Autodesk MapGuide® Enterprise 2010

Developer's Guide

April 2009

Published by:
Autodesk, Inc.
111 Mclnnis Parkway
San Rafael, CA 94903, USA

Contents

Introduction 1

What This Guide Covers

This guide describes how to use the Autodesk MapGuide Enterprise 2010 Web API and Viewer API.

It assumes you have read the *MapGuide Getting Started* guide and are familiar with using Autodesk® MapGuide Studio or MapGuide Open Source Web Studio. Most examples also assume that you have installed the sample data and sample applications supplied with Autodesk 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*.

Most of the MapGuide Web API is designed to also work on AutoCAD® Map 3D with little or no modification. For details, see the *Autodesk Geospatial Platform Developer's Guide* distributed with AutoCAD Map 3D.

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 MapGuide 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 (JSP). These languages can use the Web API to retrieve, manipulate, and update mapping data.

The current version of MapGuide Open Source Web Studio does not create or edit web layouts. It is possible, however, to create and edit web layouts using the Mapagent HTML pages at `http://ServerAddress/mapguide/mapagent/index.html`. Get an existing web layout, such as the web layout supplied with the sample applications, using the `GetResourceContent` and `GetResourceHeader` links. Edit the XML in a text editor, then save to the site repository using the `SetResource` link.

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 The MapGuide Viewer (page 15).

Preparing to Run the Examples

MapGuide includes a set of sample applications. Some of them correspond directly to chapters in this Developer's Guide. These samples are designed to show one or two key concepts at a time.

Other sample applications are more full-featured. These are designed to show some of the capabilities of MapGuide. They are not discussed in detail in this guide, but they do build upon the basic concepts.

The sample applications are available on the installation CD. See the manual *Installing Sample Data and Viewer Sample Application* for details.

Complete examples are available from *http://mapguide.osgeo.org/downloads.html*. There are two required components: the source code and a package file for creating the web layouts. The Sheboygan sample data must also be installed.

NOTE The Web API supports .NET, Java, and PHP. For simplicity, the examples in this guide use PHP. However, many of the sample applications are available in all development languages.

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

This guide includes many code snippets. In most cases, the snippets are incomplete, lacking initialization and error-checking. For more complete versions, refer to the sample applications.

The sample applications also include links to the MapGuide documentation, but the links only work if the documentation files are visible to the web server. By default, the installation program installs the documentation in the *...\WebServerExtensions\Help* folder. If you copy or move the *Help* folder to *...\WebServerExtensions\www\Help* the documentation will be available directly from the main page of the sample applications.

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.

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/Layouts/SamplesPhp.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);
```

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 or some other method to edit the web layout resource, you can create and customize the layout, changing how it looks in a browser and what features are enabled.

Displaying the web layout requires a compatible web browser and a MapGuide 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.

The simplest way to display a web layout is to pass its resource identifier as a GET parameter to the Viewer URL. For example, the following will display a web layout using the AJAX Viewer running on `localhost`:

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

Authentication

All MapGuide sites require authentication with user id and password. If authentication succeeds, MapGuide creates a session, identified by a unique session id. This keeps the state consistent between the viewer and the server across multiple HTTP requests. Subsequent access to the site requires the session id instead of the user id. By default, the Viewer handles authentication itself, and it prompts for user id and password when it first loads. There are situations, though, where it is better to authenticate before loading the Viewer page.

One common example is a site offering read-only access to visitors. For this situation, the default MapGuide installation includes a user “Anonymous” with an empty password.

To perform authentication before the Viewer loads, embed the Viewer in another page using a `<frame>` or `<iframe>` element. The outer page can do any necessary authentication, create a session, then pass the web layout and session id to the Viewer frame.

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\MapGuideEnterprise2010\\';
$extensionsDir = $installDir . 'WebServerExtensions\www\\';
$viewerDir = $installDir . 'mapviewerphp\\';

include $viewerDir . 'constants.php';

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

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

$sessionId = $site->CreateSession();
$webLayout =
   "Library://Samples/Layouts/SamplesPhp.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* is any PHP, ASP.NET, or JSP page that makes use of the MapGuide Web API. These pages 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 pages 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 a MapGuide page 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 *MapGuide 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/Layouts/SamplesPhp.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 pages. 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 page 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.

NOTE Custom pages are loaded by the Viewer page, so a relative URL for a custom page must start at the Viewer directory, then go up one level to reach the `mapguide` directory. For example, a custom page located at `www/mapguide/samplesphp/index.php` would use the following relative URL in the web layout

```
../samplesphp/index.php
```

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

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 web layout automatically. The home task page of this layout contains links to examples that correspond to chapters in the Developer's Guide.

MapGuide Page Flow

Most MapGuide pages follow a similar processing flow. First, they create a connection with the site server using an existing session id. Then they open connections to any needed site services. The exact services required depend on the page 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 The MapGuide Viewer (page 15) for details about sending data to the Viewer.

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

The following 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.

TIP This sample is very similar to the `Hello Map` sample in the Developer's Guide samples.

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

      // Define some common locations
      $installDir =
        'C:\Program Files\Autodesk\MapGuideEnterprise2010\\';
      $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.
         $args = ($_SERVER['REQUEST_METHOD'] == "POST")
           ? $_POST : $_GET;
         $mgSessionId = $args['SESSION']
         $mgMapName = $args['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>
```

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, other pages can get an encrypted copy of the user credentials that can be used for authentication.

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 page 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 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);
```

The MapGuide Viewer 2

Introduction

MapGuide supports two ways to display maps: basic web layouts and flexible web layouts. Flexible web layouts work in all major browsers on Windows, Macintosh, and Linux. They use JavaScript and so require no browser plugins or proprietary technologies. Since they provide more flexibilty, Autodesk recommends using flexible web layouts for new development. Customizing flexible web layouts is described in Flexible Web Layouts (page 105). This chapter describes the MapGuide Viewer, which is used to display basic web layouts.

The DWF Viewer and AJAX Viewer

TIP The Hello Viewer sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

The MapGuide Viewer is a browser-based method for displaying map data in a MapGuide application. It is a complete application, with support for standard mapping functionality like zooming, theming, and selecting features. There are two different versions of the Viewer:

- DWF Viewer, which runs only within Internet Explorer, and requires a browser plug-in
- AJAX Viewer, which runs within Internet Explorer, Mozilla Firefox, and Safari, without requiring a browser plug-in

As much as possible, the two versions of the Viewer operate in the same way, so deciding which to use depends on the needs of the application. If all the end users of the application use Internet Explorer, the DWF Viewer my be an

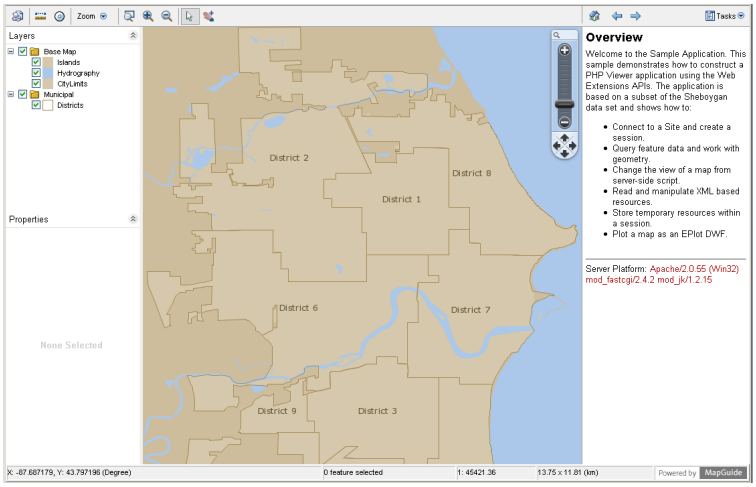
appropriate choice, but if some of them use a different browser the AJAX Viewer is a better choice.

Most MapGuide applications use a Viewer (or flexible web layouts), though it is possible to create applications that perform data analysis or manipulation but do not display anything. For example, a MapGuide application can be used as a back-end to another mapping application.

The standard Viewer displays a map along with the following optional components surrounding the map:

- Tool bar
- Layers pane
- Properties pane
- Status bar
- Task bar
- Task list (normally hidden, but available as a drop-down from the task bar)
- Task pane
- Context (right-click) menu
- Zoom slider (AJAX Viewer only)

MapGuide Viewer



The tool bar, task list, task pane, and context menu can contain a combination of pre-defined and custom MapGuide commands.

A *web layout* defines how the Viewer looks and operates for a map. One function of a web layout is to define which optional components display with the map. All of the optional components can be disabled, leaving just the map itself.

The web layout also defines any custom functionality added to the web page through custom commands.

Custom Commands

Custom commands can be of two types:

- JavaScript commands
- Web Server Extensions pages, written in PHP, ASP.NET, or JSP

JavaScript commands are defined in the web layout as commands of type Invoke Script. They are used primarily to interact with the Viewer, and can use the Viewer API.

Web Server Extensions pages can be added to the web layout in two different ways. In one method, the web layout includes a home page. This home page is loaded in the task pane when the map first displays, and can be re-loaded by clicking the Home icon in the task bar. The home page can load other pages as needed.

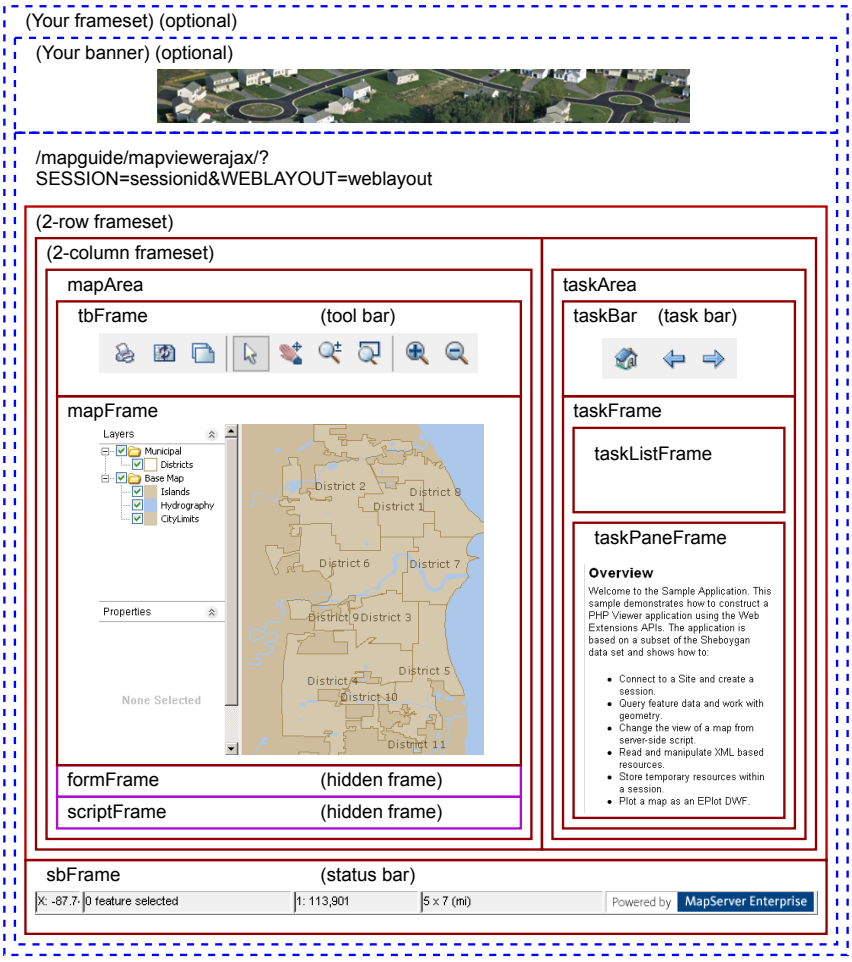
In addition, other task pages can be defined in the web layout as commands of type Invoke URL. These commands can be added to the tool bar, task list, or context menu. When a user selects one of these commands the corresponding URL is often loaded into the task pane, though it can also be loaded into a hidden frame so it is not visible.

Because Web Server Extensions pages are created at the web tier before being passed to the Viewer, they can use both the Web Server Extensions API and the Viewer API.

Understanding Viewer Frames

The MapGuide Viewers use HTML frames to divide the viewer area. Refer to the diagram below 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 frame, form frame, and script frame.
`tbFrame`	Frame containing the tool bar. Add new commands to the tool bar by modifying the web layout.
`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 to load and execute pages without them being visible to the user. This is often used for executing client-side JavaScript.
`taskArea`	Frame set containing the task bar and the task frame.
`taskBar`	Frame containing the task bar.
`taskFrame`	Frame used to hold the task list frame and the task pane frame.
`taskListFrame`	Frame used for displaying the task list. This is normally hidden, and is shown when a user clicks the task list button in the task bar. Add new commands to the task list by modifying the web layout.
`taskPaneFrame`	Frame used for displaying and executing MapGuide pages. A web layout has a default home page that displays in the task pane when the layout loads. Custom commands of type Invoke URL also load in the task pane.
`sbFrame`	Frame containing the status bar.



Viewer Frames

MapGuide Viewer API

The MapGuide Viewer API is a set of JavaScript functions used to interact with 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:

- Define an Invoke Script command in the web layout. Use this technique when you want to call the Viewer API directly from the tool bar, task list, or context menu.
- Load a page into the hidden script frame and execute the script when the page loads. Use this technique when you want the Viewer to change as a result of an action in the MapGuide page, without reloading the page.
- Execute the JavaScript call from a page loaded in the task pane frame. The JavaScript can execute when the page first loads or as a result of user interaction.

It is important to know the relationships between the frames. JavaScript executes in the context of a single frame, but it can call functions from other frames by locating them in the frame hierarchy. The following frames are children of the main Viewer frame:

- `tbFrame`
- `mapFrame`
- `formFrame`
- `scriptFrame`
- `taskFrame`

The `taskPaneFrame` is a child of the `taskFrame`.

Custom JavaScript code can execute in the context of the main frame, the script frame, or the task pane frame.

JavaScript defined as an Invoke Script command executes in the context of the main frame. To execute functions in one of the other frames, identify the function with the frame name and function name. For example, the following calls the `ZoomToView()` function of the `mapFrame` from the main frame:

```
mapFrame.ZoomToView(xLoc, yLoc, newScale, true);
```

JavaScript loaded into the `scriptFrame` must go up 1 level in the hierarchy using `parent`. For example:

```
parent.mapFrame.ZoomToView(xLoc, yLoc, newScale, true);
```

JavaScript loaded into the `taskPaneFrame` must go up 2 levels in the hierarchy using `parent.parent`. For example:

```
parent.parent.mapFrame.ZoomToView(xLoc, yLoc, newScale, true);
```

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 with an Invoke Script Command

Use this technique when you want to call the API directly from the tool bar, task list, or context menu.

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);
```

Add the button to the tool bar. When a user clicks the button, the map view repositions 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()`.

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 creating a separate page to load in the script frame and passing the necessary parameters when the page loads.

The Hello Viewer sample application contains a file named `gotopoint.php` that is designed to 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. The essential parts of `gotopoint.php` are:

```
<script language="javascript">

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

</script>

<body onLoad="OnPageLoad()">

</body>
```

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

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

NOTE This technique is also appropriate for calling the Web API without reloading the task pane. See the Modifying Maps and Layers sample for an example.

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 or as a result of an `onclick` event. 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>
```

Use a similar technique to call custom JavaScript based on an action in the task pane, like clicking a link.

Extending Map Initialization Functionality

At times, it may be necessary to perform some initialization functions when the map first loads. To accomplish this, a page loaded into the task pane can hook into the standard map initialization process.

For example, when a browser first connects to a MapGuide site, it specifies a web layout. The site uses this layout to determine which Viewer components to enable and which map to display in the map area. At the time that the task pane first loads, the map name is not yet known. It may be required for some operations, though.

The Hello Viewer Sample

The Hello Viewer sample, installed with the Developer's Guide samples, shows simple examples of using the Viewer API from different parts of a web layout.

The tool bar for the sample contains a custom Invoke Script command that calls the `ZoomToView()` function of the `mapFrame`. This is executed in the context of the main frame, so the function is available using

```
mapFrame.ZoomToView()
```

The task pane loads a page that shows two other ways of calling `ZoomToView()`. One way loads a custom page into the hidden `scriptFrame`. The page reads `GET` parameters and passes them to the JavaScript function call. This is executed in the context of the `scriptFrame`, so `ZoomToView()` is available using

```
parent.mapFrame.ZoomToView()
```

Another way calls `ZoomToView()` directly when a link is clicked, using the JavaScript `onclick` event. This is executed in the context of the `taskPaneFrame`, so `ZoomToView()` is available using

```
parent.parent.mapFrame.ZoomToView()
```

The Developer's Guide samples also demonstrate a more advanced method for using JavaScript in a Viewer. The file *index.php* includes an external JavaScript file that solves 2 problems:

- When a map is first loading, the task pane displays before the map has been fully initialized. This can cause problems if users click any links in the task pane that depend on the map being available.
- The first time the Viewer loads a page into the task pane, it passes `SESSION` and `WEBLAYOUT` as `GET` parameters. The map name is not known until after the web layout has loaded. When a user clicks the Home button, the Viewer reloads the home page in the task pane, but passes `SESSION` and `MAPNAME` as `GET` parameters instead. In some cases, it may be useful for the home page to have the map name when it first loads.

To deal with these problems, the Hello Viewer sample loads *pageLoadFunctions.js*, which attaches a function to the `window.onload` event of the page in the task pane. This function does the following:

- Replaces the `OnMapLoaded()` function of the main frame. This function is called after the map has been fully initialized. The new version performs some initialization (see below), then calls the original `OnMapLoaded()`.
- Saves the contents of the task pane page and replaces it with the text "Loading...".
- After the map is fully initialized, it calls the new version of `OnMapLoaded()`. At this point, the map name is known, and is available from the `mapFrame.GetMapName()` function. The new version of `OnMapLoaded()` restores the contents of the task pane page, then it searches all `<a>` elements, replacing "`MAPNAME=unknown`" with the correct map name in the `href` attributes.

See the Hello Viewer sample for links to view *index.php* and *pageLoadFuctions.js*.

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. The Developer's Guide samples use this technique. The main page for the samples, *main.php*, creates 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. The following code contains the important parts of *main.php*.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<?php
require_once('common/common.php');

try
{
  // Initialize the web extensions,
  MgInitializeWebTier ($webconfigFilePath);

  // Connect to the site server and create a session
  $userInfo = new MgUserInformation("Author", "author");
  $site = new MgSite();
  $site->Open($userInfo);
}
catch (MgException $e)
{
  echo "Could not connect to the MapGuide site server.";
    die();
}

try
{
  $sessionId = $site->CreateSession();

  // Define some constants
  $webLayout = "Library://Samples/Layouts/SamplesPHP.WebLayout";
  $title = "Samples";
}
catch (MgException $e)
{
  echo "ERROR: " . $e->GetMessage("eng") . "\n";
  echo $e->GetStackTrace("eng") . "\n";
}
?>

<html>
  <head>
    <title><?= $title ?></title>
  </head>

  <frameset rows="110,*">
    <frame src="common/Title.php?TitleText=<?= $title ?>"
```

```
      name="TitleFrame" scrolling="NO" noresize />
    <frame
    src="/mapguide/mapviewerajax/?
    SESSION=<?= $sessionId ?>&
    WEBLAYOUT=<?= $webLayout ?>" name="ViewerFrame" />
  </frameset>
</html>
```

Interacting With Layers 3

Overview of Layers

TIP The Interacting With Layers sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

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 31) 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 control 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 MapGuide Studio. They can also be created and modified dynamically using the Web Extensions API. See Modifying Maps and Layers (page 57) 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
On	Off	10000	Off
Off	On	10000	Off

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 57).

Changing Basic Properties

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

Example

The following example toggles the 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 57) 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 4

Overview of Features

TIP The Working With Feature Data sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

Understanding features is fundamental to being able to use the Autodesk 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 39) for details about selection.

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

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

Selecting with the Web API

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

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

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

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

Basic Filters

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

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

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

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

Examples

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

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

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

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

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

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

Spatial Filters

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

NOTE For more information about geometry, see Representation of Geometry (page 73).

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 Sheboygan sample data.

```
$districtQuery = new MgFeatureQueryOptions();
$districtQuery->SetFilter("Autogenerated_SDF_ID = 1");

$layer = $map->GetLayers()->GetItem('Districts');
$featureReader = $layer->SelectFeatures($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: Selection

The following example creates a selection, then lists properties from the selected features. See the Working With Feature Data sample, in the Developer's Guide samples, for the complete version.

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

```
$map = new MgMap($siteConnection);
$map->Open($mapName);

// Get the geometry for the boundaries of District 1

$districtQuery = new MgFeatureQueryOptions();
$districtQuery->SetFilter("Autogenerated_SDF_ID = 1");

$layer = $map->GetLayers()->GetItem('Districts');
$featureReader = $layer->SelectFeatures($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.

$layer = $map->GetLayers()->GetItem('Parcels');
$featureReader = $layer->SelectFeatures($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>';
```

Active Selections

A map may have an active selection, which is a list of features on the map that have been selected and 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 48) 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. Use `taskPaneFrame` or `scriptFrame` as the target frame, depending whether the loaded page should be visible or not.

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 a small 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 only):

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. Note that this will also work for the AJAX Viewer.

To retrieve and manipulate the active selection for a map (AJAX or 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 `formFrame`. 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 or 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 or as a result of an `onClick` event in the task pane. For example, the task pane of the Working With Feature Data sample contains a JavaScript function executed by an `onClick` event.

```
function listSelection()
{
  xmlSel = parent.parent.mapFrame.GetSelectionXML();
  params = new Array("SESSION",
    parent.parent.mapFrame.GetSessionId(),
    "MAPNAME", parent.parent.mapFrame.GetMapName(),
    "SELECTION", xmlSel);
  pageUrl =
    "/mapguide/samplesphp/working_with_feature_data/
    listselection.php";
  parent.parent.formFrame.Submit(pageUrl, params,
    "taskPaneFrame");
}
```

This submits a request to `listselection.php`, which contains the following:

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

// ------------------------------------------------------------
// Use the following code for AJAX or DWF Viewers
// This requires passing selection data via HTTP POST

if (isset($_POST['SELECTION']) && $_POST['SELECTION'] != '')
{
  $selection = new MgSelection($map, $_POST['SELECTION']);
  $layers = $selection->GetLayers();
}
else
$layers = 0;
// ------------------------------------------------------------

if ($layers)
{
  $queryOptions = new MgFeatureQueryOptions();
  for ($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

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

Example: Listing Selected Parcels (AJAX Viewer only)

To simplify the page in Example: Listing Selected Parcels (AJAX or DWF Viewer) (page 50) so it runs in the AJAX Viewer only, replace the section of code that reads the selection data from the HTTP POST parameter:

```
if (isset($_POST['SELECTION']) && $_POST['SELECTION'] != '')
{
  $selection = new MgSelection($map, $_POST['SELECTION']);
  $layers = $selection->GetLayers();
}
else
$layers = 0;
```

with the following, which retrieves the selection from the run-time map state:

```
// ------------------------------------------------------
// Use the following code for AJAX Viewers only.
// This does not require passing selection data via HTTP POST.
//
$selection = new MgSelection($map);
$selection->Open($resourceService, $mapName);
$layers = $selection->GetLayers();
```

There is no need to create a JavaScript function to call this page using the `Submit()` method of the `formFrame`. It can be run directly as a link from the calling page, passing just the `SESSION` and `MAPNAME`.

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 39). 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: Selection (page 43).

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.

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

  <h1 class="AppHeading">Select features</h1>

  <?php
  include '../common/common.php';

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

  try
  {

    // Initialize the Web Extensions and connect to the Server
    // using the Web Extensions session identifier

    MgInitializeWebTier ($webconfigFilePath);

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

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

    // Get the geometry for the boundaries of District 1

    $districtQuery = new MgFeatureQueryOptions();
    $districtQuery->SetFilter("Autogenerated_SDF_ID = 1");

    $layer = $map->GetLayers()->GetItem('Districts');
    $featureReader = $layer->SelectFeatures($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 the selection as XML.

    $layer = $map->GetLayers()->GetItem('Parcels');
    $featureReader = $layer->SelectFeatures($queryOptions);

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

    echo 'Selecting parcels owned by Schmitt in District 1';
  }
  catch (MgException $e)
  {
    echo $e->GetMessage();
    echo $e->GetDetails();
  }
  ?>

</body>

<script language="javascript">

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

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

```
  }

</script>
```

Modifying Maps and Layers 5

Introduction

TIP The Modifying Maps and Layers sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

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 68).

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 MapGuide 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 MapGuide 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 MapGuide Studio's Layer Editor, as described in the MapGuide Studio Help.

This example

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

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

The page then goes on to save the XML to a resource and loads that resource into the map, as described in Adding Layers To A Map (page 68).

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 68).

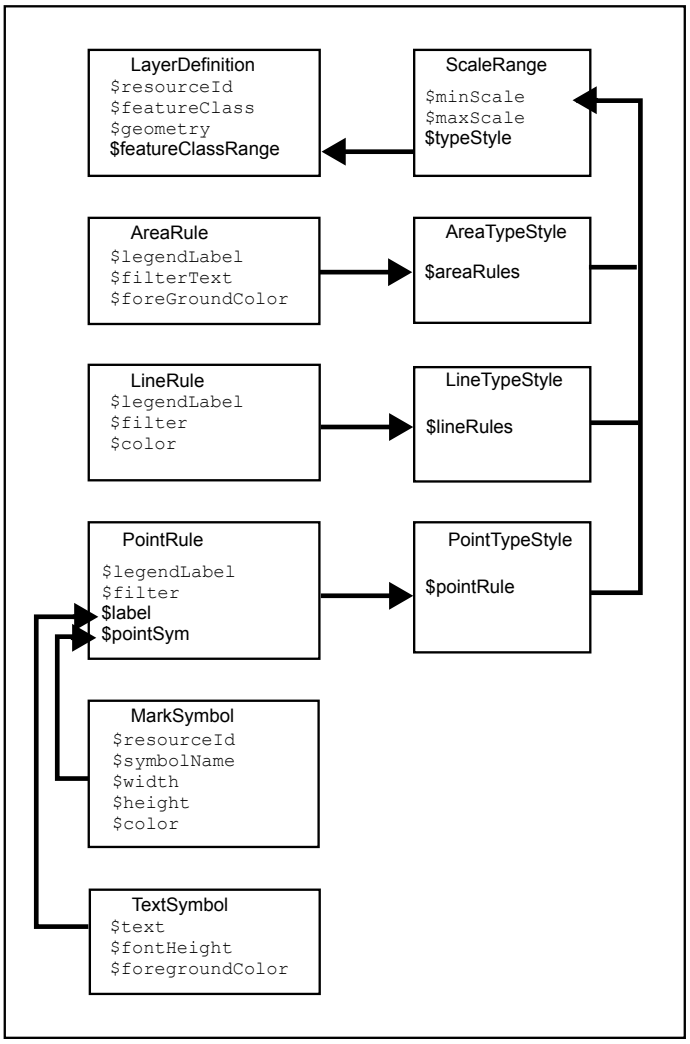
Another Way To Create Layers

The method described in the previous section is easy to use, but requires a layer definition be created first through the MapGuide 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 MapGuide Studio UI, or modify the XML of the layer definition.)

The `layerdefinitionfactory` is only available for PHP. For development using ASP.NET, a good alternative is to use the Visual Studio tool *xsd.exe* to generate .NET classes for the `LayerDefinition` schema.



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

Function	Parameter	Description
	`$featureClass`	The feature class to use. For example, `SHP_Schema:Parcels`. Equivalent to the Feature class field in MapGuide Studio's layer editor.
	`$geometry`	The geometry to use from the feature class. For example, `SHPGEOM`. Equivalent to the Geometry field in MapGuide Studio's layer editor.
	`$feature-ClassRange`	A scale range created by filling in a scale range template (`ScaleRange.templ`).
`CreateScaleRange()`	`$minScale`	The minimum scale range to which this rule applies. Equivalent to the From field in MapGuide Studio's layer editor.
	`$maxScale`	The maximum scale range to which this rule applies. Equivalent to the To field in MapGuide Studio's layer editor.
	`$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 Legend Label field in MapGuide Studio's layer editor.
	`$filterText`	The filter expression that determines which features match this rule. For example, `SQFT >= 1 AND SQFT < 800`. Equivalent to the Condition field in MapGuide Studio's layer editor.
	`$foreGroundColor`	The color to be applied to areas that match this rule. Equivalent to the Foreground color field in MapGuide Studio's layer editor.

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

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

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

Example - Creating A Layer That Uses Area Rules

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

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

/// Create three area rules for three different
// scale ranges.
$areaRule1 = $factory->CreateAreaRule(    '1 to 800',
  'SQFT &gt;= 1 AND SQFT &lt; 800',     'FFFFFF00');
$areaRule2 = $factory->CreateAreaRule( '800 to 1600',
  'SQFT &gt;= 800 AND SQFT &lt; 1600',  'FFFFBF20');
$areaRule3 = $factory->CreateAreaRule('1600 to 2400',
  'SQFT &gt;= 1600 AND SQFT &lt; 2400', 'FFFF8040');

// Create an area type style.
$areaTypeStyle = $factory->CreateAreaTypeStyle(
  $areaRule1 . $areaRule2 . $areaRule3);

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

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

The script then saves the XML to a resource and loads that resource into the map. See Adding Layers To A Map (page 68).

Example - Using Line Rules

Creating line-based rules is very similar.

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

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

// Create a line type style.
$lineTypeStyle = $factory->
  CreateLineTypeStyle($lineRule);

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

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

Example - Using Point Rules

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

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

    // Create a mark symbol
    $resourceId =
'Library://Samples/Sheboygan/Symbols/BasicSymbols.SymbolLibrary';
    $symbolName = 'PushPin';
    $width = '24';  // points
    $height = '24'; // points
    $color = 'FFFF0000';
    $markSymbol = $factory->CreateMarkSymbol($resourceId,
      $symbolName, $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
require_once('../common/common.php');

//////////////////////////////////////////////////////////
function add_layer_definition_to_map($layerDefinition,
  $layerName, $layerLegendLabel, $mgSessionId,
  $resourceService, &$map)
// Adds the layer definition (XML) to the map.
// Returns the layer.
{
    // Validate the XML.
    $domDocument = new DOMDocument;
    $domDocument->loadXML($layerDefinition);
    if (! $domDocument->schemaValidate(
       "$schemaDirectory\LayerDefinition-1.1.0.xsd") )
    {
        echo "ERROR: The new XML document is invalid.
          <BR>\n.";
        return NULL;
    }
    // Save the new layer definition to the session
    // repository
    $byteSource = new MgByteSource($layerDefinition,
      strlen($layerDefinition));
    $byteSource->SetMimeType(MgMimeType::Xml);
    $resourceID = new MgResourceIdentifier(
       "Session:$mgSessionId//$layerName.LayerDefinition");
    $resourceService->SetResource($resourceID,
      $byteSource->GetReader(), null);
    $newLayer = add_layer_resource_to_map($resourceID,
      $resourceService, $layerName, $layerLegendLabel,
      $map);
    return $newLayer;
}
```

This function adds a layer resource to a map.

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

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

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

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

Making Changes Permanent

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

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

```
$byteSource = new MgByteSource($layerDefinition, strlen($layerDefin
ition));
$byteSource->SetMimeType(MgMimeType::Xml);
$resourceId =
  new MgResourceIdentifier("Library://LayerName.LayerDefinition");
$resourceService->SetResource(
  $resourceId, $byteSource->GetReader(), null);
```

Analyzing Features 6

Introduction

TIP The Analyzing Features sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

Autodesk 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

Autodesk 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.
- Autodesk 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 Autodesk MapGuide internal representation, use an `MgWktReaderWriter` object. Call `MgWktReaderWriter.Read()` to convert AGF text to `MgGeometry`. Call `MgWktReaderWriter.Write()` to convert `MgGeometry` to AGF text.

To convert between binary AGF and the Autodesk MapGuide internal representation, use an `MgAgfReaderWriter` object. Call `MgAgfReaderWriter.Read()` to convert binary AGF to `MgGeometry`. Call `MgAgfReaderWriter.Write()` to convert `MgGeometry` to binary AGF.

For example, if you have a WKT representation of the geometry, you could create a geometry object as follows:

```
MgWktReaderWriter wktReaderWriter = new MgWktReaderWriter();
MgGeometry geometry = wktReaderWriter.Read(wktGeometry);
```

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 39). 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.

Autodesk 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 an `MgCoordinateSystemFactory` and 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();

$coordSysFactory = new MgCoordinateSystemFactory();
$xform = $coordSysFactory->GetTransform($srs1, $srs2);
$geometry1xform = $geometry1->Transform($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 = $srs->GetMeasure();
$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 Autodesk 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 57).

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 >= 0");
$commands->Add($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 38) 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 create a buffer, get the geometry of the feature to be buffered. If the feature is being processed in an `MgFeatureReader` as part of a selection, this requires getting the geometry data from the feature reader and converting it to an `MgGeometry` object. For example:

```
$geometryData =
  $featureReader->GetGeometry($geometryName);
$featureGeometry = $agfReaderWriter->Read($geometryData);
```

If the buffer is to be calculated using coordinate system units, create an `MgCoordinateSystemMeasure` object from the coordinate system for the map. For example:

```
$mapWktSrs = $currentMap->GetMapSRS();
$coordSysFactory =
  new MgCoordinateSystemFactory();
$srs = $coordSysFactory->Create($mapWktSrs);
$srsMeasure = $srs->GetMeasure();
```

Use the coordinate system measure to determine the buffer size in the coordinate system, and create the buffer object from the geometry to be buffered.

```
$srsDist =
  $srs->ConvertMetersToCoordinateSystemUnits($bufferDist);
$bufferGeometry =
  $featureGeometry->Buffer($srsDist, $srsMeasure);
```

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 48). 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 = $srs->GetMeasure();

// 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 >= 0"));

  $featureService->UpdateFeatures($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,
    $bufferFeatureResId, $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));
}

$results = $featureService->UpdateFeatures($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 57).

```
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->getElementsByTagName(
```

```
      '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 7

Introduction

TIP The Digitizing and Redlining sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

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 `task_pane_digitizing.php` in the `digitizing_features` directory 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 `DigitzePoint()` 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. `DigizitzePoint()` 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 `task_pane_redlining.php` in the `digitizing_features` directory 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 with the Viewer API, using the `Submit()` method of the `formFrame`.

```
function OnLineDigitized(line) {
  // Send the Javascript variables to 'draw_line.php',
  // via the form frame
  var params = new Array("x0", line.Point(0).X,
    "y0", line.Point(0).Y,
    "x1", line.Point(1).X,
    "y1", line.Point(1).Y,
    "SESSION", "<?= $sessionId ?>",
    "MAPNAME", "<?= $mapName ?>");
  parent.parent.formFrame.Submit(
    "/mapguide/samplesphp/digitizing_features/draw_line.php",
    params, "scriptFrame");
}
```

This submits the coordinates to the server-side function to draw the line. It uses the hidden `scriptFrame` so the page output is not visible.

Creating a Feature Source

The next step is create a feature source

See `draw_line.php` in the `digitizing_features` directory 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("spatial context",
    $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 68).

Custom Output 8

Introduction

TIP The Custom Output sample, in the Developer's Guide samples, demonstrates concepts from this chapter.

Autodesk 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 Autodesk 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 Autodesk Design Review. 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 Autodesk MapGuide DWF Viewer are available, such as the ability to turn layers on and off.
- The ePlot requires the Autodesk Design Review, either standalone or as a plug-in for Internet Explorer.
- Images can be copied to the Windows clipboard and used in other applications.
- Autodesk Design Review is a free program that is only available on Windows. Visit *http://www.autodesk.com* to download.

Characteristics of an eMap DWF:

- An eMap DWF is dynamic, with all the zooming and panning capabilities of the Autodesk 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 Design Review, either standalone or as a plug-in for Internet Explorer.
- The eMap requires an active Autodesk MapGuide session. If the session times out the map cannot be displayed.
- Individual views from an eMap DWF can be saved as ePlot DWFs.
- Autodesk Design Review is a free program that 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 Autodesk 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;
```

Flexible Web Layouts 9

Introduction

Flexible web layouts are an alternative to the MapGuide viewers described in The MapGuide Viewer (page 15). They work in major browsers on Windows, Macintosh, and Linux, including Internet Explorer and Mozilla Firefox. They use JavaScript and so require no browser plugins or proprietary technologies.

This chapter assumes you are familiar with configuring flexible web layouts through MapGuide Studio, which provides a user interface for routine configuration. This chapter describes some of the ways you can extend their default functionality.

Related Technologies

Flexible web layouts make use of additional libraries and technologies for some functionality. At times it may be useful to refer to documentation for these libraries. See the following for more information:

- Fusion, the open source technology used by Flexible Web Layouts.
  http://trac.osgeo.org/fusion/wiki
- OpenLayers, a JavaScript library for displaying map data in a browser.
  http://www.osgeo.org/openlayers
- Jx, a JavaScript library for building UI components like dialogs.
  http://jxlib.org

Creating Templates

When you create a flexible web layout in MapGuide Studio, you can base it on one of a number of templates, such as the Aqua template. These templates work for many applications and can be customized when needed. However, if you need to create a new template, this section describes how to do so.

A template for a flexible web layout contains various interrelated pieces that contribute to how the template looks and what it does. All the files for a given template are stored in the folder *web_server_extensions*\www\fusion\templates\mapguide. Some important pieces are described in the following table.

Name	Description
Template Info XML file	Contains information about the template and a list of the panels available for the template. These correspond to the tabs shown in MapGuide Studio on the Configure Components of the Selected Template section of the Web Layout Editor. The template XML file has the same name as the template. For example, the Aqua template uses `aqua.xml`. **NOTE** The template XML file is used internally by MapGuide Studio. It is not required for displaying a flexible web layout in a browser. MapGuide Studio saves any necessary data from the template XML file into the application definition.
ApplicationDefinition	Generated by MapGuide Studio and stored in the site repository. The application definition defines which components (menus and widgets) are available in a given template. The application definition uses the ApplicationDefinition.xsd schema. **NOTE** The default templates each contain a file named *ApplicationDefinition.xml*. This is an alternate location for the application definition. See Application Definitions (page 113) for details.
index.html	Home page for displaying a flexible web layout in a browser. The index.html file defines the arrangement and operation of the components from the application definition. For example, the selection panel can be docked or floating, and

Name	Description
	can be initially hidden or visible, depending on how it is initialized in index.html.
CSS files	Style definitions for the template.
Image files	Buttons, icons, and related items.

The simplest way to understand the files in a flexible web layout is to experiment with the existing templates. Follow the instructions below to create a new template or a copy of an existing template.

1. Create a sub-folder below the *web_server_extensions*\www\fusion\templates\mapguide folder (where *web_server_extensions* is the directory where the MapGuide web server extensions are installed, typically C:\Program Files\Autodesk\MapGuideEnterprise2010\WebServerExtensions). Give this new folder the name of your template, for example HelloMap.
2. Copy the CSS files from an existing template to your template's folder, or create new ones.
3. Copy the images directory from an existing template to your template's folder, or create your own icons.
4. Create a preview graphic for your template and store in your template's folder. This graphic will be shown in the list of templates when you create a flexible web layout in MapGuide Studio. The name of the graphic is arbitrary. The dimensions should be 126 x 96 pixels. You will use this name later when you create a TemplateInfo file.
5. In the template's folder, create an HTML file called index.html. This file defines the layout of the template.
   - Use a valid, strict doctype for your template pages. If you omit the doctype, use an invalid doctype, or use a transitional doctype, most browsers will revert to "Quirks Mode." The layout will appear to function correctly, but you may notice some minor issues and your application may appear differently depending on the browser. By using a valid HTML or XHTML doctype, browsers will use "Standards Compliant Mode" and your application will work consistently between browsers.
   - In the <Head> section, import the CSS file(s).
     For example:

```
<style type="text/css">
  @import url(template.css);
</style>
```

This is important, as the default CSS styles set all components to be initially invisible.

- In the `<Head>` section, include the `fusion.js` library.
  For example:

```
<script type="text/javascript"
  src="../../../lib/fusion.js">
</script>
```

  Make sure that the `src` of the script tag that points to `fusion.js` is a valid path. It can be relative (as in the example above) or absolute (starting with `http://`). If it is absolute, then the URL must be pointing to the same server as the URL you use to load the application.

- In the `<Head>` section, define a `window.onload` function. In it, use JavaScript and the Jx library to position the elements of the layout including the components. Also, register for the `FUSION_ERROR` and `FUSION_INITIALIZED` events.

  For example:

```
<script type="text/javascript">
  window.onload = function() {
    Fusion.initializeLocale();

    var main = new Jx.Layout('AppContainer',
      {left: 0, right: 0, top: null, bottom: null});
    new Jx.Layout('Help', {height: 25, left: 0,
      right: 0, top: 34, bottom: null});
    new Jx.Layout('Map', {width: null, height: null,
      left: 25, right: 0, top: 59, bottom: 21});
    main.resize();
    $('AppContainer').style.visibility = 'visible';

    Fusion.registerForEvent(
      Fusion.Event.FUSION_ERROR, fusionError);
    Fusion.registerForEvent(
      Fusion.Event.FUSION_INITIALIZED, fusionInitialized);
    Fusion.initialize();
  }
</script>
```

  For more information on the Fusion object, see Fusion API (page 122).

For more information on events, see Events (page 124).

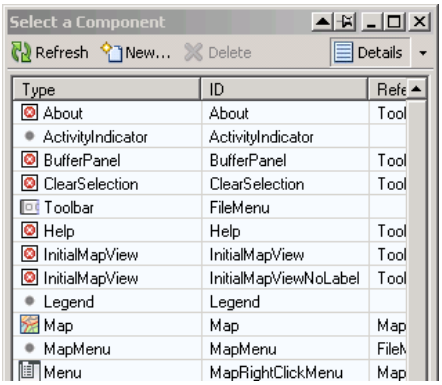
- In the `<Head>` section, define the `fusionError` and `fusionInitialized` functions. For example:

```
<script type="text/javascript">
  var fusionError = function(eventId, error) {
    console.log('Fusion Error: \n' + error.toString());
}

var fusionInitialized = function() {
}
</script>
```

  For an example of a template where the `fusionInitialized` function is used, see *web_server_extensions*`\www\fusion\templates\mapguide\aqua\index.html`.

- In the `<Body>` section, insert `<div>` elements to determine which components will appear on the template. Any HTML element with an ID that matches a component ID will be replaced with that component. For example, `<DIV id="Help">` will be replaced with the Help component. And `<DIV id="Toolbar">` will be replaced with a toolbar; the components that appear on this toolbar will be configured through MapGuide Studio. To see the available IDs, click Attach Component while editing a flexible web layout in MapGuide Studio.



  Components will be inserted in any DOM container object with a matching ID, not just in `<DIV>`s. So you can also use Jx methods, such as `Jx.Panel` and `Jx.Layout` to structure your page. For example, this will create an object that will have a legend inserted:

```
var p1 = new Jx.Panel({label: 'Legend'});
```

If there is no matching component for the ID, then when the user configures the flexible web layout, MapGuide Studio will display an empty tab for it (one with no components added). The user can then add components to it through MapGuide Studio. The ID will be used for the tab's title, so use IDs that will help the user understand what he or she is configuring.

- So in this example, the complete `index.html` file would look like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Hello Map</title>

  <meta http-equiv="Content-type"
    content="text/html; charset=UTF-8">

  <style type="text/css">
    @import url(template.css);
  </style>

  <script type="text/javascript"
    src="../../../lib/fusion.js"></script>

  <script type="text/javascript">
    window.onload = function() {
      Fusion.initializeLocale();

      var main = new Jx.Layout('AppContainer',
        {left: 0, right: 0, top: null, bottom: null});
      new Jx.Layout('Help', {height: 25, left: 0,
        right: 0, top: 34, bottom: null});
      new Jx.Layout('Map', {width: null, height: null,
        left: 25, right: 0, top: 59, bottom: 21});
      main.resize();
      $('AppContainer').style.visibility = 'visible';

      Fusion.registerForEvent(
        Fusion.Event.FUSION_ERROR, fusionError);
      Fusion.registerForEvent(
       Fusion.Event.FUSION_INITIALIZED, fusionInitialized);
      Fusion.initialize();
    }
```

```
    var fusionError = function(eventId, error) {
      console.log('Fusion Error: \n' + error.toString());
    }

    var fusionInitialized = function() {
    }

  </script>
</head>

<body>
  <div id="AppContainer" style="visibility: hidden;">
    <div id="Help"></div>
    <div id="Map"></div>
  </div>
</body>
</html>
```

6 Create a `<TemplateInfo>` XML file based on the `ApplicationDefinitionInfo` XML schema. (The XML schemas are installed in *`mapguide_server`*`\Schema` (where *`mapguide_server`* is the installation directory of the MapGuide server, typically `C:\Program Files\Autodesk\MapGuideEnterprise2010\Server`).

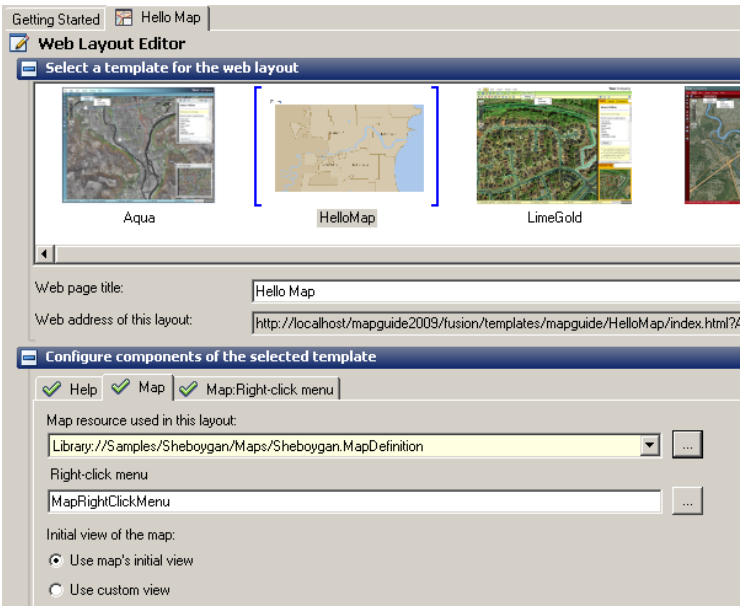
- Name the file with the name of your template plus `.xml`, for example, `HelloMap.xml`.
- Store it in the parent folder of the template. In other words, store it in *`web_server_extensions`*`\www\fusion\templates\mapguide`.
- In the `LocationUrl` element, point to the location of your template's `index.html` file.
- In the `PreviewImageUrl` element, use the name of the preview graphic you created earlier.
- Create a `<Panel>` element for each component (such as a toolbar) in the template. Each panel here will be shown as a tab when the template is configured in MapGuide Studio.
  The order of the `<Panel>`s does not matter; the layout is controlled through the `index.html`file.

  For the HelloMap example, the TemplateInfo file is:

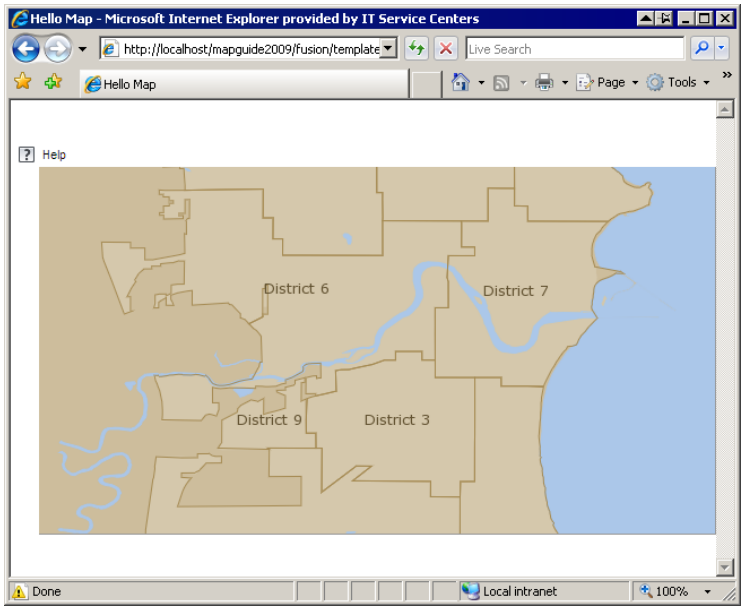
  ```
  <TemplateInfo>
  ```

```
    <Name>HelloMap</Name>
    <LocationUrl>
      fusion/templates/mapguide/HelloMap/index.html
    </LocationUrl>
    <Description>HelloMap template</Description>
    <PreviewImageUrl>
      fusion/templates/mapguide/HelloMap/preview.png
    </PreviewImageUrl>
    <Panel>
      <Name>Help</Name>
      <Label>Help</Label>
      <Description>The help button</Description>
    </Panel>
    <Panel>
      <Name>Map</Name>
      <Label>Map</Label>
      <Description>The main map display</Description>
    </Panel>
  </TemplateInfo>
```

The template can now be used in MapGuide Studio just like any of the default templates.



If you create a flexible web layout based on this template, it will look like this:



NOTE If you edit a template's `index.html` file, right-click the preview image in MapGuide Studio and choose Refresh to see the changes. If you change any of the other template files, such as modifying or adding a `<Panel>` element, also right-click on one of the panel tabs and choose Refresh Panels to update the panel definitions.

Application Definitions

An application definition defines the components of a flexible web layout and how they are arranged. When a flexible web layout loads in a browser it uses the application definition to set up the menus and dialogs. An application definition conforms to the `ApplicationDefinition.xsd` schema.

MapGuide Studio is the preferred method for creating an application definition because its user interface manages the complexity of the underlying XML structure. Using MapGuide Studio it is simple to add new widgets to an existing flexible web layout and rearrange or modify existing widgets. The resulting application definition is stored as an `ApplicationDefinition` resource in the Autodesk MapGuide site repository.

It is possible, however, to store an application definition in an external form, outside the site repository. Normally, when a flexible web layout loads in a browser, it checks for an application definition resource name passed as parameter to `index.html`. If this exists, then the application definition is loaded from the site repository. For example, the layout used in Flexible Web Layouts Examples (page 127) passes the following as a `GET` parameter in the URL:

```
?ApplicationDefinition=Library%3a%2f%2fSamples%2f
FlexibleWebLayouts%2fExamples.ApplicationDefinition
```

However, if the application definition path is not passed to `index.html` then the flexible web layout looks for the file `ApplicationDefinition.xml`, located in the same folder as `index.html`.

The default templates all include an `ApplicationDefinition.xml` file. This should be used an an example only, and does not necessarily represent the contents of an `ApplicationDefinition` resource stored in the site repository. Changing the flexible web layout using MapGuide Studio will change the `ApplicationDefinition` resource but will not have any effect on the external `ApplicationDefinition.xml` file.

Creating Components

MapGuide Studio comes with a set of components that meet many customer's needs, but may not meet the needs of advanced developers. This is a brief overview of components. For more detailed examples see Flexible Web Layouts Examples (page 127).

Components are of two types: containers and widgets. Menus, toolbars, and splitter bars are containers, and are used to hold other components. Widgets are functional pieces, usually self-contained. For example, the map is a widget, as are items like the selection panel and the measure tool.

Widgets require additional files that define what the widget does. Depending on the function of the widget, it may require some or all of the following files:

- JavaScript file with code to initialize and operate the widget
- XML file for use by MapGuide Studio, describing the widget and any parameters it may require
- Additional JavaScript code in other files, such as `index.html` for the template. See Example 3: Dialogs and Events (page 135) for an example.

- PHP files to implement functionality on the web tier. See Example 4: Updating the Site Repository (page 140) for an example.

The Map Component

The Map component is the primary interface between the application and the spatial data represented by the map. Most components either display information about the map or allow the user to interact with the map in some way.

The Map component supports the following API:

- `loadMap(`*`mapDefinition`*`)` – causes the Map component to load the specified MapDefinition.
- `reloadMap()` – causes the Map component to reload the current MapDefinition. This is necessary when the map state has changed in certain ways (adding or removing layers in the map, for instance) and is primarily an internal function.
- `setExtents(`*`minx, miny, maxx, maxy`*`)` – used to set the map extents to a particular bounding box programmatically.
- `drawMap()` – used to render a map image and load it in the browser. Normally, this is called automatically as required, but occasionally it may be required to be called programmatically when the state of the map has changed on the server without the knowledge of the Map component.
- `query(`*`options`*`)` – used to query the Map in some way and create (or modify) a selection. The *`options`* argument is a JavaScript object that can contain the following properties:
  - *`geometry`* – a WKT string containing a geometry that defines the spatial area to be queried. The default value is to not limit the query to a spatial extent.
  - *`maxFeatures`* – an integer value that determines the maximum number of features to be returned. A value of -1 means all features. The default value is -1.
  - *`persistent`* – a boolean value that determines whether the query results should be saved as a visual selection on the map, or not. The default value is true.

- *selectionType* – a string value that determines how features are selected with relation to the geometry. The value can be:
  - INTERSECTS – the feature is selected if any part of the geometry and the feature intersect. This is the default value.
  - CONTAINS – the feature is selected if the geometry contains the feature
- *filter* – a string value that represents a valid FDO SQL `where` clause that is used to select features based on attribute values. This may be combined with a geometry value. The default value is no filter.
- *layers* – an array of layer names that are to be queried. If no layer names are provided, then all layers will be queried. The default is to query all layers.
- *extendSelection* – a boolean value that controls whether the results of this query will be added to the current persistent selection or will replace the current persistent selection.

- `getSessionId()` – returns the current session ID.
- `hasSelection()` See Events (page 124).
- `getSelection(`*callback*`)` See Events (page 124).
- `clearSelection()` See Events (page 124).

A Map component can have a default MapDefinition that is automatically loaded when the application is loaded. But it is not mandatory to specify a default map. When no default map is specified, the Map component is still initialized. The MapDefinition will then be loaded in response to another component (such as the MapMenu component) or some application-specific code. Regardless of how it happens, when a MapDefinition has been loaded, the Map component will trigger a MAP_LOADED event. Most components are not useful if there is no map loaded, so they use the MAP_LOADED event to determine when they should be enabled. This means that most components will appear initially disabled until the map has been loaded. There are some notable exceptions, including the Map Menu component which is used to provide a drop-down menu of MapDefinitions that the user can pick from.

Once the Map is loaded, the following events may be triggered:

- MAP_SESSION_CREATED. The Map component is responsible for creating and maintaining a session with the server. When the session has been created,

this event is triggered. Nothing can happen until this event has been triggered.

- MAP_LOADING. The Map component triggers this event when it is starting to load a new Map. This is primarily used by components to prepare themselves for the new map by discarding their current state and temporarily disabling themselves.
- MAP_LOADED. The Map component triggers this event when a map has been loaded and is ready.
- MAP_EXTENTS_CHANGED. The Map component triggers this event for any navigation that changes the current extents.
- MAP_BUSY_CHANGED. The Map component maintains a reference count of asynchronous events as they start and finish. An application can use this event to display a processing image so that the user is aware that some asynchronous activity is happening.
- MAP_RESIZED. The Map component triggers this event when the size of the map is changed.
- MAP_SELECTION_ON. The Map component triggers this event when a new selection has been created.
- MAP_SELECTION_OFF. The Map component triggers this event when the current selection has been cleared.
- MAP_ACTIVE_LAYER_CHANGED. The Map component allows for a single layer to be marked as active by the application. This event is triggered when the active layer is changed.
- MAP_GENERIC_EVENT. Most components rely directly on their Map component for everything. In some cases, though, components need to be informed of changes in other components. In these cases, the Map component can act as a broker for events through the MAP_GENERIC_EVENT. Components that employ the MAP_GENERIC_EVENT normally do so for a specific internal purpose, and the application should not normally register for this event.

Working With Selections

There are several components that allow the user to interactively select features on the Map. MapGuide takes care of updating the Map image with the current

selection if necessary, but does not display attributes of the selected features to the user. That is up to the application.

Regardless of how the features are selected, the Map component provides the API for an application to retrieve and work with the user's selection. There are two events that can be used by an application to know when the user selection has changed:

- `MAP_SELECTION_ON`. The Map component triggers this event when a new selection has been created.
- `MAP_SELECTION_OFF`. The Map component triggers this event when the current selection has been cleared.

When the application receives a `MAP_SELECTION_ON` event from the Map component, it can use the following functions to work with the selection:

- `hasSelection()` returns a boolean value which indicates if there is currently a selection or not.
- `getSelection(`*`callback`*`)` retrieves the current selection. Retrieving the selection is potentially an asynchronous operation and so the callee provides a callback function that is called when the selection is ready. The callback function is passed a single argument, a Selection object, described below.
- `clearSelection()` is used to clear the current selection. This removes the selection from the map and invalidates the current selection object.

For an example, see Example 2: Selections (page 133)

An application will typically call `getSelection()` in response to the `MAP_SELECTION_ON` event. Typical code for this might look like:

```
window.onload=function() {
  // ...

  Fusion.registerForEvent(
    Fusion.Event.FUSION_ERROR,
    fusionError);
  Fusion.registerForEvent(
    Fusion.Event.FUSION_INITIALIZED,
    fusionInitialized);
  Fusion.initialize();
}

var theMap;
function fusionInitialized() {
  theMap = Fusion.getWidgetById('Map');
  theMap.registerForEvent(Fusion.Event.MAP_SELECTION_ON,
    OpenLayers.Function.bind(this.selectionOn, this));
  theMap.registerForEvent(Fusion.Event.MAP_SELECTION_OFF,
    OpenLayers.Function.bind(this.selectionOff, this));
}

function selectionOn() {
  //a new selection has been made, request it
  theMap.getSelection(displaySelection);
}

function displaySelection(selection) {
  //display the selection to the user in some way ...
}

function selectionOff() {
  //clear the selection results
}
```

The parameter passed to the callback routine is an associative array that contains selections from one or more maps. In most cases, there will be one element in the array, with a selection object for the main map. This selection object provides the following API:

- `getNumLayers()` returns the number of layers that have features selected.
- `getNumElements()` returns the total number of features that are selected.
- `getLowerLeftCoord()` returns the lower, left coordinate of the bounding box of all selected features.

- `getUpperRightCoord()` returns the upper, right coordinate of the bounding box of all selected features.
- `getLayerByName(`*`name`*`)` gets the layer selection object for a layer from the name of the layer. This returns null if there is no layer with the requested name in the selection results.
- `getLayer(`*`index`*`)` gets the layer selection object for the requested layer where *`index`* is between 0 and one less than the value returned by `getNumLayers()`.

An application will typically loop over the layers in a selection and retrieve individual results using the Layer Selection object returned by `getLayer()` or `getLayerByName()`. Layer selection objects have the following API:

- `getName()` returns the name of the layer that the selected features are in.
- `getNumElements()` returns the number of features selected in this layer.
- `getNumProperties()` returns the number of data properties, or attributes, of the features in this layer.
- `getPropertyNames()` returns an array of the names of each of the properties.
- `getPropertyTypes()` returns an array of the types of the properties.
- `getElementValue(elementIndex, propertyIndex)` returns the actual value of a given property for a given element.

The following code is an example of how to use the Selection and Layer Selection objects to create a tabular display of selected features.

```
function displaySelection(multiSelection) {
  var theMap = Fusion.getWidgetById("Map");
  if (!theMap.hasSelection())
  {
    alert("Nothing selected");
    return;
  }
  var selection = multiSelection[theMap.getMapName()];

  //display the selection to the user in some way ...
  //make sure something got selected ...
  if (selection && selection.getNumLayers() > 0)
  {
    //obtain a reference to the HTML Element that the results
    //will be placed in
    var resultElm = $('selectionResultDiv');
    resultElm.innerHTML = '';
    for (var i=0; i<selection.getNumLayers(); i++) {
      var selectionLayer = selection.getLayer(i);
      var propNames = selectionLayer.getPropertyNames();
      var span = document.createElement('span');
      span.className = 'selectionResultsTitle';
      span.innerHTML = 'Layer ' + selectionLayer.getName();
      resultElm.appendChild(span);
      var table = document.createElement('table');
      table.className = 'selectionResultsTable';
      resultElm.appendChild(table);
      //set up the table header to be the property names
      var thead = document.createElement('thead');
      table.appendChild(thead);
      var tr = document.createElement('tr');
      thead.appendChild(tr);
      for (var j=0; j<propNames.length; j++) {
        var td = document.createElement('td');
        td.innerHTML = propNames[j];
        tr.appendChild(td);
      }
      //output the selection values
      var tbody = document.createElement('tbody');
      table.appendChild(tbody);
      for (var j=0; j<selectionLayer.getNumElements(); j++) {
        var tr = document.createElement('tr');
        tbody.appendChild(tr);
```

```
        for (var k=0; k<propNames.length; k++) {
          var td = document.createElement('td');
          td.innerHTML = selectionLayer.getElementValue(j, k);
          tr.appendChild(td);
        }
      }
    }
  } else {
    //could display a message of some sort saying nothing was
    //selected
  }
}
```

Fusion API

The Fusion API is the JavaScript programming interface available to developers building flexible web layout applications. When building an application, you can implement functionality as new components or as custom JavaScript on top of the Fusion API. The primary mechanism for interacting with the Fusion API is through the global Fusion object. The Fusion object is created when you include `fusion.js` in your page and initialize your application (via `Fusion.initialize()`).

Methods

The Fusion object contains these methods:

- `registerForEvent(`*`event_id, callback`*`)` – used to register a callback function for events emitted by Fusion itself. Most events are actually emitted by components, and you will need to register with those components specifically if you want to receive their events.
- `deregisterForEvent(`*`event_id, callback`*`)` – used to remove a callback associated with an event. This can only be called if the callback has already been registered via `registerForEvent`. The callback must be the same function pointer that was passed to `registerForEvent`. If you are using the `bind()` method to bind a function to an object instance, you should save a reference to the result of calling `bind()` and use that for calling `registerForEvent` if you plan to later call `deregisterForEvent`.
- `initializeLocale(`*`optional_locale`*`)` – should be called at the beginning of the `window.onload`i function if `String.Translate` is going to be used.

- `ajaxRequest(scriptUrl, options)` – a helper function for calling server-side scripts using XmlHttpRequest by modifying the `scriptUrl` to use the redirect script if necessary. This is primarily used by components that need to call a server-side script to perform some action so that they are portable across a variety of uses. Custom JavaScript using the Fusion API generally should not need to use this function. The options are passed through to prototype's `Ajax.Request` and must conform to the specifications for `Ajax.Request`.
- `getMapById(id)` – returns a reference to a Map component by the ID of the element the map was created in. If no map was created in an element with this ID, it returns null.
- `getMapByName(name)` – returns a reference to a Map component by the name of the Map. If no map component has a map of the requested name, it returns null.
- `getMapByIndice(index)` – returns the map at the requested index. (The index is zero-based.) A Map's index depends on its position in the Web Layout file relative to other maps.
- `getWidgetById(id)` – returns a reference to a component by the ID of the element the component was created in. If no component was created in an element with this ID, it returns null.
- `getWidgetsByType(type)` – returns an array with references to all the components of the given type. If no components of a given type are found, an empty array is returned.
- `getConfigurationItem(arch, key)` – returns a Fusion configuration value for the given architecture and key.
- `getScriptLanguage()` – returns the server-side scripting language for the application, for example "PHP". Flexible web layouts are designed to support multiple server-side scripting languages. All communication with server-side scripts should use the `getScriptLanguage` function to ensure compatibility with future versions.
- `error(code, message)` – triggers an error. Applications can register for the `FUSION_ERROR` event to be informed of errors that occur.

The following methods work with units of measurement. For more information, see Units (page 125).

- `unitFromName(name)` – returns the numeric code for a unit of measurement from its name. Most functions that deal with measurements use the numeric code.
- `unitName(code)` – returns the name of a unit of measurement from a numeric code.
- `unitAbbr(code)` – returns the abbreviation of a unit of measure from a numeric code.
- `toMeter(code, amount)` – converts a measurement into meters from the given numeric code.
- `fromMeter(code, amount)` – converts a measurement from meters into the units represented by the numeric code.
- `convert(fromCode, toCode, amount)` – converts a measurement from one unit of measurement into another.

Events

The event code is designed to provide an asynchronous notification mechanism that can be used to register for, and receive notification of, key events that happen asynchronously. The following terms are used with respect to events:

1. event id: a unique identifier for an event, represented by a JavaScript variable that is all in upper case (e.g `FUSION_INITIALIZED`). All event IDs are properties of the `Fusion.Event` object.
2. trigger: when an event occurs, it is "triggered" and all the registered callback functions are notified
3. register: provide a callback function that is called when an event is triggered
4. deregister: remove a callback function that was previously registered
5. publish: anything that can trigger an event must publish all the valid event IDs

The event mechanism is implemented by two functions: `registerForEvent` and `deregisterForEvent`. Both functions have the same signature, taking an

event ID as the first parameter and a callback function pointer as the second parameter.

The Fusion object provides two specific events that can be used by applications to get notification of when Fusion initialization has completed and when an error occurs anywhere in Fusion. These events are:

- `FUSION_ERROR`. This is triggered when an internal error happens. Details on the error are passed to the callback function. Applications should register for this event before calling `Fusion.initialize()` to ensure that they receive errors that happen during initialization.
- `FUSION_INITIALIZED`. This is triggered when Fusion's initialization is complete and the application is running. This signals that it is safe for the application to communicate with specific components. Note that the Map component, specifically, will be ready but may not have actually loaded the map. There is a separate event for that. (See The Map Component (page 115).) Applications should register for this event before calling `Fusion.initialize()`.

Components are designed to be completely independent of one another, allowing them to be added to, or removed from, applications with little or no impact on the other components in the application. However, there are cases (especially with the Map component) where it is important that components be able to communicate with other components or with the application as a whole. For these situations, there is an event mechanism that allows components and applications to register for and trigger events. The event mechanism allows components to be independent of each other, but still provide a high level of integration when required. For more information on the events supported by the Map component, see The Map Component (page 115) and Working With Selections (page 117).

To register a callback function for a component event, the application must first obtain a reference to the component through one of the methods of the Fusion object (typically `getWidgetById`) and then call `registerForEvent` passing one of the event IDs that is valid for that component.

Units

The Fusion API provides convenience functions for converting between units of measurement. Units of measurement are specified by a string in most configuration files, but the API uses a numeric code internally. These numeric codes are defined as attributes of the Fusion object. The following table lists

the Fusion attribute and associated unit name and abbreviations that are associated with them.

If the units are not specified, the default units are `Fusion.UNKNOWN`.

Unit Code	Name	Abbreviation	Valid Strings
Fusion.UNKNOWN	Unknown	unk	unknown
Fusion.INCHES	Inches	in	inches inch in
Fusion.FEET	Feet	ft	feet ft
Fusion.YARDS	Yards	yd	yards yard yd
Fusion.MILES	Miles	mi	miles mile mi
Fusion.NAUTICALMILES	Nautical Miles	nm	nautical miles nautical mile nm
Fusion.MILLIMETERS	Millimeters	mm	millimeters millimeter mm
Fusion.CENTIMETERS	Centimeters	cm	centimeters centimeter cm
Fusion.METERS	Meters	m	meters meter m
Fusion.KILOMETERS	Kilometers	km	kilometers kilometer km
Fusion.DEGREES	Degrees	°	degrees degree deg
Fusion.DECIMALDEGREES	Decimal Degrees	°	decimal degrees dd
Fusion.DMS	Degrees Minutes Seconds	°	degrees minutes seconds
Fusion.PIXELS	Pixels	px	pixels pixel px

Flexible Web Layouts Examples

10

Overview

The examples in this chapter demonstrate some important techniques for use with Autodesk MapGuide flexible web layouts.

The examples use a custom template that is based on the Slate template.

NOTE These examples require the Sheboygan sample data. See the *Installing Sample Data and Sample Applications* PDF for details.

Installing the Examples

Complete source code is available for all the examples in this chapter on the installation DVD in `Samples\FlexibleWebLayoutsSamples`.

There are two installation files required. A MapGuide package file, `FlexibleWebLayoutsExamples.mgp`, contains the application definition for a flexible web layout. Load this into the MapGuide site repository using the Site Administrator. This creates the following resource:

```
Library://Samples/FlexibleWebLayouts/
Examples.ApplicationDefinition
```

A zip file, `FlexibleWebLayoutsExamples.zip`, contains source files. Unzip this into the *`installDir`*`\WebServerExtensions\www` folder with "Use folder names" selected in WinZip. It installs the following files:

- `fusion\templates\mapguide\examples\*`

- `fusion\widgets\Example*.js`
- `fusion\widgets\widgetinfo\example*.xml`
- `fusion\widgets\example*\*`

The descriptions of the examples assume that both the package file and the zip file have been installed.

Running the Examples

To run the examples, make sure that the Autodesk MapGuide Server is running. Connect to

```
http://server:port/mapguide2010/fusion/templates/mapguide/
examples/index.html?ApplicationDefinition=
Library%3a%2f%2fSamples%2fFlexibleWebLayouts%2f
Examples.ApplicationDefinition
```

Modify the host name, port number, or path to the Autodesk MapGuide root as required.

NOTE MapGuide Studio provides a shortcut to the correct web page. Inside MapGuide Studio, open the sample flexible web layout, `Samples/FlexibleWebLayouts/Examples.ApplicationDefinition`. Click View in Browser to open the example template and layout in a browser.

Firefox and Firebug

Although flexible web layouts are designed to work with most current browsers, Mozilla Firefox and the Firebug extension are recommended for developing new layouts and widgets. Together they provide strong support for developing web applications.

The Firebug extension includes features for inspecting the HTML, DOM, JavaScript, and CSS on the pages. It also includes a JavaScript debugger and many other features.

You can get the Mozilla Firefox browser at *www.mozilla.com* and the Firebug extension at *www.getfirebug.com*.

Some of the following examples write information to the Firebug console. If you prefer to use a different browser for development, you can modify the code to use another method of output.

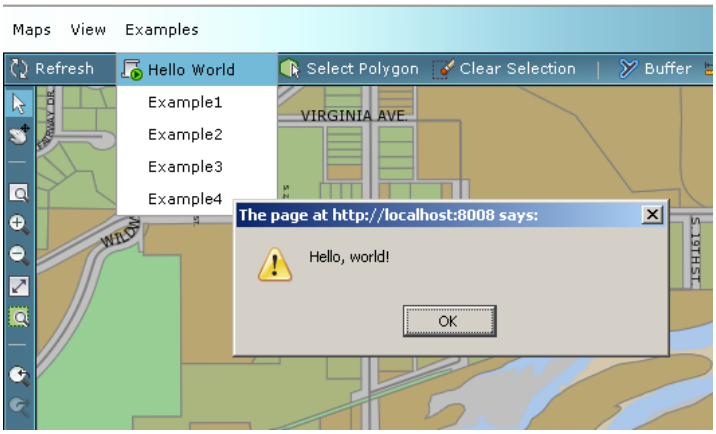
Hello World: A Simple Invoke Script

Autodesk MapGuide ships with a set of widgets that handle many common requirements. Some of these widgets allow for simple customization to adapt them to particular needs.

One example is the Invoke Script widget. This can be used to run JavaScript commands from within a flexible web layout.

The example layout includes an Invoke Script widget that displays a JavaScript alert.

To run the Hello World example, open the flexible web layout in a browser. Click Examples menu ➤ Hello World. This displays the alert.



Example 1: Creating a Widget

A widget is a functional piece of code that can be added to a flexible web layout. Most widgets are self-contained, so they can be added to the layout without affecting other functionality. Widgets use JavaScript for client-side interaction, and some also require pages on the web tier, written in a supported language for the Autodesk MapGuide Web API.

Example 1 is a widget that is designed to be called as a command from a menu. It displays a JavaScript alert like Hello World: A Simple Invoke Script (page 129), but it uses an external JavaScript file. It uses the following files in the folder *WebServerExtensionsInstallDir*\www\fusion:

- widgets\Example1.js

- widgets\widgetinfo\example1.xml

The JavaScript file, Example1.js, contains a class definition for the widget. All widgets must inherit from the base class Fusion.Widget. Because this is a simple widget, it only has an initialize method and an activate method. More complex widgets require additional methods.

```
Fusion.Widget.Example1 = OpenLayers.Class(Fusion.Widget,
  {
    uiClass: Jx.Button,

    initializeWidget: function(widgetTag)
    {
      this.enable();
    },

    /**
     * Function: execute
     *
     * Says hello
     *
     */
    activate: function()
    {
      console.log('Example1.execute');
      alert("Hello World!");
    }
  }
);
```

NOTE The class name must match the file name of the JavaScript file. In this example, the class name is Fusion.Widget.Example1 and the file name is Example1.js.

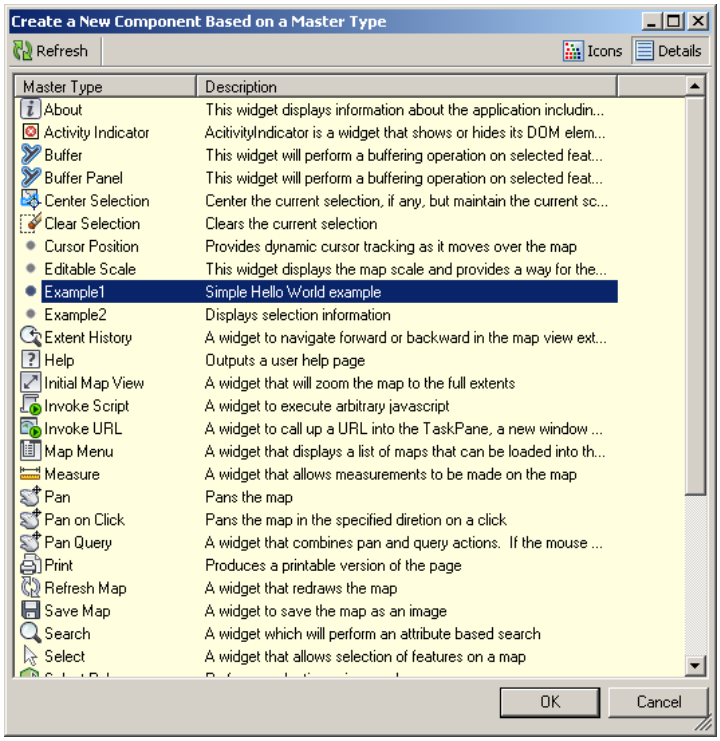
The widget information file, widgetinfo\example1.xml, is used by MapGuide Studio to describe the widget. This widget is designed to be incorporated into a menu, so its <ContainableBy> element is set to Any. See Example 3: Dialogs and Events (page 135) for an example of a widget that is a stand-alone dialog.

```
<WidgetInfo>
  <Type>Example1</Type>
  <LocalizedType></LocalizedType>
  <Description>Simple Hello World example</Description>
  <Location></Location>
  <Label>Example1</Label>
  <Tooltip>Click to say hello</Tooltip>
  <StatusText></StatusText>
  <ImageUrl></ImageUrl>
  <ImageClass></ImageClass>
  <StandardUi>true</StandardUi>
  <ContainableBy>Any</ContainableBy>
</WidgetInfo>
```

NOTE The widget type must match the class name in the JavaScript file.

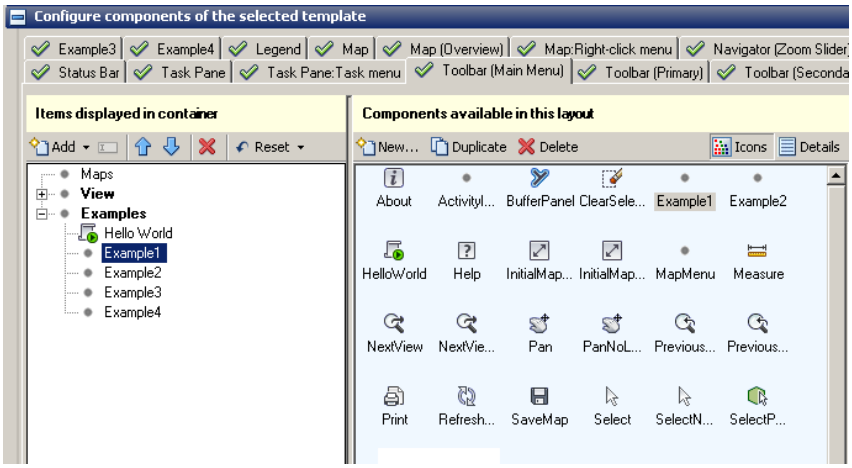
The widget information file defines a master widget type. MapGuide Studio uses this type to create instances of the widget and add them to flexible web layouts. The sample flexible web layout already has this widget added to the menu. If you want to add a new instance to another location in the web layout, perform the following procedure:

1. Select the tab for the container where you want to place the new component.
2. In the Components Available in this Layout section, click New.
3. The list of master widget types displays. Select the master type and click OK.



4 This creates a new instance of the master type. Change any of the parameters for the widget, as required.

5 Drag the widget to the list of items in the container to add it to the layout.



A single instance of a widget can be referenced in more than one place. The parameters for the widget are the same for every instance. For example, the widget for Example 1 could be added to the map context menu as well as the Examples menu. It would perform the same function regardless of where it was called.

Different instances of the same master type can have different behavior. For example, the standard templates have various instances of InvokeScript widgets that perform different operations.

Example 2: Selections

Example 2 is a widget that is designed to be called as a command from a menu. It requires Mozilla Firefox and the Firebug extension because it writes to the Firebug console. It uses the following files in the folder *WebServerExtensionsInstallDir*\www\fusion:

- widgets\Example2.js
- widgets\widgetinfo\example2.xml

This example writes information about the currently selected parcels to the Firebug console. For more information about selections, see The Map Component (page 115) and Working With Selections (page 117).

In a flexible web layout, the current selection is available from the map object using the `getSelection()` method. This is an asynchronous method that

accepts a callback routine as a parameter. For example, the following sets up a callback for the current selection:

```
var theMap = Fusion.getWidgetById("Map");
theMap.getSelection(this.displaySelection );
```

The callback method must accept a single parameter, which is a selection object:

```
displaySelection : function(selection){
  // process the selection object
}
```

It is possible for a single selection object to contain selected features from multiple maps and multiple layers in those maps. The `selection` object passed to the callback routine is an associative array where each element in the array has the selection data for a single map.

To get the selection for a single map, get the element corresponding to that map. This is usually the main map. For example:

```
var theMap = Fusion.getWidgetById("Map");
if (!theMap.hasSelection())
{
  alert("Nothing selected");
  return;
}
var oSelection = selection[theMap.getMapName()];
```

To get the selected features for a layer, call either `getLayerByName()` or `getLayer()`. For example, to get the selected features for the Parcels layer, call

```
var thisLayer = oSelection.getLayerByName('Parcels');
```

To loop through all the layers in a selection, do the following:

```
for (var layerNum = 0; layerNum < oSelection.getNumLayers();
    layerNum++)
{
  var thisLayer = oSelection.getLayer(layerNum);
  var selectedFeaturesThisLayer = thisLayer.getNumElements();
  if (selectedFeaturesThisLayer > 0)
  {
    // Process the selected features
  }
}
```

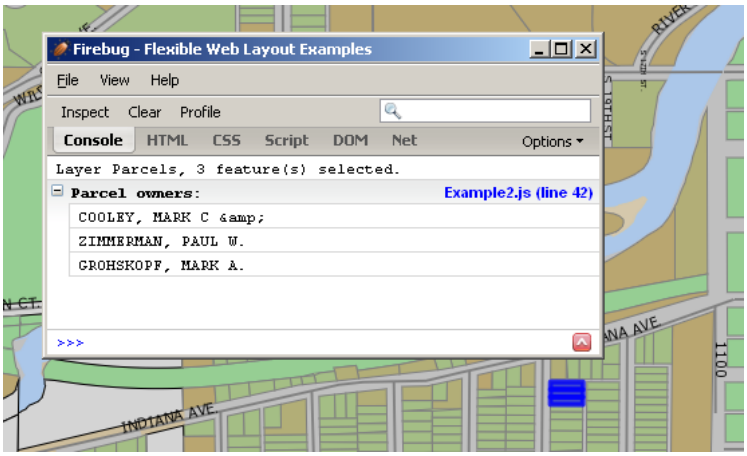
To process each selected feature, do the following:

```
for (var featureNum = 0; featureNum < selectedFeaturesThisLayer;
    featureNum++)
{
  // process feature
}
```

The layer definition in MapGuide Studio defines what properties of the selected features are available to the JavaScript API on the client. Each property has a name and a value.

```
var propNames = thisLayer.getPropertyNames();
var nProperties = thisLayer.getNumProperties();
for (var propNum = 0; propNum < nProperties; propNum++)
{
  // process property
  var thisPropName = propNames[propNum];
  var thisPropValue = thisLayer.getElementValue(featureNum,
    ownerPropNum)
}
```

`Example2.js` combines these concepts to write the owner names of selected parcels to the Firebug console.



Example 3: Dialogs and Events

Many of the standard widgets, such as the selection panel and the legend, are separate dialogs. Depending on the template, they can behave somewhat differently. For example, in the Aqua template the dialogs are floating, and in the Slate template many of them are docked at the side of the map display.

Example 3 creates a dialog that displays some details about the current selection. It updates automatically when the selection changes. It uses the following files in the folder *WebServerExtensionsInstallDir*\www\fusion:

- widgets\Example3.js
- widgets\widgetinfo\example3.xml
- templates\mapguide\examples.xml
- templates\mapguide\examples\index.html

There are a few different concepts shown in this example:

- dialogs
- opening dialogs programatically
- responding to events

Dialogs

The Jx library includes full support for many different types of dialog. The style used is defined by the template, in the `window.onload()` function of `index.html`. For a floating dialog, create a `Jx.Dialog` object, as in the following:

```
dExample3 = new Jx.Dialog({
  id: 'dialogExample3',
  label: OpenLayers.i18n('Example 3'),
  modal: false,
  resize: true,
  horizontal: '50 left',
  vertical: '85 top',
  width: 400,
  height: 400,
  contentId: 'Example3'
  });
```

For examples of other types of dialog, see `index.html` for the other standard templates.

The properties passed to the `Jx.Dialog` constructor define the initial state of the dialog, such as size and location. The value of the `ContentID` property must correspond to a `<div>` element in the HTML. For example, the dialog defined above must have the following in `index.html`:

```
<div id="Example3"></div>
```

This `ContentID` must also correspond to the class name of the widget as defined in the JavaScript file. `Example3.js` contains the following class definition:

```
Fusion.Widget.Example3 = OpenLayers.Class(Fusion.Widget,
  {
  // ...
  }
);
```

NOTE The `id` property of the `Jx.Dialog` constructor (`dialogExample3` in the example above) refers to a `<div>` element that contains the entire dialog, including outside borders, the title bar, and other parts. The Jx library creates this outer element when it initializes the dialog. In most cases, the widget code should only work with the content area of the dialog, identified with the `ContentID` property.

Within MapGuide Studio, widgets that are dialogs are treated differently from widgets that are run from menus. The widget information file for Example 3, `widgets\widgetinfo\example3.xml`, contains the following:

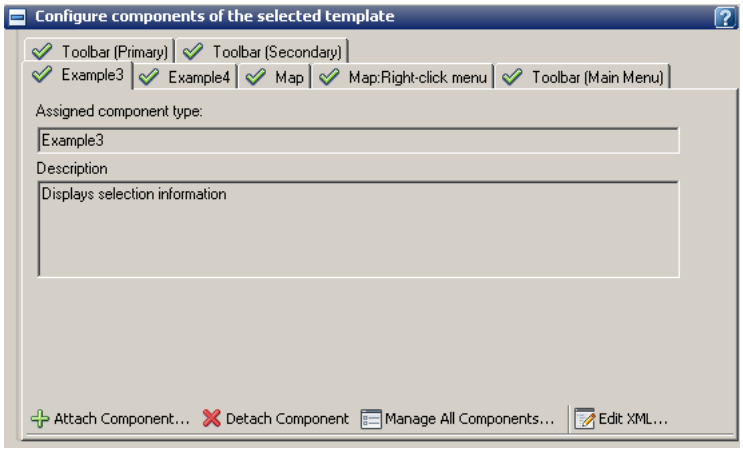
```
<WidgetInfo>
  <Type>Example3</Type>
  <LocalizedType>Example3</LocalizedType>
  <Description>Displays selection information</Description>
  <Location></Location>
  <Label>Example3</Label>
  <Tooltip></Tooltip>
  <StatusText></StatusText>
  <ImageUrl></ImageUrl>
  <ImageClass></ImageClass>
  <StandardUi>false</StandardUi>
  <ContainableBy></ContainableBy>
</WidgetInfo>
```

The `<ContainableBy>` element is empty because this widget is a separate dialog and cannot be added to a menu. In MapGuide Studio, this widget does not appear in the component list for toolbars and other containers.

The template definition file, `templates\mapguide\examples.xml`, contains a `<Panel>` element that can hold dialogs or containers:

```
<Panel>
<Name>Example3</Name>
<Label>Example3</Label>
<Description>The dialog displaying example 3</Description>
</Panel>
```

By default, MapGuide Studio places widgets with matching names into the appropriate panel. The panel name matches the `<div>` ID in the `index.html` file. If a given template uses a dialog-style widget it must have a matching `<div>` element. The `<Panel>` elements create tabs in the MapGuide Studio UI.



Events

Many dialogs and other widgets react to events in the map, such as a user selecting a feature.

There are two steps that must be done in a widget that handles events:

- Create a callback routine to handle the event.
- Register the callback routine for the event.

Example 3 uses the events `Fusion.Event.MAP_SELECTION_ON` and `Fusion.Event.MAP_SELECTION_OFF`, which trigger when the selection changes.

The initialization code for the `Example3` widget, in `Example3.js`, registers event handlers for both these events, as follows:

```
this.getMap().registerForEvent(
  Fusion.Event.MAP_SELECTION_ON,
  OpenLayers.Function.bind(this.updateSelection, this));
this.getMap().registerForEvent(
  Fusion.Event.MAP_SELECTION_OFF,
  OpenLayers.Function.bind(this.clearSelection, this));
```

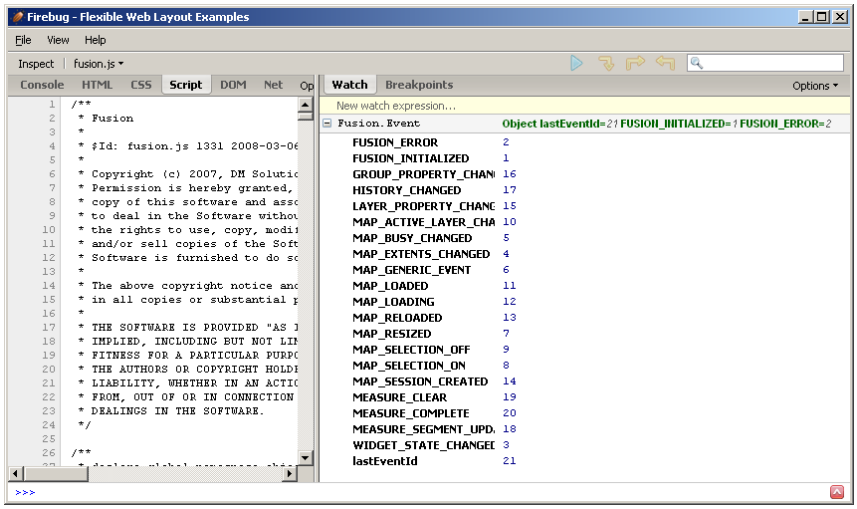
The event handler for `Fusion.Event.MAP_SELECTION_OFF` is simple:

```
clearSelection: function() {
  this.showInformation('No features are selected.');
},
```

The event handler for `Fusion.Event.MAP_SELECTION_ON` is a bit more complex because it needs to process the selection. It calls `getSelection()` with a callback routine to process the current selection. See Example 2: Selections (page 133) for more details about processing selections.

```
updateSelection: function() {
  this.getMap().getSelection(this.listSelection.bind(this));
},
```

TIP For a quick list of available events, open the Script tab of the Firebug extension. Enter `Fusion.Event` as a watch expression.



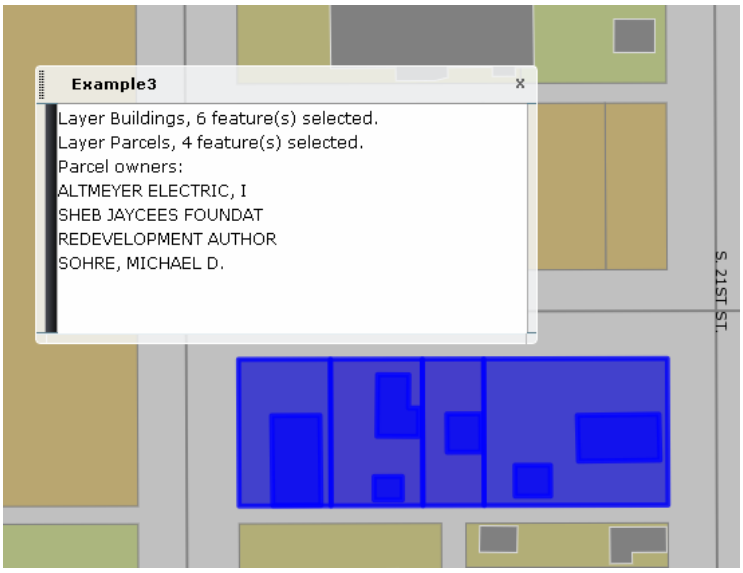
Running the Example

The initialization code for the `Example3` widget, in `index.html`, hides the dialog. There is a menu option to display the dialog, Examples menu ➤ Example3. This is defined in the Application Definition as an Invoke Script command that calls the following JavaScript routine:

```
var showExample3 = function() {
    dExample3.open();
}
```

The routine is defined in `index.html` where it is available to the Invoke Script command.

When the dialog is open it reacts to changes in the selection. Any time the selection changes, the dialog updates to show a list of the currently selected features.



Example 4: Updating the Site Repository

Example 4 creates a dialog that displays the owner of the current selection and allows it to be edited. The updated owner name is sent to a page on the web tier for updating in the feature source. It uses the following files in the folder *`WebServerExtensionsInstallDir`*`\www\fusion`:

- `widgets\Example4.js`
- `widgets\widgetinfo\example4.xml`
- `widgets\Example4\Example4.php`
- `templates\mapguide\examples.xml`
- `templates\mapguide\examples\index.html`

Example 4 is very similar in structure to Example 3: Dialogs and Events (page 135). It uses a dialog that updates based on the events `Fusion.Event.MAP_SELECTION_ON` and `Fusion.Event.MAP_SELECTION_OFF`. Instead of simply displaying the selection information, though, the dialog has an input field so the user can edit the owner of the currently selected parcel.

The initialization code for the dialog, in `Example4.js`, creates the form with an input text field and button, and sets the `onclick` action for the button.

```
this.messageDiv = document.createElement('div');
this.formInput = document.createElement('form');
this.formInput.innerHTML =
   '<p><label>Property owner: <input type="text" ' +
   'name="propertyOwner" ' +
   'value="propertyowner" size="30" / ></label></p>' +
   '<p><input name="inputButton" type="button" value="Update" ' +
   'style="width: 70px;" class="Ctrl" /></p>';

this.formInput.inputButton.onclick = this.updatePropertyOwner;

this.messageDiv.appendChild(this.formInput);
this.domObj.appendChild(this.messageDiv);
```

The widget updates the dialog whenever the selection changes.

If a user clicks the update button, the widget sends the new owner name to a page on the web tier, using an `Ajax.Request` object from the Prototype JavaScript framework. It expects the web page to return results in a JSON object.

```
updatePropertyOwner : function()
{
  var thisWidget = Fusion.getWidgetById('Example4');
  var theMap = Fusion.getWidgetById('Map');
  var reqParameters = {};
  reqParameters.session = Fusion.sessionId;
  reqParameters.mapname = theMap.getMapName();
  reqParameters.layer = 'Parcels';
  reqParameters.propName = 'RNAME'; // Must use name from feature
 source, not display name
  reqParameters.newValue = thisWidget.formInput.propertyOwn
er.value;

  this.sBaseUrl = '/widgets/Example4/Example4.php';

  Fusion.ajaxRequest(this.sBaseUrl,
    {
      method:'post',
      parameters: reqParameters,

      onSuccess: function(transport)
      {
        var jsonContent = eval("(" + transport.responseText + ")");

        // Optionally, write the results to the Firebug console
        console.group('Returned values');
        for (var prop in jsonContent)
        {
          console.log(prop + ': ' + jsonContent[prop]);
        }
        console.groupEnd();
        if (jsonContent.error)
        {
          alert('There was an error:\n' + jsonContent.errorMsg);
        }
        else
        {
          alert('Update successful!');
        }
      },

      onFailure: function(ajaxResponse)
      {
```

```
            alert('Error:\n' + ajaxResponse.status + ' ' +
              ajaxResponse.statusText);
          }
        }
      );
    }
```

NOTE The client-side JavaScript uses the display names for properties, as defined in the layer definition in MapGuide Studio. The Autodesk MapGuide Web API uses the property names as defined in the feature class. In this example, the property in the feature class is `RNAME` and the display name is `Owner`.

The web page, `Example4.php`, updates the property for the selected feature. This is standard use of the Web API as described in Modifying Maps and Layers (page 57) and other sample programs. There are a few items that are specific to flexible web layouts, though.

It includes the file `Common.php` which performs some standard initialization for widgets, including creating an `MgResourceService` object `$resourceService` and setting the `$mapName` variable.

```
$fusionMGpath = '../../layers/MapGuide/php/';
include $fusionMGpath . 'Common.php';
```

Because the client-side JavaScript function expects to receive JSON-formatted results, the web page must set the appropriate headers.

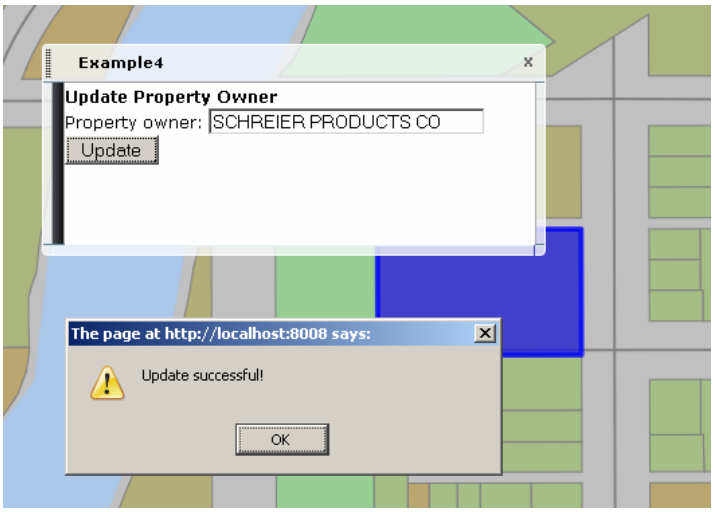
```
header('Content-type: text/x-json');
header('X-JSON: true');
```

It must also format the results properly. In this case, the web page returns the original `POST` arguments with some added members to indicate success or failure of the request. The added members are simply added to the arguments array. For example:

```
$args['error'] = false;
```

The entire arguments array is returned to the client.

```
echo json_encode($args);
```



NOTE When first loaded, the feature source for the SDF layer in the Sheboygan data may be set to read-only. `Example4.php` contains code to make it writable. This may not apply to other feature sources.

```
$reader =
  $resourceService->GetResourceContent($layerFeatureResource);
$xmlString = $reader->ToString();
$domDoc = DOMDocument::loadXML($xmlString);
$xpath = new DOMXPath($domDoc);
$query = "//Parameter[Name='ReadOnly']";
$nodes = $xpath->query($query);
foreach ($nodes as $node)
{
  $query = "Value";
  $subNodes = $xpath->query($query, $node);
  foreach ($subNodes as $subNode)
  {
    $subNode->nodeValue = 'FALSE';
  }
}
$newXml = $domDoc->saveXML();
$resourceService->SetResource($layerFeatureResource,
  new MgByteReader($newXml, MgMimeType::Xml), NULL);
```

Example 5: Anonymous Login

Normally, when a user opens a flexible web layout in a web browser they will be prompted for a username and password. (You will not see this if you use the View In Browser button in MapGuide Studio.)

To enable anonymous login, the examples include the file *WebServerExtensionsInstallDir*\www\fusion\FlexViewer.php:

```
<?php
  $fusionMGpath = './layers/MapGuide/php/';
  include $fusionMGpath . 'Common.php';
  $locale = GetDefaultLocale();
  $appdef = "";
  $template = "";
  $session = $siteConnection->GetSite()->CreateSession();
  GetRequestParameters();
  $viewerSrc = 'templates/mapguide/' . $template
    . '/index.html';
  $viewerSrc = $viewerSrc . '?APPLICATIONDEFINITION='
    . $appdef;
  $viewerSrc = $viewerSrc . '&SESSION=' . $session;
  header( 'Location: ' . $viewerSrc ) ;
  function GetParameters($params)
  {
    global $appdef, $template;
    $appdef = $params['APPLICATIONDEFINITION'];
    $template = $params['TEMPLATE'];
  }
  function GetRequestParameters()
  {
    if($_SERVER['REQUEST_METHOD'] == "POST")
      GetParameters($_POST);
    else
      GetParameters($_GET);
  }

?>
```

You can now log in anonymously (without supplying a username and password) through the URL:

```
http://server:port/mapguide2010/fusion/FlexViewer.php
?APPLICATIONDEFINITION=Library%3a%2f%2fSamples%2f
FlexibleWebLayouts%2fExamples.ApplicationDefinition
&TEMPLATE=examples
```

NOTE Since an anonymous user does not have rights to modify the repository, you will not be able to update the parcel owner in Example 4 if you log in this way.

Using MapGuide Logging 11

Introduction

This chapter describes MapGuide's logging capabilities and gives examples of how they can be used. It is intended for technical individuals such as MapGuide application developers and system administrators.

Logs and Logging Detail

MapGuide provides three logs, the access log, error log and trace log, to aid in debugging and performance tuning.

Access Log

The access log has been enhanced to include the thread identifier of the thread executing the operation. This thread identifier along with the timestamp can be used to trace execution of an operation through the error.log and trace.log. A sample log excerpt is given below. 2332 and 2344 are the thread identifiers.

```
<2008-07-16T12:55:48>
   2332 Anonymous
   RenderDynamicOverlay.1.0.0:4(MgMap,MgSelection,PNG,true)
   Success
<2008-07-16T12:55:48>
   2344 Anonymous
   GenerateLegendImage.1.0.0:7(MgResourceIdentifier,53344,
   16,16,PNG8,-1,-1)
   Success
```

Error Log

The error log has been enhanced to include:

- The thread identifier.
- Display of input parameters for method calls configurable on a per-service basis.
- Non-fatal warning statements such as FDO errors when rendering a layer on a map. This is configurable on a per-service basis.

A sample `error.log` showing a warning is given below. This warning was generated during the stylization of a layer. The operation was executed on thread 2340.

```
<2008-07-17T14:57:29> 2340 Anonymous
 Warning: An exception occurred in FDO component.
         Geometry property value encountered!
 StackTrace:
  - MgStylizationUtil.ExceptionTrap() line 199 file
    .\GeometryAdapter.cpp
    An exception occurred in FDO component.
    Geometry property value encountered!
```

Trace Log

The trace log includes:

- The thread identifier.
- Display of input parameters for method calls configurable on a per-service basis.
- Non-fatal warning statements such as FDO errors when rendering a layer on a map. This is configurable on a per-service basis.
- Timestamped begin (BGN) and end (END) statements for each method call.

Here a sample excerpt from the trace log.

```
<2008-07-17T14:57:29>
 2364 BGN     MgStylizationUtil.StylizeLayers
 Map=Sheboygan,LayerId=Library://Samples/Sheboygan/Layers/
 CityLimits.LayerDefinition
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.GetSpatialContexts
 Id=Library://Samples/Sheboygan/Data/CityLimits.FeatureSource,Act
iveOnly=0
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.GetSpatialContexts
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.SelectFeatures
 Resource=Library://Samples/Sheboygan/Data/
 CityLimits.FeatureSource,ClassName=SHP_Schema:CityLimits,
 Options={{Operator=1}{GeomProp=SHPGEOM}{GeomOp=
 EnvelopeIntersects}{Geometry=POLYGON
 ((-87.780335659913959 43.691398128787803,
 -87.680172841948519 43.691398128787803,
 -87.680172841948519 43.797520000480297,
 -87.780335659913959 43.797520000480297,
 -87.780335659913959 43.691398128787803))}}
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.SelectFeatures
<2008-07-17T14:57:29>
 2364
 Warning: An exception occurred in FDO component.
        Geometry property value encountered!
<2008-07-17T14:57:29>
 2364 END     MgStylizationUtil.StylizeLayers
```

Configurable Log Detail

Log detail can be configured on a per-service basis using the `LogsDetail` parameter in the general properties section of `serverconfig.ini`:

```
[GeneralProperties]
LogsDetail = MappingService:3,FeatureService:3,RenderingService:3,
ResourceService:3
```

The allowable service entries are:

- MappingService
- RenderingService

- FeatureService
- ResourceService
- SiteService

The detail level is defined as follows:

- Level 0 - Errors without parameters
  - This is the default level of detail.
  - Errors (exceptions) are logged without method parameters for the specified service.
  - Warnings are not logged for this service.
  - Traces are not logged for this service.
  - Call parameter generation is disabled at this level to improve performance.
- Level 1 - Warnings and Errors with parameters
  - Errors (exceptions) are logged with method parameters for the specified service.
  - Warnings are logged with parameters for this service.
  - Traces are not logged for this service.
- Level 2 - Trace, Warnings, and Errors
  - All Level 1 detail, plus traces with parameters are logged for published Service API calls.
- Level 3 - Internal Trace, Trace, Warnings, and Errors
  - All Level 2 detail, plus traces with parameters are logged for internal API calls.

Turning on a level 2 or level 3 trace log for all services in a production environment can generate more than 100MB of log information per hour. Disk space should be continuously monitored when the MapGuide Server is run under these conditions.

NOTE Since the detail level is configurable for each service, the logs can be tailored to debug specific error conditions. For example, tracing the execution of the Resource service is not very useful when you are debugging a database connectivity problem.

Sample Cases

This section describes how to use the logging functionality to debug or tune some common issues in MapGuide.

Debugging and Tuning Feature Sources

To debug feature source issues, turn on warning level detail for the feature service:

```
[GeneralProperties]
LogsDetail = FeatureService:1
```

This provides more detail for any thrown exceptions by including the method parameters in the stack trace.

```
<2008-07-28T17:04:10> 3324   Anonymous
 Error: An exception occurred in FDO component.
        Failed to connect to 'calcrtora'. Invalid data
        source, user name or password.
 StackTrace:
  - MgFeatureServiceHandler.ProcessOperation() line 83 file
    f:\mgdev1.2\os\server\src\services\feature\
    FeatureServiceHandler.cpp
  - MgOpDescribeSchemaAsXml.Execute() line 107 file
    f:\mgdev1.2\os\server\src\services\feature\
    OpDescribeSchemaAsXml.cpp
  - MgServerFeatureService.DescribeSchemaAsXml
    (Resource=Library://Data/NT_NA.FeatureSource,
    SchemaName=) line 333 file f:\mgdev1.2\os\server\src\
    services\feature\ServerFeatureService.cpp
  - MgServerDescribeSchema.DescribeSchema() line 500 file
    f:\mgdev1.2\os\server\src\services\feature\
    ServerDescribeSchema.cpp
  - MgServerDescribeSchema.ExecuteDescribeSchema()
    line 173 file f:\mgdev1.2\os\server\src\services\
    feature\ServerDescribeSchema.cpp
  - MgFdoConnectionManager.Open() line 979 file
    f:\MgDev1.2\OS\Server\src\Common\Manager\
    FdoConnectionManager.cpp An exception occurred in
    FDO component.
Failed to connect to 'calcrtora'. Invalid data source,
user name or password.
```

Additional information can be obtained by enabling trace level detail:

```
[GeneralProperties]
LogsDetail = FeatureService:2
```

The trace log provides more context. For example, the `DescribeSchemaAsXml` call below is failing after the resource content and resource data have been retrieved from the repository. The trace log also provides timestamps for the start and end of each feature service operation. These timestamps can be used to determine the execution time of feature service operations.

```
<2008-07-28T17:04:09.900546> 3324
   BGN MgServerFeatureService.DescribeSchemaAsXml
   Resource=Library://Data/NT_NA.FeatureSource,SchemaName=
<2008-07-28T17:04:09.900546> 3324
   BGN MgServerResourceService.GetResourceContent
   Id=Library://Data/NT_NA.FeatureSource,Tags=Substitution
<2008-07-28T17:04:09.900546> 3324
   END MgServerResourceService.GetResourceContent
<2008-07-28T17:04:09.900546> 3324
   BGN MgServerResourceService.GetResourceData
   Id=Library://Data/NT_NA.FeatureSource,DataName=
   config.xml,Tags=Substitution
<2008-07-28T17:04:09.900546> 3324
   END MgServerResourceService.GetResourceData
<2008-07-28T17:04:10.884997> 3324
   END MgServerFeatureService.DescribeSchemaAsXml
<2008-07-28T17:04:10.884997> 3324
 Error: An exception occurred in FDO component.
        Failed to connect to 'calcrtora'. Invalid data
        source, user name or password.
<2008-07-28T18:20:01.259646> 1160
   BGN MgServerFeatureService.DescribeSchemaAsXml
   Resource=Library://Data/NT_NA.FeatureSource,SchemaName=
<2008-07-28T18:20:01.259646> 1160
   BGN MgServerResourceService.GetResourceContent
   Id=Library://Data/NT_NA.FeatureSource,Tags=Substitution
<2008-07-28T18:20:01.259646> 1160
   END MgServerResourceService.GetResourceContent
<2008-07-28T18:20:01.259646> 1160
   BGN MgServerResourceService.GetResourceData
   Id=Library://Data/NT_NA.FeatureSource,DataName=
   config.xml,Tags=Substitution
<2008-07-28T18:20:01.259646> 1160
   END MgServerResourceService.GetResourceData
<2008-07-28T18:20:06.275657> 1160
   END MgServerFeatureService.DescribeSchemaAsXml
<2008-07-28T18:20:06.275657> 1160
```

The timestamps for the BGN and END of `MgFeatureService.DescribeSchemaAsXml` are 5 seconds apart indicating that the `DescribeSchema` call is relatively slow for the `NT_NA.FeatureSource`.

Debugging Broken Layers

By default MapGuide silently suppresses any errors which occur when rendering a layer. So to help debug any errors which occur when it is rendering a layer, turn on warning level logging so that warnings are logged to both the error log and the trace log.

```
[GeneralProperties]
LogsDetail = MappingService:1,FeatureService:1,
RenderingService:1
```

The warning message alone will not provide enough detail to determine which layer is causing problems:

```
<2008-07-17T14:57:29> 2364 Anonymous
 Warning: An exception occurred in FDO component.
        Geometry property value encountered!
 StackTrace:
  - MgStylizationUtil.ExceptionTrap() line 199 file
  .\GeometryAdapter.cpp
  An exception occurred in FDO component.
```

The trace log can then be turned on to provide sufficient detail. The available detail level depends on the service. Most services can return level 2 detail. Mapping Service can return level 3 detail. The `MgStylizationUtil.StylizerLayers` trace entry is level 3 detail for Mapping Service.

To return level 3 detail for the Mapping Service and level 2 detail for the Feature and Rendering Service, the `LogsDetail` parameter can be set as follows:

```
[GeneralProperties]
LogsDetail = MappingService:3,FeatureService:2,
RenderingService:2
```

This results in:

```
<2008-07-17T14:57:29>
 2364 BGN     MgStylizationUtil.StylizeLayers
 Map=Sheboygan,LayerId=Library://Samples/Sheboygan/Layers/
 CityLimits.LayerDefinition
<2008-07-17T14:57:29>
 1544 MgClientHandler::Initialize() -
 Address: 27.0.0.1:2812
<2008-07-17T14:57:29>
 1544 MgClientHandler::Initialize() -
 Address: 27.0.0.1:2811
<2008-07-17T14:57:29>
 2300 MgServerMappingService::GenerateLegendImage
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.GetSpatialContexts
 Id=Library://Samples/Sheboygan/Data/
 CityLimits.FeatureSource,ActiveOnly=0
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.GetSpatialContexts
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.SelectFeatures
 Resource=Library://Samples/Sheboygan/Data/
 CityLimits.FeatureSource,ClassName=SHP_Schema:CityLimits,
 Options={{Operator=1}{GeomProp=SHPGEOM}
 {GeomOp=EnvelopeIntersects}{Geometry=POLYGON
 ((-87.780335659913959 43.691398128787803,
 -87.680172841948519 43.691398128787803,
 -87.680172841948519 43.797520000480297,
 -87.780335659913959 43.797520000480297,
 -87.780335659913959 43.691398128787803))}}
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.SelectFeatures
<2008-07-17T14:57:29>
 2364 Warning: An exception occurred in FDO component.
 Geometry property value encountered!
<2008-07-17T14:57:29>
 2364 END     MgStylizationUtil.StylizeLayers
```

From the error log, the warning message was generated on thread 2364. Using the timestamp and the thread identifier the call stack can be traced using the trace log. Starting at the warning statement in the trace log, thread 2364 can be backtraced through the `MgServerFeatureService.SelectFeatures` call and into the `MgStylizationUtil.StylizeLayers` call. The BGN/END statements show operation nesting for each thread. By using Excel or a text editor, the other threads can be removed from the log.

```
<2008-07-17T14:57:29>
 2364 BGN MgStylizationUtil.StylizeLayers
 Map=Sheboygan,LayerId=Library://Samples/Sheboygan/Layers/
 CityLimits.LayerDefinition
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.GetSpatialContexts
 Id=Library://Samples/Sheboygan/Data/
 CityLimits.FeatureSource,ActiveOnly=0
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.GetSpatialContexts
<2008-07-17T14:57:29>
 2364 BGN     MgServerFeatureService.SelectFeatures
 Resource=Library://Samples/Sheboygan/Data/
 CityLimits.FeatureSource,ClassName=SHP_Schema:CityLimits,
 Options={{Operator=1}{GeomProp=SHPGEOM}
 {GeomOp=EnvelopeIntersects}{Geometry=POLYGON
 ((-87.780335659913959 43.691398128787803,
 -87.680172841948519 43.691398128787803,
 -87.680172841948519 43.797520000480297,
 -87.780335659913959 43.797520000480297,
 -87.780335659913959 43.691398128787803))}}
<2008-07-17T14:57:29>
 2364 END     MgServerFeatureService.SelectFeatures
<2008-07-17T14:57:29>
 2364 Warning: An exception occurred in FDO component.
      Geometry property value encountered!
<2008-07-17T14:57:29>
 2364 END MgStylizationUtil.StylizeLayers
```

The FDO exception is emitted immediately after the `SelectFeatures` call. This is logical because `SelectFeatures` returns an `MgFeatureReader` and the first `ReadNext()` executed by the stylizer causes the error to occur.

From the log, the warning message occurs between the BGN/END blocks of `StylizeLayers`. This means the CityLimits layer definition is the layer causing the error. This allows the you to concentrate your effort on a single layer instead of walking through every layer in a map.

Index

G

H

I

J

L

M

O

P

R

S

T

U

V

W