

# **Geomajas Javascript API plug-in guide**

**Geomajas Developers and Geosparc**

---

# **Geomajas Javascript API plug-in guide**

by Geomajas Developers and Geosparc

1.0.0-SNAPSHOT

Copyright © 2011-2012 Geosparc nv

---

---

# Table of Contents

1. Introduction .....	1
2. Using the JavaScript API plug-in .....	2
1. Stand-alone or on top of GWT? .....	2
1.1. Stand-alone JavaScript library .....	2
1.2. JavaScript API on top of a GWT application .....	3
2. Configuring the JavaScript API for a GWT application .....	3
3. What can the API do? .....	4
3. How-to .....	5
1. How to add a map to the HTML page .....	5
2. How to retrieve the list of layers .....	5
3. How to check for client-server communication .....	6
4. How to change cursors on the map .....	6
5. How to search features in a layer .....	6
5.1. Searching features by ID .....	6
5.2. Searching features using bounds .....	7
6. How to use existing map controllers .....	7
7. How to create custom map controllers .....	8
8. How to navigate the map through code .....	9
8.1. How to translate the map .....	9
8.2. How to zoom in and out .....	9
8.3. How to apply a new bounding box .....	9
9. How to select features, or connect to selection events .....	9
9.1. How to select or deselect features .....	10
9.2. How to be notified of feature selection events .....	10

---

## List of Examples

2.1. Add a map to the HTML page .....	2
2.2. Plug-in dependency .....	3
2.3. Adding the GWT module .....	3
3.1. Initializing a map when Geomajas loads .....	5
3.2. Reacting to the map initialization event .....	5
3.3. Removing a handler registration .....	6
3.4. Reacting to client-server communication .....	6
3.5. Removing handler registrations .....	6
3.6. Applying cursors .....	6
3.7. Searching features by ID .....	7
3.8. Searching features by bounds .....	7
3.9. Using pre-defined map controllers .....	7
3.10. Creating custom map controllers .....	8
3.11. Map controller activation/deactivation .....	8
3.12. Switching to the fallback map controller .....	9
3.13. Translating the map (panning) .....	9
3.14. Zooming in and out .....	9
3.15. Applying a new bounding box .....	9
3.16. Selecting features .....	10
3.17. Deselecting features .....	10
3.18. Is a feature selected? .....	10
3.19. Catching feature selection events .....	10
3.20. Removing the event handler .....	11

---

# Chapter 1. Introduction

The goal of this plug-in is to provide a pure JavaScript API for the Geomajas client.

By default, Geomajas uses the GWT technology to develop client side code in. Since the GWT Java code is by default compiled into unreadable and obfuscated JavaScript code, this plug-in basically provides a wrapper around the GWT code wherein names and packages are pre-defined.

A secondary goal for this plug-in is to provide the same API for all Geomajas GWT based faces. This means that for all GWT based faces, there will be a different implementation using the same API.

## **Note**

At first this plug-in will focus on the implementation for the GWT face, later for the PureGWT face.

---

# Chapter 2. Using the JavaScript API plug-in

This chapter will focus on how to integrate this plug-in in different environments. In essence there are 2 main options to choose from:

- Using the stand-alone version of this JavaScript library in combination with a Geomajas back-end.
- Using the JavaScript API on top of a GWT application.

Both options provide totally different ways of integrating the Geomajas JavaScript API within an existing architecture. Both options will be explained in detail below.

## 1. Stand-alone or on top of GWT?

As explained above, there are 2 main ways of using this JavaScript API within your applications. Both provide a totally different approach to build and integrate Geomajas application within an existing architecture. As a result it is important to understand both options in order to better apply the reasoning on your situation.

### 1.1. Stand-alone JavaScript library

This option is pretty straightforward as you could regard it as any other JavaScript library out there: make sure the browser loads the correct JavaScript files and CSS classes and you are up-and-running...except that the Geomajas server is still required.

Remember that the JavaScript API is a wrapper around the GWT face or PureGWT face, both of whom make heavy use of the Geomajas server. Consequently this JavaScript API requires the same server to be present.

When we say "the Geomajas server", all we mean is some server that provides the communication options the Geomajas back-end libraries provide. This means that you can use any Java based server application, as long as the Geomajas back-end Jars are in there somewhere, or you emulate the Geomajas client-server communication completely (not recommended).

Anyway, at the heart of the stand-alone JavaScript API, there will be a completely empty GWT application definition. As it's completely empty, it does not generate any GUI whatsoever. You would typically start out by creating a new map and attaching it to an HTML element.

#### Example 2.1. Add a map to the HTML page

```
var map;  
  
function onGeomajasLoad() {  
    map = Geomajas().createMap("app", "mapMain", "js-map-element");  
}
```

The line above will create a new map within the "app" configuration, therein searching the "mapMain" map definition and attaching it to a HTML element with ID="js-map-element". The "app" and "mapMain" parameters are defined within the server-side XML configuration.

The method "onGeomajasLoad()" is automatically called when the page loads, so the above code would fetch the map configuration on load and then attach the map object onto the "js-map-element" DOM element.

This option cuts GWT out of the equation. Choose this approach if you don't want to use GWT but still want a strong GIS framework on the web.

## 1.2. JavaScript API on top of a GWT application

This option can be used in multiple ways. Basically you would create a GWT application with a JavaScript API. This means that you could walk the middle road and use a mix of Java and JavaScript, or more likely, you want to write a full blown GWT application, but you still need it to be accessible from within JavaScript (to be used in an IFrame of another application perhaps).

This approach will have you build your application in GWT, as if you're using the GWT face or PureGWT face, and just have the JavaScript API in there for external parties that require client-side integration. Let it be said that the Geomajas team is always a strong proposer of back-end integration, as it provides all integrating within a secured environment, but sometimes you simply don't have that choice....

After building the GWT application, the JavaScript API will be available for use as if it were a stand-alone JavaScript library (as in the previous section).

*Read on to find out how to configure a GWT application to provide the JavaScript API.*

## 2. Configuring the JavaScript API for a GWT application

When using the JavaScript API on top of a GWT application, you basically have to set up your GWT application and then add the JavaScript API to it. This means getting the correct jars on the classpath and adding the GWT module for the JavaScript API to the GWT compiler.

Here's an example Maven configuration of getting the jars on the classpath (example for the GWT face):

### Example 2.2. Plug-in dependency

```
<dependency>
  <groupId>org.geomajas.plugin</groupId>
  <artifactId>geomajas-plugin-javascript-api-gwt</artifactId>
  <version>1.0.0</version>
</dependency>

<dependency>
  <groupId>org.geomajas.plugin</groupId>
  <artifactId>geomajas-plugin-javascript-api-gwt</artifactId>
  <version>1.0.0</version>
  <classifier>sources</classifier>
</dependency>
```

Note that both the jars and sources jars have to be added. As always the GWT compiler requires the original Java code to be on the classpath in order to compile it to JavaScript.

Next you have to actually register the JavaScript API to the GWT compiler. This is done by adding the JavaScript API GWT module to the application, by adding the following line to the application's .gwt.xml file (example for the GWT face):

### Example 2.3. Adding the GWT module

```
<inherits name="org.geomajas.plugin.jsapi.gwt.JavascriptApi"/>
```

That's it! When building your GWT application, it will automatically publish it's map API to JavaScript.

## Note

The configuration in this section adds the JavaScript API for the GWT face to your application. The JavaScript API for the PureGWT face is not yet ready.

## 3. What can the API do?

Ofcourse, this API is a subset of the shared functionality of the GWT face and PureGWT face. It is after all a wrapper around the Java API. Therefore, it will not exceed the Java functionalities. Having said that, the API is still pretty extensive. It provides the most access to the map, the layer model, the viewport, the features, the geometries, etc. It also provides the most important events (map initialization event, feature selection events, ...).

On top of the default map API, more plug-ins are and will become available for the JavaScript API. The editing plug-in for example also has a JavaScript API which extends this API (providing rich geometry editing services).

## Tip

For a detailed list of all classes and methods available, have a look at the JavaScriptDoc (those should be available at the plug-in page on the Geomajas website).

---

# Chapter 3. How-to

This chapter provides a series of examples on how to use the JavaScript API to accomplish different goals. Many of the examples in here are also available in the showcase application that comes with this plug-in.

## 1. How to add a map to the HTML page

Adding a map when the page loads, can be done through the following code:

### Example 3.1. Initializing a map when Geomajas loads

```
var map;  
  
function onGeomajasLoad() {  
    map = Geomajas().createMap("app", "mapMain", "js-map-element");  
}
```

The "onGeomajasLoad" method is called automatically after Geomajas has loaded. This is the perfect time to add a map to the HTML page. The parameters are defined as follows:

- *"app"*: The XML application configuration as defined in the Geomajas back-end configuration.
- *"mapMain"*: The map definition as defined in the Geomajas back-end configuration.
- *"js-map-element"*: The DOM element wherein to place the map. It is best to make sure this element has a fixed width and height.

## 2. How to retrieve the list of layers

It is important to understand that when the Geomajas client loads, the map is absolutely empty. This is due to the fact that Geomajas fetches its map configuration from the server. When the Geomajas client loads, the first action it undertakes is to fetch the correct map configuration, and only when it receives this from the server, can the map initialize itself. Only then are the layers available.

Fortunately the map will fire an event when its configuration is loaded. Actually it fires an event every time its configuration changes. It might be the case that during run-time, the back-end map configuration changes, so this map "configuration change event" might occur multiple times.

The following example shows how to react to this event:

### Example 3.2. Reacting to the map initialization event

```
var registration = map.getEventBus().addLayersModelChangedHandler(function(event) {  
    for (var i = 0; i < map.getLayersModel().getLayerCount(); i++) {  
        var layer = map.getLayersModel().getLayerAt(i);  
        // Do something with the layer: layer.getTitle()...  
    }  
});
```

Note first of all that the map has a central EventBus for events. This is where such event handlers can be registered. The map also has a "LayersModel" which represents the list of layers and which provides methods that operate on layers.

Also note that registering handlers (of any event type), will always return a registration to that handler. Through this registration it is possible to remove the handler from the EventBus again:

**Example 3.3. Removing a handler registration**

```
registration.removeHandler();
```

## 3. How to check for client-server communication

Whenever the Geomajas client starts and stops talking to the server, it fires events. It is possible to catch these events to, for example, show some busy state within the application.

**Example 3.4. Reacting to client-server communication**

```
var handlerRegistration1 = Geomajas().addDispatchStartedHandler(function(event)
    // Client-server communication starts. Quickly, do something!
});

var handlerRegistration2 = Geomajas().addDispatchStoppedHandler(function(event)
    // Client-server communication stops. Quickly, do something!
});
```

Whenever you register an event handler, a registration is returned. If at any time, the event handler is not needed anymore, you can cleanly remove it again through the following statement:

**Example 3.5. Removing handler registrations**

```
handlerRegistration1.removeHandler();
handlerRegistration2.removeHandler();
```

## 4. How to change cursors on the map

It is possible to change the cursor on the map, either through default cursors or an image:

**Example 3.6. Applying cursors**

```
map.setCursor('default');
map.setCursor('images/metal-cursor.cur');
```

## 5. How to search features in a layer

This section will describe 2 methods for searching features. Every map has a `FeatureSearchService`, which is used for feature retrieval.

### 5.1. Searching features by ID

In the example above, we assume that there already is a map present, and that this map has already initialized itself. By initialized, we mean that it has already fetched its configuration from the server (otherwise layers are not available).

In the example below, we search for a feature with ID "feature-ID-1" within the second layer of the map. Note that an array is expected when searching features by ID, so that multiple features can be looked up in a single request. When the answer returns from the server, a callback is executed with a `featureHolder` that contains (hopefully) the requested features.

### Example 3.7. Searching features by ID

```

var layer = map.getLayersModel().getLayerAt(1); // make sure this is a vector layer
var service = map.getFeatureSearchService();

service.searchById(layer, ["feature-ID-1"], function(featureHolder) {
    if (featureHolder == null) {
        // Feature "feature-ID-1" could not be found....
    } else {
        // Let's see what we have...
        var feature = featureHolder.getFeatures()[0];
    }
});

```

## 5.2. Searching features using bounds

Another option is to search for features within a certain bounding box. In the example below, the map's current bounding box is used to search features in. As in the previous example, a callback will be executed when the answer from the server returns.

### Example 3.8. Searching features by bounds

```

var layer = map.getLayersModel().getLayerAt(1); // make sure this is a vector layer
var bounds = map.getViewPort().getBounds(); // Get the current map bounds
var service = map.getFeatureSearchService();

service.searchInBounds(layer, bounds, function(featureHolder) {
    if (featureHolder == null) {
        // "No features found...
    } else {
        for (var i=0; i < featureHolder.getFeatures().length; i++) {
            var feature = featureHolder.getFeatures()[i];
            // Quickly, do something!
        }
    }
});

```

## 6. How to use existing map controllers

The top Geomajas service has the possibility to create instances of some of the default controllers that the GWT faces provide. These controllers can then be applied onto the map:

### Example 3.9. Using pre-defined map controllers

```

var mapController = Geomajas().createMapController(map, id);
map.setMapController(mapController);

```

The default controllers are created using a pre-defined ID. This is the list of supported ID's:

- *'PanMode'*: The default navigation controller. Allows to pan by dragging and zoom in and out using the mouse wheel.
- *'MeasureDistanceMode'*: Controller that let's the user measure distances.
- *'FeatureInfoMode'*: Controller that fetches feature information by clicking on the map.
- *'SelectionMode'*: Selection controller that supports multiple selection. Supports selection by rectangle by dragging on the map, and also supports additional selection through the shift button.

- *SingleSelectionMode*: Selection controller that only allows a single feature to be selected at any given time.
- *EditMode*: A generic editing controller. Works through the right mouse button. Note that an editing plug-in is available with far more powerful editing tools.

## 7. How to create custom map controllers

It is also possible to create a custom controller to react on mouse events from the map. Next to the default mouse events, it is also possible to attach handlers for activation and deactivation of your controller. Let us first analyze the creation of a new custom controller that picks up on the location of the mouse:

### Example 3.10. Creating custom map controllers

```
// Create the new MapController:
var mapController = new org.geomajas.jsapi.controller.MapController();

// Apply handlers for Mouse Events:
mapController.setMouseMoveHandler(function(event) {
    // Screen space: pixel position
    var screenLocation = mapController.getLocation(event, "screen");

    // World space: real-world location
    var worldLocation = mapController.getLocation(event, "world");

    // Do something with these locations...
    // worldLocation.getX(), worldLocation.getY(), ...
});
mapController.setMouseOverHandler(function(event) {
    // Do something useful...
});
mapController.setMouseOutHandler(function(event) {
    // Do something useful...
});
mapController.setDownHandler(function(event) {
    // Do something useful...
});
mapController.setUpHandler(function(event) {
    // Do something useful...
});
```

Now that we have created our controller, we might also want to be notified of the controllers' activation and deactivation events. On deactivation we might have to clean up the controller.

### Example 3.11. Map controller activation/deactivation

```
// Apply activation (for init) and deactivation (for cleanup) handlers:
mapController.setActivationHandler(function() {
    // on controller activation: initialize something.
});
mapController.setDeactivationHandler(function() {
    // on controller deactivation: clean up.
});

// Apply the MapController on the map:
map.setMapController(mapController);
```

The above code will set the activation and deactivation handlers and finally apply the controller on the map. In order to revert to the fallback controller on the (which by default is a navigation controller), set a "null" controller on the map:

**Example 3.12. Switching to the fallback map controller**

```
map.setMapController(null);
```

## 8. How to navigate the map through code

Navigating the map through JavaScript code requires the ViewPort. The ViewPort keeps track of current position and scale and provides methods for navigation. This section provides a few examples of how to use the ViewPort.

### 8.1. How to translate the map

Translating (a.k.a. panning) the map is done by first acquiring the current position, then adding up the translation deltas and finally applying the new position again:

**Example 3.13. Translating the map (panning)**

```
var position = map.getViewPort().getPosition();
var newX = position.getX() + distanceX;
var newY = position.getY() + distanceY;
var newPosition = new org.geomajas.jsapi.spatial.Coordinate(newX, newY);
map.getViewPort().applyPosition(newPosition);
```

### 8.2. How to zoom in and out

Zooming in or out is done by first acquiring the current scale and then multiplying or dividing it by some scale delta:

**Example 3.14. Zooming in and out**

```
var scale = map.getViewPort().getScale();
map.getViewPort().applyScale(scale * delta);
```

Applying a bigger scale value will zoom in, applying a smaller scale value will zoom out.

### 8.3. How to apply a new bounding box

Applying a new bounding box is pretty straightforward. Just create a new bounding box with the required values (position) and apply it on the ViewPort:

**Example 3.15. Applying a new bounding box**

```
var bbox = new org.geomajas.jsapi.spatial.Bbox(x, y, width, height);
map.getViewPort().applyBounds(bbox);
```

## 9. How to select features, or connect to selection events

This section will tackle some of the typical issues regarding feature selection, deselection and how to be notified of feature selection events.

## 9.1. How to select or deselect features

In order to select a feature, you must first acquire it. The following example therefore will select a feature after searching for it on the back-end. If you already have the feature available in your JavaScript code, there is no need to search it again.

### Example 3.16. Selecting features

```
// Get the layer wherein to search and select a feature:
var layer = map.getLayersModel().getLayer("clientLayerCountries");

// Search a feature by ID and select it once we have it:
map.getFeatureSearchService().searchById(layer, [id], function(featureHolder) {
    layer.selectFeature(featureHolder.getFeatures()[0]);
});
```

For more information on searching for features, see "How to search features in a layer". The key in this example is that you select features through their layer.

In order to deselect the feature again, there are 2 options: deselect the specific feature or deselect all features within the layer:

### Example 3.17. Deselecting features

```
layer.deselectFeature(feature);

// or

layer.clearSelectedFeatures();
```

It might also be of interest to know whether or not a specific feature is selected or not. This you can ask:

### Example 3.18. Is a feature selected?

```
var selected = eval(layer.isFeatureSelected(id).toString()); // Boolean to boolean
```

Note that the feature ID is used to inquire about the feature selection state. Also the eval method of the toString method is used to convert the Boolean to a JavaScript boolean. This is a clumsy conversion that is the result of the Java to JavaScript conversion within the GWT client.

## 9.2. How to be notified of feature selection events

Whenever a feature is selected or deselected on the map (either through code or through the user who uses the SelectionController on the map), a feature selection or deselection event is fired. It is possible to register handlers to these events in order to catch them.

### Example 3.19. Catching feature selection events

```
var registration = map.getEventBus().addFeatureSelectionHandler(function(event) {
    var feature = event.getFeature();
    // The feature is selected. Do something with it...
}, function(event) {
    var feature = event.getFeature();
    // The feature is deselected. Do something with it...
});
```

The map's central EventBus stores all handlers and fires all events. For feature selection, you may have noticed that both the selection and deselection events are registered at once.

Also note that registering handlers (of any event type), will always return a registration to that handler. Through this registration it is possible to remove the handler from the EventBus again:

**Example 3.20. Removing the event handler**

```
registration.removeHandler();
```