JMap 6.5

Developer Manual

Table of Contents

Introduction

The JMap SDK is comprised of documents, source code examples and tools to help developers customize JMap applications and extend their functionality.

JMap is made up of several different parts. Its main parts are: JMap Server, JMap Admin, JMap Pro, JMap Web, and JMap Mobile. Each part is based on its own technological environment, which determines how programming is carried out.

The following table describes the technological environment of each JMap component.

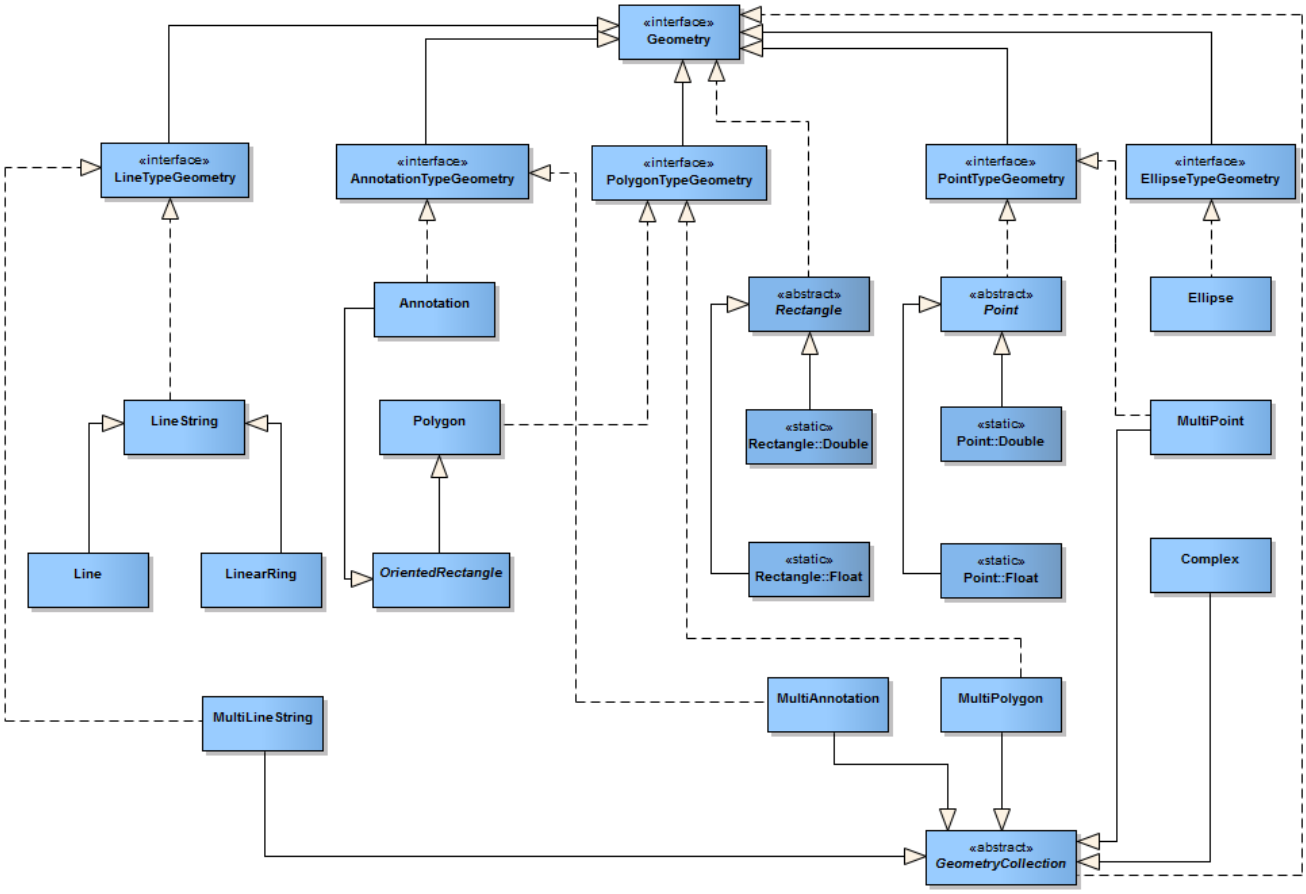
Component	Technologies
JMap Server	Java
JMap Admin	Java Server Faces (JSF)
JMap Pro	Java
JMap Web	HTML 5, CSS, Javascript, JSON

Note: At the moment, it is not possible to develop for JMap Mobile using the JMap 6.5 SDK.

General Concepts

The API of JMap geometries is used for development purposes in JMap Pro and JMap Server.

In JMap, all vector data elements that appear on the map are based on geometries. They are simple classifications of basic geometric elements such as points, lines, and surfaces. In JMap, the classification scheme of geometries has been largely inspired by the model of geometries published by the Open Geospatial Consortium. Geometries do not contain attributes, identifiers, or display properties, but only two-dimensional x and y coordinates.



Modèle de classes simplifié des géométries de JMap

In JMap, the geometry classes can be found in the *com.kheops.jmap.spatial* package. The interface implemented by all geometries is *Geometry* .

The *Point* is a type of basic geometry used to compose all geometries. This type of geometry only contains a set of coordinates (x,y). This class is abstract and, therefore, its derivatives— *Point.Float* (simple precision) and *Point.Double* (double precision)—must be used.

```
// Creating a new point specifying coordinates
Point pt1 = new Point.Double(-73., 45.);

// Creating a new point by cloning an existing point
Point pt2 = (Point) pt1.clone();

// Changing the location of an existing point
pt1.setLocation(-74., 45);
```

The main geometry classes are the following:

Geometry classes	
Point	x,y coordinates used to compose all other geometries.
Curve	Abstract type from which all linear geometries are derived (Line, LineString, etc.).
Line	Simple line defined by two points.
LineString	A line with several parts. Composed of N nodes and N-1 segments.
LinearRing	A Line String where the first and last nodes are equal. Forms a closed loop.
Surface	Abstract type from which all surface geometries (rectangle, polygon, etc.) are derived.
Polygon	Polygon composed of one exterior LinearRing and of 0, 1 or several LinearRings that are holes.
Rectangle	Non-oriented rectangular surface (horizontal and vertical sides)
OrientedRectangle	A rectangular surface that can be oriented.
Ellipse	Ellipse composed of a central point, an a radius and a b radius.
Annotation	Derived from *OrientedRectangle* , the *Annotation* class defines the extent of a message to be displayed on the map.

Composite versions of geometries exist to support collections of geometries of the same type. These classes are: *MultiPoint* , *MultiCurve* , *MultiLineString* , *MultiSurface* and *MultiPolygon* . In addition, the *Complex* class is a special type of collection comprising various types of geometries.

To simplify the management of all these types of geometries, 5 interfaces are available: *PointTypeGeometry* , *LineTypeGeometry* , *PolygonTypeGeometry* , *AnnotationTypeGeometry* and *EllipseTypeGeometry* . These interfaces combine all types of geometries, including collections.

Geometry precision

JMap allows you to create single precision or double precision geometries. Combined with other strategies, simple precision is mainly used to compress data in order to optimize system performance. As a programmer, you should only use double precision.

```
// Creating a new point with single precision
Point pt1 = new Point.Float(-73., 45.);

// Creating a new point with double precision
Point pt2 = new Point.Double(-73., 45.);
```

Operations on geometries

Spatial operators

Spatial operators are used to make geometric calculations on the geometries. These calculations can generate 3 types of results:

- Numeric results (e.g. calculate the distance between 2 geometries)
- Boolean results (e.g. test to see if 2 geometries intersect)
- Geometric results (calculate the union of 2 geometries)

The *GeometryUtil* class provides simple methods to perform calculations on geometries. The methods most often used are presented in the following table:

Spatial relation tests (boolean)	
contains(Geometry, Geometry, PrecisionModel)	Test to determine if the first geometry contains the second geometry. The order of the geometries is important.
crosses(Geometry, Geometry, PrecisionModel)	Test to determine if the first geometry crosses the second geometry. The order of the geometries is important.
disjoint(Geometr, Geometry,	Test to determine if the first geometry is disjointed from the second geometry. The order of the geometries is important.

PrecisionModel)	
intersects(Geometry, Geometry, PrecisionModel)	Test to determine if the two geometries intersect. The order of the geometries is not important.
overlaps(Geometry, Geometry, PrecisionModel)	Test to determine if the two geometries overlap. The order of the geometries is not important.
relate(Geometry, Geometry, char[], PrecisionModel)	Test to determine if the two geometries possess the spatial relationships defined in the parameter matrix. The order of the geometries is important. For more information, refer to http://en.wikipedia.org/wiki/DE-9IM.
spatiallyEquals(Geometry, Geometry, PrecisionModel)	Test to determine if the two geometries are spatially equal (all the coordinates are identical). The order of the geometries is not important.
touches(Geometry, Geometry, PrecisionModel)	Test to determine if the two geometries touch each other. The order of the geometries is not important.
within(Geometry, Geometry, PrecisionModel)	Test to determine if the first geometry is within the second geometry. The order of the geometries is important.
Geometry operations resulting in new geometries	
buffer(Geometry, double, double,	Returns a geometry that represents a buffer zone around the specified geometry. The size of the zone is specified as a parameter.

PrecisionModel)	
convexHull(Geometry, boolean, PrecisionModel)	Returns a convex geometry created from the specified geometry.
difference(Geometry, Geometry, PrecisionModel)	Returns the geometry resulting from the difference between the first and second geometries. The order of the parameters is important.
diffSym(Geometry, Geometry, PrecisionModel)	Returns the geometry resulting from the symmetrical difference between the two geometries. The order of the parameters is not important.
intersection(Geometry, Geometry, PrecisionModel)	Returns the geometry resulting from the intersection of the two geometries. The order of the parameters is not important.
intersectionMultiple(Geometry[], PrecisionModel)	Returns the geometry resulting from the intersection of all geometries passed as parameters. The order of the parameters is not important.
union(Geometry, Geometry, PrecisionModel)	Returns the geometry resulting from the union of the two geometries. The order of the parameters is not important.
unionMultiple(Geometry[], PrecisionModel)	Returns the geometry resulting from the union of all geometries passed as parameters. The order of the parameters is not important.
Calculations	

distance(Geometry, Geometry, PrecisionModel)	Calculates the shortest distance separating both geometries. The order of the parameters is not important.

Precision models

The *PrecisionModel* class allows you to define the tolerance being considered in performing calculations using the geometries. Using a precision model that is well adapted to the geometries can give more precise results when making calculations.

The precision model generally varies according to the measurement unit of the data (meters, degrees, etc.). The *PrecisionModelFactory* class allows you to obtain an instance of *PrecisionModel* for a given unit. Each instance of the *Layer* class has its own instance of *PrecisionModel* that can be used. It is also possible to ask JMap to determine the optimal precision model for a specific geometry.

The following example shows different ways of generating an appropriate precision model.

```
// Obtain the optimal precision model for data of the current project
Project project = ...
PrecisionModel precisionModel1 = PrecisionModelFactory.getInstance(project.getMapUnit())

// Obtain the optimal precision model for unit Meter
PrecisionModel precisionModel2 = PrecisionModelFactory.METER;

// Obtain the optimal precision model of a layer
Layer layer = ...
PrecisionModel precisionModel3 = layer.getPrecisionModel();

// Obtain the optimal precision model for the specified geometry
Geometry geometry = ...
PrecisionModel precisionModel4 = PrecisionModelFactory.getInstance(geometry);
```

Transformations

You can perform transformations of all sorts on geometries. The *Geometry* interface contains the method *transform(Transformation),* which receives an instance of *Transformation* . This method creates a clone of the geometry, applies the specified transformation on it and returns the newly transformed geometry. Transformations can change the coordinates of the geometry and even modify the nature of the geometry itself. They can be used to apply a map projection or a generalization to geometries, for instance.

It is also possible to apply transformations using the *transform()* method of the *Transformation* class. In this case, the geometry may not always be cloned, depending on the type of transformation. Refer to the documentation of each transformation for more information.

The *UnaryTransformation* class is the base class of the transformations that only modify the coordinates composing the geometry. If you must implement your own transformation class, you will probably derive it from this class.

The following table shows the transformations that are available with the JMap API.

Available transformations

ProjectionTransformation	Applies the projection specified in the constructor's parameters to a geometry. The projection can also be applied in reverse.
OffsetTransformation	Applies an x and y translation to a geometry, as specified in the constructor's parameters.
SmoothTransformation	Applies a generalization to a geometry, according to the tolerance specified in the constructor's parameters.
PrecisionTransformation.TO_FLOAT	Transforms a double precision geometry into a single precision geometry.
PrecisionTransformation.TO_DOUBLE	Transforms a single precision geometry into a double precision geometry.

The following code example shows how to use the precision change transformation.

```
Point[] points = new Point[]
{
  new Point.Double(10., 10.),
  new Point.Double(20., 20.),
  new Point.Double(30., 30.)
};

LineString doubleLineString = new LineString( points );

// The transformation can be done by the geometry. After the execution, doubleLineString
// will still contain single precision coordinates.
LineString singleLineString1 = (LineString)doubleLineString.transform(PrecisionTransform

// The transformation is made directly on the geometry. After the execution, doubleLineS
// will contain single precision coordinates. Moreover, singleLineString2 refers to the
// instance as doubleLineString.
LineString singleLineString2 = (LineString)PrecisionTransformation.TO_FLOAT.transform(do
```

The following code example shows how to use the map projection change transformation.

```
Point[] points = new Point[]
{
  new Point.Double(-73.12345, 45.12345),
  new Point.Double(-73.54321, 45.54321)
};

LineString lineString = new LineString( points );

// Define the source and the destination projections
Projection fromProjection = new LongitudeLatitude();

Projection toProjection = new Mtm();
toProjection.setParams("8"); // zone 8

// Create a combined projection object that converts points from one projection
// to the other
final CombinedProjection combinedProj = new CombinedProjection(fromProjection, toProject

// Apply the projection transformation
lineString = (LineString)lineString.transform(new ProjectionTransformation(combinedProj)
```

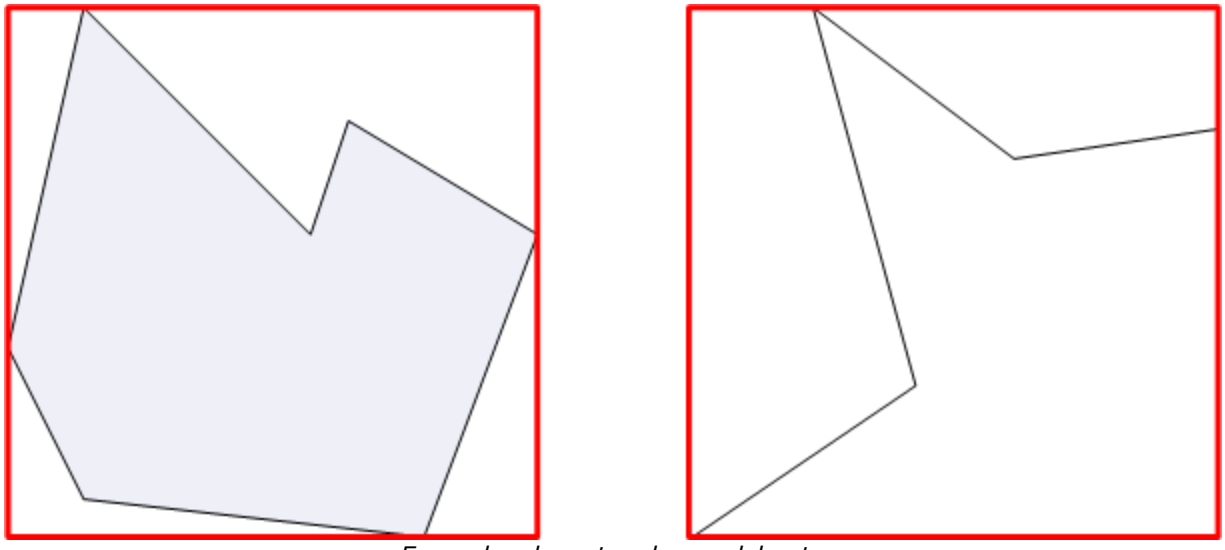
The following example shows how to create an anonymous transformation class using the adapter *TransformationAdapter* . The transformation converts the points into ellipses.

```
Point[] points = new Point[]
{
  new Point.Double(10., 10.),
  new Point.Double(20., 20.),
  new Point.Double(30., 30.)
};

// Create a new transformation which creates a new ellipse at the
// specified location.
Transformation tr = new TransformationAdapter()
{
  public Geometry transform(Point point)
  {
    return new Ellipse(
      (Point)point.clone(), new Dimension.Double(5, 3), 0
    );
  }
};

// Apply the transformation on all points
Ellipse[] ellipses = new Ellipse[points.length];
for (int i = 0; i < ellipses.length; i++)
  ellipses[i] = (Ellipse)tr.transform(points[i]);
```

Bounding rectangle

The bounding rectangle (MBR for Minimum Bounding Rectangle) of a geometry is the smallest orthogonal rectangle that totally encompasses the geometry. The MBR is used in JMap to do quick spatial analyses of a large number of geometries before using more precise (yet slower) algorithms on a smaller number of geometries. As prescribed by the Geometry interface of JMap, all types of geometries implement the getBounds() method that generates the MBR of the geometry.

The MBR of a point is a rectangle with a width of 0 and a height of 0. The MBR of a vertical line is a rectangle with a width of 0 and the MBR of a horizontal line is a rectangle with a height of 0.



Exemples de rectangles englobants

JMap Pro and JMap Server Development

The purpose of this section is to guide Java developers in developing mapping applications based on JMap Pro and JMap Server 6.5 using JMap's Java API.

Although the JMap Pro application can be customized, it is strongly recommended to implement all new features and customizations by developing extensions with JMap's Java API.

JMap Pro is a Java-based client application. It can run as a Java applet in a browser, as a standalone JavaWebStart (JNLP) application or as a standalone application launched at the command line (which is especially useful for development). In all cases, the application is the same, and all extensions developed are compatible.

JMap Server is a server application developed in Java. You can develop extensions for JMap Server using JMap's Java API.

Ant

For programming, the JMap 6.5 SDK uses Ant to perform various tasks using scripts. Ant can be downloaded from http://ant.apache.org.

When used from Eclipse, Ant may be unable to find the Java compiler in order to compile classes. In such a case, you must add a reference to the tools.jar library provided with the Java JDK in the Ant configuration. For more information, refer to this article: http://wiki.eclipse.org/FAQ_Why_can't_my_Ant_build_find_javac%3F.

Eclipse

Eclipse is the preferred Java development environment for JMap. It can be downloaded from this address: http://www.eclipse.org. However, it is quite possible to develop JMap applications using the environment and tools of your choice. The following information applies to the *Eclipse Luna* (version 4.4) environment.

Integrate JMap 6.5 SDK as an Eclipse project

To simplify development, it is recommended to integrate the JMap 6.5 SDK as an *Eclipse* project. Your development should be in projects that are separate from the SDK to keep it intact as a source of reference.

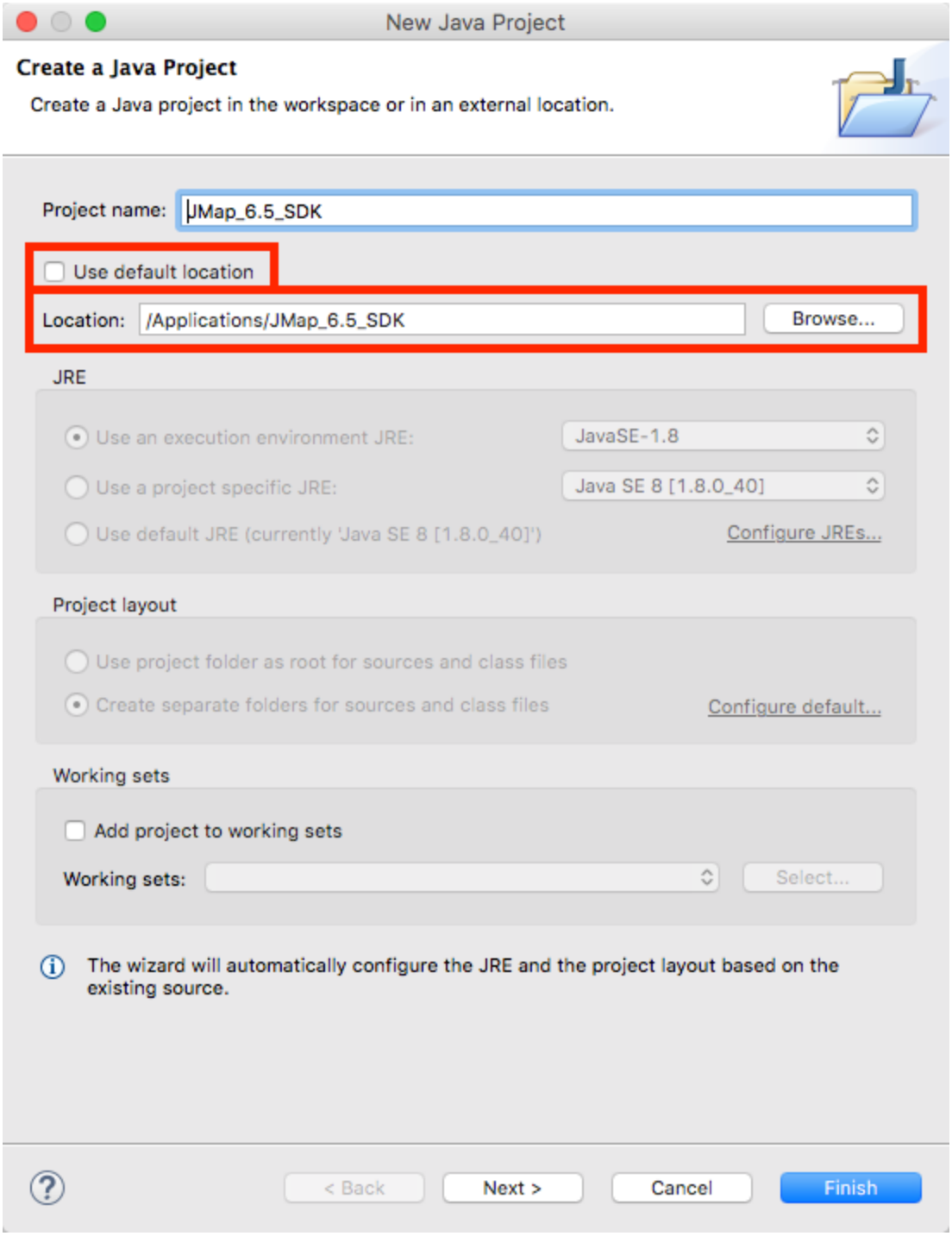
Step 1

In an Eclipse worskpace (existing or new), create a Java project for the JMap 6.5 SDK.

File -> New -> Java Project

Step 2

Select the path of the JMap 6.5 SDK that was specified when it was installed. To do this, you must disable the *Use default location* option to let Eclipse browse a folder outside of the current workspace. The project name is set automatically when the folder is selected. Press *Finish* .



Create a new project

To start new development work using the JMap 6.5 SDK (such as creating a new JMap extension), it is recommended to create separate projects. Each new project should point to the libraries

distributed with the SDK to allow for compilation. The example below shows how to create a new JMap project.

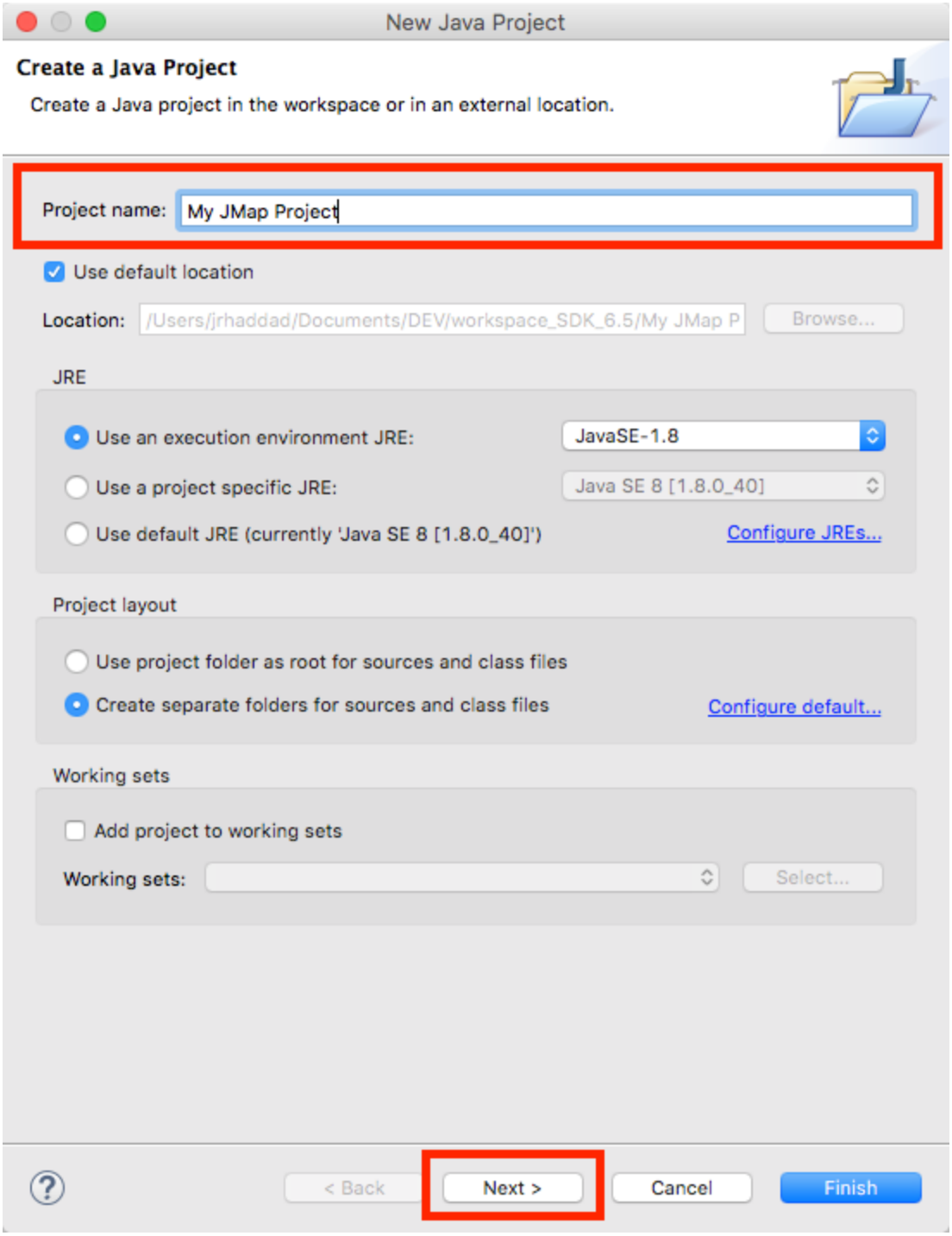
Step 1

In the same Eclipse workspace containing the project for the JMap 6.5 SDK, create a new Java project.

File -> New -> Java Project

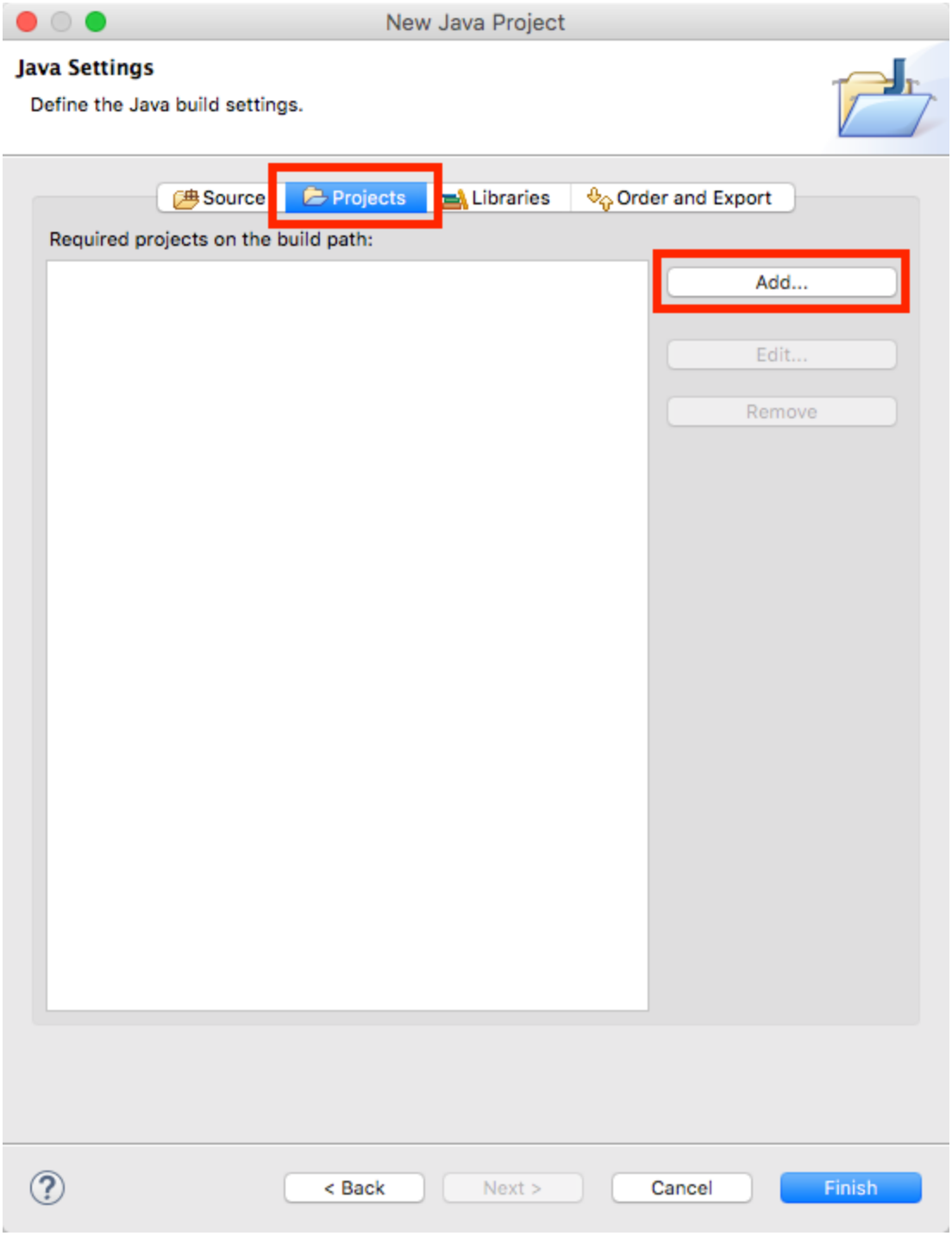
Step 2

Give your project a name then press ***Next*** .



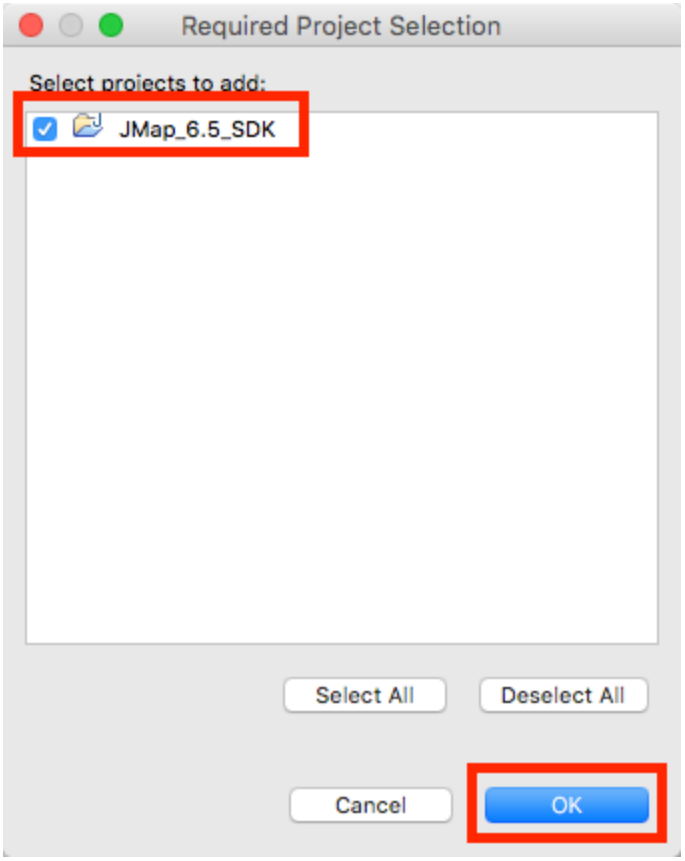
Step 3

Select the ***Projects*** tab and press ***Add...*** .



Step 4

Select the JMap SDK project. All of the SDK's libraries will become available for your project. Press ***OK***.



Step 5

Press ***Finish*** .

The JMap 6.5 SDK comes with many examples located in the JDK_HOME/examples folder.

The Ant script ***build_examples.xml*** is used to compile and run the examples.

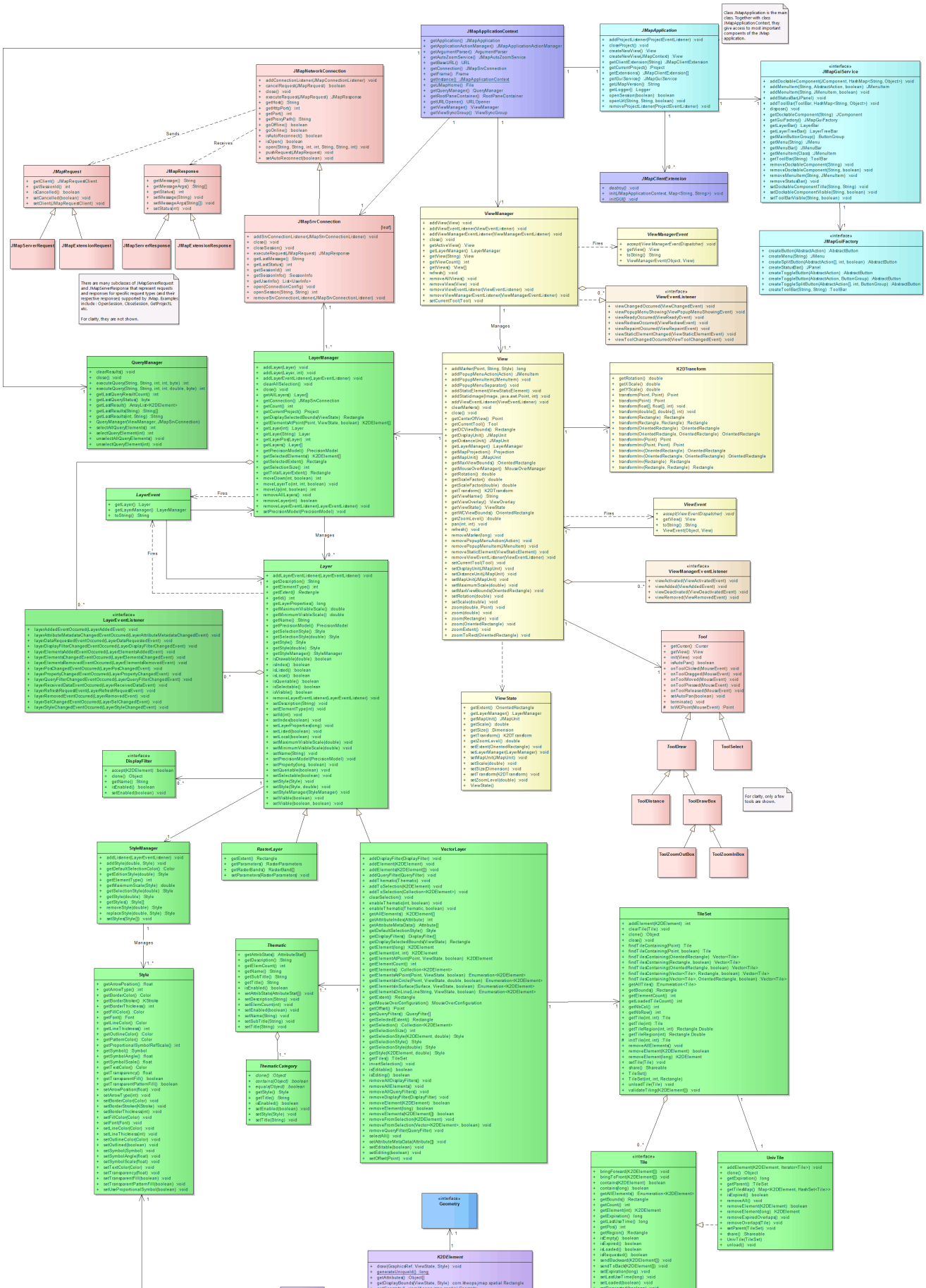
Many simple examples can be run in JMap Pro from a unique GUI using the ***Showcase*** extension.

The ***Hello World*** extension demonstrates a basic JMap Pro client extension that communicates with a JMap Server extension.

The ***Webextensions*** extension shows how to develop an action that can respond to JMap Web queries (refer to JMap Web Development).

The following class diagram shows a simplified version of JMap Pro's architecture.

JMap Pro Simplified Class Diagram



In JMap, map elements are objects (points, lines, polygons, text, etc.) that make up the map. The classes of these elements are all derived from the abstract class *K2DElement*. Map elements are organized into layers that are displayed in a JMap application within a view (View class). Each map element is associated with a geometry (*Geometry* interface) and has a digital identifier as well as a certain number of attributes. The map element does not directly contain any geometry information. Instead, it is the associated geometry that contains all of the geometry coordinates. Element identifiers found within the same layer must be unique. The *GeneratedUniqueId()* utility method of the *K2DElement* abstract class allows you to generate a sequence of unique identifiers for the elements.

Creating map elements

The following example demonstrates how to create map elements:

```
// Create a point geometry
Point pointGeometry = new Point.Double(100., 100.);

// Create the element using the geometry, null attributes and auto generated id
K2DPoint point = new K2DPoint(pointGeometry, null, K2DElement.generateUniqueId());

// Create a line geometry
Point pointGeometry1 = new Point.Double(100., 100.);
Point pointGeometry2 = new Point.Double(200., 200.);
Line lineGeometry = new Line(pointGeometry1, pointGeometry2);

// Create some attributes
String attrib1 = "some value";
Integer attrib2 = new Integer(999);

// Create the element using the geometry, 2 attributes and auto generated id
K2DPolyline line = new K2DPolyline(lineGeometry, new Object[] {attrib1, attrib2},
                                   K2DElement.generateUniqueId());
```

Element attributes

Map elements normally have values for their attributes. These values are the descriptive data of the map elements. All elements within the same layer possess the same list of attributes (same names, same types). Note that the elements contain only the values of the attributes (table or chart of objects) and not their definition. The definition of these attributes is managed at the layer level, using the *Attribute* class.

The list of attributes for the elements of a layer is determined by the JMap administrator when the layer (linked attributes) is created. The types of attributes are defined using Java constants for the SQL types (*java.sql.Types* class).

Element attributes are used for many functions such as tooltips, labels, thematics, and filtering.

The following example demonstrates how to access the values of element attributes:

```
K2DElement element = ...

Object[] attributeValues = element.getAttributes();
for (int i = 0; i < attributeValues.lenght; i++)
{
  System.out.println(i + " : " + attributeValues[i]);
}
```

The following example demonstrates how to access the definition of a layer's attributes.

```
VectorLayer layer= ...

Attribute[] attributes = layer.getAttributeMetaData();
for (int i = 0; i < attributes .lenght; i++)
{
  System.out.println(i + " name : " + attributes[i].getName());
  System.out.println(i + " title : " + attributes[i].getTitle());
  System.out.println(i + " type: " + attributes[i].getType());
}
```

Element style

When elements are drawn on the map, a style object is used to determine the visual aspects of the elements. Instances of the Style class have properties such as line color, fill color, letter font, transparency, etc. Each layer will have one or more styles. The style used for displaying the elements is based on the scale of the map and whether or not it contains thematics. Thematics dictate the style of the displayed elements based on the attribute values of these elements.

The following example demonstrates how to obtain the style used for a layer according to a particular scale. Here the presence of thematics on the layer is not taken into account.

```
K2DElement element = ...
VectorLayer layer = ...

Style style = layer.getStyle(view.getScaleFactor());
```

The following example demonstrates how to obtain the style used to display a specific element at a given scale. The resulting style does not take into account the presence of thematics on the layer.

```
K2DElement element = ...
VectorLayer layer = ...

Style style = layer.getStyle(element, view.getScaleFactor());
```

Style properties can be modified by calling the methods of the *Style* class. The following example demonstrates how to modify the style parameters of a layer of polygons at the current map scale.

```
VectorLayer layerOfPolygons = ...

Style style = layerOfPolygons.getStyle(view.getScaleFactor());

style.setBorderColor(Color.BLACK);
style.setFillColor(Color.BLUE);
style.setTransparency(.5f); // 50% transparent
```

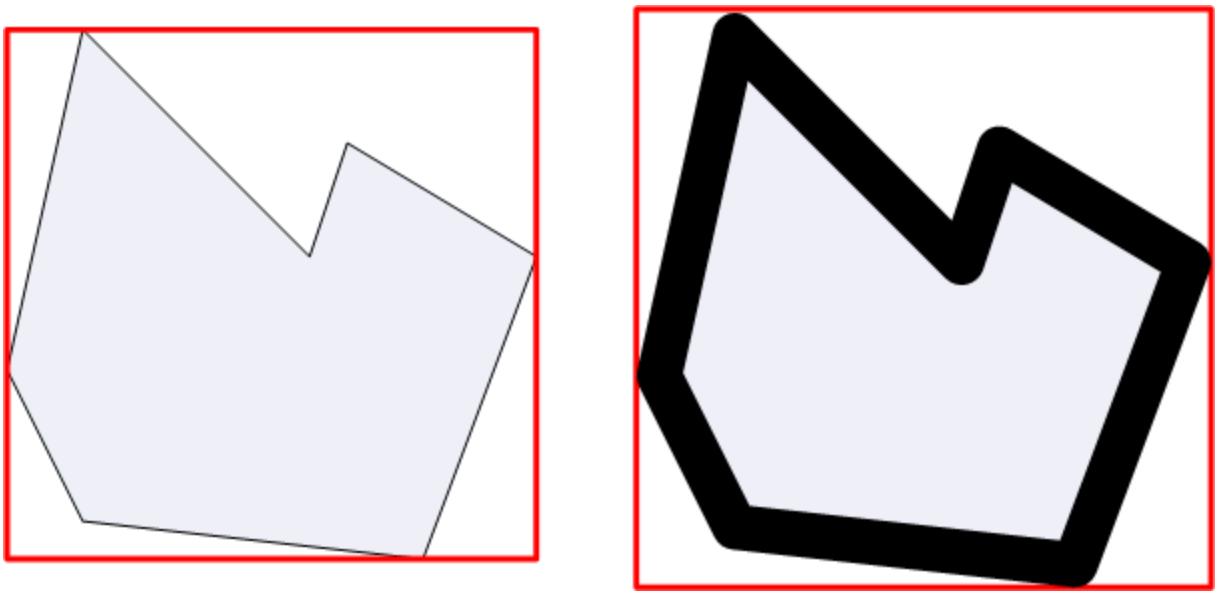
Bounding rectangle on display

The bounding rectangle of a displayed map element is the smallest rectangle in device coordinates (DC) that completely contains the element, taking its style into account. The style of an element influences the bounding rectangle displayed. For example, a thick polygon border will increase the size of the bounding rectangle, and the size of a point symbol will determine the size of the bounding rectangle on display, and so forth.

The following code example shows how to obtain the bounding rectangle of a displayed map element:

```
K2DElement element = ...
View view = ...
VectorLayer layer = ...

Rectangle displayBounds = element.getDisplayBounds(view.getTransform(),
                                                   layer.getStyle(element, view.getScaleF
```

Note that the *getDisplayBounds* method takes the transformation of the view (refer to Coordinate Systems) and element style as parameters.



Rectangles englobants à l'affichage du même polygone avec des styles différents

Selection

The map elements of vector layers can be selected. A variety of tools allow the user to select elements in the vector layers of a JMap project.

The *K2DElement* class has a property called *selected* that indicates whether or not an element is selected. Selected elements are displayed using a different style, which is the selection style of the vector layer.

Vector layers manage the list of their selected elements. The API of the *VectorLayer* class offers several methods related to selecting elements. See *Layers and Layer Manager* for more information on this topic.

The following source code example shows how to select and unselect elements.

```
K2DElement element = ...
VectorLayer layer = ...

// add the element to the current selection
layer.addToSelection(element);

// test if element is selected, useless, just to demonstrate
boolean isSelected = element.isSelected();

// cycle through selection
Collection selection = layer.getSelection();
for (K2DElement element : selection)
  System.out.println(element));

// unselect element
layer.unselectElement(element);
```

```
// clear layer selection
layer.clearSelection();
```

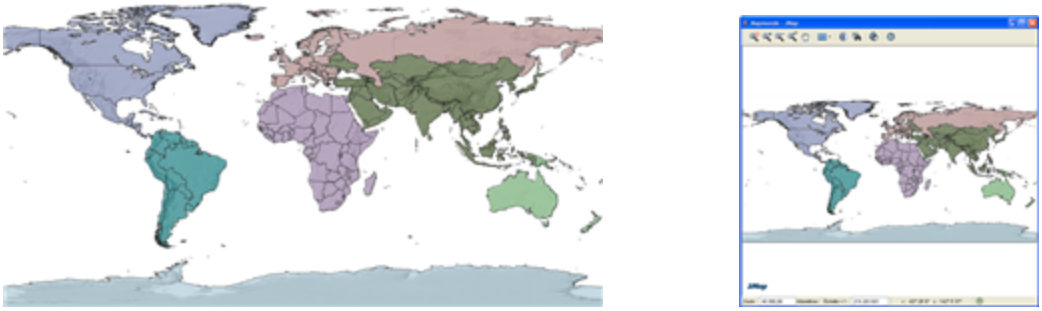
Styled elements

Styled elements (*K2DStyledElement* class) are special map elements. They have their own style and they ignore the style of the layer that contains them when they are displayed. They are useful when programmatically adding elements to the map that have their own style. Aside from this difference, styled elements behave exactly like any other map elements.

The following coding example shows how to create styled elements.

```
K2DElement element = ...
Style style = ...

K2DStyledElement styledElem = new K2DStyledElement(element, style); // element will use t
```

Two coordinate systems are used in JMap Pro programming: the WC or World Coordinates system and the DC or Device Coordinates system. The WC system is the system used for the original data and the DC system is used for the screen that will display the map.



The WC (World Coordinates) system versus the DC (Device Coordinates) system

WC

All the geometry coordinates in JMap are in WC. For example, if your data uses the MTM zone 8 projection, the values of the coordinates will be similar to (300 000, 5 000 000). If your data is not projected, it will be in longitude and latitude and its range will be from -180 to 180 degrees east-west and from -90 to 90 degrees north-south

DC

The computer screen used to display the map is divided into pixels with the coordinates (0,0) showing on the top left part of the screen. This is the DC system. When you are working with

mouse-clicked events (i.e., MouseClicked) in a JMap application, you are working with DC coordinates contained in the event and expressed in pixels.

Transforming coordinates between WC and DC

In JMap programming, WC coordinates must frequently be transformed into DC and vice-versa. When the map elements of a layer are drawn on the screen, their coordinates are converted from WC to DC on the fly in order to light up the appropriate pixels on the screen. However, when a mouse click occurs on the map, the DC coordinates of the mouse cursor are transformed into WC coordinates in order to select the appropriate element on the map.

The *K2DTransform* class contains an affine transformation matrix used to convert data between the DC and WC systems. It provides methods to transform the coordinates in both directions. Each map in JMap (*View* class) has its own transformation instance.

Main methods of the K2DTransform class

transform(OrientedRectangle)	Creates a new oriented rectangle from the specified WC point and transforms it into a DC point.
transform(Point)	Creates a new point from a WC point and transforms it into a DC point.
transform(Rectangle)	Creates a new rectangle from a specific WC rectangle and transforms it into DC.
transformInv(OrientedRectangle)	Creates a new oriented rectangle from the specified DC point and transforms it into a WC point.
transformInv(Point)	Creates a new point from a specific DC point and transforms it into a WC point.
transformInv(Rectangle)	Creates a new rectangle from a specific DC rectangle and transforms it into WC.

The following source code example demonstrates how to transform WC coordinates into DC.

```
K2DTransform transform = ...
Point pointWC;
Point pointDC;

// Transform a point from WC to DC
pointWC = new Point.Double(100., 100.);
pointDC = transform.transform(pointWC);

// Transform a point from DC to WC
```

```
pointDC = new Point.Double(100., 100.);
pointWC = transform.transformInv(pointDC);
```

The following source code example demonstrates how to transform DC coordinates resulting from a mouse click event to WC. The transformation is obtained from the view.

```
View view = ...
K2DTransform transform = view.getTransform();
MouseEvent e = ...

Point pointWC = transform.transformInv(new Point.Double(e.getX(), e.getY()));
```

The following source code example demonstrates how to transform DC coordinates using a mouse click event. This simple method is only available from a tool class derived from the *Tool* class.

```
Point pointWC = Tool.toWCPoint(e);
```

Rectangle transformations (normal or inverted) must be performed with caution: if a rotation is applied to a view, the rectangle that is returned could be overwritten. To prevent this problem, it is preferable to make calculations on an oriented rectangle initialized from the rectangle to be transformed.

```
View view = ...

final ViewState viewState = view.getViewState();

// Computes the display bounding box of the selection
final Rectangle displaySelectBounds = view.getLayerManager().getDisplaySelectedBounds(vie

if (displaySelectBounds != null)
{
  // Transforms the display bounding box into a WC extent.
  final OrientedRectangle selectBounds = viewState.getTransform().transformInv(
    new OrientedRectangle.Double(displaySelectBounds)
  );

  view.zoom(selectBounds);
  view.refresh()
}
```

View class

In JMap, the View class is the main GUI component responsible for displaying the map. The methods of this class allow you to control the map (zoom, pan, etc.), to refresh it, to get information on the scale, etc.

The table below shows the main methods of the *View* class.

Navigation	
zoom(double)	Performs a zoom at the map centre using the specified magnifying factor. For example, a factor of 2 will display a map that is two times closer. A value of 0.5 will produce a map that is two times farther. The map must be refreshed afterwards.
zoom(Rectangle)	Pans and zooms the map around the specified rectangle. The map must be refreshed afterwards.
zoomExtent()	Pans and zooms the map to show the entire area covered by the data. The map must be refreshed afterwards.
moveTo(double, double)	Makes the specified coordinates the new centre of the map view. The scale is not affected.The map must be refreshed afterwards.
moveTo(Point)	Makes the specified coordinates the new centre of the map view. The scale is not affected. The map must be refreshed afterwards.
pan(int, int)	Moves the map based on the x and y values specified in screen coordinates (DC). The map must be refreshed afterwards.
setScale(double)	Modifies the map scale based on the specified value. The map must be refreshed afterwards.
Display	
refresh()	Refreshes the map. After any method modifying the state of the map is called, the map must be refreshed in order to display the changes.
addMarker(Point, String, Style)	Adds a marker (symbol indicating a location) at the specified coordinates with the specified message and style.
removeMarker(long)	Removes the specified marker from the map.
clearMarkers()	Removes all markers from the map.

getDCViewBounds()	Returns the view bounds to screen coordinates (DC).
getWCViewBounds()	Returns the view bounds to world coordinates (WC).
getTransform()	Returns the view transformation. This is the transformation used to convert Screen Cooordinates (DC) to World Coordinates (WC) and vice-versa.
getViewOverlay()	Returns the view overlay. The overlay is a special layer designed to display volatile data. It is often used to create smooth animations (moving objects, drawing, etc.).
getScaleFactor()	Returns the current scale factor of the view as the denominator (1: denominator).
getViewState()	Returns the instance of the *ViewState* class associated with this view. See below for more information on this topic.
getZoomLevel()	Returns the current horizontal distance of the view in world coordinates (WC).
Other	
getLayerManager()	Returns the *LayerManager* instance associated with the view.
getMapProjection()	Returns the map projection (Projection) of the data on the map.
getMapUnit()	Returns the unit (JMapUnit) of the data on the map.
removePopupMenuAction(Action)	Removes an action from the view's pop-up menu.
removePopupMenuItem(JMenuItem)	Removes an item from the view's pop-up menu.
addPopupMenuAction(Action)	Adds an action to the view's pop-up menu.
addPopupMenuItem(JMenuItem)	Adds an item to the view's pop-up menu.
addViewEventListener(ViewEventListener)	Registers a listener to the events generated by the view.
removeViewEventListener(ViewEventListener)	Removes a listener from the events generated by the view.
setCurrentTool(Tool)	Replaces the active tool of the view with the tool specified as a parameter.

Events of the View class

In JMap, a view generates events in several situations. To get these events, you must register a *listener* to the view. To get events generated by all open views, it is recommended to register a *listener* on the View Manager. Read below for more information on this subject.

To receive a view's events, you must implement the *ViewEventListener* interface and register it with the view using the *addViewEventListener()* method, as shown in the example below.

```
View view = ...

view.addViewEventListener(new ViewEventListener()
{
  @Override
  public void viewToolChangedOccurred(ViewToolChangedEvent e)
  {
  }

  ...
});
```

Note that the *ViewAdapter* adapter can also be used to simplify the development of a listener.

The table below shows the events triggered by the *View* class.

View events

viewChangedOccurred (ViewChangedEvent)	Launched after the view's state changes following a navigation operation (zoom, pan, etc.).
viewToolChangedOccurred (ViewToolChangedEvent)	Launched when the active tool of the view is replaced by another tool.
viewRedrawOccurred (ViewRedrawEvent)	Launched when the view is redrawn, in whole or in part. The elements of the layers are redrawn.
viewRepaintOccurred (ViewRepaintEvent)	Launched when the view is repainted, in whole or in part. The image of the map is simply refreshed; the elements of the layers are not redrawn.
viewStaticElementChanged (ViewStaticElementEvent)	Launched when static elements (north arrow, scale bar, etc.) are added to or removed from the view.
viewPopupMenuShowing(ViewPopupMenuShowingEvent)	Launched just before the pop-up menu of the view is displayed. Allows you to change the contents of the menu before it is presented to the user.
viewReadyOccurred(ViewReadyEvent)	Launched when the view becomes ready for use, i.e. after the first time it is displayed.

For more information on events in Java, see the Java Tutorial.

ViewState class

The *ViewState* class contains some view properties that define its state at a given time (scale, scope, transformation, etc.). Several methods in the JMap API take an instance of ViewState as a parameter. It is possible to get the state of a view by calling the *getViewState()* method of the *View* class.

ViewManager class

The view manager (*ViewManager* class) is responsible for managing one or more views that are present in the application. This class has methods that allow you to perform operations on all the views of the application and to know at any time which view is active (i.e. in focus).

The table below shows the main methods of the *ViewManager* class.

Most commonly used methods of the ViewManager class

addViewEventListener(ViewEventListener)	Registers a listener to the events triggered by the active view.
addViewManagerEventListener(ViewManagerEventListener)	Registers a listener to the events triggered by the view manager (see below).
getActiveView()	Returns the active view (i.e. the view that is in focus).
getLayerManager()	Return the layer manager of the active view.
getView(String)	Returns the view with the name specified as a parameter.
getViews()	Returns a list of all existing views in the view manager.
refresh()	Performs a refresh action on all existing views in the view manager.
removeViewEventListener(ViewEventListener)	Removes a listener from the events triggered by the active view of the view manager.
removeViewManagerEventListener(ViewManagerEventListener)	Removes a listener from the events triggered by the view manager.
setCurrentTool(Tool)	Replaces the active tool of the active view with the tool specified as a parameter.

Events of the ViewManager class

The View Manager triggers events associated with the state of its views. To get these events, you must implement the *ViewManagerEventListener* interface and register it with the view manager using the *addViewManagerEventListener()* method, as shown in the following example.

```
View view = ...
ViewManager viewManager = view.getLayerManager();

viewManager.addViewManagerEventListener(new ViewManagerEventListener()
{
  @Override
  public void viewAdded(ViewAddedEvent e)
  {
  }

  ...
});
```

Note that the *ViewManagerAdapter* adapter can also be used to simplify the development of a listener.

The table below shows the events triggered by the *ViewManager* class.

View manager events

viewAdded(ViewAddedEvent)	Launched when a view is added to the View Manager.
viewActivated (ViewActivatedEvent)	Launched when a view is activated (i.e. comes into focus).
viewDeactivated (ViewDeactivatedEvent)	Launched when a view has been deactivated (i.e. it has lost focus in favor of another view).
viewRemoved(ViewRemovedEvent)	Launched when a view is removed from the View Manager.

It is also possible to register a listener to the View Manager to get the events generated by all the views associated with the layer manager. This may be more convenient than registering a listener on each view separately. To do so, use the *addViewEventListener* method of the *ViewManager* class.

ViewUtil class

The *ViewUtil* class provides useful methods to work with the view.

The table below shows the mains methods of the *ViewUtil* class.

Most commonly used methods of the ViewUtil class

mailMap(View, JMapSrvConnection,	Sends an image of the map by email to the recipients defined in the *MailMessage* type object passed as a parameter.

int, MailMessage)	
makeImage(View, int)	Produces and returns an image of the map as displayed in the view. The width (in pixels) of the image to be created is passed as a parameter.
saveAs(View)	Saves an image of the map on the hard drive. A window displays, allowing the user to select the location of the file.

Concepts

Layers (*Layer* class and its related classes) are highly important in JMap programming. They contain and manage the map data displayed on the map (*View* class). There are two main types of layers: vector layers (VectorLayer class), which contain vector data, and raster layers (RasterLayer class), which contain raster data (images).

When a project is loaded, the client application gets the layer configuration from the server and creates the related instances in the *LayerManager*. Initially, a layer does not contain any data. Its data will only be loaded, in full or in part, after the view has been refreshed, based on the loading mode and display restrictions (visibility state and display thresholds).

In JMap, there are two different modes to load layers: by tile and by region. Vector layers support both modes, but raster layers can only be loaded by region.

Tiled layers divide the region of the layer into rows and columns; each cell is a data tile (TileSet and Tile classes). Since the layers are initially empty at application startup, data tiles will be loaded by the application when they are initially displayed in a view. Once it is loaded in a layer, a tile will be kept in memory for the duration of the session, unless the session expires prematurely or the memory manager decides to delete the tile.

When a data tile request is sent to JMap Server, all geometries (Geometry interface) intersecting the region of the tile will be transferred to the client. Once the tile is received, the geometries will be transformed into elements (K2DElement class), which will then be loaded in the TileSet of the layer. To avoid duplicating an element in several data tiles, elements that are not completely included in a tile will eventually be moved to the layer's universe tile.

Layers loaded by region are based on the same structure (TileSet and Tile classes), but they only have a single tile with a variable region. When a change is applied to the transformation matrix of the view and the view is refreshed, layers loaded by region will reload their single tile with the data intersecting the extent of the view.

The Layer class and its derived classes

Layer class

The Layer class is abstract and therefore cannot be instantiated. However, it provides the basic methods for all layers.

Most commonly used methods of the Layer class

addLayerEventListener (LayerEventListener)	Adds a listener to the events generated by the layer.
getElementType()	Returns the type of elements contained in the layer. Types are defined by the constants of the *ElementTypes* class.
getId()	Returns the unique numerical ID of the layer.
getName()	Returns the layer name.
getStyleManager()	Returns the instance of the Style Manager (*StyleManager* class) used by the layer. It manages all of the layer styles, which define the graphical appearance of elements on the map.
getNextUserLayerId()	Static method that generates a new unique identifier for a user layer.
invalidate()	Invalidates the cache of a layer loaded by region. This allows the layer to be refreshed when no changes have been made to the view transformation matrix.
isDrawable(double)	Indicates if the layer is displayed at the specified scale according to the defined display thresholds.
isSelectable()	Indicates whether or not the layer objects are selectable.
isVisible()	Indicates whether or not the layer is visible.
toLayerInfo()	Returns a serializable data structure containing all of the layer's information.

VectorLayer class

The *VectorLayer* class is derived from the *Layer* class and contains only vector data. It has several specialized methods for vector data to generate element selections, perform spatial analyses, etc.

Vector layers have a set of attributes that are common to all elements of the layer. These attributes provide the descriptive data of the layer's elements.

Most commonly used methods of the VectorLayer class

addElement(K2DElement)	Adds the specified element on the layer.
addElements(K2DElement[])	Adds all specified elements on the layer.

addToSelection(K2DElement)	Adds the specified element to the list of selected elements.
addToSelection(Collection<K2DElement>)	Adds a collection of elements to the list of selected elements.
clearSelection()	Clears the list of selected objects on this layer.
getAttributeMetaData()	Returns the list of attributes (*Attribute class*) of the layer.
addDisplayFilter(DisplayFilter)	Adds a new display filter (*DisplayFilter* interface) to the layer. This filter allows you to define what will be displayed and what won't.
getElementAtPoint (Point, ViewState, boolean)	Provides the first element of the layer detected at the specified coordinates in WC in the specified map (*ViewState* class). The last parameter determines if the elements that are invisible (due to a display filter, for example) should be considered or not.
getElements()	Returns a collection containing all of the layer's elements.
getElementsAtPoint (Point WCCoord, ViewState viewScale, boolean onlyIfVisible)	Returns all of the layer's elements identified at the specified WC coordinates in the specified map (*ViewState* class). The last parameter determines if the elements that are invisible (due to a display filter, for example) should be considered or not.
getElementsInSurface (Surface, ViewState, boolean)	Returns all of the layer's elements that intersect the surface specified in WC coordinates in the specified map (*ViewState* class). The last parameter determines if the elements that are invisible (due to a display filter, for example) should be considered or not.
getExtent()	Returns the total extent of the layer data as a rectangle.
getSelectedElements()	Returns an array containing all the elements selected on the layer.

RasterLayer class

The *RasterLayer* class is derived from the Layer class and contains only raster data. Its methods are rarely used by JMap application developers.

Most commonly used methods of the RasterLayer class

getExtent()	Returns the total extent of layer data as a rectangle.
getParameters()	Returns the raster parameters (*RasterParameters* class) used by the raster layer (e.g. the size of the image, transparency, etc.).
getRasterBands()	Returns an array of attributes for the various bands (*RasterBand* class) contained in the image.

Layer class events

The layers in JMap generate events in several situations. To get the events generated by a layer, you must register a listener on the layer. To get the events generated by all the layers of the project, it is recommended to register a listener on the layer manager. See below for more information on this subject.

To receive the events of a layer, you must implement the *LayerEventListener* interface and register it with the layer using the *addLayerEventListener()* method, as shown in the following example:

```
Layer layer = ...

layer.addLayerEventListener(new LayerEventListener()
{
  @Override
  public void layerSelChangedEventOccurred(LayerSelChangedEvent e)
  {
    // TODO
  }
  ...
});
```

There is also an adapter (*LayerAdapter*) that simplifies the development of the listener. The *LayerEvent* superclass has a *getLayer()* method and a *getLayerManager()* method allowing you to access the layer or layer manager that generated the event

The following table shows the events triggered by the *Layer* class.

Most commonly used events of the Layer class

layerElementsAddedEventOccurred(LayerElementsAddedEvent)	Launched after new elements are added to the layer. The added elements are accessible in the instance of the event.
layerElementsChangedEventOccurred(LayerElementsChangedEvent)	Launched after changing elements on the layer. Changed items are available in the instance of the event.
layerElementsRemovedEventOccurred(LayerElementsRemovedEvent)	Launched after removing elements from the layer. The removed elements are available in the instance of the event.

layerSelChangedEventOccurred(LayerSelChangedEvent)	Launched after changing the selection on the layer; this can include selected or unselected events. The elements in question are available in the instance of the event.
layerStyleChangedEventOccurred(LayerStyleChangedEvent)	Launched after changing the style of the layer.

LayerManager class

In JMap, each map (*View* class) has a layer manager (*LayerManager* class). The latter is responsible for managing all of the layers to be shown on the map, as well as their order and hierarchical organization. In addition, the layer manager can listen for events on all layers and perform certain operations on them.

Most commonly used methods of the LayerManager class

addLayer(Layer)	Adds the specified layer to the highest position.
addLayerEventListener (LayerEventListener)	Adds a listener to events generated by the layer manager. The layer manager also relays all the events generated by the layers it manages.
clearAllSelection()	Clears the list of selected elements on each layer.
getAllLayers()	Returns the ordered set of layers (user layers and normal layers). The layer at the zero position is the bottom layer.
getLayer(int)	Returns the layer having the specified unique identifier.
getLayer(String)	Returns the layer having the specified unique name.
getLayerPos(Layer)	Returns the position of the specified layer.
getLayerTreeVisibility()	Returns a data structure (*LayerVisibilitySet* class) containing the visibility state of a layer. This state considers the state of the selection of groups in the layer hierarchy as well as the configuration of their visibility.
getSelectedElements()	Returns all the selected elements on all layers.
getSelectedExtent()	Returns the extent of all selected elements on all layers.
removeLayer(int)	Removes the layer located at the specified position.

LayerManager class events

The layer manager in JMap generates events. To receive the events generated by a layer manager, you must register a listener on the appropriate layer manager. Note that a listener

registered on the layer manager will get the events generated by all the layers handled by the layer manager.

To receive the layer manager's events, you must implement the *LayerEventListener* interface and register it with the layer manager using the *addLayerEventListener()* method.

Events of the layer manager

layerPosChangedEventOccurred (LayerPosChangedEvent)	Launched after changing the position of a layer in the list of layers handled by the layer manager.
layerRemovedEventOccurred (LayerRemovedEvent)	Launched after removing a layer from the list of layers handled by the layer manager.
layerAddedEventOccurred (LayerAddedEvent)	Launched after adding a layer to the list of layers handled by the layer manager.

Concepts

JMap Pro applications are developed on a modular basis to simplify the addition of new features. Applications can be broken down into three levels, two of which are scalable to allow for programming additional features.

The first level is the entry point of the application (JMapApplicationLauncher class), which covers the type of application (applet, Java Web Start or standalone Java) and instantiates the application class (instance of the JMapApplication class) to be used. The second level is the JMap Pro application; it is driven by its abstract class, JMapApplication, which provides all the services required to ensure its proper operation. Since the JMapApplication class does not provide the application's graphical interface, it is necessary to instantiate a class that inherits the JMapApplication class and that will instantiate the application's graphical components, including the layer hierarchy and button toolbars. The DockingClient class, included in this SDK, is a good example of a JMap Pro application. The third level is for JMap extensions. JMap Pro applications allow you to develop and use extension classes (JMapClientExtension class) to add new features to applications. JMap Pro extension development is explained in the following section.

Communication with JMap Server

When the application is initialized, a connection with JMap Server is established based on the application parameters that have been specified. Communication with the server is unidirectional

and allows for exchanging messages using requests and responses. The details of this communication are explained in this section.

Application services

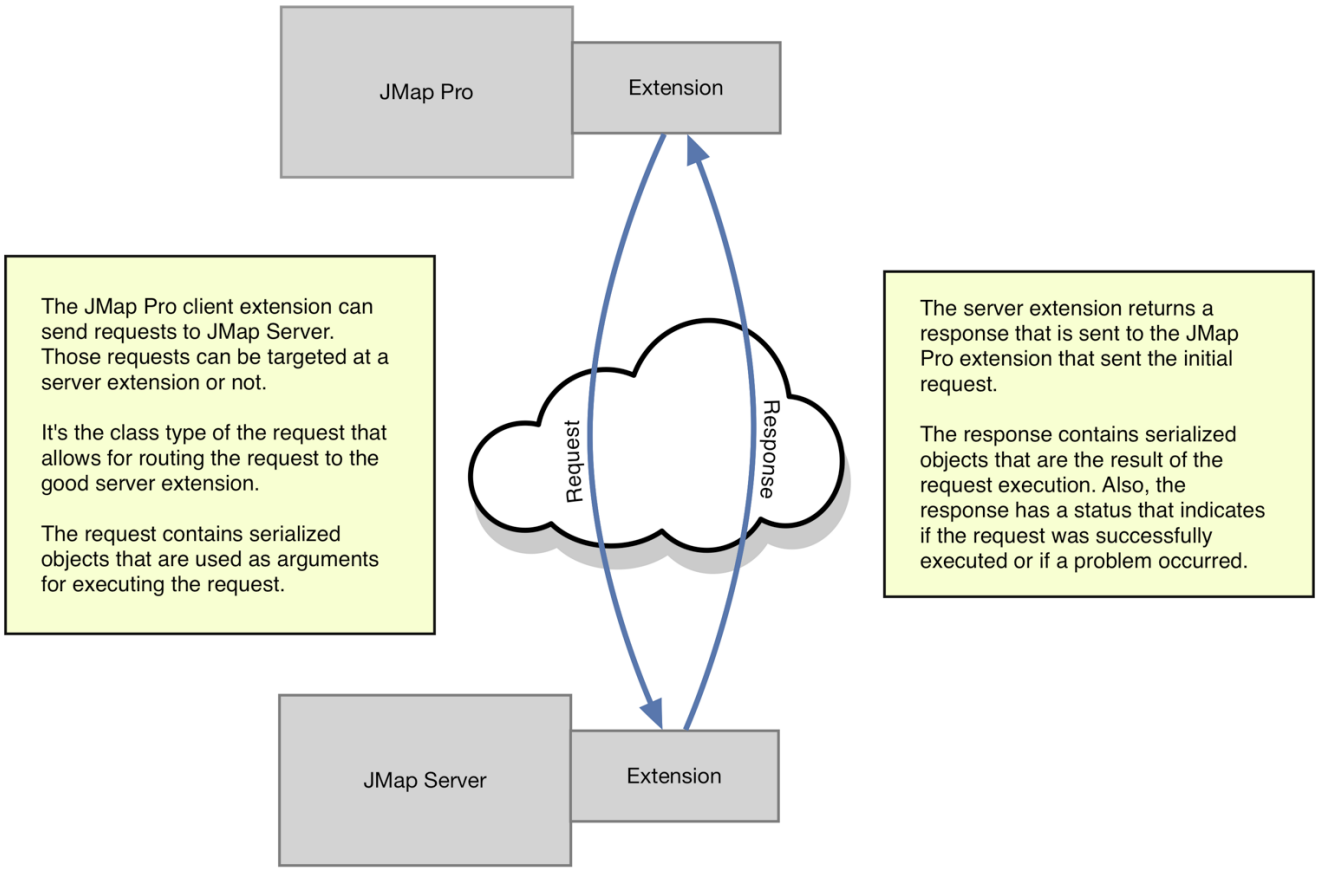
Most commonly used methods of the JMapApplication class

addProjectListener(ProjectEventListener)	Adds a project listener in the application's list of listeners.
removeProjectListener(ProjectEventListener listener)	Removes a project listener from the application's list of listeners.
createNewView()	Creates a new view (View class) initialized with the current project in the application.
getCurrentProject()	Returns the active project (Project class) in the application.
getGuiService()	Returns the JMapGuiService instance to be used in order to access and manipulate the components of the application's GUI.
getLogger()	Returns an instance of java.util.Logger to perform logging in the application.
getClientExtension(String)	Returns the extension loaded (JMapClientExtension class) for the specified class name.
getMessagingController()	Returns the messaging controller (JMapClientMessagingController class) of the application.
getEditionTransactionManager()	Returns the edition transaction manager (EditionTransactionManager class) of the application.
getUserParameterController()	Returns the user parameter controller (JMapUserParameterController class).
getUserParameter(String)	Returns the user parameter (UserParameter class) saved for the specified key.
setUserParameter(UserParameter)	Defines a user parameter to be saved.

Application context

When initializing the application, all useful instances that define the application's execution context are stored in a singleton of the JMapApplicationContext class.

When JMap Pro opens, a connection to the associated JMap Server is established. This connection is used to direct the various system requests, including requests to load the project configuration and data as well as extension requests.



Communication between JMap Pro application and JMap Server

Communication between a JMap Pro (desktop) application and JMap Server is done by exchanging requests and responses. The requests and the responses are in fact serialized objects that must normally be programmed, and these objects will contain the properties required to execute the request and return the information to the client. For example, an application managing citizen requests might have a request and response with the following properties:

Request	Response
Name of requester	Status of the saved request in the database
Type of request	Unique identifier generated upon save
Description	
x and y coordinates of the request's location	

Example of the properties of a request and response

For more information, refer to the *Programming Extension Requests* section.

Response status

Every response has a status that indicates if the request was executed successfully or if a problem occurred. The getStatus() method of the *JMapExtensionResponse* class allows you to get the status of the response. For the list of possible statuses and their description, refer to the documentation of the *JMapExtensionResponse* class.

Your extension should always check the status of a response before using it. If the status is not equal to JMapSRV_STS_SUCCESS, a special treatment should be done in order to manage the error situation. In this case, the getMessage() method allows you to generate a message explaining the cause of the error.

Sending requests to JMap Server

The *JMapSrvConnection* class is responsible for all communication between JMap client and JMap Server. The connection instance can be accessed using the JMap application context, as shown in the following example.

```
JMapSrvConnection jmapConn = JMapApplicationContext.getInstance().getConnection();
```

The following methods (inherited from the *JMapNetworkConnection* class) are used to send requests to JMap Server and to receive the responses.

Method	Description
executeRequest()	Sends the request passed as a parameter to JMap Server and returns the response generated by JMap Server or by a server extension. This is a blocking method. It is well suited to situations where one must wait for the server's response before continuing.
pushRequest()	Sends the request passed as a parameter to JMap Server. This method is non-blocking. When the response is received, the *callback()* method of the client of the request is called. This method is well suited to situations where one must quickly give back control to the user and when one can process the response in an asynchronous manner.

The following code example shows how to use the executeRequest method.

```
MyRequest request = new MyRequest("This is a test", 555);
JMapSrvConnection jmapConn = JMapApplicationContext.getInstance().getConnection();
MyResponse response = (MyResponse)jmapConn.executeRequest(request); // Execution WILL blo

if (response.getStatus() == MyResponse.JMAPSRV_STS_SUCCESS)
{
  // Do something useful with the response
}
else
{
  // Handle error here
}
```

The following code example shows how to use the pushRequest method.

```
MyRequest request = new MyRequest("This is a test", 555);
request.setClient(new JMapRequestClient()
{
  // Will be called when the response is received from JMapServer
  @Override
  public void callback(JMapRequest request, JMapResponse response)
  {
    if (response.getStatus() == MyResponse.JMAPSRV_STS_SUCCESS)
    {
      // Do something useful with the response
    }
    else
    {
      // Handle error here
    }
  }
});
JMapSrvConnection jmapConn = JMapApplicationContext.getInstance().getConnection();
jmapConn.pushRequest(request); // Execution WILL NOT block here
```

The following tables present the keys of JMap Pro's GUI components. These keys are used to control these components through programming, notably in the methods of the *JMapGuiService* class.

Toolbars	
ZOOM_PAN	Toolbar containing buttons for zooming and panning.

Toolbars	
SELECTION	Toolbar containing buttons for selection/deselection.
ROTATION	Toolbar containing buttons for rotation and cancellation of the rotation.
MEASURE_LABEL	Toolbar containing buttons for measuring, clearing measurements, labelling and clearing labels.
INFOS_SEARCH	Toolbar containing buttons for reporting and search tools.
PRINT_MAIL	Toolbar containing buttons for printing and emailing.

Menus	
PROJECT	Projects menu containing items for loading projects, context management, personal layers management, etc.
VIEW	View menu containing items for controlling the display of the various components of the GUI such as the layer manager, overview, etc.
TOOLS	Tools menu containing the items of various tools and extensions.
MAP	Maps menu containing map window management items.
HELP	Help menu containing help topic items.

Windows	
LAYERS	Map layer manager.
CONTEXTS	Contexts management window.
ELEMENTS_EXPLORER	Window containing the elements explorers of the layers.
SELECTION_EXPLORER	Window containing the selection explorer.
MESSAGES_VIEWER	JMap messages management window.

Windows	
PERSONAL_LAYERS	Personal layers management window.
QUERIES	Spatial and attribute queries window.
GO_TO_COORDINATE	Window for entering coordinates to be reached on the map.
GEOMETRY_INFO_PANEL	Window to display geometric properties of map elements.
OVERVIEW	Map overview window.

A JMap tool provides interaction between the user and the map using the mouse. For example, when the user activates the *selection tool* and clicks on the map to select an item, it is the selection tool class that performs the work. This tool is programmed to make a selection at the location that is clicked on the map. Similarly, when the distance measurement tool is used, the tool class calculates and displays the distance between the 2 points clicked by the user. Thus, developing a JMap tool allows you to implement custom actions.

In general, only one tool at a time can be enabled. In order to be enabled, a tool must be the active tool in a JMap view. To do so, you must use the method *setCureentTool(Tool)* of the *View* or *ViewManager* class. Note that the method used to activate the tool, such as a button or a menu item, has no connection with the operation of the tool itself.

When the user changes the active the tool (e.g. by pressing a button), the code will essentially be as follows:

```
JMapApplicationContext.getInstance().getViewManager().setCurrentTool(new MyTool());
```

When a tool is active, it receives all the mouse events that are generated by the active view. The Tool class is responsible for processing these events and taking the appropriate actions.

To develop a new tool, you must program a class derived from the abstract class *Tool* and implement only the methods needed to perform the tool's function. For example, if the tool must perform an action when the user clicks the mouse, you must implement the *onToolClicked()* method.

Tool class methods	
init()	This method is called when the tool becomes the active tool of a view. The code for this method should be used as necessary to prepare the work of the tool. The view is passed as a parameter to this method.
getCursor()	This method is called by the view when the tool becomes active to allow your tool to provide its own mouse cursor. The cursor will be visible on the view as long as your tool remains the active tool.
onToolPressed()	This method is called when the user presses on one of the mouse buttons within the view.
onToolReleased()	This method is called when the user releases a mouse button within the view.
onToolClicked()	This method is called after the user has completed a mouse click within the view.
onToolMoved()	This method is called repeatedly when the user moves the mouse inside the view.
onToolDragged()	This method is called repeatedly when the user moves the mouse inside the view while keeping a button pressed.
terminate()	This method is called when the tool becomes inactive, i.e. when another tool is activated on the screen. The code for this method could be used, as needed, to perform a termination action or to free up resources.

The following code example shows a simple tool that displays the properties of the first element found at the coordinates of the mouse cursor when the user presses and releases the left mouse button on the map. Only vector layers are considered and the search is performed from the highest layer to the lowest layer, in the layer display order. Other methods of the Tool class are not implemented because they are not required for the operation of this tool.

```
 public class ElementInfoTool extends Tool
  {
    public void onToolReleased(MouseEvent e)
    {
      super.onToolReleased(e);

      System.out.println("ElementInfoTool.onToolReleased");

      // Obtain array of layers to cycle them in reverse order
      final Layer aLayers[] = this.view.getLayerManager().getAllLayers();

      // Get the layer visibility status from the layer manager (also includes the
hierarchy visibility)
      final LayerVisibilitySet layersVisibility =
```

```
this.view.getLayerManager().getLayerTreeVisibility();

      // Transform mouse x,y coordinate to WC coordinate
      // Point wcCoord = view.getTransform().transformInv(new Point.Double(e.getX(),
e.getY()));
      final Point wcCoord = toWCPoint(e);  // method inherited from class Tool

      final ViewState viewState = this.view.getViewState();

      // Cycle through every layer in reverse order (from top position to bottom
      // position) looking for an element under the x,y position.
      for (int i = aLayers.length - 1; i >= 0; i--)
      {
        // Consider only vector layers
        if (!(aLayers[i] instanceof VectorLayer))
          continue;

        final VectorLayer vectorLayer = (VectorLayer) aLayers[i];

        // Layer must be visible, selectable and displayed at the current scale
        if (layersVisibility.isVisible(vectorLayer.getId()) && vectorLayer.isSelectable()
&& vectorLayer.isDrawable(viewState.getScale()))
        {
          final K2DElement elem = vectorLayer.getElementAtPoint(wcCoord, viewState, true);

          if (elem != null)
          {
            String message = "Selected element on layer " + vectorLayer.getName() + ":\n\n"
                           + "Class:" + elem.getClass().getName() + '\n'
                           + "Id: " + elem.getId() + '\n'
                           + "Geometry: " + elem.getGeometry().getClass() + '\n'
                           + "Display bounds: " + elem.getDisplayBounds(viewState,
vectorLayer.getStyle(elem, viewState.getScale())) + '\n'
                           + "Attributes:\n";

            final Object[] attribs = elem.getAttributes();
            final Attribute[] attribDefinitions = vectorLayer.getAttributeMetaData();

            for (int j = 0; j < attribs.length; j++)
            {
              message += "  " + attribDefinitions[j].getName() + " : " + attribs[j] + '\n';
            }

            JOptionPane.showMessageDialog(this.view, message);

            break;
          }
        }
      }
    }
  }
```

Drawing tools

You can implement tools to draw on the map simply by deriving them from existing JMap classes for drawing tools.

By deriving from these classes, all the basic operations are inherited. The following options are available:

- Persistent and volatile modes (determines whether items are created);
- Select the layer that will receive the items drawn;
- Style parameters of drawn elements (colors, line types, etc.);
- Display dimensions on the map while drawing.

The following table lists the classes for drawing tools available in JMap. All these classes are derived from the *ToolDraw* class.

Drawing classes derived from ToolDraw	
ToolDrawBox	Draws a box (rectangle) on the map.
ToolDrawCircle	Draws a circle on the map.
ToolDrawLine	Draws a simple line on the map.
ToolDrawMultiLine	Draws a multiline on the map.
ToolDrawPolygon	Draws a polygon on the map.

The following table lists the most commonly used methods of the *ToolDraw* class.

ToolDraw class methods	
getStyleContainer()	Returns the *StyleContainer* object that contains the styles of the various types of elements (polygons, lines, etc.) that can be drawn. This method is called to change the style of the elements that will be drawn by the tool.
setDrawLayer(VectorLayer)	Specifies the layer that will receive the items drawn by the tool, if the elements are persistent (see *setPersistant (boolean)* method below). If no layer is specified, a system layer is used by default.

setPersistant(boolean)	Determines whether the items will be stored on a layer or not (see *setDrawLayer (VectorLayer)* method above). If they are not stored, the elements disappear immediately after the drawing operation is completed.

The following code example shows a tool derived from *ToolDrawPolygon* that inherits all of its drawing capabilities.

```
public class CreatePolygonTool extends ToolDrawPolygon
{
  private final static String LAYER_NAME = "Zones";

  private VectorLayer targetLayer; // The layer that will hold the polygons

  public void init(View view)
  {
    super.init(view);

    // Make sure the layer exists. If not create it and register it in the
    // layer manager of the view.
    final LayerManager layerMgr = view.getLayerManager();

    this.targetLayer = (VectorLayer)layerMgr.getLayer(LAYER_NAME);

    if (this.targetLayer == null)
    {
      // The layer does not exist, create it
      this.targetLayer = new VectorLayer(Layer.getNextUserLayerId(),     // avoid id conf
                                         LAYER_NAME,
                                         ElementTypes.TYPE_POLYGON);

      // Tell JMap that information on this layer does not come from the
      // server
      this.targetLayer.setLocal(true);

      // Set the mouse-over text for the layer. It will be displayed
      // when the user stops the mouse pointer over an element.
      this.targetLayer.setMouseOverConfiguration(new MouseOverConfiguration("This is a zo

      // Add the layer to the layer manager list. All layers need to be
      // registered in the layer manager in order to be displayed in the
      // view. The layer is added to the top.
      layerMgr.addLayer(this.targetLayer);
    }

    // Tell the superclass that the resulting elements should be persisted
    // when completed. That means that they will be placed on a layer.
    // Otherwise, they would be deleted as soon as they are completed.
    setPersistent(true);

    // Tell the superclass to place the resulting elements on the target layer.
    // Otherwise, they would be placed on the default drawing layer
    // (id = LayerManager.LAYER_ID_USER_DRAWINGS).
    setDrawLayer(this.targetLayer);

    // Modify the style of the surface elements (polygons).
    final Style surfaceStyle = getStyleContainer().getSurfaceStyle();

    surfaceStyle.setFillColor(Color.GREEN);
    surfaceStyle.setTransparency(.50f);

    // To have the valid drawing style displayed in the layer bar, make the layer
    // default style identical to the drawing style.
    this.targetLayer.setStyle(surfaceStyle, 0.);
  }
```

```
}
```

JMap Pro extensions are modules developed in Java that can be added to JMap Pro to enhance its features. Extensions are specified as parameters for applications and are initialized when said applications are launched. Typically, extensions are integrated to the GUI of a JMap Pro application by inserting the buttons and menus that activate the functions provided by the extension.

To develop an extension and make it available to users, you must perform the following two steps:

1. Develop your extension by creating a Java class derived from the *JMapClientExtension* class;
2. Deploy your extension in JMap Server so it may be accessed by the JMap Pro applications.

See the following sections for more information.

To program JMap Pro extensions, it is important to follow a set of simple rules. This section describes these programming rules.

JMapClientExtension class

The first step towards developing a JMap Pro extension consists of programming a class derived from the abstract class *JMapClientExtension*. This class contains the following 3 methods, which are called at different moments in the life cycle of the extension:

JMapClientExtension class methods	
init(JMapApplicationContext, Map<String, String>)	This method is called when the extension is loaded by the JMap Pro application. At this point, the graphical interfaces have not yet been initialized. In this method, you can put any code that will prepare the operation of your extension. This could include loading a settings file, verifying dependencies, etc. The method receives an instance of *JMapApplicationContext* as a parameter. This class provides access to all the resources of the JMap application. The method also receives a collection (map) of parameters. These parameters provide useful information for the extension. See below for more information.
initGUI()	This method is called by the JMap application when its graphical interface has been initialized. In this method, you should put the code that initializes the GUI of your extension. This could include adding buttons or toolbars, menus, windows, etc. See the GUI

	Integration section for more information. This GUI will allow users to access the functions offered by your extension.
destroy( )	This method is called by the JMap application just before it exits. In this method, you must put the code required to terminate your extension. This could include the code to release resources, save settings, etc.

Parameters are passed to the extension with the init() method, using a Map<String, String> collection. In JMap, there are two predefined parameters that match constants defined in the JMapClientExtension class:

- EXTENSION_PARAMETER_GUI_VISIBLE: Indicates whether the extension's GUI components (other than toolbars) must be visible when the application is launched.

- EXTENSION_PARAMETER_TOOLBAR_VISIBLE: Indicates whether the extension's toolbars must be visible when the application is launched.

These parameters contain true or false values and they are set by the JMap Administrator when a JMap Pro application is deployed. Your extension must comply with these parameters. An extension can also receive its own settings, which would be set by the JMap Administrator.

The following code example shows how to access the settings from the init() method.

```
public void init(JMapApplicationContext appContext, Map<String, String> mapExtensionParam
{
  String param = mapExtensionParameters.get(JMapClientExtension.EXTENSION_PARAMETER_TOOLB
  boolean toolbarVisible = Boolean.parseBoolean(param);

  ...
}
```

JMapApplicationContext class

The JMapApplicationContext class is very useful for extension development because it provides access to a set of general application resources. This class is a singleton. Therefore, the instance can easily be accessed from anywhere using the static JMapApplicationContext.getInstance() method.

Most commonly used methods of the JMapApplicationContext class	
getConnection()	Returns the instance of the connection (*JMapSrvConnection* class) to JMap Server.
getFrame()	Returns the instance of the main application window.
getViewManager()	Returns the instance of the View Manager (*ViewManager* class) of the application.

getApplication()	Returns the instance of the JMap application (*JMapApplication* class).
getJMapHome()	Returns the path to the main JMap folder on the user's computer. This is where JMap writes its data.

JMapApplication class

The *JMapApplication* class represents the JMap application. It offers methods that provide access to application resources (GUI, event logs, etc.). It also provides methods to perform tasks in a simple way (create a new map, close the project, etc.).

Most commonly used methods of the JMapApplication class

createNewView()	Creates a new map view that automatically appears in the application.
getCurrentProject()	Returns the instance of the project (*Project* class) opened in the application.
getGuiService()	Returns the instance of the application's *JMapGuiService* class. See the GUI Integration section for more information.
getLogger()	Returns the instance of the application's *Logger* class. Used to record messages from your extension in the JMap application's logs.

As a developer of JMap Pro extensions, you will probably be developing applications that need to communicate between the client side and JMap Server. Whether you need to communicate with your own extension on the server side or to make general requests to JMap Server, the programming principle is the same.

For more information on client-server communication in JMap, refer to the Client-Server Communication section.

Programming the request

To create your request class, you must observe the two following rules:

- The class should be derived from the *JMapExtensionRequest* class;
- All of the class's properties must be serializable.

You are free to add all the properties and methods that are useful to executing your request. These properties and methods can be used by your extension on the server side.

The following source code example demonstrates how to program a simple request.

```
// This request is used to save a new citizen complaint
public class SaveComplaintExtensionRequest extends JMapExtensionRequest
{
  private String citizenName;
  private int requestType;

  public void setCitizenName(String citizenName)
  {
    this.citizenName = citizenName;
  }

  public String getCitizenName()
  {
    return this.citizenName;
  }

  public void setRequestType(int requestType)
  {
    this.requestType = requestType;
  }

  public int getRequestType()
  {
    return this.requestType;
  }
}
```

Programming the response

To create your response class, you must observe the two following rules:

- The class should be derived from the *JMapExtensionResponse* class;
- All of the class' properties should be serializable.

You are free to add all properties and methods that are useful to returning the information regarding the request's execution to the client. These properties and methods can operate from the client side by your extension after the request is executed on the server.

The following source code example demonstrates how to program a simple response.

```
// This response is returned to client after a citizen complaint save request was executed
public class SaveComplaintExtensionResponse extends JMapExtensionResponse
{
  private long uniqueId;

  public void setUniqueId(long uniqueId)
  {
    this.uniqueId = uniqueId;
  }

  public long getUniqueId()
  {
    return this.uniqueId;
  }
}
```

Being derived from the *JMapExtensionResponse* class, your response class will inherit some useful properties, including the response status and an explanatory message in case of a problem.

Response status

Each response has a status that indicates if the request was executed successfully or if a problem occurred. The getStatus() method of the *JMapExtensionResponse* class allows you to get the status of the response. For the list of possible statuses and their description, refer to the documentation of the *JMapExtensionResponse* class.

Your extension should always check the status of a response before using it. If the status is not equal to JMapSRV_STS_SUCCESS, a special process must occur in order to manage the error. In this case, the getMessage() method allows you to generate a message explaining the cause of the error.

Some client extensions can be completely invisible to the user but the majority of extensions are integrated into the JMap Pro GUI. This integration can take many forms. The *initGui()* method must be used to initialize the graphical interface of an extension.

JMapGuiService class

The *JMapGuiService* class is used to access the components of the graphical user interface (menus, toolbars, etc.) and to add or remove components. To simplify access to the graphical components, each component has a unique key. This key must be provided when a component is added or removed. For a complete list of available keys, see the section List of GUI components. The JMapGuiFactory class described below provides methods for creating GUI components.

The main methods of the *JMapGuiService* class are listed below:

Most commonly used methods of the JMapGuiService class

addMenuItem()	Adds an item to an existing menu by specifying the menu key.
getMenu()	Retrieves a menu using its key. The returned object can then be used to modify the menu.
addToolBar()	Adds a toolbar to a specific position in the application interface.
getToolBar()	Retrieves a toolbar using its key. The returned object can then be used to modify the toolbar.
addDockableComponent()	Adds a dockable window at a specific position in the application interface.
removeDockableComponent()	Allows you to remove a dockable window using its key.
getGuiFactory()	Returns the JMapGuiFactory instance. This class allows you to create GUI components such as buttons and toolbars.

JMapGuiFactory class

The *JMapGuiFactory* class is used to create buttons, menus and toolbars that are compatible with JMap applications. Methods such as createButton () createMenu () createToolBar (), etc. are called to create the appropriate components.

Integration examples

Adding a button to an existing toolbar

The following code shows how to add a button to an existing toolbar. The toolbar is retrieved using its key. For a complete list of available keys, see section List of GUI components. Note that the action associated with the button is not shown in the example.

```
// This button will be in a group thus only onebutton can be pressed among this group.
AbstractButton button = guiFactory.createToggleButton(new ButtonAction(),
    appContext.getApplication().getGuiService().getMainButtonGroup());

// Add the button on the Zoom and Pan toolbar.
appContext.getApplication().getGuiService().getToolBar("ZOOM_PAN").add(button);
```

Adding a toolbar

The following code example shows how to create a toolbar using the JMapGuiFactory class. In this example, the JMapApplicationActionManager class is used to access the buttons' actions. This class can manage single instances of actions. Afterwards, two buttons are created and added to the bar. Finally, the toolbar is added vertically, on the right side of the application. Notice the HashMap of properties that is used to place the toolbar. This principle is used with all components to specify their characteristics.

```
// Create the new toolbar
final ToolBar toolBar = guiFactory.createToolBar("SHOWCASE", "JMap example showcase");

// Create and add buttons on toolbar
toolBar.add(
    guiFactory.createToggleButton(applicationActionManager.getAction(CreatePolygonAction.
  );

toolBar.add(
    guiFactory.createButton(applicationActionManager.getAction(ShowDockingPanelAction.cla
  );

// Adds the new toolbar vectically to the right side of the application frame.
HashMap<String, Object> properties = new HashMap<String, Object>();
properties.put("CONTEXT_INIT_SIDE", new Integer(SwingConstants.EAST)); // Dock to the eas
jmapGuiService.addToolBar(toolBar, properties);
```

Adding a window

The *addDockableComponent()* method adds components such as windows to the interface. It takes a HashMap of key-value pairs as a parameter to define the display settings of the new window. All keys and possible values are defined in the method's documentation.

```
// Create a panel with the needed components
final JPanel panel = new JPanel();
...

// Set the properties of the dockable panel.
HashMap<String, Object> properties = new HashMap<String, Object>();
properties.put("KEY", COMPONENT_KEY);                    // Will be used to reference th
properties.put("TITLE", "JMap 6.5 SDK showcase.");       // Will be displayed on the tit
properties.put("CONTEXT_INIT_SIDE", SwingConstants.EAST); // Tells JMap to dock the panel

appContext.getApplication().getGuiService().addDockableComponent(panel, properties);
```

Adding a menu

The following code example shows how to create and add a menu to the interface.

```
// Create menu and add items to it
JMenu menu = new JMenu("SDK");

JMenuItem item1 = new JMenuItem("Item 1");
JMenuItem item2 = new JMenuItem("Item 2");

this.menu.add(item1);
this.menu.add(item2);

// Add menu to menu bar at position 4
appContext.getApplication().getGuiService().getMenuBar().add(menu, 4);
```

Adding a menu item to an existing menu

The following code example shows how to add items to existing application menus. Note that a key is used to retrieve existing menus. For a complete list of available keys, see the section *List of GUI components* .

```
// Add 2 menu items to existing menus TOOLS and HELP
JMenuItem item3 = new JMenuItem("Item 3");
JMenuItem item4 = new JMenuItem("Item 4");

appContext.getApplication().getGuiService().addMenuItem("TOOLS", item3, false);
appContext.getApplication().getGuiService().addMenuItem("HELP", item4, false);
```

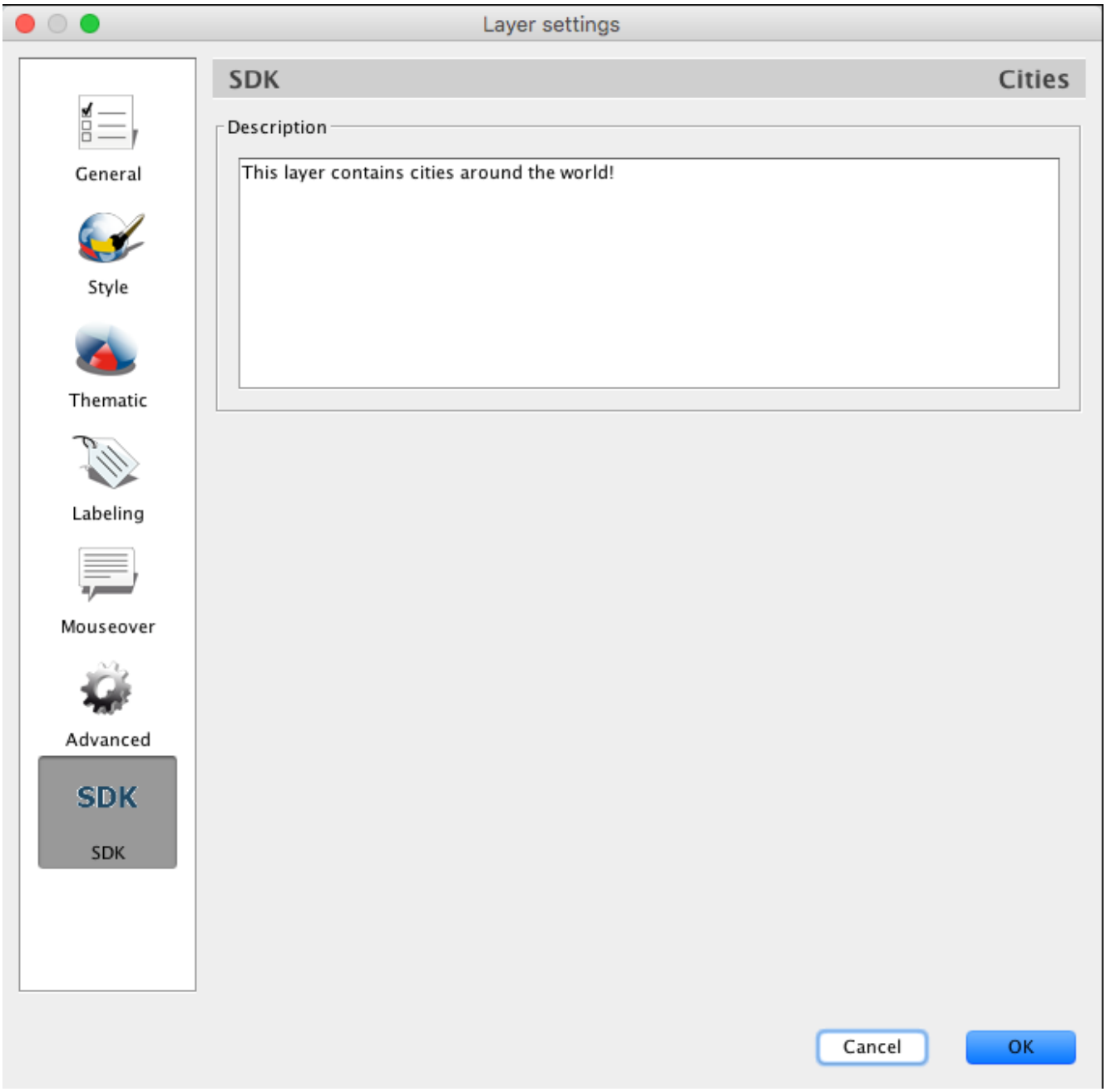
Adding an item to the map pop-up menu

The following code shows how to add a section of items to the map's pop-up menu. This menu appears when the user right-clicks on the map. In addition, the menu item is only active if the user has clicked on at least one element of a map layer. Otherwise, the menu item will be disabled.

```
final ViewMenuAction menuAction = new ViewMenuAction();

// Obtain the currently active view
View view = JMapApplicationContext.getInstance().getViewManager().getActiveView();
if (view != null)
{
  // Add a separatr and a menu item
  view.addPopupMenuSeparator();
  view.addPopupMenuAction(menuAction);
}

// Register an event listener with the View Manager so that we are notified when the view
JMapApplicationContext.getInstance().getViewManager().addViewEventListener(new ViewAdapte
{
  @Override
  public void viewPopupMenuShowing(ViewPopupMenuShowingEvent e)
  {
    // Enable map menu item only if the user clicked on some element
    K2DElement[] elements = e.getView().getLayerManager().getElementsAtPoint(e.getWcCoord
    menuAction.setEnabled(elements.length > 0);
  }
});
```

Adding a section to a layer's settings window

It is possible to add sections to the settings window of a layer, as shown in the image below. To do this, you must implement your own class derived from the abstract class *AbstractLayerSettingsCustomPage* and implement that class's 3 abstract methods. You must provide a title and icon for your section by passing them to the constructor of the superclass; they will be displayed in the settings window.



Additional layer setting page created programatically

Abstract methods of the AbstractLayerSettingsCustomPage class

getContentPanel()	In this method, you must create and return the the instance of JPanel that displays the GUI of the settings in your section. This is the graphical interface that will be displayed.
validateSettings()	This method is called when the user clicks on the OK button to confirm the changes and close the window. Your method should return true only if the parameters entered

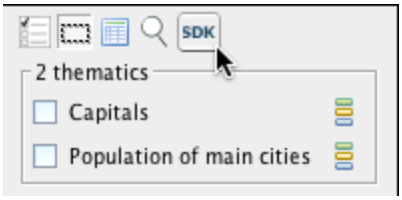
	are valid. Otherwise, the window will refuse to close.
applySettings()	This method is called when the user clicks on the OK button to confirm the changes and close the window, and after calling the validateSetting() method. You must apply the parameter changes on the layer.

The getLayer() method of the *AbstractLayerSettingsCustomPage* class lets you know to which layer the parameters apply. To save your settings section, you must use the method addLayerSettingsCustomPage of the ShowLayerSettingsAction class, as shown in the following source code example.

```
// Obtain a reference to the ShowLayerSettingsAction instance
ShowLayerSettingsAction showLayerSettingsAction = (ShowLayerSettingsAction)appContext.get
                                                                                .get

// Add the custom panel for the Cities layer
showLayerSettingsAction.addLayerSettingsCustomPage(SDKLayerSettingsPanel.class.getName(),
```

Adding a custom button to the layer manager

You can add buttons associated with a layer in the layer manager. These buttons can be used to trigger custom actions that are specific to a layer. The buttons are added in different ways in the hierarchical section and list section.



Example of custom button added to a layer

The following source code example shows how to add a button to the Cities layer.

```
// Obtain layer instance from layer manager
Layer layerCities = appContext.getViewManager().getLayerManager().getLayer("Cities");

JMapGuiService guiService = appContext.getApplication().getGuiService();
JMapGuiFactory guiFactory = guiService.getGuiFactory();

AbstractButton abstractButton = guiFactory.createButton(new ButtonAction());

// Add the button to the 'More options' of the LayerTreeBar (hierarchy) for the layer Cit
guiService.getLayerTreeBar().addCustomButton(abstractButton, layerCities.getId());

// Add the button to the LayerPanel of the LayerBar (list) for the layer Cities
LayerPanel layerPanel = guiService.getLayerBar().getLayerPanel(layerCities.getId());
```

```
Button button = new Button(new ButtonAction(), false, new Dimension(18, 18));
layerPanel.addCustomButton(button);
```

To be deployed within JMap Pro applications, client extensions must follow certain rules. If these rules are met, the extension appears in the deployment section of JMap Admin.

1 - Group extension classes in an archive (JAR)

All classes and resources (images, etc.) of the extension must be contained in a single JAR archive file. Use a meaningful and unique name, as this name will be used in the following steps.

2 - Include a manifest file

The extension archive must include a manifest.mf file with the following entries:

Variable	Description
extension_class	Identifies the main class of the extension, derived from the abstract class JMapClientExtension.
extension_name	Specifies the name of the extension. This name appears in JMapAdmin when deploying applications.
extension_version	Specifies the version number of the extension. This information appears in JMapAdmin when deploying applications. The version number is used only to simplify the management of extensions.

Here is an example of a manifest file's content:

```
extension_class: jmap.extensions.edition.EditionExtension
extension_name: Edition
extension_version: 1.0.0049
```

3 - Provide a JNLP file

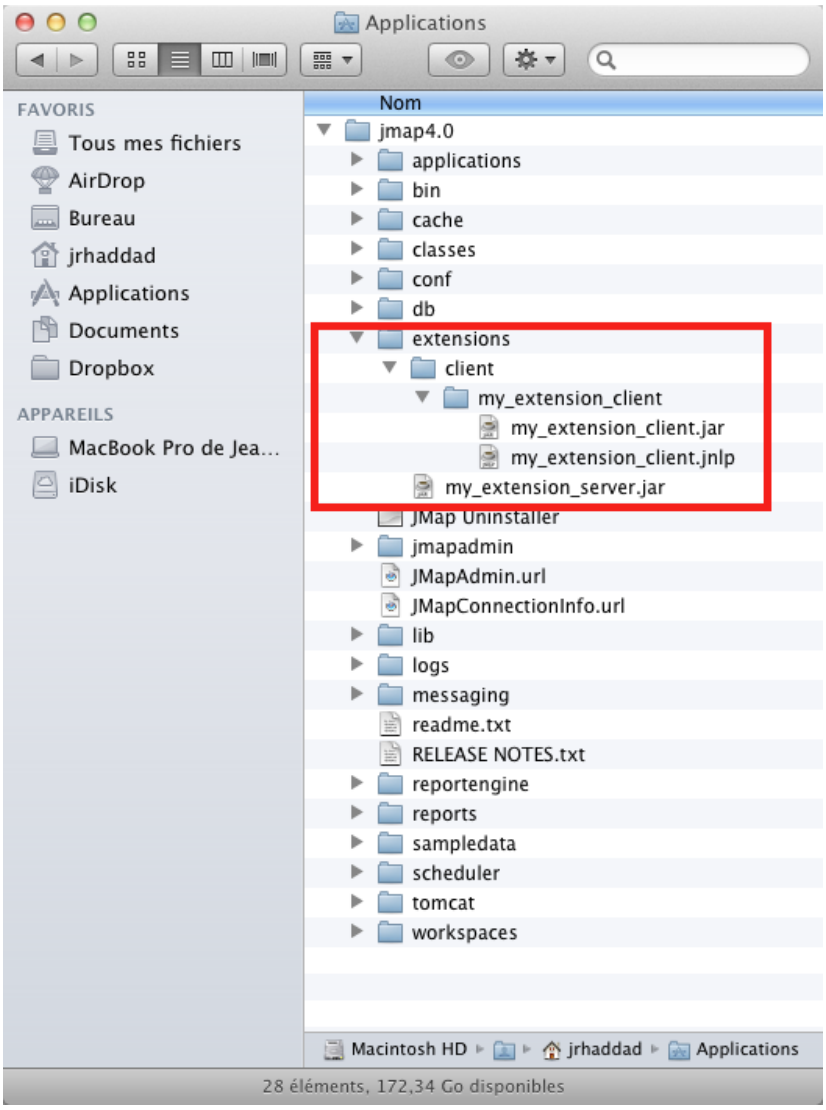
The JNLP file is required. It describes the deployed library. The file must have the same name as the extension's JAR file (except for the .jnlp extension). The following example shows the parts that must be changed in bold.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for Extension libraries -->
<jnlp
  spec="1.0+"
  codebase="http://$JMAPSERVER_HOST$:$JMAPSERVER_WEBPORT$$PATH$/edition_client"
  href="edition_client.jnlp">
```

```
  <information>
    <title>Edition Extension</title>
    <vendor>K2 Geospatial</vendor>
    <description>Edition Extension</description>
    <description kind="short">Edition Extension</description>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <jar href="edition_client.jar"/>
  </resources>
  <component-desc/>
</jnlp>
```

4 - Place the files in the correct directory

All files that make up the extension (JAR, JNLP, and other files) must be placed in a directory created specifically for the extension and located within the directory for Client Extensions (JMap_HOME / extensions / client). **The name of the extension's directory must be identical to the name of the extension's JAR**. The following image shows how files and directories are organized in Windows for K2 Geospatial's *Edition* extension.



The signature of a Java library certifies that the library comes from an identifiable source and that its content has not been altered. The user who executes the library's code can trust the authenticity of the library. This is even more important when the library requires system access that can cause security issues such as data access, network access, etc.

Signing Java libraries is done using the jarsigner tool included with the Java JDK. To do this, you must have a Java signing certificate issued for your organization. These certificates can be purchased from a reputable security company such as Thawte (http://www.thawte.com/) or Verisign (http://www.verisign.com/). Once you have your certificate, you must import it in a keystore using the keytool, which is also included with the Java JDK. Note that the keytool can also produce development certificates. These certificates are not deemed to be from reliable sources and they will generate warning messages. However, they are helpful for development and in-house testing purposes. For

more information on this topic, refer to the documentation on the Java JDK's security tools (http://docs.oracle.com/javase/8/docs/technotes/tools/index.html).

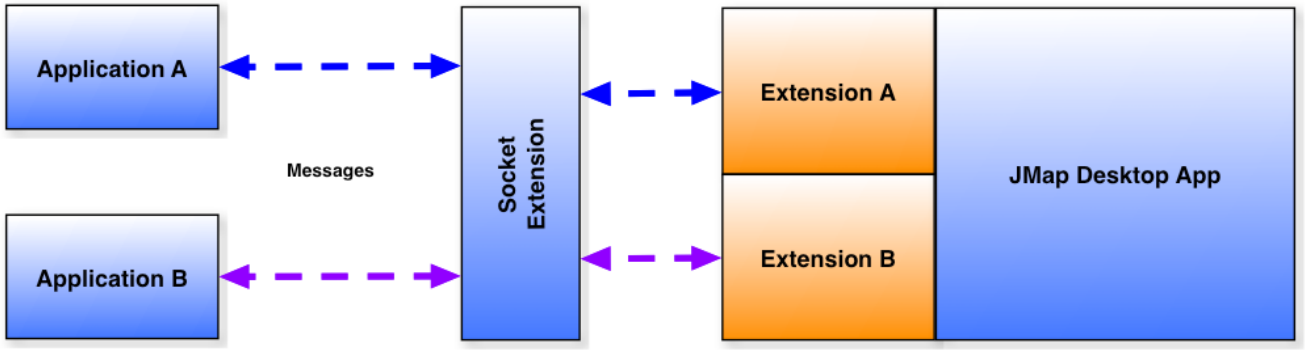
The libraries of JMap Pro extensions must be signed in order to be properly deployed in a JMap Pro application. If you use the *extension builder* provided with the JMap SDK, the Ant scripts generated handle the signing of your extension. Thus, when you run these scripts to compile and build the library for your extension, it is signed automatically. This signature is done using the development certificate provided with the SDK. If you want to use your own certificate, you must replace the JDK_HOME/tools/extensionbuilder/RES/*keystore.jks* file by your own keystore, in which you will have imported your certificate.

There are several ways to allow communication between a JMap Pro application and another client application, also called an external application.The external application is an application running locally on the same computer as JMap Pro. We are not talking here of web applications (HTML, javascript), but rather *client* applications. The JMap Socket extension provides this communication in a simple and effective way.

JMap Socket extension

The Socket extension is a generic extension that allows a bidirectional socket communication between external applications and JMap Pro extensions. This extension does not directly process the messages. Its role is limited to providing communication. Here is a typical scenario describing the use of this extension:

An external application sends the identifier of an element to JMap and it is automatically located and selected on the map. Conversely, the user clicks on a map element, and JMap sends the ID of the element to the external application, which displays a form with the properties of the element.



General operation

The Socket extension provides a standard means of communication allowing JMap extensions to communicate with external applications. External applications are applications running locally on the same computer as JMap Pro. These applications can be written in any programming language (Java, C + +, Windev, C #, VB.NET, etc.). Note that web applications (HTML, javascript, etc.) are not supported by this integration method. Other methods are better suited to web applications (see Java Communication - Javascript).

To communicate, external applications and JMap extensions send messages that are in fact byte arrays. The contents of these byte arrays is interpreted on either side. The Socket extension does not interpret their content.

The external application always connects to JMap first. It does so by opening a socket to the port configured in the Socket extension. Thereafter, the external application will once again initiate communication. This first contact is a special communication called a handshake. During the handshake, the external application indicates its unique identifier (a character string identifying the application).

As for the JMap extension that wants to communicate with the external application, it must register with the Socket extension. It must provide the same identifier as the external application. This is the identifier that links the two.

Subsequently, the external application and JMap extension can send each other messages in any direction. Each party (the external application or extension) is responsible for interpreting the messages and taking the appropriate actions.

Optionally, the Socket extension can launch the external application if it does not respond. Once it is launched, the external application may initiate communication as described above.

Programming on the external application side

The code examples that follow are written in Java . An external application programmed in another language should implement the same logic, in that application's language.

1. Opening a socket using IP address 127.0.0.1 and port 12351):

```
Socket socket = new Socket(“127.0.0.1”, 12351);
```

2. Sending a handshake message

The handshake message is received and interpreted by the Socket extension only. Note that the number of bytes of the message must precede the contents of the message. The message is sent as a byte array. The character encoding (charset) of this message is defined in the configuration of the Socket (socket.properties file) extension.

```
DataOutputStream dos = new DataOutputStream(socket.getOutputStream());

String handshake = "handshake:MyIdentifier";
dos.writeInt(handshake.getBytes().length);
dos.write(handshake.getBytes());
dos.flush();
```

3. Sending other messages

Subsequent messages are sent the same way as the handshake message, however, they will be received and interpreted by the extensions registered with the same identifier. Extensions that receive messages must therefore be able to understand the content of messages and perform the corresponding actions. The character encoding of the message is free and must be agreed to by the extension and the external application.

```
DataOutputStream dos = new DataOutputStream(socket.getOutputStream());

String message = "locate :id=333";
dos.writeInt(message.getBytes().length);
dos.write(message.getBytes());
dos.flush();
```

4. Receiving messages from JMap

Receiving messages is done using the same socket as for sending. The length of the message is read first, followed by the byte array of the message. Message reception should be done asynchronously, in a separate thread.

```
DataInputStream dis = new DataInputStream(socket.getInputStream());

int length = dis.readInt();
if (length > 0)
{
  byte[] messageBytes = new byte[length];
  dis.readFully(messageBytes);
}
```

Programming on the JMap extension side

The following example shows how to register a JMap extension with the socket extension to be notified when a connection or disconnection occurs and when messages from the external application are received. This code could be placed in the init() method of the JMap extension.

```
SocketClientExtension.register(new SocketClient()
```

```
{
  @Override
  public void socketMessageReceived(byte[] bytes)
  {
    String data = new String(bytes /*, Charset.forName("ISO-8859-5")*/);

    // Interpret content of data…
  }

  @Override
  public void socketConnectionOpened()
  {
  }

  @Override
  public void socketConnectionClosed()
  {
  }

  @Override
  public String getIdentifier()
  {
    return "MyIdentifier";
  }

  @Override
  public String getExecutable()
  {
    // Optional - return null if not needed
    return "c:/external_app.exe";
  }
});
```

The following example shows how to send a message to the external application from the JMap extension.

```
String message = "openform: id =99";
byte[] bytes = message.getBytes();
SocketClientManager.getInstance().sendMessage("MyIdentifier" , bytes);
```

Java applets are executed in the environment of a web browser. Therefore, they can interact, in both directions, with the javascript code in the web page that contains the applet. This communication allows you to create HTML interfaces to control the map and perform simple integrations with other applications running in the environment of the web browser. This does not apply to JMap Pro applications deployed with the JavaWebStart method (not in a web browser) because no web page is involved in these cases.

Javascript to Java

The following sample web page is a normal startup page for a JMap Pro applet to which javascript functions have been added (*pan* and *zoom*). Hyperlinks are also used to call these functions. When triggered, the functions call the JMap API to control the map.

The JMap Pro application has a *getApplicationContext()* method that is used to access all the components of the application.

```
<%@page contentType="text/html;charset=ISO-8859-1"%>

<%
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String parameters = null;

  if (username != null && username.length() != 0)
  {
    parameters = "?username=" + username;
    if (password != null)
      parameters += "&password=" + password;
  }
%>

<html>
  <head>
  <title>
    aa
  </title>
  <script src="library/deployJava.js"></script>
  <script src="library/jmap.js"></script>
</head>

<BODY BGCOLOR="#ffffff" topmargin="0" leftmargin="0" rightmargin="0" bottommargin="0">

<script>
  function zoom(factor)
  {
    document.jmap.getApplicationContext().getViewManager().getActiveView().zoom(factor);
    document.jmap.getApplicationContext().getViewManager().getActiveView().refresh();
  }
  function pan(x, y)
  {
    document.jmap.getApplicationContext().getViewManager().getActiveView().pan(x, y);
    document.jmap.getApplicationContext().getViewManager().getActiveView().refresh();
  }
</script>

<a href="javascript:zoom(2.);">Zoom in</a>
<a href="javascript:zoom(0.5);">Zoom out</a>
<a href="javascript:pan(0, 200);">Pan north</a>
<a href="javascript:pan(0, -200);">Pan south</a>
<a href="javascript:pan(200, 0);">Pan west</a>
<a href="javascript:pan(-200, 0);">Pan east</a>

<script>
  var attributes =
  {
    name: "jmap",
    codebase: "http://127.0.0.1:8080/aa",
    code: "com.kheops.jmap.client.application.JMapApplicationLauncher",
    archive: "dockingClient.jar,jmap_application.jar,jmap_client.jar,jmap_client_images.j
    width: "100%",
    height: "100%",
    mayscript: true,
    separate_jvm: true,
    parameters: "-appclassname jmap.viewers.docking.AppDocking -project &quot;The World&q
  };
```

```
    var parameters = {fontSize:16, jnlp_href:'dockingClient.jnlp'} ;

    deployJava.runApplet(attributes, parameters, '1.6.0_10');
  </script>

  </body>
  </html>
```

Java to Javascript

From a JMap Pro applet, it is possible to call javascript functions that are in the web page containing the applet. This allows for interaction between the JMap Pro application and its HTML environment. For example, it would be possible to select an item on the map and display its information in an HTML page.

To enable this java-javascript communication, the ***mayscript: true*** parameter must be specified in the attributes used to start the applet, as shown in the following example.

```
<script>
  var attributes =
  {
    name: "jmap",
    codebase: "http://127.0.0.1:8080/aa",
    code: "com.kheops.jmap.client.application.JMapApplicationLauncher",
    archive: "dockingClient.jar,jmap_application.jar,jmap_client.jar,jmap_client_images.jar
    width: "100%",
    height: "100%",
    mayscript: true,
    separate_jvm: true,
    parameters: "-appclassname jmap.viewers.docking.AppDocking -project &quot;The World&quo
  };

  var parameters = {fontSize:16, jnlp_href:'dockingClient.jnlp'} ;

  deployJava.runApplet(attributes, parameters, '1.6.0_10');
</script>
```

To call a javascript function from your Java code (possibly your JMap extension), you must use the JSObject API, as shown in the following example.

```
 JSObject w = JSObject.getWindow((Applet)appContext.getRootPaneContainer());
 w.eval("show_form(" + id + ");");
```

In this example, the getWindow method receives the instance of the applet as a parameter. The getRootPaneContainer method of the JMapApplicationContext class returns the top level container of the application, the latter being the instance of the Applet when the application is running inside the web browser.

The javascript function to be called and its parameters are taken as parameters by the eval method. In this example, an ID is passed to the javascript function.

The JMap Pro application takes certain parameters when it is started up. These parameters specify the address of the JMap Server, the communication ports, the project to open, and many other options.

Parameters are passed to the application in various ways, depending on its starting mode. Java and JavaWebStart applet parameters are passed in the application's JNLP file. Application parameters are passed to the command line or can be specified in an Ant script.

The following example shows the parameters passed to the command line to start a JMap Pro application that opens the project *The World* and loads the extension *Showcase* .

```
-appclassname jmap.viewers.docking.AppDocking -server jmap3.k2geospatial.com -directport
-project "The world" -extensions jmap.examples.showcase.extension.ShowCaseClientExtensio
```

The following table describes the various parameters:

Parameters (* = mandatory)	
-appclassname *	Main class of the application to run. Currently, the only possible value is jmap.viewers.docking.AppDocking.
-server *	The name or IP address of the JMap server to which the application must connect.
-directport	IP communication port for direct connections with JMap Server.
-httpport	IP communication port for connections through HTTP proxy with JMap Server.
-project	The project that will open by default. Must be enclosed in quotes if the name contains spaces.
-language	Language of the application's GUIs. Supported values are fr, en, es, pt, and default. The default value means that the language used will be the default language of the user's operating system.

Parameters (* = mandatory)	
-country	The country, used with the language, to determine the display formats of dates and numbers.
-username	The username to log on to the application.
-password	The password to log on to the application.
-sessionid	Specifies the session number in order to connect to a session that is already open on the JMap server.
-autozoom	Instructs the JMap application to locate a position or element automatically upon startup. The syntax is: autozoom-REGION, x, y, width, height OR autozoom-OBJECT; LayerName; attribute, value OR autozoom-OBJECT; LayerName; attribute, value; maxScale
-connection	Type of connection to use between the application and the JMap Server. Possible values are: - direct: Opens a direct connection to JMap Server using the direct port. - proxy: Opens a proxy HTTP connection to JMap Server using the HTTP port. - any: Attempts to open a direct connection. In case of failure, switches to connection through HTTP proxy.
-proxypath	If the connection type is HTTP-proxy, specifies a relative path to the HTTP proxy.
-serverid	If the connection type is HTTP-proxy, specifies on which JMap Server instance the connection must open when multiple instances of JMapServer are available. This way, the HTTP proxy can be used to direct the queries. Server IDs are configured in the jmsconnections.xml files.
-showconnectionmoredetails	Determines whether the login window should show the list of available projects on JMap Server. Possible values: true, false

Parameters (* = mandatory)	
-usediskcache	Determines whether the disk cache is enabled or not. Possible values?true, false
-diskcachepath	If the disk cache is enabled, determines the folder where the cached data will be saved.
-diskcachesize	If the disk cache is enabled, determines the maximum size of the total data cache. Data will automatically be deleted when the cache reaches the size limit. The value is expressed in bytes. A value of -1 indicates an unlimited size.
-usememorycache	Determines whether the memory cache is enabled or not. If the cache is enabled, the data in memory is handled in the following way: when the space becomes full, i.e. the cache reaches the size limit (-maxmemory parameter), data is automatically removed from the memory. The amount of data removed depends on the specified percentage (percentreleasememory parameter). Possible values?true, false
-maxmemory	If the in-memory cache is enabled, determines the maximum size of the data in memory. The value is expressed in bytes. The default value is 33554432 (32MB).
-percentreleasememory	Determines the percentage of memory to free when the cache becomes full. The percentage is based on the total size of the cache. The value is an integer between 1 and 100.
-logos	List of logos to show on the map as well as their position and transparency. Example:-logos "? Jmaplogo.gif x = 5 & y = 5 & transparency = 30.0 & relativeTo = NE"
-northarrow	Display settings for a north arrow on the map, including the model, position, size, etc. Example:-northarrow Simple3D, 0,50,5,5
-displayscalebar	Determines whether the scale bar must be displayed on the map.

Parameters (* = mandatory)	
	Possible values?true, false
-extensions	The list of extensions to initialize at application startup, separated by commas. Example: *-extensions jmap.extensions.googlemap.client.GoogleMapsExtension,jmap.examples.showcase.extension.ShowCaseClientExtension*

Remote debugging environment

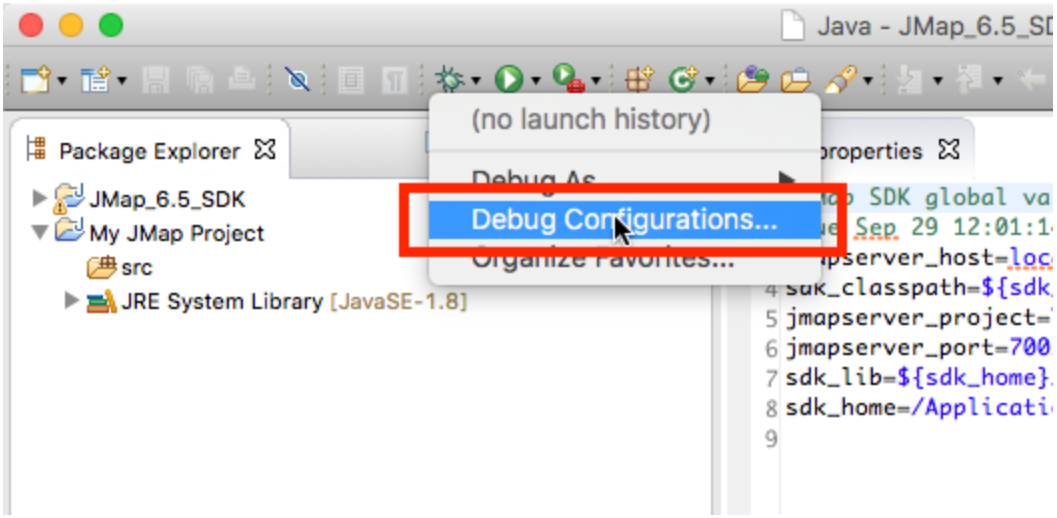
It may be useful to have remote debugging for the JMap Server extensions. Once deployed, these extensions are executed directly in the JMap Server process. It is therefore only possible to debug such extensions remotely. The Java Runtime Environment allows for remote debugging by adding a special parameter in the JMap Server environment. You will then be able to perform a server connection from Eclipse and monitor the execution of your JMap extensions step by step.

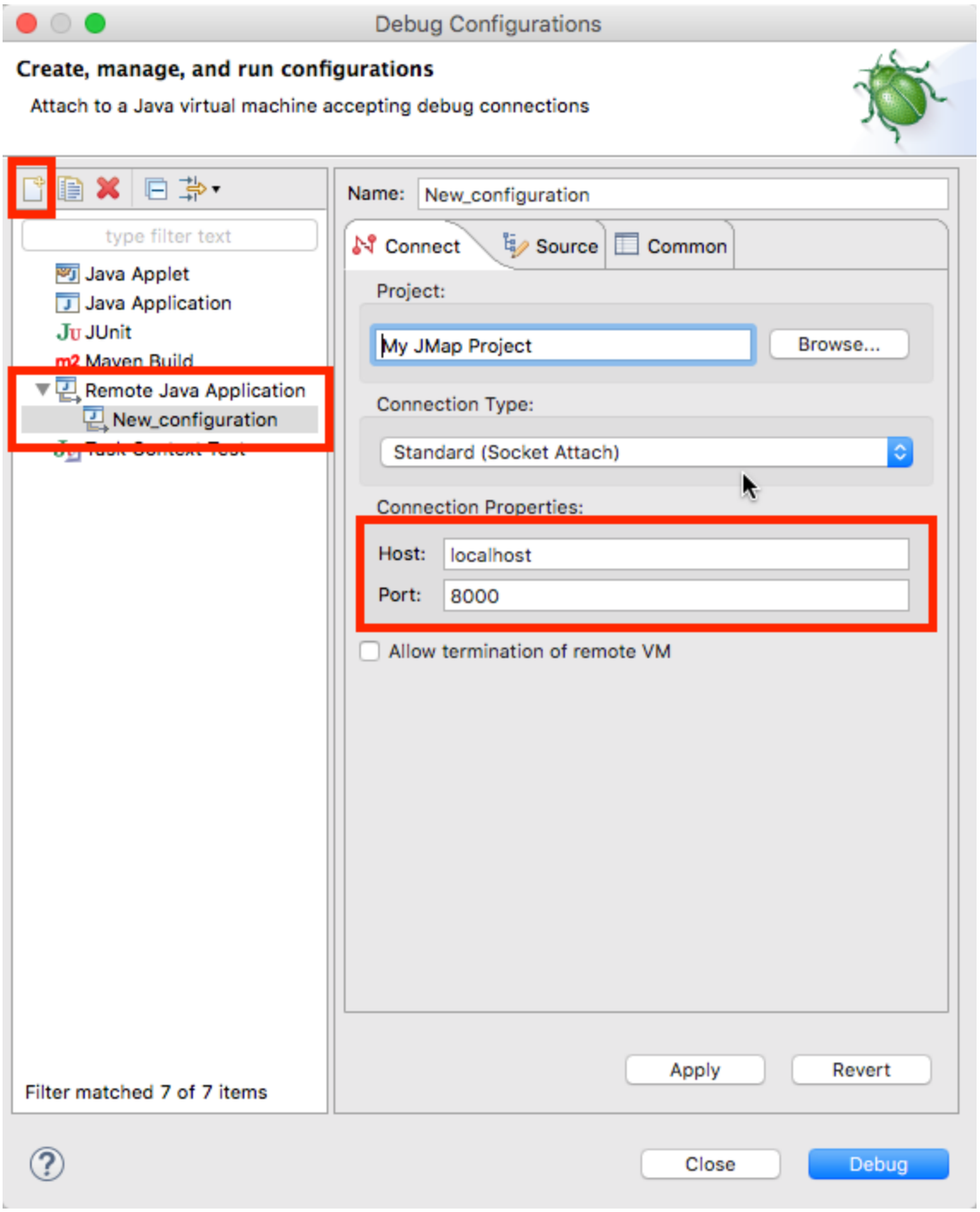
To activate remote debugging in JMap Server, you must modify the file *startjmapserver.vmoptions* located in the *JMap_HOME/bin* directory. The line starting with *-Xdebug* should be added.

```
-Xmx768m
-XX:MaxPermSize=256m
-Djava.awt.headless=true
-Dfile.encoding=ISO-8859-1
-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000
```

Next, you must restart JMap Server. JMap Server will then be in debug mode and it will be awaiting commands from Eclipse. It is not recommended to activate the debug mode in a production environment, as it significantly decreases system performance.

In Eclipse, you must create a remote debugging configuration using the same parameters given in the configuration file shown above (these are the default parameters).





JMap server extensions are modules developed in Java that are added to JMap Server to respond to new types of queries and perform tasks on the server side. Server extensions can contain

configuration interfaces that are integrated to the Extensions section of JMap Admin. Often, a server extension works with a client extension.

To develop a server extension and make it available, you must perform the two following steps:

1. Develop an extension by creating a Java class that implements the JMapServerExtension interface.
2. (Optional) Develop a JMap Admin configuration interface for your extension.
3. Deploy your extension in JMap Server.

See the following sections for more information.

The JMapServerExtension Interface

The first step towards developing a JMap Server extension is to write a class that implements the *JMapServerExtension* interface. This interface includes the following 3 methods, which are called at different moments in the life cycle of the extension:

Methods of the JMapServerExtension interface	
init()	This method is called when JMap Server is launched, when extensions are initially loaded or when the administrator requests to reset the extension from JMap Admin. It is used to initialize the extension. In this method, you can put any code used to prepare the operation of your extension. This could include loading a settings file, checking dependencies, etc.
processRequest (JMapExtensionRequest)	This method is called when JMap Server receives a request destined to your extension. In this method, you must include the code needed to process the request. In addition, the method must return a response resulting from processing the request.
destroy()	This method is called when JMap Server shuts down or when the administrator requests to reset the extension from JMap Admin. It is used to execute the code needed to close the extension. This could include closing files or connections to other systems.

To do its job, your extension can use the services offered by JMap Server. This includes spatial data extraction, access to relational databases connected to JMap Server, access to the system log (log files), etc.

For more information on the services offered by JMap Server, see the *JMap Server Services* section.

JMapExtensionRequest class

You must implement a class derived from the *JMapExtensionRequest* class. This class provides your extension with all the information it needs to perform its work. Requests are typically initiated on the client side and passed to JMap Server for processing by your extension. In this class, you can include all the required properties, but make sure these are all serializable.

When your extension is initialized, the type of your query (full class name, including package) is associated with your extension. This way, when JMap Server receives this type of query, it is automatically directed to your extension (*processRequest* method). For more information on how to make the connection between your query and your extension, refer to the *Deploying Server Extensions* section.

For more information on programming requests, refer to the *Client-Server Communication* section.

JMapExtensionResponse class

You must also implement a class derived from the *JMapServerResponse* class. This class is intended to provide information resulting from the execution of your query. The processRequest method of your extension must return an instance of this class. Depending on the nature of the query, the response may either return a large amount of information or just a query execution status (success, failure, etc.). You can include all the required properties in your class, but make sure they are all serializable.

For more information on programming responses, see the *Client-Server Communication* section.

JMap Server Class

The *JMapServer* class is the main class from which you can access JMap Server's various services. This class is a singleton. You can therefore access it from anywhere using the static method *JMapServer.getInstance()* .

JMapHome

JMap Server's home path is accessed using the static method *getJMapHome()* of the *JMapServer* class. It can be useful to know this path when you wish to read or write data in JMap Server's subdirectories.

Logging

The JMap Server logging tool can record events in log files. The tool's class is *Logger* and it is a singleton. Thus, you can access the single instance using the static *Logger.getInstance()* method.

The various versions of the log method are used to store messages, depending on the type of information to be recorded.

The following table shows the most commonly used methods of the Logger class.

Most commonly used methods of the Logger class	
log(int, String)	Logs a message of the specified level.
log(int, String, String)	Logs a message of the specified level associated with the specified user.
log(int, String, Throwable)	Logs a message of the specified level and the trace of the exception passed as a parameter.
log(int, String, Throwable, String)	Logs a message of the specified level, associated with the specified user, and the trace of the exception passed as a parameter.
setLogLevel(int)	Changes the level of the messages that will be recorded.

The various message levels are defined by constants of the *Logger* class.

- LEVEL_DEBUG
- LEVEL_INFO
- LEVEL_WARNING
- LEVEL_ERROR
- LEVEL_FATAL

The following code example shows how to log a message with the various methods.

```
// Logs a message of level INFO
Logger.getInstance().log(Logger.LEVEL_INFO, "Extension ABC recieved a request for ...")

// Logs a message of level WARNING, tagged to user etardif
Logger.getInstance().log(Logger.LEVEL_WARNING, "Something occurred in Extension ABC ...

// Logs a message of level ERROR, and includes the exception stack trace
Exception e = ...;
Logger.getInstance().log(Logger.LEVEL_ERROR, "An unexpected error occurred in Extension
```

Databases

The connections to the relational databases that JMap Server is connected are available through programming. This greatly simplifies data access because you do not need to open, close and manage the connections to these databases.

JMap Server manages database connections in connection pools. These pools function as follows: when a connection is required, it is borrowed from the pool. It is then used briefly to execute one or more queries. Lastly and most importantly, the connection is returned to the pool and is once again available for other needs. Thus, connections are never closed.

The *getDBConnPool(int)* and *getDBConnPool(String)* methods allow you to get a connection pool (instance of the *DatabaseConnectionPool* class) using its numeric ID or name.

The following table shows the most commonly used methods of the *DatabaseConnectionPool* class.

Most commonly used methods of the DatabaseConnectionPool class	
borrowConnection()	Borrows a JDBC connection from the pool. The connection is then reserved exclusively.
returnConnection(Connection)	Returns a JDBC connection borrowed from the pool. It is very important to call this method after using a connection.
getStatus()	Provides the status of the connection pool. Can be called to validate that the pool is working correctly before borrowing a connection. The possible statuses are defined by constants of the *ConnectionPoolInfo* class (CONNECTION_NOT_TESTED, CONNECTION_ERROR or CONNECTION_OK).

The following source code example shows how to use a pool of connections to databases.

```
DatabaseConnectionPool pool = JMapServer.getInstance().getDBConnPool("parcels");
Connection conn = null;

try
{
  conn = pool.borrowConnection();

  // use connection to do queries.....

}
catch (SQLException e)
{
  e.printStackTrace();
}
finally
{
  // It is very important to return the connection to the pool.
  // Doing it in a finally clause is a good practice
  if (conn != null)
    pool.returnConnection(conn);
}
```

Extracting spatial data

Spatial data can be extracted using the JMap Server data manager (*JMapServerDataManager* class). The data manager can be accessed using the *getDataManager()* method of the *JMapServer* class.

The following table shows the most commonly used methods of the *JMapServerDataManager* class.

Most commonly used methods of the JMapServerDataManager class	
extractElements (JMapServerProject, JMapServerVectorLayer, QueryFilter[], Attribute[])	Extracts the spatial data and attributes that pass the filters passed as parameters, for the specified of the specified project. Only attributes that are specified in the last parameter are included in the result.
extractElements (String)	Extracts spatial data and its attributes based on the query passed as parameters. The general syntax for queries is the following: select element from $ $ source {$ {project} $ PROJECT_NAME layer layer_name {}} WHERE condition Example: select element from $ $ source {$ {project} The World Countries {$ layer}} Where COUNTRY = 'Peru'
extractElements (String, long[])	Extracts spatial data and its attributes based on to the query passed as parameters and the list of identifiers specified. Only items whose identifier is in the list are returned.
extractElements (String, long[], OrientedRectangle)	Extracts spatial data and its attributes based on the query passed as parameters, the list of identifiers and the region specified. Only elements whose identifier is in the list and who intersect the region are returned.

To extract data, you can also use filters. Filters are objects that control what data must be extracted based on various criteria.

Filter classes are all derived from the abstract class *QueryFilter*. The following table shows all the types of filters that are available.

Types of query filters

AttributeFilter	Sets a condition based on a data attribute. The condition is defined by an attribute and a set of values?Only data for which the attribute has a value included in the set of values will pass the filter.
GeometryTypes Filter	Sets a condition based on the type of data geometry. The condition is defined by one or more types of geometries. Only data with a geometry type matching the filter will pass the filter.
SpatialQueryFilter	Sets a spatial condition. Only data meeting the condition will pass the filter. The condition is defined by a geometry and a constraint. For example: all geometries that intersect the specified polygon, all geometries that contain the specified point, etc.
SQLQueryFilter	Sets a condition in SQL. This is the equivalent of the where clause in an SQL query. The where clause is interpreted by the database system that contains the data.

The following source code example shows how to extract spatial data using a spatial filter.

```
JMapServerProject serverProject = ...
JMapServerVectorLayer serverLayer = ...
Polygon region = ...

final JMapServerDataManager dataMgr = JMapServer.getInstance().getDataManager();

// Create a new spatial filter for all elements that intersect the polygon
final SpatialQueryFilter newFilter = new SpatialQueryFilter();

newFilter.setGeometry(region);
newFilter.setType(SpatialQueryFilter.SPATIAL_OP_INTERSECTS);

// Set the projection on the filter to indicate the coordinate system of the geometry
newFilter.setProjection(serverProject.getProject().getMapProjection());

// Do the extraction using the data manager. All attributes of the layer will be included
JMapGeoElement[] result = dataMgr.extractElements(serverProject,
                                                  serverLayer,
                                                  new QueryFilter[]{newFilter},
                                                  serverLayer.getBoundAttributes() );
```

Sending emails

The *sendMail()* static method of the *MailService* class allows you to send emails from JMap Server. In order for the emails to be sent successfully, JMap Server must be connected to an SMTP server. This connection can be configured during the JMap installation or it can be defined in the Settings section of JMap Admin.

```
String to = "jo32@gmail.com;ann122@hotmail.com";
String from = "admin@123map.com";

try
{
  MailService.sendMail(MailService.toAddresses(to), "Map extraction completed", "The da
}
catch (AddressException e)
{
  e.printStackTrace();
}
catch (Exception e)
{
  e.printStackTrace();
}
```

Session Manager

The JMap Server Session Manager (*JMapServerSessionManager* class) is responsible for managing active sessions in the system. Among other things, it provides access to the user connected to a session by using the session ID. This session number is accessible from every request received by JMap Server, including queries destined to server extensions. The session manager can therefore be used to know the identity of the user who initiated a query.

The following code example shows how to access the user who initiated a query.

```
public JMapExtensionResponse processRequest(JMapExtensionRequest request)
{
  int sessionId = request.getSessionId();
  User user = JMapServer.getInstance().getSessionManager().getSessionUser(sessionId);
  System.out.println("##### Request originating from user: " + user.getName() + " (" + us

  ...
}
```

User Manager

The User Manager (*UserManager* interface) provides access to the list of users and groups used by JMap Server for access control. It also allows you to access user information. If you develop a server extension that must manage its own list of permissions, it can be very useful to access the list of users that the system uses.

The *getUserManager()* method of the *JMapServer* class returns the User Manager (class that implements the *UserManager* interface) being used.

Most commonly used methods of the UserManager class

getUser(String)	Returns the user (instance of the *User* class) whose name is specified as a parameter.
getGroup(String)	Returns the group (instance of the *Group* class) whose name is specified as a parameter.
users()	Returns the list of users (instances of the *User* class) used by JMap Server.
groups()	Returns the list of groups (instances of the *Group* class) used by JMap Server.

The *User* class contains the information on a user (full name, email address, etc.).

Workspaces

In JMap, *workspaces* are spaces used for individual storage of user data. Each workspace is actually a separate subdirectory of JMap Server. The workspaces are used to store contexts created by users. As a programmer, you can use them to store data.

The *getWorkSpaceManager()* method of the *JMapServer* class allows you to access the workspaces manager (*WorkSpaceManager* class), which provides some useful methods related to worskpaces. By default, workspaces are located in the JMap_HOME / workspaces directory.

The following table shows the most useful methods of the *WorkSpaceManager* class.

Most commonly used methods of the WorkSpaceManager class	
getUserWSDirectory(String)	Returns the full path to the workspace directory for the user specified as a parameter.
emptyUserWS(String)	Erases all the content of the workspace for the user specified as a parameter.
deleteUserWS(String)	Clears the workspace directory of the user specified as a parameter.

To be deployed in JMap Server, server extensions must follow certain rules. If these rules are met, the extension appears in the Extensions section of JMap Admin.

1 - Group extension classes in an archive (JAR)

All the classes of the extension must be contained in a single JAR archive file. Use a meaningful and unique name.

2 - Include a manifest file

The extension archive must include a manifest.mf file with the following entries:

Variable	Description
extension_class	Identifies the main class of the extension. This is the class that implements the JMapServerExtension interface.
extension_request	Identifies the class used as the query for this extension. This is the entry that will route requests to your extension. If your extension supports multiple classes of requests, they must all derive from this class.
extension_response	Identifies the class used as a response to this extension. If your extension supports several classes of responses, they must all be derived from this class.
extension_name	Specifies the name of the extension. This name appears in JMap Admin in the Extensions section.
extension_version	Specifies the version number of the extension. This information appears in JMap Admin in the Extensions section. The version number is used only to simplify the management of extensions.

Here is an example of the contents of a manifest file:

```
extension_class: jmap.extensions.tracking.server.TrackingExtension
extension_name: Tracking
extension_request: jmap.extensions.tracking.common.TrackingRequest
extension_response: jmap.extensions.tracking.common.TrackingResponse
extension_version: 5.0.0010
```

3 - Place the file in the correct directory

The JAR file of the extension must be placed in the server extensions directory (*JMAP_HOME/extensions*).

4 - (Optional) Place the files from the configuration interface of the extension in the appropriate directory

The files that make up the configuration interface (JSP pages) must be copied to the directory used for this purpose (*JMAP_HOME/jmapadmin/extensions*).

Each server extension may include a configuration interface integrated to JMap Admin. Using this interface, JMap administrators can configure the operational parameters of your extension (e.g. security, database connectivity, a layer selection, etc.). Note that this interface is completely optional.

This interface consists of one or more JSP pages. The main JSP page (which is called first) must absolutely have the same name as the extension class. For example, if the class name of the server extension is

jmap.extensions.tracking.server.TrackingServerExtension

then the main JSP page must be

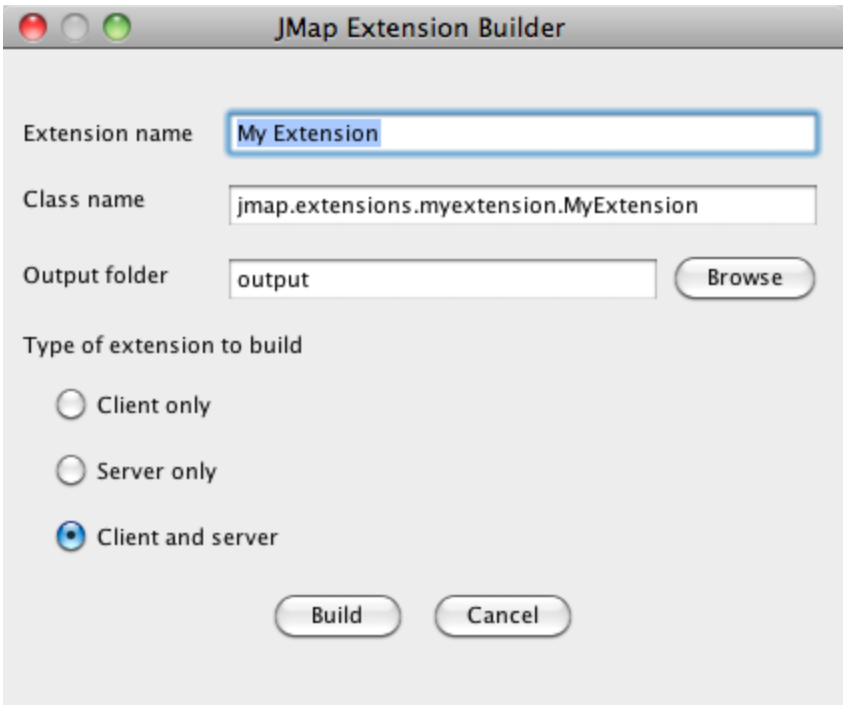
jmap.extensions.tracking.server.TrackingServerExtension.jsp

For more information on programming these JSP pages, refer to the examples included in the SDK.

The Java API documentation for JMap 6.5 is available online at http://dev.k2geospatial.com/jmap/javadoc/6.5/.

A report showing the API differences between JMap 6.0 and JMap 6.5 is available online at http://dev.k2geospatial.com/jmap/javadoc/6.5/changes.html.

The JMap Extensiom Builder is a tool that allows you to quickly generate the source code for JMap client and server extensions. The generated source code is basic, and you must complete it in order to implement the functions of your extension. This tool helps you save time by allowing developers to focus on the real purpose of their development activity instead of dealing with the technical details of programming and deploying JMap extensions.



The tool has a GUI and may also be used in the command line, without a GUI. The parameters accepted by the extension builder are shown below. Note that they are all optional, and default values are used for the missing parameters.

Parameter	Description
-name *extensionname*	The extension name as it will appear in JMap Admin. It can contain spaces. e.g. tracking goods, network analysis, etc.
-class *classname*	The full name (with package) of the main class of the extension. The names of the other classes needed by the extension will be derived from that name. e.g. jmap.extensions.tracking
-dest *destfolder*	The target directory for the generated files. If the directory already exists, it will be deleted and recreated. The default value is **output**.
-target *target*	The type of extension to generate. Valid values are **client**, **server** or **both**. The default is **both**. Different classes and packages are created according to the selected option.
-gui *true|false*	Determines whether the GUI should be displayed or not. Possible values are **true** and **false**. The default value is **true**. If the GUI is

	used, its fields are initialized with the values provided as parameters.

The easiest way to run the extension builder is to use the Ant script provided with the tool. The parameters passed to the tool can be modified within the Ant script.

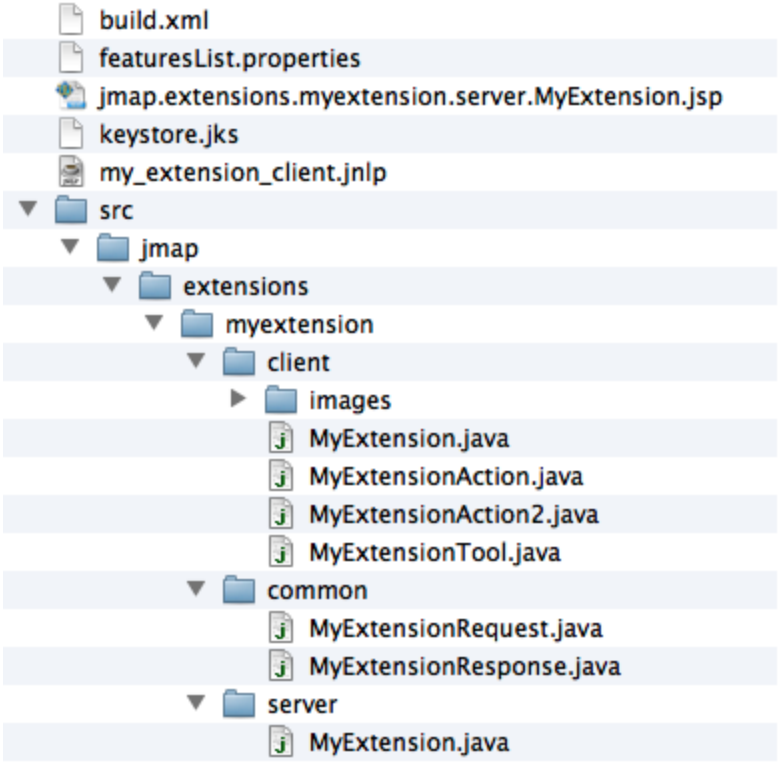
```
ant -f extensionbuilder.xml
```

The extension builder can also be executed using the command line (on Mac or Linux, replace "" with "" in the classpath).

```
java -classpath extensionbuilder.jar;../../lib/kheops_util.jar jmap.sdk.tools.extensionbuilder.ExtensionBuilder -name extensionname -class classname -dest destfolder -target target -gui true\false
```

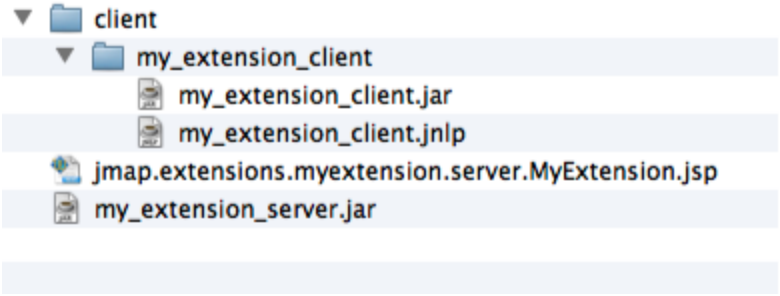
After running the tool, the target directory contains the sources of the extension generated as well as some other files. If a client extension was requested (*client* or *both* options), there will be a JNLP file, a keystore.jks file (for your client extension's signature), and a featureslist.properties file. If a server extension was requested (*server* or *both* options), there will be a JSP file (configuration interface of server extension). An Ant script (build.xml) is also created in the directory. This script can be used to compile the extension and generate the JAR archives for the client and server.

The following image shows the structure of the generated files.



La structure des fichiers générés par le générateur d'extensions

To run the Ant script, simply run the ant command from the destination directory. After the script has been run, the *dist* directory will contain all the files to be deployed on JMap Server. The following image shows the structure of the generated files.



La structures des fichiers générés par l'exécution du script Ant de l'extension

JMap Web Development

This section details how to develop for JMap Web as well as how to integrate JMap Web into other web applications.

JMap Web is a web application that uses technologies such as HTML5, JavaScript, CSS and JSON. The JMap Web application is built on top of third-party JavaScript libraries including OpenLayers, jQuery and many others.

Brief Overview of JMap Web's Design

JMap Web provides a web application capable of interacting with JMap Server. Its user interface is primarily intended for use on laptop and desktop web browsers.

The key elements of a deployed JMap Web application are the following:

- The client: Can be described as a single-page application (index.jsp). The client is considered the web page as it is rendered in the user's web browser.
- A Web Map Service: Provides access to the map's tiles.
- The AJAX Dispatcher: A web service to which HTTP requests are sent. The AJAX Dispatcher then dispatches the request to the appropriate "Action Handler". This will be further discussed in the Sending Server Requests and Custom Actions section of this document.

Third-party JavaScript Libraries Used

At the time of writing this document, JMap Web currently uses the following third-party Javascript libraries. Your extensions may leverage these libraries without any additional configuration. The following table includes a brief description of how JMap Web uses these libraries.

Bibliothèque	**Description**
DataTables (v1.9.4)	jQuery plugin that easily enables the creation of powerful HTML tables for data representation. This is primarily used to display search results.
fancyBox (v2.15)	Displays content in a "lightbox" style interface. JMap Web uses fancyBox to display the full size version of documents included in mouseover popups.
Google Maps (v3)	Necessary to access Google Maps's Roadmap, Terrain, Satellite and Hybrid base layers.

jQuery (v1.9.1)	jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.
jQuery-ui (v1.11.4)	Library of user interface components relying on the jQuery framework. Currently, JMap Web uses a custom build that includes only the autocomplete component.
moment.js (v2.10.6)	Library used to parse, validate, manipulate, and display dates in JavaScript.
malihu-custom-scrollbar-plugin (v3.0.7)	Styles the appearance of scroll bars for overflowing mouseover popups. Its primary function is to allow a more consistent appearance in various browsers.
OpenLayers (v2.13.1)	This is JMap Web's most important library. OpenLayers allows JMap Web to display a map and manage the vast majority the user's interaction with it.
Proj4js (v1.1.0)	Proj4js is a library that's used to translate geographic coordinates from one map projection system to another. Proj4js is used by OpenLayers, but isn't included with it. In order to perform transformations for named projections (ex: “EPSG:3857”), projections must be defined in Proj4js. By default, only a small number of projections are defined. JMap Web includes a Proj4js projection definition file for each projection that's supported by JMap Server. Those projection definitions will be loaded into memory as they are required.
Twitter Bootstrap (3.1.1)	«Front-end framework» used to create user interface elements.
bootstrap-datetimepicker (v4.15.35)	Date/time picker widget based on Twitter Bootstrap.
bootstrap-multiselect (v0.9.10)	JQuery multiselect plugin based on Twitter Bootstrap.

JMap Javascript Libraries and Their Architecture

In addition to the libraries previously mentioned, JMap Web also includes its own libraries based on OpenLayers's Class base type. This allows JMap Web to benefit from object oriented programming paradigms in a Javascript context (which usually favors a more prototypal and functionnal approach).

A JMap Javascript library essentially consists of a collection of Javascript files, styles sheets (CSS) and resources (images, sounds, etc.).

JMap Javascript libraries have been designed for use in different application templates of the "JMap Web/JMap Mobile" deployment type. By spreading the code across several libraries, code reuse is

possible in various environments such as JMap Web and JMap Mobile. JMap Javascript libraries define unique behavior for specific contexts.

JMap Web uses the following JMap Javascript libraries:

- core
- desktop_ui

Reusable model classes are usually defined in the *core* library. As such, it is frequently used and is included in the vast majority of "JMap Web/JMap Mobile" application templates.

The *core* library is written in JavaScript ES5. In addition to JMap model classes, it includes custom OpenLayers controls and layers, HTML5 functionality wrappers and more. The *core* library is JMap Web and JMap Mobile's lowest level Javascript library. It handles general tasks such as the initialization process of the map. All other library is then loaded on top of it as an extension.

As for *desktop_ui* , it mostly contains classes that focus on user interface elements. It directly makes use of the DataTables, jQuery and Twitter Bootstrap Javascript libraries. All visual elements that are not managed by OpenLayers are defined within *desktop_ui* .

Included in JMap 6.5 is JMap Web's public API which allows developers to easily integrate with JMap functionalities.

A generated JSdoc documentation of the javascript API is available online at http://dev.k2geospatial.com/jmap/web/api/6.5/.

Before moving onto the creation of a JMap Web extension, this section will explain JMap Web's initialization process.

Note : This section refers to the JMap Web application template as it is included when JMap Server is installed. The process, as briefly described here, differs when you choose to embed a JMap deployment within your existing web application. For more details, consult the Embedding a JMap Deployment Into Your own Application section of this document.

The JMap Web application template's files are located in $JMAP_HOME$/applications/templates/html/web.

index.jsp

This is the web document that will be served by JMap Server as the application is being requested by the user. Open this file in a text editor application. As you can notice, both jQuery and a `jmap.min.js` are loaded in the `head` portion of the document.

Further down, the map is initialized once the document is ready for manipulation.

```
$(document).ready(function() {
  var options = {
```

```
    jmapUrl: '$APPLICATION_PROTOCOL$://$APPLICATION_HOST$:$APPLICATION_WEBPORT$$PATH$',
    mapConfig: {
      // Configuration properties
    },
    onMapInit: function() {
      console.log('Map was initialized.');

      var resizeUi = function() {
        if (JMap.app && JMap.app.map)
        {
          $(JMap.app.map.div).height($(window).height()).width($(window).width());
          JMap.app.map.updateSize();
          JMap.app.popupManager.updateDrawnPopups();
        }
      };

      $(window).on('resize', resizeUi);

      // Object auto zoom handling...
    }
  };

  // Region auto zoom handling...

  $('#map').height($(window).height()).width($(window).width());
  JMap.initialize(document.getElementById('map'), options);
});
```

JMap.initialize(map, options)

The `JMap.initialize` function is defined in the jmap.min.js file. It loads all necessary dependencies and will then initialize all of the application's required objects. Among those objects are the globally available `JMap.app` and `JMap.app.map`.

The `JMap.app` object is an instance of the `core` library's Application class. The `JMap.app.map` object is an instance of the OpenLayers.Map class.

Argument	Description
map	**Required** {DOMElement|String} The element or id of an existing element in your page that will contain the map.
options	**Required** {Object} Should at least provide a `jmapUrl` {String} property. Supports the following property keys: • jmapUrl: Required. {String} The url of your JMap deployment. • mapConfig: {Object} Additional map configuration options. • onMapInit: {Function} Handler to be called once the map initialisation process is completed. Useful to add new `OpenLayers.Control`s once the OpenLayers library is loaded.

The `mapConfig` object may include the following properties to customize the application. The default value is identified in bold.

mapConfig properties	Description
Map Options	
displayUnits	**null** | 'm' | 'km' | 'ft' | 'mi' | 'in' | '°' {String} The map unit. Defaults to the JMap project's value.
initialZoom	**null** {Array} Map coordinates that describe bounds: [left, bottom, right, top]
mapUnits	**null** | 'm' | 'km' | 'ft' | 'mi' | 'in' | '°' {String} The map unit. Defaults to the JMap project's value.
maximumExtent	**null** {Array} Map coordinates that describe bounds: [left, bottom, right, top]
projection	**null** {Object} If supplied, it must contain a string `code` property that corresponds to an EPSG code
Additional Map Layers	
addShowPositionLayer	true | **false**
Services on Launch	
activateGeolocationServiceOnLaunch	true | **false**
loadGoogleMapsApiOnLaunch	true | **false** - Will automatically be set to true if the deployment contains at least one Google layer or if any of the following is true: addGoogleDirections, addGoogleGeocoding, addGoogleStreetView.
Map and Navigation Options	
addGeolocateButton	**true** | false
addInitialViewButton	**true** | false
addMapOverview	**true** | false
addMousePosition	**true** | false
addPanControls	
addScaleBar	**true** | false
addZoomInButton	**true** | false
addZoomOutButton	**true** | false
isMapOverviewMaximized	true | **false**
JMap Tools	
addInfoReportTool	**true** | false

addMeasureAreaTool	**true** | false
addMeasureDistanceTool	**true** | false
addMeasureCircularAreaTool	**true** | false
addMouseOverTool	**true** | false
addRedLiningTool	**true** | false

JMap Edition Functionalities

addEditionTools	true | **false**
addEditionCreateElementTools	**true** | false - The addEditionTools property must also be set to true for this to take effect.
addEditionSelectElementTool	**true** | false - The addEditionTools property must also be set to true for this to take effect.
addEditionShowElementFormButton	**true** | false - The addEditionTools property must also be set to true for this to take effect.

JMap Selection Functionalities

addSelectionTools	true | **false** - The addSelectionTools property must also be set to true for this to take effect.
addCircleSelectionTool	**true** | false - The addSelectionTools property must also be set to true for this to take effect.
addLineSelectionTool	**true** | false - The addSelectionTools property must also be set to true for this to take effect.
addPointSelectionTool	**true** | false - The addSelectionTools property must also be set to true for this to take effect.
addRectangleSelectionTool	**true** | false - The addSelectionTools property must also be set to true for this to take effect.
addShapeSelectionTool	**true** | false - The addSelectionTools property must also be set to true for this to take effect.

Other Functionalities

addFullScreenButton	**true** | false
addLayersMainMenuItem	**true** | false
addLogo	**true** | false
addLogoutAsideMenuLink	**true** | false
addParametersAsideMenuLink	**true** | false
addPrintButton	**true** | false

addSearchMainMenuItem	**true** | false

Third Party Functionalities

addGoogleDirections	**true** | false
addGoogleGeocoding	**true** | false
addGoogleStreetView	**true** | false

After the Initialization Process

Once the map is successfully initialized, you may access it via Javascript by using the `JMap.app` global variable.

Map manipulation is possible by accessing the `JMap.app.map` global variable. This variable is the instance of `OpenLayers.Map in` use and you can interact with it by using the OpenLayers API.

URL Parameters

JMap Web supports the following query string parameters:

```
autozoom {String}
```

Zooms on a specified area or object on the initial view of a map. Two types of autoZooms can be used, *region* or *object* , the type is determined by the first argument: 'type'.

AutoZoom **Types**	**Description**
Region	The user specifies the viewing area by specifying a rectangle... Syntax: “type;x;y;width;height” • type: {String} The type of this request. In this case, must be "region". • x: {Number} The X value of the lower left coordinate of the rectangle. • y: {Number} The Y value of the lower left coordinate of the rectangle. • width: {Number} Width of the rectangle. • height: {Number} Height of the rectangle. Example: ?autozoom=region;9;39;20;20
Object	The user specifies an object to zoom to... Syntax: “type;layerName;field;value;maxScale”

- type: {String} The type of this request. In this case, must be "object".
- layerName: {String} The name of the layer which contains the object to zoom to.
- field: {String} The field that contains the value of the object to zoom to.
- value: {Number | String} The specified value of the object to zoom to. If the field is a string, use ''. See example below.
- maxScale: {Number, optional} The maximum scale the map should respect when displaying results.

Number value example: ?autozoom=object;citiesLayer;city_id;1032

String value example: ?
autozoom=object;citiesLayer;city_name;'montreal';15

This section provides an overview of developing extensions for JMap Web.

Extensions are modular and usually offer functions suited for specific tasks. On its own, JMap Web provides a basic set of features. JMap Web extensions allow the customization and the inclusion of new features to deployments based on the JMap Web application template.

Just like JMap Pro extensions, JMap administrators may choose which extensions they want to include during the application deployment assistant. The selected extensions will be loaded as part of JMap Web's initialization process.

Similar to a JMap Javascript library, a JMap Web extension consists of a collection of Javascript files, style sheets (CSS) and resources (images, sounds, etc.).

You can use the **JMap Web extension builder** tool to rapidly scaffold a Web extension's files. Open a command line/terminal window or use a file browser and navigate to the *tools/webextensionbuilder* directory of the JMap 6.5 SDK installed on your computer.

Using the JMap Web extension builder

You will find a `webextensionbuilder.xml` file. Open it in a text editor. Edit it in order to specify a list of arguments that will allow you to create your own extension. The following arguments are required:

Argument	Description
fullname	{String} The complete human-readable name of the extension. This is what the JMap administrators and users will see. A simplified *shortname* will be derived from this value. The shortname will notably be used as a name for various files.
namespace	{String} The name of your extension's Javascript namespace (a global object variable). If you plan on having several of your extensions deployed at once, you may specify a namespace that contains **at most** one period "." in order to seperate the items of your namespace. Example: `MyCompany.HelloWorld`.
version	{String} Your extension's version number. No specific format is required.
dest	Location on disk where the extension will be created.

Example of a modified `webextensionbuilder.xml` file used to create your extension:

```
<project name="JMap 6.5 SDK - Web Extension Builder" basedir="." default="run">

  <!-- set global properties for this build -->
  <property file="../../sdk.properties"/>

  <target name="run">
    <java fork="true" classname="jmap.sdk.tools.webextensionbuilder.WebExtensionBuilder"
          classpath="webextensionbuilder.jar;${sdk_classpath}">
      <!-- Use default parameter values. Uncomment following line to use other parameters
      <arg line="-fullname 'Web SDK Documentation Example' -namespace Example -version 1.0
    </java>
  </target>
</project>
```

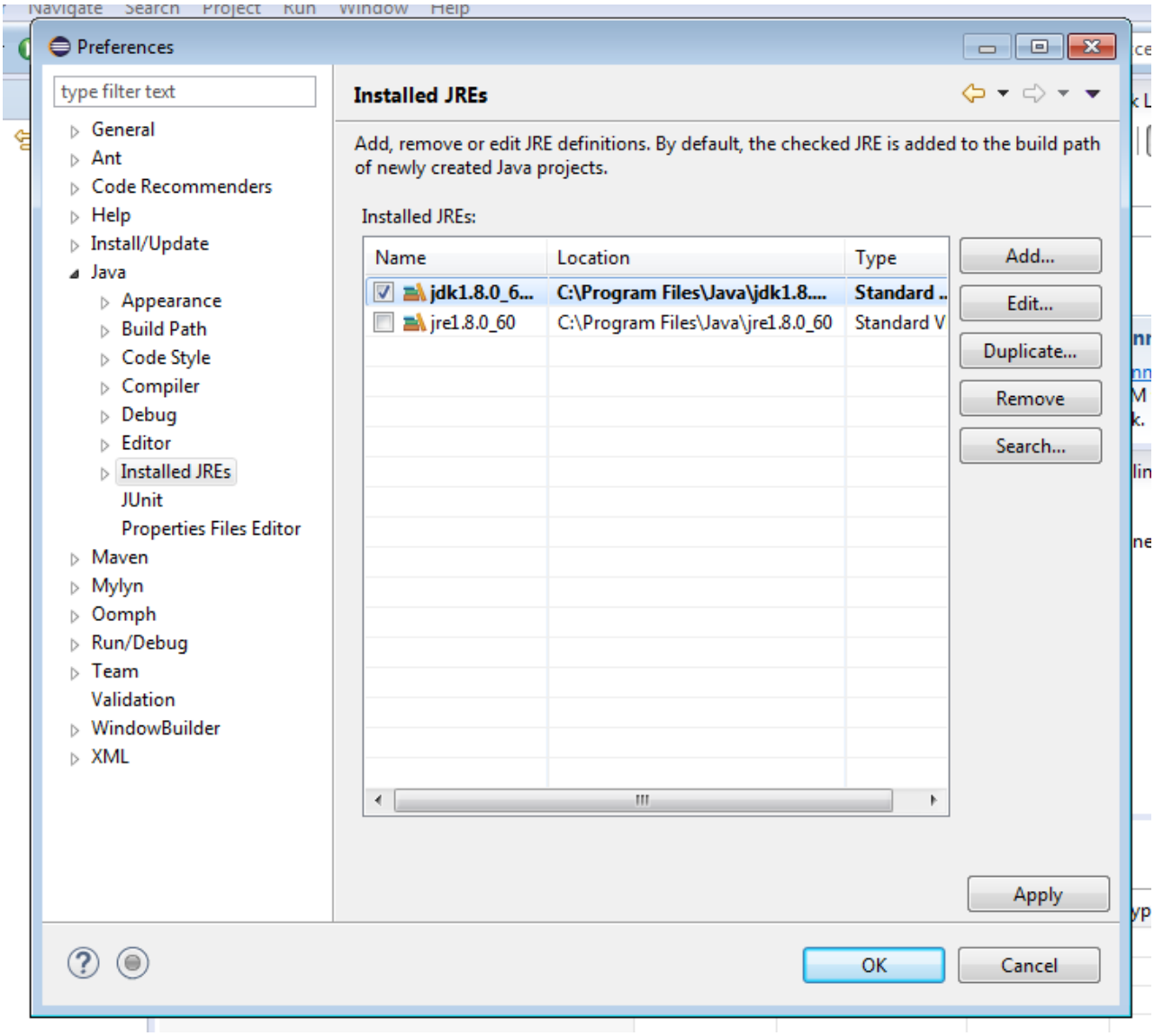
Creating the extension using ant on the command line

From the terminal, invoke the ant script:

```
ant -f webextensionbuilder.xml
```

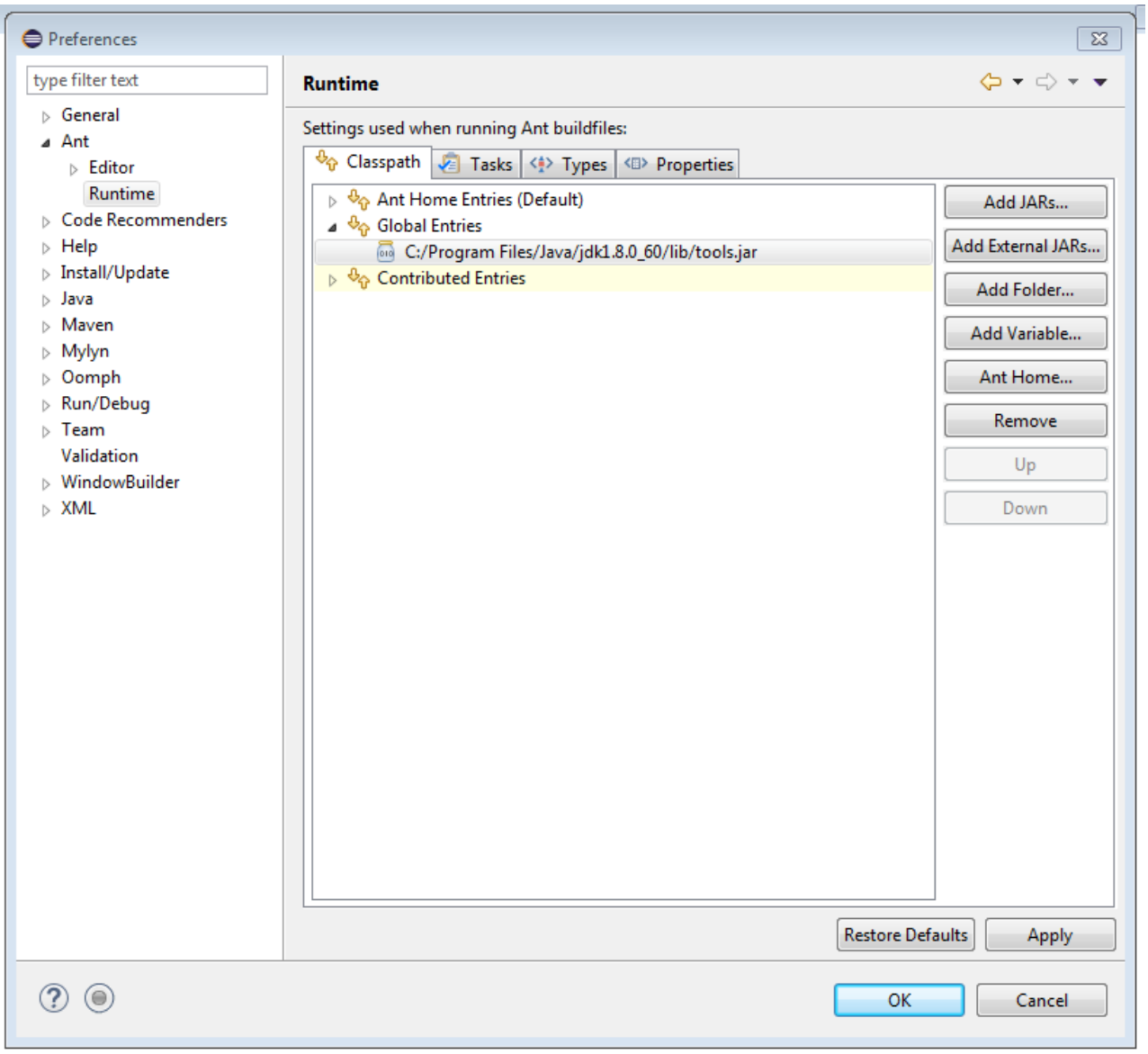
Creating the extension using ant through Eclipse

To create a JMap Web extension using Eclipse, the following should be installed on your system, make sure a Java Developer Kit is installed and configured as the default JRE in Eclipse:



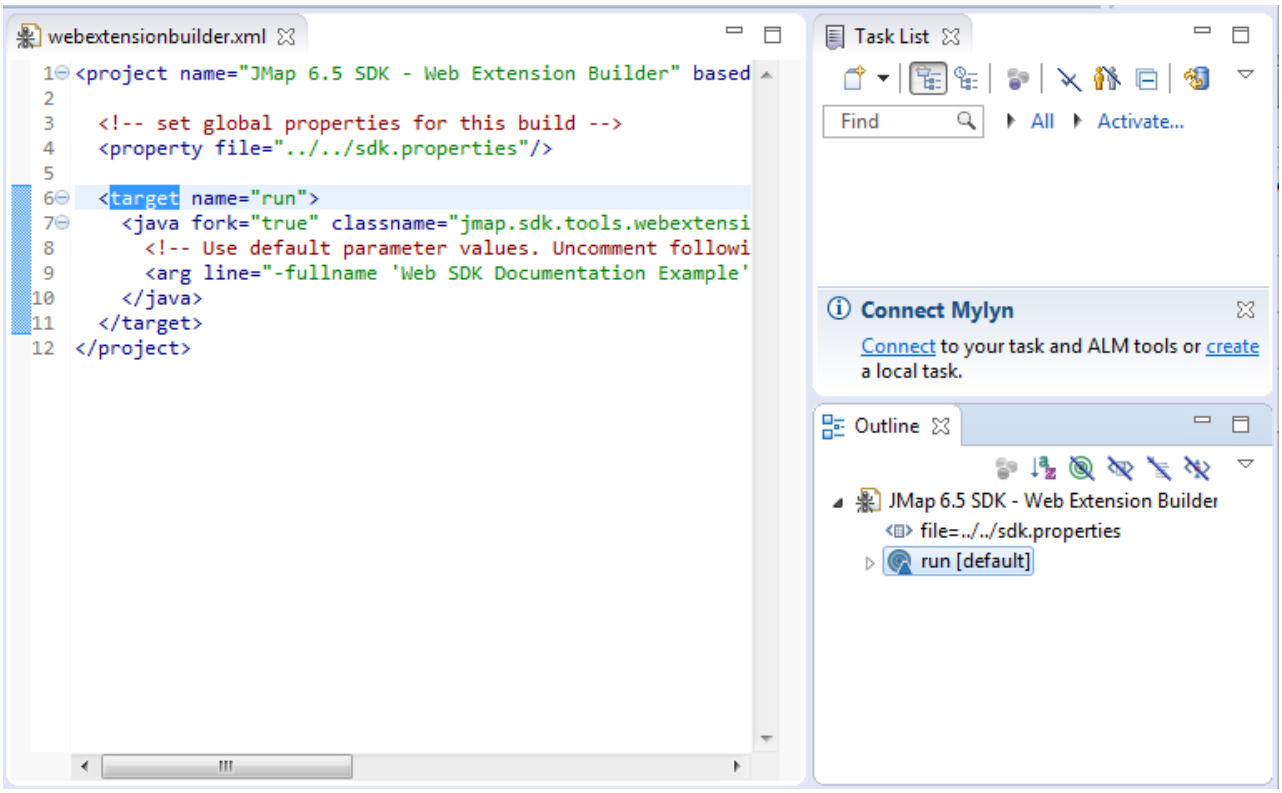
Eclipse's Installed JREs

You should also make sure that your JDK's tools.jar file is added to Eclipse's Ant Runtime Configuration:



Eclipse's Ant Runtime Configuration. Add JDK_HOME/lib/tools.jar to the Global Entries protion of the class path.

Execute the build task by opening the `webextensionbuilder.xml` file in Eclipse. Execute the run target.



Execute the "run" target.

Understanding the extension's files

After the ant script's execution, you will have an extension's boilerplate code at the specified location.

```
output
+- - web_sdk_documentation_example
   +- - actions
   ¦   +- - build.xml
   ¦   +- - readme.markdown
   ¦   +- - src
   ¦       +- - Example
   ¦           +- - SampleAction.java
   +- - assets
   ¦   +- - css
   ¦   ¦   +- - web_sdk_documentation_example.css
   ¦   +- - js
   ¦       +- - web_sdk_documentation_example.js
   +- - extension.json

7 directories, 6 files
```

By default and for demonstration purposes, an AJAX Dispatcher action called *web_sdk_documentation_example_sample_action* is included. For more details on AJAX Dispatcher actions, refer yourself to the Sending Server Requests and Custom Actions example.

Your extension's *generated* CSS and Javascript assets will be loaded during the JMap initialization process.

All other CSS or Javascript resources will need to be loaded programmatically in the generated Javascript file (in this case the web_sdk_documentation_example.js file).

Since extension files are loaded at runtime, it is important that you do not rename or move the generated CSS and Javascript files. The name of those files should always be the extension's short name as defined in the extension.json file.

The source code of the generated Javascript file includes comments that describe the **required** properties and functions that make up your JMap Web extension. It is recommend that you read the contents of this file so that you can familiarize yourself with its content.

Among the generated files is *extension.json* . This file will serve as a manifest for JMap Server. It must contain valid JSON. This file must not be removed or renamed.

Interacting With OpenLayers

As previously mentioned, a *JMap.app* global variable is defined during initialization. One of that object's responsibility is to capture a reference to your application's Map.

JMap.app.map is your application's instance of the OpenLayers.Map class. JMap's Javascript libraries use the OpenLayers API in order to accomplish many tasks. Tasks such as layer manipulation, operations on coordinates/bounds as well as managing the map's controls and popups.

OpenLayers offers a large gallery of development examples. It is recommended that you use this resource throughout your development as a way to familiarize yourself with their API. The gallery and the API documentation are indispensable resources that you should consult frequently.

Examples

This portion of the documentation offers examples of several tasks that can be accomplished in JMap Web extensions.

Adding a Button

The OpenLayers API contains a OpenLayers.Control.Button class. It allows you to easily add a button to JMap Web's interface. This button will then be able to launch event listeners whenever it is clicked. It may also be styled by your extension's stylesheet.

As a part of JMap Web's interface, the main control panel presents a collection of `OpenLayers.Control.Button`s. This panel is in fact an instance of OpenLayers.Control.Panel.

In this example, you will create your own button and add it to the main control panel.

Add the following snippet of code to your extension's init method. Update your deployed application afterwards.

```
/*
 * Returns an array of OpenLayers.Control instances that match the criteria.
 */
var panels = JMap.app.map.getControlsBy('displayClass', 'jmapMainPanel');

/*
 * By default, there's only one instance of OpenLayers.Control.Pane that has
 * its displayClass set to 'jmapMainPanel'.
 *
 * Therefore, you can directly access the first item in
 * the array. This is for demonstration purposes only.
 */
var myPanel = panels[0];

var myButton = new OpenLayers.Control.Button({
  /*
   * CSS class names applied to your button's <div> will be prefixed with this
   * string.
   */
  displayClass: 'myButtonClass',
  /*
   * The function that will be launched once your button clicks.
   */
  trigger: function(event) {
    alert('The button was clicked');
  }
});

/*
 * Add the newly created button to the map.
 */
myPanel.addControls([myButton]);
```

A new button should be visible. It is being represented by the following DOMElement node:

```
<div class="myButtonClassItemInactive olButton"></div>
```

At the time of the initialization of the button, it is possible to specify a *type* corresponding to either *OpenLayers.Control.TYPE_BUTTON* (default), *OpenLayers.Control.TYPE_TOGGLE* (alternates between an active/inactive state on each click) or *OpenLayers.Control.TYPE_TOOL* (only one button among the OpenLayers.Control.Panel can be active at any given time).

To encourage a consistent appearance for buttons in the main panel, the following styles are applied.

```
.jmap .jmapMainPanel > .olButton {
    display: inline-block;
    height: 40px;
    width: 40px;

    cursor: pointer;

    background-color: #636D6F;
    background-image: url("../images/Sprite_40x40.png");
    background-repeat: no-repeat;
    box-shadow: 0px 0px 3px rgba(19, 18, 18, 0.26);


    moz-box-shadow: 0px 0px 3px rgba(19, 18, 18, 0.26);
    webkit-box-shadow: 0px 0px 3px rgba(19, 18, 18, 0.26);
}
```

It is recommended that you use these styles as a starting point, however you can define your own styles using css:

```
.myButtonClassItemInactive {
  background-color: #F00 !important;
}

/*
 * Only applicable if you specified a `OpenLayers.Control.TYPE_TOGGLE`
 * button type.
 */
.myButtonClassItemActive {
  background-color: #0F0 !important;
}
```

You don't need to specify a `trigger` property if you specified a `OpenLayers.Control.TYPE_TOGGLE` button type. You should instead register an event listener on your button.

```
myButton.events.register('activate', this, function(event) {
  alert('On!');
});

myButton.events.register('deactivate', this, function(event) {
  alert('Off!');
});
```

Managing Tools

Note: Knowledge of OpenLayer's Class base type is strongly recommended before continuing with this documentation.

Several classes were developed in order to support various types of map interactions when touch/mouse events are performed. They are the following:

JMap.Html5Core.Control.TouchClickControl : This class inherits from *OpenLayers.Control* . A single instance is created and added to the map during the initialization process. That control's *handler* property can detect two types of events: *click* and *holdreleased* . Your tools can implement functions allowing you to react to both of these types of events.

JMap.Html5Core.Tool.ToolManager : During the initialization process, a single instance of this class is produced and is made accessible via the *JMap.app.clickToolManager* variable. This object keeps track of all registered tools as well as the **single currently activated tool**. That object's *currentTool* will be the recipient of the *click* and *holdreleased* events detected by the *JMap.Html5Core.Control.TouchClickControl* instance.

JMap.Html5Core.Tool.Tool: Superclass from which all tools must inherit. Your *click* and *holdreleased* event listeners will be defined as part of your Tool subclass or instance.

The following code creates a tool:

```
var sampleTool = new JMap.Html5Core.Tool.Tool({
  name: 'Sample Tool',
  clickHandler: function(event) {
    var coord = JMap.app.map.getLonLatFromPixel(event.xy);
    alert('Clicked at lon: ' + coord.lon + ', lat: ' + coord.lat);
  },
  holdReleasedHandler: function(event) {
    alert('`holdreleased` event performed.');
  },
});
```

After it has been instantiated, you need to register it against the `JMap.app.clickToolManager` and activate it.

```
JMap.app.clickToolManager.addTool(sampleTool);
JMap.app.clickToolManager.setCurrentTool(sampleTool);
```

If a tool was already activated at the time that *setCurrentTool()* was called, that tool will be deactivated.

Working With Vector Layers

OpenLayers offers multiple classes allowing the visualization of various data sources. The *OpenLayers.Layer.Vector* class is particularly useful to display vector data features created programmatically.

The following code excerpt creates and adds a vector layer to an initialized application. Using your own custom tool, you will be able to create Point geometries wherever *click* events are performed.

```
var vectorLayer = new OpenLayers.Layer.Vector('Vector Layer Tool Example', {});
JMap.app.map.addLayer(vectorLayer);

var vectorLayerTool = new JMap.Html5Core.Tool.Tool({
  name: 'Vector Layer Tool',
  clickHandler: function(event) {
    var coord = JMap.app.map.getLonLatFromPixel(event.xy),
        point = new OpenLayers.Geometry.Point(coord.lon, coord.lat),
        feature = new OpenLayers.Feature.Vector(point);

    vectorLayer.addFeatures([feature]);
  },
  holdReleasedHandler: function(event) {
    if (typeof vectorLayer === 'object' && vectorLayer instanceof OpenLayers.Layer.Vector
      vectorLayer.removeAllFeatures();
  },
});
```

Similarly, as described in the previous example, the following code will register your newly created tool against the *JMap.app.clickToolManager* and activate it.

```
JMap.app.clickToolManager.addTool(vectorLayerTool);
JMap.app.clickToolManager.setCurrentTool(vectorLayerTool);
```

Editing Vector Data

Although vector data editing in JMap Web is now offered as part of JMap 6.5, the JMap Web public API does not offer methods to do so programmatically. However, you can use OpenLayers's API to create, edit and delete your own features associated to your own vector layers that were created programmatically.

Here are a few links detailing OpenLayers's vector data editing functions:

- Drawing Simple Vector Features Example
- OpenLayers Draw Feature Example
- Drag Feature Example
- OpenLayers Select Feature Example
- OpenLayers Modify Feature Example
- Feature Events Example

Working With Vector Styles

It is possible to define the appearance of your vector layer features. This is generally accomplished by associating a OpenLayers.StyleMap to your layer.

A StyleMap may contain any of the properties described here in order to modify the appearance of layer features.

- Feature Styles Example

Recommendations

When in a production environment, it is recomended that your extension's assets be minified in order to limit the number of HTTP requests on load. It also allows you to obfuscate your code up to a certain extent. We use Google's Closure Compiler and recommend it.

As previously mentioned, if your extension has dependencies, they will need to be loaded programmatically in your extension's generated javascript file.

OpenLayers's API documentation is generated by parsing the source code's comments. This means that methods and properties may not appear in the docs if they are not properly commented. Directly browsing OpenLayer's source code may help you if something in their documentation is not clear.

Extension Development Workflow

Developers may want to operate differently when developing JMap Web extensions. The two following workflows tend to be preferred.

1. The developer works in the generated extension's folder. They usualy modify the extension's assets, copy the extension files to the $JMAP_HOME$/extensions/web directory, update their applications and test their changes in the updated application.
2. The developer works directly in the deployed application's files.

Both approaches have their own pros and cons.

The first technique, while slower, allows for a safer and more incremental process. Keeping track of code changes is easier.

The second technique is faster, however you will need to consistently merge your changes to your version controlled checkout. This approach may also lead to data loss if you ever update your application and your changes were not included in your $JMAP_HOME$/extensions/web directory.

In addition to executing code in the user's browser, a JMap Web extension may define **actions** that can leverage JMap Server's API. These actions will be registered against your deployment's AJAX Dispatcher service and may be triggered via HTTP requests.

A JMap Web extension can contain several actions. You also may create additional non-action classes that will be referenced within your actions.

Creating an Action

Create a class that inherits from *AbstractHttpRequestAction* . To follow along this example, the name of your class should be *MyFirstAction* .

```
package myextension;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import jmap.http.ajax.servlets.AbstractHttpRequestAction;


public class MyFirstAction extends AbstractHttpRequestAction
{
  @Override
  public void execute(HttpServletRequest request, HttpServletResponse response) throws IOEx
  {

  }
}
```

As you may have noticed, the action requires the *HttpServletRequest* and *HttpServletResponse* classes in order to compile. Add Tomcat server's *servlet-api* JAR to your build path. That particular JAR file may be found within the *$$JMAP_HOME$$/tomcat/lib* directory.

An action requires an `execute` method. That method will be called whenever an HTTP request specifically requesting your action is received by the AJAX Dispatcher.

For the purposes of this example, the following is a typical *Hello World!* action.

```
package myextension;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import jmap.http.ajax.servlets.AbstractHttpRequestAction;


public class MyFirstAction extends AbstractHttpRequestAction
{
  @Override
  public void execute(HttpServletRequest request, HttpServletResponse response) throws IOEx
  {
    final PrintWriter writer = response.getWriter();
    String name = request.getParameter("name");

    if (name != null && name.trim() != "")
      writer.print("Hello, " + name);
    else
      throw new IllegalArgumentException("Invalid argument. Must specify a `name` parameter
  }
}
```

Producing a JAR File for Your Action

In order to be included as part of a JMap Web extension, your server-side Java code must be compiled and packaged as a single JAR file.

Using Ant

When using JMap 6.0 SDK's webextensionbuilder tool (described in the "Programming JMap Web Extensions" section) to produce an extension's boilerplate code, a *SampleAction* is provided as a starting point. A build.xml file is also provided. You can use that file with Ant to compile and produce your extension's JAR file.

Copy your MyFirstAction.java file to your extension's *actions/src* directory. Since the MyFirstAction class is defined within the *myextension* package, create a *myextension* directory under *actions/src* and put your MyFirstAction.java file in it. At this point, you should have the following items in your actions directory:

```
actions/
+-- build.xml
+-- readme.markdown
+-- src
    +-- Example
    ¦   +-- SampleAction.java
    +-- myextension
        +-- MyFirstAction.java

3 directories, 4 files
```

Using the command line/terminal navigate to your extension's *actions* directory and execute the following command.

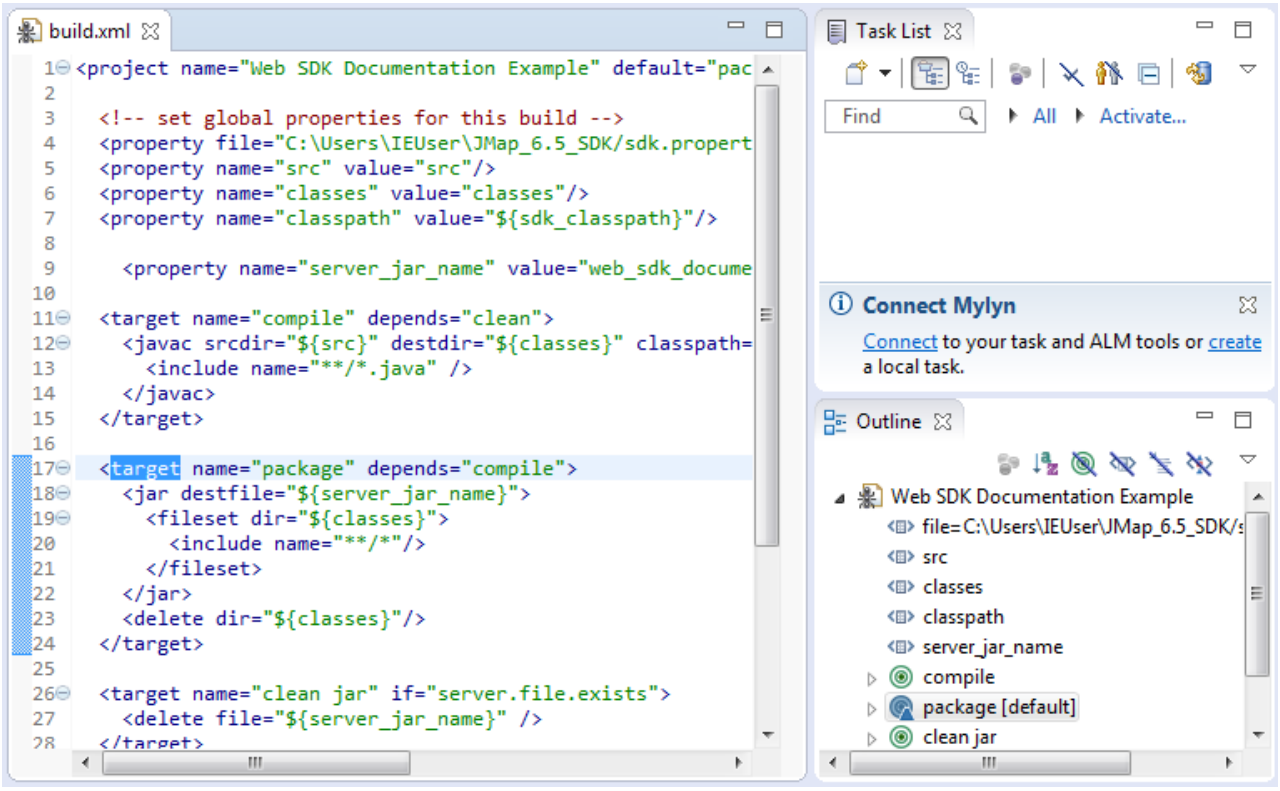
```
ant -f build.xml
```

Once you have your JAR file, make sure it's in your web extension's *actions* directory.

Using Ant through Eclipse

As was the case when using the webextensionbuilder tool, you may use Ant through Eclipse to produce your extension's server jar file. This requires that a JDK is set as Eclipse's default JRE and that the JDK's tools.jar file is added to the Ant runtime configuration's classpath. These steps were previously detailed here.

1. Open the your extension's actions/build.xml file in Eclipse.
2. Execute the "package" ant target.



Execute the "package" ant target.

Once you have your JAR file, make sure it's in your web extension's *actions* directory.

Identifying Your Extension's Actions

As part of your extension, the *extension.json* file informs JMap Server about itself. You must now indicate that the extension contains actions. Open that file in a text editor.

The *actions* property is an array that can contain several object literals each describing individual actions.

The *actions* array's object literals **must** have the following property keys:

Action property keys	**Description**
name	{String} This is the name of the action as it will be registered against the AJAX Dispatcher. Must be unique within a single JMap Web deployment. HTTP requests must specify this value for the *action* request parameter. Does not need to correspond to your action's class's source file name.
classname	{String} The fully qualified name of your class. Must include packages.
version	{String} Specifies a version. Mostly useful for debugging purposes.

This snippet is how you would represent the *MyFirstAction* action as part of the *web_sdk_documentation_example* extension that was created earlier:

```
{
    "actions": [
        {
            "name": "hello",
            "className": "myextension.MyFirstAction",
            "version": "1.0.0"
        }
    ],
    "fullname": "Web SDK Documentation Example",
    "namespace": "Example",
    "shortname": "web_sdk_documentation_example",
    "version": "1.0"
}
```

Your extension may include as many actions as you want. Just be sure to include them within your extension's *extension.json* file.

After editing the *extension.json* file, be sure that it contains valid JSON. JSONLint is an online JSON validator that can be used for that purpose.

Calling your action from your JMap Web Extension

As previously mentioned, your action's *execute* method will be called once the AJAX Dispatcher receives an HTTP request that specifies your action's name as a parameter.

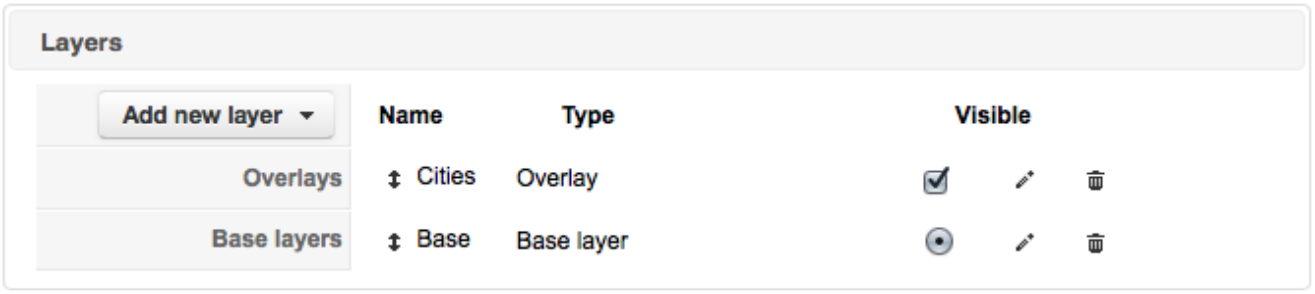
The following example demonstrates how you can submit an HTTP Request to your action using jQuery.ajax(). To test this, you can either include this snippet in your extension's *init* function or copy and paste it in a browser's Javascript console currently logging your deployed application.

```
$.ajax(JMap.app.ajaxDispatcher, {
  data: {
    "action": "hello", // The action name as defined in the extension.json file.
    "name": "Developer" // The `name` parameter as expected by the action.
  }
}).error(function(jqXHR, textStatus, errorThrown) {
  alert(textStatus);
}).success(function(data, textStatus, jqXHR) {
  alert(data);
});
```

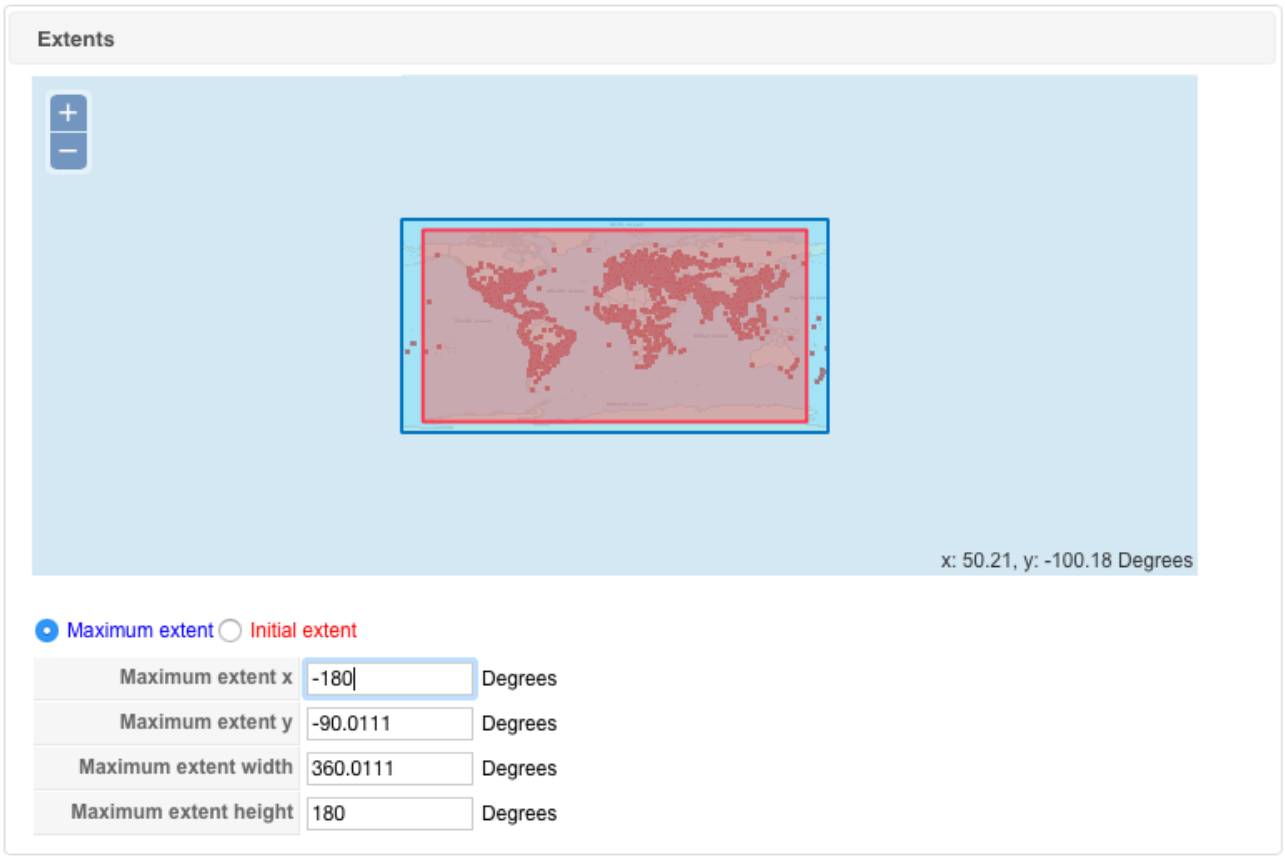
The JMap Web application template offers many actions to facilitate JMap Server interactions. This section will present some of them and how you can use them.

Note: The majority of actions require map state information as parameters. For the most part, these can be obtained using OpenLayers's API.

All examples in this section refer to a deployment based on the **The World** project. The deployment covers the project's complete extent and includes the following layers: "Base" and "Cities". Refer yourself to the following screen shots in order to create a similar deployment.

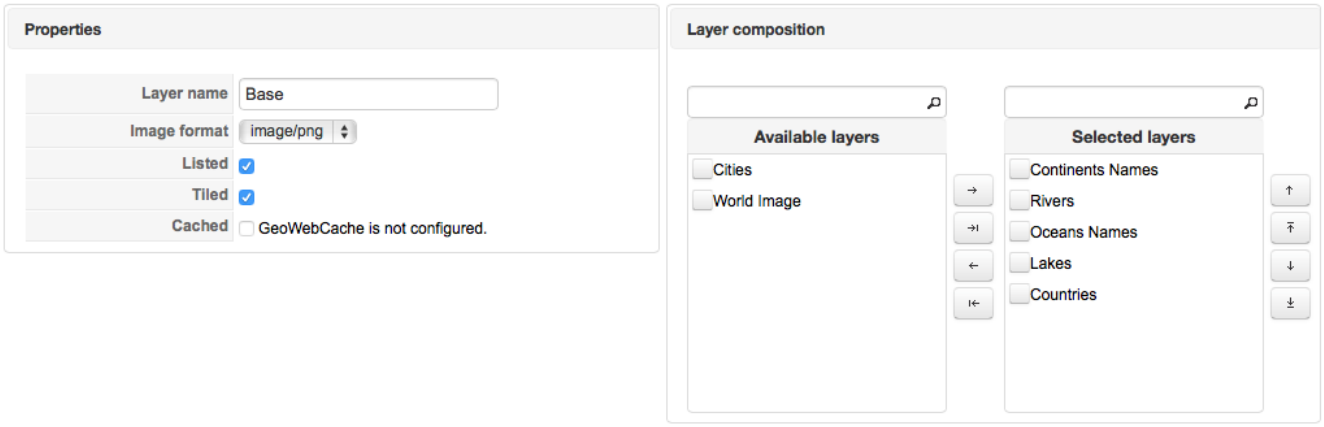


Configured layers

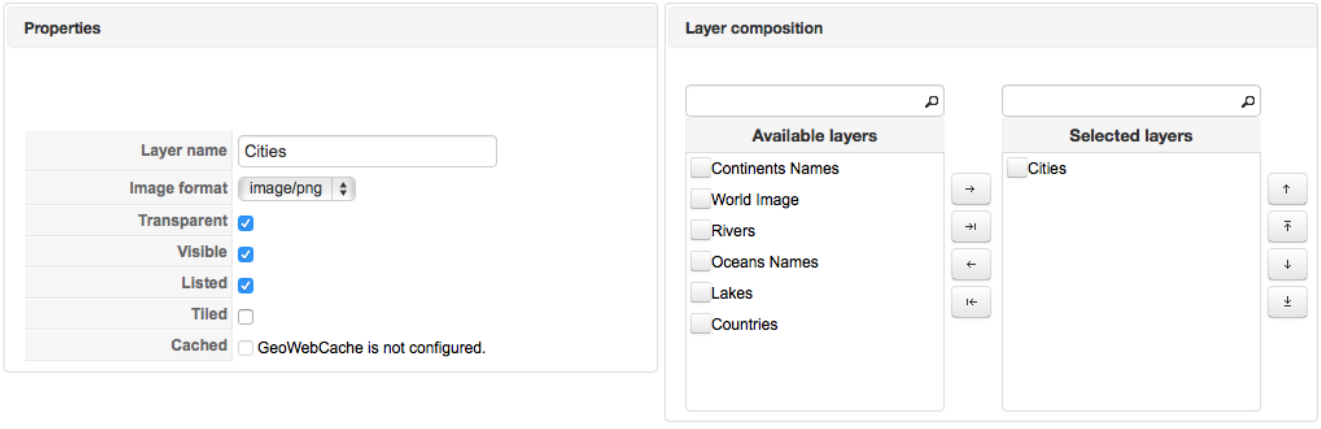


Geographic properties, part 1

Geographic properties, part 2



"Base" layer configuration



"Cities" layer configuration.

'extractelements' Action

The *extractelements* action provides a list of a JMap layer's elements.

the 'geometry' request type

The *geometry* request will extract elements that intersect or are contained within a supplied geometry.

Parameters

The following is the list of parameters that can be provided to the *extractelements* action.

extractelements parameters	Description
request	{String} **"geometry" required** This string identifies the type of request that's being issued to the action. Other types of requests may be added to this action in the future.
geometry	{String} **required** A WKT geometry string defining the region for which to execute the element extraction process.
res	{Number} **required** The map's current resolution.
bbox	{String} **required** The map's current extent as it's displayed in the browser at the time of the request. Specified in a bounding box format (example: `"-100,-50,100,50"`).
drill	{Boolean} default: **false** Indicates whether or not you wish to continue analyzing the layer stack once the first match is found.
layers	{String} **required** Specifies for which layer(s) you want to perform the element extraction. Many layers may be requested at once. In that case, you should separate the elements of your layer list using commas. Use the drill parameter accordingly in order to obtain elements across multiple layers.

Example

Performs an element extraction request for the "Cities" layer around the Scotland area.

```
$.ajax(JMap.app.ajaxDispatcher, {data: {
  'action': 'extractelements',
  'request': 'geometry',
  'geometry': 'POLYGON((-6.378882 54.903806,-1.336158 54.903806,-1.336158 59.540037,-6.3788
  'res': JMap.app.map.getResolution(),
  'bbox': JMap.app.map.getExtent().toBBOX(),
  'layers': 'Cities'
}}).success(function(data, textStatus, jqXHR) {
  console.log(data);
});
```

Response

Each element's ID, geometry (supplied as a WKT formatted string), attribute values, bounds and centered point are returned. The layer's bound attributes are also included as well as their SQL type. An element's attributes's index corresponds to the layer's attributes as configured in JMap Admin. If elements from many layers match the request parameters, an array of these object literals will be returned.

```
{
    "layerId": 7,
    "elementAttributes": [
        {
            "sqlType": 12,
            "name": "CITY"
        },
        {
            "sqlType": 12,
            "name": "COUNTRY"
        },
        {
            "sqlType": 12,
            "name": "CAP"
        },
        {
            "sqlType": 4,
            "name": "POP2000"
        }
    ],
    "elements": [
        {
            "centeredPoint": {
                "x": -1.3899999735879476,
                "y": 54.910001831054686
            },
            "bounds": {
                "x": -1.3899999735879476,
                "width": 0,
                "y": 54.910001831054686,
                "height": 0
            },
            "attributeValues": [
                "Sunderland",
                "UK - England and Wales",
                "0",
                181100
            ],
            "geometry": "POINT(-1.3899999735879476 54.910001831054686)",
            "id": 622
        },
        {
            "centeredPoint": {
                "x": -4.2699998496102864,
                "y": 55.87000091552734
            },
            "bounds": {
                "x": -4.2699998496102864,
                "width": 0,
                "y": 55.87000091552734,
                "height": 0
            },
            "attributeValues": [
                "Glasgow",
                "Scotland",
                "0",
                610700
            ],
            "geometry": "POINT(-4.2699998496102864 55.87000091552734)",
            "id": 624
```

```
},
{
    "centeredPoint": {
        "x": -3.2200001357125814,
        "y": 55.949998931884764
    },
    "bounds": {
        "x": -3.2200001357125814,
        "width": 0,
        "y": 55.949998931884764,
        "height": 0
    },
    "attributeValues": [
        "Edinburgh",
        "UK - England and Wales",
        "0",
        382600
    ],
    "geometry": "POINT(-3.2200001357125814 55.949998931884764)",
    "id": 625
},
{
    "centeredPoint": {
        "x": -2.9999998686837728,
        "y": 56.469999389648436
    },
    "bounds": {
        "x": -2.9999998686837728,
        "width": 0,
        "y": 56.469999389648436,
        "height": 0
    },
    "attributeValues": [
        "Dundee",
        "Scotland",
        "0",
        148900
    ],
    "geometry": "POINT(-2.9999998686837728 56.469999389648436)",
    "id": 627
},
{
    "centeredPoint": {
        "x": -2.1000000117349202,
        "y": 57.14999969482422
    },
    "bounds": {
        "x": -2.1000000117349202,
        "width": 0,
        "y": 57.14999969482422,
        "height": 0
    },
    "attributeValues": [
        "Aberdeen",
        "Scotland",
        "0",
        188500
    ],
    "geometry": "POINT(-2.1000000117349202 57.14999969482422)",
    "id": 628
```

```
        },
        {
            "centeredPoint": {
                "x": -1.6000000117349202,
                "y": 54.99999816894531
            },
            "bounds": {
                "x": -1.6000000117349202,
                "width": 0,
                "y": 54.99999816894531,
                "height": 0
            },
            "attributeValues": [
                "Newcastle upon Tyne",
                "UK - England and Wales",
                "0",
                980000
            ],
            "geometry": "POINT(-1.6000000117349202 54.99999816894531)",
            "id": 1899
        }
    ]
}
```


'layerinfo' Action

The *layerinfo* action provides details regarding layers of the deployment's associated project.

the 'geteditablelayers' request type

The *geteditablelayers* request will return layer various layer details on which the currently authenticated user may perform data editing tasks.

Parameters

This action does not require any parameters. Instead, it uses the current session information to perform this operation.

Example

```
$.ajax(JMap.app.ajaxDispatcher, {data: {
  'action': 'layerinfo',
  'request': 'geteditablelayers'
}}).success(function(data, textStatus, jqXHR) {
  console.log(data);
});
```

Response

The server response exposes an array of *editableLayers* object literals. Each of the array's objects contains the following keys:

geteditablel ayers	Description

response property keys	
fields	{Array} Array of object literals exposing the name bound layer fields as well as their SQL data type.
forms	{Array} The layer's configured forms.
id	{Number} The layer's ID.
idFieldName	{String} The name of the layer attribute that serves as an identifier.
offset	{Object} An object literal that contains the layer's current offset.
permissions	{Number} An integer between 0 and 15 (inclusive) that describes the current user's permissions towards the layer. The following bit masks signify different permissions. • 0x0000: None. • 0x0001: May add layer elements. • 0x0010: May edit layer elements. • 0x0100: May delete layer elements. • 0x1000: May edit layer elements's attributes.

forms are returned as an array of object literals. They have the following structure:

Response`s forms objects`s property keys	Description
id	{Number} The form's unique ID.
json	{String} A JSON representation of the form's configuration.
name	{String} The form's name as defined in JMap Admin.
permissions	{Number} An integer between 0 and 7 (inclusive) that describes the current user's permissions towards the form. A form's permissions only concerns external forms. Layer attributes forms's permissions are defined as part of the layer permissions. • 0x000: None. • 0x001: May add data in external forms. • 0x010: May edit data in external forms. • 0x100: May delete data from external forms.
type	{String} Identifies the form type. One of the following:

- LAYER_ATTRIBUTES_FORM
- LAYER_ATTRIBUTES_SUB_FORM
- EXTERNAL_ATTRIBUTES_FORM
- EXTERNAL_ATTRIBUTES_SUB_FORM

uidAttributeName {String} Only concerns external forms. Is the name of the layer's attribute that will be used as a foreign key to establish child-parent relationships.

Example

This is an example of a *geteditablelayers* response. You may have different results. In any case, they should reflect the state of your configured permissions and forms.


```
{
    "editableLayers": [
        {
            "offset": {
                "x": 9.5367431640625e-7,
                "y": -3.186320304870577
            },
            "permissions": 0,
            "id": 4,
            "fields": [
                {
                    "name": "COUNTRY",
                    "serverDataType": 12
                },
                {
                    "name": "CONTINENT",
                    "serverDataType": 12
                },
                {
                    "name": "POP_1994",
                    "serverDataType": 8
                },
                {
                    "name": "POP_GRW_RT",
                    "serverDataType": 8
                },
                {
                    "name": "POP_0_14",
                    "serverDataType": 8
                },
                {
                    "name": "POP_15_64",
                    "serverDataType": 8
                },
                {
                    "name": "POP_65PLUS",
                    "serverDataType": 8
                },
                {
                    "name": "POP_AREA",
                    "serverDataType": 8
```

```
            }
        ],
        "idFieldName": "JMAP_ID",
        "forms": []
    },
    {
        "offset": {
            "x": 0.56195068359375,
            "y": 13.687942504882812
        },
        "permissions": 0,
        "id": 6,
        "fields": [
            {
                "name": "LAKE_NAME",
                "serverDataType": 12
            },
            {
                "name": "VOLUME_CKM",
                "serverDataType": 8
            },
            {
                "name": "AREA_SKM",
                "serverDataType": 8
            }
        ],
        "idFieldName": "JMAP_ID",
        "forms": []
    },
    {
        "offset": {
            "x": 20.934576233500025,
            "y": 10.050437668683657
        },
        "permissions": 0,
        "id": 5,
        "fields": [],
        "idFieldName": "JMAP_ID",
        "forms": []
    },
    {
        "offset": {
            "x": 16.215000694478334,
            "y": 16.463705277985326
        },
        "permissions": 0,
        "id": 3,
        "fields": [
            {
                "name": "HYD_NAME",
                "serverDataType": 12
            },
            {
                "name": "LENGTH_KM",
                "serverDataType": 4
            }
        ],
        "idFieldName": "JMAP_ID",
        "forms": []
    },
```

```
        {
            "offset": {
                "x": 31.452202349999993,
                "y": -14.421794805000005
            },
            "permissions": 0,
            "id": 2,
            "fields": [
                {
                    "name": "CONTINENTNAME",
                    "serverDataType": 12
                }
            ],
            "idFieldName": "JMAP_ID",
            "forms": []
        },
        {
            "offset": {
                "x": 1.6169597031546061,
                "y": 8.079999999999998
            },
            "permissions": 15,
            "id": 7,
            "fields": [
                {
                    "name": "CITY",
                    "serverDataType": 12
                },
                {
                    "name": "COUNTRY",
                    "serverDataType": 12
                },
                {
                    "name": "CAP",
                    "serverDataType": 12
                },
                {
                    "name": "POP2000",
                    "serverDataType": 4
                }
            ],
            "idFieldName": "JMAP_ID",
            "forms": [
                {
                    "permissions": 0,
                    "name": "Form",
                    "json": "{\"formSections\":[{\"name\":\"Section 1\",\"nbRows\":3,\"fiel
                    "id": 1,
                    "type": "LAYER_ATTRIBUTES_FORM",
                    "uidAttributeName": null
                }
            ]
        }
    ]
}
```

'loadformdata' Action

The *loadformdata* action provides **external** form data for a requested layer element.

Parameters

loadformdata expects the following parameter:

loadformdata parameters	Description
data	{String} **required** JSON String that represents a Javascript object literal composed of the following properties: • elementId: {Number} The element (for which data is requested)'s unique ID. • formId: {Number} The external form's ID. • layerId: {Number} The layer (on which the element exists/the form is configured)'s ID. • listFields: {Array[Strings]} A list of form fields for which you want values. An empty array will return data for all fields. • mapValues: Javascript object literal that contains the element's layer's layer attributes form's fields values.

Example

This example refers to another project for which external forms were configured.

```
var data = {
  elementId: 6,
  formId: 2,
  layerId: 1,
  listFields: [],
  "mapValues": {
    "MOBILE_JMAP_ID": 6,
    "MOBILE_JMAP_GEOMETRY": "POINT(-8189010.0 5701129.5)",
    "AUTHOR": "jrhaddad",
    "CREATION_TIME": 1415767972000,
    "MODIFICATION_TIME": 1418400517000,
    "ABR_CODE": "342",
    "ABR_NAME": "ES34F",
    "ABR_LOC_TYPE": "Arrêt bus",
    "ABR_WHEEL_CHAIR": "Non accessible",
    "ABR_DATE_INSP": null,
    "ABR_STATUS": null
  }
};

$.ajax(JMap.app.ajaxDispatcher, {data: {
  'action': 'loadformdata',
  'data': JSON.stringify(data)
}}).success(function(data, textStatus, jqXHR) {
  console.log(data);
});
```

Response

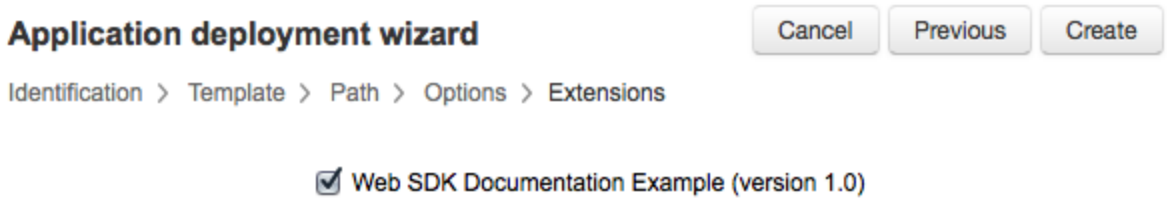
The server responds by providing a two dimensional `rows` array of Javascript object literals representing field values. Each item in the `rows` array represents a row of data. Multiple rows are returned when obtaining data that was entered using a *EXTERNAL_ATTRIBUTES_SUB_FORM* type form.

The name of the form field, its value and the data's SQL type are returned.

```
{
    "rows": [
        [
            {
                "name": "insp_abribus.id_inspection",
                "value": 5,
                "type": 4
            },
            {
                "name": "insp_abribus.id_abribus",
                "value": 6,
                "type": 4
            },
            {
                "name": "insp_abribus.date_inspection",
                "value": 1415854800000,
                "type": 93
            },
            {
                "name": "insp_abribus.etat",
                "value": "Bon état",
                "type": 12
            },
            {
                "name": "insp_abribus.observations",
                "value": "Tout est ok",
                "type": 12
            }
        ]
    ]
}
```


In order to deploy an extension with your JMap Web deployment, you must first install the extension on your JMap Server. You install a JMap Web extension by copying its parent directory (named after the extension's shortname) to the *$JMAP_HOME$/extensions/web* directory.

As part of the JMap Web application deployment process, you should now see a list of Web extensions you may wish to include. Choose your newly created "Web SDK Documentation Example" and complete the wizard.



Deploying the example extension

JMap Web's design allows the embedding of JMap Web deployments into your own web applications. This section details the necessary steps to enable application embedding.

Copying the JMap Web Libraries and Dependencies to Your Web Server

Your deployed JMap Web applications, as they are now being served by JMap's embedded Tomcat instance, contain the libraries and web services required for application embedding. Your application's files are located in the */applications/deployed* directory of your JMap Server.

Among those files is the *jmap* directory. It contains all necessary resources required to embed your JMap deployment. **This directory and its contents must be copied to your web server and be accessible by your web application's pages.**

JMap Web's dependencies will be loaded as part of the initialization process prior to displaying the map. For more details regarding the initialization process refer to the JMap Web's Initialization Process section of this document.

- JQuery is required during the map initialization process and as such must be present in your application's document.
- Including JMap Web's dependencies's stylesheets will alter the styles of your document. Currently, there is unfortunately no way to avoid this.

Authorizing Cross-Origin Resource Sharing

In order to allow JMap Web Deployments to be embedded, a few tweaks to your deployment's web.xml files are necessary.

- Modifying the deployment's *web.xml* file will require you to unload/load your deployment. This can be done in JMap Admin's deployment section.
- All modifications to your deployment's files will be loss if you update the deployment or its template's files.

1. Enable HTTP Authentication for the *JMapLoginFilter_index* filter.

```
<filter>
   <filter-name>JMapLoginFilter_index</filter-name>
   <!-- ... -->
   <init-param>
     <param-name>httpauthentication</param-name>
     <param-value>true</param-value>
   </init-param>
 </filter>
```

2. Instanciate and map the CORS filter. Add the following declarations before the first *ByPassFilter* mapping.

```
<filter>
  <filter-name>CorsFilter</filter-name>
  <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
  <init-param>
    <param-name>cors.allowed.headers</param-name>
    <param-value>Content-Type,X-Requested-With,accept,Origin,Access-Control-Request-Method,
  </init-param>
  <init-param>
    <param-name>cors.exposed.headers</param-name>
    <param-value>Access-Control-Allow-Origin,Access-Control-Allow-Credentials</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CorsFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Embedding a map

Once the *jmap* directory is copied to your web server, you will need to add the following script references in the `<head>` portion of the document where you wish to embed your deployment:

```
<script type="text/JavaScript" src="jmap/vendor/jquery-1.9.1.min.js"></script>
<script type="text/JavaScript" src="jmap/jmap.min.js"></script>
```

- You might need to adjust the relative paths mentioned above.
- Including JQuery is not necessary if you already use it. JMap Web requires a minimum version of 1.9.1 to assure most functionalities.

To initialize the map, you will need to provide an empty div into which the map will be created.

```
<div id="map" style="height: 600px; width: 600px;"></div>
```

Tip: Don't see the map? Make sure your *div* is properly sized.

Include the following Javascript to trigger map initialization.

```
<script type="text/JavaScript">
  $(document).ready(function() {
    var options = {
      // The URL of your deployed application. This must be an address that your users may
      jmapUrl: 'http://192.168.0.80:8080/web_deployment'
    };

    JMap.initialize(document.getElementById('map'), options);
  });
</script>
```

The *JMap.initialize* function that is called above is the same as the one that was detailed previously here. You may customize the embedded application by specifying those same configuration properties in a *mapConfig* object literal supplied in the *options* argument.

embed_example.html

Included in your deployed application's directory is an *embed_example.html* file. It serves as an example of a JMap Web embedded application. Open that file in a text editor to see how application embedding is done.

You can also copy that file **and the jmap directory** to a seperate web server in order to confirm that Cross-Origin Resource Sharing was properly enabled.

Dealing with Authentication

If you enabled controlled access on your JMap Web deployment, you will need to authenticate as a JMap User prior to initializing the map.

The following is provided as an example and IS NOT a recommended way of authenticating yourself to JMap Server:

```
<script type="text/JavaScript">
    $(document).ready(function() {
      $.ajaxSetup({
        xhrFields: {
          withCredentials: true
        }
      });

      $.ajax('http://192.168.0.80:8080/web_deployment/ajaxdispatch', {
        data: {
          action: 'ping'
        },
        beforeSend: function(xhr) {
          // Replace 'USERNAME:PASSWORD' with a valid string value.
          xhr.setRequestHeader('Authorization', 'Basic ' + btoa('USERNAME:PASSWORD'));
        },
      }).done(function(data, textStatus, jqXHR) {
        var options = {
          jmapUrl: 'http://192.168.0.80:8080/web_deployment'
          mapConfig: {},
          onMapInit: function() {
            console.log('Map was initialized.');
          }
        };

        JMap.initialize(document.getElementById('map'), options);
      });
    });
</script>
```

To Contact Us

By phone

You can contact us during business hours (8:30 AM - 4:30 PM, Monday through Friday) at +1 514.285.1211.

On the web

You can visit our web site for more information about our products or for technical support at k2geospatial.com.

By email

Technical support: support@k2geospatial.com

Sales: sales@k2geospatial.com

Our street address is

K2 Geospatial
740 Notre-Dame Street West, suite 1260
Montréal, Quebec (Canada) H3C 3X6