Editing Data Online with the Autodesk MapGuide® Enterprise API

Dongjin Xing – Autodesk, Inc.

DE205-3 Autodesk MapGuide Enterprise enables you to view your geospatial data online and edit the data online in a browser. Join this class and you'll learn about the geospatial data structure and how to create, delete, and update data. You'll learn how to use the Viewer API in accessing geometries in the browser and how to send the geometry information to the server for feature creation and backend data storage. Knowledge of C# or VB.NET is required for this session.

About the Speaker:

Dongjin Xing is a technical consultant for Autodesk Developer Network, helping Autodesk partners worldwide create solutions for geospatial applications such as MapGuide Enterprise, Map 3D, Civil 3D, and AutoCAD. Prior to joining Autodesk, he worked for ESRI providing API consulting, support, and training services for the ArcGIS product line. This is his second year as a speaker at Autodesk University.

Autodesk University 2007

Editing Data Online with the Autodesk MapGuide® Enterprise API

Introduction

In MapGuide Enterprise, map data can be edited online by the user. Existing data can be modified and new data can be added in the backend data storage such as SDF, Oracle Spatial, and so on. This editing operation includes both spatial editing on the geometry and attribute editing in the attribute table.

New feature source can also be created programmatically and added to the map display on the fly. For example, a new SDF file can be created as the result of a query and a new layer based on the SDF can then be added to the map.

In the API, editing is performed by the feature service

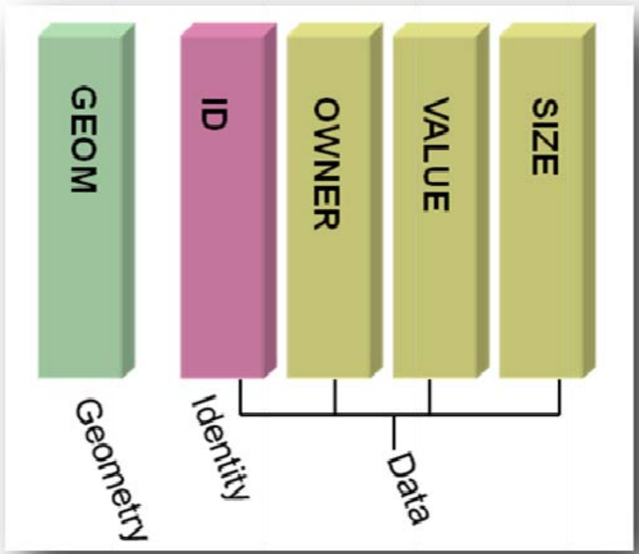
In this session, we will cover the following tasks.

1. Deleting, updating, and inserting features on an existing feature source
2. Spatial editing and geometry creation
3. Creating new SDFs
4. Adding new layers to map display

Feature Source Structure

The geospatial data in MapGuide Enterprise is stored in feature sources. A typical example of feature source is SDF file, Shapefile, Oracle Spatial table, or ArcSDE feature class. Feature source has a similar structure as database tables.

Typically, in a feature source, there are several fields, which can be categorized into two types. The first type is geometry field, which contains the geometric shape of the geospatial object. The other type is data field, which contains the attribute information of the geospatial object. Normally, one data field is used as the identity field, whose value will uniquely identify the feature.

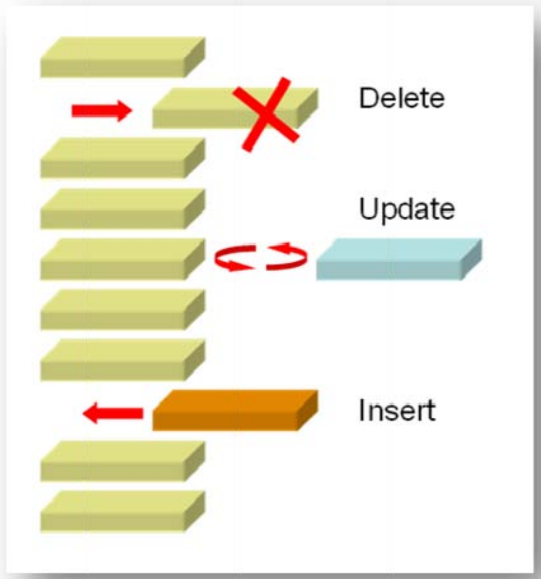


A feature consists of both the geometry and the attribute information.

The picture on the left shows a feature class definition. In this feature class, there is a geometry field, GEOM. There are also 5 data fields, ID, OWNER, VALUE, and SIZE. ID field is used as the identity field.

Feature Editing

Features in the feature source can be updated or deleted, and new features can be inserted into the feature source as illustrated here..



Deleting Features

```
MgDeleteFeatures deleteFeature1 = new MgDeleteFeatures("Parcels", "ID=2354");
MgDeleteFeatures deleteFeatures = new MgDeleteFeatures("Parcels",
      "OWNER LIKE 'JOHN%'");
MgDeleteFeatures deleteFeature3 = new MgDeleteFeatures("Parcels",
      "GEOM INTERSECTS GEOMFROMTEXT('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))')");
```

The above is the code snippet for deleting 3 features from a feature class named *Parcels*. The MgDeleteFeatures class takes two parameters. The first is the name of the feature class from which features will be deleted. The second is the query string, which specifies what feature would be deleted. The query string is the same as the one used to perform queries in MapGuide Enterprise. For more information on query, please refer to the MapGuide Enterprise documentation.

In the above example, the first line will delete the feature whose ID field equals 2354. The second will delete all the features whose OWNER field has a value like JOHN%, such as JOHN, JOHNSON, or JOHNNY. The third will delete all features whose geometry intersects with a polygon that is defined by this vertex string, 0 0, 2 0, 2 2, 0 2, 0 0.

Updating Features

```
MgUpdateFeatures updateFeature1 = new MgUpdateFeatures
      ("Parcels", properties, "ID=2354");
MgUpdateFeatures updateFeature2 = new MgUpdateFeatures
      ("Parcels", properties, "OWNER LIKE 'JOHN%'");
MgUpdateFeatures updateFeature3 = new MgUpdateFeatures
      ("Parcels", properties,
      "GEOM INTERSECTS GEOMFROMTEXT('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))')");
```

MgUpdateFeatures class is used for editing features. Like MgDeleteFeatures, it also uses a query string to indicate the features to be edited. MgUpdateFeatures takes three parameters. The first parameter is the name of the feature class in which the features will be updated. The second parameter contains the new values for all the fields to be updated on the selected features. We will discuss this parameter in detail later. The third parameter, like in MgDeleteFeatures, is the query string.

Inserting Features

```
MgInsertFeatures insertFeature = new MgInsertFeatures
      ("Parcels", properties);
```

MgInsertFeatures is used for adding new features to the feature class. It takes the name of the feature class and the new feature value to be added.

Committing Edits

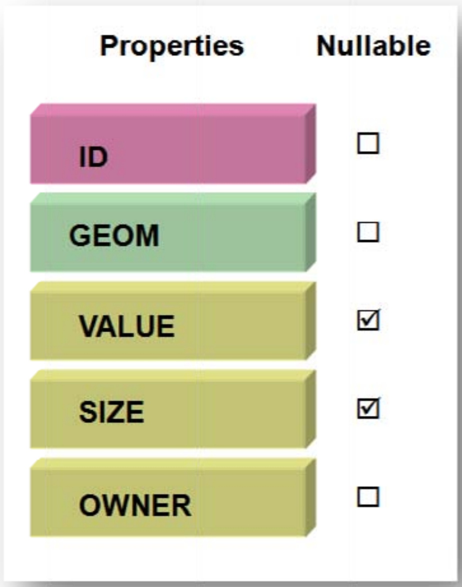
After we have used MgDeleteFeatures, MgUpdateFeatures, and MgInsertFeatures to specify the edits to be performed on the feature class, we need to commit the edits to make the editing operation take place.

```
MgFeatureCommandCollection commands = new MgFeatureCommandCollection();

commands.Add(deleteFeature1);
commands.Add(deleteFeature2);
commands.Add(deleteFeature3);

commands.Add(updateFeature1);
commands.Add(updateFeature2);
commands.Add(updateFeature3);

commands.Add(insertFeature);

MgLayer layer = map.GetLayers().GetItem("Parcels");
layer.UpdateFeatures(commands);
```

MgFeatureCommandCollection is used to commit all the edits. We add all the editing commands from above to this object. Then we call MgLayer.UpdateFeatures to actually perform all the editting operations.

Passing New Feature Values

In the code snippet for updating and inserting feature above, we use variable *properties* to pass the new values of the feature. In this section, we will talk about this variable in detail.



The above picture is a feature source's definition. Field ID is a data field, which is used as the identity field. GEOM field is the geometry field. It has three additional data field, VALUE, SIZE, and OWNER.

In the earlier code snippet for updating features, we updated a feature as shown below.

```
MgUpdateFeatures updateFeature1 = new MgUpdateFeatures
      ("Parcels", properties, "ID=2354");
```

In this feature class, if we want to change the OWNER field to "Smith", VALUE field to 250,000, and GEOM field to a new polygon, this is how we popolute the variable *properties*.

```
MgPropertyCollection properties = new MgPropertyCollection();
MgAgfReaderWriter agfWriter = new MgAgfReaderWriter();
MgPolygon poly = getNewGeometry();
properties.Add(new MgGeometryProperty("GEOM", agfWriter.Write(poly)));
properties.Add(new MgStringProperty("OWNER", "Smith"));
properties.Add(new MgInt32Property("VALUE", 250000));
```

You have noticed that the variable *properties* are encapsulated by the MgPropertyCollection object. To populate this object, we simply need to add corresponding property values to it.

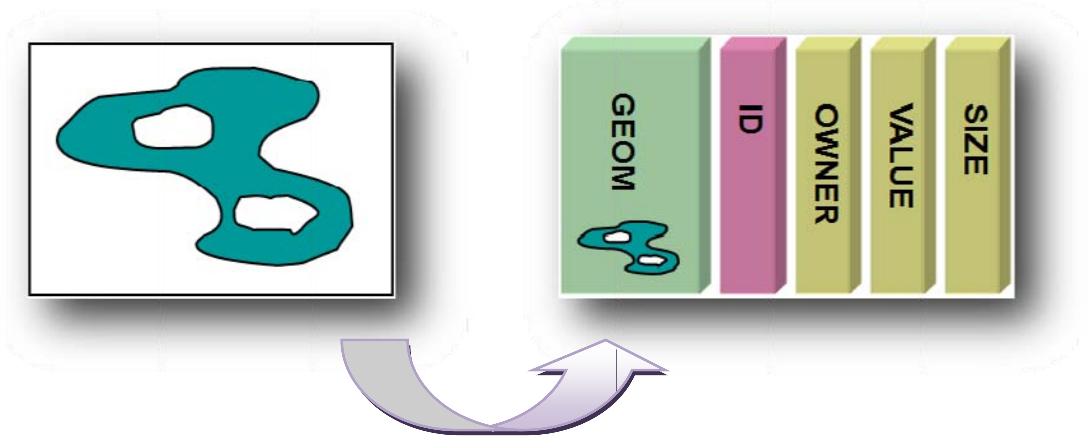
In the above feature class definition, out of the five fields, VALUE and SIZE fields are nullable, which means when inserting a new feature, we don't have to provide the values for these two fields. However, we must provide values for the remaining three fields; ID, GEOM, and OWNER.

Based on this definition, if we want to insert a new feature to the feature source, we must populate the MgPropertyCollection object with at least the field values for ID, GEOM, and OWNER.

```
MgPropertyCollection properties = new MgPropertyCollection();
MgAgfReaderWriter agfWriter = new MgAgfReaderWriter();
MgPolygon poly = getNewGeometry();
properties.Add(new MgGeometryProperty("GEOM", agfWriter.Write(poly)));
properties.Add(new MgStringProperty("OWNER", "Smith");
properties.Add(new MgInt32Property("ID", 324323));
```

Creating Geometry

In the code snippets for feature updating and inserting shown earlier, we also edited the geometry field by providing an MgGeometryProperty in the MgPropertyCollection. Now the question is how to create a new geometry object programmatically with the API.



Assuming that we already have a string of X/Y coordinates in the array of x_coordinates and y_coordinates. The following code will create a polygon.

```
MgGeometryFactory geoFactory = new MgGeometryFactory();
MgCoordinateCollection coordCol = new MgCoordinateCollection();

for (int i = 0; i < num; i++)
{
    MgCoordinate coord = geoFactory.CreateCoordinateXY(
      x_coordinates[i], y_coordinates[i]);
    coordCol.Add(coord);
}

MgLinearRing outRing = geoFactory.CreateLinearRing(coordCol);
MgPolygon polygon = geoFactory.CreatePolygon(outRing, null);
```

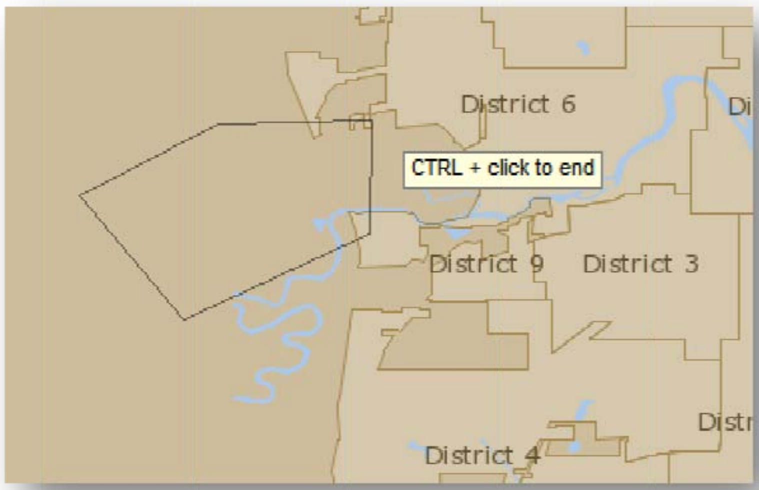
A key object in the above code snippet is the MgGeometryFactory. This is the factory object to create any geometry in MapGuide Enterprise. We first create a coordinate from the X/Y arrays. Then we add the coordinates into a collection. With the collection, we create an MgLinearRing. This ring is the outer ring for the polygon. Because the polygon doesn't have any holes, we don't have to create the inner rings for it. At last, we create the polygon from the outer ring.

Please note that MgGeometryFactory doesn't validate the X/Y coordinates in creating the geometry. You must follow the rules yourself. Particularly, you need to pay attention to these two rules.

1. When creating a polygon, ensure that the first vertex on the polygon is the same as the last one.
2. For a polygon, the outer ring must traverse in the counter-clockwise direction and inner rings in the clockwise direction . Be sure to provide the coordinates therefore in the proper order.

Digitizing Geometry in MapGuide Enterprise Viewer

You can create a new geometry such as a polygon in the viewer with your mouse. MapGuide Enterprise viewer API has the JavaScript function to allow you to click on the map to digitize the line string like the following. Then the line string coordinates can be collected in the map unit and submitted to the server to create the geometry with the MgGeometryFactory object.



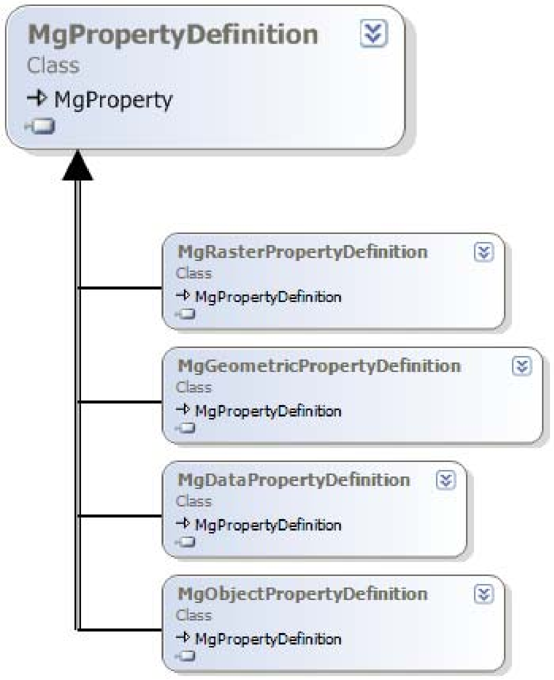
The following is the JavaScript functions to digitize the polygon and get its coordinate values.

The Viewer API method DigitizePolygon will put the viewer into the polygon digitization mode. It takes one parameter, which is the name of the the event handler function called when you hold the CTRL key and click the last point to signify that the polygon is finished. In our case, OnPolygonDigitized is the event handler, in which we parse the *poly* variable that contains the vertex coordinates and submit the result to the server.

```
<script type="text/javascript">
      function DigitizePolygon() {
            parent.parent.mapFrame.DigitizePolygon(OnPolygonDigitized);
      }

      function OnPolygonDigitized(poly) {
            str = poly.Count + "~";
            for(var i = 0; i < poly.Count; i++) {
                  pt = poly.Point(i);
                  str += pt.X + "!" + pt.Y + "_";
            }

            document.getElementById("coordinates").value = str;
            form.submit()
      }
</script>
```

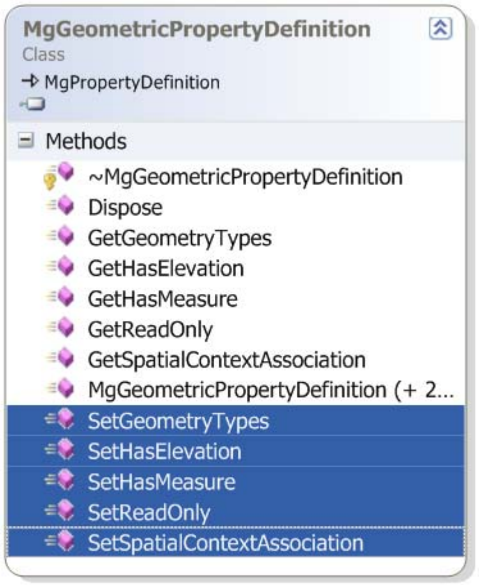
Creating New SDF files

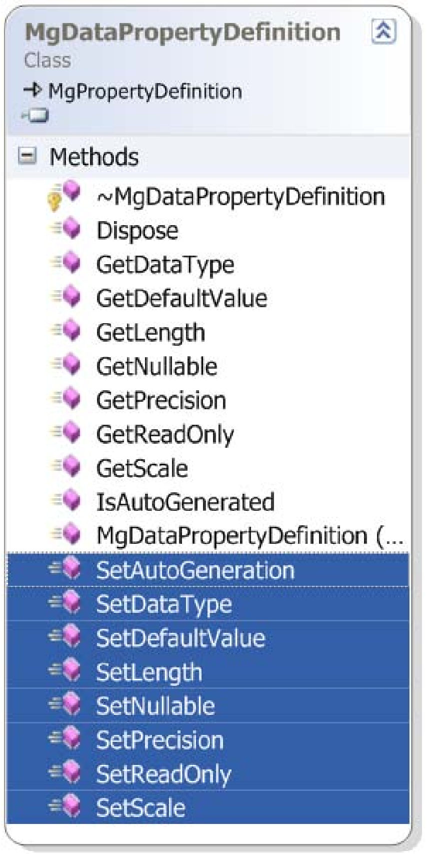
We can create new SDF files programmatically by creating a new feature source definition. MgClassDefinition and MgPropertyDefinition are two of the main classes used to create a new feature source definition.



MgClassDefinition is used to define the feature source and MgPropertyDefinition is used to define the individual property, also called field, in the feature source.

MgPropertyDefinition is a base class, which has 4 derived classed. The two most oftern used derived classes are MgGeometricPropertyDefinition for the geometry properties and MgDataPropertyDefinition for the data properties.


MgGeometricPropertyDefinition
Class
MgPropertyDefinition
Methods
~MgGeometricPropertyDefinition
Dispose
GetGeometryTypes
GetHasElevation
GetHasMeasure
GetReadOnly
GetSpatialContextAssociation
MgGeometricPropertyDefinition (+ 2...
SetGeometryTypes
SetHasElevation
SetHasMeasure
SetReadOnly
SetSpatialContextAssociation



```
MgClassDefinition parcelClass = new MgClassDefinition();
parcelClass.SetName("tempParcel");
```

The first step is to create a new MgClassDefinition object and give it a name.

```
MgPropertyDefinitionCollection props = parcelClass.GetProperties();

MgDataPropertyDefinition id = new MgDataPropertyDefinition("ID");
id.SetDataType(MgPropertyType.Int32);
id.SetReadOnly(true);
id.SetNullable(false);
id.SetAutoGeneration(true);
props.Add(id);

MgPropertyDefinitionCollection idProps = parcelClass.GetIdentityProperties();
idProps.Add(id);

MgGeometricPropertyDefinition geom = new
MgGeometricPropertyDefinition("GEOM");
geom.SetGeometryTypes(MgFeatureGeometricType.Surface);
geom.SetHasElevation(false);
geom.SetHasMeasure(false);
geom.SetSpatialContextAssociation("LL84");
props.Add(geom);
```

```
parcelClass.SetDefaultGeometryPropertyName("GEOM");

MgDataPropertyDefinition acre = new MgDataPropertyDefinition("ACRE");
acre.SetDataType(MgPropertyType.String);
acre.SetLength(256);
props.Add(acre);
```

Then we create property definitions for the geometry property and data properties and add them to the MgPropertyDefinitionCollection object.

```
MgFeatureSchema schema = new MgFeatureSchema();
schema.SetName("SchemaParcels");
schema.GetClasses().Add(parcelClass);

string ll84Wkt = "GEOGCS[\"LL84\",DATUM[\"WGS_1984\",SPHEROID[\"WGS
84\",6378137,298.25722293287],TOWGS84[0,0,0,0,0,0,0]],PRIMEM[\"Greenwich\",0]
,UNIT[\"Degrees\",1]]";

MgCreateSdfParams sdfParams = new MgCreateSdfParams("LL84", ll84Wkt, schema);
featureService.CreateFeatureSource(resId, sdfParams);
```

Lastly, a feature schema is created and the previous MgClassDefinition object is added to the feature schema. So far, SDF is the only feature source that we can create in the MapGuide Enterprise API. We use MgCreateSdfParams to set up the parameters for the SDF file, such as the geographic coordinate systme. The last step is to use MgFeatureService.CreateFeatureSource to physically create the SDF file at the specified location. Variable resId is the resource identifier, which specifies the location in the repository for the SDF feature source.

Adding New Layers

After we have created the SDF file, we can add it to a map as a new layer dynamically. Map layer is the cartographic presentation of the feature source. It doesn't contain any geospatial data, which comes from the feature source. Map layer consist of the styling and theming information of the feature source. In another word, map layer specifies how the geospatial data should be displayed.

Map layer definition is an XML file in the repository. The following is a typical layer definition XML.. The ResourceId node specifies the feature source referenced by the layer, in this case, *Library://Sheboygan/Data/Parcels.FeatureSource*. If we change this to the value of variable *resId* on the last line of the code above, this layer will reference the SDF file we just created.

```
<?xml version="1.0" encoding="UTF-8"?>
<LayerDefinition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="LayerDefinition-1.1.0.xsd" version="1.1.0">
 <VectorLayerDefinition>
  <ResourceId>Library://Sheboygan/Data/Parcels.FeatureSource</ResourceId>
  <FeatureName>SHP_Schema:Parcels</FeatureName>
  <FeatureNameType>FeatureClass</FeatureNameType>
  ...
  ...
  <Geometry>SHPGEOM</Geometry>
  ...
  <VectorScaleRange>
```

```
   <MaxScale>10000</MaxScale>
   <AreaTypeStyle>
    <AreaRule>
     <LegendLabel>Zone:  AGR</LegendLabel>
     <Filter>RTYPE = &apos;AGR&apos;</Filter>
     <AreaSymbolization2D>
      <Fill>
       <FillPattern>Solid</FillPattern>
       <ForegroundColor>FFC19E6A</ForegroundColor>
       <BackgroundColor>FF000000</BackgroundColor>
      </Fill>
      <Stroke>
       <LineStyle>Solid</LineStyle>
       <Thickness>0</Thickness>
       <Color>FF808080</Color>
       <Unit>Inches</Unit>
       <SizeContext>DeviceUnits</SizeContext>
      </Stroke>
     </AreaSymbolization2D>
    </AreaRule>
```

You can find more information on the XML's schema in the MapGuide Enterprise API reference. We also have a session dedicated to resource management in the AU. The session ID is DE115-3.

To create a new layer, we mainly need to upload the above layer definition XML to the server repository.

```
MgResourceIdentifier resId = new MgResourceIdentifier
      ("Library://Resources/NewParcel.LayerDefinition");
MgByteSource content = new MgByteSource(@"C:\Temp\LayerDefinition.xml");
resourceService.SetResource(resId, content.GetReader(), null);

MgLayer parcelLayer = new MgLayer(tempParcelLayerId, resService);
parcelLayer.SetName("New Parcels");
parcelLayer.SetLegendLabel("New Parcels");
parcelLayer.SetDisplayInLegend(true);
parcelLayer.SetSelectable(false);

MgMap map = new MgMap();
map.Open(resService, "Sheboygan");
map.GetLayers().Insert(0, parcelLayer);

parcelLayer.SetVisible(true);
parcelLayer.ForceRefresh();
map.Save(resService);
```

In the above code snippet, we first upload the layer definition XML from the local drive to the repository by using MgResourceService.SetResource. Then we create a new *MyLayer* object from the resource and set up its properties lsuch as name, legend, and so on. We add the layer to the map and save the map. This will add a new layer to the map. The new layer can be seen after refreshing the viewer..