Geomajas Editing plug-in guide

Geomajas Developers and Geosparc

Geomajas Editing plug-in guide

by Geomajas Developers and Geosparc

1.0.0-M2

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

This plug-in provides a set of services and utilities for visually editing geometries on the map within a GWT environment. It uses the Geomajas Geometry Project as the base geometry type, and the `MapController` definition from the Geomajas Common GWT face library. On top of the visual geometry editing, this plug-in also provides services for snapping, splitting and merging.

1. Model-view-controller

In essence the editing follows the tried and tested Model-View-Controller principle:

- Model: The central `GeometryEditService`. It keeps track of the location and status of all vertices and edges of the geometry being edited.
- View: The `GeometryEditService` will fires events that are being picked up by a specific renderer. Any change in the shape of the geometry or the status of it's vertices/edges will be caught and visually displayed on the map.
- Controller: A series of handlers will execute methods in the `GeometryEditService`, based upon user interaction on the map.

> **Note**
>
> As the rendering is face specific, the focus has first gone to the GWT face. An implementation for the PureGWT face will follow later.

The renderer is one example of a listener to the many events that the `GeometryEditService` fires, but essentially anyone can listen to those events. If you need to react to any change in a geometry's shape, just add the correct handler.

2. The GeometryIndex concept

Before trying to figure out how the `GeometryEditService` works, it is important to understand the `GeometryIndex` concept. A `GeometryIndex` is an index within a geometry that points to a single vertex, edge or sub-geometry. All operations within the `GeometryEditService` operate on a list such `GeometryIndices`.

Take for example the "move" operation. This operation will move the given `GeometryIndex` to the specified location. This operation is used when the user drags a vertex around on the map, but this operation could also be used to let the user drag an interior ring within a polygon, or an entire LineString within a MultiLineString, or event the whole geometry.

The GeometryIndex is based on the internal structure of the geometry, which may contain 4 or more levels:

1. Geometry collection level: this is the highest structural level for geometry collections: multipolygon, multilinestring or arbitrary geometry collections. In theory a geometry collection may contain other geometry collections, but this is rarely encountered.
2. Geometry level: this is the level of a basic geometry like polygon, linestring or point
3. Ring level: for a polygon, this is the level of the linear rings. There is usually an exterior ring (boundary), but there may also be additional interior rings (holes)
4. Vertex or edge level: this is the level of the individual vertices and edges. A single edge connects 2 vertices.

The elements at each level have a fixed ordering, which makes it possible to uniquely determine such an element by its order at each level of the structural tree. This combination of order numbers, together with a type to distinguish between edges, vertices or higher level structures (which we generally call geometries) forms the `GeometryIndex`.

Lets give some examples to clarify this. The following table shows at the left column a geometry in WKT format with a highlighted section and the corresponding `GeometryIndex`. The GeometryIndex is an array of integers combined with a type. For edge, the type is `edge`, for vertex it is `vertex` and for all other structures it is `geometry`. The last row contains a multipolygon with 2 polygons. The highlighted section is a couple of points that determines an edge of the interior ring of the first polygon. The indices are 0 (for the first polygon), 1 (for the interior ring) and 2 for being the 3rd edge of this ring (counting starts with index 0 in all cases).

Table 1.1. Geometry index samples

WKT	Geometry index
POINT(**0 0**)	[0], type = vertex
LINESTRING (30 10, **10 30**, 40 40)	[1], type = vertex
LINESTRING (**30 10**, 10 30, **40 40**)	[2], type = edge
POLYGON ((35 10, 10 20, 15 40, 45 45, 35 10), **(20 30, 35 35, 30 20, 20 30)**)	[1], type = geometry
POLYGON ((35 10, 10 20, 15 40, 45 45, 35 10), (20 30, 35 35, **30 20**, 20 30))	[1,2], type = vertex
MULTIPOLYGON (((35 10, 10 20, 15 40, 45 45, 35 10), (20 30, 35 35, **30 20, 20 30**)),((35 10, 10 20, 15 40, 45 45, 35 10)))	[0,1,2], type = edge

3. The central services

There are 3 central services that help in the editing process. All three have a very distinct responsibility:

- *GeometryEditService*: Defines the editing workflow and the basic operations (with undo/redo) that are supported. Also allows to add handlers to all events.
- *GeometryIndexService*: This service defines operations for creating and manipulating GeometryIndices. It also supports retrieving information based upon a certain geometry and index. For example what are the adjacent vertices to a certain edge within a given geometry?
- *GeometryIndexStateService*: Keeps track of the state of all indices that make up the geometry being edited. It allows for selecting/deselecting, enabling/disabling, highlighting, etc any vertices/edges/sub-geometries during the editing process. This state can then be used by the controllers. For example, a controller could allow only selected vertices to be dragged by the user.

There are more services then the 3 mentioned above such as a `SnapService`, `GeometryMergeService` and `GeometrySplitService`, but those just add more functionality to the basic set that the 3 above already provide.

Chapter 2. Java/GWT Editing API

1. Maven configuration

In order to use this plug-in in combination with the GWT face, the following Maven dependency is required:

Example 2.1. Maven dependency for GWT face

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-editing-gwt</artifactId>
</dependency>
```

2. the GeometryEditor object for the GWT face

Before going deeper into the inner workings of the different services and how they work together, we need to explain the GeometryEditor. This is the top level editing object that makes all of the other services work together. It has a GeometryEditService, a renderer for the map, a SnapService, and a registry for controller on the map.

This is where you start when trying to edit geometries within the GWT. Because the rendering is face specific, this object too is face specific. A GeometryEditor can simply be constructed using a map:

Example 2.2. Constructing a GeometryEditor

```
GeometryEditor editor = new GeometryEditor(mapWidget);
```

3. The GeometryEditService

Central service for all operations concerning the geometry editing process. This process should work together with a set of controllers on the map that execute methods from this service after which events are fired for a renderer to act upon. This service makes use of the `GeometryIndexService` to identify sub-geometries, vertices and edges. All operations work on a set of such indices. This allows for great flexibility in the operations that can be performed on geometries.

3.1. Geometry edit workflow

Editing a geometry comes down to starting the editing process, applying some operations and then stopping the process again. Starting and stopping happens through the `start` and `stop` methods. Know also that operations onto the geometry really do apply on the same geometry that was passed with the `start` method. In other words, this service changes the original geometry. If you want to support some roll-back functionality within your code, make sure to create a clone of the geometry before starting this edit service.

3.2. GeometryEditState

At any time during the editing process, the `GeometryEditService` has a general state that tells you what's going on. This state is defined in the `GeometryEditState`. Currently there are 3 possible states for the editing process to be in:

- *IDLE*: The default state.

- *INSERTING*: The user is currently inserting new points into the geometry. The `GeometryEditService` has an "insertIndex" (of the type `GeometryIndex`), that points to the next suggested insert location. The controllers pick up on this index to insert points (or edges, or geometries).
- *DRAGGING*: The user is currently dragging a part of the geometry. The `GeometryIndexStateService` can select vertices/edges/sub-geometries, which can then be dragged around.

As you may have noticed from the descriptions, the GeometryEditState is used mainly within the controllers that operate on the map. An insert controller will only be active when the edit state is "INSERTING". Likewise a drag controller will only be active when the edit state is "DRAGGING".

4. Using snapping while editing

The editing plug-in has support for snapping while inserting or dragging. The controllers are equipped with a `SnapService` which can convert the mouse event locations into snapped locations, before they are passed to the `GeometryEditService` for operation execution.

The SnapService works through a series of rules that need to be active. Without any snapping rules, the SnapService will no snap. Adding snapping rules, goes through the "addSnappingRule" method, and requires the following parameters:

- *algorithm*: The snapping algorithm to be used. For example, snap to end-points only, or also to edges, or...
- *sourceProvider*: The provider of target geometries where to snap. For example, snap to features of a layer.
- *distance*: The maximum distance to bridge during snapping. Expressed in the unit of the map CRS.
- *highPriority*: High priority means that this rule will always be executed. Low priority means that if a previous * snapping algorithm has found a snapping candidate, this algorithm will not be executed anymore.

5. Merging geometries

6. Splitting geometries

Chapter 3. JavaScript Api for Editing

The editing plug-in also provides a JavaScript API for client-side integration with other technologies. The API resembles the Java/GWT API as closely as possible (package names are different).

1. Maven configuration

2. The GeometryEditService

The JavaScript counterpart of the GeometryEditService is basically a wrapper around the GWT version. It therefore has the same methods, and works the same way.

3. Using snapping

The snapping options in JavaScript are not as rich as they are in GWT. No separate SnappingService is available. What can be done, is configuring snapping options in the XML configuration of the vector layers, and using that configuration directly. The JavaScript "GeometryEditor" has methods for activating those snapping rules during editing.

On top of that, the GeometryEditor also has the ability to turn snapping on and off while inserting vertices or while dragging vertices. As with the GWT snapping you need at least one snapping rule for snapping to occur.

4. Merging geometries

The JavaScript counterpart of the GeometryMergeService is basically a wrapper around the GWT version. It therefore has the same methods, and works the same way.

5. Splitting geometries

The JavaScript counterpart of the GeometrySplitService is basically a wrapper around the GWT version. It therefore has the same methods, and works the same way.

Chapter 4. Java How-to

This chapter shows how to perform some of the most common editing operations.

1. How to create a new geometry

This section describes how to let the user draw a new geometry of some pre-defined type. The idea is that the user can click on the map to insert vertices into the geometry. This requires three steps:

1. Set up a `GeometryEditor` for the map. The editor is responsible for drawing the edited geometry and setting the correct event handler for capturing user events

2. Prepare an initial (empty) geometry for the editor. The editor always operates on a single geometry, which has to be set programmatically.

3. Prepare the intial state of editing. During the editing phase, the editor is in one of the 3 main states: idle (waiting for the user to select), inserting (inserting vertices) or dragging (dragging a part of the geometry). To start drawing on an empty geometry, the inserting state has to be activated and the insert index (index of the vertex that will be inserted) should be set.

The following code has to be executed:

```
GeometryEditor editor = new GeometryEditorImpl(map); // (1)
Geometry polygon = new Geometry(Geometry.POLYGON, 0, 0);  // (2)
editor.getEditService().start(polygon); // (2)
try {
    GeometryIndex index = editor.getEditService().addEmptyChild(); // (3)
    editor.getEditService().setInsertIndex(editor.getEditService().getIndexServ
    editor.getEditService().setEditingState(GeometryEditState.INSERTING); // (3
} catch (GeometryOperationFailedException e) {
    editor.getEditService().stop();
    Window.alert("Exception during editing: " + e.getMessage());
}
```

From there on, the user can take over. Depending on the use case, the editing service could be stopped by letting the user click outside the finished geometry

```
editor.getBaseController().setClickToStop(true);
```

or by explicitly stopping the service:

```
editor.getEditService().stop();
```

The editing happens on the same object that was originally passed to the service.

2. How to add an interior ring

The following steps have to be taken:

1. Add an extra empty ring to the polygon

2. Prepare the editing state to start inserting at the first child index of the ring

The following code assumes that the polygon being edited already has an exterior ring:

```
   try {
        GeometryEditService service = editor.getEditService();
        GeometryIndex ringIndex = service.addEmptyChild(); // (1)
```

```
            // Free drawing means inserting mode. First create a new empty child, t
            // index to the child's first vertex:
            service.setInsertIndex(service.getIndexService().addChildren(ringIndex,
            service.setEditingState(GeometryEditState.INSERTING);// (2)
        } catch (GeometryOperationFailedException e) {
            Window.alert("Error during editing: " + e.getMessage());
        }
```

3. How to delete an interior ring

The following steps have to be taken:

1. Find the correct index for the ring
2. Call the service to remove the ring

The following code deletes the first interior ring if the polygon has one:

```
GeometryEditService service = editor.getEditService();
Geometry geometry = service.getGeometry();
if(geometry.getGeometries().length > 1) {
    GeometryIndex ringIndex = service.create(GeometryIndexType.TYPE_GEOMETRY, 1
    service.remove(Collections.singletonList(ringIndex)); // (2)
}
```

Chapter 5. JavaScript How-to

This chapter shows how to perform some of the most common editing operations.

1. How to create a new geometry

This section describes how to let the user draw a new geometry of some pre-defined type. The idea is that the user can click on the map to insert vertices into the geometry. This requires three steps:

1. Set up a `GeometryEditor` for the map. The editor is responsible for drawing the edited geometry and setting the correct event handler for capturing user events
2. Prepare an initial (empty) geometry for the editor. The editor always operates on a single geometry, which has to be set programmatically.
3. Prepare the intial state of editing. During the editing phase, the editor is in one of the 3 main states: idle (waiting for the user to select), inserting (inserting vertices) or dragging (dragging a part of the geometry). To start drawing on an empty geometry, the inserting state has to be activated and the insert index (index of the vertex that will be inserted) should be set.

The following code has to be executed (`onGeomajasLoad()` is called automatically on page load):

```
var map; // The map object.
var editor; // Geometry editor. Holds the central editing service, the renderer
var service; // The central editing service.

function onGeomajasLoad() {
 map = Geomajas().createMap("app", "mapMain", "js-map-element");
 editor = new org.geomajas.plugin.editing.GeometryEditor(); // (1)
 editor.setMap(map); // (1)
 service = editor.getService();
}

function drawPolygon() {
  var geometry = new org.geomajas.jsapi.spatial.Geometry("Polygon", 0, 0); // (
  service.start(geometry);// (2)
  service.addEmptyChild(); // (3)
  var index = service.getIndexService().create("vertex", [0, 0]); // (3)
  service.setInsertIndex(index); // (3)
  service.setEditingState("inserting"); // (3)
}
```

From there on, the user can take over. Depending on the use case, the editing service could be stopped by letting the user click outside the finished geometry

```
editor.getBaseController().setClickToStop(true);
```

or by explicitly stopping the service:

```
editor.getEditService().stop();
```

The editing happens on the same object that was originally passed to the service.

2. How to add an interior ring

The following steps have to be taken:

1. Add an extra empty ring to the polygon

2. Prepare the editing state to start inserting at the first child index of the ring

The following code assumes that the polygon being edited already has an exterior ring:

```
function insertHole() {
  var geometry = service.getGeometry();
  var ringIndex = service.addEmptyChild(); // (1)
  var indexValue = ringIndex.getValue();
  var index = service.getIndexService().create("vertex", [indexValue, 0]); // (
  service.setInsertIndex(index); // (2)
  service.setEditingState("inserting"); // (2)
}
```

3. How to delete an interior ring

The following steps have to be taken:

1. Find the correct index for the ring

2. Call the service to remove the ring

The following code deletes the first interior ring if the polygon has one:

```
function deletefirstHole() {
    var geometry = service.getGeometry();
    if(geometry.getGeometries().length > 1) {
        var index = service.getIndexService().create("geometry", [1]); // (1)
        service.remove([index]); // (2)
    }
}
```

4. How to register a geometry handler

Geometry handlers are necessary to apply custom editing actions when certain mouse events occur. The concept is similar to normal map controllers, but geometry handlers add information about the geometry index of the part of the geometry on which the mouse event occurs. To register a geometry handler, the following steps have to be taken:

1. Create an instance of `org.geomajas.plugin.editing.handler.GeometryHandlerFactory`

2. Set mouse event functions for each of the mouse events that have to be captured

3. Register the factory.

4. Redraw the edited geometry by calling `editor.getRenderer().redraw()`.This is necessary to activate the handler.

5. Deregister the factory and call redraw when finished

The following code illustrates such registration in the case of a custom handler for deletion of interior rings.

```
function registerDeleteHole() {
    factory = new org.geomajas.plugin.editing.handler.GeometryHandlerFactory();
    factory.setUpHandler(function(event) {   // (2)
        var index = factory.getIndex();
        if (service.getIndexStateService().isMarkedForDeletion(index)) {
            try {
```

```
                    service.remove([index]);
                } catch (e) {
                    alert("Error occurred while deleting the inner ring: " + e.getM
                }
            }
        });
        factory.setMouseOverHandler(function(event) {  // (2)
            var index = factory.getIndex();
            var geometryType = service.getIndexService().getGeometryType(service.ge
            if (geometryType == "LinearRing") {
                if (service.getIndexService().getValue(index) > 0) {
                    // Only inner rings must be marked. The outer shell can remain a
                    service.getIndexStateService().markForDeletionBegin([index]);
                }
            }
        });
        factory.setMouseOutHandler(function(event) {  // (2)
            var index = factory.getIndex();
            if (service.getIndexStateService().isMarkedForDeletion(index)) {
                service.getIndexStateService().markForDeletionEnd([index]);
            }
        });
        EditingHandlerRegistry().addGeometryHandlerFactory(factory);  // (3)
        editor.getRenderer().redraw();  // (4)
    }

    function unRegisterDeleteHole() {
        if(factory) {
            EditingHandlerRegistry().removeGeometryHandlerFactory(factory); // (5)
            editor.getRenderer().redraw();  // (5)
        }
    }
```