Sasha Maps

Alexander Maryanovsky

January 25, 2013

Contents

1 Introduction

Sasha Maps is a GWT library similar in capabilities to OpenLayers, or the client-side of Google Maps. The core functionality is displaying and interacting with a map. A map consists of a set of huge images, each image represents a "zoom" at which the user may view the map. The larger the zoom, the larger and more detailed is the image representing it. Each such image is cut by the server into fixed-size images (tiles), and served this way to the client. This allows the client to download only the tiles it needs, instead of the entire image, as normally only a small portion of it is visible to the user.

Overview of the major features

- Displaying a tiled map. The library has built-in support for the OpenStreetMap tile protocol and *Tiled WMS* maps. To display your own maps, you need only to implement a few Java interfaces, as described in the following chapters.
- The user may interact with the map using any of the standard ways, such as dragging, zooming in and out via the mouse scroll wheel, recentering on a clicked spot, zooming into a clicked spot and others. Touch gestures for mobile devices (iOS, Android) are supported as well - double tapping to zoom, pinching etc.
- A developer may implement his own, custom ways for the user to interact with the map.
- Overlays may be added to, and removed from, the map. Overlays are widgets which are anchored at a specified location on the map. These may be used to mark certain locations on the map, such as points of interest.
- Widgets may be added to the map widget. These, unlike overlays, are positioned relative to the map widget, rather than relative to the map. These may be used to add buttons, labels, zoom controls etc. to the map widget.
- Several useful map widgets are provided:
  - An overview map widget which displays the map several zooms out, and highlights the portion visible on the main map.
  - A zoom widget which shows the current zoom and lets the user to quickly zoom in and out, or jump to a specific zoom.
  - A scale widget which displays the scale of view (i.e. how many real-world distance units, such as kilometers, there are in a certain amount of pixels).
- Locations on the map are described by their latitude and longitude, similarly to Google Maps.
- Supports all major browsers, including Internet Explorer 6/7/8, Firefox, Opera and all WebKit-based browsers (Safari, including Mobile Safari on iOS devices, Chrome etc.)
- Very fast and responsive.
- Small footprint, meaning short download times for the user. A demo application showing most of the features compiles to about 160K of JavaScript. A simple interactive map takes 110K.

2 Hello, World

Below is a small GWT application, demonstrating how to set up Sasha Maps. In the demo, we will use OpenStreetMap tiles and their protocol.

```
package com.maryanovsky.mapdemo.client;

// Imports removed for brevity

/**
 * The entry point of the Map demo.
 */

public class TutorialDemo implements EntryPoint{


    /**
     * Starts the map demo.
     */

    public void onModuleLoad(){
        TileLayer tileLayer = new OsmTileLayer("http://tile.openstreetmap.org", 0, 16);
        Map map = new Map(OsmMercatorProjection.INSTANCE, tileLayer, 0, 16);

        LatLng initialLocation = new LatLng(60.050317, 30.350161);
        int initialZoom = 13;

        MapWidget mapWidget = new MapWidget(
            new MapLocationModel(0, 16, initialLocation, initialZoom));
        mapWidget.setMap(map);

        StandardActions.getInstance().addAll(mapWidget);

        UiUtils.addFullSize(RootPanel.get(), mapWidget);
    }


}
```

I will assume the reader is already familiar with Java and GWT and will avoid explaining the boilerplate. Let's start, then:

```
TileLayer tileLayer = new OsmTileLayer("http://tile.openstreetmap.org", 0, 16);
```

The tile layer we will display. Here we will use a single tile layer - from the OpenStreetMap project. You can also display several stacked tile layers on the same map, but for simplicity, we will use just one here. We specify that the minimum zoom of the tile layer will be 0 and the maximum 16.

```
Map map = new Map(OsmMercatorProjection.INSTANCE, tileLayer, 0, 16);
```

Create the map. Note that, unlike in Google Maps API, the `Map` object is not the widget seen by the user - it is merely a description of the map. It consists of the projection, the list of tile layers, and the range of supported zooms. In our example, we use OSM's implementation of the Mercator projection, the tile layer we obtained in the previous line, and a zoom range of 0 to 16. The reason we have to specify the zoom range again is that a map, potentially consisting of a collection of tile layers, may allow a different range of zooms than a single of its tile layers.

```
LatLng initialLocation = new LatLng(60.050317, 30.350161);
int initialZoom = 13;
```

Here we define the initial location of the center of the map, specified via its latitude and longitude coordinates, and the initially displayed zoom.

```
MapWidget mapWidget = new MapWidget(
    new MapLocationModel(0, 16, initialLocation, initialZoom));
```

We create a new `MapWidget`, the widget which will display the map. We initialize it with a location model bound to the zoom range 0 to 16, and with the initial position at our chosen location and zoom. Note that again, we specify the zoom range. We could have specified a smaller range, if we wanted to restrict the user to less than all the supported zooms.

```
mapWidget.setMap(map);
```

The map widget is set to display the map we created earlier.

```
StandardActions.getInstance().addAll(mapWidget);
```

We set all the standard user actions to perform their standard function on the map widget. For example, dragging will pan the map, while double-clicking will zoom-in and recenter on the clicked point. Note that the actual actions and their corresponding functions depend on the target platform. When running on the iPhone, for example, the user actions will be tapping, pinching etc., and the corresponding actions will be different.

If you wish to have more fine-grained control over the actions, explore the `actions` subpackage. You can also implement your own actions, which is explained later in this tutorial.

```
UiUtils.addFullSize(RootPanel.get(), mapWidget);
```

Finally, we add the map widget to the root panel of the containing HTML page, at 100% of its size.

3 Custom Maps

In the demo above, we have referred to OpenStreetMap twice - once when we created the tile layer (`OsmTileLayer`) and again when we needed the projection (`OsmMercatorProjection`). These are ready implementations of interfaces which, if you wish to display your own maps, you need to implement yourself. These two interfaces are `TileLayer` and `Projection`. The `TileLayer` interface defines how the map is cut into tiles and is responsible for creating the tile objects. The `Projection` interface defines how latitude-longitude coordinates translate to pixel coordinates (and vice versa) within the large map image for each zoom.

3.1 TileLayer

```
public interface TileLayer{
    SizeView getTileSize();
    Tile getTile(int x, int y, int zoom);
    void releaseTile(Tile tile);
}
```

As you can see, the `TileLayer` interface defines three methods - `getTileSize`, which specifies the size of each tile (in pixels), `getTile`, which returns the `Tile` at a specified location, and `releaseTile`, which is invoked to perform any clean up related to the `Tile` when it's no longer needed.

The getTile method takes pixel coordinates relative to the top-left of the map at the specified zoom, and returns an instance of the abstract class `Tile`. If you want to implement tiles that do something special other than simply displaying an image, you will want to look into subclassing `Tile`. Usually, however, you will want to use its implementation - `ImageTile` from the `tiles` subpackage. All `ImageTile` needs is the URL of the tile. Also, if you are using `ImageTile`, you may leave the `releaseTile` method implementation empty, as `ImageTile` doesn't require any special cleanup.

An implementation of `TileLayer` might look like this:

```
import com.maryanovsky.map.client.*;
import com.maryanovsky.map.tile.ImageTile;
import com.maryanovsky.gwtutils.client.geom.*;

public class ExampleTileLayer implements TileLayer{

    private static final SizeView TILE_SIZE = new Size(256, 256);

    public SizeView getTileSize(){
        return TILE_SIZE;
    }

    public Tile getTile(int x, int y, int zoom){
        if ((zoom < 0) || (zoom > 5)) // Outside the supported zoom range
            return null;

        // Calculate tile indices
        x /= TILE_SIZE.getWidth();
        y /= TILE_SIZE.getHeight();

        // Create tile image URL
        String tileUrl = ‘‘http://tiles.example-map.com/tile?’’ +
            ‘‘x=’’ + x + ‘‘&y=’’ + y + ‘‘&zoom=’’ + zoom;

        // Create and return the tile
        return ImageTile.makeImageTile(tileUrl);
    }

    public void releaseTile(Tile tile){
        // No cleanup needed for ImageTile
    }

}
```

ImageTileLayer

If you are indeed implementing a tile layer where each tile is a single image identified by its index in a grid, `ImageTileLayer` will do almost all of the hard work, leaving you only to specify the tile size and image URLs. Here's how the above example tile layer would look were we to subclass `ImageTileLayer`:

```
import com.maryanovsky.map.client.*;
import com.maryanovsky.gwtutils.client.geom.*;

public class ExampleTileLayer extends ImageTileLayer{

    public ExampleTileLayer(){
        super(new Size(256, 256),
          "http://tiles.example-map.com/tile?x={x}&y={y}&zoom={zoom}");
    }

    public String getTileUrl(int xIndex, int yIndex, int zoom){
        if ((zoom < 0) || (zoom > 5)) // Outside the supported zoom range
            return null;

        return super.getTileUrl(xIndex, yIndex, zoom);
    }
}
```

A few things to note in the above code:

- The second argument to `ImageTileLayer`'s constructor is a pattern for the tile image URL. As you might've guessed, the `{x}`, `{y}` and `{zoom}` substrings in it are replaced with the corresponding values for each tile.
- Actually, the values for `{x}` and `{y}` are *not* the x and y pixel coordinates, but the indices of the tile in the tile grid of the corresponding zoom. They are obtained from the x and y pixel coordinates by diving the coordinates by the tile width and height, respectively.
- The `getTileUrl` method's `xIndex` and `yIndex` arguments are those very indices.
- Although not shown in this example, in addition to checking the zoom range, it's recommended to also check whether `xIndex` and `yIndex` are in the appropriate range. It's up to you whether to return null or wrap around the tile grid if these indices are outside the expected range.

There's actually a neat feature to the URL pattern that isn't shown above - you can use it to get `ImageTileLayer` to generate semi-random tile URLs. You do this by putting a list of values inside square brackets as part of the pattern; only one of these values will then be picked when generating the URL. For example, the pattern `http://tiles[1,2].example-map.com` could end up generating either `http://tiles1.example-map.com` or `http://tiles2.example-map.com`. The selected value will be uniformly distributed, but consistent across different invocations.

Why would you need such a thing? Most browsers limit the number of simultaneous connections they are willing to open to a single domain. This, however, is highly problematic for tile-based mapping applications, because it prevents many tiles from being downloaded simultaneously (and thus quickly), even when the bandwidth allows it. The workaround is to set up several domain names (that usually all point to the same actual server) that allow access to the tiles, and have the client pick one of them for each tile. You want the selected domain to be consistent, though, so that each tile image can be cached after the first time it's downloaded. This is exactly what this feature lets you do!

3.2 Projection

```
public interface Projection{
    Point fromLatLngToPixel(LatLng latLng, int zoom);
    LatLng fromPixelToLatLng(PointView pixel, int zoom);
    double getZoomMagnification(int startZoom, int endZoom);
    SizeView getWrapSize(int zoom);
}
```

The two methods `fromLatLngToPixel` and `fromPixelToLatLng` convert between latitude-longitude and pixel coordinates. Pixel coordinates are relative to the top-left of the large image for each zoom, and they increase down and to the right, just like normal images do.

The `getZoomMagnification` method returns (approximately) how much larger each pixel is at `startZoom` than at `endZoom`, in terms of the real-world distance (meters) covered by it. With OpenStreetMap, for example, each successive zoom increases the pixels per meter scale by two, so the implementation for it simply returns `Math.pow(2.0, endZoom - startZoom)`.

The `getWrapSize` method is needed for displaying maps that wrap around on either the X or the Y axis. For example, most interactive maps of the Earth will seamlessly wrap at the $\pm 180^{\text{o}}$ longitude line. This method should return the number of pixels, at the given zoon, after which the map wraps. The width property of the returned `SizeView` object corresponds to the wrap on the X axis; height to the Y axis. A value of 0 in one of the properties indicates that the map does not wrap on the corresponding axis. Most earth maps, for example, do not wrap on the Y (longitude) axis.

AbstractProjection

Many projections are implemented in a similar way, and to avoid repeating this code in each one, Sasha Maps provides a convenient partial (abstract) implementation - `AbstractProjection` in the `projections` subpackage. `AbstractProjection` divides the responsibility of handling the projection itself and the zoom magnification between two separate objects. Projecting is delegated to two abstract methods - `fromLatLngToPixelImpl(LatLng)` and `fromPixelToLatLngImpl(double, double)`, which need only to project at some predefined "native" zoom. The definition of zoom magnification is delegated to an implementation of a new interface, `ZoomStrategy`, which has a single method, `getZoomMagnification`, with the same semantics as `Projection.getZoomMagnification`.

The `projections` subpackage also provides some ready implementations of commonly used projections:

- `ScaleProjection`, which simply scales the $[-180, 180] \times [-90, 90]$ range of legal latitude-longitude values to the size of the map at each zoom.
- `MercatorProjection`, which implements the commonly used mercator projection.

Additionally, two commonly used implementations of `ZoomStrategy` are provided:

- In `FixedCoefZoomStrategy`, each zoom magnifies by a constant factor more than the previous one.
- `ResolutionListZoomStrategy` takes a list of resolutions (amount of real-world units per pixel), one for each zoom, and uses them to calculate the magnifications between the zooms.

4 Custom Actions

The demo application assigned standard actions for interacting with the map (such as dragging a map, zooming in/out), to certain user actions (dragging, clicking, etc.). On touch-enabled devices, StandardActions will assign these actions to the appropriate touch events instead (this can be controlled via the `interactionMode` compile-time GWT property). Now we will see how you can implement your own, custom, actions.

4.1 On the Desktop

`MapWidget` exposes a source of low-level mouse events via the `getMouseEventsSource()` method. You can simply register a mouse handler with that source and perform whatever custom action you want in your handler. GwtUtils, however, provides a convenient set of "gesture recognizers" which listen to these low-level mouse events and recognize certain high-level gestures. For example, instead of implementing the logic of what sequence of mouse down/move/up events constitutes a drag gesture, you can create a `DragRecognizer` (from the `com.maryanovsky.gwtutils.client.ui.mouse.gestures.drag` package), register it as a handler of mouse events from the mouse events source and then register your code as a handler of drag events with the recognizer. Here, for example, is how the standard action for dragging the map could be implemented:

```
DragRecognizer dragRecognizer = new DragRecognizer(mapWidget));
dragRecognizer.setMouseEventsSource(mapWidget.getMouseEventsSource());
dragRecognizer.addDragMoveHandler(new DragMoveHandler(){
    public void onDragMove(DragMoveEvent evt){
        mapWidget.setTemporaryOffset(evt.getOffset().getOpposite());
    }
});
dragRecognizer.addDragEndHandler(new DragEndHandler(){
    public void onDragEnd(DragEndEvent evt){
        mapWidget.applyTemporaryOffset();
    }
});
```

Additionally, all the recognizers can be customized with a `Condition` parameter that allows them to "report" only a subset of the gestures they recognize. For example, you may wish to receive only single-clicks with the right mouse button. `ClickRecognizer`, by default, recognizes and reports clicks with any mouse button, and with any amount of clicks. You could simply inspect the `ClickEvent` yourself in the handler, but a better way is to pass a `Condition` argument to the recognizer which will inspect the `ClickEvent` before it has been sent to handlers (and only return `true` for single-right-clicks). The latter way is cleaner, as it separates the gesture-detection code from the action taken upon it. See the documentation of the various recognizers for more information.

4.2 On Mobile, Touch Devices

Handling and responding to touch events is very similar to handling mouse events on the desktop. `MapWidget` exposes the source of low-level touch events with `getTouchEventsSource()`. To this you attach gesture recognizers from one of the `com.maryanovsky.gwtutils.client.ui.touch.gestures` subpackages. You then register your own gesture handlers with the gesture recognizers. Touch gesture recognizers sport the same filtering of gestures via a `Condition` parameter as mouse gesture recognizers.

5 Overlays

Overlays are entities which are notified whenever the location displayed by a map widget changes (even during a temporary change, such as a drag gesture). Normally, an overlay adds some sort of widget to the map widget's overlay panel and uses these notifications to update its location, so that the widget appears to be attached to the map. In fact, the `WidgetOverlay` class, which implements the `Overlay` interface, does just that with any widget you care to give it a reference to.

Let's take a look at `Overlay` first:

```
public interface Overlay{
        void added(MapWidget mapWidget);
        void updated(MapWidget mapWidget, boolean isTemporary);
        void removed(MapWidget mapWidget);
}
```

All of Overlay's method are notification methods. `added` and `removed` are invoked when the overlay is added and removed from the map respectively, while `updated` is invoked when the map widget's location changes (the `isTemporary` parameter specifies whether the change is temporary, such as during a drag-map gesture, or permanent, such as when the drag gesture ends). When the `added` and `removed` methods are invoked, the overlay is expected to add/remove whatever the widgets are that the overlay places on the map (an image, for example) to/from `mapWidget.getOverlayPanel()`. When `updated` is invoked, the overlay should adjust the location of its widgets, or remove them altogether, if they strolled outside the visible portion of the map (and add them back again, if they strolled back in).

Mostly, however, you will want to use the convenient `WidgetOverlay` class, which does all the work for you. It just needs the widget you would like to place on the map, its location (in latitude-longitude) and the offset of the widget's top-left point, in pixels, from that point. The offset is needed, if for example, you want to put an image of an arrow on the map, and you want the arrow tip, rather than the top-left of the image, pointing at your sweetheart's house. Additionally, WidgetOverlay provides convenient methods for making the widget draggable. `WidgetOverlay.makeOffsetDraggable`, for example, will put the overlay into a mode where its offset changes when the widget is dragged, while `WidgetOverlay.makeAnchorDraggable` makes it adjust its anchor instead (the difference between the two modes becomes obvious when you consider what happens when the zoom changes).

5.1 Interactive Widgets

If you are adding user-interactive widgets to the map (such as a button, text box, draggable widget etc.), you will need at the very least to prevent its events from bubbling to the `MapWidget`. If you don't, the map widget will receive and respond to the event as usual. With a draggable widget, for example, it will mean the map will move along with your widget when you drag it, which is probably not what you intended.

Additionally, you may need to prevent the events' default action from occuring. Whether this is needed or not depends on the widget. For a draggable image, for example, you would want to prevent the default action (which is normally to select the image and/or drag it outside the browser). A text box, on the other hand, needs its default mouse/click action to become focusable.

One way to prevent bubbling and the default action, is to override the `onBrowserEvent(Event)` method of your widget and invoke `stopPropagation()` and/or `preventDefault()` manually. A more convenient way is to place all of your widget's UI in an `EventCancellingPanel` (from the `ui` subpackage of GwtUtils).

Note that this applies to both widgets within overlays and regular widgets added to the map (as explained in the next section).

6 Widgets on the Map

`MapWidget` provides you with a panel overlaying the map widget, placed at its origin and sized to its dimensions. Through this panel, which you obtain via `mapWidget.getWidgetPanel(),` you can add widgets to the map widget which do not move when the map's location changes. To place a widget in one of the map widget's corners, use the `UiUtils.addInCorner` method of the GwtUtils library. *Important:* the note regarding interactive widgets on this page applies to regular widgets as well (the standard widgets provided by the library, however, already manage their events as expected).

The library provides some widgets commonly used in interactive maps which you can find in various subpackages of the `com.maryanovsky.map.client.widgets` package. Many of these widgets can be completely customized with regards to how they look. They do, however, provide a default look, and to properly use that, you will need to import their CSS into your page's HTML. You can do this by importing either the individual widget's css file or the "widgets/widgets.css" file, which in turn imports all of the widgets' css files. The individual widget's css file is named after the widget's subpackage and is found in the directory corresponding to its subpackage relative to the `com.maryanovsky.map.client` package. For example, the scale widget's package is `com.maryanovsky.map.client.widgets.scale` and its css file is therefore "widgets/scale/scale.css".

6.1 Overview Map Widget

`com.maryanovsky.map.client.widgets.overview.OverviewMapWidget` is a subclass of `MapWidget` which follows the location of another, "main", `MapWidget`. To create an `OverviewMapWidget`, you must give its constructor the main `MapWidget`, and an object defining the meaning of "follow" - a `Synchronization` (an inner static interface of `OverviewMapWidget`). I will not go into the details of what `Synchronization` does, as you will rarely want to implement it yourself. Instead, use its friendly implementation, the `StandardSynchronization` (also an inner static class of `OverviewMapWidget`).

`StandardSynchronization`'s constructor takes two parameters - the first determines how far zoomed out the overview map will be, relative to the main map, and the second, whether the overview map widget will follow its main counterpart using animated or non-animated panning. The second parameter is obvious, but the first one requires a little explaining. How far the overview map is zoomed out is determined dynamically, by comparing the sizes of the two map widgets and by making sure that a certain portion of the area (of the world) displayed by the overview map is covered by the area displayed by the main map widget. The exact (relative) size of that portion is the first argument to `StandardSynchronization`'s constructor. If you pass `0.5`, for example, the overview map widget will display at the largest zoom such that the area displayed by the main map widget will occupy at most half of the overview widget. If you don't understand the above, or just don't care, use the recommended value of `0.4` or simply invoke the one-argument constructor.

Here's an example that creates an `OverviewMapWidget` and adds it to a corner of a `mapWidget`:

```
MapWidget overviewMapWidget = new OverviewMapWidget(mapWidget,
    new OverviewMapWidget.StandardSynchronization(0.4, true));
overviewMapWidget.setMap(mapWidget.getMap());
overviewMapWidget.setSize("150px", "150px");
DesktopStandardActions.addDragPan(overviewMapWidget);
UiUtils.addInCorner(mapWidget.getWidgetPanel(), overviewMapWidget,
    Alignment.BOTTOM_RIGHT, new Size(0, 0));
```

6.2 Zoom Widget

`com.maryanovsky.map.client.widgets.zoom.LargeZoomWidget` shows the user what zoom the map is currently at, and lets him zoom in or out and to quickly jump to a desired zoom. By default it looks similarly to Google Maps' zoom widget, but can be customized however you like. To use the default look, you need to import the "widgets/zoom/zoom.css" file into your page's CSS. Other than that, using the zoom widget is straightforward:

```
LargeZoomWidget zoomWidget = new LargeZoomWidget(mapWidget);
UiUtils.addInCorner(mapWidget.getWidgetPanel(), zoomWidget,
    Alignment.TOP_LEFT, new Size(0, 0));
```

To customize the look of a `LargeZoomWidget`, you must implement the `LargeZoomWidget.Images` interface and pass an instance of the implementing class to the appropriate `LargeZoomWidget` constructor. To implement the interface, you need 6 images:

- The zoom in and zoom out buttons.
- The top and bottom parts of the zoom "ladder".
- The "step" of the zoom "ladder". This part is repeated for every zoom level.
- The "mark" image. This is the image indicating the current zoom.

Of course these images need to blend together, to provide an illusion of a seamless widget.

6.3 Scale Widget

`com.maryanovsky.map.client.widgets.scale.ScaleWidget` is a non-interactive widget which shows the user the scale of the map; that is, how many real-world distance units there are in a certain amount of pixels currently visible on the map. `ScaleWidget`'s constructor takes two arguments: the map widget and a `ScaleWidget.Scale`. The 2nd argument is an interface which defines the size of the scale widget in pixels and the text it displays. If your application is showing a map of the Earth, however, you can use an existing implementation - `EarthScale`. You can either instantiate `EarthScale` yourself (it takes the preferred size of the widget, a list of sizes in meters it may display and a list of size names to display) or use one of the static method which give you an instance. There are currently two such methods: `getEnglishMetricInstance(int)` for metric units and `getEnglishImperialInstance(int)`, for imperial units.

Here's an example which creates and adds a scale widget which uses metric units and a preferred size of 100 pixels:

```
ScaleWidget scaleWidget =
    new ScaleWidget(mapWidget, EarthScale.getEnglishMetricInstance(100));
UiUtils.addInCorner(mapWidget.getWidgetPanel(), scaleWidget,
    Alignment.BOTTOM_LEFT, new Size(16, 16));
```

6.4 Bubble Widget

`com.maryanovsky.map.client.widgets.bubble.BubbleWidget` is a comics-style "talk" bubble which can display arbitrary content. It is usually used to show information about a certain location on the map with "leg" of the bubble pointing to that location. Using a `BubbleWidget` is very straightforward:

```
Label content = new Label(‘‘Hello, Paris!’’);
BubbleWidget bubble = new BubbleWidget(content);
bubble.setAnchor(new LatLng(48.856667, 2.350833));
mapWidget.addOverlay(bubble);
```

As you can see, we create the content widget (which can be anything, but in the example we use a simple `Label`), then create the bubble, set its anchor and add it as an overlay to the map.

It is possible to customize the looks of the `BubbleWidget` in several ways.

- To use a different image for the bubble "leg", you need to implement the `BubbleWidget.Images` interface and pass an instance of it to the `BubbleWidget(Images, Alignment, Widget)` constructor. The alignment argument specifies how the bubble is aligned relative to its anchor. In the default looks, for example, it is `Alignment.TOP_RIGHT`.
- Minor changes to the looks of the bubble can be made by changing the CSS of its various elements. The style names are documented in the class's javadoc comment.
- More significant changes can be made by overriding the `createContentHolderWrapper(Widget)` method. This allows you to place the content holder panel (where the content goes) into other panels, creating whatever look you may want.

7 Map Widget Events

`MapWidget` provides fairly standard (in the Java world) methods for listening to changes in its properties. Properties such as the displayed map, the size of the widget and the temporary offset of the map (the one used during a drag gesture) can be tracked by registering a `PropertyChangeListener` (although not of the `java.beans` variety, as that package did not make it into GWT, but from the `com.maryanovsky.gwtutils.client.event` package). The location of the map, including the zoom, is encapsulated by the `MapLocationModel` class, which lets you register a `ChangeListener` (again, from the `com.maryanovsky.gwtutils.client.event` package) with it.

8 Tile Protocols

Sasha Maps has built-in implementations for some of the most popular tile protocols: Web Map Service and OpenStreetMap.

8.1 Web Map Service (WMS)

One of the tile protocols Sasha Maps supports is the widely used Web Map Service protocol (the tiled variant defined by the WMS Tiling Client Recommendation - that means your WMS server's `GetCapabilities` response must include a `TileSet` element under `VendorSpecificCapabilities`). Support for WMS maps can be found in the `opengis.wms` subpackage. WMS maps are a bit more complicated to use than OpenStreetMap ones, but fortunately the library hides most of this complexity from you.

To use a WMS map you first need to obtain the `Capabilities` object describing the service's capabilities. If you are using your own WMS server, this is not a problem - you just send an HTTP request and let the library parse the response and pass you back the `Capabilities` object. If you are using someone else's WMS server or if your WMS server is at a different host than your maps client, the same-host security restriction kicks in, and you will be unable to send a request to the WMS server without a proxy. Fortunately, there are many free proxies available on the web, and it is even not very difficult to write your own. Of course you can also simply grab the XML document that is sent in response to a WMS Capabilities request and put it as a regular XML file on your web server.

Assuming you will be querying a real WMS server for its capabilities, you will need to use the `Capabilities.makeGetRequestUrl` method to build the full URL of the request from the base URL of the service. Upon receiving the response, you can create a `Capabilities` object by passing the response XML document to the `Capabilities` constructor. Alternatively, you can use the `XmlResponseHandler` implementation, `Capabilities.ResponseHandler`, to do all the work for you, including error handling.

Altogether, here is how it would look:

```
try{
    ResultCallback<Capabilities> callback = new ResultCallback<Capabilities>(){
        public void requestFinished(){}
        public void resultReceived(Capabilities capabilities){
            showMap(capabilities);
        }
    };

    String url = Capabilities.makeGetRequestUrl("http://mymaps.com/wms-c/");
    RequestBuilder requestBuilder = new RequestBuilder(RequestBuilder.GET, url);
    requestBuilder.sendRequest(null,
        new Capabilities.ResponseHandler(url, callback));
} catch (RequestException e){
    Window.alert(e.getMessage());
}
```

With a `Capabilities` object in hand, you need to query it for the `TileSet` you want to display. There are two ways to do this. The `Capabilities.getTileSets` method simply returns the list of `TileSet`s defined by the service. If you know the name of the layer you want to display, however, there is a more convenient method - `Capabilities.getTileSetsByLayerName(String)`. This method will find and return all the `TileSet`s which display the specified layer (normally there is just one), sorted by the number of layers displayed in the `TileSet`, in increasing order. This lets you pick the `TileSet` with the least number of unwanted layers:

```
TileSet tileSet = capabilities.getTileSetsByLayerName("basic").get(0);
```

Armed with the `Capabilities` object and the right `TileSet`, you can now create the `Map`:

```
Map map = Wms.createMap(capabilities, tileSet, true);
```

The last argument to `Wms.createMap` specifies whether the map will wrap at the maximum longitude. Now that you have a `Map` object, you create a `MapLocationModel` and a `MapWidget` the same way you normally would (query the map for the supported zoom range).

Note that `Wms.createMap` only works if the SRS of the `TileSet` is supported (currently, EPSG:4326 and OSGEO:41001 are supported). If it isn't, you have more work ahead of you in implementing `Projection`. You can then, however, use `TileSetLayer` for your `TileLayer`, so at least that part has been taken care of.

8.2 OpenStreetMap

The demo application described in this tutorial uses OpenStreetMap tiles and there's not much more to it, so I won't repeat the details here.

9 Mobile Devices: iOS, Android etc.

Modern mobile browsers can do almost everything their desktop counterparts do, so developing with Sasha Maps for mobile devices is not very different from developing for the desktop. There are a few things to note, however.

- By default, most mobile browsers render your webpage at a logical size much larger than the device screen, and then scale it down. When displaying maps, however, you usually want the map to be displayed at its native resolution. For this, you need to add the following to the head section of your webpage:

  ```
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
                 user-scalable=no">
  ```

  This makes the browser render your page at the actual size of the screen.

- In order for map panning to be fast and smooth on the mobile device, Sasha Maps disables (by default) the creation and loading of any new tiles while the map is being dragged. You can control this behaviour via the `MapWidget.setLoadTilesDuringTemporaryChanges` method.

- To prevent too many tile images from being loaded (mobile traffic is still expensive sometimes), and to speed up map panning/zooming, it's recommended to tweak `MapWidget`'s `tileCoverSize` property. You will usually want a smaller value on mobile devices than on the desktop.

- GwtUtils' `ImageButton` works as-is on touch devices. Instead of responding to mouse clicks, it responds to taps, but fires a regular click event.

- `WidgetOverlay` lets you easily make the overlay draggable via the `makeOffsetDraggableByTouch` and `makeAnchorDraggableByTouch` methods which take a "drag handle" parameter. The drag handle is a `HasAllTouchHandlers` interface, which is implemented by many of GWT's widgets, including `Image`. Don't forget to prevent propagation and default action for your handle, which you can easily do by wrapping it in GwtUtils' `EventCancellingPanel` with an `Event.TOUCHEVENTS` mask.

- Some devices, such as retina-screen iPhone/iPad or the Galaxy Nexus, report less logical pixels per inch than their screens actually have. To fully utilize the real pixel density of such a screen, you need to scale down the tile images by the ratio of device pixels per logical pixel. The `Map` class provides an easy way to do this: simply create your `Map` instance as you normally would, and then invoke `getVersionScaledToDisplay()` on it to obtain a scaled version, which you would then continue using as before. To check the ratio of device-pixels per logical-pixel yourself, see GwtUtils' `BrowserUtils.getDevicePixelRatio()` method.
  Note that scaling the tile images as shown above will increase the number of tiles needed to cover the view, and therefore the bandwidth and amount of traffic needed to display a map. Take this into consideration when deciding whether you prefer to scale the tiles or not.
  For full effect on high-resolution screens, it's best to actually re-render your tiles at a higher size/resolution before scaling them down, rather than just reusing the regular tile images.

10 Disclaimers

Sasha Maps is now reasonably mature, and production ready. It has been used in several large applications. Sasha Maps is continuously being tested on the latest versions of popular modern browsers. If you encounter any bugs, please email me at msasha@gmail.com.

Sasha Maps is inspired by Google Maps, but does not use any code from it. It is written entirely from scratch, by me, Alexander Maryanovsky.

The maps it is capable of displaying are not my property, but the property of whoever owns them. To actually display them on your website or in your application, you normally need permission from the owner.

I hope you enjoy using my library as much as I enjoyed (and continue to enjoy) writing it.

11 Resources

- Sasha Maps homepage
- Sasha Maps demo
- Sasha Maps mobile demo
- GWT (Google Web Toolkit)
- The OpenStreetMap project
- Web Map Service
- Apple's Safari Documentation