

BlomURBEX™

Javascript API

V 5.7.1

Blom

Audience: **BLOM partners and developers**

Abstract: This document describes **BlomURBEX™ Javascript API** for accessing the BlomURBEX™ services

Date	Blom Document
January 2017	BlomURBEX Javascript API v5.7.1 r1.0.pdf

Revision History

Document Number	Issue Date	Reason for Change
BUAPI_J_1010	July 2008	Original Document. Ortho and Oblique images view methods. Basic API defined.
BUAPI_J_2010	September 2008	Measurement functions added
BUAPI_J_2110	November 2008	Cross-domain support added
BUAPI_J_2120	February 2009	New template used and minor changes introduced
BUAPI_J_3010	March 2009	New release. See Chapter 17 for the Release Notes.
BUAPI_J_3011	April 2009	Additional options for text creation functions
BUAPI_J_3012	July 2009	Note for IE8 developers. More information about Markers from Text files
BUAPI_J_4010	August 2009	New controls for Multi-Oblique and Multi-Year Data Added more control over measurements
BUAPI_J_4510	September 2010	New geocoding capabilities, access to WMS services, new controls
BUAPI_J_4520	January 2011	New cursor configuration for markers.
BUAPI_J_4530	January 2011	Diagonal measurement and control added.
BUAPI_J_5010	April 2011	New Urbex.Util.GetUserAvailableData year capability
BUAPI_J_5020	September 2011	Vertical Area measurement and control added
BUAPI_J_5030	November 2011	Full review of documentation and all 5.0 features added
BUAPI_J_5040	November 2011	Minor changes and review of release notes
BUAPI_J_5050	December 2011	Type correction
BUAPI_J_5510	June 2012	New functionality for API 5.5
BUAPI_J_5520	September 2013	New functionality for API 5.6 (Mobile events)
BUAPI_J_5530	November 2013	New functionality for API 5.7
BUAPI_J_5531	November 2015	Fixed path to API 5.7 javascript file
BUAPI_J_5532	January 2016	Added HTTPS support
BUAPI_J_5533	January 2017	Tiles server configured as domain dependent

Table of Contents

BlomURBEX™ Javascript API.....	i
1 Scope	1
1.1 Compatibility	1
2 Map Creation	2
2.1 Accessing the API.....	2
2.2 Creating a Map	2
2.3 Map Creation Parameters	2
2.4 Example: Minimum Application: The BlomURBEX "Hello, World"	7
2.5 Internet Explorer 8 development.....	7
2.5.1 API Version 4.5 and onwards	7
2.5.2 API versions older than 4.5.....	8
3 Basic Classes	9
3.1 <i>Urbex.XY</i>	9
3.1.1 Description	9
3.1.2 Construction	9
3.1.3 Properties.....	9
3.1.4 Example:.....	9
3.2 <i>Urbex.Size</i>	10
3.2.1 Description	10
3.2.2 Construction	10
3.2.3 Properties.....	10
3.2.4 Example	10
3.3 <i>Urbex.Pixel</i>	11
3.3.1 Description	11
3.3.2 Construction	11
3.3.3 Properties.....	11
3.3.4 Example	11
3.4 <i>Urbex.Bounds</i>	12
3.4.1 Description	12
3.4.2 Construction	12
3.4.3 Properties.....	12

3.4.4	Example	12
4	Basic Map Settings	13
4.1	<i>setWorldCenter method</i>	<i>13</i>
4.1.1	Setting a new Centre	13
4.1.2	Setting the Centre and zoom simultaneously.....	13
4.1.3	Change Zoom	13
4.1.4	Change Maximum Digital Zoom	13
4.2	<i>Pan method</i>	<i>15</i>
4.2.1	Parameters	15
4.2.2	Example	15
4.3	<i>Intelligent Map Behaviour.....</i>	<i>16</i>
4.4	<i>Auto rotate tiles Map Behaviour</i>	<i>16</i>
4.5	<i>Force always Ortho in mosaic mode Map Behaviour</i>	<i>16</i>
	<code>forceAlwaysOrthoInMosaicView:false</code> Limiting the zoom	18
4.5.1	Parameters.....	19
4.5.2	Zoom limits and intelligent behaviour	19
4.5.3	Example	19
4.6	<i>Changing Map Image Display.....</i>	<i>20</i>
4.7	<i>Changing map's tiles scale.....</i>	<i>22</i>
5	Changing View mode, Orientation and Projection.....	24
5.1	<i>Change view type</i>	<i>24</i>
5.1.1	Examples.....	24
5.2	<i>Change orientation in ortho view</i>	<i>25</i>
5.2.1	Examples.....	25
5.3	<i>Change orientation in oblique mode.....</i>	<i>26</i>
5.3.1	Example	26
5.4	<i>Change projection</i>	<i>27</i>
5.4.1	Example	27
6	Retrieving map information.....	28
6.1	Centre	28
6.2	Zoom level	29
6.3	Digital Zoom.....	30
6.4	Extent.....	30

6.5	<i>View Type and Orientation</i>	30
6.5.1	<i>View Type</i>	30
6.5.2	<i>Orientation</i>	31
6.5.3	<i>Example</i>	31
6.6	<i>View Projection</i>	32
6.7	<i>Retrieving the layers</i>	32
7	<i>Managing Controls</i>	33
7.1	<i>Available controls</i>	33
7.2	<i>Adding controls to the map</i>	37
7.3	<i>Removing non-default controls on the map</i>	38
7.4	<i>Exploring the controls collection</i>	38
7.5	<i>Customizing controls</i>	40
7.5.1	<i>Urbex.Control.Navigation</i>	40
7.5.2	<i>Urbex.Control.ZoomIn</i>	40
7.5.3	<i>Urbex.Control.ZoomOut</i>	40
7.5.4	<i>Urbex.Control.ZoomBar</i>	41
7.5.5	<i>Urbex.Control.ChangeView</i>	41
7.5.6	<i>Urbex.Control.ChangeView2</i>	42
7.5.7	<i>Urbex.Control.ChangeOrientation</i>	42
7.5.8	<i>Urbex.Control.LayerSwitcher</i>	43
7.5.9	<i>Urbex.Control.ReturnPoint</i>	43
7.5.10	<i>Urbex.Control.GetMap</i>	44
7.5.11	<i>Urbex.Control.Length</i>	44
7.5.12	<i>Urbex.Control.Area</i>	44
7.5.13	<i>Urbex.Control.Height</i>	45
7.5.14	<i>Urbex.Control.Bearing</i>	45
7.5.15	<i>Urbex.Control.Elevation</i>	45
7.5.16	<i>Urbex.Control.Eraser</i>	46
7.5.17	<i>Urbex.Control.North</i>	46
7.5.18	<i>Urbex.Control.Scale</i>	46
7.5.19	<i>Urbex.Control.Metadata</i>	47
7.5.20	<i>Urbex.Control.ShowStreets</i>	47
7.5.21	<i>Urbex.Control.MultiOblique</i>	47
7.5.22	<i>Urbex.Control.Year</i>	48
7.5.23	<i>Urbex.Control.ReverseGeocoding</i>	48
7.5.24	<i>Urbex.Control.Baselayer</i>	48

BlomURBEX™ Javascript API v5.7 r1.0

7.5.25	Urbex.Control.Overlays	49
7.5.26	Urbex.Control.Diagonal	49
7.5.27	Urbex.Control.VerticalArea.....	50
7.5.28	Urbex.Control.LayerSelector	50
7.5.29	ZoomBar and ZoomOut controls	51
7.6	<i>Adding custom controls to the map.....</i>	51
7.6.1	Adding a custom control with the Urbex.Util.extend() method	51
7.6.2	Adding custom controls inheriting from Urbex.Control class	52
7.6.3	Developing custom controls	53
8	Markers	62
8.1	<i>Adding Markers</i>	62
8.1.1	Using Marker layers	62
8.1.2	Using TextMarkers layers	67
8.1.3	Icons	68
8.2	<i>Deleting Markers</i>	68
8.2.1	removeMarker.....	69
8.2.2	removeLayer.....	69
9	Vectors	70
9.1	<i>Construction</i>	70
9.2	<i>Methods</i>	70
9.3	<i>Features.....</i>	70
9.4	<i>Geometries.....</i>	71
9.4.1	Geometry Point	71
9.4.2	Geometry LineString	72
9.4.3	Geometry LinearRing	72
9.4.4	Geometry Polygon	73
9.5	<i>Creation of Vector Features from WKT</i>	74
9.6	<i>Styles</i>	75
9.6.1	Default style	75
9.6.2	Creating new styles	76
9.7	<i>Editable Vector Layers</i>	77
9.7.1	Concept.....	77
9.7.2	featuremodified event	77
10	Text data	78

10.1	<i>Adding Texts</i>	78
10.2	<i>Deleting Texts</i>	81
11	Annotations	82
11.1	<i>Annotations Object</i>	82
11.1.1	Annotations Object Reference	82
12	Geocoding and reverse geocoding	84
12.1	<i>Urbex.Geocoder</i>	84
12.1.1	Geocoding Providers	84
12.1.2	Creating a Geocoder Standalone object	86
12.1.3	Using map's geocoder	86
12.1.4	Geocoding	86
12.1.5	Reverse geocoding.....	88
13	Routing	90
13.1	<i>Urbex.Router</i>	90
13.1.1	Routing Providers	90
13.1.2	Creating a Router Standalone object	91
13.1.3	Using map's router	91
13.1.4	Routing	91
13.1.5	Displaying routes in the map.....	95
13.1.6	Clearing routes from the map.....	95
14	WMS layers	96
15	WFS layers	98
15.1	<i>Adding WFS Layers to the map</i>	98
15.1.1	WFS Layer useful methods and properties	99
16	POIs	100
16.1	<i>POI Concept</i>	100
16.2	<i>Displaying POIs in the Map</i>	100
16.2.1	POI object initialization.....	100
16.2.2	Getting all categories available for a certain provider	101
16.2.3	Displaying POIs of a certain category into the map	103
16.2.4	Complete example of POI loading and displaying sequence	104
17	Coordinates Conversion	106
17.1	<i>WGS84 to Spherical Mercator</i>	106

17.2	<i>UTM to Spherical Mercator.....</i>	<i>106</i>
17.3	<i>WGS84 to UTM</i>	<i>107</i>
17.4	<i>General coordinates conversion.....</i>	<i>107</i>
17.5	<i>Screen coordinates to the map's reference system.</i>	<i>108</i>
17.6	<i>World Coordinates to Screen coordinates</i>	<i>108</i>
17.7	<i>World coordinates to Oblique coordinates on map's oblique image.....</i>	<i>109</i>
17.8	<i>Oblique coordinates to World coordinates on map's oblique image.....</i>	<i>110</i>
18	Event handling	112
18.1	<i>Event construction.....</i>	<i>112</i>
18.2	<i>Methods</i>	<i>112</i>
18.2.1	<i>register</i>	<i>112</i>
18.2.2	<i>unregister.....</i>	<i>113</i>
18.2.3	<i>triggerEvent.....</i>	<i>113</i>
18.3	<i>Not managed events.....</i>	<i>114</i>
18.3.1	<i>observe.....</i>	<i>114</i>
18.3.2	<i>isLeftClick.....</i>	<i>115</i>
18.3.3	<i>stop.....</i>	<i>115</i>
18.4	<i>Event types</i>	<i>115</i>
18.4.1	<i>Map Events.....</i>	<i>115</i>
18.4.2	<i>TextMarkers Events</i>	<i>118</i>
18.4.3	<i>Control Events</i>	<i>119</i>
18.4.4	<i>Layer.Vector Events</i>	<i>119</i>
19	Retrieving user available information.....	121
19.1	<i>Urbex.Util.GetUserAvailableExtents.....</i>	<i>121</i>
19.2	<i>Urbex.Util.GetUserAvailableData</i>	<i>122</i>
19.3	<i>Urbex.Util.GetUserAvailableBaselayers</i>	<i>123</i>
19.4	<i>Urbex.Util.GetUserAvailableOverlays.....</i>	<i>124</i>
19.5	<i>Urbex.Util.GetUserBlomSTREETCredentials</i>	<i>125</i>
19.6	<i>Urbex.Util.GetUserAvailableProviders</i>	<i>125</i>
20	Measurements Functions.....	127
20.1	<i>Length measurements</i>	<i>127</i>
20.2	<i>Area measurements.....</i>	<i>129</i>
20.3	<i>Height measurements.....</i>	<i>131</i>

20.4	<i>Bearing measurements</i>	132
20.5	<i>Elevation measurements</i>	133
20.6	<i>Ground Length measurements</i>	134
20.7	<i>Diagonal measurements.....</i>	135
20.8	<i>Vertical Area measurements</i>	136
20.9	<i>Retrieving the graphical elements associated to a measurement.</i>	138
20.9.1	Popups.....	138
20.9.2	Example	139
20.10	<i>Customizing the graphical elements associated with a measurement:</i>	140
20.10.1	Example	141
20.11	<i>Cleaning measurements displayed on the map</i>	143
20.12	<i>Turn off measurements controls.</i>	143
21	Handlers.....	145
21.1	Constructor	145
21.2	Properties.....	145
21.3	Methods.....	145
21.4	Constants.....	146
21.5	Handler types	146
21.6	Example.....	146
22	Packages and dependencies	149
23	Proxy configuration	152
23.1	<i>Urbex.Proxy.....</i>	152
23.2	<i>Proxy levels.....</i>	153
23.3	<i>Proxy setup in the client side</i>	153
23.4	<i>Proxy setup in the server side.....</i>	154
23.4.1	The PHP Proxy.....	154
23.4.2	The ASP Proxy.....	157
24	Release Notes.....	159
24.1	<i>Changes in version 5.5.....</i>	159
	Third parties copyright notice.....	160

1 Scope

This document describes the Javascript API for the Blom Urban Explorer (BlomURBEX) service.

The main language used for developing interactive and attractive WEB applications is JavaScript, so in order to facilitate our clients or third parties, the development of applications against BlomURBEX, Blom has developed this JavaScript API.

This JavaScript API provides a simpler access to BlomURBEX, than direct HTTP requests to the server. The API hides from the developers the underlying HTTP communication with BlomURBEX servers.

JavaScript API offers a set of objects and classes that can be easily added to the HTML page. These objects offer a simple set of functionality to the developer, and handle all the communication with the server, display the proper images, allow the user to interact with the images and also provide information about what is shown in the images or about measurements performed on the image.

The main object (the Map Object) will be used to display the images and handle all user interaction. It is able to display both Ortho, Ortho-rectified and Oblique images.

1.1 Compatibility

The JavaScript API is compatible with Internet Explorer 7 and onwards, Firefox 3.0 and onwards, Chrome 1.0 and onwards and Safari 3.2 and onwards.

HTTPS is supported by the API for all BlomURBEX services (third-party services are not included).

2 Map Creation

This chapter will discuss the basics for creating a map object from JavaScript that will display BLOM Urban Explorer data and will allow user interaction with it.

2.1 Accessing the API

The first step needed to be able to connect to the Urban Explorer and use the JavaScript API is to include the API and the style sheet. This is achieved by including the following code into the HEAD section of the HTML page:

```
<link
  rel="stylesheet"
  href="http://www.blomurbex.com/api/5.7/style.css"
  type="text/css" />

<script src="http://www.blomurbex.com/api/5.7/Urbex.js"></script>
```

2.2 Creating a Map

To create a Map in a web page it is necessary to create an object of the **Urbex.Map** class. To create an initial view in the map is necessary to pass a minimum of mandatory parameters in the view creation, for example:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'userToken',
  x: xCoordinate,
  y: yCoordinate} );
```

This creates a map displaying an 'ortho' view with 'Ortho' orientation, centred in the x, y coordinates passed. The Map will be displayed on the specified DIV component. The coordinates of the centre of the image have to be, by default, in the spherical Mercator projection system.

The 'ortho' view, with 'ortho' orientation is an orthogonal image.

2.3 Map Creation Parameters

When creating a new map, the full list of parameters that can be specified is:

```
var newMap = new Urbex.Map(div, options);
```

Expanding the options parameter:

```
var newMap = new Urbex.Map(div, {
```

```
usertoken,  
x,  
y,  
projection,  
view,  
orientation,  
zoom,  
loadcontrols,  
controls,  
projection,  
MaximumDigitalZoom,  
Proxy,  
initEventsRegistry});
```

Some of the options are mandatory, and some are optional. Below there is an explanation of all parameters and options.

- **div** (mandatory): The map must be created in an HTML 'div' element of the web page. The ID of this DIV element is specified by this parameter. The Map will have the dimensions of the DIV element.

This is an example of a valid DIV element to display a map:

```
<div style="position: relative; width: 800px; height: 600px; top: 0px;  
left: 0px; z-index: 1" id="map"></div>
```

- **options:**

- **usertoken** (mandatory): {String} Identifying of user. This is a string that identifies the user and what data will be available for the user. This data needs to be requested to Blom.
- **x** (mandatory): {Float} X coordinate to centre the view
- **y** (mandatory): {Float} Y coordinate to centre the view
- **projection** (optional): {String} Projection used to specified the coordinates for the centre of the map, if none is provided it is assumed that they are in spherical Mercator (projection: 'EPSG:3785'). Only this projection is supported for this version.
- **view** (optional): {String} Type of view that will be displayed on the map. There are two possible values for this parameter: Ortho View (view: 'ortho') and Oblique View (view: 'OBLIQUE').

BlomURBEX™ Javascript API v5.7 r1.0

With this parameter it can be controlled if the map will show orthogonal images (either ortho images or ortho-rectified oblique images) or if it's showing Natural Oblique Images.

If the parameter is not specified, the default View will be Ortho

- **orientation** (optional): {String} This parameter specifies the orientation of the view. In Ortho view there are five possible values for this parameter:
 - Orthophotos (orientation: 'ORTHO'), this is the default if the parameter is not specified and the view is ortho.
 - Oblique rectified north (orientation: 'NORTH').
 - Oblique rectified south (orientation: 'SOUTH').
 - Oblique rectified east (orientation: 'EAST').
 - Oblique rectified west (orientation: 'WEST').

In Oblique view there are four possible values for this parameter:

- Real oblique north (orientation: 'NORTH'), this is the default if the parameter is not specified and the view is oblique.
 - Real oblique south (orientation: 'SOUTH').
 - Real oblique east (orientation: 'EAST').
 - Real oblique west (orientation: 'WEST').
- **zoom** (optional): {Integer} Zoom in which the map was loaded. There are 20 levels for view ortho (0 is the farthest, 20 is the nearest) and 4 levels for view oblique (0 is the farthest, 3 is the nearest)

The default values for the zoom level are:

- level 17 for Ortho View
 - level 3 for Oblique View
- **loadcontrols** (optional): {String} This parameter indicates controls to be loaded on map.

BlomURBEX™ Javascript API v5.7 r1.0

- To have a map without any control use: loadcontrols: 'no'. This map won't have any interaction at-all with the user. The web application will have to handle any desired interaction.
 - To load the map with minimal controls: pan navigation and wheel-mouse zoom, use: loadcontrols: 'lite'.
 - To load the map with navigation controls: pan navigation and wheel-mouse zoom, zoom-in button, zoom-out button, slider zoombar, change view button, and change orientation button, use: loadcontrols: 'navigation'.
 - The default controls are: pan navigation, wheel-mouse zoom, zoom-in button, zoom-out button, slider zoombar, change view button, change orientation button, measure length, area, height, bearing, elevation and eraser button. These will be the controls loaded whether loadcontrols: 'full' is specified or if this parameter is not used.
- **controls** (optional): {Object} <Urbex.Control> object or objects collection, to be loaded in the map during the map creation, beside the controls added according to the value of loadcontrols. The proper use of these two options will define what controls will be on the map.

This property will be one or more controls in a list, so it's necessary to create the controls before creating the map:

```
var objLayerSwitcher = new Urbex.Control.LayerSwitcher();
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringtoken',
  x: xCoordinate,
  y: yCoordinate,
  controls: [objLayerSwitcher]});
```

Or:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringtoken',
  x: xCoordinate,
  y: yCoordinate,
  controls: [new Urbex.Control.LayerSwitcher()]});
```

BlomURBEX™ Javascript API v5.7 r1.0

- **projection** (optional): {String} The EPSG identifier of the map's projection. The default value is spherical mercator (EPSG:3785). For a further compatibility with other projection taxonomies the string must contain the 'EPSG:' prefix.
- **maximumDigitalZoom** (optional): {Integer} The number of extra zoom levels added to the oblique view. Digital zoom implies pixel interpolation. By **default** this value is set to **2**
- **proxy** (optional): {Object} <Urbex.Proxy> Urbex's requests can be piped through a proxy. For more information, see the Proxy configuration chapter.
- **initEventsRegistered** (optional): {Array of objects}. The events attached to the map from the first map load. This array should contain an object for every event the developer wants to bind to the map before it loads for the first time. Every event object must have three attributes:
 - **eventName**: The name of the events (further information on the 'Events' section (18).
 - **object**: Object to pass as a parameter to the 'register' function. Further information on section 18.
 - **Callback**: Callback function which will be executed whenever the event registered fires.

Below there are a few examples of map creation:

- This example shows how to create a map with minimal parameters:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringtoken',  
  x: -521345,  
  y: 4960372 });
```

- This example shows how to create a map with west orientation in oblique view using several parameters:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -521345,  
  y: 4960372,  
  projection: 'EPSG:3785',  
  view: 'OBLIQUE',  
  orientation: 'WEST',  
  zoom: 2,
```

```
loadcontrols: 'lite',
controls: [new Urbex.Control.LayerSwitcher]
initEventsRegistered: [{callback: function(){alert('obliqueloaded')}},
object: newMap, eventName: 'areameasured']] });
```

2.4 Example: Minimum Application: The BlomURBEX “Hello, World”

The following code shows the minimum code needed for having an html page with an Urban Explorer viewer:

```
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Urban Explorer Basic Example</title>
    <link
      rel="stylesheet"
      href="http://www.blomurbex.com/api/5.7/style.css"
      type="text/css" />
    <script src="http://www.blomurbex.com/api/5.7/Urbex.js"></script>
    <script type="text/javascript">
      function init() {
        var newMap = new Urbex.Map('map', {
          usertoken: 'stringToken',
          x: -523121,
          y: 4960371});
      }
    </script>
  </head>
  <body onload="init()">
    <div style="position: relative; width: 800px; height: 550px; top:
0px; left: 0px; z-index: 1" id="map"></div>
  </body>
</html>
```

2.5 Internet Explorer 8 development

2.5.1 API Version 4.5 and onwards

Since version 4.5 of Javascript API, vector rendering in IE8 was solved, and no extra action needs to be taken by developers.

2.5.2 API versions older than 4.5

Final release of IE8 has introduced changes to the VML renderer engine that haven't been documented by Microsoft. For now the quickest solution to be able to display vector data with IE8 is to include in the html page the new directive:

```
<!meta http-equiv="X-UA-Compatible" content="IE=7" />
```

so that IE8 will use the IE7 rendering Engine.

Another solution is NOT to include the <!DOCTYPE> directive in the html document.



3 Basic Classes

This section covers some basic and useful API classes and their associated methods.

3.1 Urbex.XY

3.1.1 Description

This class represents a pair of coordinates, x and y, in your map units.

3.1.2 Construction

To create a new map location use:

```
Urbex.XY({Float} x, {Float} y);
```

3.1.3 Properties

- **x**: {Float}, represents the X-axis coordinate in map units.
- **y**: {Float}, represents the Y-axis coordinate in map units.

3.1.4 Example:

```
var point = new Urbex.XY(-356300, 4950100);  
var x = point.x;  
var y = point.y;  
  
x = -356300, y = 4950100
```


3.2 Urbex.Size

3.2.1 Description

Instances of this class represent a width/height pair.

3.2.2 Construction

To create a new instance of Urbex.Size:

```
Urbex.Size({Number} w, {Number} h);
```

3.2.3 Properties

- **w**: {Number}, represents width.
- **h**: {Number}, represents height.

3.2.4 Example

```
var size = new Urbex.Size (800, 600);  
var w = size.w;  
var h = size.h;  
  
w = 800, h = 600
```



3.3 Urbex.Pixel

3.3.1 Description

This class represents a screen coordinate, in x and y coordinates.

3.3.2 Construction

To create a new instance of Urbex.Pixel:

```
Urbex.Pixel ({Number} x, {Number} y);
```

3.3.3 Properties

- **x**: {Number}, the x coordinate.
- **y**: {Number}, the y coordinate.

3.3.4 Example

```
var pixel = new Urbex.Pixel (100, 100);  
var x = size.x;  
var y = size.y;  
  
x = 100, y = 100
```



3.4 Urbex.Bounds

3.4.1 Description

Instances of this class represent bounding boxes. Data stored as left, bottom, right, top floats.

All values are initialized to null, however, it is necessary to initialize them to proper values before using the bounds for anything.

3.4.2 Construction

To create a new instance of Urbex.Pixel:

```
Urbex.Bounds({Number}left, {Number}bottom, {Number}right, {Number}top);
```

3.4.3 Properties

- **left:** {Number}, The left bounds of the box.
- **bottom:** {Number}, the bottom bounds of the box.
- **right:** {Number}, the right bounds.
- **top:** {Number}, the top bounds.

3.4.4 Example

```
var bound = new Urbex.Bounds(-350300, 4950100, -350680, 4950680);  
var left = bound.left;  
var bottom = bound.bottom;  
var right = bound.rigth;  
var top = bound.top;  
  
left = -350300, bottom = 4950100, right = -350680, top = 4950680
```

4 Basic Map Settings

This section covers methods of the `Urbex.Map` class allowing basic navigation functions in the map (panning and zooming)

4.1 **setWorldCenter** method

4.1.1 Setting a new Centre

It is possible to change the central point shown in the map invoking the **setWorldCenter** method of the `Urbex.Map` object, just giving a new pair of coordinates:

```
var newCenter = new Urbex.XY(-523456, 4901247);  
newMap.setWorldCenter(newCenter);
```

The centre point coordinates must be specified in Spherical Mercator projection. If the map is in oblique view mode, the best available oblique in the current orientation will be displayed.

Zoom level is not changed when using this function.

4.1.2 Setting the Centre and zoom simultaneously

It is possible to change both the central point and the zoom level, using the same method, but including another parameter for the new zoom level.

To set a new centre and zoom:

```
var newCenter = new Urbex.XY(-523456, 4901247);  
var newZoom = 18;  
newMap.setWorldCenter(newCenter, newZoom);
```

4.1.3 Change Zoom

To set only a new zoom maintaining the central point use:

```
var newZoom = 18;  
newMap.setWorldCenter(null, newZoom);
```

4.1.4 Change Maximum Digital Zoom

On the oblique images a digital zoom is supported. The number of levels of digital zoom allowed can be changed using the **setMaximumDigitalZoom** method of the `Urbex.Map` object.

BlomURBEX™ Javascript API v5.7 r1.0

This method will not take any effect if it is called while the map is in the oblique view because it may leave the map in an illegal state. Note that a "maximum digital zoom levels" set to zero will disable the digital zoom.

```
var maximumDigitalZoom = 2;  
newMap.setMaximumDigitalZoom(maximumDigitalZoom);
```



4.2 Pan method

This method allows to move the viewport a number of pixels without considering coordinates, **Urbex.Map.pan(dx, dy, options)**.

4.2.1 Parameters

- **dx:** {Integer}, (Mandatory), displacement in pixels for the x axis in horizontal. If dx is positive the movement of the viewport is to the right and if dx is negative the movement is to the left.
- **dy:** {Integer}, (Mandatory), displacement in pixels for the y axis in vertical. If dy is positive the movement of the viewport is upwards and if dy is negative the movement is downwards.
- **options:** (Optional)
 - **animate:** {Boolean}, if animation is set to true, the image will drag from the current point to the destination point smoothly in a continuous motion. If animation is set to false, the image will "jump" the specified number of pixels in just one motion. The default value is "false".
 - **dragging:** {Boolean}, Specifies whether or not to trigger movestart/end events. If dragging is set to true events will be triggered. If dragging is set to false events will not be triggered. The default value is "false"

4.2.2 Example

This example produces a 100 pixel viewport displacement to the right, without animation on the viewport movement.

```
var newMap;

function init() {
  newMap = new Urbex.Map('map', {
    usertoken: 'stringToken',
    x: -526488.22373,
    y: 5108391.05558});
}

function toggle() {
  newMap.pan(100, 0, {animation: false});
}
```

4.3 Intelligent Map Behaviour

At closer zoom levels, oblique rectified images show big distortion on the buildings, while at furthest zoom levels they do not offer better information than the ortho images.

The Map can be configured so that any request that attempts to change the zoom level of the view will check if the map is displaying any orientation other than ORTHO, and in this case decide what set of images should be displayed: ortho, oblique rectified or natural obliques.

By default this behaviour is disabled, to activate it the **isIntelligent** map property must be set to **true**:

```
newMap = new Urbex.Map('map', {  
  usertoken: 'stringToken',  
  x: -526488.22373,  
  y: 5108391.05558});  
  
newMap.isIntelligent = true;
```

4.4 Auto rotate tiles Map Behaviour

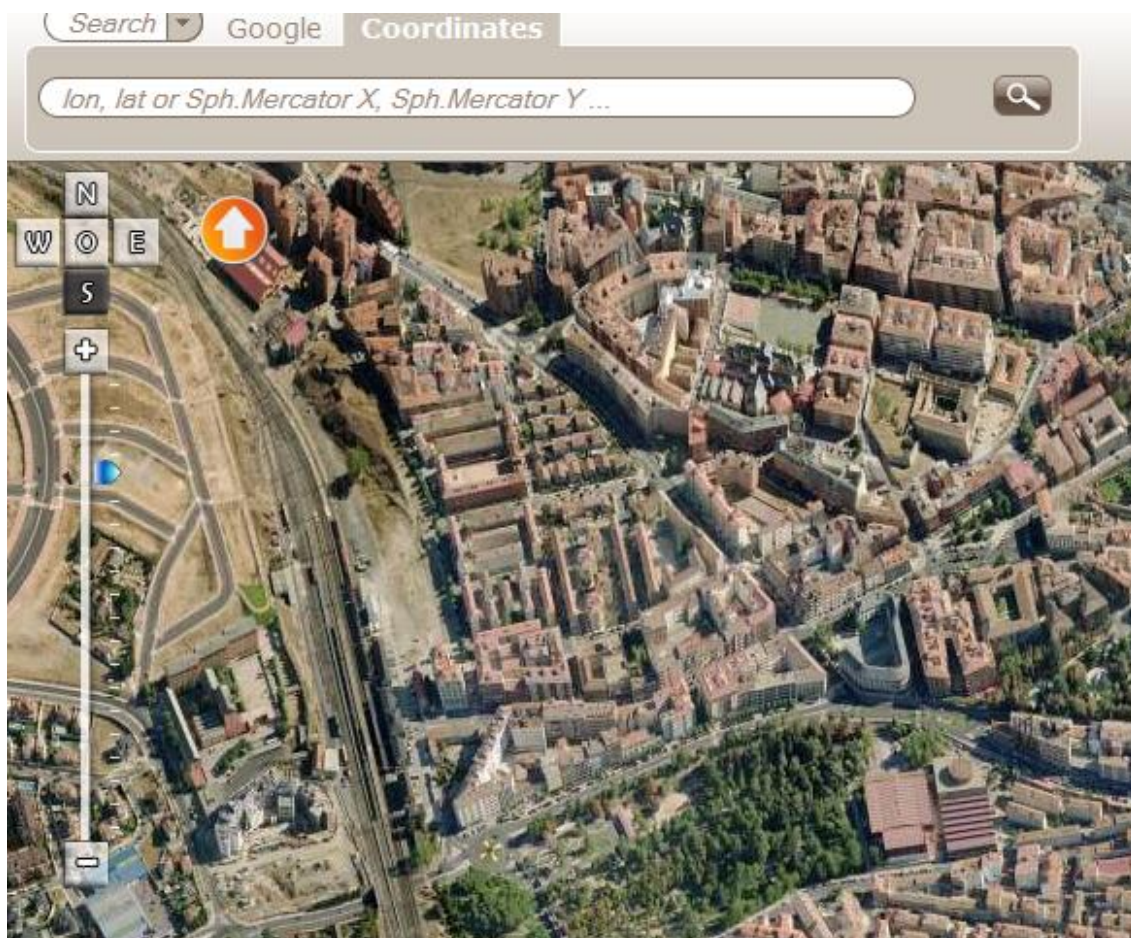
The Map can be configured so that tiles can be rotated or not when use different orientation views N, S, W and E.

```
newMap = new Urbex.Map('map', {  
  usertoken: 'stringToken',  
  x: -526488.22373,  
  y: 5108391.05558}  
  autoRotateTiles:true);
```

4.5 Force always Ortho in mosaic mode Map Behaviour

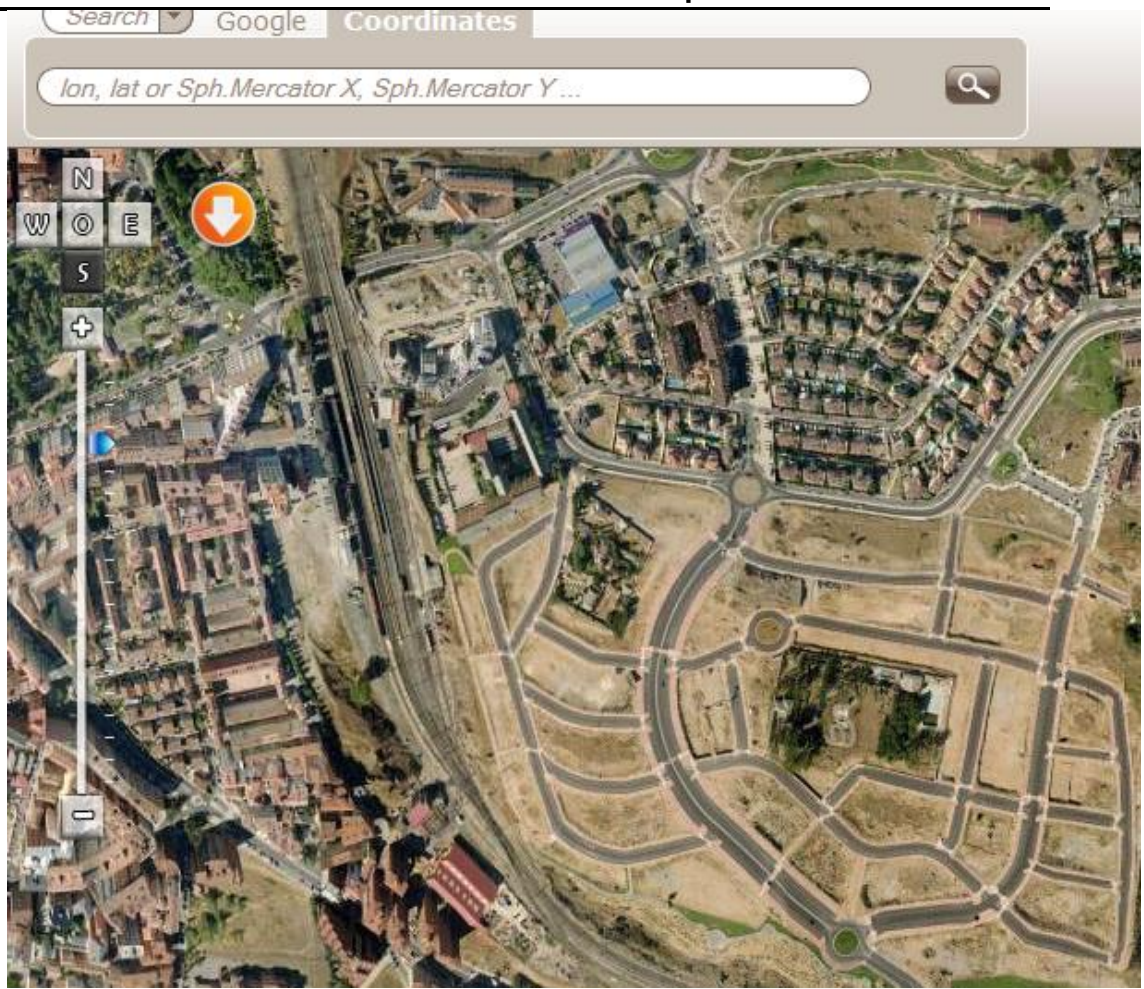
The Map can be configured so that tiles show always the north side up in each view.

```
newMap = new Urbex.Map('map', {  
  usertoken: 'stringToken',  
  x: -526488.22373,  
  y: 5108391.05558}  
  forceAlwaysOrthoInMosaicView:true);
```

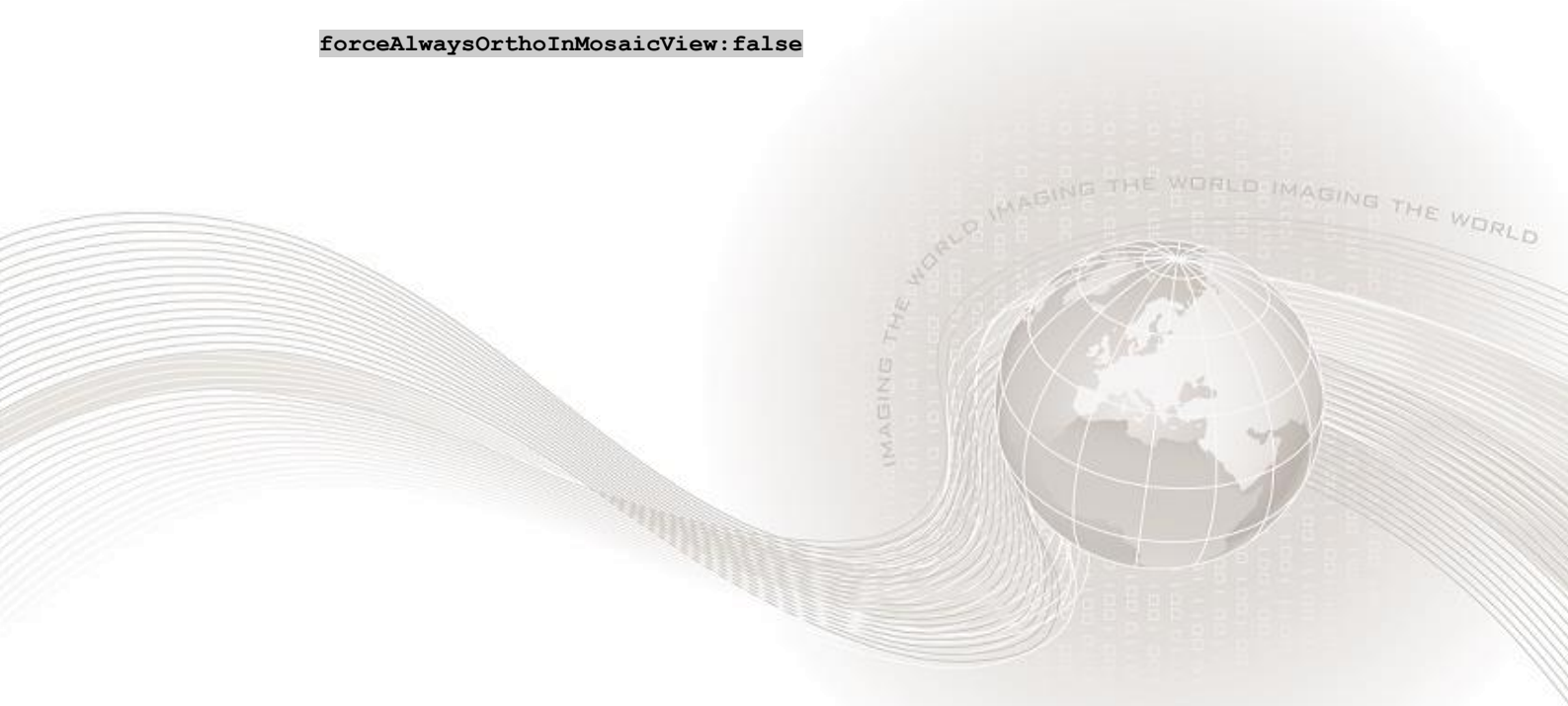


```
forceAlwaysOrthoInMosaicView:true
```





`forceAlwaysOrthoInMosaicView:false`



Limiting the zoom

The zoom capabilities of the map can be limited to let the user access only to some zoom levels of the images.

The method **Urbex.Map.setZoomLimits(minZoom, maxZoom, minObl, maxObl)** allows the developer to set the range of available zoom levels.

4.5.1 Parameters

- **minZoom:** {Integer}, Minimum allowed zoom in ortho view, a null value will cancel this restriction. The minimum allowed value is 1
- **maxZoom:** {Integer}, Maximum allowed zoom in ortho view, a null value will cancel this restriction
- **minObl:** {Integer}, Minimum allowed zoom in oblique view, a null value will cancel this restriction. The minimum allowed value is 1
- **maxObl:** {Integer}, Maximum allowed zoom in oblique view, a null value will cancel this restriction

4.5.2 Zoom limits and intelligent behaviour

There are some key zoom levels that are needed so that the map can offer the intelligent behaviour. If the developer takes them out from the available set of zoom levels, this behaviour will be cancelled.

In order for the intelligent behaviour to work:

- The minimum zoom level in oblique view must be 1.
- The maximum zoom level in ortho view must be greater than 16.

4.5.3 Example

The next example will set zoom limits that allow the intelligent behaviour.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'usertoken',  
  x: -523121,  
  y: 4960371});  
  
newMap.isIntelligent = true;  
newMap.setZoomLimits(10, 17, 1, 3);
```

The next example will set zoom limits that cancel the intelligent behaviour:


```
var newMap = new Urbex.Map('divName', {
  usertoken: 'usertoken',
  x: -523121,
  y: 4960371});

newMap.isIntelligent = true;
newMap.setZoomLimits(10, 15, 2, 4);
```

4.6 Changing Map Image Display

There are two map parameters that will determine what images will be displayed on the map: **map.TIME** and **map.BASELAYER**

- **map.TIME:** This map variable can either be a year (e.g 2009), a year interval (e.g. 2007-2009) or it can be left empty. If **map.TIME** is set to a year, **setView** will look for map images taken that year, and if no available images meet this condition, it will display the closest map images taken before that year. On the other hand, if **map.TIME** is set to a year interval, **setView** would look for map images taken some time between that interval. If no images can be found, the map will display an empty (transparent) image. **Map.TIME** can also be set to make the map display images only for a given year, by setting its value to an interval with the same start and end year. If, on the other hand, **map.TIME** is left empty, the map will display the newest images the user is allowed to see.
- **map.BASELAYER:** This map variable controls the order of preference for the display of the available ortho images datasets available in BlomUrbex.

The available ortho datasets are:

- **satellite:** Satellite images
- **countryortho:** ortho images of a country or region
- **pictometry:** ortho mosaic generated from pictometry data

The variable **BASELAYER** will be a string containing any combination of this values (ortho datasets) separated by commas. The order in which the datasets appear in the **BASELAYER** variable, will define the priority of the dataset. If some dataset is not present, no images from that dataset will be used.

Each user can have a different set of priorities (defined when the user is created), and by default it is: pictometry, countryortho and

BlomURBEX™ Javascript API v5.7 r1.0

satellite. Setting this variable will allow to change the by default behaviour.

After setting these variables, it's necessary to refresh the map to see the changes.

Following example shows a full HTML page which displays a map centered in Rome, alongside with 2 buttons to change the preferred baselayer from pictometry to countryortho and the other way around. There is also a text area where the user could write the year or year interval he desires for the map to display the images. Whenever a button is pressed, either map.BASELAYER variable or map.TIME variable will be altered, and the map will be refreshed using setView() function.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Urban Explorer Change Map Location</title>

    <script src="http://www.blomurbex.com/api/5.7/Urbex.js"></script>
    <script type="text/javascript">

      function init() {
        newMap = new Urbex.Map('map', {
          usertoken: 'stringToken',
          loadcontrols: 'no',
          x: 1389525.984992505,
          y: 5145333.536935514,
          zoom:17,
          view: 'ortho',
          orientation: 'ortho'});

        document.getElementById('textYears').value = newMap.TIME;
      }

      function setPictometry() {
        newMap.BASELAYER="pictometry,countryortho,satellite";
        newMap.setView();
      }

      function setCountryOrtho() {
        newMap.BASELAYER="countryortho,pictometry,satellite";
        newMap.setView();
      }
    </script>
  </head>
  <body>
    <div id="map">
      <img alt="Map of Rome" data-bbox="200 340 850 880"/>
    </div>
    <div>
      <input type="button" value="Pictometry to CountryOrtho" />
      <input type="button" value="CountryOrtho to Pictometry" />
      <input type="text" id="textYears" value="1990-2010" />
    </div>
  </body>
</html>
```

```
}

function setYear(){

    newMap.TIME = document.getElementById('textYears').value;
    newMap.setView();
}

</script>
</head>

<body onload="init()">

    <div style="position: relative; width: 800px; height: 550px; top:
0px; left: 0px; z-index: 1" id="map"></div>

    <input type="button" id="boton" value="Pictometry"
onclick="javascript:setPictometry();" />

    <input type="button" id="boton" value="CountryOrtho"
onclick="javascript:setCountryOrtho();" />

    &nbsp; &nbsp; &nbsp; Write the year or year interval you desire: &nbsp; &nbsp;

    <input type="text" id="textYears" size="20" />

    <input type="button" id="boton" value="Go"
onclick="javascript:setYear();" />

</body>
</html>
```

4.7 Changing map's tiles scale

This API gives the option to change the scale applied to the tiles when displaying them, so that users can see a mosaic of tiles of a certain zoomlevel building a map of a lower level of zoom than the tiles.

Every map has a 'scale' parameter set initially to 1 (the default scale value). This scale means that tiles won't be scaled and that a mosaic of zoomlevel X will be containing tiles of that zoomlevel X.

Any value greater than 1 will force the tiles to scale and will compound a mosaic of zoomlevel X with tiles of zoomlevel (X+scale-1). In that case, every 256x256 square on screen will be covered by a number of tiles equal to 2 raised to the power of 2*(scale-1) tiles. So, in scale = 1, every tile will cover a 256x256 square. In scale 2, 4 tiles will, in scale 3, 8 tiles will, and so on.

BlomURBEX™ Javascript API v5.7 r1.0

Users can change this parameter either using the TileScale control or by code just accessing the map's attribute 'scale', changing its value and resetting the map, like in the example.

```
map.scale = 2;  
map.setView();
```



5 Changing View mode, Orientation and Projection

5.1 Change view type

To change the view type between ortho and oblique is necessary to invoke the **setView** method of the **Urbex.Map** object, which parameters are:

```
setView(  
    {UrbexXY} centre,  
    {String} view,  
    {String} orientation,  
    {Number} zoom)
```

5.1.1 Examples

To change from the actual view to an oblique view with south orientation, centred in a new point and change the zoom level:

```
var newCenter = new Urbex.XY(-523456, 4901247);  
var newMap = 'OBLIQUE';  
var newOrientation = 'SOUTH';  
var newZoom = 0;  
newMap.setView(newCenter, newMap, newOrientation, newZoom);
```

In the example above, it is also shown how to declare a variable of type **Urbex.XY**, which is used to specify the new desired centre.

To keep the same centre point and zoom level, and only change the View Mode and Orientation:

```
var newMap = 'OBLIQUE';  
var newOrientation = 'SOUTH';  
newMap.setView(null, newMap, newOrientation, null);
```

If the orientation parameter is set to null, the map will show the default orientation for the view mode: 'ORTHO' orientation for 'ORTHO' view mode, and 'NORTH' orientation for 'OBLIQUE' view.

For example: the view will be loaded with its default orientation:

```
var newMap = 'OBLIQUE';  
newMap.setView(null, newMap, null, null);
```

To change the orientation displayed in the actual View there are other methods available, which will be covered in the following sections.

5.2 Change orientation in ortho view

To change the orientation of the view while in ortho mode, **setOrthoOrientation** method of the **Urbex.Map** object, will be used, with the following parameters:

```
setOrthoOrientation(  
  {String} orientation,  
  {Urbex.XY} centre,  
  {Number} zoom)
```

There are five possible orientations in ortho mode: 'ORTHO', 'NORTH', 'SOUTH', 'EAST' and 'WEST'. And they correspond to the ortho images and the ortho-rectified oblique images.

5.2.1 Examples

To change to 'NORTH' orientation, changing the centre and zoom in the view at the same time:

```
newOrientation = 'NORTH';  
var newCenter = Urbex.XY(-523456, 4901247);  
var newZoom = 20;  
newMap.setOrthoOrientation(newOrientation, newCenter, newZoom);
```

For changing the orientation without changing the centre and zoom:

```
newOrientation = 'SOUTH';  
newMap.setOrthoOrientation(newOrientation, null, null);
```

5.3 Change orientation in oblique mode

To change the orientation while the map is in oblique view mode, the **setObliqueOrientation** method of the **Urbex.Map** object will be used with the following parameters:

```
setObliqueOrientation(  
  {String} orientation),  
  {Urbex.XY} centre,  
  {Integer} zoom)
```

There could be four possible orientations in an oblique view: 'NORTH', 'SOUTH', 'EAST' and 'WEST'.

5.3.1 Example

To change to a south orientation of an oblique view, while changing the centre and zoom in the view at the same time:

```
newOrientation = 'SOUTH';  
var newCenter = Urbex.XY(-523456, 4901247);  
var newZoom = 20;  
newMap.setObliqueOrientation(newOrientation, newCenter, newZoom);
```

For changing the orientation without changing the centre and zoom:

```
newOrientation = 'EAST';  
newMap.setObliqueOrientation(newOrientation, null, null);
```

5.4 Change projection

To change the map projection, use the **setProjection** method of the **Urbex.Map** object, with the following parameters:

```
setProjection({String} projection).
```

The map will calculate all its data in the new given projection (centre, extent, resolution...) and will accordingly request images to BlomURBEX server. Currently not all world projections are supported by BlomURBEX; if the user specifies a non-supported projection there will be no images returned and a transparent map will be showed.

5.4.1 Example

To change settings to an UTM30 projection:

```
Utm30Code = 'EPSG:32630';  
newMap.setProjection(Utm30Code);
```



6 Retrieving map information

This section covers the `Urbex.Map` methods to request information about the current map.

6.1 Centre

To obtain the current centre of the map view there are two available methods:

- **`Urbex.Map.getCenter()`**: Returns an **`Urbex.XY`** object that represents the centre point of the view. In ortho view, it returns the world coordinates in the same projection than the map. In oblique view it returns the pixel coordinates of the centre point in the current oblique view.

```
var newMap = new Urbex.Map('divName', {usertoken: 'stringtoken', x: -523456, y: 4901247, view: 'OBLIQUE', orientation: 'WEST'});
var center = newMap.getCenter();
x = center.x;
y = center.y;

x = 1158 and y = 888
```

- **`Urbex.Map.getWorldCenter(listener)`**: Returns an **`Urbex.XY`** object that represents the centre point of the instantiated view. It always returns the world coordinates in the same projection than the map (by default Spherical Mercator).
 - **`Listener {Function}`**: Mandatory. **`Listener`** is a function where to obtain the reference to the `Urbex Map` and the centre coordinates of the map.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523456,
  y: 4901247,
  view: 'OBLIQUE',
  orientation: 'WEST'});
newMap.getWorldCenter(callback);

function callback(map, center) {
  x = center.x;
  y = center.y;
}
```

```
x = -523456 and y = 4901247
```

6.2 Zoom level

To obtain the current zoom level of the view the following methods are available:

- **Urbex.Map.getZoom():** Returns an Integer that represents the zoom level of the current view. Returns the real zoom level in ortho and oblique view.

For example, after using:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringtoken',  
  x: -523456,  
  y: 4901247,  
  view: 'OBLIQUE',  
  orientation: 'WEST',  
  zoom: 3});  
  
var zoom = newMap.getZoom();
```

The value of zoom will be (as long as the user doesn't change the view mode or the orientation with the map controls):

```
zoom = 3
```

- **Urbex.Map.getWorldZoom():** Returns an Integer that represents the zoom level of the current view but always referred to the Ortho View.

For example, after using:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringtoken',  
  x: -523456,  
  y: 4901247});  
  
var newView = 'OBLIQUE';  
var newOrientation = 'SOUTH';  
var newZoom = 2;  
  
newMap.setView(null, newView, newOrientation, newZoom);  
var zoom = newMap.getWorldZoom();
```

The value of zoom will be (as long as the user doesn't change the view mode or the orientation with the map controls):


```
zoom = 17
```

- **Urbex.Map.getMinZoom():** Returns the value of the minimum zoom available for the current view (see 0 Limiting the zoom).
- **Urbex.Map.getMaxZoom():** Returns the value of the maximum zoom available for the current view (see 0 Limiting the zoom)

6.3 Digital Zoom

The number of allowed levels of digital zoom can be obtained by calling the **getMaximumDigitalZoom** function.

For example, to increase the maximum number of digital zoom levels, the following code can be used:

```
newMap.setMaximumDigitalZoom(newMap.getMaximumDigitalZoom() + 1);
```

The default maximum digital zoom levels value is 2.

The digital zoom only applies to Oblique images.

6.4 Extent

To obtain the Extent of the map:

- **Urbex.Map.getExtent():** Returns a **Urbex.Bounds** object, which represents the extent or bounds of the current view. The values returned will be world coordinates in Ortho views and pixel coordinates in the Oblique Views.

```
var bounds = newMap.getExtent();  
var xmin = bounds.left  
var ymin = bounds.bottom  
var xmax = bounds.top  
var ymax = bounds.rigth
```

6.5 View Type and Orientation

6.5.1 View Type

To obtain the **view type** of a map, the **activeView** property can be retrieved. This property is a string that identifies the view type displayed in the map; it can be either 'ORTHO' or 'OBLIQUE'.

6.5.2 Orientation

To obtain the **orientation** of the current view displayed on a map, the **orientation** property can be retrieved. This property is a string that identifies the orientation of the displayed image.

In Ortho view there are five possible values for this property:

- 'ORTHO' when displaying Orthophotos.
- 'NORTH' when displaying Orthorectified Oblique North.
- 'SOUTH' when displaying Orthorectified Oblique South.
- 'EAST' when displaying Orthorectified Oblique East.
- 'WEST' when displaying Orthorectified Oblique West.

In Oblique view there are four possible values for this property:

- 'NORTH': Real oblique north
- 'SOUTH' Real oblique south
- 'EAST' Real oblique east
- 'WEST' Real oblique west

6.5.3 Example

In this example:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523456,  
  y: 4901247});  
viewType = newMap.activeView;  
orientation = newMap.later;
```

The value of viewType and orientation will be (as long as the user doesn't change the view mode or the orientation with the map controls):

```
viewType = 'ORTHO', orientation = 'ORTHO'
```

Another example:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523456,
```

```
y: 4901247,  
view: 'OBLIQUE',  
orientation: 'WEST',  
zoom: 3});  
viewType = newMap.activeView;  
orientation = newMap.orientation;
```

The results should be:

```
viewType = 'OBLIQUE', orientation = 'WEST'
```

6.6 View Projection

The current projection can be retrieved using the **getProjection** method:

```
var myCurrentProjection = newMap.getProjection();  
if (myCurrentProjection == 'EPSG:3785') {  
    // DO MY SPHERICAL MERCATOR STUFF  
} else if (myCurrentProjection == 'EPSG:32630') {  
    // DO MY UTM30 STUFF  
} else {  
    // DO MY OTHER PROJECTION STUFF  
}
```

The default projection is Spherical Mercator.

6.7 Retrieving the layers.

Sometimes it will be necessary to get a layer from the map, just for convenience or just because the developer needs to use a layer created by the map. The **getLayersByName** method can be used for retrieving the map's layers. This method needs one parameter only:

- **name:** The name of the map, it can be a string with the exact name, or a regular expression with the name pattern.

The result is an array with the matching layers.

The next example finds out the number of layers of the map by retrieving all the layers and counting them:

```
var allLayers = newMap.getLayersByName(/.*/);  
alert(allLayers.length);
```

7 Managing Controls

7.1 Available controls

The JavaScript API offers several controls that will allow interaction between the user and the map. These controls are the following:

- **Urbex.Control.Navigation:** This control will respond to click and drag events triggered by the mouse on the image.
 - **Pan navigation:** Allows the dragging of images
 - **Wheel mouse zoom:** the zoom level is changed through the wheel on the mouse
 - **SHIFT zoom box:** Pressing the Shift Key on the keyboard while dragging the mouse with the left button pressed, will allow performing a zoom in to a particular area.
- **Urbex.Control.ZoomIn:** Zoom in. Increments the zoom level. This control is linked with the ZoomBar control. When the zoom increases the slider button climbs up on the zoom bar.
- **Urbex.Control.ZoomOut:** Zoom out. Reduces the zoom level. This control is linked with the ZoomBar control. When the zoom decreases the slider button climbs down on the zoombar.
- **Urbex.Control.ZoomBar:** This control presents a graphic display of the actual zoom level and allows changing the zoom level with a slider button.
- **Urbex.Control.ChangeView:** Allows changing between ortho and oblique view and vice versa, with only a one button click. The appearance and text of the button changes as the map changes to another view.
- **Urbex.Control.ChangeView2:** Allows changing between ortho and oblique view and vice versa, with two buttons. The active current view can be known because the corresponding button stays active and its appearance changes accordingly.
- **Urbex.Control.ChangeOrientation:** Allows changing the shot orientation shown in the actual view. The active current view is known because the corresponding button stays active.
- **Urbex.Control.LayerSwitcher:** Controls the visibility of the Overlays layers via check buttons. One click brings up a list of layers and overlays, another click hides the list.

- **Urbex.Control.ReturnPoint:** Returns coordinates of the selected point on the map. One click brings up a div with the coordinates, click into the map to obtain the corresponding coordinates to the selected point, another click on the control hide the div with the coordinates.
- **Urbex.Control.GetMap:** Downloads the current image displayed on the map via a WMS request. Result is provided as a JPG image.
- **Urbex.Control.Length:** Measures distances between two or more points in oblique and ortho views. When this button is selected the rest of measurement controls are disabled, until the control is de-selected clicking the control again, or the Eraser control is selected.
- **Urbex.Control.Area:** It measures areas in oblique and ortho views. When this button is selected the rest of measurement controls are disabled, until the control is de-selected clicking the control again, or the Eraser control is selected.
- **Urbex.Control.Height:** It measures heights only in oblique views. When this button is selected the rest of measurement controls are disabled, until the control is de-selected clicking the control again, or the Eraser control is selected.
- **Urbex.Control.Bearing:** It measures bearings only in oblique views. When this button is selected the rest of measurement controls are disabled, until the control is de-selected clicking the control again, or the Eraser control is selected.
- **Urbex.Control.Elevation:** It measures elevations in oblique and ortho views. When this button is selected the rest of measurement controls are disabled, until the control is de-selected clicking the control again or the Eraser control is selected.
- **Urbex.Control.Eraser:** Disables measuring controls activated and clears the measuring elements added to the map.
- **Urbex.Control.Click:** When it is active, the map will trigger a locationmeasured event on each click of the user.
- **Urbex.Control.North:** It shows an arrow that always points to the north.
- **Urbex.Control.Scale:** It displays the scale of the actual view.
- **Urbex.Control.Metadata:** It is visible in oblique view only. This control shows the capture date of the oblique image displayed.

- **Urbex.Control.ShowStreets:** It shows and hides an overlay layer that displays the streets and street names.
- **Urbex.Control.MultiOblique:** When the map is displaying an oblique view, MultiOblique control shows a list of all Oblique images covering the centre point of the map, with the same orientation as the map. It also allows the user to switch between them. When the map is displaying an ortho view, this control is disabled.

When clicked, this control shows the list of oblique images, and when clicking any of them it is displayed on the map. Whenever the MultiOblique control is on, the ChangeView control is disabled so the user cannot change the view while browsing the oblique images list.

- **Urbex.Control.Year:** This control allows the user to select the year of which he wants the images to be displayed. When clicked, it shows a timeline between the current year and 2004, if no year limits have been specified. In that timeline, the user is able to select the time range he desires (it will be stored on the variable **TIME** of the map javascript object). When turned off, this control refreshes the map so that it will display images according to the new time selection made by the user.

The user can either select a single year, a range of years or the best image. If he selects a single year, the map will display the images available that year. On the other hand, if the user selects a range of years, the map will display the newest images available dated on that time range. If the user selects the options for the best image, the newest image will be displayed. In all cases, if no images available meet these criteria, a transparent image will be displayed. See section 4.6 for an explanation about the **map.TIME** variable.

The user can set specific year limits for the timeline displayed on the Year control. In order to do so, an object must be passed as a parameter to the control's constructor. This object should include two properties, firstYear and lastYear, with the numeric value of the year limits. If no limits are specified when creating the control, it will limit itself between current year and 2004.

```
new Urbex.Control.Year({firstYear: 2006, lastYear: 2009});
```

- **Urbex.Control.ReverseGeocoding:** This control will allow users to pick a point in the map and it will display the address for that point using the reverse geocoding services of the API.
- **Urbex.Control.TileScale:** This control will allow the user to choose the scale the tiles will be resized with using the attribute 'scale' from

BlomURBEX™ Javascript API v5.7 r1.0

the map. By default, tiles won't be scaled. Each one of them will be a 256x256 pixels image in the screen (scale = 1). If scale is bigger than one, the map will scale the tiles, preserving the global zoom level the user is watching.

That means that, if the user is watching a zoom level X and wants to scale in half the tiles (scale = 2), then the new tiles requested will have of zoomlevel X+1, so that the user still sees the same resolution even when resized.

The control will display 2 arrows to increase or decrease the scale, and 2 textboxes showing the zoom level the user is viewing and the actual zoomlevel of the tiles being displayed. When scale = 1, both will be the same value.

The maximum scale level is set to 3 (the tiles will be resized to a quarter of its original size). That is because a scale = 3 situation will force the map to download four times as much tiles as when scale = 1, so it may collapse the navigator and make its response slower.

- **Urbex.Control.Baselayer:** This control allows the user to define the baselayer priority for requesting and displaying images as detailed on section 4.6.

The control allows the user to re-order the baselayers already selected and to add new ones and remove existing ones.

- **Urbex.Control.Overlays:** This control shows the user all the available overlays he is allowed to display, and lets him select and display them.
- **Urbex.Controls.Diagonal:** This control will only be active in oblique mode. It measures the hypotenuse of a rectangle triangle formed by 3 points: The first and second point will be on the ground and the third point will be at the same xy coordinates as the second point but at a certain height.

This may be very useful for things like measuring the ladder length a fire truck will need if a fire starts in certain building.

- **Urbex.Control.VerticalArea:** This control will only be active in oblique mode. It measures the area of façade surfaces. The user should pick at least three points. The two first points will indicate the baseline of the façade to measure, so the server will assume that these two points are on the floor. All points besides these two will be assumed to be on the vertical plane defined by the first two points.

- **Urbex.Control.LayerSelector:** This control will be automatically activated whenever there are 2 or more 'clickable' layers in the map. It allows the user to choose which one of the clickable layers will be active, thus, will receive the click events. A clickable layer is a layer that may accept user interaction with it, as, for example, a Markers layers does.

7.2 Adding controls to the map

When the map is created it is possible to specify which controls have to be shown on the map. This can be easily done via the **loadcontrols** parameter in Map constructor. **loadcontrols** can have four possible values:

- **loadcontrols:** 'no'. The map is created without any control. This means that the map does not offer any user interaction.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -521345,  
  y: 4960372,  
  loadcontrols: 'no'});
```

- **loadcontrols:** 'lite'; loads the control for allowing minimal navigation with the mouse.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -521345,  
  y: 4960372,  
  loadcontrols: 'lite'});
```

- **loadcontrols:** 'navigation'; loads the navigation control for allowing minimal navigation through the mouse and ZoomIn, ZoomOut, ZoomBar, ChangeView and ChangeOrientation.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -521345,  
  y: 4960372,  
  loadcontrols: 'navigation'});
```

- **loadcontrols:** 'full'; loads a full set of controls, offering all navigation capabilities. Controls loaded are: navigation, ZoomIn, ZoomOut, ZoomBar, ChangeView, ChangeOrientation, Length, area, height, bearing, elevation and eraser.

```
var newMap = new Urbex.Map('divName', {
```

```
usertoken: 'stringToken',  
x: -521345,  
y: 4960372,  
loadcontrols: 'full'}});
```

To load a control configuration different from the default ones offered by the **loadcontrols** parameter, there is an option for passing a list of controls to the map creator, so that all controls in the list are loaded in the map.

To load **non-default** controls at the view creation with the **controls** property use the following code:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -521345,  
  y: 4960372,  
  controls: [new Urbex.Control.LayerSwitcher()]});
```

To load **non-default** controls on the map after its creation, use the **addControl** method on the map. This method will accept a control object and will add it to the map:

```
var layerSwitcher = new Urbex.Control.LayerSwitcher();  
newMap.addControl(layerSwitcher);  
  
var ReturnPoint = new Urbex.Control.ReturnPoint();  
newMap.addControl(ReturnPoint);
```

or:

```
newMap.addControl(new Urbex.Control.LayerSwitcher());  
newMap.addControl(new Urbex.Control.ReturnPoint());
```

7.3 Removing non-default controls on the map

Controls can be removed from the map by using the **Urbex.Map.removeControl** method. The method is expecting to receive the control object to be eliminated from the map:

```
var layerSwitcher = new Urbex.Control.LayerSwitcher();  
newMap.addControl(layerSwitcher);  
newMap.removeControl(layerSwitcher);
```

7.4 Exploring the controls collection

The map maintains a list with all the controls that are registered on it. To explore this controls collection use the **controls** property of the Map.

- **controls:** {Array(**Urbex.Control**)} List of controls associated with the view.

This example deletes the **zoomBar** control from the map by exploring the **controls** property.

```
var controlsColl = newMap.controls;
for(var i=0; i < controlsColl.length; i++) {
    var idControl = controlsColl[i].id;
    if (idControl.toUpperCase().indexOf('ZOOMBAR') != -1) {
        var remControl = newMap.getControl(idControl);
        if (remControl != null) {
            newMap.removeControl(remControl);
            break;
        }
    }
}
```



7.5 Customizing controls

It is possible to modify several properties associated with the controls, such as: location, size, images or tool tip. To allow this customization each control provides several methods.

7.5.1 Urbex.Control.Navigation

- **setZoomLimits**({Integer} minZoom, {Integer} maxZoom): Sets the zoom limits of the control to navigate through the entire set of zoom levels of the map. Sets the minimum zoom and maximum zoom allowed to the control.

7.5.2 Urbex.Control.ZoomIn

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the ZoomIn control.
- **setSize**({Urbex.Size} size): Sets the size of the ZoomIn control.
- **setImage**({String} url): Allows to define the image of the ZoomIn control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the ZoomIn control.
- **setZoomLimits**({Integer} minZoom, {Integer} maxZoom): Sets the zoom limits of the control to navigate through the entire set of zoom levels of the map. Sets the minimum zoom and maximum zoom allowed to the control.

7.5.3 Urbex.Control.ZoomOut

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the ZoomOut control.
- **setSize**({Urbex.Size} size): Sets the size of the ZoomOut control.
- **setImage**({String} url): Allows to define the image of the ZoomOut control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the ZoomOut control.
- **setZoomLimits**({Integer} minZoom, {Integer} maxZoom): Sets the zoom limits of the control to navigate through the entire set of zoom levels of the map. Sets the minimum zoom and maximum zoom allowed to the control.

7.5.4 Urbex.Control.ZoomBar

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the ZoomBar control.
- **setSizeBar**({Urbex.Size} size): Sets the size of the ZoomBar control, it specifies the width of the bar and the height of ONE step in the bar.
- **setSizeSlider**({Urbex.Size} size): Sets the size of the ZoomBar slider image.
- **setImageBar**({String} url): Allows to define the image of the ZoomBar image control. To ensure ie6 compatibility the image should be in both png format and gif format.
- **setImageSlider**({String} url): Allows to define the image of the ZoomBar slider image.
- **setTitleBar**({String} tip): Sets the tip to show when the mouse hovers the ZoomBar image.
- **setTitleSlider**({String} tip): Sets the tip to show when the mouse hovers the ZoomBar slider image.
- **setZoomLimits**({Integer} minZoom, {Integer} maxZoom): Sets the zoom limits of the control to navigate through the entire set of zoom levels of the map. Sets the minimum zoom and maximum zoom allowed to the control.

The zoom bar dynamically calculates its total size, so the setSize method (available for other controls) cannot be used. If this method is called for ZoomBar the call will raise an exception.

7.5.5 Urbex.Control.ChangeView

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the ChangeView control.
- **setSize**({Urbex.Size} size): Sets the size of the ChangeView image.
- **setImageOrtho**({String} url): Allows to define the image of the ChangeView ortho image.
- **setImageOblique**({String} url): Allows to define the image of the ChangeView oblique image.

- **setTitle({String} tip):** Sets the tip to show when the mouse hovers the ChangeView control.

7.5.6 Urbex.Control.ChangeView2

- **setPosition({Urbex.Pixel} position):** Establishes the position on the map of the ChangeView2 control.
- **setSize({Urbex.Size} size):** Sets the size of the ChangeView2 image.
- **setImageOrtho({String} url):** Sets the image of the ChangeView2 ortho image not selected.
- **setImageOblique({String} url):** Sets the image of the ChangeView2 oblique image not selected.
- **setSelectionOrtho({String} url):** Allows to define the image of the ChangeView2 ortho image selected.
- **setSelectionOblique({String} url):** Allows to define the image of the ChangeView2 oblique image selected.
- **setTitleOrtho({String} tip):** Sets the tip to show when the mouse hovers the ChangeView ortho image.
- **setTitleOblique({String} tip):** Sets the tip to show when the mouse hovers the ChangeView oblique image.

7.5.7 Urbex.Control.ChangeOrientation

- **setPosition({Urbex.Pixel} Background, {Urbex.Pixel} North, {Urbex.Pixel} South, {Urbex.Pixel} Ortho, {Urbex.Pixel} East, {Urbex.Pixel} West):** Establishes the position on the map of the ChangeOrientation buttons.
- **setSize({Urbex.Size} Background, {Urbex.Size} North, {Urbex.Size} South, {Urbex.Size} Ortho, {Urbex.Size} East, {Urbex.Size} West):** Sets the size of the ChangeOrientation buttons.
- **setImage ({String} Background, {String} North, {String} South, {String} Ortho, {String} East, {String} West):** Allows to define the image of the ChangeOrientation buttons when they are not selected.
- **setImageSelect({String} Background, {String} North, {String} South, {String} Ortho, {String} East, {String} West):** Sets the image of the ChangeOrientation buttons when selected.

- **setTitle**({String} North, {String} South, {String} Ortho, {String} East, {String} West): Sets the tip shown on the ChangeOrientation buttons when mouse hovers.
 - **Background:** background image of the ChangeOrientation control
 - **North:** North button of the control
 - **South:** South button of the control
 - **Ortho:** Ortho button of the control
 - **East:** East Button of the control
 - **West:** West Button of the control

7.5.8 Urbex.Control.LayerSwitcher

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the LayerSwitcher control.
- **setSize**({Urbex.Size} size): Sets the size of the LayerSwitcher images.
- **setImageOpen**({String} url): Allows to define the image of the LayerSwitcher image that opens the control.
- **setImageClose**({String} url): Allows to define the image of the LayerSwitcher image that closes the control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the LayerSwitcher image.

7.5.9 Urbex.Control.ReturnPoint

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the ReturnPoint control.
- **setSize**({Urbex.Size} size): Sets the size of the ReturnPoint images.
- **setImageOpen**({String} url): Sets the image of the ReturnPoint image that open the control.
- **setImageClose**({String} url): Sets the image of the ReturnPoint image that close the control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the ReturnPoint image.

7.5.10 Urbex.Control.GetMap

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the GetMap control.
- **setSize**({Urbex.Size} size): Establishes the size of the GetMap image.
- **setImage** ({String} url): Sets the image of the GetMap image control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the GetMap control.

7.5.11 Urbex.Control.Length

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the Length control.
- **setSize**({Urbex.Size} size): Sets the size of the Length images.
- **setImageOn**({String} url): Allows to define the image of the Length control used when the control is turned on.
- **setImageOff**({String} url): Allows to define the image of the Length control used when the control is turned off.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Length control.

7.5.12 Urbex.Control.Area

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the Area control.
- **setSize**({Urbex.Size} size): Establishes the size of the Area images.
- **setImageOn**({String} url): Sets the image of the Area control used when the control is turned on.
- **setImageOff**({String} url): Sets the image of the Area control used when the control is turned off.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Area control.

7.5.13 Urbex.Control.Height

- **setPosition({Urbex.Pixel} position):** Sets the position on the map of the Height control.
- **setSize({Urbex.Size} size):** Allow to define the size of the Height images.
- **setImageOn(url):** (url) {String}, Sets the image of the Height control used when the control is turned on.
- **setImageOff({String} url):** Establishes the image of the Height control used when the control is turned off.
- **setImageInactive({String} url):** Establishes the image of the Height control used when the control is not active.
- **setTitle({String} tip):** Sets the tip to show when the mouse hovers the Height control.

7.5.14 Urbex.Control.Bearing

- **setPosition({Urbex.Pixel} position):** Establishes the position on the map of the Bearing control.
- **setSize({Urbex.Size} size):** Sets the size of the Bearing images.
- **setImageOn({String} url):** Sets the image of the Bearing control used when the control is turned on.
- **setImageOff({String} url):** Allows to define the image of the Bearing control used when the control is turned off.
- **setTitle({String} tip):** Sets the tip to show when the mouse hovers the Bearing control.

7.5.15 Urbex.Control.Elevation

- **setPosition({Urbex.Pixel} position):** Establishes the position on the map of the Elevation control.
- **setSize({Urbex.Size} size):** Sets the size of the Elevation images.
- **setImageOn({String} url):** Allows to define the image of the Elevation control used when the control is turned on.
- **setImageOff({String} url):** Allows defining the image of the Elevation control used when the control is turned off.

- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Elevation control.

7.5.16 Urbex.Control.Eraser

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the Eraser control.
- **setSize**({Urbex.Size} size): Sets the size of the Eraser images.
- **setImage** ({String} url): Allows to define the image of the Eraser control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Eraser control.

7.5.17 Urbex.Control.North

- **setPosition**({Urbex.Pixel} position, {Urbex.Pixel} arrow): Establishes the position of the North control on the map, and the position of the arrow image inside the control.
- **setSize**({Urbex.Size} size, {Urbex.Size} Arrow): Sets the size of the control and the size of the arrow image.
- **setImage** ({String} Background, {String} Arrow): Allows to define the images of the North control. The Arrow url is a pattern to find the images. Thirty five arrow images must exist; the images have to be accessible by appending _1 ... _35 to the image's name (i.e. img/arrow.png, img/arrow_1.png ... img/arrow_35.png). Derived images are the result of rotating the base image n*10 degrees clockwise (i.e. img/arrow.png [0 degrees], img/arrow_1.png [10 degrees], img/arrow_2.png [20 degrees] ... img/arrow_35.png [350 degrees]). All images must be both in PNG and GIF format to ensure IE6 compatibility.
- **setTitle** ({String} tip): Sets the tip to show when the mouse hovers the North control.

7.5.18 Urbex.Control.Scale

- **setPosition**({Urbex.Pixel} position): Establishes the position of the Scale control on the map.
- **setTitle** ({String} tip): Sets the tip to show when the mouse hovers the Scale control.

7.5.19 Urbex.Control.Metadata

- **setPosition**({Urbex.Pixel} position): Establishes the position of the Metadata control on the map.
- **setTitle** ({String} tip): Sets the tip to show when the mouse hovers the Metadata control.
- **setFormat** (String format): Sets the format in which the control will display the date of the image showing. Default format is DD/MM/YY. Available values for String *format* are:
 - **'year'**: Only the year is displayed, in format YYYY
 - **'month'**: Only month and year are displayed, in format MM/YYYY
 - Any other value or null will be interpreted as default format.

7.5.20 Urbex.Control.ShowStreets

- **setPosition**({Urbex.Pixel} position): Establishes the position of the ShowStreets control on the map.
- **setSize**({Urbex.Size} size): Sets the size of the ShowStreets images.
- **setImageOn**({String} url): Allows to define the image used in the ShowStreets control when it is turned on.
- **setImageOff**({String} url): Allows to define the image used in the ShowStreets control used when it is turned off.
- **setTitle** ({String} tip): Sets the tip to show when the mouse hovers the ShowStreets control.
- **setLayerName** ({String} tip): Sets the name of the layer with the overlay information. It is **Streets** by default.

7.5.21 Urbex.Control.MultiOblique

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the MultiOblique control.
- **setSize**({Urbex.Size} size): Sets the size of the MultiOblique icon image.

BlomURBEX™ Javascript API v5.7 r1.0

- **setImageOpen**({String} url): Allows to define the image of the MultiOblique control used to open the control.
- **setImageClose**({String} url): Allows to define the image of the MultiOblique control used to close the control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the MultiOblique image.

7.5.22 Urbex.Control.Year

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the Year control.
- **setSize**({Urbex.Size} size): Sets the size of the Year icon image.
- **setImageOpen**({String} url): Allows to define the image of the Year control used to open the control.
- **setImageClose**({String} url): Allows to define the image of the Year control used to close the control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Year image.

7.5.23 Urbex.Control.ReverseGeocoding

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the control.
- **setSize**({Urbex.Size} size): Sets the size of the control images.
- **setImageOpen**({String} url): Sets the image of the image that opens the control.
- **setImageClose**({String} url): Sets the image of the image that closes the control.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the ReturnPoint image.

7.5.24 Urbex.Control.Baselayer

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the control.
- **setButtonPosition**({Urbex.Pixel} position): Establishes the position on the map of the button of the control.

- **setSize({Urbex.Size} size):** Sets the size of the control.
- **setButtonSize({Urbex.Size} size):** Sets the size of the button of the control.
- **setImageOpen({String} url):** Sets the image of the image that opens the control.
- **setImageClose({String} url):** Sets the image of the image that closes the control.
- **setTitle({String} tip):** Sets the tip to show when the mouse hovers the ReturnPoint image.

7.5.25 Urbex.Control.Overlays

- **setPosition({Urbex.Pixel} position):** Establishes the position on the map of the control.
- **setButtonPosition({Urbex.Pixel} position):** Establishes the position on the map of the button of the control.
- **setSize({Urbex.Size} size):** Sets the size of the control.
- **setButtonSize({Urbex.Size} size):** Sets the size of the button of the control.
- **setImageOpen({String} url):** Sets the image of the image that opens the control.
- **setImageClose({String} url):** Sets the image of the image that closes the control.
- **setTitle({String} tip):** Sets the tip to show when the mouse hovers the ReturnPoint image.

7.5.26 Urbex.Control.Diagonal

- **setPosition({Urbex.Pixel} position):** Sets the position on the map of the Height control.
- **setSize({Urbex.Size} size):** Allow to define the size of the Height images.
- **setImageOn(url):** (url) {String}, Sets the image of the Height control used when the control is turned on.

- **setImageOff**({String} url): Establishes the image of the Height control used when the control is turned off.
- **setImageInactive**({String} url): Establishes the image of the Height control used when the control is not active.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the Height control.

7.5.27 Urbex.Control.VerticalArea

- **setPosition**({Urbex.Pixel} position): Establishes the position on the map of the Vertical Area control.
- **setSize**({Urbex.Size} size): Establishes the size of the Vertical Area images.
- **setImageOn**({String} url): Sets the image of the Vertical Area control used when the control is turned on.
- **setImageOff**({String} url): Sets the image of the Vertical Area control used when the control is turned off.
- **setImageInactive**({String} url): Establishes the image of the Vertical Area control used when the control is not active.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the **Vertical** Area control.

7.5.28 Urbex.Control.LayerSelector

- **setPosition**({Urbex.Pixel} position): Sets the position on the map of the control.
- **setSize**({Urbex.Size} size): Allow to define the size of the Height images.
- **setImageOpen**(url): (url) {String}, Sets the image of the Height control used when the control is turned on.
- **setImageClose**({String} url): Establishes the image of the Height control used when the control is turned off.
- **setTitle**({String} tip): Sets the tip to show when the mouse hovers the control.

7.5.29 ZoomBar and ZoomOut controls

The zoom bar dynamically resizes itself to represent the set of zoom levels available each moment. So the total height of the control changes when commuting between ortho and oblique views or if the `setZoomLimits` method is called. The ZoomBar Control can update the position of a control after any resize, so that the ZoomOut control can be always attached to it (like default urbex controls).

The property **controlZoomOut** of the ZoomBar control handles a reference to an attached control. If it is set, the ZoomBar control will update the position of the attached control whenever it resizes itself. This control can be any of the available controls or one created by the developer.

The next example will attach the north control to the bottom of the ZoomBar instead of the zoomOut control:

```
var vie = new Urbex.Control.ChangeView2();
var nav = new Urbex.Control.Navigation();
var bar = new Urbex.Control.ZoomBar();
var nor = new Urbex.Control.North();

var newMap = new Urbex.Map('divName', {
  usertoken: 'usertoken',
  x: -523121,
  y: 4960371,
  controls: [vie, nav, bar, nor],
  loadcontrols: 'no'});

vie.setControlPosition(new Urbex.XY(30, 10));
bar.setControlPosition(new Urbex.XY(10, 10));
nor.setControlPosition(new Urbex.XY( 3, 163));
bar.controlZoomOut = nor;
```

7.6 Adding custom controls to the map

There are two ways to create custom controls to use within the map.

Note: For an in-depth explanation of the `Urbex.Handler` objects please refer to the **Handlers** chapter.

7.6.1 Adding a custom control with the `Urbex.Util.extend()` method

The next code fragment is a quick example of how to create and add a custom control to the map using the **`Urbex.Util.extend()`** method. This

control intercepts shift-mouse drag to display the extent of the bounding box dragged out by the user.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -521345,
  y: 4960372,
  loadcontrols: 'no'});

var control = new Urbex.Control();
Urbex.Util.extend(control, {
  draw: function () {
    // this Handler.Box will intercept the shift-mousedown
    this.box = new Urbex.Handler.Box(
      control,
      {'done': this.notice},
      {'keyMask': Urbex.Handler.MOD_SHIFT});
    this.box.activate();
  },
  notice: function (bounds) {
    var xymin = newMap.getLonLatFromPixel(
      new Urbex.Pixel(bounds.left, bounds.bottom));
    var xymax = newMap.getLonLatFromPixel(
      new Urbex.Pixel(bounds.right, bounds.top));
    alert(
      xymin.x.toFixed(4) + ', ' +
      xymin.y.toFixed(4) + ', ' +
      xymax.x.toFixed(4) + ', ' +
      xymax.y.toFixed(4));
  }
});

newMap.addControl(control);
```

7.6.2 Adding custom controls inheriting from Urbex.Control class

The next sample shows how to create and add a custom control to the map using **Urbex.Control** class. Control is the same as above.

First, create a JavaScript file, for example: ModShift.js, and add the next code.

```
Urbex.Control.ModShift = Urbex.Class(Urbex.Control, {
  box: null,
  initialize: function(options) {
    Urbex.Control.prototype.initialize.apply(this, arguments);
  },
```

```
draw: function () {
    this.box = new Urbex.Handler.Box(
        this,
        {'done': this.notice},
        {'keyMask': Urbex.Handler.MOD_SHIFT});
    this.box.activate();
},
notice: function (bounds) {
    var xymin = this.map.getLonLatFromPixel(
        new Urbex.Pixel(bounds.left, bounds.bottom));
    var xymax = this.map.getLonLatFromPixel(
        new Urbex.Pixel(bounds.right, bounds.top));
    alert(
        xymin.x.toFixed(4) + ', ' +
        xymin.y.toFixed(4) + ', ' +
        xymax.x.toFixed(4) + ', ' +
        xymax.y.toFixed(4));
},
CLASS_NAME: "Urbex.Control.ModShift"
});
```

Then, add the reference to the JavaScript file in the HTML page.

```
<script src="./ModShift.js"></script>
```

Finally, create the control and add this to the map.

```
var newMap = new Urbex.Map('divName', {
    usertoken: 'stringToken',
    x: -521345,
    y: 4960372,
    loadcontrols: 'no'});
var control = new Urbex.Control.ModShift();
newMap.addControl(control);
```

7.6.3 Developing custom controls

7.6.3.1 *The Urbex.Control class*

The **Urbex.Control** class is the base class for all controls.

7.6.3.2 *Main properties*

- **div**: A DOMELEMENT object. This is the div tag where all graphical elements, if any, are placed.

- **map:** A **Urbex.Map** object. This is the map object where the control has been added in.
- **buttons:** A set of **DOMElement** objects nested in the **div** element. These are graphic elements added to the control.

7.6.3.3 *Layout properties*

- **controlPosition:** The position of the control inside the map. Negative values represent left or bottom alignments.
- **controlSize:** The size of the control. This property is used to calculate the position of the control and the relative positions of the buttons.
- **sizes:** The sizes of each one of the buttons in the control. This property is used to calculate the positions of the buttons.
- **positions:** The relative positions of the buttons inside the control. Negative values represent right or bottom alignments.

7.6.3.4 *Drawing properties*

- **controlTitle:** A tip about the control.
- **titles:** Tips about the buttons.
- **imagesOn:** Set of images for the buttons representing an **on** state,
- **imagesOff:** Set of images for the buttons representing an **off** state,

7.6.3.5 *Methods*

The methods of the **Urbex.Util.Control** class are:

- **initialize(options):** The constructor method.
- **destroy():** The destructor method.
- **draw():** The method called the first time the control is drawn.
- **addDivButton():** Adds a **div** button to the control.
- **addImageButton():** Adds a image button to the control.
- **refresh():** Method to redraw the control, this method calculates the real positions of the control and the buttons, and updates some properties of the control, then calls **customRefresh**.

BlomURBEX™ Javascript API v5.7 r1.0

- **customRefresh(buttons, positions, sizes, titles, imagesOn, imagesOff):** Method used to redraw the control.
- **hide():** Method to hide the control.
- **show():** Method to make the control reappear if it was hidden.

Here is a complete HTML example of how to use hide() and show() methods. In this example, one control is added to the map, to check the effect of invoking show() and hide() on that control.

There will be two buttons available in this example: 'Hide' button to hide the control and 'Show' button to make it reappear:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Urban Explorer Change Map Location</title>

    <script src="http://www.blomurbex.com/api/5.7/Urbex.js"></script>
    <script type="text/javascript">

      var newMap;
      var control = new Urbex.Control.Length();

      function init() {
        newMap = new Urbex.Map('map', {
          usertoken: 'stringToken',
          loadcontrols: 'no',
          x: -523121,
          y: 4960371,
          zoom: 3,
          view: 'oblique',
          orientation: 'west'});

        newMap.addControl(control);
      }
      function hideControl() {
        control.hide();
      }

      function showControl() {
        control.show();
      }
    </script>
  </head>
</html>
```

```
</script>
</head>

<body onload="init()">
  <div style="position: relative; width: 800px; height: 550px; top:
0px; left: 0px; z-index: 1" id="map"></div>
  <input type="button" id="boton" value="Hide"
onclick="javascript:hideControl();" />
  <input type="button" id="boton" value="Show"
onclick="javascript:showControl();" />
</body>
</html>
```

7.6.3.6 *The Urbex.Control life cycle*

The **Urbex.Control** basic life cycle is represented in the next ordered list:

1. initialize
2. setMap
3. draw
4. activate
5. deactivate
6. destroy

It is important to notice that before calling the draw method it is necessary to have the map property set.

7.6.3.7 *BoxArea Example*

For a best comprehension of how to develop a control, the following example provides a functional example. The developed control in this example will allow the user to obtain the area of rectangular surfaces.

```
// define the project namespace
Project = {}

Project.BoxArea = Urbex.Class(Urbex.Control, {

  // the status is modified by the map when
  // offMeasureControls is called
  // a. offMeasureControls is called in ORTHO view
```

BlomURBEX™ Javascript API v5.7 r1.0

```
// status is set to 'OFF'
// b. offMeasureControls is called in OBLIQUE view
// status is set to 'INACTIVE'
status: 'OFF',

/** Constructor */
initialize: function(options) {

    // 1. extend
    Urbex.Util.extend(this, options);

    // 2. call the parent constructor
    Urbex.Control.prototype.initialize.apply(
        this, arguments);

    // 3. initialize all properties

    // a. location
    this.controlSize = new Urbex.Size(24, 24);
    this.controlPosition = new Urbex.Pixel(-1, 0);

    // b. buttons titles
    this.titles = [];
    this.titles[0] = 'BoxArea Measurement';

    // c. buttons sizes
    this.sizes = [];
    this.sizes[0] = this.size ?
        this.size : new Urbex.Size(24, 24);

    // d. buttons location inside the control
    this.positions = [];
    this.positions[0] = this.position ?
        this.position : new Urbex.Pixel(0, 0);

    // e. buttons images
    this.imagesOn = [];
    this.imagesOn[0] = this.imageon ?
        this.imageon :
            Urbex.Util.getImagesLocation() + "length_on.png";

    // f. buttons images
    this.imagesOff = [];
    this.imagesOff[0] = this.imageoff ?
        this.imageoff :
            Urbex.Util.getImagesLocation() + "length_off.png";
```

```
},

/** Destructor */
destroy: function() {

    // 1. call the parent destructor
    Urbex.Control.prototype.destroy.apply(this, arguments);

    // 2. do your stuff, nothing in this example
},

/** First draw */
draw: function() {

    // 1. call the parent's draw method
    Urbex.Control.prototype.draw.apply(this, []);

    // 2. initialize the buttons
    this.buttons = [];

    //    a. create the button
    var button = this.addImageButton(
        this.CLASS_NAME + '_button',
        this.titles[0],
        this.imagesOn[0],
        this.positions[0],
        this.sizes[0]);

    //    b. register for Browser events
    var bind = Urbex.Function.bindAsEventListener;
    Urbex.Event.observe(
        button,
        "mousedown",
        bind(this.changeState, this));
    Urbex.Event.observe(
        button,
        "dblclick",
        bind(this.cancelEvent, this));
    Urbex.Event.observe(
        button,
        "click",
        bind(this.cancelEvent, this));

    // 3. register for map events

    // 4. do your stuff
```


BlomURBEX™ Javascript API v5.7 r1.0

```
this.handler = new Urbex.Handler.Box(
    this, this, { keyMask: null });

this.vector = new Urbex.Layer.Vector('BoxArea Vectors');
this.text = new Urbex.Layer.Texts('BoxArea Text');

this.map.addLayer(this.vector);
this.map.addLayer(this.text);

// 5. refresh the control
this.refresh();

// 6. always return this.div
return this.div;
},

/** activate/desactivate */
changeState: function(eventInfo) {

    // 1. check the event info
    if (eventInfo != null) {
        if (!Urbex.Event.isLeftClick(eventInfo)) {
            return;
        }
        Urbex.Event.stop(eventInfo);
    }

    // 2. do your stuff
    if (this.status != 'ON') {
        // Deactivates any Active Measure control.
        this.map.getMeasurementObj().offMeasureControls();

        this.status = 'ON';
        this.handler.activate();
    } else {

        this.status = 'OFF';
        this.handler.deactivate();
    }

    // 3. refresh the control
    this.refresh();
},

/** just not propagate the event */
cancelEvent: function(eventInfo) {
```

BlomURBEX™ Javascript API v5.7 r1.0

```
Urbex.Event.stop(eventInfo);
},

/** box handler event */
done: function(result) {

    var map = this.map;
    var text = this.text;
    var vector = this.vector;

    var callback = function(xyList) {

        var points = [];
        for (var i in xyList) {
            points[i] =
                new Urbex.Geometry.Point(
                    xyList[i].x,
                    xyList[i].y);
        }

        var area = 0;

        var areaCallback = function(map, value) {
            text.addText(
                xyList[0],
                'myTextId',
                'Area (m<sup>2</sup><br/>' +
                    Math.round(value*1000) / 1000.0,
                {
                    fontFamily: 'Verdana',
                    fontWeight: 'bolder',
                    fontSize: '10pt',
                    color: 'rgb(0,0,0)',
                    url: null,
                    size: new Urbex.Size(1000, 1000)
                });
        };

        map.getMeasurementObj().getArea(points, areaCallback);

        var lines = new Urbex.Geometry.LinearRing(points);
        var polygon = new Urbex.Geometry.Polygon(lines);

        vector.addFeatures([new Urbex.Feature.Vector(polygon)]);
    };
};
```

BlomURBEX™ Javascript API v5.7 r1.0

```
var points = [
    new Urbex.XY(result.left, result.top),
    new Urbex.XY(result.left, result.bottom),
    new Urbex.XY(result.right, result.bottom),
    new Urbex.XY(result.right, result.top)
];

Urbex.Util.GetWorldCoordinates(this.map, points, callback);
},

/** refresh */
customRefresh: function(
    buttons,
    positions,
    sizes,
    titles,
    imagesOn,
    imagesOff) {

    // 1. update general display
    buttons[0].title = titles[0];

    buttons[0].style.top = positions[0].y;
    buttons[0].style.left = positions[0].x;

    buttons[0].style.width = sizes[0].w + 'px';
    buttons[0].style.height = sizes[0].h + 'px';

    switch(this.status) {
        case 'ON':
            buttons[0].getElementsByTagName('img')[0].src = imagesOn[0];
            break;
        case 'OFF':
        case 'INACTIVE':
        default:
            buttons[0].getElementsByTagName('img')[0].src = imagesOff[0];
    }
},

/** class name */
CLASS_NAME: "Project.BoxArea"
});
```

8 Markers

Markers can be added to the map using a special layer object, the **Urbex.Layer.Markers** layer.

8.1 Adding Markers

8.1.1 Using Marker layers

The first step for adding a Markers layer to the Map is to create an **Urbex.Layer.Markers** layer. When creating this kind of layers, it will be necessary to give them a name.

```
var poi = new Urbex.Layer.Markers('Points of Interest');
```

Once it has been created it must be added to the map using the **addLayer** method of the **Urbex.Map**.

- **addLayer** (Urbex.Layer): add a new layer to the map.

```
newMap.addLayer(poi);
```

The layer has a few options to customize the markers behavior.

- **minZoomLevel** (optional): {Integer}, the minimum level of zoom in which the marker will be visible. By **default** this value is **15**.
- **maxZoomLevel** (optional): {Integer}, the maximum level of zoom in which the marker will be visible. By **default** this value is **20**.

So a little more complex layer could be:

```
var poi = new Urbex.Layer.Markers(  
  'Points of Interest',  
  {minZoomLevel: 12, maxZoomLevel: 16});
```

With this, the layer is created and added to the map but is empty. It will be necessary to add Markers to the Markers Layer.

There are two available factory methods in the **Urbex.Layers.Markers** class that allow creating and adding Markers:

- **addMarker** (point, id, options): {Urbex.Marker} To add one **Marker** to the **Markers** layer.

BlomURBEX™ Javascript API v5.7 r1.0

- **point** (mandatory): {Urbex.XY}, position where the maker will be created
- **id** (mandatory): {String}, id of the marker
- **options**:
 - **description** (optional): {String}, description of the marker, accepts HTML code.
 - **url** (optional): {String}, URL that will be called when there is a mouse click on the Marker.
 - **Icon** (optional): {Urbex.Icon}, image that will be used to display the marker
 - **title** (optional): {String}, marker's tooltip.
 - **offsetX** (optional): {Integer}, horizontal offset between the marker and its popup.
 - **offsetY** (optional): {Integer}, vertical offset between the marker and its popup.
 - **popActive** (optional): {Boolean}, flag that determines if a popup will be displayed when the marker is clicked.
 - **minZoom** (optional): {Integer}, the minimum level of zoom in which the marker will be visible.
 - **maxZoom** (optional): {Integer}, the maximum level of zoom in which the marker will be visible.
 - **cursor** (optional): {string}: CSS string for the cursor to be displayed when mousing over the marker
- **addMarkers** (points, id, options): {Array<Urbex.Marker>}. To add a collection of **Markers** to the **Markers** layer.
 - **points** (mandatory): {Array(Urbex.XY)}, positions where the markers will be created
 - **id** (mandatory): {String}, id of the markers collection
 - **options**:

BlomURBEX™ Javascript API v5.7 r1.0

- **description** (optional): {String}, description of the markers in the collection, accepts HTML code.
- **url** (optional): {String}, URL that will be called when there is a mouse click on the Marker
- **icon** (optional): {Urbex.Icon}, image that will be used to display all the markers in the collection
- **title** (optional): {String}, marker's tooltip.
- **offsetX** (optional): {Integer}, horizontal offset between the marker and its popup.
- **offsetY** (optional): {Integer}, vertical offset between the marker and its popup.
- **popActive** (optional): {Boolean}, flag that determines if a popup will be displayed when the marker is clicked.
- **minZoom** (optional): {Integer}, the minimum level of zoom in which the marker will be visible.
- **maxZoom** (optional): {Integer}, the maximum level of zoom in which the marker will be visible.
- **cursor** (optional): {string}: CSS string for the cursor to be displayed when mousing over the marker.

If the **description**, the **URL** of the **title** is missing or null, the Markers won't have description, URL or title properties.

If no **Icon** is specified, then a default image will be used to represent the Marker.

Default values for the other properties are:

- offsetX: 0
- offsetY: 0
- popActive: true
- minZoom: null
- maxZoom: null

BlomURBEX™ Javascript API v5.7 r1.0

The null value of **minZoom** and **maxZoom** means that these properties are not used. Both minZoom and maxZoom must be set in order to enable this characteristic.

To decide if a marker is visible at a zoom level, first the layer is checked. If the layer is visible at that zoom level then the marker settings are checked to see if the marker has to be displayed

The icons images used to represent the markers are objects of type **Urbex.Icon** and are created from the URL which points to where the image for the markers is available.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var poi = new Urbex.Layer.Markers('Points of Interest');
newMap.addLayer(poi);

var im = 'http://www.blomurbex.com/api/5.7/img/marker-blue.png';
var sz = new Urbex.Size(50,50);
var icon = new Urbex.Icon(im, sz);

var position = new Urbex.XY(-523121, 4960371);

var id = 'id';

var html = '<h1>UrBex</h1>';

poi.addMarker(position, id, {description: html, icon: icon});
```

Both methods **addMarker** and **addMarkers** return the references of the created markers. Saving these references or getting them through the **getMarkersById(id): {Array<Urbex.Marker>}** method will allow to change the markers' properties. The following functions will enable the properties change.

- **setUrl(url)**
- **setTitle(title)**
- **setOffsetX(offset)**
- **setOffsetY(offset)**
- **setPopActive(active)**

- **setMinZoom**(zoom)
- **setMaxZoom**(zoom)

After changing any property of any marker it is advisable to refresh the markers layer to actually update the current view using the **refresh** function.

```
poi.refresh();
```

8.1.1.1 **Example code:**

1. creates and adds a 'Pharmacies' layer with just one marker with a description, an url and an icon.
2. creates and adds a second markers layer called 'Points of Interest' with six markers sharing the same description and url, and it's positions are displayed by using the default icon

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var pharm = new Urbex.Layer.Markers('Pharmacies');
newMap.addLayer(pharm);

var icon = new Urbex.Icon(
  'http://www.blomurbex.com/api/5.7/img/marker-blue.png');

pharm.addMarker(
  new Urbex.XY(-523800, 4960392),
  'id1',
  {
    description: 'description',
    url: 'http://www.blomasa.com',
    icon: icon
  });

var poi = new Urbex.Layer.Markers('Points of Interest');
newMap.addLayer(poi);

var points = [
  new Urbex.XY(-523121, 4960371),
  new Urbex.XY(-523311, 4960364),
  new Urbex.XY(-522909, 4960381),
  new Urbex.XY(-523421, 4960359),
```

```
new Urbex.XY(-522830, 4960341),
new Urbex.XY(-523501, 4960351)];
poi.addMarkers(points, 'id2');
```

8.1.2 Using TextMarkers layers

The second method for adding a Markers layer to the Map is to create an **Urbex.Layer.TextMarkers**. This layer creates markers from data provided in a text file.

The file used is a tsv file (each field in a line is separated by a tab character, and each registry is separated by a carriage return).

- The first line of the file must contain a header with the fields' names. The order of the fields is fixed, so this line is only for readability purposes.
- The file cannot contain any comment or blank line.

Each line must give a value to all the fields.

The field order is:

1. Latitude. (In Spherical Mercator)
2. Longitude. (In Spherical Mercator)
3. Title.
4. Description.
5. Size of the icon.
6. Offset of the icon's center. (see 8.1.3)
7. Image of the icon.

8.1.2.1 **Example text file:**

Lat	lon	title	description	iconSize	iconOffset	icon
4960364	-523311	title	description	21,25	-10,-25	marker.png
4960381	-522909	title	html	21,25	-10,-25	marker.png

8.1.2.2 **Example code:**

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y:4960371});
```

```
var markers = new Urbex.Layer.TextMarkers(
    'text',
    { location: './08.1.2.1 - a - marker tsv file.markers' });
newMap.addLayer(markers);
```

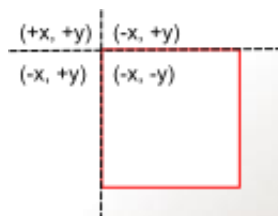
8.1.3 Icons

Urbex.Icon class represents an image. These icons are used by the markers to point to locations on the map.

The icon is an opaque class that can be instantiated and used as parameter in the Urbex API, nothing else can be done with the icons.

The icon constructor has four parameters:

- **url:** {String} Url of the image.
- **size:** {Urbex.Size} Size of the image.
- **offset:** {Urbex.Pixel} Position of the hot-spot on the images. The point of the image described by this parameter will be used to point to a location of the map. The image below this paragraph illustrates the reference system for setting the offset. The default offset is the center of the image.



Some examples of offsets:

- **Center (default offset):** new Urbex.Pixel(-width/2, -height/2);
- **Upper left corner:** new Urbex.Pixel(0, 0);
- **Lower right corner:** new Urbex.Pixel(-width, -height);

8.2 Deleting Markers

Markers can be deleted by providing their Ids, so a group of Markers sharing the same ID will be treated as a unit.

8.2.1 removeMarker

The **removeMarker** method from the **Urbex.Layer.Markers** Class is used for Marker removal.

- **removeMarker(id):** {String}, removes a marker or collection or markers identified by the provided **id**.

Example:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var pharm = new Urbex.Layer.Markers('Pharmacies');
newMap.addLayer(pharm);

var icon = new Urbex.Icon('http://www.blomurbex.com/api/5.7/img/marker-blue.png');
pharm.addMarker(new Urbex.XY(-523011, 4960371), 'id1', {
  description: 'description',
  url: 'http://www.blomasa.com',
  icon: icon
});
pharm.addMarker(new Urbex.XY(-523231, 4960371), 'id2', {
  description: 'description 2',
  url: 'http://www.blomasa.com'
})
pharm.removeMarker('id1');
```

8.2.2 removeLayer

To remove a **Markers** layer from the Map, use the **removeLayer** method in the **Urbex.Map** Class. This method takes as a parameter the layer that must be removed from the map.

```
var poi = new Urbex.Layer.Markers('Points of Interest');
newMap.addLayer(poi);
newMap.removeLayer(poi);
```

9 Vectors

BlomURBEX allows adding vector information to the map, thus superimposing customer or third parties data to the images and data provided.

To add vectors to the Map it is necessary to create a layer of type **vector** and add it to the map. Instances of **Urbex.Layer.Vector** are used to render vector data from a variety of sources.

```
var layer = new Urbex.Layer.Vector({String} name);
```

9.1 Construction

Following is an example for creating a vector layer:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523121,  
  y: 4960371});  
var vectorlayer = new Urbex.Layer.Vector('vectors');  
newMap.addLayer(vectorlayer);
```

9.2 Methods

Urbex.Layer.Vector has three public methods:

- **addFeatures({Array (Urbex.Feature.Vector)})**: Adds Features to the layer.
- **destroyFeatures({Array (Urbex.Feature.Vector)})**: Erases and destroys features on the layer.
- **getFeatureById({String} Id)**: Retrieve the feature identified by the Id (the Id equals to the feature.id property).

9.3 Features

The vector Features that need to be added to the vector layer for rendering are created by creating objects of the Class: **Urbex.Feature.Vector**. A vector feature uses the **Urbex.Geometry** classes as a description of its geometry, but a vector feature has also other properties besides its geometry:

```
var feature = new Urbex.Feature.Vector(  
  {Urbex.Geometry} geometry,  
  {Object} attributes,
```

```
{Object} style);
```

Where:

- **geometry:** {Urbex.Geometry} Geometry description.
- **attributes:** {Object} [Optional] object that holds arbitrary properties that describe the feature.
- **style:** {Object} [Optional] rendering style of the object.

9.4 Geometries

Geometry is a description of a geographic object. All geographic objects added to BlomURBEX will have to be specified in the Map base coordinate system, by default Spherical Mercator (EPSG:3785).

This is a base class, typical geometry types are described by subclasses of this class: Geometry Point, Geometry LineString, Geometry LinearRing and Geometry Polygon.

9.4.1 Geometry Point

Urbex.Geometry.Point Constructs a point geometry.

Example code:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var vectorlayer = new Urbex.Layer.Vector('vectors');
newMap.addLayer(vectorlayer);

var x = -523121;
var y = 4960371;

var geometry = new Urbex.Geometry.Point(x, y);
var feature = new Urbex.Feature.Vector(geometry);

vectorlayer.addFeatures([feature]);
```

9.4.2 Geometry LineString

Urbex.Geometry.LineString is a curve which can never be less than two points long.

Example code:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var vectorlayer = new Urbex.Layer.Vector('vectors');
newMap.addLayer(vectorlayer);

var x = -523171.16180;
var y = 4960396.08090;
var x2 = -523172.35613;
var y2 = 4960356.66806;

var points = [
  new Urbex.Geometry.Point(x, y),
  new Urbex.Geometry.Point(x2, y2)];

var geometry = new Urbex.Geometry.LineString(points);
var feature = new Urbex.Feature.Vector(geometry);

vectorlayer.addFeatures([feature]);
```

9.4.3 Geometry LinearRing

Urbex.Geometry.LinearRing is a special LineString which is closed. It closes itself automatically by adding a copy of the first point as the last point.

Linear rings are constructed with an array of **Urbex.Geometry.Point**. This array can represent a closed or open ring. If the ring is open (the last point does not equal the first point), the constructor will close the ring. If the ring is already closed (the last point does equal the first point), it will be left closed.

Example code:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});
```

```
var vectorlayer = new Urbex.Layer.Vector('vectors');
newMap.addLayer(vectorlayer);

var x = -523171.16180;
var y = 4960396.08090;
var x2 = -523172.35613;
var y2 = 4960356.66806;
var x3 = -523131.74896;
var y3 = 4960353.08507;

var points = [
  new Urbex.Geometry.Point(x, y),
  new Urbex.Geometry.Point(x2, y2),
  new Urbex.Geometry.Point(x3, y3),
  new Urbex.Geometry.Point(x, y)];

var geometry = new Urbex.Geometry.LinearRing(points);
var feature = new Urbex.Feature.Vector(geometry);

vectorlayer.addFeatures([feature]);
```

9.4.3.1 *Properties*

- **getArea()**: calculates the enclosed area of the closed linearRing

```
var feature = new Urbex.Feature.Vector(geometry);
var area = feature.getArea();
```

9.4.4 Geometry Polygon

Urbex.Geometry.Polygon Constructs a polygon geometry. A polygon is a collection of **Urbex.Geometry.LinearRings**.

Example code:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var vectorlayer = new Urbex.Layer.Vector('vectors');
newMap.addLayer(vectorlayer);

var x = -523171.16180;
var y = 4960396.08090;
var x2 = -523172.35613;
```



```
var y2 = 4960356.66806;
var x3 = -523131.74896;
var y3 = 4960353.08507;
var x4 = -523116.22269;
var y4 = 4960410.41284;

var points = [
  new Urbex.Geometry.Point(x, y),
  new Urbex.Geometry.Point(x2, y2),
  new Urbex.Geometry.Point(x3, y3),
  new Urbex.Geometry.Point(x4, y4),
  new Urbex.Geometry.Point(x, y)];

var lines = new Urbex.Geometry.LinearRing(points);
var geometry = new Urbex.Geometry.Polygon(lines);
var feature = new Urbex.Feature.Vector(geometry);

vectorlayer.addFeatures([feature]);
```

9.5 Creation of Vector Features from WKT

It is possible to create Vector Features directly from a WKT stream. There is a formatter that allows transformation between WKT and vector features:

Urbex.Format.WKT

To work with the formatter it's necessary to implement an object of this class, and then call the **read** method of the object, passing a string containing the **WKT**:

```
var sampleWKT="POLYGON ((1399952.6816295157 7494570.845487378 0,
1399971.4180772903 7494551.292016115 0, 1399954.3430761702
7494535.574319827 0, 1399935.8159744523 7494553.695125455 0,
1399952.6816295157 7494570.845487378 0))";

// New Vector layer
var vecLayer = new Urbex.Layer.Vector('Vectors');
newMap.addLayer(vecLayer);

// Use the WKT-reader
var transformer = new Urbex.Format.WKT();
var feat = transformer.read(sampleWKT);

vecLayer.addFeatures(feat);
```

9.6 Styles

Urbex features can have a number of style attributes.

- **fillColor:** fill color of polygons and points, expressed as hexadecimal color values
- **fillOpacity:** fill opacity of polygons and points, from 0 transparent to 1 opaque
- **strokeColor:** edge color of polygons and points and fill color of lines, expressed as hexadecimal color values
- **strokeOpacity:** edge opacity of polygons and points and fill opacity of lines, from 0 transparent to 1 opaque
- **strokeWidth:** edge width of polygons and points and width of lines, defined in pixels
- **strokeLinecap:** termination type of the segments in polygons and lines, accepted values are 'round' for rounded termination and 'square' for angular termination
- **pointRadius:** point radius, defined in pixels
- **cursor:** type of cursor. 'crosshair' to cross pointer, 'pointer' to windows hand pointer, nothing to default windows pointer.
- **Title:** Message to add as a title to the feature vector. This message will show when mouse hovers the vector.

9.6.1 Default style

The 'default' style will be used if no other style is specified. This style is defined as follows:

- fillColor: "#ee9900"
- fillOpacity: 0.4
- strokeColor: "#ee9900"
- strokeOpacity: 1
- strokeWidth: 1
- strokeLinecap: "round"

- pointRadius: 6
- cursor: "crosshair"

9.6.2 Creating new styles

To create a new style use the **Urbex.Feature.Vector.Style** constant as follows:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var vectorlayer = new Urbex.Layer.Vector('vectors');
newMap.addLayer(vectorlayer);

var x = -523171.16180;
var y = 4960396.08090;
var x2 = -523172.35613;
var y2 = 4960356.66806;

var points = [
  new Urbex.Geometry.Point(x, y),
  new Urbex.Geometry.Point(x2, y2)];

var geometry = new Urbex.Geometry.LineString(points);

var style = {
  'userdefined': {
    fillColor: 'yellow',
    fillOpacity: 0.2,
    strokeColor: 'yellow',
    strokeOpacity: 1,
    strokeLinecap: 'round',
    strokeWidth: 4,
    pointRadius: 6,
    cursor: 'pointer',
    title: 'a vector with new style'
  },
}

var feature = new Urbex.Feature.Vector(
  geometry,
  null,
  style['userdefined']);

vectorlayer.addFeatures([feature]);
```

9.7 Editable Vector Layers

9.7.1 Concept

An editable vector layer is a particular kind of Vector Layer that allows the user of the map to modify the vector features by dragging their vertices around the map.

It works exactly the same way as a Vector layer, but its features are able to be modified and moved.

To use it, just instantiate `Urbex.Layer.VectorEditable` class instead of `Urbex.Layer.Vector`.

9.7.2 featuremodified event

The layer will fire a 'featuremodified' event everytime a feature is modified. Just by subscribing to this event, a developer can create a callback function that shall be called on every feature modification. Following code snippet explains how

```
var editableVectorLayer = new
Urbex.Layer.VectorEditable("BlomStreetMeasurementsLayer");

map.addLayer(editableVectorLayer);

var callback = function(evt) {
    var feature = evt.feature;
    //Do something with the new vector feature
}

editableVectorLayer.events.register("featuremodified", this, callback);
```

10 Text data

Texts can be added to the map through a special layer object, the **Urbex.Layer.Texts** layer.

10.1 Adding Texts

For adding a Texts layer to the Map it will be necessary to create an **Urbex.Layer.Texts** layer. When creating this kind of layers, it will be necessary to give it a name.

```
var titles = new Urbex.Layer.Texts('Labels');
```

Once it has been created it must be added to the map using the **addLayer** method of the **Urbex.Map**.

- **addLayer** (Urbex.Layer): add a new layer to the map.

```
newMap.addLayer(titles);
```

With this, the layer is created and added to the map but is empty. It will be necessary to add texts to the Texts Layer. This is done via the **Urbex.Texts** method:

- **addText** (point, id, text, options) To add one **Text** to the **Texts** layer.
 - **point** (mandatory): {Urbex.XY}, position where the object text will be created
 - **id** (mandatory): {String}, id of the object text
 - **text** (mandatory): {String}, text label of the object
 - **options**:
 - **fontFamily** (optional): {String}, font family of the text label, default value is 'Arial'
 - **fontWeight** (optional): {String}, Sets the weight of the font of the object, default value is '400'

Possible values:

- 'normal', normal characters
- 'bold', thick characters
- 'bolder', thicker characters
- 'lighter', lighter characters

BlomURBEX™ Javascript API v5.7 r1.0















- '100', '200', '300', '400', '500', '600', '700', '800', '900' (400 is the same as normal and 700 is the same as bold).
- **fontSize** (optional): {String}, Sets a value that indicates the font size used for text in the object, default is '10pt'

Possible values:

- xx-small, x-small, small, medium, large, x-large, xx-large (Default value is medium)
- smaller, smaller size than the parent element
- larger, larger size than the parent element
- length, fixed size
- %, percent of the parent element (relative units)
- px, floating-point number in pixels (relative units)
- pt, floating-point number in points' (absolute units)
- **color** (optional): {String}, sets the foreground colour of the object, default is 'rgb(0,0,0)', i.e. Black.

Possible values:

- colours are defined using a hexadecimal notation for the combination of Red, Green, and Blue colour values (RGB). The lowest value that can be given to one light source is 0 (hex #00). The highest value is 255 (hex #FF). For example: rgb(0,0,0) or '#000000' for black, rgb(255,255,255) or '#FFFFFF' for white.
- colours by name:

	aqua
	black
	blue
	fuchsia
	gray
	green
	lime
	maroon
	navy
	olive
	purple
	red
	silver
	teal

white

yellow

- **backgroundColor**(optional):{String}, sets the background colour of the object, by default it is set to 'transparent'.
- **url** (optional): {String}, URL that will be called when there is a mouse click on the Text.
- **relativePosition**: {String}, A code to define the alignment of the box that contains the text. This is a string of two characters, one for vertical alignment: 't' (top), 'l' (lower), 'm' (middle); and one for horizontal alignment: 'l' (left), 'r' (right), 'm' (middle). The default alignment is 'tl' (top-left).
- **offsetX**: {Integer}, A horizontal offset in pixels to correct the position of the text.
- **offsetY**: {Integer}, A vertical offset in pixels to correct the position of the text.

Example code:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'usertoken',
  x: -523600,
  y: 4960382});

var titles = new Urbex.Layer.Texts('Labels');
newMap.addLayer(titles);

titles.addText(
  new Urbex.XY(-523600, 4960382),
  'id0',
  '<b>Plaza de Toros</b>',
  {
    fontFamily: 'Arial',
    fontWeight: '300',
    fontSize: '10pt',
    color: 'rgb(0,0,0)',
    backgroundColor: 'rgb(200,200,200)',
    opacity: 0.6,
    url: 'http://www.blomasa.com',
    relativePosition: 'tl'
  });
```

```
titles.addText(  
  new Urbex.XY(-523600, 4960382),  
  'id1',  
  '<b>Plaza de Toros</b>',  
  {  
    fontFamily: 'Arial',  
    fontWeight: '300',  
    fontSize: '10pt',  
    color: 'rgb(0,0,0)',  
    backgroundColor: 'rgb(200,200,200)',  
    opacity: 0.6,  
    url: 'http://www.blomasa.com',  
    relativePosition: 'lr',  
    offsetX: 10,  
    offsetY: -10  
  });
```

10.2 Deleting Texts

Texts can be deleted by providing their Ids. The **removeText** method from the **Urbex.Layer.Texts** Class is used for this.

- **removeText(id):** {String}, removes a text identified by the provided **id**.

Example:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523121,  
  y: 4960371});  
  
var labels = new Urbex.Layer.Texts('Titles');  
newMap.addLayer(labels);  
  
labels.addText(new Urbex.XY(-523800, 4960392), 'id1', 'Plaza de Toros');  
labels.removeText('id1');
```

To remove a **Texts** layer from the Map, use the **removeLayer** method in the **Urbex.Map** Class. This method receives the layer that must be removed from the map as parameter.

```
var streets = new Urbex.Layer.Text('labels');  
newMap.addLayer(streets);  
newMap.removeLayer(streets);
```

11 Annotations

BlomURBEX Javascript API allows user to make annotations on top of the map, to mark positions, shapes or add text annotations to remarkable places.

Annotations can be either vectors (Points, lines or polygons) or texts, and their style is fully configurable.

Whenever the user creates some kind of annotation, the map will create an `Urbex.Layer.VectorTexts`, a layer capable of plotting vectors and writing text popups. **All these layers will group annotations of the same colour** (for example, all white vectors and all texts with the background white will belong to the same `VectorTexts` layer).

So, there will be as many layers as different colours of annotations the user creates on the map.

11.1 Annotations Object

All annotations operations inside the map are centralized on the annotations object. To get hold of a reference to the annotations object attached to the map, a method called **`getAnnotationsObj()`** shall be called.

```
var annObj = map.getAnnotationsObj();
```

This object offers a whole lot of methods to create annotations, erase annotations and change the colour of the annotations.

Important notice: to preserve functionality of annotations object, all colours passed to it (inside style objects, options object for texts or as string parameters) need to be in #RGB format).

11.1.1 Annotations Object Reference

All methods that may have any interest for developers using annotations object are described in this section.

- **`drawLine(style)`**: Activates a Handler in the map that allows the user to draw lines on it. It accepts a vector style object to configure the style of the lines to be drawn. This handler will be active until **`stopAnnotations()`** is called.
- **`drawPolygon(style)`**: Activates a Handler in the map that allows the user to draw polygons on it. It accepts a vector style object to

configure the style of the polygons to be drawn. This handler will be active until **stopAnnotations()** is called.

- **drawPoint(style):** Activates a Handler in the map that allows the user to draw points on it. It accepts a vector style object to configure the style of the points to be drawn. This handler will be active until **stopAnnotations()** is called.
- **drawText(text, options):** Activates a handler that allows the user to draw text popups with the text content stated by variable **text** by clicking in the map. This handler will be active until **stopAnnotations()** is called. It accepts an **options** object to configure the text popups such as the one described on section 10.
- **stopAnnotations():** Whenever any annotation handler is active in the map, it will deactivate it so map normal usage will be active again.
- **cleanAnnotations():** This method will clear all annotations in the map.
- **cleanAnnotationsOfParticularColor(color):** This method will clean all annotations of given color. Color will be the RGB definition (example: #FFFFFF).
- **changeColor(sourceColor, destinationColor)** This function will change the color of the annotations layer in the source color to the destination color. Thus, all vectors and background colors of the texts on this layer will change to destinationColor.
- **addAnnotationsItems(features, texts, color):** This function can be used for adding individual features or text to an existing annotations layers, the one in the colour stated by **color** parameter. If **features** or **texts** are not present, only the present parameters will be drawn. If there was no previous **color** annotations layer, it will be created.

12 Geocoding and reverse geocoding

This Javascript API allows users to geocode a certain address (that is, translate an address to its geographical coordinates) or do the opposite, get the address of a certain point (reverse geocoding).

In order to do that, the API provides a Javascript object `Urbex.Geocoder` which has functions to perform this geocoding and reverse geocoding operations.

12.1 Urbex.Geocoder

`Urbex.Geocoder` is the class users must instantiate in order to perform geocoding operations.

Users can choose between two different ways of using the object: They can create a standalone Geocoder javascript object or, if they created a map, it provides a Geocoder ready to use.

Users can choose to geocode or reverse geocode using different providers. The geocoding object will have a 'service' attribute that the users can set to choose which one to use.

12.1.1 Geocoding Providers

BlomURBEX allows users to use several different geocoding providers: OpenStreetMaps, OnYourMaps, Ubiest and TomTom

Each of these providers has its own particularities.

12.1.1.1 Free provider (OpenStreetMaps)

Every user with a valid user token for BlomURBEX services should be able to use this geocoding provider.

This provider will geocode a free form address, a string with the following structure:

Number street, municipality, country subdivision, country

Depending on the country the street address will be:

- Number Street
- Street Number

Do not include extra commas, since it will interfere with the string parsing and as a result a false geocoding can occur.

Geocoder's default configuration uses this provider (parameter 'service' = 'free').

This service is provided using OpenStreetMaps.

12.1.1.2 Ubiest Provider

Not all users will be allowed to use this geocoding provider. Users must request access to this service provider, since it is an extra to the basic services.

This provider works in a different way. It does not accept a free form address as a whole string. Instead, it accepts an structured address, with following fields separated by commas:

```
countryCode: 3-letter ISO country code
country: country name
region: Region inside the country
province: Province name.
city: city/town/municipality.
address: Street/Square/avenue ... name
number: number.
```

Not all parameters are required. The only compulsory parameters is '**city**', but the more parameters filled, the better accuracy for geocoding responses. Whenever a parameter is not present, the commas surrounding it must be kept in the string. Otherwise, Ubiest geocoding won't be able to parse the address. An example of valid address sentence can be:

```
var address = ",,italia,,,treviso,Via reginato,85"
```

To use this service, set the parameter 'service' to 'ubiest'.

12.1.1.3 Non-free provider: OYM

OnYourMap is another geocoding provider that can be used through BlomURBEX. Its usage is similar to the free provider, using a freeform address to geocode.

To use this provider, users have to set the 'service' property of the geocoder object to 'oym'.

12.1.1.4 Non-free provider: Tomtom

Tomtom is another extra-valued provider for geocoding. It works as OYM and OSM, by using a free form address.

To use this provider, developers have to set the 'service' property of the geocoder object to 'tomtom'.

12.1.2 Creating a Geocoder Standalone object

Users not creating a map can instantiate the class Urbex.Geocoder and have a fully functional Geocoder object. But, as this object is not synchronized with any map, users are forced to provide their BlomURBEX usertoken to the object in order to make it work.

This object also accepts a 'service' parameter indicating which geocoding service to use. Possible values for this parameter are: 'free' for free geocoding and 'provider1' for non-free geocoding (not all user will be allowed to use this provider).

```
var geocoder = new Urbex.Geocoder({  
  usertoken: 'token',  
  service: 'free'});
```

12.1.3 Using map's geocoder

When a map is created, it automatically creates and stores internally a geocoder object, that can be reached as in the example:

```
var geocoder = map.geocoder;
```

This geocoder will be synchronized with the map. That means that it will get the usertoken and the zoom level (necessary for reverse geocoding operations, as will be detailed later) from the map.

This geocoder object's default service configuration will be to use the free geocoding service. If the user desires to change this configuration, it can be done as follows

```
map.geocoder.service = 'oym';
```

12.1.4 Geocoding

Once an instance of Geocoder has been retrieved, users can use its function **geocode** to perform geocoding operations. This function accepts the following parameters:

BlomURBEX™ Javascript API v5.7 r1.0

- **Address:** A string containing the address to be geocoded by the free service. This string must be left empty when using the non-free service.
- **Callback:** A function that will be called when the geocoding information is retrieved. An object containing the response data will be addressed to the callback function. This object will be the result of evaluating a JSON like the following example:

```
{'name': 'buckingham palace',  
'candidates': [  
  {coordX:-15859.115116690808,coordY:6710370.827624246,address:"Buckingham  
Palace, Buckingham Palace Road, St James's, Westminster, City of  
Westminster, Greater London, SW1E 5EA, United Kingdom"},  
  
  {coordX:-16022.225441824945,coordY:6709839.559636481,address:"Buckingham  
Palace, Buckingham Palace Road, Pimlico, Westminster, City of  
Westminster, Greater London, SW1W 0PP, United Kingdom"},  
  
  {coordX:-16025.95464476652,coordY:6709818.745718166,address:"Buckingham  
Palace, Buckingham Palace Road, Pimlico, Westminster, City of  
Westminster, Greater London, SW1W 0PP, United Kingdom"},  
  
  {coordX:-15859.115116690808,coordY:6710370.827624246,address:"Buckingham  
Palace, Buckingham Palace Road, St James's, Westminster, City of  
Westminster, Greater London, SW1E 5EA, United Kingdom"},  
  
  {coordX:-15859.115116690808,coordY:6710370.827624246,address:"Buckingham  
Palace, Buckingham Palace Road, St James's, Westminster, City of  
Westminster, Greater London, SW1E 5EA, United Kingdom"}  
]  
}
```

So, callback function can access both attributes of this object.

- The attribute 'name' contains the address as it was requested.
- The attribute 'candidates' is an array containing all candidates to be the point attached to the address. Every item in that array is a javascript object with 3 attributes:
 - coordX: X coordinate (Mercator) for the point.
 - coordY: Y coordinate (Mercator) for the point.
 - Address: Whole address delivered by the geocoding service for that point.

BlomURBEX™ Javascript API v5.7 r1.0

If an error occurred, the response will be a string starting with 'ERROR' following with the error explanation. To check if the response went well, users can do the following:

```
if (!response || (""+response).indexOf('ERROR') != -1)) {  
    alert("Address Not Found");  
} else {  
    //Process response object  
}
```

This example defines a callback function that will set a map's centre to the point delivered as the first candidate for a geocoding request. This example uses the free geocoding service, but it is also valid for every provider working with freeform address.

```
var callback = function (response){  
  
    var newcenter = new Urbex.XY(response.candidates[0].coordX,  
response.candidates[0].coordY);  
  
    map.setView(newcenter);  
}  
  
map.geocoder.geocode("buckingham palace", callback);
```

12.1.5 Reverse geocoding

The function ***reverseGeocode*** from the object geocoder will do this task. It accepts 4 parameters:

- **X:** X coordinate (spherical Mercator) for the point to be reverse geocoded.
- **Y:** Y coordinate (spherical Mercator) for the point to be reverse geocoded.
- **Callback:** A callback function that will be called when the response is ready. A string with the whole address of the point will be supplied to this callback function.

If an error occurred, the response will be a string starting with 'ERROR' following with the error explanation. To check if the response went well, users can do the following:

```
if (!response || (""+response).indexOf('ERROR') != -1)) {  
    alert("Address Not Found");  
}
```



```
} else {  
    //Process response object  
}
```

- **Zoom (optional):** The zoom level for the reverse geocoding enquiry. Higher zoom level will deliver more detailed results (street, number), lower zoomlevel will deliver less detailed results (city, country). When using map's geocoder, this function will take map's zoom. If no zoom is provided in a Standalone geocoder, the higher zoom value will be the default value for the request.

This example shows how to reverse geocode a point and alert its address.

```
var callback = function (response){  
  
    alert(response);  
}  
  
map.geocoder.reverseGeocode(-15859.115116690808, 6710370.827624246,  
callback);
```



13 Routing

This Javascript API allows users to get the routes between two specific points selected on map, or between two textual addresses using various service providers.

In order to do that, the API provides a Javascript object `Urbex.Router` which has functions to perform these routing operations.

13.1 Urbex.Router

`Urbex.Router` is the class users must instantiate in order to perform routing operations.

Users can choose between two different ways of using the object: They can create a standalone Router javascript object or, if they created a map, it provides a Router ready to use (`map.router`), just the same way it happened with Geocoder object.

Users can choose to get the route using different providers. The router object will have a 'service' attribute that the users can set to choose which one to use.

13.1.1 Routing Providers

BlomURBEX allows users to use three different routing providers: Ubiest, OYM (On Your Map) and TomTom. Each of these providers has its own particularities. Users may or may not be able to use any of these providers depending on their permissions in BlomURBEX platform.

13.1.1.1 Ubiest

This provider will route a structured address, a string with the following structure:

`CountryCode,Country,Region,Province,City,Address,Street`

Ubiest interface also provides a user to select or specify two point coordinates as origin and destination to get the route.

13.1.1.2 OYM(On Your Map)

This provider will route a free form address.

OYM interface also provides a user to select or specify two point coordinates as origin(`x1,y1`) and destination(`x2,y2`) to get the route.

13.1.1.3 TomTom

TomTom routing works the same as OYM: It supports either a freeform address or two coordinate pairs.

13.1.2 Creating a Router Standalone object

Users not creating a map can instantiate the class `Urbex.Router` and have a fully functional Router object. But, as this object is not synchronized with any map, users are forced to provide their BlomURBEX usertoken to the object in order to make it work.

This object also accepts a 'service' parameter indicating which routing service to use. Possible values for this parameter are: 'ubiest' for Ubiest routing and 'oym' for On Your Map routing.

```
var router = new Urbex.Router({  
  usertoken: 'token',  
  service: 'tomtom'});
```

13.1.3 Using map's router

When a map is created, it automatically creates and stores internally a router object that can be reached as in the example:

```
var router = map.router;
```

This router will be synchronized with the map. This router object's service configuration can be changed as follows:

```
map.router.service = 'ubiest'; or map.router.service = 'oym';
```

13.1.4 Routing

Routing can either be performed between two different addresses or two different points (coordinates). Function `getRoute` is in charge

13.1.4.1 Function `getRoute`

Once an instance of Router has been retrieved, users can use its function **`getRoute`** to perform routing operations. This function accepts several parameters. Ones are common for all providers, and others are provider-specific.

13.1.4.1.1 Common Parameters

These parameters can be present

BlomURBEX™ Javascript API v5.7 r1.0

- **Options:** A Javascript object containing parameters to address with the request to BlomURBEX routing services. These options will depend on the provider, as stated on last sections. So there are parameters common for all providers, and parameters provider-specific
 - **origin (common):** Point of origin, if routing between points: String with x and y coordinates separated by comma (in Spherical Mercator projection).
 - **destination (common):** Point of destination, , if routing between points: String with x and y coordinates separated by comma (in Spherical Mercator projection).
 - **mode (common):** Routing mode.
 - "walking" for a walking routing.
 - "car" or omitted (default) for car routing.
 - **routeType (common):** Type of route:
 - "shortest" for shortest route
 - "fastest" or omitted (default) for fastest route
 - **originAddress (freeform address providers):** Freeform address of origin, if routing between addresses.
 - **destinationAddress (freeform address providers):** Freeform address of destination, if routing between addresses.
 - **Parameterized providers (ubiest):** Some more parameters are accepted for parameterized providers: originCountryCode, originCountry, originRegion, originProvince, originCity, originAddress, originStreetm destinationCountryCode, destinationCountry, destinationRegion, destinationProvince, destinationCity, destinationAddresssm destinationStreet. Only cities are mandatory, is routing between addresses.
- **Callback:** A function that will be called when the routing information is retriever. An object containing the response data will be addressed to the callback function. This object will be the result of evaluating a JSON like the following example:

```
{ "route": [ { "description": "", "startAddress": "", "endAddress": "", "duration": "16.72", "distance": "1.13KM", "step": [ { "description": "Albert Rd", "startPoint": "-0.144370,50.828614", "endPoint": "-
```


BlomURBEX™ Javascript API v5.7 r1.0

```
0.144370,50.828614","duration":"","distance":"","{"description":"Buckingham Rd","startPoint":"-0.143990,50.828588","endPoint":"-0.143990,50.828588","duration":"","distance":"","{"description":"Guildford Rd","startPoint":"-0.143920,50.828873","endPoint":"-0.143920,50.828873","duration":"","distance":"","{"description":"Trafalgar St","startPoint":"-0.141710,50.828721","endPoint":"-0.141710,50.828721","duration":"","distance":"","{"description":"Blackman St","startPoint":"-0.138750,50.828317","endPoint":"-0.138750,50.828317","duration":"","distance":"","{"description":"Cheapside","startPoint":"-0.138590,50.829580","endPoint":"-0.138590,50.829580","duration":"","distance":"","{"description":"London Rd","startPoint":"-0.135660,50.829460","endPoint":"-0.135660,50.829460","duration":"","distance":"","{"description":"St Peters Pl","startPoint":"-0.135640,50.829409","endPoint":"-0.135640,50.829409","duration":"","distance":"","{"description":"St Peters Pl","startPoint":"-0.134830,50.829302","endPoint":"-0.134830,50.829302","duration":"","distance":"","{"description":"Waterloo Pl","startPoint":"-0.133970,50.828980","endPoint":"-0.133970,50.828980","duration":"","distance":"","{"description":"Waterloo Pl","startPoint":"-0.134400,50.828487","endPoint":"-0.134400,50.828487","duration":"","distance":""}}]],"overviewPolyline":["-0.144370,50.828613","-0.144280,50.828601","-0.143990,50.828589","-0.143920,50.828874","-0.143089,50.828828","-0.142749,50.828809","-0.142500,50.828796","-0.142370,50.828792","-0.142150,50.828772","-0.141710,50.828722","-0.141370,50.828671","-0.141100,50.828633","-0.140719,50.828582","-0.140290,50.828531","-0.140049,50.828500","-0.139730,50.828450","-0.139420,50.828411","-0.138929,50.828341","-0.138750,50.828317","-0.138589,50.829580","-0.138410,50.829548","-0.138170,50.829529","-0.137640,50.829490","-0.137260,50.829459","-0.136750,50.829459","-0.136040,50.829459","-0.135659,50.829459","-0.135640,50.829408","-0.135460,50.829322","-0.135410,50.829307","-0.135350,50.829302","-0.135170,50.829302","-0.134830,50.829302","-0.134560,50.829252","-0.134370,50.829182","-0.133970,50.828980","-0.134079,50.828910","-0.134250,50.828758","-0.134330,50.828671","-0.134360,50.828633","-0.134370,50.828601","-0.134400,50.828519","-0.134400,50.828488"]];}
```

So, callback function can access all attributes of this object.

- The attribute 'route' contains all the necessary information required to route between an origin and a destination as it was requested.
- Every item in that array is a javascript object with following attributes:
 - Description: route description if any.

BlomURBEX™ Javascript API v5.7 r1.0

- startAddress: textual origin address
- endAddress: textual destination address
- duration: time duration of the route
- distance: distance of the route
- step: Array of steps comprising the route. Each step in turn has description, startAddress, endAddress, duration and distance attributes.
- overviewPolyline: Array of spherical Mercator points which helps to plot a continuous route on the map.

If an error occurred, the response will be a string starting with 'ERROR' following with the error explanation. To check if the response went well, users can do the following:

```
if (!response || (""+response).indexOf('ERROR') != -1)) {  
    alert("Route Not Found");  
} else {  
    //Process response object  
}
```

This example uses the ubiest routing service.

```
var callback = function(response) {  
  
    if (!response || (""+response).indexOf('ERROR') != -1)) {  
        alert("Route not found");  
    } else {  
        //Do something with the route  
    }  
  
    });  
var params = {  
    "origin": "",  
    "originCountry": italia,  
    "originCity": quinto di treviso,  
    "originAddress": "",  
    "originStreet": "",  
    "destination": "",  
    "destinationCountry": italia,  
    "destinationCity": treviso,  
    "destinationAddress": "",  
    "destinationStreet": "",
```

```
        "mode": car
    };
    map.router.getRoute(params, callback);
```

13.1.5 Displaying routes in the map

Routing object also provides functions to display the routes retrieved in the map. It will display the overview polyline as well as the key points with its indications.

This function, called **drawRoute**, will only work for router objects in association with a map, not for standalone router objects (since, in this case, there won't be an available map for the routes to be plotted into).

Function **drawRoute** accepts these parameters:

- **Response:** Response object returned by function **getRoute**.
- **LineStyle:** A vector style object (see vectors for more information) for overview polyline style. If not provided, default style for routing will be applied.
- **pointStyle:** A vector style object (see vectors for more information) for routing indication points style. If not provided, default style for routing will be applied.
- **dontDrawSummary:** Avoid drawing the summary of the route as a 'popup' window on top of the map

An example of usage of this function integrated with **getRoute** function can be found below

```
Var g_oRouter = map.router;

g_oRouter.getRoute(options, function(response) {

    if (!response || (""+response).indexOf('ERROR') != -1)) {
        alert("Route not found");
    } else {
        g_oRouter.drawRoute(response);
    }
});
```

13.1.6 Clearing routes from the map

The router object also offers the method **clearRoutes** for deleting both the route vector and points, and the route summary, if drawn.

14 WMS layers

Since version 4.5, BlomURBEX javascript API offers the possibility for adding WMS layers as baselayers overlays to the maps.

It is as simple as creating an `Urbex.Layer.WMS` and adding it to the map, as in the example following:

```
Var layer = new Urbex.Layer.WMS(layername, url,params, options);  
  
newMap.addLayer(layer);
```

where:

- Layername would be the name for the new layer
- url would be the URL of the WMS server where the images would be requested.
- Params would be some params for the request string to address to the GetMap WMS service denoted by 'url'
- Options would be some options that will be added as attributes to the layer object. Example of useful parameters would be:
 - singleTile: Boolean indicating that the layer will cover its whole extent with a single tile (thus, only one request to the WMS server would be sent). False is the default value.
 - Ratio: (only useful if singleTile is activated) a number indicating which portion of the whole viewport is covered by the single tile. Values above 1 would make the tile bigger than the viewport, values under 1 would make it smaller.
 - isBaselayer: Boolean indicating that this layer is a baselayer. False is the default value.

A complete example on how to add a WMS layer as an overlay (with fake url and params):

```
Var layer = new Urbex.Layer.WMS('wmslayer',  
'http://mywmsServer',{format:'image/png',transparent:'true'},  
{singleTile:true, ratio:1});  
  
map.addLayer(layer);
```

BlomURBEX™ Javascript API v5.7 r1.0

A complete example on how to add a WMS layer as a baselayer is given below. In this example, just by tagging the new layer as a baselayer, the `destroy()` method of the old baselayer automatically adds it as a new baselayer

```
Var layer = new Urbex.Layer.WMS('wmslayer',  
'http://mywmsServer',{format:'image/png',transparent:'true'},  
{singleTile:true, ratio:1, isBaselayer:true});  
  
newMap.addLayer(layer);  
  
map.baselayer.destroy();
```



15 WFS layers

Since version 5.5, BlomURBEX javascript API offers the possibility for adding WFS layers to the maps.

On creating a WFs layer, we need to specify following parameters:

- Name of the layer.
- URL of WFS server.
- Parameters object: Key/value pairs for additional parameters to add to the WFS requests to the server
- Options object: Additional options for the LayerWFS object. Special attention to:
 - user: username if HTTP Basic authentication is needed,.
 - password: password for HTTP Basic authentication if needed.
- Callback: A callback function that will be called once the GetCapabilities request is sent to the server. More on this later on.

Example of creation:

```
Var layer = new Urbex.Layer.WFS(layername, url,params, options,  
WFScallback);
```

15.1 Adding WFS Layers to the map

Adding WFS layers to the map is not as easy as adding WMS layers to the map. Once the WFS layer is added to the map, it first will request for the Capabilities document of the server (through a GetCapabilities request).

Once the request has been performed, the layer object will parse the response, and create a 'typenames' array as an attribute of the layer that will contain an object for every category the WFS server offers. These objects will have following parameter structure:

- Name: name of the typename
- Description: Description of the typename

Then, the callback set in the initialization will be executed, and this typenames array will be passed on to it as a parameter. This callback can be used to display the user of the map all the typenames in the server, for him to choose the active one.

Once all the typenames have been fetched, developers can use method **setTypeNames(typename)** to set the active typename. Once the active typename has been set, the layer will start to request for the WFS information in the viewport and rendering it as vectors.

15.1.1 WFS Layer useful methods and properties

- **setTypeNames(typename):** Sets the active typename for requesting features from. Will override last active typename.
- **WFSLayer.typenames:** Array containing all available typenames in the server.
- **setFeatureStyle(pointStyle, lineStyle, polygonStyle):** Overrides the default styling of WFS vector features. *pointStyle*, *lineStyle* and *polygonStyle* should be vector style objects (see vector section for more details).



16 POIs

Since version 5.0 of the API, integration with BlomURBEX POI platform is possible.

16.1 POI Concept

POI stands for 'Points Of Interest'. BlomURBEX POI platform can host and serve georeferenced data for points of interests, categorized, with custom icons and from different providers.

Every POI provider will have different POI categories, and each individual POI will belong at least to one category (it could belong to more than one).

16.2 Displaying POIs in the Map

Javascript API offers a POI Object (Urbex.POIs) that will handle all actions regarding POIs requests and displaying them in the map. Every map will provide with a POI object ready to use (initializing it first).

Map class also will offer some functions to interact with this POI object.

16.2.1 POI object initialization

In order to use POI platform, initialization of the POI object is required. In this initialization, BlomURBEX POI platform will be queried and all POI providers featuring in the platform and that the usertoken of the map is entitled to see/use will be fetched.

This initialization should be done using the function ***initPoisObject*** of the map.

This function accepts a callback function as a parameter, which will be called whenever the initialization is finished. It will be provided with two parameters:

- **Success:** a Boolean indicating whether the initialization was correct or not.
- **Map:** a reference to the map, to be used if needed.

```
var callback = function(success, map){  
  if (success) {  
    //Do something  
  } else {  
    //Notify error.  
  }  
}
```

```
}  
  
map.initPOIsObject(callback);
```

Once initialized, a reference to the POI object can be obtained as ***map.poisObj***, and a reference to a Javascript object with all the list of the providers can be obtained in the property ***providers*** of the POI object.

This object will be an array of objects with data related to each provider, indexed by the provider's name. Useful details for the about these objects will be further described.

```
var poisObject = map.poisObj;  
var providers = poisObject.providers;
```

16.2.1.1 POI providers information storage

As described, whenever initializing the POI object, all information regarding each provider is stored inside ***poiObject.providers*** Javascript Object. This object will have a member for every provider with the same name as the provider. That is equivalent as having an array indexed by a String instead of an index.

Let's see it with a little example: Say, for example, that there is a provider called 'AAA' and its information needs to be fetched. The way to access this provider's information is by accessing the object ***poiObject.providers['AAA']***.

Every provider will have an information object inside this array, with following properties:

- **id**: The external ID of the provider. User may not need to worry about this property. (String)
- **name**: Name of the provider. (String)
- **categories**: An array of the categories belonging to this provider. How to fetch these categories will be explained in the next section. (Array)

16.2.2 Getting all categories available for a certain provider

Once the POI object has been initialized and the full list of available providers has been retrieved, then the POI object enables the user to ask for the list of all categories available for every provider.

In order to do that, the method ***requestCategoriesForProvider*** in the POI object needs to be called. This method will accept two parameters:

- Provider object: Object obtained from the list ***poisObject.providers***.
- Callback: A callback function to be called whenever this request is finished. This function will be given two parameters:
 - Success: Boolean stating the result of the request
 - Map: A reference to the map to be used if needed.

Once the categories have been fetched, they are stored inside ***poisObj.providers[providerName].categories*** array. This array will contain an object with relevant information for every POI category of that particular provider.

Users can check whether or not the providers have already loaded categories prior to requesting for them in order not to request for them twice, like in the example:

```
var provider = 'providerName';
var poisObject = map.poisObj;
var providersObject = poisObject.providers;
var provider = providersObject[provider];

if (!provider.categories || provider.categories.length){

    var callback = function(success, map) {
        //Do something with categories
    }

    poisObject.requestCategoriesForProvider(provider, callback);
}
```

16.2.2.1 Categories information storage

As described, an object with relevant information about each category is stored inside an array of categories inside the provider information object (see 16.2.1.1).

For every category, this object will contain following properties

- name: Name of the category (String).
- id: External ID of the category: This id will be used for reference when asking for all the POIs inside a category.
- version: Version of the category (String).
- text: Text for the category (String).

- **iconUrl:** URL for requesting the icon image for this category (String).
- **parent:** ID of the parent category.

16.2.3 Displaying POIs of a certain category into the map

Once the providers and, at least, one list of provider categories have been loaded, then user can display POIs belonging to any of these categories on top of the map imagery. In order to do that, users should use ***addPOIsLayerToMap*** method of POI Object.

Users will have 3 different ways of representing POIs on top of the map:

- Using icons stored in the platform by the provider for the categories and the POIs.
- Using generic marker icons, with different colours for different categories.
- Using their own icons.

Function ***addPOIsLayerToMap*** will allow this three ways of working. This is an example on how to call it:

```
map.poisObj.addPOILayerToMap(category, name, usePOIIcon, customIcon);
```

The function accepts 4 parameters:

- **Category:** id of the category to be displayed in the map. (String)
- **Name:** Name of the layer. (String)
- **usePOIIcon:** Boolean stating if user wants to use the default POI icon (the one stored in BlomURBEX Platform. **Important notice:** In case that the POI do not have any icon defined in the platform, this will result in a default imagery delivering). If false, and no custom icon is provided, then the default icon for markers will be used for these POIs. POI Object will ensure to make a round robin between all available colours of default icons so that new categories added won't repeat icon until all colours available are being in use.
- **customIcon:** An Urbex.Icon object to be used as icon instead of the default icon. If a custom icon is provided, then Boolean usePOIIcon does not have any effect. Here is an example on how to create a custom icon, defining its URL, its size and a function to calculate the offset to the point that should signal the POI.


```
var url = http://url.png;  
var size = new Urbex.Size(25, 25);  
var calculateOffset = function(size){  
    return new Urbex.Pixel(-(size.w / 2), -(size.h / 2));  
};  
  
icon_ = new Urbex.Icon(url, size, null, calculateOffset);
```

16.2.4 Complete example of POI loading and displaying sequence

Here is a complete example on how to load and display three random POI categories for one sample provider, 'Geomobile', using the icon stored in the platform for them.

```
var initCallback = function(success, map){  
    if (success) {  
  
        var poisObject = map.poisObj;  
        var providersObject = poisObject.providers;  
        var provider = providersObject['Geomobile'];  
  
        var displayPOICategory = function(category){  
            poisObject.addPOIsLayerToMap(category.id, 'MyPOIs', true);  
        }  
  
        if (!provider.categories || provider.categories.length){  
  
            var categoriesCallback = function(success, map) {  
                if (success) {  
                    displayPOICategory(provider.categories[0]);  
                    displayPOICategory(provider.categories[1]);  
                    displayPOICategory(provider.categories[2]);  
                } else {  
                    //Notify error  
                }  
            }  
  
            poisObject.requestCategoriesForProvider(provider,  
categoriesCallback);  
        } else {  
            displayPOICategory(provider.categories[0]);  
        }  
    }  
}
```

BlomURBEX™ Javascript API v5.7 r1.0

```
    } else {  
        //Notify error.  
    }  
}  
  
newMap.initPOIsObject(initCallback);
```



17 Coordinates Conversion

The **Urbex.Util** Class offers several methods for coordinate conversion from Spherical Mercator to LatLong WGS84 and UTM projections and vice versa. These methods will help development and integration with other applications or data providers.

There is also a general method for coordinate conversion for conversions different from the aforementioned ones. This method makes server calls, and so the conversions using this method are slower than using the specific methods.

17.1 WGS84 to Spherical Mercator

- **WGS84ToSphericalMercator** (Longitude, Latitude). Will return an array containing the X and Y values of the reprojected coordinates from WGS84 LATLONG to Spherical Mercator

```
var result = Urbex.Util.WGS84ToSphericalMercator(4.3, 40.1);  
var X=result[0];  
var Y=result[1];
```

- **SphericalMercatorToWGS84**(X, Y). Will return an array containing the Longitude and Latitude corresponding to the conversion from Spherical Mercator coordinates to WGS84 LATLONG.

```
var result = Urbex.Util.SphericalMercatorToWGS84(-523800, 4960392);  
var lon=result[0];  
var lat=result[1];
```

17.2 UTM to Spherical Mercator

- **UtmToSphericalMercator** (X, Y, Zone). Will return an array containing the X and Y values of the reprojected coordinates from UTM to Spherical Mercator.

```
var result = Urbex.Util.UtmToSphericalMercator(-523121, 4960371, 30);  
var X=result[0];  
var Y=result[1];
```

- **SphericalMercatorToUtm** (X, Y, Zone). Will return an array containing the X and Y values of the reprojected coordinates from Spherical Mercator to UTM.

```
var result = Urbex.Util.SphericalMercatorToUtm(-523800, 4960392, 30);  
var lon=result[0];  
var lat=result[1];
```

17.3 WGS84 to UTM

- **WGS84ToUtm** (Longitude, Latitude, Zone). Will return an array containing the X and Y values of the reprojected coordinates from WGS84 LATLONG to UTM.

```
var result = Urbex.Util.WGS84ToUtm(4.3, 40.1, 30);  
var X=result[0];  
var Y=result[1];
```

- **UtmToWGS84** (X, Y, Zone). Will return an array containing the X and Y values of the reprojected coordinates from UTM to WGS84 LATLONG.

```
var result = Urbex.Util.UtmToWGS84(-523121, 4960371, 30);  
var lon=result[0];  
var lat=result[1];
```

17.4 General coordinates conversion

To convert coordinates between coordinate systems other than 'UTM', 'WGS84' and 'Spherical Mercator' use the following method:

- **coordinateConversion** (map, X, Y, FromProjection, ToProjection, listener), Will return an array containing the X and Y values in the specified 'ToProjection' system, **listener**, is a function that receives the result of transformation and another parameter with the map reference. The function needs a reference to the current Urbex.Map object. In the following example, the coordinates will be transformed from EPSG:2338 to EPSG: 32766 and the map will be centered on the new coordinates received.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523121,  
  y:4960371});  
  
function callback(map, result) {  
  var X=result[0];  
  var Y=result[1];  
  
  map.setView(new Urbex.XY(X, Y));  
}  
  
Urbex.Util.coordinateConversion(  
  newMap,  
  -523121,
```

```
4960371,  
'EPSG:2338',  
'EPSG:32766',  
callback);
```

17.5 Screen coordinates to the map's reference system.

To convert coordinates from screen to the map's coordinate system use the following function:

- **GetWorldCoordinates**(map, xyList, callback), Will return an array containing the values of each element in the **xyList** (that contains viewport coordinates) transformed to **map's** coordinate system. **Callback** is a function that receives the result of transformation and another parameter with the map reference.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  x: -523121,  
  y: 4960371});  
  
var callback = function(xyList) {  
  var points = [];  
  for (var i in xyList) {  
    // do your stuff;  
  }  
}  
  
var points = [  
  new Urbex.XY(100, 100),  
  new Urbex.XY(500, 500)  
];  
  
Urbex.Util.GetWorldCoordinates(newMap, points, callback);
```

17.6 World Coordinates to Screen coordinates

To convert coordinates from map's reference system into pixel coordinates in the viewport, use the following function:

- **GetScreenCoordinatesFromWorldCoordinates**(map, lonlatList, callback), Will return an array containing the values of each element in the **lonlatList** (that contains world coordinates on the map's reference system) transformed to **viewport** pixel coordinates.

Callback is a function that receives an array of Urbex.XY elements with the result of transformation.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var callback = function(xylist) {
  var points = [];
  for (var i in xyList) {
    // do your stuff;
  }
}

var points = [
  new Urbex.XY(100, 100),
  new Urbex.XY(500, 500)
];

Urbex.Util.GetScreenCoordinatesFromWorldCoordinates(newMap, points,
callback);
```

17.7 World coordinates to Oblique coordinates on map's oblique image

To convert coordinates from map's reference system to oblique coordinates in the current oblique image the map is displaying, use the following function:

- **GetObliqueCoordinatesFromWorldCoordinates**(map, lonlatList, callback), Will return an array containing the values of each element in the **lonlatList** (that contains world coordinates on the map's reference system) transformed to oblique coordinates on the oblique image the map passes as parameter is displaying. So it is mandatory that the map is in oblique mode **Callback** is a function that receives an array of Urbex.XY elements with the result of transformation.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  view: 'OBLIQUE',
  orientation: 'EAST',
  x: -523121,
  y: 4960371});

var callback = function(xylist) {
```

```
var points = [];  
for (var i in xyList) {  
    // do your stuff;  
}  
}  
  
var points = [  
    new Urbex.XY(100, 100),  
    new Urbex.XY(500, 500)  
];  
  
Urbex.Util.GetObliqueCoordinatesFromWorldCoordinates(newMap, points,  
callback);
```

17.8 Oblique coordinates to World coordinates on map's oblique image

To convert coordinates from oblique coordinates into map's reference system in the current oblique image the map is displaying, use the following function:

- **GetWorldCoordinatesFromObliqueCoordinates**(map, lonlatList, callback), Will return an array containing the values of each element in the **lonlatList** (that contains oblique pixel coordinates on the oblique image the map passed as parameter is displaying) transformed into coordinates on the map's reference system. So it is mandatory that the map is in oblique mode. **Callback** is a function that receives an array of Urbex.XY elements with the result of transformation.

```
var newMap = new Urbex.Map('divName', {  
    usertoken: 'stringToken',  
    view: 'OBLIQUE',  
    orientation: 'EAST',  
    x: -523121,  
    y: 4960371});  
  
var callback = function(xyList) {  
    var points = [];  
    for (var i in xyList) {  
        // do your stuff;  
    }  
}  
  
var points = [  
    new Urbex.XY(100, 100),  
    new Urbex.XY(500, 500)
```

```
];  
  
Urbex.Util.GetObliqueCoordinatesFromWorldCoordinates(newMap, points,  
callback);
```



18 Event handling

BlomURBEX is able to capture and manage events in order to create complex applications on top of its core functionality. This chapter covers event creation and available methods and types. For a discussion on handlers, please refer to chapter 21.

18.1 Event construction

To manage events it is necessary to construct an `Urbex.Events` object.

```
var events = new Urbex.Events(  
    object,  
    addedElement,  
    eventTypes,  
    fallThrough);
```

Accepted parameters are:

- **object:** {Object} (mandatory) The js object to which this Events object is being associated.
- **addedElement:** {DOMElement} (mandatory). A dom element to respond to browser events.
- **eventTypes:** {Array(String)} (mandatory). Array of custom application events.
- **fallThrough:** {Boolean} (mandatory). Allows events to fall through after these have been handled.

The events supported (described by type on section 18.4) are automatically built into the API, only in the case of wanting to build a new event would use **Urbex.Events** constructor.

Each object map has its own **Urbex.events** object that will manage its events.

18.2 Methods

This section covers the different methods available in the `Urbex.Events` object.

18.2.1 register

- **register(type, obj, function):** Registers a listener to an event associated to an Urbex object.

```
UrbexObject.events.register(type, obj, callback);
```

Where:

- **UrbexObject:** {Object} Urbex object which triggers events (the objects and their associated events are described later)
- **type:** {String} Name of the event to register
- **obj:** {Object} Object passed as argument to the callback function. It is possible to reference this object in the callback function with the keyword **'this'**. If no object is specified, default is the **UrbexObject** 'object' property
- **callback:** {Function} function called when the event is triggered. If no function is specified, this method does nothing.

18.2.2 unregister

- **unregister(type, obj, function):** Unregisters a listener to an event on the events object.

```
UrbexObject.events.unregister(type, obj, callback);
```

Where:

- **UrbexObject:** {Object} Urbex object which triggers events (the objects and their associated events are describe later)
- **type:** {String} Name of the event to unregister
- **obj:** {Object} Object passed as argument to the callback function. It is possible to reference this object in the callback function with the keyword **'this'**. It must be the same object that was used in the register method. If no object is specified, default is the UrbexObject 'object' property.
- **callback:** {Function} The callback function. If no callback is specified, this function does nothing.

18.2.3 triggerEvent

- **triggerEvent():** Triggers a specified registered event.

```
Boolean result = UrbexObject.events.triggerEvent(type, evt);
```

Where:

- **UrbexObject:** {Object} Urbex object which triggers events (the objects and their associated events are described later)
- **type:** {String} Name of the event to trigger
- **evt:** {Event}
- **result:** {Boolean}, Value returned by the last listener of the event. If a listener returns false, the chain of listeners will stop being called.

Callbacks will be called with a reference to an object. Properties of this object depend on the exact event that has been triggered.

All objects passed to callbacks have at least the following arguments, and it is possible to accede to them from the function callback with the keyword **'this'**.

- **object** {Object} A reference to the UrbexObject object that triggered the event.
- **element** {DOMElement} A reference to the DOM element that matches the UrbexObject that triggered the event.

18.3 Not managed events

Urbex.Event object offers static methods that can be used to deal with browser events. These methods help the developer abstracting its code from the different browsers implementation of events. These methods are detailed below:

18.3.1 observe

This method registers a callback for the events identified by a name thrown by an element.

```
Urbex.Event.observe(element, name, callback);
```

- **element:** {DOMElement | String} Element or element's identifier. The source of the events we are registering for.
- **name:** {String} Name of the type of events we are registering for. These are standard DOM level 3 names.
- **callback:** {Function} The function that will be called when the event is raised. The context of this function will be undetermined. The callback will always receive an object with the event info which implementation is browser dependent.

18.3.2 isLeftClick

This method helps to determine if a mouseclick event has been raised because of a left click or any other mouse button click.

```
Urbex.Event.isLeftClick(eventInfo)
```

- **eventInfo:** {Event} Event's info object.

18.3.3 stop

This method helps canceling the bubbling phase of an event defined in DOM level 3.

```
Urbex.Event.stop(eventInfo)
```

- **eventInfo:** {Event} Event's info object.

18.4 Event types

This section covers the different event types available in BlomURBEX.

Listeners will be called with a reference to an event object.

18.4.1 Map Events

Following is a list of supported **urbex.map** event types:

- **preaddlayer.** triggered before a layer has been added. The event object will include a **layer** property that references the layer to be added.
- **addlayer.** Triggered after a layer has been added. The event object will include a **layer** property that references the added layer.
- **removelayer.** triggered after a layer has been removed. The event object will include a **layer** property that references the removed layer.
- **changelayer.** Triggered after a layer name change, order change, or visibility change (due to resolution thresholds). Listeners will receive an event object with **layer** and **property** properties. The **layer** property will be a reference to the changed layer. The **property** property will be a key to the changed property (name, visibility, or order).
- **movestart.** Triggered after starting a drag, pan, or zoom action.
- **move.** Triggered after each drag, pan, or zoom action.

BlomURBEX™ Javascript API v5.7 r1.0

- **moveend.** triggered after a drag, pan, or zoom actions completes. Listeners will receive an event object with the extent of the view in the same projection of the map. This event will have as properties the coordinates of the four points defining the extent of the view **ulx, uly, urx, ury, llx, lly, lrx, lry** and center image parameters **x** and **y**.
- **mouseover.** triggered when the mouse hovers the map
- **mouseout.** triggered after mouse pointer gets out of the map
- **mousemove.** Triggered after mouse hovers the map.
- **changebaselayer** triggered after the base layer changes
- **areameasured.** Triggered after the area control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **bearingmeasured.** Triggered after the bearing control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **lengthmeasured.** Triggered after the length control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **elevationmeasured.** Triggered after the elevation control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **heightmeasured.** Triggered after the height control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **locationmeasured.** Triggered after the 'returnpoint' control ends measuring. Listeners will receive an event object with **measure** property expressed in Urbex.XY type.
- **obliqueloaded.** Triggered after the oblique changes. Listeners will receive an event object with **id, width, height, center** (the center of the map in oblique coordinates) and extent oblique properties **ulx, uly, urx, ury, llx, lly, lrx, lry** of the new oblique image.
- **viewchanged.** Triggered after the view changes. Listeners will receive an event object with **view** property.
- **orientationchanged.** triggered after the orientation changed. Listeners will receive an event object with **orientation** property.

- **extentchanged.** Triggered after the map extent changes. Listeners will receive an event object with the extent of the view in the same projection of the map. This event will have as properties the coordinates of the four points defining the extent of the view **ulx, uly, urx, ury, llx, lly, lrx, lry** and center image parameters **x** and **y**.
- **zoomlimitschanged.** Triggered after the map zoom limits changes. Listeners will receive an event object with the new limits. This event will have the next properties: **min** (the new minimum zoom in ortho view), **max** (the new maximum zoom in ortho view), **minOblique** (the new minimum zoom in oblique view), **maxOblique** (the new maximum zoom in oblique view).
- **diagonalmeasured.** Triggered after the diagonal control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.
- **verticalareameasured.** Triggered after the vertical area control ends measuring. Listeners will receive an event object with **measure**, and **geometryid** properties.

18.4.1.1 Examples

Extent changed:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  zoom: 15,
  x: -523121,
  y: 4960371,
  view: 'ORTHO',
  orientation: 'NORTH'
});

function listener(evt){
  alert('event extentchanged triggered');
}

var forceEvent = function() {
  newMap.events.register('extentchanged', newMap, listener);
  newMap.pan(1, 0); // trigger event extentchanged
}
setTimeout(forceEvent, 5000);
```

Oblique loaded:

```
var newMap = new Urbex.Map('divName', {
```

```
usertoken: 'stringToken',
zoom: 3,
x: -523121,
y: 4960371,
view: 'OBLIQUE',
orientation: 'NORTH'
});

function listener(evt){
    alert('event obliqueloaded triggered');
    //id of the new oblique image
    var id = evt.id;
}

var forceEvent = function() {
    newMap.events.register('obliqueloaded', newMap, listener);
    newMap.setView(
        new Urbex.XY(-523121, 4960371),
        'OBLIQUE',
        'SOUTH'); // trigger event obliqueloaded
}
setTimeout(forceEvent, 5000);
```

18.4.2 TextMarkers Events

Following is a list of supported **Urbex.Layer.TextMarker** event types:

- **loadstart.** Triggered when layer loading starts.
- **loadend.** Triggered when layer loading ends.

Listeners will receive as parameter the layer that has triggered the event.

18.4.2.1 *Example*

```
var newMap = new Urbex.Map('divName', {
    usertoken: 'stringToken',
    zoom: 15,
    x: -523121,
    y: 4960371
});

function listener(evt) {
    alert('event loadend triggered');
}

var forceEvent = function() {
```



```
var poi = new Urbex.Layer.TextMarkers(  
    'text',  
    {location: '../test/mymarkers.markers'});  
poi.events.register('loadend', poi, listener);  
newMap.addLayer(poi); // trigger event loadend  
}  
setTimeout(forceEvent, 5000);
```

18.4.3 Control Events

Following is a list of supported **Urbex.Control** event types:

- **activate.** Triggered when the control is activated.
- **deactivate.** Triggered when the control is deactivated.

18.4.3.1 Example

```
var newMap = new Urbex.Map( 'divName', {  
    usertoken: 'stringToken',  
    zoom: 15,  
    x: -523121,  
    y:4960371  
});  
  
function listener(evt) {  
    alert('event activate triggered');  
}  
  
var forceEvent = function() {  
    var click = new Urbex.Control.Click();  
    newMap.addControl(click);  
    click.events.register('activate', click, listener);  
    click.activate(); // trigger event activate  
}  
setTimeout(forceEvent, 5000);
```

18.4.4 Layer.Vector Events

Following is a list of supported **Feature** event types:

- **beforefeatureadded.** Triggered before a feature is added. Listeners will receive an object with a **feature** property referencing the feature to be added.

- **featureadded.** Triggered after a feature is added. The event object passed to listeners will have a **feature** property with a reference to the added feature.

18.4.4.1 *Example*

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

function listener() {
  alert('event featureadded triggered');
}

var forceEvent = function() {
  var vectorlayer = new Urbex.Layer.Vector('vectors');
  newMap.addLayer(vectorlayer);

  vectorlayer.events.register(
    'featureadded',
    vectorlayer,
    listener);

  var x = -523121;
  var y = 4960371;
  var geometry = new Urbex.Geometry.Point(x, y);
  var feature = new Urbex.Feature.Vector(geometry);
  vectorlayer.addFeatures([feature]); // trigger event
}

setTimeout(forceEvent, 5000);
```

19 Retrieving user available information

There are some function useful for the developer that will retrieve which information is available for displaying on a certain point on the map.

19.1 Urbex.Util.GetUserAvailableExtents

This method retrieves all the extents available for the user (represented by the usertoken property of the map). Then, it calls a callback function passed as a parameter and passes an array with all the available extent as a parameter to it.

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **type:** This parameter can either be 'ortho' or 'oblique', indicating whether the ortho available extents or the oblique available ones will be retrieved.
- **Callback:** A function to be called when the request for the information is completed. This function will receive an array as a parameter, containing a text representation for every extent available for the user.

The string representation for an extent is simply a concatenation of some of the extent parameters separated by a colon, as follows:

Name : maximum zoom : minimum zoom : X and Y cords for the edge points of the geometry : WKT representation on geographic coordinates

Example:

Getting all user available ortho extents

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseArray) {
  ... do some stuff with the response array ...
}

Urbex.Util.GetUserAvailableExtents(newMap, 'ortho', callback);
```

19.2 Urbex.Util.GetUserAvailableData

This method retrieves all available data for a user on a certain point at a certain zoom level. It requests for all layers the user is entitled to see or display in that point. When the information is retrieved, it calls a callback function passed as a parameter and passes an array with all the available layers as a parameter to it.

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **x:** x coordinate (in map's projection) of the point requested.
- **y:** y coordinate (in map's projection) of the point requested.
- **zoom:** Zoom level requested. It should be ortho zoom level if 'ortho' or '' type are requested (see below) or oblique zoom level if 'oblique' type is requested.
- **type:** This parameter can either be an empty string '', 'ortho' or 'oblique', indicating whether all the available data, only the ortho available data or only the oblique available data will be retrieved.
- **Callback:** A function to be called when the request for the information is completed. This function will receive an array as a parameter, containing a text representation for every data layer available for the user in the point and zoom level given.

The string representation for a data layer is simply a concatenation of some of the datalayer parameters separated by a colon, as follows:

Name : orientation (ortho, rectified orientation or oblique orientation) : number of dataset the layer belongs to : WKT representation of the layer available extent on geographic coordinates (if present, see below) : year (if present, see below)

- **yearMode:** If set to 'true', the year of the data delivered will be also present in the response. If set to false or not present, the year won't appear (for compliance reasons with older versions of the API)
- **deliverExtents:** Boolean stating if extent information about the layers will be delivered or not. The delivery of extents may make the process slower, so if speed is needed in this response and no extents

are really needed, it is recommended to set this to false. If not present, then by defaults extents will be delivered.

Example:

Getting all user available ortho layers on the map center.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseArray) {
  ... do some stuff with the response array ...
}

Urbex.Util.GetUserAvailableData(newMap, newMap.center.x,
newMap.center.y, newMap.zoom, 'ortho', callback, true);
```

19.3 Urbex.Util.GetUserAvailableBaselayers

This function will return a list with all available baselayer names for the current user. All available baselayers can be displayed inside a map. If a user tries to display a non-permitted baselayer, a blank screen will appear.

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **baselayers:** A list of baselayer names separated by commas that you want to check. If omitted, all available baselayers will be delivered. If present, only the allowed baselayers among the ones on the list will be returned.
- **Callback:** A function to be called when the request for the information is completed. This function will receive an array as a parameter, the names of every baselayer allowed.

Example:

Getting user available baselayers among 'countryortho' and 'satellite'


```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseArray) {
  ... do some stuff with the response array ...
}

Urbex.Util.GetUserAvailableOverlays(newMap, "countryortho,satellite",
callback);
```

19.4 Urbex.Util.GetUserAvailableOverlays

This function will return a list with all available overlay names for the current user. All overlays allowed can be shown on top of the baselayer inside the map.

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **Callback:** A function to be called when the request for the information is completed. This function will receive an array as a parameter, the names of every overlay allowed.

Example:

Getting user available overlays:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseArray) {
  ... do some stuff with the response array ...
}

Urbex.Util.GetUserAvailableOverlays(newMap, callback);
```

19.5 Urbex.Util.GetUserBlomSTREETCredentials

This function will ask the server for the credentials of this user to use BlomSTREET service (since this server, for the moment, is independent from BlomURBEX so it need a different credentials). If the user has no credentials, then he won't be able to use BlomSTREET service.

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **Callback:** A function to be called when the request for the information is completed. This function will receive a javascript object as parameter, which will have 3 properties: *user*, *password* and *apikey*

Example:

Getting user available overlays:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseObject){
  var user = responseObject.user;
  var password = responseObject.password;
  var apikey = responseObject.apikey;

  ... do some stuff with the response ...
}

Urbex.Util.GetUserBlomSTREETCredentials(newMap, callback);
```

19.6 Urbex.Util.GetUserAvailableProviders

This function will ask the server for the user available external providers to use with BlomURBEX services provided by external providers: Geocoding, routing, map images or POIs

The parameters that should be passed to this method are:

- **map:** A reference to the map object.
- **Callback:** A function to be called when the request for the information is completed. This function will receive a javascript object as parameter, with following properties:
 - **geocoding:** An array with the names of all allowed providers for geocoding.
 - **routing:** An array with the names of all allowed providers for routing.
 - **map:** An array with the names of all allowed providers for requesting map images.
 - **Hosted POIs:** An array with the names of all allowed providers for POIs hosted in BlomURBEX platform.
 - **External POIs:** An array with the names of all allowed providers for POIs not hosted on BlomURBEX platform.

An example of response will be the following object:

```
{ "geocoding": [ 'ubiest', 'osm', 'oym' ], "routing": [ 'oym', 'ubiest' ], "map": [ 'oym', 'osm', 'ubiest' ], "Hosted POIs": [ 'geomobile' ] }
```

Example:

Getting user available overlays:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371
});

var callback = function(responseObject){

  ... do some stuff with the response ...
}

Urbex.Util.GetUserAllowedProviders(newMap, callback);
```

20 Measurements Functions

There are five types of measurement functions in Urbex: Length, Area, Height, Bearing and Elevation.

To make measurements it is necessary to get the reference to the **Urbex.Measures** object, which can be accessed through a map getter: **getMeasurementObj()**.

For example:

```
var measureObj = new newMap.getMeasurementObj();
```

20.1 Length measurements

To measure lengths use the **Urbex.Measures.getLength()** method. Lengths can be measured in ortho and oblique views.

- **Urbex.Measures.getLength**(point1, point2, listener)
 - **point1**: {Urbex.XY} Initial measure point, in map coordinates.
 - **point2**: {Urbex.XY} Final measure point, in map coordinates.
 - **listener**: {Function} Function which will receive the map reference and the result of the length measurement.

Example, in ortho view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});
var x1 = -523121;
var y1 = 4960371;
var x2 = -523180;
var y2 = 4960320;

function callback(map, length) {
  alert(length);
}

var measureObj = newMap.getMeasurementObj();
measureObj.getLength(
  new Urbex.XY(x1, y1),
  new Urbex.XY(x2, y2),
```

```
callback);  
  
// length = 59.172;
```

Example, in oblique view:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  orientation: 'NORTH',  
  view: 'OBLIQUE',  
  x: -523121,  
  y: 4960371});  
var x1 = 1880;  
var y1 = 234;  
var x2 = 1680;  
var y2 = 456;  
  
function measureLength() {  
  
  function callback(map, length) {  
    alert(length);  
  }  
  
  var measureObj = newMap.getMeasurementObj();  
  measureObj.getLength(new Urbex.XY(x1, y1), new Urbex.XY(x2, y2),  
callback);  
  
  // length = 93.788;  
}  
  
// the oblique view loading is an asynchronous operation  
// it is necessary to use events to ensure the usage of the map in a  
// correct state  
newMap.events.register("obliqueloaded", null, measureLength);
```


20.2 Area measurements

To measure areas use the **Urbex.Measures.getArea()** method. Areas can be measured in ortho and oblique views.

- **Urbex.Measures.getArea**(points, listener)
 - **points:** {Array (Urbex.XY)} vertex polygon, in map coordinates.
 - **listener:** {Function} Function which will retrieve the map reference and the result of the area measurement.

Example, in ortho view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});
var x1 = -523171.16180;
var y1 = 4960396.08090;
var x2 = -523172.35613;
var y2 = 4960356.66806;
var x3 = -523131.74896;
var y3 = 4960353.08507;
var x4 = -523116.22269;
var y4 = 4960410.41284;

var points = [
  new Urbex.XY(x1, y1),
  new Urbex.XY(x2, y2),
  new Urbex.XY(x3, y3),
  new Urbex.XY(x4, y4),
  new Urbex.XY(x1, y1)];

function callback(map, area) {
  alert(area);
}

var measureObj = newMap.getMeasurementObj();
measureObj.getArea(points, callback);

// area = 1302.709;
```

Example, in oblique view:

```
var newMap = new Urbex.Map('divName', {
```

BlomURBEX™ Javascript API v5.7 r1.0

```
usertoken: 'stringToken',
orientation: 'NORTH',
view: 'OBLIQUE',
x: -523121,
y: 4960371,
});
var x1 = 1880;
var y1 = 367;
var x2 = 1745;
var y2 = 100;
var x3 = 1800;
var y3 = 324;
var x4 = 1234;
var y4 = 123;

var points = [
  new Urbex.XY(x1, y1),
  new Urbex.XY(x2, y2),
  new Urbex.XY(x3, y3),
  new Urbex.XY(x4, y4),
  new Urbex.XY(x1, y1)];

function measureArea() {

  function callback(map, area) {
    alert(area);
  }

  var measureObj = newMap.getMeasurementObj();
  measureObj.getArea(points, callback);

  // area = 43578.893;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloaded", null, measureArea);
```

20.3 Height measurements

To measure Height use the **Urbex.Measures.getHeight()** method. Heights can only be measured in oblique views.

- **Urbex.Measures.getHeight(point1, point2, listener)**
 - **point1:** {Urbex.XY} Initial measure point, in map coordinates.
 - **point2:** {Urbex.XY} Final measure point, in map coordinates.
 - **listener:** {Function} Function which will retrieve the map reference and the result of the height measure.

Example:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121, y:4960371,
  view: 'OBLIQUE',
  orientation: 'NORTH'});

var x1 = 1880;
var y1 = 969;
var x2 = 1800;
var y2 = 900;

function measureHeight() {

  function callback(map, height) {
    alert(height);
  }

  var measureObj = newMap.getMeasurementObj();
  measureObj.getHeight(
    new Urbex.XY(x1,y1),
    new Urbex.XY(x2,y2),
    callback);

  // height = 33.250;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloaded", null, measureHeight);
```

20.4 Bearing measurements

To measure Bearing use the **Urbex.Measures.getBearing()** method. Bearings can only be measured in oblique views.

- **Urbex.Measures.getBearing**(point1, point2)
 - **point1**: {Urbex.XY} Initial measure point, in map coordinates.
 - **point2**: {Urbex.XY} Final measure point, in map coordinates.
 - **listener**: {Function} Function which will retrieve the map reference and the result of the height measure.

Example:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  orientation: 'NORTH',
  view: 'OBLIQUE',
  x: -523121,
  y: 4960371});

var x1 = 1880;
var y1 = 969;
var x2 = 1800;
var y2 = 900;

function measureBearing() {

  function callback(map, bearing) {
    alert(bearing);
  }

  var measureObj = newMap.getMeasurementObj();
  measureObj.getBearing(new Urbex.XY(x1,y1), new Urbex.XY(x2,y2),
  callback);

  // bearing = 218.702;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloaded", null, measureBearing);
```

20.5 Elevation measurements

To measure Elevation use the **Urbex.Measures.getElevation()** method. Elevations can be measured in ortho and oblique views.

- **Urbex.Measures.getElevation(point)**
 - **point:** {Urbex.XY} measure point, in map coordinates.
 - **listener:** {Function} Function which will retrieve the map reference and the result of the height measure.

Example, in ortho view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});
var x = -523100;
var y = 4960300;

function callback(map, elevation) {
  alert(elevation);
}

var measureObj = newMap.getMeasurementObj();
measureObj.getElevation(new Urbex.XY(x,y), callback);

// elevation = 1066.831;
```

For example, in oblique view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  orientation: 'NORTH',
  view: 'OBLIQUE',
  x: -523121,
  y: 4960371
});

var x = 1890;
var y = 456;

function measureElevation() {

  function callback(map, elevation) {
    alert(elevation);
  }
}
```



```
}

var measureObj = newMap.getMeasurementObj();
measureObj.getElevation(new Urbex.XY(x, y), callback);

// elevation = 1076.356;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloading", null, measureElevation);
```

20.6 Ground Length measurements

To measure ground lengths use the **Urbex.Measures.getGroundLength()** method. Ground lengths can be measured in ortho and oblique views.

- **Urbex.Measures.getLength**(point1, point2, listener)
 - **point1**: {Urbex.XY} Initial measure point, in map coordinates.
 - **point2**: {Urbex.XY} Final measure point, in map coordinates.
 - **listener**: {Function} Function which will receive the map reference and the result of the length measurement.

Example, in ortho view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -523121,
  y: 4960371});

var x1 = -523121;
var y1 = 4960371;
var x2 = -523180;
var y2 = 4960320;

function callback(map, length) {
  alert(length);
}

var measureObj = newMap.getMeasurementObj();
measureObj.getGroundLength(
  new Urbex.XY(x1, y1),
  new Urbex.XY(x2, y2),
```

```
callback);  
  
// length = 59.172;
```

Example, in oblique view:

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',  
  orientation: 'NORTH',  
  view: 'OBLIQUE',  
  x: -523121,  
  y: 4960371});  
var x1 = 1880;  
var y1 = 234;  
var x2 = 1680;  
var y2 = 456;  
  
function measureLength() {  
  
  function callback(map, length) {  
    alert(length);  
  }  
  
  var measureObj = newMap.getMeasurementObj();  
  measureObj.getLength(new Urbex.XY(x1, y1), new Urbex.XY(x2, y2),  
callback);  
  
  // length = 93.788;  
}  
  
// the oblique view loading is an asynchronous operation  
// it is necessary to use events to ensure the usage of the map in a  
// correct state  
newMap.events.register("obliqueloading", null, measureLength);
```

20.7 Diagonal measurements

To measure Height use the **Urbex.Measures.getHeight()** method. Diagonals can only be measured in oblique views.

- **Urbex.Measures.getHeight(point1, point2, point3, listener)**
 - **point1:** {Urbex.XY} First point of the triangle, in map coordinates.
 - **point2:** {Urbex.XY} Second point of the triangle, in map coordinates.

- **Point3:** {Urbex.XY} Third point of the triangle, in map coordinates.
- **listener:** {Function} Function which will retrieve the map reference and the result of the diagonal measure.

Example:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  x: -411920,
  y: 4926681,
  view: 'OBLIQUE',
  orientation: 'NORTH'});

function measureDiagonal() {

  function callback(map, diagonal) {
    alert(diagonal);
  }

  var measureObj = newMap.getMeasurementObj();
  newMap.getMeasurementObj().getDiagonal(
    new Urbex.XY(2903,-2566),
    new Urbex.XY(2858,-2489),
    new Urbex.XY(2862,-2379),
    callback
  );

  // diagonal = 29.8244;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloaded", null, measureDiagonal);
```

20.8 Vertical Area measurements

To measure vertical areas use the **Urbex.Measures.getVerticalArea()** method. Vertical Areas can only be measured in oblique views.

- **Urbex.Measures.getVerticalArea(points, listener)**
 - **points:** {Array (Urbex.XY)} vertex polygon, in map coordinates.

listener: {Function} Function which will retrieve the map reference and the result of the vertical area measurement.

Example, in oblique view:

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken'
  orientation: 'NORTH',
    view: 'OBLIQUE',
    x: -522408.869815037,
    y: 4962063,
});

var x1 = 1847.5;
var y1 = -2088.5;
var x2 = 1969;
var y2 = -2089.5;
var x3 = 1959.5;
var y3 = -2014.5;
var x4 = 1835;
var y4 = -2013.5;

var points = [
  new Urbex.XY(x1, y1),
  new Urbex.XY(x2, y2),
  new Urbex.XY(x3, y3),
  new Urbex.XY(x4, y4),
  new Urbex.XY(x1, y1)];

function measureVerticalArea() {

  function callback(map, verticalarea) {
    alert(verticalarea);
  }

  var measureObj = newMap.getMeasurementObj();
  measureObj.getVerticalArea(points, callback);

  // vericalarea = 283.9839328463204;
}

// the oblique view loading is an asynchronous operation
// it is necessary to use events to ensure the usage of the map in a
// correct state
newMap.events.register("obliqueloaded", null, measureVerticalArea);
```

20.9 Retrieving the graphical elements associated to a measurement.

When the user draws a graphical element using the measurement controls the application automatically creates a feature and a popup associated to the measurement. The identifier for retrieving these elements can be obtained from the measurement event.

Once the identifier has been obtained, the **getFeatureById** method from the vector layer (the vector layer can be retrieved with the **getVectorLayer** method from the **MeasureObj**), and the **getMeasurePopup** from the measures object can be used.

20.9.1 Popups

The popup is a little label of information. With its reference the developer can use the **getDOMEElement** function to get the **DOMEElement** and alter its properties.

- **getDOMEElement()**: Get the **DOMEElement** to alter the content and/or information on the popup. The developer must not modify the position through the **DOMEElement**.

Other actions on popups (move or delete) must be done via the utility methods in **Urbex.Measure** object:

- **modifyMeasurePopup(id, position)**: Set the new position (in world coordinates) of the popup.
 - **id**: {String} Popup's identifier.
 - **position**: {Urbex.XY} New location of the popup.
- **removeMeasurePopups(id)**: Remove the popup.
 - **id**: {String} Popup's identifier.
- **setMeasurePopupAlign(id, alignment)**: Set the popup alignment.
 - **id**: {String} Popup's identifier.
 - **alignment**: {String} Popup's alignment: This is a string of two characters, one for vertical alignment: 't' (top), 'l' (lower), 'm' (middle); and one for horizontal alignment: 'l' (left), 'r' (right), 'm' (middle). The default alignment is 'tl' (top-left)
- **setMeasurePopupVisible(id, visible)**: Sets the popup visibility.

- **id:** {String} Popup's identifier.
- **visibility:** {Boolean} **true** shows the popup, and **false** hides it.

20.9.2 Example

In the following example the map will draw the vector and popups associated to area measurements in green.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  orientation: 'NORTH',
  view: 'OBLIQUE',
  x: -523121,
  y: 4960371,
  loadcontrols: 'full'
});

function changeColor(eventInfo) {
  if (eventInfo.geometryid) {
    var measurementObj = newMap.getMeasurementObj();
    var vectorLayerObj = measurementObj.getVectorLayer();

    var popup = measurementObj.getMeasurePopup(eventInfo.geometryid);
    var feature = vectorLayerObj.getFeatureById(eventInfo.geometryid);

    var style = {
      fillColor: 'green',
      fillOpacity: 0.3,

      strokeColor: 'green',
      strokeOpacity: 0.5,
      strokeLinecap: 'round',
      strokeWidth: 2,
      pointRadius: 6,
      cursor: ''
    };

    popup.getDOMElement().style.background = 'green';
    popup.getDOMElement().style.opacity = 0.5;
    feature.setStyle(style);

    vectorLayerObj.redraw();
  }
}
```

```
newMap.events.register("areameasured", null, changeColor);
```

20.10 Customizing the graphical elements associated with a measurement:

Graphical elements which will be drawn whenever the user makes a measurement can be previously customized using some methods from the **MeasureObj** of the map.

There are two methods associated with each magnitude measurement: **setMagnitudeStyle**, which will customize all the Vector features drawn when measuring a certain magnitude, and **setMagnitudeStyleText**, which will customize all the popups displaying the measurement value for that magnitude

The complete list of methods for customizing measurements' graphical elements is displayed below:

- **setHeightStyle(style):** Sets the style for every Vector Feature associated to a height measurement.
 - **style:** {Object} Set of properties to apply to the features' style, according to what explained on section 9.6.
- **setHeightTextStyle(style):** Sets the style for every popup displaying height measurement values.
 - **Style:** {Object} Set of properties to apply to the popup's CSS style, in keeping with Javascript-CSS syntax. Only the following CSS properties are supported:
 - **border.**
 - **backgroundColor.**
 - **opacity.**
 - **fontStyle.**
 - **fontWeight.**
 - **fontFamily.**
 - **fontSize.**
 - **color.**
 - **padding.**

- **offset.** This is the only non-css property you can set to the popups' style. It indicates the separation between the popup and the point where it should be located when no offset is specified. Its String value must represent a pair of numbers separated by a comma, e.g. "5,5", being the first number the offset in the x axis and the second number the offset in the y axis.
- **setLengthStyle(style):** Same as **setHeightStyle**, associated to length measurements.
- **setLengthTextStyle(style):** Same as **setHeightTextStyle** associated to length measurements.
- **setAreaStyle(style):** Same as **setHeightStyle**, associated to area measurements.
- **setAreaTextStyle(style):** Same as **setHeightTextStyle** associated to area measurements
- **setBearingStyle(style):** Same as **setHeightStyle**, associated to bearing measurements.
- **setBearingTextStyle(style):** Same as **setHeightTextStyle**, associated to bearing measurements
- **setElevationStyle(style):** Same as **setHeightStyle**, associated to elevation measurements.
- **setElevationTextStyle(style):** Same as **setHeightTextStyle**, associated to elevation measurements.
- **setGroundLengthStyle(style):** Same as **setHeightStyle**, associated to ground length measurements.
- **setGroundLengthTextStyle(style):** Same as **setHeightTextStyle** associated to ground length measurements.

20.10.1 Example

Example A

This is the same example as the one in section 13.6.2. Map will draw all Vector layers and popups associated to area measurements in green, but now there is no need to use events registry.

```
var newMap = new Urbex.Map('divName', {  
  usertoken: 'stringToken',
```

```
orientation: 'NORTH',
view: 'OBLIQUE',
x: -523121,
y: 4960371,
loadcontrols: 'full'
});

var style = {
  fillColor: 'green',
  fillOpacity: 0.3,

  strokeColor: 'green',
  strokeOpacity: 0.5,
  strokeLinecap: 'round',
  strokeWidth: 2,
  pointRadius: 6,
  cursor: ''
};

var textStyle = {
  backgroundColor: 'green',
  opacity: '0.5'
}

var mesObj = newMap.getMeasurementObj();

mesObj.setAreaStyle(style);
mesObj.setAreaTextStyle(textStyle);
```

Example B

This example will force the map to draw all vector features associated to length measurements in yellow, and will set the length measurement popups style for them to be displayed in blue, covered by a green border, with a red, 14px Arial font and a 5px padding (separation between the text and the borders of the popup). The popup will have an offset of 5,5, so it will be separated 5 px down and 5px right from its original position.

```
var newMap = new Urbex.Map('divName', {
  usertoken: 'stringToken',
  orientation: 'NORTH',
  view: 'OBLIQUE',
  x: -523121,
  y: 4960371,
  loadcontrols: 'full'
});
```

```
var style = {
  fillColor: 'yellow',
  fillOpacity: 0.5,

  strokeColor: 'yellow',
  strokeOpacity: 0.7,
  strokeLinecap: 'round',
  strokeWidth: 2,
  pointRadius: 6,
  cursor: ''
};
var textStyle = {
  backgroundColor: "blue",
  color:"red",
  fontFamily:"Arial",
  fontSize: "14px",
  padding: "5px",
  offset:"5,5",
  fontWeight:"bold",
  border:"1px solid green",
  opacity:"0.7"
}
var mesObj = newMap.getMeasurementObj();

mesObj.setLengthStyle(style);
mesObj.setLengthTextStyle(textStyle);
```

20.11 Cleaning measurements displayed on the map

There is a method available in the Measurements object for clearing any measurements displayed over the map, when the measurement is performed via the Urbex tools.

To clear any measurements on the map, use:

```
aMap.getMeasurementObj().cleanMeasures();
```

20.12 Turn off measurements controls.

The method **offMeasureControls** in `Urbex.Measures` will de-activate any measurement control that is active.

```
aMap.get MeasurementObj().offMeasureControls();
```



21 Handlers

Handlers are created and used by controls, which ultimately have the responsibility of making changes to the state of the application. Handlers themselves may make temporary changes, but in general are expected to return the application in the same state that they found it.

21.1 Constructor

Base class to construct a higher-level handler for event sequences:

- **Urbex.Handler**({Urbex.Control} control, {Object} callback)
 - **control:** The control that initialized this handler. The control is assumed to have a valid map property; that map is used in the handler's own setMap method.
 - **callback:** An object whose properties correspond to abstracted events or sequences of browser events. The values for these properties are functions defined by the control that gets called by the handler.

21.2 Properties

- **control:** {Urbex.Control}, The control that initialized this handler. The control is assumed to have a valid map property - that map is used in the handler's own setMap method.
- **keymask:** {Integer}, Use bitwise operators and one or more of the Urbex.Handler constants to construct a keyMask. The mask uses the following constants: MOD_NONE, MOD_SHIFT, MOD_CTRL, MOD_ALT. The result of the operations bit to bit of a constant or of a set of constants they provoke different behaviors for the same handler.

```
// handler only responds if the Shift key is down
handler.keyMask = OpenLayers.Handler.MOD_SHIFT;

// handler only responds if Ctrl-Shift is down
handler.keyMask = OpenLayers.Handler.MOD_SHIFT |
    OpenLayers.Handler.MOD_CTRL;
```

21.3 Methods

- **activate():** Turn on the handler. Returns false if the handler was already active.

- **deactivate():** Turn off the handler. Returns false if the handler was already inactive.

21.4 Constants

- **Urbex.Handler.MOD_NONE:** returns false if any key is down.
- **Urbex.Handler.MOD_SHIFT:** returns false if Shift is down.
- **Urbex.Handler.MOD_CTRL:** returns false if Ctrl is down.
- **Urbex.Handler.MOD_ALT:** returns false if Alt is down.

21.5 Handler types

- **Box:** Handler for dragging a rectangle across the map.
- **Click:** A handler for mouse clicks.
- **Drag:** The drag handler is used to deal with sequences of browser events related to dragging.
- **Feature:** Handler to respond to mouse events related to a drawn feature. Callbacks with the following keys will be notified of the following events associated with features: click, clickout, over, out, and dblclick.
- **Keyboard:** A handler for keyboard events.
- **MouseWheel:** Handler for wheel up/down events.
- **Path:** Handler to draw a path on the map. Path is displayed on mouse down, moves on mouse move, and is finished on mouse up.
- **Point:** Handler to draw a point on the map.
- **Area:** Handler to draw a polygon on the map.
- **Circle:** Handler to draw a circle on the map.

21.6 Example

This example shows the use of the click handler.

```
Project = {};  
  
Project.Click = Urbex.Class(Urbex.Control, {  
  
    initialize: function(options) {
```

BlomURBEX™ Javascript API v5.7 r1.0

```
Urbex.Util.extend(this, options);
Urbex.Control.prototype.initialize.apply(this, arguments);

this.handler = new Urbex.Handler.Click(
    this,
    this,
    {'double': true});
},

click: function(evt) {
    var msg = "click " + evt.xy;
    var output = document.getElementById('console');
    output.innerHTML = msg + '<br />' + output.innerHTML;
},

dblclick: function(evt) {
    var msg = "dblclick " + evt.xy;
    var output = document.getElementById('console');
    output.innerHTML = msg + '<br />' + output.innerHTML;
}
});
```

And the code to use the example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>

    <title>Urban Explorer Basic Example Viewer</title>

    <linkrel="stylesheet"
      href="http://www.blomurbex.com/api/5.7/style.css"
      type="text/css" />
    <script src="http://www.blomurbex.com/api/5.7/Urbex.js"></script>
    <script src="./ControlClick.js"></script>

    <script>

function init() {

    var control = new Project.Click()

    var newMap = new Urbex.Map('divName', {
      usertoken: 'usertoken',
      orientation: 'NORTH',
      view: 'OBLIQUE',
```

BlomURBEX™ Javascript API v5.7 r1.0

```
x: -523121,  
y: 4960371,  
loadcontrols: 'no',  
controls: [control]  
});  
  
control.activate();  
}  
</script>  
</head>  
  
<body onload="init()">  
  <div style="width: 300px; height: 300px; border: 1px solid gray;"  
    id="divName"></div>  
  <div style="width: 300px; height: 300px; border: 1px solid gray;"  
    id="console"></div>  
</body>  
</html>
```



22 Packages and dependencies

This version of the BlomURBEX JavaScript API can now be delivered into a single package with the full functionality (as earlier versions) or split into several packages.

The single package is called as in previous versions of the API:

Urbex.js

If the application being developed does not need all the functionality there are five additional packages:

Urbex_Base.js:

The basic package allows showing maps and updating them programmatically. It offers the functionality to show maps with any view (ortho, ortho rectified and oblique), any orientation (north, south, east and west), and with any overlay layer (i.e. street vector). Additionally, it also offers geocoding capabilities.

When using only this package, no control is available and all operations within the map must be done via scripting.

Urbex_Navigation.js:

The navigation package enables some controls to navigate on the map with the mouse. These controls are: ChangeView, ChangeView2, ChangeOrientation, ZoomIn, ZoomOut, ZoomBar, ZoomBox, DragPan, and Navigation.

Urbex_Overlay.js:

This file adds to the base package the capability to show markers, texts, and draw vector graphics on the map.

Urbex_Measurement.js:

The measurement package adds the controls for measuring on the map (length, height, bearing...). These controls are: Height, Length, Area, Bearing, Elevation, Eraser, DrawFeature, GroundLength and Diagonal.

Urbex_Extra.js:

BlomURBEX™ Javascript API v5.7 r1.0

This package adds some extra controls. These controls are: LayerSwitcher, ReturnPoint, Metadata, GetMap, Scale, North, Year, MultiOblique and ReverseGeocoding.

The dependencies between the different packages are showed in the next diagram:

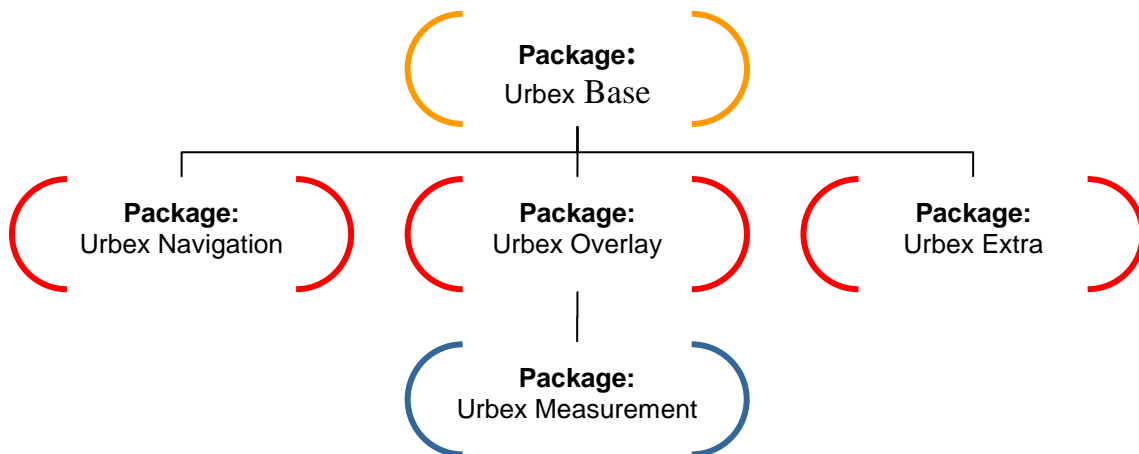


Figure 1: Urbex Packages Dependencies

When developing an application, clients should decide the packages they will need and decide which ones they will load. An example of an application that uses only the navigation tools, but not the measurement tools could be:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/trans.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Urban Explorer Basic Example</title>

    <link
      rel="stylesheet"
      href="http://www.blomurbex.com/api/5.7/style.css"
      type="text/css" />
    <scriptsrc="http://www.blomurbex.com/api/5.7/Urbex_Base.js">
    </script>
    <script src="http://www.blomurbex.com/api/5.7/Urbex_Navigation.js">
    </script>
    <script type="text/javascript">
      function init() {
        var newMap = new Urbex.Map('map', {
          usertoken: 'StringToken',
  
```

BlomURBEX™ Javascript API v5.7 r1.0

```
x: -523121,  
y: 4960371,  
loadcontrols: 'navigation'}};  
}  
</script>  
</head>  
  
<body onload="init()">  
  <div style="position: relative; width: 800px; height: 550px; top: 0px;  
left: 0px; z-index: 1" id="map"></div>  
  </body>  
</html>
```



23 Proxy configuration

BlomURBEX services can be piped through a proxy to allow the application developers to add non functional requirements.

The default BlomURBEX configuration allows direct communication between the user's machine and the BlomURBEX servers. This is the best approach for free applications that expect much traffic. In these cases a proxy will mean more costs to the application provider (on hardware and bandwidth needs). On the other hand, restricted applications will want to have much more control on the use of the service. The proxy approach will be the best option to these applications. Using this technique the application provider may add authentication, authorization, logging and any needed business logic.

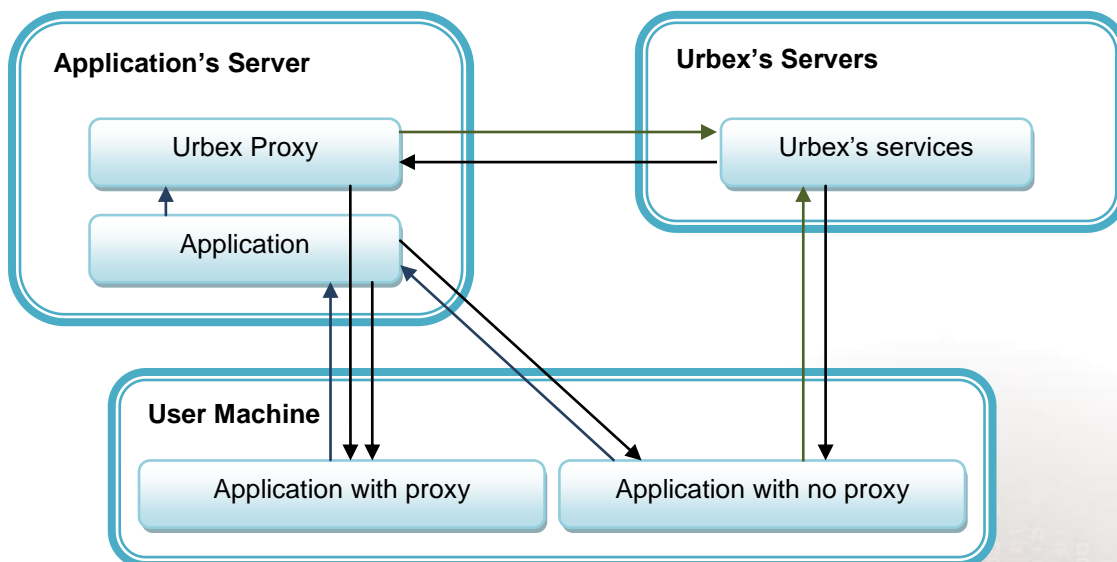


Figure 2: Proxy server configurations

In the above diagram, differences between the direct approach and the proxy approach can be clearly seen. Communication through the proxy is secure because the user's machine never should have a valid user token. The valid token is provided by the proxy.

23.1 Urbex.Proxy

Urbex.Proxy defines a proxy. This class is used by the map to choose the target of its requests. To create a new object of Urbex.Proxy two parameters must be passed:

- Proxy server address.

- Proxy Level (see below).

23.2 Proxy levels

As not all BlomURBEX functionalities may be critical for applications providers there are several proxy levels defined. A proxy level is defined by the kind of requests piped by the proxy. The different proxy levels are:

- **Urbex.Proxy.NO_PROXY**: This is equivalent to not configuring any proxy. No requests will be routed through the proxy.
- **Urbex.Proxy.DOWNLOAD_PROXY**: This is the lower level proxy. Only GetMap and ExtractOblique requests will be routed.
- **Urbex.Proxy.NO_TILE_PROXY**: With this configuration all requests except tile requests will be routed.
- **Urbex.Proxy.FULL_PROXY**: A full proxy will pipe all requests.

23.3 Proxy setup in the client side

The proxy must be provided to the map as an option via the **Urbex.Proxy** class.

When using the full proxy approach, it is important to remove any valid token from the source to prevent any misuse of the token.

To provide an invalid token to the map the value **Urbex.Proxy.DUMMY_TOKEN** should be used.

The following code shows an example of proxy setup.

```
var blomView = new Urbex.Map(  
  'map',  
  {  
    zoom: 14,  
    isIntelligent: true,  
    x: -523121, // ávila mercator  
    y: 4960371, // ávila mercator  
    view: 'ORTHO',  
    orientation: 'ORTHO',  
    loadcontrols: 'full',  
    usertoken: Urbex.Proxy.DUMMY_TOKEN,  
    proxy: new Urbex.Proxy(  
      'www.clienthost.com/proxy.php',  
      Urbex.Proxy.FULL_PROXY)  
  } // options  
);
```


If the application does not use a full proxy approach, Blom will provide two tokens: one for the application's server and one for the client scripts, with different access permissions to each one of them. In these cases the map must be configured with the proper user token.

```
var blomView = new Urbex.Map(  
  'map',  
  {  
    zoom: 14,  
    isIntelligent: true,  
    x: -523121, // ávila mercator  
    y: 4960371, // ávila mercator  
    view: 'ORTHO',  
    orientation: 'ORTHO',  
    loadcontrols: 'full',  
    usertoken: 'client_side_token',  
    proxy: new Urbex.Proxy(  
      'www.clienthost.com/proxy.php',  
      Urbex.Proxy.DOWNLOAD_PROXY)  
    } // options  
);
```

23.4 Proxy setup in the server side

With the BlomURBEX API two proxies are provided: one php-coded, and another one coded in asp.net. Developers can use any of these proxies directly in the applications, or they can use them as a start point of a proxy that fulfills the application needs.

The proxy code can be found in the following URLs

- PHP Proxy

<http://www.blomurbex.com/api/5.0/proxy/php.zip>

- ASP Proxy

<http://www.blomurbex.com/api/5.0/proxy/asp.zip>

23.4.1 The PHP Proxy

PHP proxy has three files:

First file is **blom_urbex_proxy/class_http.php** that defines blom_urbex_http class. This class is internally used by blom_urbex_proxy class.

Second file is **blom_urbex_proxy/class_proxy.php** that defines blom_urbex_proxy class. This class is the proxy class that should be used to pipe the request from the application servers to blom servers.

The request to blomUrbex server is done through the pipeRequest method (see the example).

Finally **blom.ini** is a configuration file where the developer should write the blom server's address and the application's user token. This is a standard ini file with two sections: server and user, and one property in each section: address and token (see the example below).

The long names of the classes have been chosen for compatibility purposes.

23.4.1.1 *A basic example for using the provided proxy class in php*

An example of a client's proxy developed based on class_proxy.php:

```
<?php
require_once("blom_urbex_proxy/class_proxy.php");
session_start();
if ($proxy = new blom_urbex_proxy()) {
    if (!$proxy->pipeRequest()) {
        // do some logging, proxy has written the response
    } else {
        // do some logging, proxy has written the response
    }
} else {
    header("HTTP/1.0 501 Script Error");
    echo "ERROR, cannot instantiate proxy class";
}
?>
```

The code of class_urbex.ini file:

```
; blom's servers settings
[server]
address=www.blomurbex.com
; user settings
[user]
token=privateToken
```

Finally the application page (see 16.3 Proxy setup in the client side):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/trans.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

BlomURBEX™ Javascript API v5.7 r1.0

```
<head>
<title>Urban Explorer Basic Example Viewer</title>
<linkrel="stylesheet"
      href="http://www.blomurbex.com/api/5.7/style.css"
      type="text/css" />
<script src=' http://www.blomurbex.com/api/5.7/Urbex.js'></script>
<script type='text/javascript'>

var blomView;

function init(){

blomView = new Urbex.Map(
  'map',
  {
    zoom: 14,
    isIntelligent: true,
    x: -523121, // ávila mercator
    y: 4960371, // ávila mercator
    view: 'ORTHO',
    orientation: 'ORTHO',
    loadcontrols: 'full',
    controls:
    [
      new Urbex.Control.LayerSwitcher(),
      new Urbex.Control.ReturnPoint(),
      new Urbex.Control.GetMap(),
      new Urbex.Control.North(),
      new Urbex.Control.ShowStreets(),
      new Urbex.Control.Metadata()
    ],
    usertoken: 'publicToken',
    proxy: new Urbex.Proxy(
      'www.youraddress.com/proxy.php',
      Urbex.Proxy.FULL_PROXY)
  } // options
);

</script>
</head>

<body onload='init()'>

<div>
<div style='width: 60%; height: 400px'>
```

```
<div
  id='map'
  style='width:100%; height:400px'></div>
</div>

</div>
</body>
</html>
```

23.4.2 The ASP Proxy

The ASP proxy defines one class only. **blom.urbex.Proxy** is parameterized through its constructor and pipes the request using the **pipeRequest** method (see the example).

23.4.2.1 A basic example using asp.net

Create a web project using visual studio and include the file Proxy.cs that defines the class blom.urbex.Proxy. Then create a handler to process the urbex requests (ie. ProxyHandler), so the code of this handler should be at least:

```
using System;
using System.Web;

namespace com.company.project.package
{
    public class ProxyHandler : IHttpHandler
    {
        public void ProcessRequest(HttpContext context)
        {
            blom.urbex.Proxy proxy = new blom.urbex.Proxy(
                "www.blomurbex.com",
                "token");
            proxy.pipeRequest(
                context.Request,
                context.Response);
        }

        public bool IsReusable
        {
            get
            {
                return true;
            }
        }
    }
}
```

```
}
```

The application page is identical to the php application page, only the proxy option must be changed:

```
proxy: new Urbex.Proxy(  
  'www.youraddress.com/ProxyHandler.ashx',  
  Urbex.Proxy.FULL_PROXY)
```



24 Release Notes

24.1 Changes in version 5.5

This new release of the JavaScript API offers new functionalities:

- New `Urbex.Util.GetUserAvailableProviders` function to deliver all available external providers.
- New Layer WFS for WFS connection to external servers.
- Changes in Routing API.
- Improvement of geocoding documentation.
- New Layer Vector Editable
- New `LayerSelectorControl`



Third parties copyright notice

This JavaScript API is based in part of the code of the OpenLayers library.

Copyright (c) 2005-2008 MetaCarta, Inc.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted (subject to the limitations in the disclaimer below) provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- * Neither the name of MetaCarta, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

NO EXPRESS OR IMPLIED LICENSES TO ANY PARTY'S PATENT RIGHTS ARE GRANTED BY THIS LICENSE. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.