

Autodesk MapGuide® Enterprise 2007

Developer's Guide

The Autodesk logo is displayed in white text on a black rectangular background. The text is oriented vertically, reading from bottom to top.

June 2006

Copyright© 2006 Autodesk, Inc.

All Rights Reserved

This publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose. AUTODESK, INC., MAKES NO WARRANTY, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS, AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS-IS" BASIS. IN NO EVENT SHALL AUTODESK, INC., BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF PURCHASE OR USE OF THESE MATERIALS. THE SOLE AND EXCLUSIVE LIABILITY TO AUTODESK, INC., REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE MATERIALS DESCRIBED HEREIN.

Autodesk, Inc., reserves the right to revise and improve its products as it sees fit. This publication describes the state of the product at the time of publication, and may not reflect the product at all times in the future.

Trademarks

Autodesk, Autodesk Map, Autodesk MapGuide are registered trademarks of Autodesk, Inc., in the USA and/or other countries. DWF is a trademark of Autodesk, Inc., in the USA and/or other countries. All other brand names, product names or trademarks belong to their respective holders.

Third Party Software Program Credits

InstallShield™ 3.0. Copyright© 1997 InstallShield Software Corporation. All rights reserved.

Typefaces from the Bitstream® typeface library copyright 1992.

Typefaces from Payne Loving Trust© 1996. All rights reserved.

Printed manual and help produced with Idiom WorldServer™.

Portions Copyright© 1991-1996 Arthur D. Applegate. All rights reserved.

The Director General of the Geographic Survey Institute has issued the approval for the coordinates exchange numbered TKY2JGD for Japan

Geodetic Datum 2000, also known as technical information No H1-N0.2 of the Geographic Survey Institute, to be installed and used within this software product (Approval No.: 646 issued by GSI, April 8, 2002).

OSTN97 © Crown Copyright 1997. All rights reserved.

OSTN02 © Crown copyright 2002. All rights reserved.

OSGM02 © Crown copyright 2002, © Ordnance Survey Ireland, 2002.

Portions of this product are copyright © 2000 Earth Resource Mapping, Inc.

Portions of this software are copyright © 1996-2002 The FreeType Project (www.freetype.org). All rights reserved.

libpng is copyright © 2004 Glenn Randers-Pehrson.

LibTiff is copyright © 1988-1997 Sam Leffler and copyright © 1991-1997 Silicon Graphics, Inc.

Coordinate Systems Mapping Library © 2004 Mentor Software Inc.

Copyright © 2005, LizardTech, a division of Celartem, Inc. All rights reserved. MrSID technology protected by U.S. Patent No 5,710,835 and patents pending.

Portions copyright 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004 by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004 by Boutell.Com, Inc.

Portions relating to GD2 format copyright 1999, 2000, 2001, 2002, 2003, 2004 Philip Warner.

Portions relating to PNG copyright 1999, 2000, 2001, 2002, 2003, 2004 Greg Roelofs.

Portions relating to gdtf.c copyright 1999, 2000, 2001, 2002, 2003, 2004 John Ellson (ellson@graphviz.org).

Portions relating to gdf.c copyright 2001, 2002, 2003, 2004 John Ellson (ellson@graphviz.org).

Portions relating to JPEG and to color quantization copyright 2000, 2001, 2002, 2003, 2004, Doug Becker and copyright © 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004 Thomas G. Lane.

This software is based in part on the work of the Independent JPEG Group.

Portions relating to GIF compression copyright 1989 by Jef Poskanzer and David Rowley, with modifications for thread safety by Thomas Boutell.

Portions relating to GIF decompression copyright 1990, 1991, 1993 by David Koblas, with modifications for thread safety by Thomas Boutell.

Portions relating to WBMP copyright 2000, 2001, 2002, 2003, 2004 Maurice Szmurlo and Johan Van den Brande.

Portions relating to GIF animations copyright 2004 Jaakko Hyvätti (jaakko.hyvatti@iki.fi)

This product includes PHP, freely available from <http://www.php.net/>

GOVERNMENT USE

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 12.212 (Commercial Computer Software-Restricted Rights) and DFAR 227.7202 (Rights in Technical Data and Computer Software), as applicable.

Published By: Autodesk, Inc.

111 McInnis Parkway

San Rafael, CA 94903, USA

Contents

Chapter 1	Introduction	1
	What This Guide Covers	2
	Essential Concepts	2
	Preparing to Run the Examples	3
	Hello, Map – Displaying a Web Layout	3
	Hello, Map 2 – Adding a Custom Command	4
	Web Layouts and MapGuide Server Pages	5
	MapGuide Page Flow	6
	Example Code	7
	Running the Example	9
	How This Page Works	10
	Understanding Viewer Frames	11
	Interactions Between Frames	13
	MapGuide Viewer API	13
	Calling the Viewer API from the Task Pane	14
	Calling the Viewer API from the Script Frame	15
	Calling the Viewer API with an Invoke Script Command	16
	MapGuide Web API	16
	Embedding a Viewer in Your Own Page	17
	Session Management	17
	Resources and Repositories	19
	Library and Session	19

	Maps	20
	Understanding Services	21
Chapter 2	Interacting With Layers	23
	Overview of Layers	24
	Basic Layer Properties	24
	Layer Groups	24
	Base Layer Groups	25
	Layer Style	26
	Layer Visibility	26
	Example: Actual Visibility	26
	Refresh and Zoom	27
	Example	27
	Enumerating Map Layers	28
	Example	28
	Manipulating Layers	29
	Changing Basic Properties	29
	Example	29
	Changing Visibility	30
Chapter 3	Working With Feature Data	33
	Overview of Features	34
	Querying Feature Data	35
	Feature Readers	35
	Selecting with the Web API	35
	Basic Filters	36
	Spatial Filters	37
	Example: Listing Selected Features	39
	Active Selections	42
	Selecting with the Viewer	42
	Passing Viewer Information to the Web Server	43
	Additional Parameters to an Invoke URL Command	44
	Passing Parameters From an Invoke Script command	44
	Passing Parameters From the Task Pane Frame	45
	Working With the Active Selection	46
	Example: Listing Selected Parcels (AJAX Viewer)	47
	Example: Listing Selected Parcels (DWF Viewer)	49
	Setting the Active Selection With the Web API	50
	Example: Setting the Active Selection	50
Chapter 4	Modifying Maps and Layers	55
	Introduction	56
	Adding An Existing Layer To A Map	56
	Creating Layers By Modifying XML	56

	Another Way To Create Layers	59
	Example - Creating A Layer That Uses Area Rules	63
	Example - Using Line Rules	64
	Example - Using Point Rules	65
	Adding Layers To A Map	67
	Making Changes Permanent	70
Chapter 5	Analyzing Features	71
	Introduction	72
	Representation of Geometry	72
	Geometry Objects	72
	Comparing Geometry Objects	73
	Coordinate Systems	74
	Measuring Distance	76
	Temporary Feature Sources	76
	Inserting, Deleting, and Updating Features	79
	Creating a Buffer	80
	Example	81
Chapter 6	Digitizing and Redlining	89
	Introduction	90
	Digitizing	90
	Redlining	90
	Passing Coordinates	91
	Creating a Feature Source	92
	Creating A Layer	94
Chapter 7	Custom Output	95
	Introduction	96
	Rendering Service	97
	Mapping Service	98
	Index	101

Introduction



In this chapter

- [What This Guide Covers](#)
- [Essential Concepts](#)
- [Preparing to Run the Examples](#)
- [Hello, Map – Displaying a Web Layout](#)
- [Hello, Map 2 – Adding a Custom Command](#)
- [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 Autodesk MapGuide Enterprise 2007 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 11).

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.

Complete examples are available on the installation CD. There are two required components: the source code and a package file for creating the web layouts. The Sheboygan sample application must also be installed.

Follow the instructions in README.txt for installing and using the examples.

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.

Hello, Map – Displaying a Web Layout

A web layout describes how the map looks when it is displayed in a web browser. Using MapGuide Studio, you can create and customize the layout, changing how it looks in a browser and what features are enabled.

To display the web layout, create an HTML page that includes a Viewer . There are two Viewers, depending on the needs of your site. The DWF Viewer runs as a control within the Internet Explorer browser. It requires that users install the Autodesk DWF Viewer.

The AJAX Viewer does not require installing any additional software. It runs using most browsers, including Internet Explorer, Mozilla Firefox, and Safari.

To display a web layout using either Viewer, create a MapGuide session and pass the session identifier and the web layout identifier to the Viewer page. The following example displays a web layout using the AJAX Viewer. It performs some basic initialization and creates a session, then displays a Viewer page using the session identifier and the web layout.

```

<?php

$installDir =
    'C:\Program Files\Autodesk\MapGuideEnterprise2007\';
$extensionsDir = $installDir . 'WebServerExtensions\www\';
$viewDir = $installDir . 'mapviewerphp\';

include $viewDir . 'constants.php';

MgInitializeWebTier($extensionsDir . 'webconfig.ini');

$site = new MgSite();
$site->Open(new MgUserInformation("Anonymous", ""));

$sessionId = $site->CreateSession();
$webLayout =
    "Library://Samples/Sheboygan/Layouts/SheboyganPhp.WebLayout";

?>

<html>

<head>
<title>Simple Sample Application</title>
</head>

<body marginheight="0" marginwidth="0">
<iframe id="viewerFrame" width="100%" height="100%" frameborder=0

    scrolling="no"
    src="/mapguide/mapviewerajax/?SESSION=<?=$sessionId ?>&
    WEBLAYOUT=<?=$webLayout ?>"></iframe>
</body>
</html>

```

Hello, Map 2 – Adding a Custom Command

Web layouts can include custom commands added to the toolbar, context menu, task list, or task pane area of the Viewer. These custom commands make up the MapGuide application.

This next 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 MapGuide 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 web layout also includes a URL to a home task that displays in the task pane. The home task can open other pages.

To create a new MSP and make it available as a command from the task list, do the following:

- Edit the web layout using MapGuide Studio.

- 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.
- Add the command to the Task Bar Menu.

It is also possible to add custom commands to the toolbar and the context menu.

For most of the examples in this guide, however, the pages will be links from a home page loaded in the task pane frame.

NOTE Installing the package that comes with the Developer's Guide samples creates a series of web layouts automatically. Each layout contains examples that correspond to one of the chapters in the Developer's Guide.

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 13) 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 9) 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 17) for details.

```

<html>
<head><title>Hello, map</title></head>
<body>
  <p>
    <?php

    // Define some common locations
    $installDir =
      'C:\Program Files\Autodesk\MapGuideEnterprise2007\';
    $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

The example is installed with the Developer's Guide samples. To install and run it separately from the samples, 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

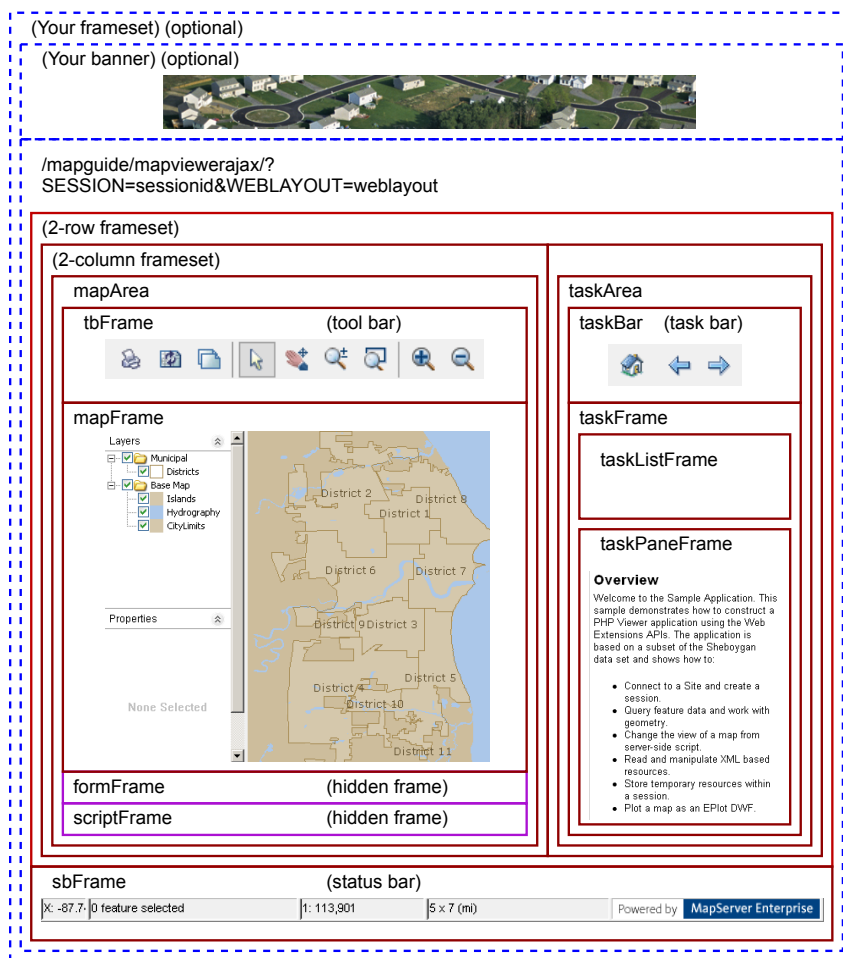
displays the spatial reference system used by the map, but you can also get more complex information about the layers that make up the map.

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.

Name	Description
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 13) 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 perform 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 17) 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 MapGuide 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 26) 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 55) 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 13) 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:

```

$location = -87.7116768; // Or calculate values
$ylocation = 43.7766789973;
$mapScale = 2000;
echo "<p><a href=\"gotopoint.php?\" .
    \"X=$location&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 55).

Changing Basic Properties

To query or change any of the basic layer properties like name, label, or group, use the `MgLayer::GetProperty()` and `MgLayer::SetProperty()` 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 label of the Roads layer between Roads and Streets.

```

MgInitializeWebTier ($webconfigFilePath);

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

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

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

$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);

```

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 55) for details.

The following example turns on the visibility for the Roads layer.

```
$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 35) 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::GetPropertyType()` 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 programmatically 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.

Passing Viewer Information to the Web Server

The Viewers, especially the DWF Viewer, manage many details about the state of the active map. These details are not always directly available to the Web Server. They are local to the Viewer, and are available through JavaScript calls to the Viewer API.

A good example of this is the active selection. The DWF Viewer maintains the active selection. To avoid excessive traffic between the Viewer and the Web Server, the Viewer does not notify the Web Server when the active selection changes.

Because of this, when the Viewer makes a request to a MapGuide Server Page on the Web Server, it must often pass information as part of the request. Some common methods for passing this information are:

- as an additional parameter to an Invoke URL command in a web layout
- through an Invoke Script command that executes the `Submit` method of the hidden `formFrame`
- through an `onClick` or other event that executes the `Submit` method of the hidden `formFrame`

The best method to use depends on the requirements of the application. If you are invoking the request from a command defined in a web layout, you can pass the information either as an additional parameter to an Invoke URL command or through an Invoke Script command. Invoke URL is simpler, but it offers a restricted set of parameters. Invoke Script has complete access to all the JavaScript calls in the Viewer API.

If you are invoking the request from a page in the task pane, you can execute JavaScript as part of an `onClick` event or a form action.

Additional Parameters to an Invoke URL Command

With this release of MapGuide, the current selection is the only variable that can be passed as part of an Invoke URL command.

To pass the current selection, edit the web layout. Define a new Invoke URL command. On the Additional Parameters tab, enter a key and value. The key must be a valid HTTP POST key. For the value, enter `$CurrentSelection`. Add the command to the toolbar, context menu, or task bar menu.

When the command is executed, the current selection is passed to the page, along with the standard variables like `SESSION` and `MAPNAME`.

For example, if you define the key `SEL` to have the value `$CurrentSelection`, then when the URL is invoked

```
$selection = $_POST['SEL'];
```

gets the current selection, in XML format.

See [Working With the Active Selection](#) (page 46) for details about using the XML data.

Passing Parameters From an Invoke Script command

An Invoke Script command in a web layout can be used to pass custom parameters to a page. The parameters can be any values that are available via the Viewer API.

To pass parameters, edit the web layout. Define a new Invoke Script command. On the Additional Parameters tab, enter the JavaScript code to retrieve the values to be passed. Add the command to the toolbar, context menu, or task bar menu.

The JavaScript code can call Viewer API functions or other functions to retrieve values. To pass the parameters to a page, call the `submit` method of the `formFrame` with the parameters, the page URL, and the name of the target frame.

For example, if you wrap custom HTML around the Viewer, you could define JavaScript functions in the wrapper, then call these functions from an Invoke

Script command. See [Example: Listing Selected Parcels \(DWF Viewer\)](#) (page 49).

NOTE The parameters must include standard parameters like `SESSION` and `MAPNAME`, if they are needed.

Passing Parameters From the Task Pane Frame

Passing parameters from the task pane frame is similar to passing them from an Invoke Script command. Use the Viewer API to retrieve values and call the `Submit` method of the `formFrame` to pass the values to another page.

For example, the following function passes the map view scale and the center point as parameters to a page that opens in a new window.

```
function submitRequest(pageUrl)
{
    xmlSel = parent.parent.mapFrame.GetSelectionXML();
    mapScale = parent.parent.mapFrame.GetScale();
    mapCenter = parent.parent.mapFrame.GetCenter();
    params = new Array(
        "SESSION", parent.parent.mapFrame.GetSessionId(),
        "MAPNAME", parent.parent.mapFrame.GetMapName(),
        "SELECTION", xmlSel,
        "SCALE", mapScale,
        "CENTERX", mapCenter.X,
        "CENTERY", mapCenter.Y
    );
    parent.parent.formFrame.Submit(pageUrl, params, "_blank");
}
```

To call the function, execute it as part of an `onClick` event or as the action in a form. For example, clicking the following link would execute the function:

```
<a href="#"
    onClick="submitRequest(
        '/mapguide/devguide/custom_output/property_report.php');
    return false;">
    Click for report</a>
```

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 50), 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 35). 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 39).

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 associate 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 67).

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
$xmlPath = new DOMXPath($domDocument);
$query = '//AreaRule/Filter';
// Get a list of all the <AreaRule><Filter> elements in
// the XML.
$nodes = $xmlPath->query($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 = $xmlPath->query($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 in [Adding Layers To A Map](#) (page 67).

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.

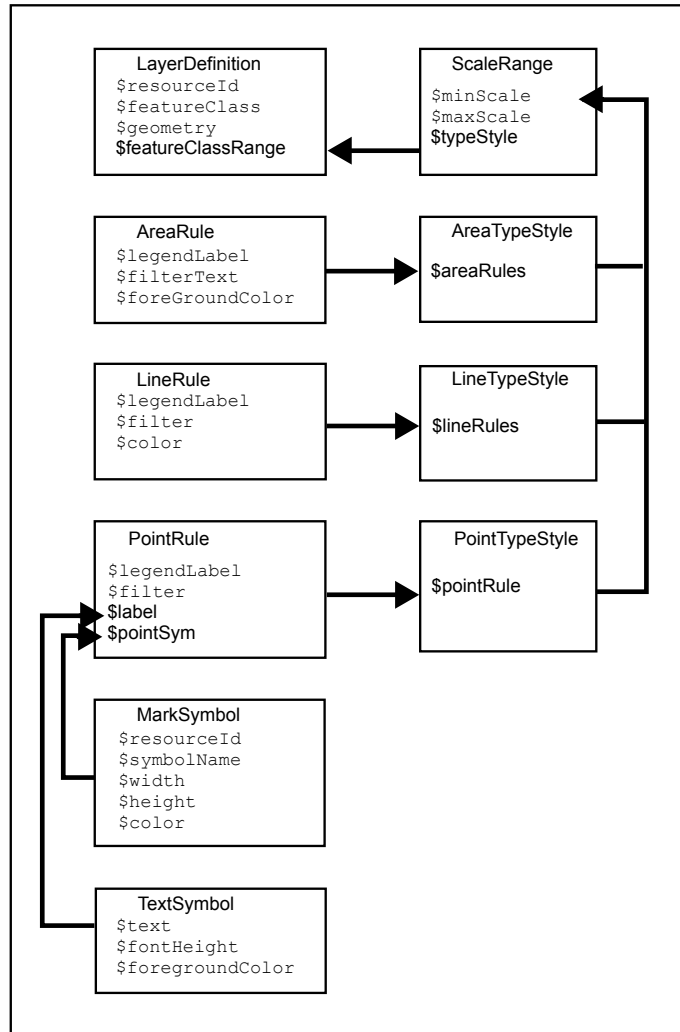
See [Adding Layers To A Map](#) (page 67).

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: <code>Library://Samples/Sheboygan/Data/Parcels.FeatureSource</code> . 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.
	\$feature-ClassRange	A scale range created by filling in a scale range template (ScaleRange.template).
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
	<code>\$foregroundColor</code>	The color to be applied to areas that match this rule. Equivalent to the Foreground color field in Studio's layer editor.
<code>CreateTextSymbol()</code>	<code>\$text</code>	The string for the text.
	<code>\$fontHeight</code>	The height for the font.
	<code>\$foregroundColor</code>	The foreground color.
<code>CreatePointTypeStyle()</code>	<code>\$pointRule</code>	One or more point rules, created by <code>CreatePointRule()</code> .
<code>CreatePointRule()</code>	<code>\$legendLabel</code>	The label shown beside this rule in the legend. Equivalent to the Legend label field in Studio's layer editor.
	<code>\$filter</code>	The filter expression that determines which features match this rule. Equivalent to the Condition field in Studio's layer editor.
	<code>\$label</code>	The text symbol, created by <code>CreateTextSymbol()</code> .
<code>CreateMarkSymbol()</code>	<code>\$pointSym</code>	A mark symbol created by <code>CreateMarkSymbol()</code> .
	<code>\$resourceId</code>	The resource ID of the symbol used to mark each point. For example, <code>library://Samples/Sheboygan/Symbols/BasicSymbols.SymbolLibrary</code> . Equivalent to the Location field in the Select a symbol from a Symbol Library dialog in Studio's layer editor.
	<code>\$symbolName</code>	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.
CreateLineStyleStyle()	\$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 >= 1 AND SQFT < 800', 'FFFFFF00');
$areaRule2 = $factory->CreateAreaRule( '800 to 1600',
    'SQFT >= 800 AND SQFT < 1600', 'FFFFBF20');
$areaRule3 = $factory->CreateAreaRule('1600 to 2400',
    'SQFT >= 1600 AND SQFT < 2400', 'FFF8040');
// 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 definition.
$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 67).

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 definition.
$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 definition (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($layerDefin
    ition));
$byteSource->SetMimeType(MgMimeType::Xml);
$resourceId = new MgResourceIdentifier("Library://LayerName.Layer
    Definition");
$resourceService->SetResource($resourceId, $byteSource->GetRead
    er(), null);

```

Analyzing Features

5

In this chapter

- [Introduction](#)
- [Representation of Geometry](#)
- [Coordinate Systems](#)
- [Measuring Distance](#)
- [Temporary Feature Sources](#)
- [Creating a Buffer](#)
- [Example](#)

Introduction

MapGuide includes methods for analyzing map features, including comparing the spatial relationships between features, measuring distances, and creating buffer areas around features.

Analyzing features requires knowing how the features are represented and what spatial reference systems are being used. If different spatial reference systems are being used, it is important to be able to convert between them.

Representation of Geometry

MapGuide can represent geometric data in three different forms:

- AGF text format, which is an extension of the Open Geospatial Consortium (OGC) Well Known Text (WKT) format. This is used to represent geometry as a character string.
- Binary AGF format. This is used by the FDO technology supporting the Feature Service.
- MapGuide internal representation, using `MgGeometry` and classes derived from it.

NOTE This guide and the Web API Reference will often use the term `WKT` to mean AGF text format. Be aware that AGF Text values do not always conform to the OGC WKT standard. See the Geometry module in the Web API Reference for details.

To convert between AGF text and the MapGuide internal representation, use an `MgWktReaderWriter` object.

To convert between binary AGF and the MapGuide internal representation, use an `MgAgfReaderWriter` object.

Geometry Objects

`MgGeometry` is the base class for all the geometry types. The simple geometry types are:

- `MgPoint` — a single point

- `MgLineString` — a series of connected line segments
- `MgCurveString` — a series of connected curve segments
- `MgPolygon` — a polygon with sides formed from line segments
- `MgCurvePolygon` — a polygon with sides formed from curve segments

The curve segments are circular arcs, defined by a start point, an end point, and a control point.

Complex types are formed by aggregating simple types. The complex types are:

- `MgMultiPoint` — a group of points
- `MgMultiLineString` — a group of line strings
- `MgMultiCurveString` — a group of curve strings
- `MgMultiPolygon` — a group of polygons
- `MgMultiCurvePolygon` — a group of curve polygons
- `MgMultiGeometry` — a group of simple geometry objects of any type

Comparing Geometry Objects

The `MgGeometry` class contains methods for comparing different geometry objects. These are similar to the spatial filters described in [Selecting with the Web API](#) (page 35). Methods to test spatial relationships include:

- `Contains()`
- `Crosses()`
- `Disjoint()`
- `Equals()`
- `Intersects()`
- `Overlaps()`
- `Touches()`
- `Within()`

For example, if you have an `MgLineString` object `$line` and an `MgPolygon` object `$polygon`, you can test if the line crosses the polygon with a call to

```
$line->Crosses($polygon)
```

Methods to create new geometry objects from the point set of two other geometries include:

- `Difference()`
- `Intersection()`
- `SymmetricDifference()`
- `Union()`

Complete details are in the Geometry module of the Web API reference, under Spatial Relationships.

Coordinate Systems

A single map will often combine data from different sources, and the different sources may use different coordinate systems. The map has its own coordinate system, and any feature sources used in the map may have different coordinate systems. It is important for display and analysis that all locations are transformed to the same coordinate system.

NOTE A coordinate system can also be called a spatial reference system (SRS) or a coordinate reference system (CRS). This guide uses the abbreviation SRS.

MapGuide supports three different types of coordinate system:

- Arbitrary X-Y
- Geographic, or latitude/longitude
- Projected

An `MgCoordinateSystem` object represents a coordinate system.

NOTE You cannot transform between arbitrary X-Y coordinates and either geographic or projected coordinates.

To create an `MgCoordinateSystem` object from an `MgMap` object,

- Get the WKT representation of the map coordinate system, using `MgMap::GetMapSRS()`.
- Create an `MgCoordinateSystem` object, using `MgCoordinateSystemFactory::Create()`.

To create an `MgCoordinateSystem` object from a map layer,

- Get the feature source for the layer.
- Get the active spatial context for the feature source.
- Convert the spatial context to a WKT.
- Create an `MgCoordinateSystem` object from the WKT.

To transform geometry from one coordinate system to another, create an `MgCoordinateSystemTransform` object using the two coordinate systems. Apply this transform to the `MgGeometry` object.

For example, if you have geometry representing a feature on a layer that uses one coordinate system, and you want to compare it to a feature on another layer that uses a different coordinate system, perform the following steps:

```
$featureSource1 = $layer1->GetFeatureSourceId();
$contexts1 = $featureService->GetSpatialContexts(
    $featureSource1, true);
$contexts1->ReadNext();
$srs1 = $contexts1->GetCoordinateSystemWkt();

$featureSource2 = $layer2->GetFeatureSourceId();
$contexts2 = $featureService->GetSpatialContexts(
    $featureSource2, true);
$contexts2->ReadNext();
$srs2 = $contexts2->GetCoordinateSystemWkt();

$xform = new MgCoordinateSystemTransform($srs1, $srs2);
$geometry|xform->$geometry1->Transform($xform);
```

Measuring Distance

Measuring distance in geographic or projected coordinate systems requires great circle calculations. Both `MgGeometry::Buffer()` and `MgGeometry::Distance()` accept a measurement parameter that defines the great circle to be used. If the measurement parameter is null, the calculation is done using a linear algorithm.

Create the measurement parameter, an `MgCoordinateSystemMeasure` object, from the `MgCoordinateSystem` object.

Distance is calculated in the units of the SRS. `MgCoordinateSystem` includes two methods, `ConvertCoordinateSystemUnitsToMeters()` and `ConvertMetersToCoordinateSystemUnits()` to convert to and from linear distances.

For example, to calculate the distance between two `MgGeometry` objects `$a` and `$b`, using the coordinate system `$srs`, perform the following steps:

```
$measure = new MgCoordinateSystemMeasure($srs);  
$distInMapUnits = $a->Distance($b, $measure);  
$distInMeters = $srs->ConvertCoordinateSystemUnitsToMeters(  
    $distInMapUnits);
```

Another way to calculate the distance is to use `MgCoordinateSystemMeasure::GetDistance()`, as in the following:

```
$distInMapUnits = $measure->GetDistance($a, $b);
```

Temporary Feature Sources

Many geometric analysis operations require creating new features and new feature sources. For example, drawing a buffer around a point on a map requires a layer to display the buffer polygon, and the layer requires a feature source.

To create a temporary feature source, perform the following steps:

- Create a feature class definition.
- Determine what properties you need to store for the features. Add the property definitions to the feature class definition.
- Create a feature schema containing the feature class definition.

- Determine the SRS for the feature source. This can be the same as the SRS used for the map.
- Create a feature source using the schema and the SRS. The feature source can be stored in the session repository.

It is possible for a single feature source to contain more than one feature class. A feature source that is to be used for temporary data, however, normally contains one feature class.

A feature schema (`MgFeatureSchema` object) contains class definitions (`MgClassDefinition` objects) for each feature class in the schema.

Each class definition contains property definitions for each property in the feature class. The property definitions can be the following types:

- `MgDataPropertyDefinition`
- `MgGeometryPropertyDefinition`
- `MgObjectPropertyDefinition`
- `MgRasterPropertyDefinition`

`MgDataPropertyDefinition` is used to define simple properties like numbers or strings. `MgGeometryPropertyDefinition` is used to define geometric properties. Most feature classes will have a geometric property to describe the feature's location.

For example, the following creates a temporary feature source to hold buffer features. The feature source contains a single feature class named `BufferClass`.

Features in `BufferClass` have two properties. `ID` is an autogenerated unique ID number, and `BufferGeometry` contains the geometry for the buffer polygon.

The FDO technology supporting the Feature Service allows for multiple spatial reference systems within a single feature source. However, this capability is dependent on the data provider, and does not apply to the SDF provider that is used for creating feature sources within MapGuide. For temporary feature sources, you must define a single default SRS for the feature source, and you must set any geometry properties to use the same SRS. The name of the SRS is user-defined.

```

$bufferClass = new MgClassDefinition();
$bufferClass->SetName('BufferClass');
$properties = $bufferClass->GetProperties();

$idProperty = new MgDataPropertyDefinition('ID');
$idProperty->SetDataType(MgPropertyType::Int32);
$idProperty->SetReadOnly(true);
$idProperty->SetNullable(false);
$idProperty->SetAutoGeneration(true);
$properties->Add($idProperty);

$polygonProperty = new
    MgGeometricPropertyDefinition('BufferGeometry');
$polygonProperty->
    SetGeometryTypes(MgFeatureGeometricType::Surface);
$polygonProperty->SetHasElevation(false);
$polygonProperty->SetHasMeasure(false);
$polygonProperty->SetReadOnly(false);
$polygonProperty->SetSpatialContextAssociation('defaultSrs');
$properties->Add($polygonProperty);

$idProperties = $bufferClass->GetIdentityProperties();
$idProperties->Add($idProperty);

$bufferClass->SetDefaultGeometryPropertyName('BufferGeometry');

$bufferSchema = new MgFeatureSchema('BufferLayerSchema',
    'temporary schema to hold a buffer');
$bufferSchema->GetClasses()->Add($bufferClass);

$sdfParams = new MgCreateSdfParams('defaultSrs', $wkt,
    $bufferSchema);

$featureService->CreateFeatureSource($bufferFeatureResId,
    $sdfParams);

```

To display features from a temporary feature source in a map, create a layer definition that refers to the feature source. Use the techniques described in [Modifying Maps and Layers](#) (page 55).

Inserting, Deleting, and Updating Features

To change data in a feature source, create an `MgFeatureCommandCollection` object. This can contain commands to insert, delete, or update features in an FDO data source. The commands are executed sequentially. For FDO providers that support transaction processing, the commands can be treated as a single transaction.

Feature commands can be one of the following:

- `MgDeleteFeatures`
- `MgInsertFeatures`
- `MgUpdateFeatures`

To execute the commands, call `MgFeatureService::UpdateFeatures()`. The feature class name and property names in any of the feature commands must match the class name and property names in the feature source.

For example, to delete all features in a feature class with an identity property ID, execute the following:

```
$commands = new MgFeatureCommandCollection();
$deleteCommand = new MgDeleteFeatures($className, "ID like '%');
$commands->Add($deleteCommand);

$featureService->UpdateFeatures($featureSource, $commands, false);
```

To insert features, create an `MgPropertyCollection` object that contains the properties of the new feature. Create an `MgInsertFeatures` object and add this to the `MgFeatureCommandCollection` object.

For example, to add a new feature with a single geometry property, execute the following:

```

$commands = new MgFeatureCommandCollection();
$properties = new MgPropertyCollection();
$agfByteStream = $agfReaderWriter->Write($geometry);
$geometryProperty = new MgGeometryProperty($propertyName,
    $agfByteStream);
$properties->Add($geometryProperty);

$insertCommand = new MgInsertFeatures($className, $properties);
$commands->Add($insertCommand);

$featureService->UpdateFeatures($featureSource, $commands, false);

```

To update existing features, create an `MgPropertyCollection` object that contains the new values for the properties and a filter expression that selects the correct feature or features. See [Querying Feature Data](#) (page 35) for details about filter expressions.

Creating a Buffer

To create a buffer around a feature, use the `MgGeometry::Buffer()` method. This returns an `MgGeometry` object that you can use for further analysis. For example, you could display the buffer by creating a feature in a temporary feature source and adding a new layer to the map. You could also use the buffer geometry as part of a spatial filter. For example, you might want to find all the features within the buffer zone that match certain criteria, or you might want to find all roads that cross the buffer zone.

To display the buffer in the map, perform the following steps:

- Create a feature source for the buffer.
- Insert a buffer feature in the feature source.
- Create a layer that references the feature source.
- Add the layer to the map and make it visible.

To use the buffer as part of a query, create a spatial filter using the buffer geometry, and use this in a call to `MgFeatureService::SelectFeatures()`. For example, the following code selects parcels inside the buffer area that are of type "MFG". You can use the `MgFeatureReader` to perform tasks like generating a report of the parcels, or creating a new layer that puts point markers on each parcel.


```

$queryOptions = new MgFeatureQueryOptions();
$queryOptions->SetFilter("RTYPE = 'MFG'");
$queryOptions->SetSpatialFilter('SHPGEOM', $bufferGeometry,
    MgFeatureSpatialOperations::Inside);

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

```

Example

This example builds on the example from [Working With the Active Selection](#) (page 46). Instead of listing the parcels in the selection, it creates a series of concentric buffers around the selection, showing increasing distance. The code sections below contain the significant additions in this example. The complete source code is available with the Developer's Guide samples.

Because this example modifies the map, it must refresh the map when it loads, by executing a JavaScript function. Add the function to the page.

```

<script language="javascript">
    function OnPageLoad()
    {
        parent.parent.Refresh();
    }
</script>

```

Add an `OnLoad` command to the `<body>` element:

```

<body onLoad="OnPageLoad()">

```

The example uses a temporary map layer named `Buffer` to store the buffer feature. It creates a feature source and the layer if it does not exist. Otherwise, it deletes any existing features before creating the new buffer. The functions `CreateBufferFeatureSource()` and `CreateBufferLayer()` are in `bufferfunctions.php`, which is described below.

```

include 'bufferfunctions.php';
$bufferRingSize = 100; // measured in metres
$bufferRingCount = 5;

// Set up some objects for coordinate conversion

$mapWktSrs = $map->GetMapSRS();
$agfReaderWriter = new MgAgfReaderWriter();
$wktReaderWriter = new MgWktReaderWriter();
$coordinateSystemFactory = new MgCoordinateSystemFactory();
$srs = $coordinateSystemFactory->Create($mapWktSrs);
$srsMeasure = new MgCoordinateSystemMeasure($srs);

// Check for a buffer layer. If it exists, delete
// the current features.
// If it does not exist, create a feature source and
// a layer to hold the buffer.

try
{
    $bufferLayer = $map->GetLayers()->GetItem('Buffer');
    $bufferFeatureResId = new MgResourceIdentifier(
        $bufferLayer->GetFeatureSourceId());

    $commands = new MgFeatureCommandCollection();
    $commands->Add(new MgDeleteFeatures('BufferClass',
        "ID like '%" ));

    $featureService->UpdateFeatures($bufferFeatureResId,
        $commands, false);
}
catch (MgObjectNotFoundException $e)
{
    // When an MgObjectNotFoundException is thrown, the layer
    // does not exist and must be created.

    $bufferFeatureResId = new MgResourceIdentifier("Session:" .
        $mgSessionId . "//Buffer.FeatureSource");
    CreateBufferFeatureSource($featureService, $mapWktSrs,
        $bufferFeatureResId);
    $bufferLayer = CreateBufferLayer($resourceService, $buffer
FeatureResId, $mgSessionId);
    $map->GetLayers()->Insert(0, $bufferLayer);
}

```

```
}
```

The geometries for the selected features are merged into a single multi-geometry. Then a series of concentric buffers is created and added to the feature source. The style for the layer, which is set when function `CreateBufferLayer()` processes `bufferlayerdefinition.xml`, should define the buffer features to be partly transparent. When they are drawn on the map, the rings get progressively darker towards the center of the buffer area.

```

// Process each item in the MgFeatureReader.
// Merge them into a single feature.

$inputGeometries = new MgGeometryCollection();
while ($featureReader->ReadNext())
{
    $featureGeometryData = $featureReader->GetGeometry('SHPGEOM');
    $featureGeometry = $agfReaderWriter->Read($featureGeometryData);

    $inputGeometries->Add($featureGeometry);
}

$geometryFactory = new MgGeometryFactory();
$mergedFeatures = $geometryFactory->
    CreateMultiGeometry($inputGeometries);

// Add buffer features to the temporary feature source.
// Create multiple concentric buffers to show area.

$commands = new MgFeatureCommandCollection();
for ($bufferRing = 0; $bufferRing < $bufferRingCount;
    $bufferRing++)
{
    $bufferDist = $srs->
        ConvertMetersToCoordinateSystemUnits($bufferRingSize *
        ($bufferRing + 1));
    $bufferGeometry = $mergedFeatures->Buffer($bufferDist,
        $srsMeasure);

    $properties = new MgPropertyCollection();
    $properties->Add(new MgGeometryProperty('BufferGeometry',
        $agfReaderWriter->Write($bufferGeometry)));

    $commands->Add(new MgInsertFeatures('BufferClass',
        $properties));
}

$featureService->UpdateFeatures($bufferFeatureResId,
    $commands, false);

$bufferLayer->SetVisible(true);
$bufferLayer->ForceRefresh();
$bufferLayer->SetDisplayInLegend(true);

```

```
$map->Save($resourceService);
```

The functions `CreateBufferFeatureSource()` and `CreateBufferLayer()` are in `bufferfunctions.php`. `CreateBufferFeatureSource()` creates a temporary feature source, with a single feature class, `BufferClass`. The feature class has two properties, `ID` and `BufferGeometry`. `ID` is autogenerated, so it does not need to be added with a new feature. `CreateBufferLayer()` modifies a layer definition from an external file and saves it to the repository. For more details, see [Modifying Maps and Layers](#) (page 55).

```

function CreateBufferFeatureSource($featureService, $wkt,
    $bufferFeatureResId)
{
    $bufferClass = new MgClassDefinition();
    $bufferClass->SetName('BufferClass');
    $properties = $bufferClass->GetProperties();
    $idProperty = new MgDataPropertyDefinition('ID');
    $idProperty->SetDataType(MgPropertyType::Int32);
    $idProperty->SetReadOnly(true);
    $idProperty->SetNullable(false);
    $idProperty->SetAutoGeneration(true);
    $properties->Add($idProperty);
    $polygonProperty = new
        MgGeometricPropertyDefinition('BufferGeometry');
    $polygonProperty->
        SetGeometryTypes(MgFeatureGeometricType::Surface);
    $polygonProperty->SetHasElevation(false);
    $polygonProperty->SetHasMeasure(false);
    $polygonProperty->SetReadOnly(false);
    $polygonProperty->SetSpatialContextAssociation('defaultSrs');
    $properties->Add($polygonProperty);
    $idProperties = $bufferClass->GetIdentityProperties();
    $idProperties->Add($idProperty);
    $bufferClass->
        SetDefaultGeometryPropertyName('BufferGeometry');
    $bufferSchema = new MgFeatureSchema('BufferLayerSchema',
        'temporary schema to hold a buffer');
    $bufferSchema->GetClasses()->Add($bufferClass);
    $sdfParams = new MgCreateSdfParams('defaultSrs', $wkt,
        $bufferSchema);
    $featureService->CreateFeatureSource($bufferFeatureResId,
        $sdfParams);
}

function CreateBufferLayer($resourceService,
    $bufferFeatureResId, $sessionId)
{
    // Load the layer definition template into
    // a PHP DOM object, find the "ResourceId" element, and
    // modify its content to reference the temporary
    // feature source.
    $doc = DOMDocument::load('bufferlayerdefinition.xml');
    $featureSourceNode = $doc->getElementsByName(

```

```

        'ResourceId')->item(0);
    $featureSourceNode->nodeValue =
        $bufferFeatureResId->ToString();
    // Get the updated layer definition from the DOM object
    // and save it to the session repository using the
    // ResourceService object.
    $layerDefinition = $doc->saveXML();
    $byteSource = new MgByteSource($layerDefinition,
        strlen($layerDefinition));
    $byteSource->SetMimeType(MgMimeType::Xml);
    $tempLayerResId = new MgResourceIdentifier("Session:" .
        $sessionId . "//Buffer.LayerDefinition");
    $resourceService->SetResource($tempLayerResId,
        $byteSource->GetReader(), null);
    // Create an MgLayer object based on the new layer definition
    // and return it to the caller.
    $bufferLayer = new MgLayer($tempLayerResId, $resourceService);
    $bufferLayer->SetName("Buffer");
    $bufferLayer->SetLegendLabel("Buffer");
    $bufferLayer->SetDisplayInLegend(true);
    $bufferLayer->SetSelectable(false);
    return $bufferLayer;
}

```

There is an additional example in the Developer's Guide samples. It queries the parcels in the buffer area and selects parcels that match certain criteria. The selection is done using a query that combines a basic filter and a spatial filter.

```

    $bufferDist = $srs->
        ConvertMetersToCoordinateSystemUnits($bufferRingSize);
    $bufferGeometry = $mergedGeometries->Buffer($bufferDist,
        $srsMeasure);

    // Create a filter to select parcels within the buffer. Combine
    // a basic filter and a spatial filter to select all parcels
    // within the buffer that are of type "MFG".

    $queryOptions = new MgFeatureQueryOptions();
    $queryOptions->SetFilter("RTYPE = 'MFG'");
    $queryOptions->SetSpatialFilter('SHPGEOM', $bufferGeometry,
        MgFeatureSpatialOperations::Inside);

```

It creates an additional feature source that contains point markers for each of the selected parcels.

```

// Get the features from the feature source,
// determine the centroid of each selected feature, and
// add a point to the ParcelMarker layer to mark the
// centroid.
// Collect all the points into an MgFeatureCommandCollection,
// so they can all be added in one operation.

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

$parcelMarkerCommands = new MgFeatureCommandCollection();
while ($featureReader->ReadNext())
{
    $byteReader = $featureReader->GetGeometry('SHPGEOM');
    $geometry = $agfReaderWriter->Read($byteReader);
    $point = $geometry->GetCentroid();

    // Create an insert command for this parcel.
    $properties = new MgPropertyCollection();
    $properties->Add(new MgGeometryProperty('ParcelLocation',
        $agfReaderWriter->Write($point)));
    $parcelMarkerCommands->Add(
        new MgInsertFeatures('ParcelMarkerClass', $properties));
}
$featureReader->Close();

if ($parcelMarkerCommands->GetCount() > 0)
{
    $featureService->UpdateFeatures($parcelFeatureResId,
        $parcelMarkerCommands, false);
}
else
{
    echo '</p><p>No parcels within the buffer area match.';
}

```


Digitizing and Redlining

6

In this chapter

- [Introduction](#)
- [Digitizing](#)
- [Redlining](#)

Introduction

This chapter describes *digitizing* (capturing the user's clicks on the map and converting the locations to map coordinates) and *redlining* (drawing items such as lines or rectangles on the map in response to the user's clicks).

Digitizing

The Viewer API has a number of functions for digitizing user input. For an example of how these can be used, see `digitizing_features` ► `task_pane_digitizing.php` in the Developer Guide samples.

In this example, if the user clicks the button to digitize a point

```
<input type="button" value=" Point " onclick="DigitizePoint();">
```

the script calls the JavaScript function

```
function DigitizePoint() {  
    parent.parent.mapFrame.DigitizePoint(OnPointDigitized);  
}
```

which in turn calls the `DigitizePoint()` method of the Viewer API in the map frame. It also passes the name of a callback function, `OnPointDigitized`, which is defined in the current script. `DigitizePoint()` calls this function after it has digitized the point and passes it the digitized coordinates of the point.

You can use this callback function to process the digitized coordinates as you wish. In this example, the script simply displays them in the task pane.

```
function OnPointDigitized(point) {  
    ShowResults("X: " + point.X + ", Y: " + point.Y);  
}
```

Redlining

There are three main steps involved in redlining:

- 1 Pass the digitized coordinates from the client to the server.
- 2 Create a temporary feature source. This will be used to draw the lines on.
- 3 Create a layer to display that temporary feature source.

For example, see `digitizing_features` ➤ `task_pane_redlining.php` in the Developer Guide samples.

Passing Coordinates

The digitizing functions in the Viewer API provide us with the digitized coordinates on the client, but we usually need to pass them to a server side script. This can be done by having the callback function fill in the fields of a hidden form, and then submit the form. The form then passes the variables to a server-side script. For example, the callback function could be:

```
function OnLineDigitized(line) {
    // Send the Javascript variables to 'draw_line.php',
    // via the hidden form defined below
    document.getElementById("x0").value = line.Point(0).X;
    document.getElementById("y0").value = line.Point(0).Y;
    document.getElementById("x1").value = line.Point(1).X;
    document.getElementById("y1").value = line.Point(1).Y;

    var sessionId = "<?php echo ($_SERVER['REQUEST_METHOD'] ==
    'POST')? $_POST['SESSION']: $_GET['SESSION']; ?>";
    document.getElementById("session_id").value = sessionId;

    document.myform.submit();
}
```

That function fills out and submits this form:

```
<!-- A hidden form, used to pass the JavaScript variables to
draw_line.php -->
<form target="scriptFrame" name="myform" action="draw_line.php"
method="post" onsubmit="submitForm()">
    <input type="hidden" id="x0" name="x0">
    <input type="hidden" id="y0" name="y0">
    <input type="hidden" id="x1" name="x1">
    <input type="hidden" id="y1" name="y1">
    <input type="hidden" id="session_id" name="SESSION">
    <input type="hidden" name="MAPNAME" value="Sheboygan">
</form>
```

Creating a Feature Source

The next step is create a feature source

See `digitizing_features ► draw_line.php` in the Developer Guide samples.

```

// Create a temporary feature source to draw the lines on

// Create a feature class definition for the new feature
// source
$classDefinition = new MgClassDefinition();
$classDefinition->SetName("Lines");
$classDefinition->SetDescription("Lines to display.");
$geometryPropertyName="SHPGEOM";
$classDefinition->
    SetDefaultGeometryPropertyName( $geometryPropertyName);

// Create an identify property
$identityProperty = new MgDataPropertyDefinition("KEY");
$identityProperty->SetDataType(MgPropertyType::Int32);
$identityProperty->SetAutoGeneration(true);
$identityProperty->SetReadOnly(true);
// Add the identity property to the class definition
$classDefinition->GetIdentityProperties()->
    Add($identityProperty);
$classDefinition->GetProperties()->Add($identityProperty);

// Create a name property
$nameProperty = new MgDataPropertyDefinition("NAME");
$nameProperty->SetDataType(MgPropertyType::String);
// Add the name property to the class definition
$classDefinition->GetProperties()->Add($nameProperty);

// Create a geometry property
$geometryProperty = new
    MgGeometricPropertyDefinition($geometryPropertyName);
$geometryProperty->
    SetGeometryTypes(MgFeatureGeometricType::Surface);
// Add the geometry property to the class definition
$classDefinition->GetProperties()->Add($geometryProperty);

// Create a feature schema
$featureSchema = new MgFeatureSchema("SHP_Schema",
    "Line schema");
// Add the feature schema to the class definition
$featureSchema->GetClasses()->Add($classDefinition);

// Create the feature source
$wkt = $map->GetMapSRS();

```

```
$sdfParams = new MgCreateSdfParams("ArbitraryXY",  
    $wkt, $featureSchema);  
$featureService->CreateFeatureSource($resourceIdentifier,  
    $sdfParams);
```

Creating A Layer

The final step is to create a new layer to display the feature source, the same way it was done in [Adding Layers To A Map](#) (page 67).

Custom Output

7

In this chapter

- [Introduction](#)
- [Rendering Service](#)
- [Mapping Service](#)

Introduction

MapGuide includes services for saving map representations for use in external programs.

To save a map as a bit-mapped image (PNG or GIF), use the Rendering Service.

To save a map as a Design Web Format (DWF), use the Mapping Service. DWF files can be saved in two variations. An eMap DWF contains metadata that describes the current map view, not the map data itself. This is a compact format, but it requires access to the map agent to view the map. It is not suitable for offline viewing, and it requires a current MapGuide session.

An ePlot DWF is designed for offline viewing or printing. It can contain multiple sheets, where each sheet is a complete map image that can be viewed offline using the Autodesk DWF Viewer. Each sheet in an ePlot DWF is a static representation of a single map view state.

Characteristics of bit-mapped images:

- Images can be in PNG or GIF formats.
- An image displays a portion of the the map view state at a particular scale.
- The image is static with a fixed resolution. Zooming in creates a pixelated image.
- Images are cross-platform.
- Images are suitable for use in HTML pages, word processor documents, or graphics editing programs.

Characteristics of an ePlot DWF:

- A single ePlot can contain multiple sheets.
- Each sheet shows a single image, showing a portion of the map view at a particular scale.
- The ePlot area and scale are static, but geometric features are stored as vector graphics, so zooming is smooth.
- Some interactive features of the MapGuide DWF Viewer are available, such as the ability to turn layers on and off.
- The ePlot requires the Autodesk DWF Viewer, either standalone or as a plug-in for Internet Explorer.

- Images can be copied to the Windows clipboard and used in other applications.
- The Autodesk DWF Viewer is only available on Windows.

Characteristics of an eMap DWF:

- An eMap DWF is dynamic, with all the zooming and panning capabilities of the MapGuide DWF Viewer.
- Because the eMap Viewer uses the map agent, the entire map is available for smooth zooming and panning. The layer stylization rules apply.
- The eMap requires the Autodesk DWF Viewer, either standalone or as a plug-in for Internet Explorer.
- The eMap requires an active MapGuide session. If the session times out the map cannot be displayed.
- Individual views from an eMap DWF can be saved as ePlot DWFs.
- The Autodesk DWF Viewer is only available on Windows.

Rendering Service

The Rendering Service creates bit-mapped images of a map suitable for displaying in a browser or saving to a file. The image is returned as an `MgByteReader` object, which can be sent to a browser or saved to a file.

For example, to create a PNG image of a map area, perform the following operations. Note that the aspect ratio of the envelope should match the image dimensions or the image will be distorted.

```
$byteReader = $renderingService->RenderMap($map, $selection,
    $envelope, $imageWidth, $imageHeight, $color, 'PNG');

header("Content-type: " . $byteReader->GetMimeType() );

$buffer = '';
while ($byteReader->Read($buffer, 50000) != 0)
{
    echo $buffer;
}
```

Mapping Service

The Mapping Service creates eMap and ePlot DWFs.

Generating an eMap DWF requires the DWF version and the URI of the map agent. Note that the HTTP header must include content length information, as in the following example.

```
$dwfVersion = new MgDwfVersion("6.01", "1.2");

$mapAgentUri =
    'http://localhost:8008/mapguide/mapagent/mapagent.exe';
$byteReader = $mappingService->GenerateMap($map, $mapAgentUri,
    $dwfVersion);

$outputBuffer = '';
$buffer = '';
while ($byteReader->Read($buffer, 50000) != 0)
{
    $outputBuffer .= $buffer;
}
header('Content-Type: ' . $byteReader->GetMimeType());
header('Content-Length: ' . strlen($outputBuffer));
echo $outputBuffer;
```

An ePlot DWF is designed primarily for offline viewing or printing. It includes an `MgPlotSpecification` that defines the page size and margins. It can also include an optional `MgLayout` that defines additional components to include in the plot, like a legend or a custom logo. The layout is based on a print layout in the repository. For a description of the `PrintLayout` schema, see the [MapGuide Web API Reference](#).

To create an ePlot DWF with more than one sheet, use an `MgMapPlotCollection`, where each item in the collection is an `MgMapPlot` that describes a single sheet.

NOTE The map name becomes the sheet name in the multi-plot DWF. Because each sheet in the DWF must have a unique name, you must create a separate `MgMap` object for each sheet in the DWF.

The following example creates a multi-plot DWF with two sheets. The second sheet displays the same map area as the first, but it adds the title and legend information from the print layout.

```

$dwfVersion = new MgDwfVersion("6.01", "1.2");

$plotSpec = new MgPlotSpecification(8.5, 11,
    MgPageUnitsType::Inches);
$plotSpec->SetMargins(0.5, 0.5, 0.5, 0.5);

$plotCollection = new MgMapPlotCollection();

$plot1 = new MgMapPlot($map, $plotSpec, $layout);
$plotCollection->Add($plot1);

// Create a second map for the second sheet in the DWF. This
// second map uses the print layout
// to display a page title and legend.

$map2 = new MgMap();
$map2->Create($resourceService, $map->GetMapDefinition(),
    'Sheet 2');
$layoutRes = new MgResourceIdentifier(
    "Library://Samples/Sheboygan/Layouts/SheboyganMap.PrintLayout");
$layout = new MgLayout($layoutRes, "City of Sheboygan",
    MgPageUnitsType::Inches);
$plot2 = new MgMapPlot($map2,
    $map->GetViewCenter()->GetCoordinate(), $map->GetViewScale(),
    $plotSpec, $layout);
$plotCollection->Add($plot2);

$byteReader = $mappingService->
    GenerateMultiPlot($plotCollection, $dwfVersion);

// Now output the resulting DWF.

$outputBuffer = '';
$buffer = '';
while ($byteReader->Read($buffer, 50000) != 0)
{
    $outputBuffer .= $buffer;
}

header('Content-Type: ' . $byteReader->GetMimeType());
header('Content-Length: ' . strlen($outputBuffer));
echo $outputBuffer;

```


Index

A

- active selection 42, 46–47, 49–50
 - AJAX Viewer 47
 - DWF Viewer 49
 - in Web API 46
 - setting with Web API 50
- AGF 72
- AJAX Viewer 46
 - active selection 46
- arbitrary X-Y coordinates 74

B

- base layer groups 25
- buffer, creating 80
- buffer, example of creating 81

C

- callback functions 90
- constants.php 6
- coordinate systems 75
 - transforming between 75
- custom commands, in web layouts 4

D

- digitizing 90
- Document Object Model 56
- DOM 56
- DWF Viewer 46
 - active selection 46
- DWF, saving map as 96

E

- examples, preparing for 3

F

- feature classes 34
- feature readers 34–35
- feature source, temporary 76
- features 34, 39, 79
 - inserting, deleting, and updating 79
 - listing selected 39
- formFrame, in Viewer 11

G

- geographic coordinates 74
- geometry 73
 - comparing spatial relationships 73

H

- hellomap.php 7
- HTML page, with MapGuide Viewer 17

I

- Invoke Script command 16
- Invoke Script command, passing parameters 44
- Invoke URL command type 5
- Invoke URL command, additional parameters 44

L

- layer groups 24
- layer name 24
- layer properties 24
- layer style 26
- layer visibility 26
- layerdefinitionfactory.php 59
- layers 24

Library repository 19

M

map scale 27
maparea frame, in Viewer 11
mapFrame, in Viewer 11
MapGuide Server Page 5
mapping service 98
measuring distance 76
MgAgfReaderWriter 72
MgCoordinateSystem 74
MgCurvePolygon 73
MgCurveString 73
MgFeatureSchema 77
MgGeometry 38, 72
 creating from feature 38
MgLayerCollection object 28
MgLineString 73
MgMap 24
 GetLayers() 24
MgMap object 20
MgMultiCurvePolygon 73
MgMultiCurveString 73
MgMultiGeometry 73
MgMultiLineString 73
MgMultiPoint 73
MgMultiPolygon 73
MgPoint 72
MgPolygon 73
MgWktReaderWriter 72
MSP processing flow 6

O

onClick event 45

P

projected coordinates 74

R

redlining 90
refreshing a map 16

rendering service 96–97
resource service 10
resources 19

S

sbFrame, in Viewer 12
script frame, and Viewer API 15
scriptFrame, in Viewer 11
selecting 35, 42
 with the Viewer 42
 with the Web API 35
selecting, with Viewer 42
selection filters 36–37
 basic 36
 spatial 37
services 21
SESSION 45
session ID 6
session management 17
site connection 10
spatial operators 37
spatial reference system 74
spatial reference systems, in feature
 sources 77

T

task pane 5, 13–14
 and Viewer API 14
task pane frame, passing parameters
 from 45
taskArea, in Viewer 11
taskBar, in Viewer 11
taskFrame, in Viewer 11
taskListFrame, in Viewer 12
taskPaneFrame, in Viewer 12
tbFrame, in Viewer 11

U

user credentials 10

V

- Viewer API 13
- Viewer commands 2
- Viewer frames 11, 14
- Viewers 43
 - and map state 43

W

- web layout 3, 13
 - customizing 13
- web layout, defining 5
- webconfig.ini 10
- well-known text 72

