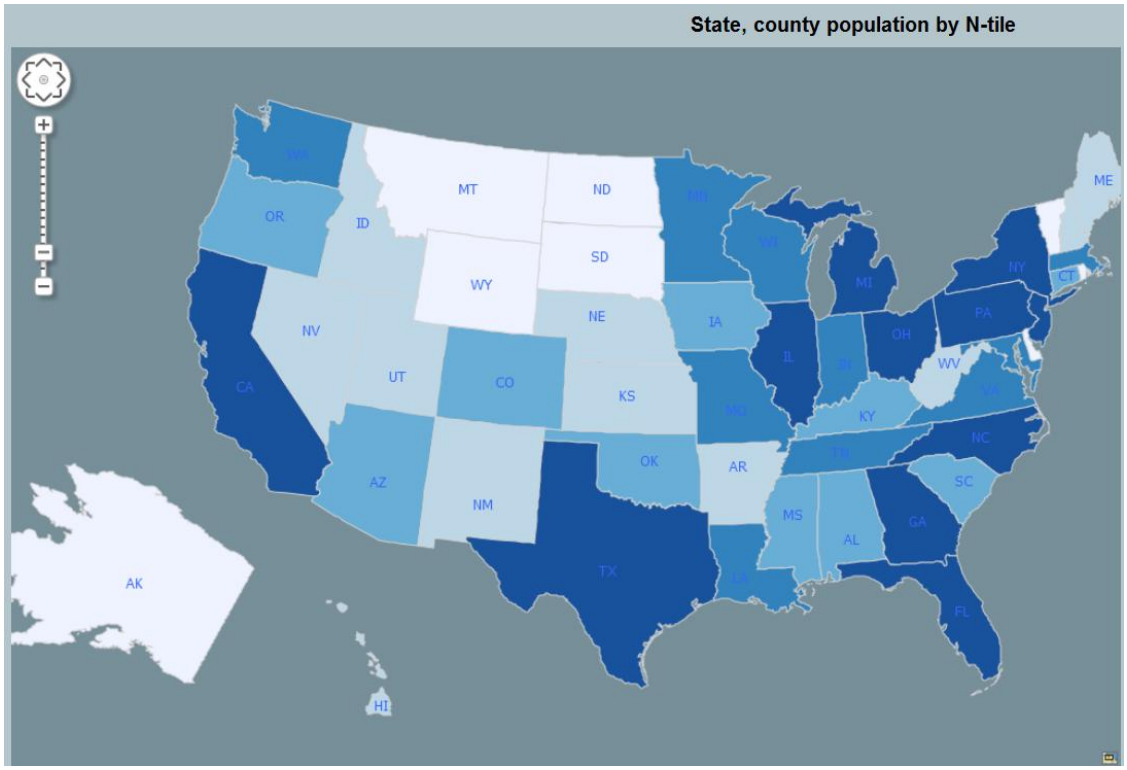

Figure 4. A simple thematic map.

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.

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.

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. 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.

MapViewer's REST API

MapViewer exposes its services through a REST API that accepts requests in the form of XML strings. This provides both the simplicity of the REST API (with just one query parameter per HTTP request, typically named xml_request), and the expressiveness of XML so that complex details about a map to be requested can be easily structured and managed. Through this 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.

The XML request/response format is also wrapped in a Java API (starting with MapViewer 9i) so that Java application developers can easily use the new API instead of manipulating XML map requests directly. Further information on both API is available in the MapViewer User Guide.

The MapBuilder Tool

MapBuilder is a standalone utility application to assist with the creation and management of mapping metadata. It is part of the MapViewer download kit on OTN. The tool helps you create, and store, styles, themes, and basemap definitions. It also provides interfaces to preview the metadata and geometries using the newly created styles (e.g. how a line or bucket style may appear on a map).

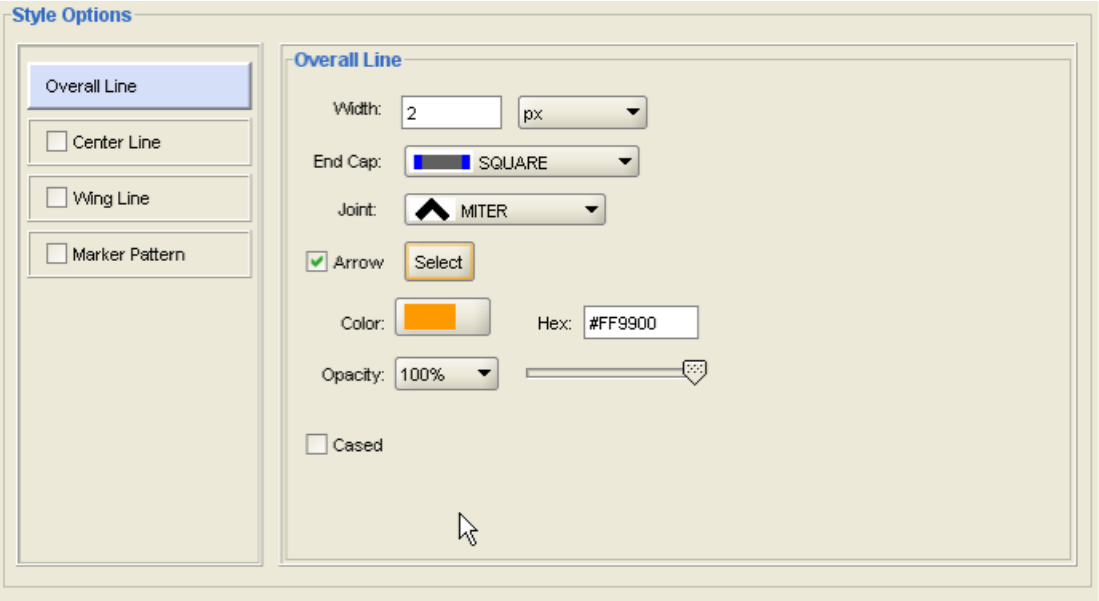

Figure 5. The MapBuilder tool for defining map metadata.

Oracle Maps

Oracle Maps is the name for a suite of technologies for developing high performance interactive web-based mapping applications. It consists of the following main components:

- A map cache server that caches and serves pregenerated map image tiles
- A feature of interest (FOI) server that renders geospatial features that are managed by Oracle Spatial and Graph
- A JavaScript client library for building interactive mapping applications. This client provides functions for browsing and interacting with maps, as well as a flexible application programming interface (API).

The map cache server (map image caching engine) automatically fetches and caches map image tiles rendered by MapViewer or other web-enabled map providers. It also serves cached map image tiles to

the clients, which are web applications developed using the Oracle Maps client API. The clients can then automatically stitch multiple map image tiles into a seamless large base map. Because the map image tiles are pre-generated and cached, the application users will experience fast map viewing performance.

The feature of interest (FOI) server (rendering engine) renders spatial feature layers managed by Oracle Spatial and Graph , as well as individual geospatial features of point, line, or polygon type that are created by an application. Such FOI, which typically include both an image to be rendered and a set of associated attribute data, are then sent to the client browser. Unlike the cached image tiles, which typically represent static content, FOI are dynamic and represent real-time database or application content. The dynamic FOIs and the static cached base map enable you to build web mapping applications.

The JavaScript mapping client is a browser side map display engine that fetches map content from the servers and presents it to client applications. It also provides customizable map-related user interaction control, such as map dragging and clicking, for the application. The JavaScript mapping client can be easily integrated with any web application or portal.

The following figure shows the architecture of a web mapping application developed using Oracle Maps.

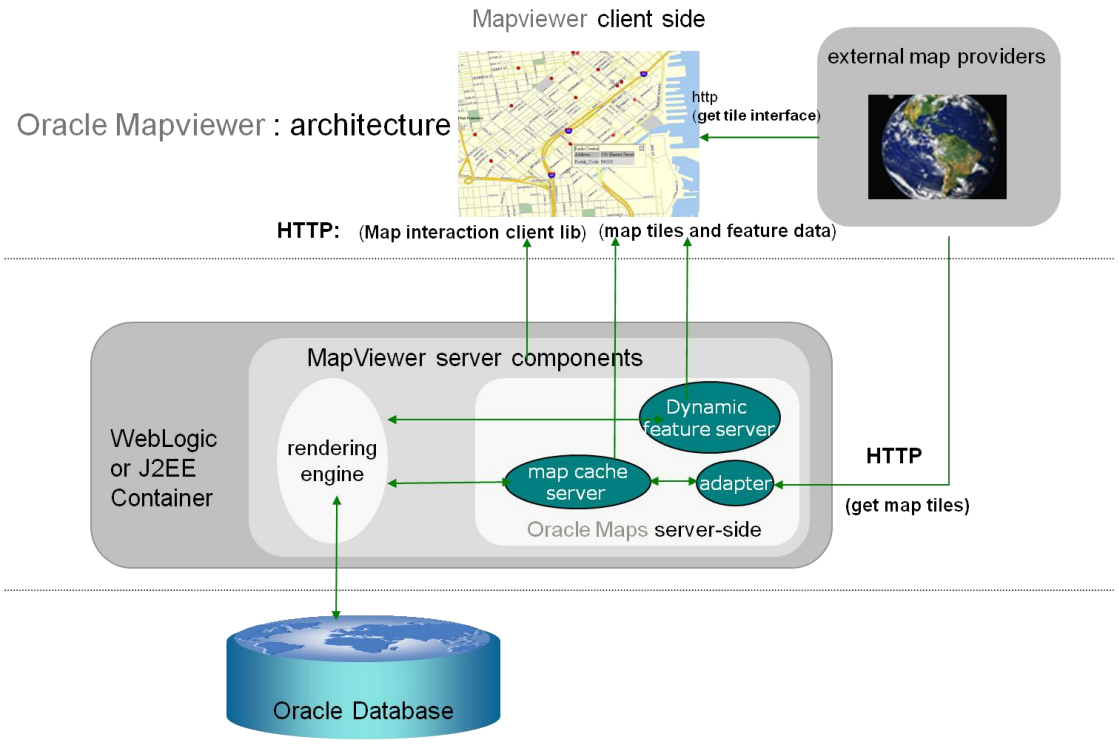

Figure 6. Oracle Maps Application Architecture

Applications interact with the Oracle Maps architecture as follows:

- The application is developed using JavaScript, and it runs inside the JavaScript engine of the web browser.
- The application invokes the JavaScript map client to fetch the map image tiles from the map cache server, and then it displays the map in the web browser.
- The application invokes the JavaScript map client to fetch dynamic spatial features from the FOI server and display them on top of the base map.
- The JavaScript map client controls map-related user interaction for the application.
- When the map cache server receives a map image tile request, it first checks to see if the requested tile is already cached. If the tile is cached, the cached tile is returned to the client. If the tile is not cached, the map cache server fetches the tile into the cache and returns it to the client. Tiles can be fetched either directly from the FMW MapViewer map rendering engine or from an external web map services provider.
- When the FOI server receives a request, it uses the FMW MapViewer map rendering engine to generate the feature images and to send these images, along with feature attributes, to the client.

The following figure shows the UI of a simple application developed using Oracle Maps. This example is included in the mvdemo.ear sample application in the quickstart kit and can be accessed at http://host:port/mvdemo/fsmc/sampleApp.html after it is installed and configured. Instructions on running the application are accessible at: http://host:port/mvdemo/fsmc/tutorial/setup.html. The MVDEMO sample application comes with over 50 tutorials illustrating various aspects of the Oracle Maps functionality.

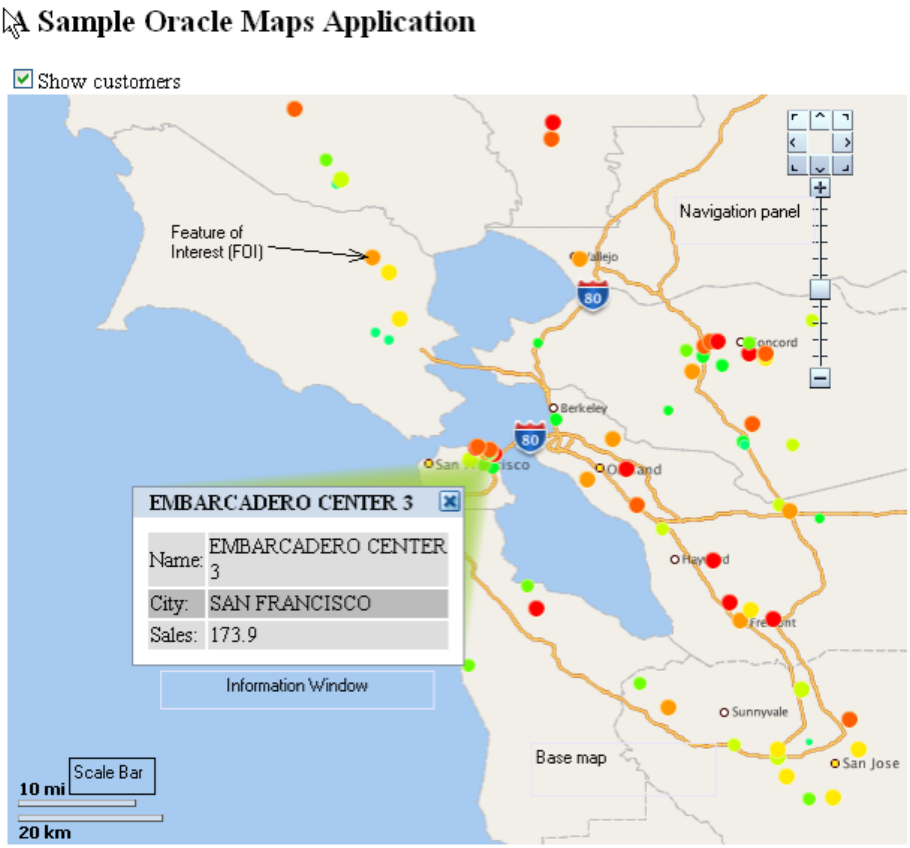

Figure 7. A sample Oracle Maps application.

The sample application displays the locations of customers on a base map. The map consists of two layers:

- The base map layer displays the ocean, county boundaries, cities, and highways.
- The FOI layer displays customer location as red dot markers on top of the base map.

In addition to these two layers, a scale bar is displayed in the lower-left corner of the map, and a navigation panel is displayed in the upper-right corner.

The application user can use the mouse to drag the map. When this happens, new image tiles and FOIs are automatically fetched for the spatial region that the map currently covers. The user can also use the built-in map navigation tool to pan and zoom the image, and can show or hide the customers (colored dot markers) by checking or unchecking the 'Show customers' box.

The source code for this application is given below.

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html" charset=UTF-8">
<TITLE>A sample Oracle Maps Application</TITLE>
<script language="Javascript" src="jslib/oraclemaps.js"></script>
```

```
<script language=javascript>
var themebasedfoi=null
function on_load_mapview()
{
       var baseURL = "http://"+document.location.host+"/mapviewer";
       // Create an MVMapView instance to display the map
       var mapview = new MVMapView(document.getElementById("map"),
                           baseURL);
       // Add a base map layer as background.
       mapview.addMapTileLayer(new MVMapTileLayer("mvdemo.demo_map"));
       // Add a theme-based FOI layer to display customers on the map
       themebasedfoi = new MVThemeBasedFOI('themebasedfoi1',
                           'mvdemo.customers');
       themebasedfoi.setBringToTopOnMouseOver(true);
       mapview.addThemeBasedFOI(themebasedfoi);
       // Set the initial map center and zoom level
       mapview.setCenter(MVSdoGeometry.createPoint(
                           -122.45,37.7706,8307));
       mapview.setZoomLevel(4);
       // Add a navigation panel on the right side of the map
       mapview.addNavigationPanel('east');
       // Add a scale bar
       mapview.addScaleBar();
       // Display the map.
       mapview.display();
}

function setLayerVisible(checkBox)
{
       // Show the theme-based FOI layer if the check box is checked
       // and hide the theme-based FOI layer otherwise.
       if(checkBox.checked)
              themebasedfoi.setVisible(true) ;

       else
              themebasedfoi.setVisible(false);

}
</script>
</head>
<body onload= javascript:on_load_mapview() >
<h2> A sample Oracle Maps Application</h2>
<INPUT TYPE="checkbox" onclick="setLayerVisible(this)"
checked/>Show customers
<div id="map" style="width: 600px; height: 500px"></div>
</body>
</html>
```

Creating the sample application with the V2 API

Oracle Maps V2 applications run inside web browsers and require only HTML5 (Canvas) support and JavaScript enabled. No additional plugins are required.

Developing applications with the V2 API is similar to the process for the V1 API. If all the spatial data used for base maps, map tile layers, and interactive layers or themes are stored in an Oracle database then the map authoring process using MapBuilder is the same for both.

Each map tile layer displayed in the client application must have a corresponding database metadata entry (i.e. in user_sdo_cached_maps) if the underlying base map and layers are managed in an Oracle database. Similarly each interactive layer must have a metadata entry (in user_sdo_themes) if it is based on database content. These tile and interactive, and the styles and styling rules for them, can be defined using the MapBuilder tool.

The source for an Oracle Maps application is typically packaged in an HTML page, which consists of the following parts:

- A script element that loads the Oracle Maps V2 client library into the browser's JavaScript engine, e.g. <script src="/mapviewer/jslib/v2/oraclemapsv2.js"></script>
- An HTML DIV element that will contain the map, e.g.
- Javascript code that creates the map client instance, sets the initial map content (tile and vector layer), the initial center and zoom, and map controls. This code should be packaged inside a function which is executed when the HTML page is loaded or ready. The function is specified in the onload attribute of the <body> element of the HTML page.

```
function on_load_mapview()
{
  var baseURL  = "http://"+document.location.host+"/mapviewer";
  // Create an OM.Map instance to display the map
  var mapview = new OM.Map(document.getElementById("map"),
                {
                              mapviewerURL:baseURL
                 });
   // Add a map tile layer as background.
  var tileLayer = new OM.layer.TileLayer(
        "baseMap",
        {
            dataSource:"mvdemo",
            tileLayer:"demo_map",
            tileServerURL:baseURL+"/mcserver"
        });
  mapview.addLayer(tileLayer);
  // Set the initial map center and zoom level
  var mapCenterLon = -122.45;
```

```
  var mapCenterLat = 37.7706;
  var mapZoom = 4;
  var mpoint = new OM.geometry.Point(mapCenterLon,mapCenterLat,8307);
  mapview.setMapCenter(mpoint);
  mapview.setMapZoomLevel(mapZoom);
  // Add a theme-based FOI layer to display customers on the map
  customersLayer = new OM.layer.VectorLayer("customers",
        {def:
                {
                type:OM.layer.VectorLayer.TYPE_PREDEFINED,
                dataSource:"mvdemo", theme:"customers",
                url: baseURL,
                loadOnDemand: false
                }
        });
  mapview.addLayer(customersLayer);
  // Add a navigation panel on the right side of the map
  var navigationPanelBar = new OM.control.NavigationPanelBar();
  navigationPanelBar.setStyle(
{backgroundColor:"#FFFFFF",buttonColor:"#008000",size:12});
  mapview.addMapDecoration(navigationPanelBar);
  // Add a scale bar
  var mapScaleBar = new OM.control.ScaleBar();
  mapview.addMapDecoration(mapScaleBar);
  // Display the map.
  //Note:Change from V1. In V2 initialization, display is done just once
  mapview.init();
}
```

- Additional HTML elements and JavaScript code implement other application-specific user-interfaces and control logic. For example the HTML <input> element and Javascript function setLayerVisible() together implement a layer visibility control. The function is specified in the onclick attribute of the <input> element defining the checkbox and hence is executed whenever the user clicks on the Show Customers check box.

```
<INPUT TYPE="checkbox" onclick="setLayerVisible(this)" checked/>Show
customers
```

The setLayerVisible function is coded as follows:

```
function setLayerVisible(checkBox)
{
  // Show the customers vector layer if the check box is checked and
  // hide it otherwise
  if(checkBox.checked)
     customersLayer.setVisible(true) ;
```

```
    else
      customersLayer.setVisible(false);
  }
```

Oracle Maps client applications running inside web browsers are pure HTML pages, containing Javascript code, which do not require any plug-ins. Therefore, you can build the application using any Web technology that delivers content as pure HTML.

Conclusion

MapViewer provides web application developers a versatile means to integrate and visualize business data with maps. It uses the basic capability included with the Oracle Database (either Oracle Spatial and Graph 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.

MapViewer 12c Technical Overview
July 2016
Author: Jayant Sharma
Contributing Authors: LJ Qian

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment