MapGuide Open Source

Developer's Guide

March 2006

Contents

Chapter 1 **Introduction** 1

What This Guide Covers 2
Essential Concepts 2
Preparing to Run the Examples 2
Recommended Directory Structure 3
Hello, Map – Basic Map Information 3
Web Layouts and MapGuide Server Pages 3
MapGuide Page Flow 4
Example Code 5
Running the Example 7
How This Page Works 7
Understanding Viewer Frames 8
Interactions Between Frames 11
MapGuide Viewer API 11
Calling the Viewer API from the Task Pane 12
Calling the Viewer API from the Script Frame 13
Calling the Viewer API with an Invoke Script Command 13
MapGuide Web API 14
Embedding a Viewer in Your Own Page 14
Session Management 14
Resources and Repositories 16
Library and Session 17

Maps . . . 17
Understanding Services . . . 18

Chapter 2 **Interacting With Layers . . . 21**
Overview of Layers . . . 22
Basic Layer Properties . . . 22
Layer Groups . . . 22
Base Layer Groups . . . 23
Layer Style . . . 24
Layer Visibility . . . 24
Example: Actual Visibility . . . 24
Refresh and Zoom . . . 25
Example . . . 25
Enumerating Map Layers . . . 26
Example . . . 26
Manipulating Layers . . . 27
Changing Basic Properties . . . 27
Example . . . 27
Changing Visibility . . . 29

Chapter 3 **Working With Feature Data . . . 31**
Overview of Features . . . 32
Querying Feature Data . . . 33
Feature Readers . . . 33
Selecting with the Web API . . . 33
Basic Filters . . . 34
Spatial Filters . . . 35
Example: Listing Selected Features . . . 37
Active Selections . . . 40
Selecting with the Viewer . . . 40
Working With the Active Selection . . . 41
Example: Listing Selected Parcels (AJAX Viewer) . . . 42
Example: Listing Selected Parcels (DWF Viewer) . . . 44
Setting the Active Selection With the Web API . . . 45
Example: Setting the Active Selection . . . 45

Chapter 4 **Modifying Maps and Layers . . . 51**
Introduction . . . 52
Adding An Existing Layer To A Map . . . 52
Creating Layers By Modifying XML . . . 52
Another Way To Create Layers . . . 55
Example - Creating A Layer That Uses Area Rules . . . 59
Example - Using Line Rules . . . 60
Example - Using Point Rules . . . 61

Adding Layers To A Map 63
Making Changes Permanent 66

Index 67

Introduction

1

In this chapter

- What This Guide Covers
- Essential Concepts
- Preparing to Run the Examples
- Hello, Map – Basic Map Information
- Understanding Viewer Frames
- Interactions Between Frames
- Embedding a Viewer in Your Own Page
- Resources and Repositories
- Understanding Services

What This Guide Covers

This guide describes how to use the MapGuide Open Source Web API.

It assumes you have read the *MapGuide Getting Started* guide and are familiar with using Autodesk® MapGuide Studio. Some examples also assume that you have installed the sample data and sample application supplied with MapGuide.

This guide provides a high-level overview of the APIs. More detailed information is provided in the on-line *MapGuide Web API Reference* and *MapGuide Viewer API Reference*.

Essential Concepts

Refer to the *MapGuide Getting Started* guide for details about the MapGuide architecture and components. It is important to understand the relationship between a MapGuide Viewer, a MapGuide Web application, and the MapGuide site. It is also important to understand resources and repositories.

Web applications reside on the Web Server. They are normally executed by requests from a MapGuide Viewer. They can in turn communicate with the MapGuide site and send data back to the Viewer.

When you define a web layout, using Studio or some other method, you also define toolbar and menu commands. These can be standard pre-defined Viewer commands like pan, zoom, and refresh, or they can be custom commands. Custom commands are a way of extending MapGuide to interact with your mapping data. The custom commands are HTML pages, generated on the server using PHP, ASP.NET, or Java. These languages can use the Web API to retrieve, manipulate, and update mapping data.

Many custom commands run in the *task area*, a section of the Viewer that is designed for user input/output. For more details about the task area and how it integrates with the rest of the Viewer, see Understanding Viewer Frames (page 8).

Preparing to Run the Examples

This guide includes many examples. Some are complete, while others are partial. To run any of the partial samples, you must add standard initialization and error-checking steps. Refer to any of the complete examples for details.

Recommended Directory Structure

When you install MapGuide Web Extensions, it creates a www directory to act as a virtual directory for your web server. Within this directory are other directories for MapGuide administration, the map agent, and the Viewers.

If you install the PHP sample application, it is placed in the www\phpviewersample directory.

For the examples in this guide, you should create a www\devguide directory. Use this for any examples and tests you write.

Hello, Map – Basic Map Information

NOTE The Web API supports .NET, Java, and PHP. For simplicity, all the examples in this guide use PHP.

To run the examples on a Linux installation, change any Windows-specific file paths to corresponding Linux paths.

This first sample MapGuide page displays some basic information about a map. It does not do any complicated processing. Its purpose is to illustrate the steps required to create a MapGuide page and have it connect to a Viewer on one side and the MapGuide site on the other.

Web Layouts and MapGuide Server Pages

A *MapGuide Server Page* (MSP) is any PHP, ASP.NET, or JSP page that makes use of the MapGuide Web API. MSPs are typically invoked by the MapGuide Viewer or browser and when processed result in HTML pages that are loaded into a MapGuide Viewer or browser frame. This is the form that will be used for most examples in this guide. It is possible, however, to create MSPs that do not return HTML or interact with the Viewer at all. These can be used for creating web services as a back-end to another mapping client or for batch processing of your data.

Creating an MSP requires initial setup, to make the proper connections between the Viewer, the page, and the MapGuide site. Much of this can be done using Studio. Refer to the *Studio Help* for details.

One part of the initial setup is creating a web layout, which defines the appearance and available functions for the Viewer. When you define a web layout, you assign it a resource name that describes its location in the repository. The full resource name looks something like this:

```
Library://Samples/Sheboygan/Layouts/SheboyganPhp.WebLayout
```

When you open the web layout using a browser with either the AJAX Viewer or the DWF Viewer, the resource name is passed as part of the Viewer URL. Special characters in the resource name are URL-encoded, so the full URL would look something like this, (with line breaks removed):

```
http://localhost/mapguide/mapviewerajax/
?WEBLAYOUT=Library%3a%2f%2fSamples%2fSheboygan%2fLayouts%2f
SheboyganPhp.WebLayout
```

Part of the web layout defines commands and the toolbars and menus that contain the commands. These commands can be built-in commands, or they can be URLs to custom MSPs. The custom MSPs are what make up your application.

To create a new MSP and make it available to a Viewer, add a command to the web layout. Set the command type to Invoke URL. Set the URL of the command to the URL of your page, and add the command to the Task Bar Menu.

It is possible to add custom commands to other menus as well. For most of the examples in this guide, however, the commands will be part of the Task Bar Menu. Output from an MSP normally appears in the task pane, though you may direct it to another frame or new window if desired.

MapGuide Page Flow

Most MSPs follow a similar processing flow. First, they create a connection with the site server. Then they open connections to any needed site services. The exact services required depend on the MSP function. For example, a page that deals with map feature data requires a feature service connection.

Once the site connection and any other service connections are open, the page can use MapGuide Web API calls to retrieve and process data. Output goes to the task pane or back to the Viewer. See MapGuide Viewer API (page 11) for details about sending data to the Viewer.

When a user first connects to a MapGuide site, the site creates a session for that user. If you use a MapGuide Viewer to display your map, the session is created automatically. This also generates a unique session ID, which the

Viewer uses to manage the run-time map state. This keeps the state consistent between the viewer and the server across multiple HTTP requests.

NOTE MapGuide pages written in PHP require one additional step because PHP does not support enumerations compiled into extensions. To deal with this limitation, PHP Web Extension pages must include `constants.php`, which is in the `mapviewerphp` folder. This is not required for ASP.NET or JSP pages.

Example Code

This sample illustrates basic page structure. It is designed to be called as a task from a Viewer. It connects to a MapGuide server and displays the map name and spatial reference system for the map currently being displayed in the Viewer. See Running the Example (page 7) for details about how to install and run this page.

It is also possible to embed a Viewer in your own page, so you can supply your own logo, page header information, or authorization. See Embedding a Viewer in Your Own Page (page 14) for details.

NOTE The following contains complete source for the page. Most of the examples in this guide will only contain excerpts of the page source. The outer HTML and the standard initialization steps will not be repeated in most cases.

```
<html>
   <head><title>Hello, map</title></head>
   <body>
      <p>
      <?php
      // Define some common locations
      $installDir =
        'C:\Program Files\MapGuideOpenSource\\';
      $extensionsDir = $installDir . 'WebServerExtensions\www\\';
      $viewerDir = $extensionsDir . 'mapviewerphp\\';
      // constants.php is required to set some enumerations
      // for PHP. The same step is not required for .NET
      // or Java applications.
      include $viewerDir . 'constants.php';
      try
      {
         // Get the session information passed from the viewer.
         $mgSessionId = ($_SERVER['REQUEST_METHOD'] == "POST")
            ? $_POST['SESSION']: $_GET['SESSION'];
         $mgMapName = ($_SERVER['REQUEST_METHOD'] == "POST")
            ? $_POST['MAPNAME']: $_GET['MAPNAME'];
         // Basic initialization needs to be done every time.
         MgInitializeWebTier("$extensionsDir\webconfig.ini");
         // Get the user information using the session id,
         // and set up a connection to the site server.
         $userInfo = new MgUserInformation($mgSessionId);
         $siteConnection = new MgSiteConnection();
         $siteConnection->Open($userInfo);
         // Get an instance of the required service(s).
         $resourceService = $siteConnection->
         CreateService(MgServiceType::ResourceService);
        // Display the spatial reference system used for the map.
         $map = new MgMap();
         $map->Open($resourceService, $mgMapName);
         $srs = $map->GetMapSRS();
         echo 'Map <strong>' . $map->GetName() .
           '</strong> uses this reference system: <br />' . $srs;
      }
      catch (MgException $e)
      {
         echo "ERROR: " . $e->GetMessage() . "<br />";
         echo $e->GetStackTrace() . "<br />";
      }
```

```
        ?>
        </p>
    </body>
</html>
```

Running the Example

To run the example, perform the following steps:

1. Save it where it is accessible by your web server.

   For this example, name it hellomap.php, and save it in the www\devguide directory.

2. Using Studio, create or modify a web layout. Create a new command. Name the command Hello Map. Set the command type to Invoke URL. Set the command URL to the URL of hellomap.php.

   ```
   ../devguide/hellomap.php
   ```

3. Add the command to the Task Menu of the web layout.

4. Open a web browser to the URL of the web layout. The URL is available when you edit the layout using Studio. Use either the DWF Viewer or the AJAX Viewer.

5. Select Hello Map from the task list. Click Tasks (at the top right of the Viewer windows), and select Hello Map.

   The task pane displays basic information about the map.

How This Page Works

This example page performs the following operations:

1. Get session information.

   When you first go to the URL containing the web layout, the Viewer initiates a new session. It prompts for a user id and password, and uses these to validate with the site server. If the user id and password are valid, the site server creates a session and sends the session id back to the viewer.

   The Viewer passes the session information every time it sends a request to a MapGuide page. The pages use this information to re-establish a session.

2 Perform basic initialization.

The webconfig.ini file contains parameters required to connect to the site server, including the IP address and port numbers to use for communication. `MgInitializeWebTier()` reads the file and gets the necessary values to find the site server and create a connection.

3 Get user information.

The site server saves the user credentials along with other session information. These credentials must be supplied when the user first connects to the site. At that time, the Viewer authenticates the user and creates a new session using the credentials. Using the session ID, the pages can get an encrypted copy of the user credentials that can be used for validation.

4 Create a site connection.

Any MapGuide pages require a connection to a site server, which manages the repository and site services.

5 Create a connection to a resource service.

Access to resources is handled by a resource service. In this case, the MSP needs a resource service in order to retrieve information about the map resource.

You may need to create connections to other services, depending on the needs of your application.

6 Retrieve map details.

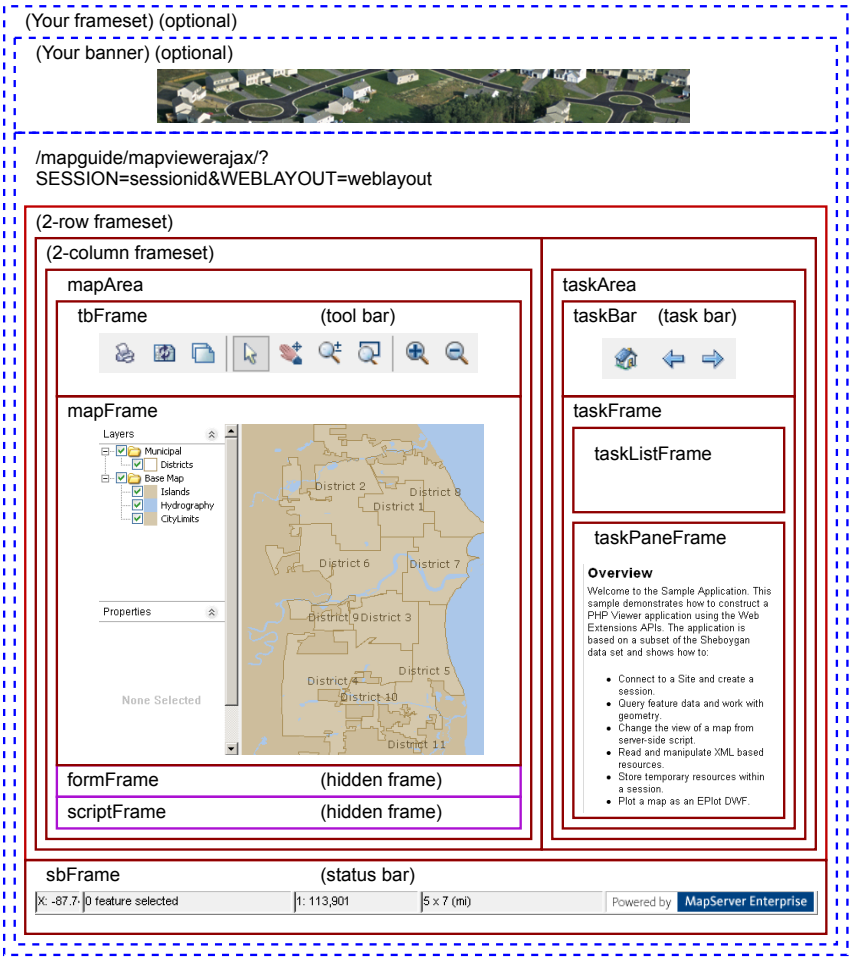
The map name is also passed by the viewer to the MapGuide page. Use this name to open a particular map resource with the resource service. Once the map is open you can get other information. This example displays the spatial reference system used by the map, but you can also get more complex information about the layers that make up the map.

Understanding Viewer Frames

The
MapGuide Viewers

use HTML frames to divide the viewer area. Refer to the diagram for the locations of the following frames and frame sets:

Name	Description
	Unnamed. Contains all the Viewer frames. This can be wrapped by an outer frame so you can embed the Viewer in your own site..
`maparea`	Frame set containing the tool bar, map area, form frame, and script frame.
`tbFrame`	Frame containing the tool bar. Add new commands to the tool bar by modifying the web layout using Studio.
`mapFrame`	Frame containing the map data. This includes the map display and the layers and properties palettes.
`formFrame`	Hidden frame that can be used to generate HTTP POST requests for sending data to the server.
`scriptFrame`	Hidden frame that can be used for executing client-side JavaScript.
`taskArea`	Frame set containing the task bar and the task frame.
`taskBar`	Frame containing the task bar.
`taskFrame`	Frame used to hold the task list frame and the task pane frame.
`taskListFrame`	Frame used for displaying the task list. This is normally hidden, and is shown when a user clicks the task list button in the task bar. Add new commands to the task list by modifying the web layout.
`taskPaneFrame`	Frame used for displaying and executing MapGuide pages.
`sbFrame`	Frame containing the status bar.



Viewer Frames

Most custom MSPs display in the task pane. When you create a custom command in a web layout and add it to the task menu, it will appear in the task list that displays when a user clicks the Tasks button.

NOTE You can customize the web layout so some of the frames do not display. For example, if you hide the task bar and task frame, then the entire task area does not display.

Interactions Between Frames

A MapGuide page may need to perform three types of interactions:

- User input and output
- Commands to manipulate the Viewer
- Communication with the site server

User input and output is done directly through the HTML code in the page itself. This can include any standard HTML interaction, like hyperlinks and form input.

Viewer commands are done with the MapGuide Viewer API, which is a collection of JavaScript commands available in the Viewer frames. See MapGuide Viewer API (page 11) for details.

Communication with the site server is done using the MapGuide Web API.

MapGuide Viewer API

The MapGuide Viewer API is a set of JavaScript functions used to control the Viewer. Many of the Viewer frames contain embedded JavaScript functions that can be called from other locations. For full details about the available functions, refer to the online *MapGuide Viewer API Reference*.

To execute any of the Viewer API functions, call them from JavaScript embedded in a page. There are three common techniques for this:

- Execute the JavaScript call when the MapGuide page loads in the task pane. You can perfom all the necessary processing in advance of loading the page, then emit a function containing the correct parameters. Use this technique when you want the Viewer to change when the page loads.
- Execute the JavaScript inside the script frame. Use this technique when you want the Viewer to change as a result of an action in the MapGuide page, without reloading the page.
- Call the Viewer API during client-side interaction with the page or using the Invoke Script command type in the web layout. Use this technique when you want to call the API directly from the tool bar.

NOTE It is important to know the relationships between the frames. Pages in the task area must refer to `parent.parent.mapFrame` in order to traverse the frame hierarchy properly. However, if the same function executes from the script frame or the task bar, it only needs to refer to `parent.mapFrame`, because the script frame and the map frame are part of the same frame set.

Many Viewer API calls will generate requests to the site server, either to refresh data in the Viewer or to notify the site server of a change in Viewer state. These requests are generated automatically.

Calling the Viewer API from the Task Pane

Use this technique when you want the Viewer API calls to be made when the page loads. For example, if you have a task in the task list that zooms the map to a pre-defined location, then you do not need any user input. The Viewer should zoom as soon as the page loads.

The map frame contains a JavaScript function to center the map to a given coordinate at a given map scale. To call this function from a page loading in the task pane, create a function that will be executed when the `onLoad` event occurs. The following is a simple example. If you add this to the task list and select the task, the displayed map will reposition to the given location.

```
<html>
<head>
   <title>Viewer Sample Application - Zoom</title>
</head>
<script language="javascript">
function OnPageLoad()
{
   parent.parent.ZoomToView(-87.7116768,
      43.7766789973, 5000, true);
}
</script>
<body onLoad="OnPageLoad()">
<h1>Zooming...</h1>
</body>
</html>
```

Calling the Viewer API from the Script Frame

Use this technique when you want the Viewer API calls to be made as a result of an action in the calling page, but you do not want to reload the page. For example, you may have a page that generates a list of locations and you would like the user to be able to jump directly to any location, while leaving the list still available in the task pane.

In this case, your page can load another page in the hidden script frame, using `target="scriptFrame"` as part of the `<a>` tag. This requires that you create a separate page to load in the script frame and that you pass the necessary parameters when the page loads.

For example, the sample application includes a page named gotopoint.php. This is designed to be run in the script frame. The `<body>` element is empty, so the page does not produce any output. Instead, it emits a JavaScript function to execute when the page loads. This function calls the `ZoomToView()` function in the Viewer API.

To execute gotopoint.php from another page, create a hyperlink that passes the coordinates and zoom amount as HTTP GET parameters. Set `target="scriptFrame"`. When a user clicks the link, gotopoint.php is loaded in the script frame, but the calling page does not change. For example, the following could be included as part of a page in the task pane:

```
<a href="gotopoint.php?X=-87.7116768&Y=43.7766789973&Scale=5000"
   target="scriptFrame">Position map</a>
```

Calling the Viewer API with an Invoke Script Command

Use this technique when you want to call the API directly from the tool bar.

For example, you may want to create a tool bar button that zooms and positions the map to show a particular location. In the web layout, create a command of type Invoke Script. Enter the API call as the script to invoke:

```
ZoomToView(-87.7116768, 43.7766789973, 5000, true);
```

When a user clicks the button, the map view will reposition to the location.

Commands of type Invoke Script always execute in the context of the main frame. This means that all main frame functions are available. To execute a

function in another frame, use the frame name as part of the function name. For example, `formFrame.Submit()`.

To add your own JavaScript functions, you can embed a Viewer in another page. See Embedding a Viewer in Your Own Page (page 14) for details. In this case, any JavaScript functions defined in the outer page are available to scripts as `parent.functionName()`, where *functionName* is the name of your function.

MapGuide Web API

The MapGuide Web API provides functions for geospatial processing using your map data. These functions may be executed in the context of the Web Server. If they are service APIs then they are executed in the context of the Site Server.

NOTE When you make calls to the Web API, this may result in changes to data in the repository. These changes are not automatically sent to the Viewer. You must also call the Viewer API function `refresh()`.

Embedding a Viewer in Your Own Page

The simplest way to incorporate a MapGuide Viewer into your own site is to create a frame set that contains a frame for your own page layout and a frame for the Viewer. In the sample application, dwfviewersample.php and ajaxviewersample.php show this technique. They both create a frame set where the top frame in the set contains a site-specific page header, and the bottom frame in the set contains the embedded Viewer.

Session Management

In most cases, your application can let the MapGuide Viewer manage the session state automatically. When the Viewer first loads it creates a session, and it passes the session ID every time it loads an application page.

At times, though, you may want to create the session yourself, in the custom HTML surrounding the Viewer. When you do this, then the session information is available to the surrounding pages, instead of being just available to the Viewer.

You can also create your own session when you want a custom login page or when you want to allow anonymous access without prompting for a user id and password.

For example, you may want to hide the tool bar and task frame completely, and provide all the MapGuide functions directly through the HTML in your custom page. In order to do this, your HTML must pass the session ID to any pages it loads.

To create a new session, authenticate with a user id and password, then use this to create a site connection and new session, as in the following example. Note that this example uses some files from the sample application. You will either need to copy the files to the devguide directory or run this example from the phpviewersample directory.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
  "http://www.w3.org/TR/html4/frameset.dtd">
<?php
   include 'AppConstants.php';
   MgInitializeWebTier ($configFilePath);
   $webLayout =
    "Library://Samples/Sheboygan/Layouts/SheboyganPhp.WebLayout";
   $user = 'Administrator';
   $password = 'admin';
   $userInfo = new MgUserInformation($user, $password);
   $site = new MgSite();
   $site->Open($userInfo);
   $sessionId = $site->CreateSession();
?>
<html>
<head>
   <title>Viewer Sample Application</title>
   <meta content="text/html; charset=utf-8"
    http-equiv="Content-Type">
   <meta http-equiv="content-script-type"
    content="text/javascript" />
   <meta http-equiv="content-style-type" content="text/css" />
   <link href="styles/globalStyles.css"
    rel="stylesheet" type="text/css">
</head>
<frameset rows="110,*" frameborder="NO" border="0"
   framespacing="0">
  <frame src="Title.php?AppName=HTML" name="TitleFrame"
   scrolling="NO" noresize>
  <frame src="/mapguide/mapviewerphp/ajaxviewer.php?
   SESSION=<?= $sessionId ?>&WEBLAYOUT=<?= $webLayout ?>"
   name="ViewerFrame">
</frameset>
</html>
```

Resources and Repositories

A MapGuide *repository* is a database that stores and manages the data for the site. The repository stores all data except data that is stored in external databases. Data stored in a repository is a *resource*.

Types of data stored in the repository:

- Feature data from SHP and SDF files
- Drawing data from DWF files
- Map symbols
- Layer definitions
- Map definitions
- Web layouts
- Connections to feature sources, including database credentials

Library and Session

Persistent data that is available to all users is stored in the Library repository. This is the repository that changes when you use Studio to create a web layout.

In addition, each session has its own repository, which stores the run-time map state. It can also be used to store other data, like temporary layers that apply only to an individual session. For example, a temporary layer might be used to overlay map symbols indicating places of interest.

Data in a session repository is destroyed when the session ends.

A resource identifier for a resource in the Library will always begin with `Library://`. For example:

```
Library://Samples/Sheboygan/Layouts/SheboyganPhp.WebLayout
```

A resource identifier for a session resource will always begin with `Session:`, followed by the session id. For example:

```
Session:70ea89fe-0000-1000-8000-005056c00008_en//layer.LayerDefinition
```

Maps

A map (`MgMap` object) is created from a map definition resource. The map definition contains basic information about the map, including things like

- the coordinate system used in the map
- the initial map extents

- references to the layer definitions for layers in the map

When the `MgMap` object is created, it is initialized with data from the map definition. As a user interacts with the map, the `MgMap` may change, but the map definition does not.

The map is saved in the session repository so it is available to all pages in the same session. You cannot save a map in the library repository.

Map creation is handled by the Viewers. When a Viewer first loads, it creates a map in the session repository. The map name is taken from the map definition name. For example, if a web layout references a map definition named `Sheboygan.MapDefinition`, then the Viewer will create a map named `Sheboygan.Map`.

If your application does not use a Viewer, you can create the map and store it in the repository yourself. To do this, your page must

- Create an `MgMap` object.
- Initialize the `MgMap` object from a map definition.
- Assign a name to the `MgMap` object.
- Save the map in the session repository.

For example, the following section of code creates an `MgMap` object named `Sheboygan.Map`, based on `Sheboygan.MapDefinition`.

```
$mapDefId = new MgResourceIdentifier(
  "Library://Samples/Sheboygan/Maps/Sheboygan.MapDefinition");
$map = new MgMap();
$mapName = $mapDefId->GetName();
$map->Create($resourceService, $mapDefId, $mapName);
$mapId = new MgResourceIdentifier(
  "Session:$sessionId//$mapName." . MgResourceType::Map);
$map->Save($resourceService, $mapId);
```

Understanding Services

The MapGuide site performs many different functions. These can be all done by a single server, or you may balance the load across multiple servers. Some essential functions must execute on the site server, while other functions may execute on support servers.

A service performs a particular set of related functions. For example, a resource service manages data in the repository, a feature service provides access to feature data sources, and a mapping service provides visualization and plotting functions.

Before a page can use a service, it must open a site connection and create an instance of the necessary service type. The following example creates a resource service and a feature service:

```
$userInfo = new MgUserInformation($mgSessionId);
$siteConnection = new MgSiteConnection();
$siteConnection->Open($userInfo);
$resourceService = $siteConnection->
   CreateService(MgServiceType::ResourceService);
$featureService = $siteConnection->
   CreateService(MgServiceType::FeatureService);
```

Interacting With Layers

2

In this chapter

- Overview of Layers
- Refresh and Zoom
- Enumerating Map Layers
- Manipulating Layers

Overview of Layers

Layers represent vector data, raster data, and drawing data in a map. Each type of layer has unique characteristics.

NOTE The word *layer* has different meanings in different contexts. A layer can refer to the layer definition in the resource repository, and it can also refer to the map layer. For the purposes of the Web Tier, a *layer* refers to a map layer, and a *layer definition* refers to the layer definition in the resource repository.

Basic Layer Properties

A map contains one or more layers (`MgLayer` objects) that are rendered to create a composite image. Each layer has properties that determine how it displays in the map and map legend. Some of the properties are:

- Layer name: A unique identifier
- Legend label: The label for the layer as it appears in the map legend.
- Visibility: whether the layer should be displayed in the map. Note that actual visibility is dependent on more than just the visibility setting for a layer. See Layer Visibility (page 24) for further details.
- Selectable: Whether features in the layer are selectable. This only applies to layers containing feature data.

The `MgMap::GetLayers()` method returns an `MgLayerCollection` object that contains all the layers in the map. The `MgLayerCollection::GetItem()` method returns an individual `MgLayer` object, by either index number in the collection or layer name.

Layers in the collection are sorted by drawing order, with the top layers at the beginning of the collection. For example, using PHP syntax, if `$layers` is a collection containing the layers in a map, then `$layers->GetItem(0)` returns the top-most layer.

Layer Groups

Layers can be optionally grouped into layer groups. Layers in the same group are displayed together in the legend.

The visibility for all layers in a group can be set at the group level. If the group visibility is turned off then none of the layers in the group will be visible, regardless of their individual visibility settings. If the group visibility is turned on, then individual layers within the group can be made visible or not visible separately.

Layer groups can be nested so a group can contain other groups. This provides a finer level of control for handling layer visibility or legend groups.

The `MgMap::GetLayerGroups()` method returns an `MgLayerGroupCollection` object that contains all the layer groups in the map.

Each layer group in a map must have a unique name, even if it is nested within another group.

Base Layer Groups

The AJAX viewer can use *base layer groups* to optimize image rendering times. Layers in a base layer group are rendered together to generate a single raster image that can be sent to the viewer. The image is divided into tiles so only the required tiles need to be rendered and sent, instead of the entire image. Tiles are cached on the server; if a tile already exists in the cache it does not need to be rendered before being sent.

Each base layer group has a series of pre-defined scales that are used for rendering. When a request is made to view a portion of the map at a given scale, the AJAX viewer renders the tiles at the pre-defined scale that is closest to the requested map view scale.

Layers within a base layer group are rendered together. Visibility settings for individual layers are ignored and the visibility setting for the group is used instead.

Layers above the base layers will generally be vector layers with transparent backgrounds. This makes the images small and relatively quick to load in the viewer.

You may have more than one base layer group. Lower layers will be hidden by higher layers unless the higher layers have transparent areas or have their visibility turned off.

NOTE A layer can only belong to one group at a time. It cannot be part of both a base layer group and a regular group.

Layer Style

The data source information and style information for a layer controls how the layer looks when it displayed on a map. This is stored in the layer definition in the repository. To change any of the data source or style information, modify the layer definition.

Layer definitions can be modified using Autodesk MapGuide Studio. They can also be created and modified dynamically using the Web Extensions API. See Modifying Maps and Layers (page 51) for details.

Layer Visibility

Whether a layer is visible in a given map depends on three criteria:

- The visibility setting for the layer
- The visibility settings for any groups that contain the layer
- The map view scale and the layer definition for that view scale

In order for a layer to be visible, its layer visibility must be on, the visibility for any group containing the layer must be on, and the layer must have a style setting defined for the current map view scale.

Example: Actual Visibility

For example, assume that there is a layer named Roads that is part of the layer group Transportation. The layer has view style defined for the scale ranges 0–10000 and 10000–24000.

The following table shows some possible settings of the various visibility and view scale settings, and their effect on the actual layer visibility.

Layer Visibility	Group Visibility	View Scale	Actual Visibility
On	On	10000	On
On	On	25000	Off

Layer Visibility	Group Visibility	View Scale	Actual Visibility
On	Off	10000	Off
Off	On	10000	Off

Refresh and Zoom

The default scale and center point of a map are set as part of the map definition. When a map first displays, the default values are used. As users change the view, the changes are not saved back to the map definition, but are used temporarily as part of the session.

JavaScript API calls on the viewer refresh the map and set the view scale and map center. See MapGuide Viewer API (page 11) for details about ways to call JavaScript functions.

To refresh a map, execute the following JavaScript function:

```
parent.Refresh();
```

To change the center point and map view scale, execute the following JavaScript function:

```
ZoomToView(XCoordinate, YCoordinate, MapScale, Refresh);
```

Example

The sample application contains a file named `gotopoint.php` that is designed to run in the script frame. The essential parts of `gotopoint.php` are:

```
<script language="javascript">

function OnPageLoad()
{
   parent.ZoomToView(<?= $_GET['X'] ?>,
   <?= $_GET['Y'] ?>,
   <?= $_GET['Scale'] ?>, true);
}

</script>

<body onLoad="OnPageLoad()">

</body>
```

To execute `gotopoint.php` from the task frame, insert code similar to the following:

```
$xLocation = -87.7116768; // Or calculate values
$yLocation = 43.7766789973;
$mapScale = 2000;
echo "<p><a href=\"gotopoint.php?" .
  "X=$xLocation&Y=$yLocation&Scale=$mapScale\"" .
  "target=\"scriptFrame\">Click to position map</a></p>";
```

Enumerating Map Layers

Map layers are contained within an `MgMap` object. To list the layers in a map, use the `MgMap::GetLayers()` method. This returns an `MgLayerCollection` object.

To retrieve a single layer, use the `MgLayerCollection::GetItem` method with either an integer index or a layer name. The layer name is the name as defined in the map, not the name of the layer definition in the repository.

For example:

```
$layer = $layers->GetItem('Roads');
```

Example

The following example lists the layers in a map, along with an indicator of the layer visibility setting.

```
$layers = $map->GetLayers(); // Get layer collection
echo "<p>Layers:<br />";
$count = $layers->GetCount();
for ($i = 0; $i < $count; $i++)
{
  $layer = $layers->GetItem($i);
  echo $layer->GetName() . ' (' .
    ($layer->GetVisible() ? 'on' : 'off') . ')<br />';
}
echo '</p>';
```

Manipulating Layers

Modifying basic layer properties and changing layer visibility settings can be done directly using API calls. More complex manipulation requires modifying layer resources in the repository. For details, see Modifying Maps and Layers (page 51).

Changing Basic Properties

To query or change any of the basic layer properties like name, label, or group, use the `MgLayer::Get`*`Property`*`()` and `MgLayer::Set`*`Property`*`()` methods, where *`Property`* is one of the layer properties. You must save and refresh the map for the changes to take effect.

Example

The following example toggles the name of the Roads layer between Roads and Streets.

```
$awSessionId = ($_SERVER['REQUEST_METHOD'] == "POST")?
   $_POST['SESSION']: $_GET['SESSION'];
try
{
   // Initialize the Web Extensions and connect to the Server using
   // the Web Extensions session identifier stored in PHP
   // session state.

   // $configFilePath is the path to the web server configuration
   MgInitializeWebTier ($configFilePath);

   $userInfo = new MgUserInformation($awSessionId);
   $siteConnection = new MgSiteConnection();
   $siteConnection->Open($userInfo);

   $resourceService =
   $siteConnection->CreateService(MgServiceType::ResourceService);

   $map = new MgMap();
   $map->Open($resourceService, 'Sheboygan');

   $layers = $map->GetLayers();

   $roadLayer = $layers->GetItem('Roads');
   $roadLabel = $roadLayer->GetLegendLabel();
   if ($roadLabel == 'Roads')
      $newLabel = 'Streets';
   else
      $newLabel = 'Roads';

   $roadLayer->SetLegendLabel($newLabel);

   // You must save the updated map or the
   // changes will not be applied
   // Also be sure to refresh the map on page load.
   $map->Save($resourceService);

   echo '<p>Layer label has been changed.</p>';

}
catch (MgLayerNotFoundException $e)
{
   echo '<p>Layer not found</p>';
```

```
}
catch (MgObjectNotFoundException $e)
{
   echo '<p>Layer not found</p>';
}
catch (MgException $e)
{
   echo $e->GetMessage();
   echo $e->GetDetails();
}
```

Changing Visibility

To query the actual layer visibility, use the `MgLayer::IsVisible()` method. There is no method to set actual visibility because it depends on other visibility settings.

To query the visibility setting for a layer, use the `MgLayer::GetVisible()` method. To change the visibility setting for a layer, use the `MgLayer::SetVisible()` method.

To query the visibility setting for a layer group, use the `MgGroup::GetVisible()` method. To change the visibility setting for a layer group, use the `MgGroup::SetVisible()` method.

To change the layer visibility for a given view scale, modify the layer resource and save it back to the repository. See Modifying Maps and Layers (page 51) for details.

The following example turns on the visibility for the Roads layer. Changing Basic Properties (page 27).

```
$layers = $map->GetLayers();
$roadsLayer = $layers->GetItem('Roads');
$roadsLayer->SetVisible(True);
```

NOTE Changing the visibility will have no effect until the map is saved and refreshed.

Working With Feature Data

3

In this chapter

- Overview of Features
- Querying Feature Data
- Active Selections

Overview of Features

Understanding features is fundamental to being able to use the MapGuide Web API. Nearly every application will need to interact with feature data in one form or another.

Features are map objects representing items like roads (polylines), lakes (polygons), or locations (points).

A *feature source* is a resource that contains a set of related features, stored in a file or database. Some common feature source types are SDF files, SHP files, or data in a spatial database.

For example, you may have a feature source that contains data for roads. Feature sources can be stored in the library repository or in a session repository. A feature source identifier describes a complete path in the repository. For example,

```
Library://Samples/Sheboygan/Data/RoadCenterLines.FeatureSource
```

Within a single feature source there may be one or more *feature classes*. A feature class describes a subset of the features in the feature source. In many cases, there is one feature class for each feature source. For example, there may be a Roads feature class in the RoadCenterLines feature source.

A feature class contains one or more features. Each feature has a geometry that defines the spatial representation of the feature. Features will also generally have one or more properties that provide additional information. For example, a feature class containing road data may have properties for the road name and the number of lanes. Feature properties can be of different types, like strings, integers, and floating point numbers. Possible types are defined in the class `MgPropertyType`.

In some cases, a feature property will be another feature. For example, a Roads feature might have a Sidewalk feature as one of its properties.

A map layer may contain the features from a feature class. The features are rendered using the feature geometry.

The Web API Feature Service provides functions for querying and updating feature data.

Querying Feature Data

In order to work with feature data, you must first select the features you are interested in. This can be done with the Viewer or through Web API calls.

Feature Readers

A *feature reader*, represented by an `MgFeatureReader` object, is used to iterate through a list of features. Typically, the feature reader is created by selecting features from a feature source.

To create a feature reader, use the `MgFeatureService::SelectFeatures()` method. See Selecting with the Web API (page 33) for details about selection.

To process the features in a feature reader, use the `MgFeatureReader::ReadNext()` method. You must call this method before being able to read the first feature. Continue calling the method to process the rest of the features.

The `MgFeatureReader::GetPropertyCount()` method returns the number of properties available for the current feature. When you know the name and type of the feature property, call one of the `MgFeatureReader::Get`*`PropertyType`*`()` methods (where *`PropertyType`* represents one of the available types) to retrieve the value. Otherwise, call `MgFeatureReader::GetPropertyName()` and `MgFeatureReader::GetPropertyType()` before retrieving the value.

Selecting with the Web API

Selections can be created programatically with the Web API. This is done by querying data in a feature source, creating a feature reader that contains the features, then converting the feature reader to a selection (`MgSelection` object).

To create a feature reader, apply a selection filter to a feature class in the feature source. A selection filter can be a *basic filter*, a *spatial filter*, or a combination of the two. The filter is stored in an `MgFeatureQueryOptions` object.

Basic filters are used to select features based on the values of feature properties. For example, you could use a basic filter to select all roads that have four or more lanes.

Spatial filters are used to select features based on their geometry. For example, you could use a spatial filter to select all roads that intersect a certain area.

Basic Filters

Basic filters perform logical tests of feature properties. You can construct complex queries by combining expressions. Expressions use the comparison operators below:

Operator	Meaning
=	Equality
<>	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
LIKE	Used for string comparisons. The "%" wildcard represents any sequence of 0 or more characters. The "_" wildcard represents any single character. For example, "LIKE Schmitt%" will search for any names beginning with "Schmitt".

The comparison operators can be used with numeric or string properties, except for the LIKE operator, which can only be used with string properties.

Combine or modify expressions with the standard boolean operators AND, OR, and NOT.

Examples

These examples assume that the feature class you are querying has an integer property named `year` and a string property named `owner`. To select all features newer than 2001, create a filter like this:

```
$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter('year > 2001');
```

To select all features built between 2001 and 2004, create a filter like this:

```
$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter('year >= 2001 and year <= 2004');
```

To select all features owned by Davis or Davies, create a filter like this:

```
$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter("owner LIKE 'Davi%s'");
```

Spatial Filters

With spatial filters, you can do comparisons using geometric properties. For example, you can select all features that are inside an area on the map, or that intersect an area.

There are two ways of using spatial filters:

- Create a separate spatial filter to apply to the feature source, using the `MgFeatureQueryOptions::SetSpatialFilter()` method.
- Include spatial properties in a basic filter created with the `MgFeatureQueryOptions::SetFilter()` method.

The `MgFeatureQueryOptions::SetSpatialFilter()` method requires an `MgGeometry` object to define the geometry and a spatial operation to compare the feature property and the geometry. The spatial operations are defined in class `MgFeatureSpatialOperations`.

To include spatial properties in a basic filter, define the geometry using WKT format. Use the `GEOMFROMTEXT()` function in the basic filter, along with one of the following spatial operations:

- CONTAINS
- COVEREDBY

- CROSSES
- DISJOINT
- EQUALS
- INTERSECTS
- OVERLAPS
- TOUCHES
- WITHIN
- INSIDE

NOTE Not all spatial operations can be used on all features. It depends on the capabilities of the FDO provider that supplies the data. This restriction applies to separate spatial filters and spatial properties that are used in a basic filter.

Creating Geometry Objects From Features

You may want to use an existing feature as part of a spatial query. To retrieve the feature's geometry and convert it into an appropriate format for a query, perform the following steps:

- Create a query that will select the feature.
- Query the feature class containing the feature using the `MgFeatureService::SelectFeatures()` method.
- Obtain the feature from the query using the `MgFeatureReader::ReadNext()` method.
- Get the geometry data from the feature using the `MgFeatureReader::GetGeometry()` method. This data is in AGF binary format.
- Convert the AGF data to an `MgGeometry` object using the `MgAgfReaderWriter::Read()` method.

For example, the following sequence creates an `MgGeometry` object representing the boundaries of District 1 in the sample data.

```
$districtQuery = new MgFeatureQueryOptions();
$districtQuery->SetFilter("Autogenerated_SDF_ID = 1");
$districtResId = new MgResourceIdentifier(
"Library://Samples/Sheboygan/Data/VotingDistricts.FeatureSource");

$featureReader = $featureService->SelectFeatures($districtResId,
   "VotingDistricts", $districtQuery);
$featureReader->ReadNext();
$districtGeometryData = $featureReader->GetGeometry('Data');
$agfReaderWriter = new MgAgfReaderWriter();
$districtGeometry = $agfReaderWriter->Read($districtGeometryData);
```

To convert an `MgGeometry` object into its WKT representation, use the `MgWktReaderWriter::Write()` method, as in the following example:

```
$wktReaderWriter = new MgWktReaderWriter();
$districtWkt = $wktReaderWriter->Write($districtGeometry);
```

Examples

The following examples assume that `$testArea` is an `MgGeometry` object defining a polygon, and `$testAreaWkt` is a WKT description of the polygon.

To create a filter to find all properties owned by Schmitt in the area, use either of the following sequences:

```
$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter("RNAME LIKE 'Schmitt%'");
$queryOptions->SetSpatialFilter('SHPGEOM', $testArea,
  MgFeatureSpatialOperations::Inside);
```

```
$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter("RNAME LIKE 'Schmitt%'
  AND SHPGEOM inside GEOMFROMTEXT('$testAreaWkt')");
```

Example: Listing Selected Features

The following example creates a selection, then lists properties from the selected features.

It selects parcels within the boundaries of District 1 that are owned by Schmitt. This requires a spatial filter and a basic filter.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>

   <head>
      <title>Selecting properties</title>
   </head>

   <body>

      <h1>Selection</h1>

         <?php
         include 'AppConstants.php';

         $mgSessionId = ($_SERVER['REQUEST_METHOD'] == "POST")?
            $_POST['SESSION']: $_GET['SESSION'];
         $mapName = ($_SERVER['REQUEST_METHOD'] == "POST")?
            $_POST['MAPNAME']: $_GET['MAPNAME'];

         try
         {

            // Initialize the Web Extensions and connect to
            // the Server using the Web Extensions session
            // identifier stored in PHP session state.

            MgInitializeWebTier ($configFilePath);

            $userInfo = new MgUserInformation($mgSessionId);
            $siteConnection = new MgSiteConnection();
            $siteConnection->Open($userInfo);

            $resourceService = $siteConnection->
               CreateService(MgServiceType::ResourceService);
            $featureService = $siteConnection->
               CreateService(MgServiceType::FeatureService);

            $map = new MgMap();
            $map->Open($resourceService, $mapName);

            // Get the geometry for the boundaries of District 1
```

```
            $districtQuery = new MgFeatureQueryOptions();
           $districtQuery->SetFilter("Autogenerated_SDF_ID = 1");
            $districtResId = new MgResourceIdentifier(
"Library://Samples/Sheboygan/Data/VotingDistricts.FeatureSource");
            $featureReader = $featureService->
               SelectFeatures($districtResId, "VotingDistricts",
               $districtQuery);
            $featureReader->ReadNext();
            $districtGeometryData = $featureReader->
               GetGeometry('Data');

            // Convert the AGF binary data to MgGeometry.

            $agfReaderWriter = new MgAgfReaderWriter();
            $districtGeometry = $agfReaderWriter->
               Read($districtGeometryData);

            // Create a filter to select the desired features.
            // Combine a basic filter and a spatial filter.

            $queryOptions = new MgFeatureQueryOptions();
            $queryOptions->SetFilter("RNAME LIKE 'Schmitt%'");
            $queryOptions->SetSpatialFilter('SHPGEOM',
               $districtGeometry,
               MgFeatureSpatialOperations::Inside);

            // Select the features.

            $featureResId = new MgResourceIdentifier(
"Library://Samples/Sheboygan/Data/Parcels.FeatureSource");
            $featureReader = $featureService->
               SelectFeatures($featureResId, "Parcels",
               $queryOptions);

            // For each selected feature, display the address.

            echo '<p>Properties owned by Schmitt ';
            echo 'in District 1</p><p>';

            while ($featureReader->ReadNext())
            {
               $val = $featureReader->GetString('RPROPAD');
               echo $val . '<br />';
```

```
            }
            echo '</p>';
        }
        catch (MgException $e)
        {
            echo $e->GetMessage();
            echo $e->GetDetails();
        }
        ?>
    </body>
</html>
```

Active Selections

A map may have an active selection, which is a list of features on the map that have been selected. Features in the active selection are highlighted in the Viewer. The active selection is part of the run-time map state, and is not stored with the map resource in the repository.

The most direct method for creating an active selection is to use the interactive selection tools in the Viewer. Applications can also create selections using the Web API and apply them to a user's view of the map.

NOTE There is a fundamental difference in how the two Viewers manage selections. In the DWF Viewer, selection is handled entirely by the Viewer. This means that the Web server must request the selection information before it can use it.

In the AJAX Viewer, any changes to the active selection require re-generation of the map image. Because of this, the Web server keeps information about the selection.

Selecting with the Viewer

In order for a feature to be selectable using the Viewer, the following criteria must be met:

- The layer containing the feature must be visible at the current map view scale.
- The selectable property for the layer must be true. Change this property in the web layout or with the `MgLayer::SetSelectable()` method.

There are different selection tools available in the Viewer. They can be enabled or disabled as part of the web layout. Each tool allows a user to select one or more features on the map.

Working With the Active Selection

For the AJAX Viewer, whenever a selection is changed by the Viewer, the selection information is sent to the web server so the map can be re-generated. For the DWF Viewer, the selection information is managed by the Viewer, so the Viewer must pass the selection information to the web server before it can be used.

If you are writing an application that will be used by both Viewers, you can use the DWF method, which will result in some additional overhead for the AJAX Viewer. Alternatively, you can write different code for each Viewer.

To retrieve and manipulate the active selection for a map (AJAX Viewer):

1. Create an `MgSelection` object for the map. Initialize it to the active selection.
2. Retrieve selected layers from the `MgSelection` object.
3. For each layer, retrieve selected feature classes. There will normally be one feature class for the layer, so you can use the `MgSelection::GetClass()` method instead of the `MgSelection::GetClasses()` method.
4. Call `MgSelection::GenerateFilter()` to create a selection filter that contains the selected features in the class.
5. Call `MgFeatureService::SelectFeatures()` to create an `MgFeatureReader` object for the selected features.
6. Process the `MgFeatureReader` object, retrieving each selected feature.

The procedure for the DWF Viewer is similar, but the Viewer must send the selection information as part of the HTTP request.

To retrieve and manipulate the active selection for a map (DWF Viewer):

1. Get the current selection using the Viewer API call `GetSelectionXML()`.

2 Pass this to the Web server as part of an HTTP request. The simplest method for this is to use the `Submit()` method of the form frame. This loads a page and passes the parameters using an HTTP POST.

3 In the page, create an `MgSelection` object for the map.

4 Initialize the `MgSelection` object with the list of features passed to the page.

5 Retrieve selected layers from the `MgSelection` object.

6 For each layer, retrieve selected feature classes. There will normally be one feature class for the layer, so you can use the `MgSelection::GetClass()` method instead of the `MgSelection::GetClasses()` method.

7 Call `MgSelection::GenerateFilter()` to create a selection filter that contains the selected features in the class.

8 Call `MgFeatureService::SelectFeatures()` to create an `MgFeatureReader` object for the selected features.

9 Process the `MgFeatureReader` object, retrieving each selected feature.

Example: Listing Selected Parcels (AJAX Viewer)

To run the following example, view a map created with the sample data. Using one of the selection tools, select one or more parcels.

NOTE The sample code below is not complete. You must perform standard initialization steps and exception trapping.

The example creates a Resource Service connection to open the map and a Feature Service connection to read the feature information. In addition, you will require an `MgFeatureQueryOptions` object for creating the list of features.

The `MgSelection` object contains selection information for a given map. It can either contain the current selection or a selection created from XML data. For this example, create the object with the current map selection.

Get the list of layers and iterate through them. When the `$layers` collection is empty, there are no features selected.

```
// Initialize the Web Extensions and connect to the Server using
// the Web Extensions session identifier stored in PHP
// session state.

MgInitializeWebTier ($configFilePath);

$userInfo = new MgUserInformation($mgSessionId);
$siteConnection = new MgSiteConnection();
$siteConnection->Open($userInfo);

$resourceService = $siteConnection->CreateService(
   MgServiceType::ResourceService);
$featureService = $siteConnection->CreateService(
   MgServiceType::FeatureService);
$queryOptions = new MgFeatureQueryOptions();

$map = new MgMap();
$map->Open($resourceService, $mapName);

// ----- Beginning of AJAX-specific code ----------
// Create the selection object by retrieving the current
// selection from the map.

$selection = new MgSelection($map);
$selection->Open($resourceService, $mapName);
// ----- End of AJAX-specific code ----------------

$layers = $selection->GetLayers();

if ($layers)
{
   for ($i = 0; $i < $layers->GetCount(); $i++)
   {

      // Only check selected features in the Parcels layer.

      $layer = $layers->GetItem($i);
      if ($layer && $layer->GetName() == 'Parcels')
      {

         // Create a filter containing the IDs of the selected
         // features on this layer
```

```
            $layerClassName = $layer->GetFeatureClassName();
            $selectionString = $selection->GenerateFilter(
               $layer, $layerClassName);

            // Get the feature resource for the selected layer
            $layerFeatureId = $layer->GetFeatureSourceId();
            $layerFeatureResource = new
               MgResourceIdentifier($layerFeatureId);

            // Apply the filter to the feature resource for the
            // selected layer. This returns
            // an MgFeatureReader of all the selected features.

            $queryOptions->SetFilter($selectionString);
            $featureReader = $featureService->SelectFeatures(
             $layerFeatureResource, $layerClassName, $queryOptions);

            // Process each item in the MgFeatureReader, displaying
the
            // owner name and address

            while ($featureReader->ReadNext())
            {
               $val = $featureReader->GetString('NAME') . ', ' .
                  $featureReader->GetString('RPROPAD');
               echo $val . '<br />';
            }
         }
      }
   }
else
   echo 'No selected layers';
```

Example: Listing Selected Parcels (DWF Viewer)

The steps for listing the selected parcels for the DWF Viewer are nearly the same as for the AJAX Viewer. The major difference is you must pass the selection information from the Viewer to your page.

One method to do this is to create a JavaScript function, then call this function from the Viewer using an Invoke Script command. In an HTML page that includes an embedded Viewer, add the following JavaScript function:

```
function listSelected()
{
   xmlSel = ViewerFrame.mapFrame.GetSelectionXML();
   params = new Array("SESSION",
      ViewerFrame.mapFrame.GetSessionId(), "SELECTION", xmlSel);
   ViewerFrame.formFrame.Submit("../devguide/dwfselection.php",
      params, "taskPaneFrame");
}
```

In your web layout, create a new command and add it to the task list. Set the command type to Invoke Script. Set the script to invoke to

```
parent.listSelected();
```

Create a page named `dwfselection.php` in the devguide directory. This can be exactly the same as Example: Setting the Active Selection (page 45), with one modification. Replace the AJAX-specific code with the following:

```
$selection = new MgSelection($map, $_POST['SELECTION']);
```

When you select your command from the task list, it runs the custom JavaScript function, which passes the selection XML to `dwfselection.php` and loads it into the task pane.

Setting the Active Selection With the Web API

To set the run-time map selection using a query, perform the following steps:

- Create a selection as described in Selecting with the Web API (page 33). This creates a feature reader containing the selected features.
- Create an `MgSelection` object to hold the features in the feature reader.
- Send the selection to the viewer, along with a call to the Viewer API function `SetSelectionXML()`.

Example: Setting the Active Selection

The following example combines the pieces needed to create a selection using the Web API and pass it back to the Viewer where it becomes the active selection for the map. It is an extension of the example shown in Example: Listing Selected Features (page 37).

The PHP code in this example creates the selection XML. Following that is a JavaScript function that calls the `SetSelectionXML()` function with the selection. This function is executed when the page loads.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>

   <head>
      <title>Server-side Selection</title>
      <meta content="text/html; charset=utf-8"
       http-equiv="Content-Type">
      <meta http-equiv="content-script-type"
       content="text/javascript" />
      <meta http-equiv="content-style-type" content="text/css" />
      <link href="styles/globalStyles.css" rel="stylesheet"
       type="text/css">
      <link href="styles/alphaStyles.css" rel="stylesheet"
       type="text/css">
   </head>

   <body class="AppFrame" onLoad="OnPageLoad()">

      <h1 class="AppHeading">Example</h1>

         <?php
         include 'AppConstants.php';

         $mgSessionId = ($_SERVER['REQUEST_METHOD'] == "POST")
            ? $_POST['SESSION']: $_GET['SESSION'];
         $mapName = ($_SERVER['REQUEST_METHOD'] == "POST")
            ? $_POST['MAPNAME']: $_GET['MAPNAME'];

         try
         {

            // Initialize the Web Extensions and connect to
            // the Server using
            // the Web Extensions session identifier stored
            // in PHP session state.

            MgInitializeWebTier ($configFilePath);

            $userInfo = new MgUserInformation($mgSessionId);
            $siteConnection = new MgSiteConnection();
            $siteConnection->Open($userInfo);
```

```
$resourceService = $siteConnection->CreateService(
   MgServiceType::ResourceService);
$featureService = $siteConnection->CreateService(
   MgServiceType::FeatureService);

$map = new MgMap();
$map->Open($resourceService, $mapName);

// Get the geometry for the boundaries of District 1

$districtQuery = new MgFeatureQueryOptions();
$districtQuery->SetFilter("Autogenerated_SDF_ID = 1");
$districtResId = new MgResourceIdentifier(
"Library://Samples/Sheboygan/Data/VotingDistricts.FeatureSource");
$featureReader = $featureService->SelectFeatures(
  $districtResId, "VotingDistricts", $districtQuery);
$featureReader->ReadNext();
$districtGeometryData = $featureReader->GetGeometry(
   'Data');

// Convert the AGF binary data to MgGeometry.

$agfReaderWriter = new MgAgfReaderWriter();
$districtGeometry = $agfReaderWriter->Read(
   $districtGeometryData);

// Create a filter to select the desired features.
// Combine a basic filter and a spatial filter.

$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter("RNAME LIKE 'Schmitt%'");
$queryOptions->SetSpatialFilter('SHPGEOM',
   $districtGeometry,
   MgFeatureSpatialOperations::Inside);

// Get the features from the feature source,
// turn it into a selection, then save as XML.

$featureResId = new MgResourceIdentifier(
"Library://Samples/Sheboygan/Data/Parcels.FeatureSource");
$featureReader = $featureService->SelectFeatures(
   $featureResId, "Parcels", $queryOptions);
```

```
            $layer = $map->GetLayers()->GetItem('Parcels');
            $selection = new MgSelection($map);
            $selection->AddFeatures($layer, $featureReader, 0);
            $selectionXml = $selection->ToXml();

            echo 'Setting selection...';
         }
         catch (MgException $e)
         {
            echo $e->GetMessage();
            echo $e->GetDetails();
         }
         ?>

   </body>

   <script language="javascript">

<!-- Emit this function and assocate it with the onLoad event -->
<!-- for the page so that it gets executed when this page loads-->
<!-- in the browser. The function calls the SetSelectionXML   -->
<!-- method on the Viewer Frame, which updates the current    -->
<!-- selection on the viewer and the server.                  -->

      function OnPageLoad()
      {
         selectionXml = '<?php echo $selectionXml; ?>';
         parent.parent.SetSelectionXML(selectionXml);
      }

   </script>

</html>
```

Modifying Maps and Layers

4

In this chapter

- Introduction
- Adding An Existing Layer To A Map
- Creating Layers By Modifying XML
- Another Way To Create Layers
- Adding Layers To A Map
- Making Changes Permanent

Introduction

This chapter describes how to modify maps and layers.

Adding An Existing Layer To A Map

If the layer already exists in the resource repository, add it to the map by getting the map's layer collection and then adding the layer to that collection.

```
$layerCollection = $map->GetLayers();
$layerCollection->Add($layer);
```

By default, newly added layers are added to the bottom of the drawing order, so they may be obscured by other layers. If you want to specify where the layer appears in the drawing order, use the `$layerCollection->Insert()` method. For an example, see Adding Layers To A Map (page 63).

NOTE In the MapGuide API, getting a collection returns a reference to the collection. So adding the layer to the layer collection immediately updates the map.

Creating Layers By Modifying XML

The easiest way to programmatically create new layers is to

1. Build a prototype layer through the Studio UI. To make the scripting simpler, this layer should have as many of the correct settings as can be determined in advance.
2. Use Studio's Save as Xml command to save the layer as an XML file.
3. Have the script load the XML file and then use the DOM (Document Object Model) to change the necessary XML elements.
4. Add the modified layer to the map.

The XML schema for layer definitions is defined by the `LayerDefinition-`*`version`*`.xsd` schema which is documented in the *MapGuide Web API Reference*. This schema closely parallels the UI in the Studio's Layer Editor, as described in the Studio Help.

This example

- loads a layer that has been created through Studio
- uses the DOM to changes the filter and its associated legend label

You can use the DOM to modify *any* layers, including ones that already exist in the map, not just new layers that you are adding to the map. You can also use the DOM to modify other resources; the XML schemas are described in the *MapGuide Web API Reference*.

```
// (initialization etc. not shown here)
// Open the map
 $map = new MgMap();
 $map->Open($resourceService, 'Sheboygan');
 // ------------------------------------------------//
 // Load a layer from XML, and use the DOM to change it
 // Load the prototype layer definition into
 // a PHP DOM object.
 $domDocument =
   DOMDocument::load('RecentlyBuilt.LayerDefinition');
 if ($domDocument == NULL)
 {
     echo "The layer definition
       'RecentlyBuilt.LayerDefinition' could not be
       found.<BR>\n";
     return;
 }
 // Change the filter
 $xpath = new DOMXPath($domDocument);
 $query = '//AreaRule/Filter';
 // Get a list of all the <AreaRule><Filter> elements in
 // the XML.
 $nodes = $xpath->query($query);
 // Find the correct node and change it
 foreach ($nodes as $node )
 {
     if ($node->nodeValue == 'YRBUILT > 1950')
     {
         $node->nodeValue = 'YRBUILT > 1980';
     }
 }
 // Change the legend label
 $query = '//LegendLabel';
 // Get a list of all the <LegendLabel> elements in the
 // XML.
 $nodes = $xpath->query($query);
 // Find the correct node and change it
 foreach ($nodes as $node )
 {
     if ($node->nodeValue == 'Built after 1950')
     {
         $node->nodeValue = 'Built after 1980';
     }
```

```
    }
// ...
```

The page then goes on to save the XML to a resource and loads that resource into the map, as described inAdding Layers To A Map (page 63).

NOTE For another example of this approach, see the `findparcelfunctions.php` and `findaddressfunctions.php` scripts in the sample application.

If you wish to modify an existing layer that is visible in other users' maps, without affecting those maps:

1. Copy the layer to the user's session repository.
2. Modify the layer and save it back to the session repository.
3. Change the user's map to refer to the modified layer.

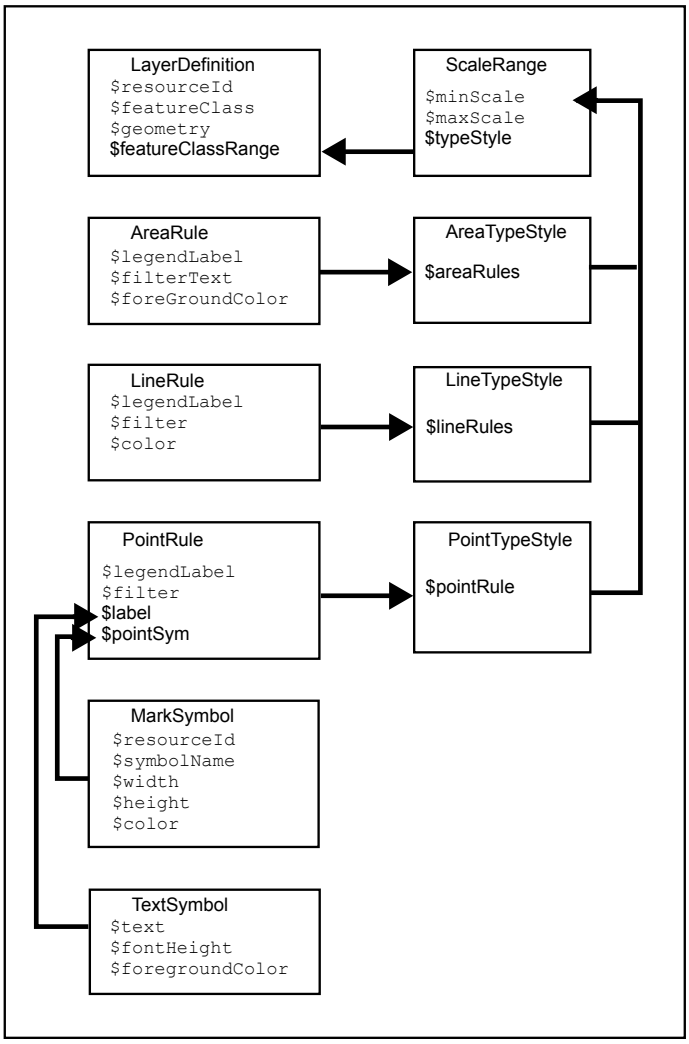
See Adding Layers To A Map (page 63).

Another Way To Create Layers

The method described in the previous section is easy to use, but requires a layer definition be created first through the Studio UI. An alternative approach is to use the methods defined in

`WebServerExtensions\www\mapviewerphp\layerdefinitionfactory.php`.

This file contains several functions, which can be used to build up a layer definition. The parameters of these functions enable you to set the most commonly used settings. (If you need to change other settings, you will have to either use the Studio UI, or modify the XML of the layer definition.)



Function	Parameter	Description
`CreateLayerDefinition()`	`$resourceId`	The repository path of the feature source for the layer. For example: `Library://Samples/Sheboygan/Data/Parcels.FeatureSource`. Equivalent to the Data resource used in this layer field in Studio's layer editor.

Function	Parameter	Description
	$featureClass	The feature class to use. For example, SHP_Schema:Parcels. Equivalent to the Feature class field in Studio's layer editor.
	$geometry	The geometry to use from the feature class. For example, SHPGEOM. Equivalent to the Geometry field in Studio's layer editor.
	$featureClass-Range	A scale range created by filling in a scale range template (ScaleRange.templ).
CreateScaleRange()	$minScale	The minimum scale range to which this rule applies. Equivalent to the From field in Studio's layer editor.
	$maxScale	The maximum scale range to which this rule applies. Equivalent to the To field in Studio's layer editor.
	$typeStyle	A type style created by using CreateAreaTypeStyle(),CreateLineTypeStyle() or CreatePointTypeStyle().
CreateAreaTypeStyle()	$areaRules	One or more area rules, created by CreateAreaRule.
CreateAreaRule()	$legendLabel	The text for the label shown beside this rule in the legend. Equivalent to the Geometry field in Studio's layer editor.
	$filterText	The filter expression that determines which features match this rule. For example, SQFT >= 1 AND SQFT < 800. Equivalent to the Condition field in Studio's layer editor.

Function	Parameter	Description
	`$foreGroundColor`	The color to be applied to areas that match this rule. Equivalent to the Foreground color field in Studio's layer editor.
`CreateTextSymbol()`	`$text`	The string for the text.
	`$fontHeight`	The height for the font.
	`$foregroundColor`	The foreground color.
`CreatePointTypeStyle()`	`$pointRule`	One or more point rules, created by `CreatePointRule()`.
`CreatePointRule()`	`$legendLabel`	The label shown beside this rule in the legend. Equivalent to the Legend label field in Studio's layer editor.
	`$filter`	The filter expression that determines which features match this rule. Equivalent to the Condition field in Studio's layer editor.
	`$label`	The text symbol, created by `CreateTextSymbol()`.
	`$pointSym`	A mark symbol created by `CreateMarkSymbol()`.
`CreateMarkSymbol()`	`$resourceId`	The resource ID of the symbol used to mark each point. For example, `library://Samples/Sheboygan/Symbols/BasicSymbols.SymbolLibrary`. Equivalent to the Location field in the Select a symbol from a Symbol Library dialog in Studio's layer editor.
	`$symbolName`	The name of the desired symbol in the symbol library.

Function	Parameter	Description
	`$width`	The width of the symbol (in points). Equivalent to the Width field in the Style Point dialog in Studio's layer editor.
	`$height`	The height of the symbol (in points). Equivalent to the Height field in the Style Point dialog in Studio's layer editor.
	`$color`	The color for the symbol. Equivalent to the Foreground color field in the Style Point dialog in Studio's layer editor.
`CreateLineTypeStyle()`	`$lineRules`	One or more line rules, created by `CreateLineRule()`.
`CreateLineRule()`	`$color`	The color to be applied to lines that match this rule. Equivalent to the Color field in Studio's layer editor.
	`$legendLabel`	The label shown beside this rule in the legend. Equivalent to the Legend Label field in Studio's layer editor.
	`$filter`	The filter expression that determines which features match this rule. Equivalent to the Condition field in Studio's layer editor.

For more information on these settings, see the Studio Help.

Example - Creating A Layer That Uses Area Rules

This example shows how to create a new layer using the factory. This layer uses three area rules to theme parcels by their square footage.

```
   // ...
 /------------------------------------------------//
$factory = new LayerDefinitionFactory();
/// Create three area rules for three different
// scale ranges.
$areaRule1 = $factory->CreateAreaRule(    '1 to 800',
  'SQFT &gt;= 1 AND SQFT &lt; 800',     'FFFFFF00');
$areaRule2 = $factory->CreateAreaRule( '800 to 1600',
  'SQFT &gt;= 800 AND SQFT &lt; 1600',  'FFFFBF20');
$areaRule3 = $factory->CreateAreaRule('1600 to 2400',
  'SQFT &gt;= 1600 AND SQFT &lt; 2400', 'FFFF8040');
// Create an area type style.
$areaTypeStyle = $factory->CreateAreaTypeStyle(
  $areaRule1 . $areaRule2 . $areaRule3);
// Create a scale range.
$minScale = '0';
$maxScale = '1000000000000';
$areaScaleRange = $factory->CreateScaleRange(
  $minScale, $maxScale, $areaTypeStyle);
// Create the layer definiton.
$featureClass = 'Library://Samples/Sheboygan/Data/'
.  'Parcels.FeatureSource';
$featureName = 'SHP_Schema:Parcels';
$geometry = 'SHPGEOM';
$layerDefinition = $factory->CreateLayerDefinition(
  $featureClass, $featureName, $geometry,
  $areaScaleRange);
//------------------------------------------------//
// ...
```

The script then saves the XML to a resource and loads that resource into the map. See Adding Layers To A Map (page 63).

Example - Using Line Rules

Creating line-based rules is very similar.

```
// ...
//------------------------------------------------//
$factory = new LayerDefinitionFactory();

// Create a line rule.
$legendLabel = '';
$filter = '';
$color = 'FF0000FF';
$lineRule = $factory->CreateLineRule(
  $legendLabel, $filter, $color);
// Create a line type style.
$lineTypeStyle = $factory->
  CreateLineTypeStyle($lineRule);

// Create a scale range.
$minScale = '0';
$maxScale = '1000000000000';
$lineScaleRange = $factory->
  CreateScaleRange($minScale, $maxScale,
  $lineTypeStyle);
// Create the layer definiton.
$featureClass = 'Library://Samples/Sheboygan/Data/'
  . 'HydrographicLines.FeatureSource';
$featureName = 'SHP_Schema:HydrographicLines';
$geometry = 'SHPGEOM';
$layerDefinition = $factory->
  CreateLayerDefinition($featureClass, $featureName,
  $geometry, $lineScaleRange);

//------------------------------------------------//
// ...
```

Example - Using Point Rules

To create point-based rules, three methods are used.

```
    // ...
    //-----------------------------------------------------//
    $factory = new LayerDefinitionFactory();
    // Create a mark symbol
    $resourceId = 'Library://Samples/Sheboygan/Symbols/BasicSym
bols.SymbolLibrary';
    $symbolName = 'PushPin';
    $width = '24';  // points
    $height = '24'; // points
    $color = 'FFFF0000';
    $markSymbol = $factory->CreateMarkSymbol($resourceId, $symbol
Name, $width, $height, $color);

    // Create a text symbol
    $text = "ID";
    $fontHeight="12";
    $foregroundColor = 'FF000000';
    $textSymbol = $factory->CreateTextSymbol($text,
      $fontHeight, $foregroundColor);
    // Create a point rule.
    $legendLabel = 'trees';
    $filter = '';
    $pointRule = $factory->CreatePointRule($legendLabel,
      $filter, $textSymbol, $markSymbol);

    // Create a point type style.
    $pointTypeStyle = $factory->
      CreatepointTypeStyle($pointRule);

    // Create a scale range.
    $minScale = '0';
    $maxScale = '1000000000000';
    $pointScaleRange = $factory->CreateScaleRange($minScale,
      $maxScale, $pointTypeStyle);
    // Create the layer definiton.
    $featureClass = 'Library://Tests/Trees.FeatureSource';
    $featureName = 'Default:Trees';
    $geometry = 'Geometry';
    $layerDefinition = $factory->
      CreateLayerDefinition($featureClass, $featureName,
      $geometry, $pointScaleRange);
    //-----------------------------------------------------//
    // ...
```

Adding Layers To A Map

The preceding examples have created or modified the XML for layer definitions in memory. To add those layers to a map:

1. Save the layer definition to a resource stored in the session repository.
2. Add that resource to the map.

This function adds takes a layer's XML, creates a resource in the session repository from it, and adds that layer resource to a map.

```
<?php
//////////////////////////////////////////////////////////
function add_layer_definition_to_map($layerDefinition,
  $layerName, $layerLegendLabel, $mgSessionId,
  $resourceService, &$map)
// Adds the layer definition (XML) to the map.
// Returns the layer.
{
    include('../Common.php');
    // Validate the XML.
    $domDocument = new DOMDocument;
    $domDocument->loadXML($layerDefinition);
    if (! $domDocument->schemaValidate(
       "$schemaDirectory\LayerDefinition-1.0.0.xsd") )
    {
        echo "ERROR: The new XML document is invalid.
          <BR>\n.";
        return NULL;
    }
    // Save the new layer definition to the session
    // repository
    $byteSource = new MgByteSource($layerDefinition,
      strlen($layerDefinition));
    $byteSource->SetMimeType(MgMimeType::Xml);
    $resourceID = new MgResourceIdentifier(
       "Session:$mgSessionId//$layerName.LayerDefinition");
    $resourceService->SetResource($resourceID,
      $byteSource->GetReader(), null);
    $newLayer = add_layer_resource_to_map($resourceID,
      $resourceService, $layerName, $layerLegendLabel,
      $map);
    return $newLayer;
}
//////////////////////////////////////////////////////////
```

This function adds a layer resource to a map.

```
function add_layer_resource_to_map($layerResourceID,
  $resourceService, $layerName, $layerLegendLabel, &$map)
// Adds a layer defition (which can be stored either in the
// Library or a session repository) to the map.
// Returns the layer.
{
    $newLayer = new MgLayer($layerResourceID,
      $resourceService);
    // Add the new layer to the map's layer collection
    $newLayer->SetName($layerName);
    $newLayer->SetVisible(true);
    $newLayer->SetLegendLabel($layerLegendLabel);
    $newLayer->SetDisplayInLegend(true);
    $layerCollection = $map->GetLayers();
    if (! $layerCollection->Contains($layerName) )
    {
        // Insert the new layer at position 0 so it is at
        // the top of the drawing order
        $layerCollection->Insert(0, $newLayer);
    }
    return $newLayer;
}
//////////////////////////////////////////////////////////
?>
```

This function adds a layer to a legend's layer group.

```
function add_layer_to_group($layer, $layerGroupName,
  $layerGroupLegendLabel, &$map)
// Adds a layer to a layer group. If necessary, it creates
// the layer group.
{
    // Get the layer group
    $layerGroupCollection = $map->GetLayerGroups();
    if ($layerGroupCollection->Contains($layerGroupName))
    {
        $layerGroup =
          $layerGroupCollection->GetItem($layerGroupName);
    }
    else
    {
        // It does not exist, so create it
        $layerGroup = new MgLayerGroup($layerGroupName);
        $layerGroup->SetVisible(true);
        $layerGroup->SetDisplayInLegend(true);

          $layerGroup->SetLegendLabel($layerGroupLegendLabel);
        $layerGroupCollection->Add($layerGroup);
    }
    // Add the layer to the group
    $layer->SetGroup($layerGroup);
}
////////////////////////////////////////////////////////////
```

Making Changes Permanent

So far, all the examples in this chapter have only affected the user's runtime version of the map. No other users see those changes, and when the current user logs out those changes will be lost.

To make changes permanent, the script can save the modified layer back into the Library.

```
$byteSource = new MgByteSource($layerDefinition, strlen($layerDef
inition));
$byteSource->SetMimeType(MgMimeType::Xml);
$resourceId = new MgResourceIdentifier("Library://LayerName.Lay
erDefinition");
$resourceService->SetResource($resourceId, $byteSource->GetRead
er(), null);
```

Index

A

active selection 40–42, 44–45
  AJAX Viewer 42
  DWF Viewer 44
  in Web API 41
  setting with Web API 45
AJAX Viewer 41
  active selection 41

C

constants.php 5

D

directory, for examples 3
Document Object Model 52
DOM 52
DWF Viewer 41
  active selection 41

E

examples, preparing for 2

F

feature classes 32
feature readers 32–33
features 32, 37
  listing selected 37
formFrame, in Viewer 9

H

hellomap.php 5
HTML page, with MapGuide Viewer 14

I

Invoke Script command 13
Invoke URL command type 4

L

layerdefinitionfactory.php 55
Library repository 17

M

maparea frame, in Viewer 9
mapFrame, in Viewer 9
MapGuide Server Page 3
MgGeometry 36
  creating from feature 36
MgMap object 17
MSP processing flow 4

R

resources 16

S

sbFrame, in Viewer 9
scriptFrame, in Viewer 9
selecting 33, 40
  with the Viewer 40
  with the Web API 33
selection filters iv, 34–35
  basic 34
  spatial iv, 35
services 18
session ID 4
session management 14

T

task pane 10
taskArea, in Viewer 9
taskBar, in Viewer 9
taskFrame, in Viewer 9
taskListFrame, in Viewer 9
taskPaneFrame, in Viewer 9
tbFrame, in Viewer 9

V

Viewer API 11
Viewer commands 2
Viewer frames 8

W

web layout, defining 4
webconfig.ini 8