Geomajas GWT client 2.3.0

Geomajas Developers and Geosparc

Geomajas GWT client 2.3.0

by Geomajas Developers and Geosparc

Version 2.3.0
Copyright © 2011-2015 Geosparc nv

Table of Contents

1. Introduction 1
2. Configuration 2
    1. Adding Geomajas Client 2.x to your project 2
    2. Using the Geomajas Client 2.x together with the Geomajas Server 2
3. Architecture 4
    1. The central Map API 4
        1.1. Getting started 4
        1.2. The Geomajas map aka the MapPresenter 4
        1.3. Map initialization 6
        1.4. Adding the map to the GWT layout 6
        1.5. Map configuration & hints 6
        1.6. Managing the view on the map 7
        1.7. Layer composition 8
        1.8. Layer API 8
    2. Events 10
        2.1. Geomajas events versus native events 10
        2.2. EventBus concept 10
        2.3. Event overview 11
    3. User interaction 14
        3.1. MapController definition 14
        3.2. Applying your own MapController on the map 14
        3.3. Working with events on the map 15
    4. Graphics & Rendering 15
        4.1. WorldSpace vs ScreenSpace 16
        4.2. Rendering containers 16
        4.3. Drawing geometries on the map 18
    5. Client/Server communication 18
        5.1. CommandService 18
        5.2. Custom client/server communication 19
    6. Widgets/controls 19
        6.1. The default map control widgets 19
        6.2. Adding widgets on top of the map 22
4. How-to 23
    1. Adding a map to a GWT 2.0 layout 23
    2. How to catch the location of mouse events on the map? 23
    3. How to enable/disable animation on a layer? 23

List of Tables

3.1. Map Events 11
3.2. ViewPort Events 11
3.3. Layer Events 12
3.4. Feature Events 13
3.5. Other Events 14

Chapter 1. Introduction

The Geomajas Client 2.x is an Ajax client that provides a lean and mean mapping API for the web. This is a stand-alone library for client-side GIS, but it really excels when used in combination with the Geomajas Server project.

The use of the Google Web Toolkit as a base, let's the user develop his applications in Java, using his favourite Java IDE (Eclipse, IntelliJ, Netbeans, ...) and all the Java tools he is used to having around. Basically GWT is an efficient Java to Javascript compiler that only compiles and obfuscates that part of the libraries that you're actually using in your application. This means that classes that are not used will not be a part of the final build. These and many other tricks are used to optimise the Javascript that comes out at the end, making the footprint for Geomajas applications as small as can be. As a result of all this optimization, the memory consumption within the browser is also kept at a minimum, while performance is kept at maximum.

An added bonus of these optimizations is that this API is also perfectly suited in mobile environments.

Chapter 2. Configuration

In order to use the Geomajas client 2.x in your web application, there are a few steps you need to take.

1. Adding Geomajas Client 2.x to your project

First of all, you will need to add the correct dependencies. Using Maven, this would be:

```
<dependency>
    <groupId>org.geomajas</groupId>
    <artifactId>geomajas-client-gwt2-impl</artifactId>
    <version>2.3.0</version>
</dependency>
```

After the libraries have been added to your application, it's time to configure the correct GWT module. When writing applications in GWT, you need to precisely mark which packages or classes make up your client, so the GWT compiler knows where to look when compiling Java classes to Javascript.

How to set up a GWT application and declare an EntryPoint is beyond the scope of this document. You'll have to visit the GWT documentation site for more information. What you do need to know is the definition of the Geomajas GWT module definition.

In your .gwt.xml file, add the following line to include the Geomajas client:

```
<!-- Geomajas Client 2.x -->
<inherits name="org.geomajas.gwt2.GeomajasClientImpl" />
```

2. Using the Geomajas Client 2.x together with the Geomajas Server

By making use of the Geomajas server project, a lot more functionalities become available, such as server-side layer definitions with build-in security etc. Should you want to make use of that, additional configuration is required.

First of all, you'll need the following dependency:

```
<dependency>
    <groupId>org.geomajas</groupId>
    <artifactId>geomajas-client-gwt2-server-extension</artifactId>
    <version>2.3.0</version>
</dependency>
```

This dependency will pull the Geomajas server project into your project.

Next, you'll have to include the GWT module into your .gwt.xml file:

```
<!-- Geomajas Client 2.x Server Extension -->
<inherits name="org.geomajas.gwt2.GeomajasClientServerExtension" />
```

Next, you will have to configure your web.xml file to set up the Geomajas WebServices. Details on how to set up the Geomajas backend can be found in the backend documentation files.

Tip

At this moment the Geomajas server project takes on a lot of responsibilities. It is therefore recommended you make use of it. For example, most layer types (WFS, Database, Shape, ...)

are provided in the form of server-layers. Their client counterparts are on the roadmap, but not available yet.

Chapter 3. Architecture

1. The central Map API

This section describes the interfaces of the most central object definitions within the API: those that make up the map model. We start by introducing the Geomajas starting point, which is used to create a new map instance. From there on, we delve deeper into the most important map concepts, such as the layer model and viewport.

1.1. Getting started

As mentioned in the "Configuration" chapter, this client provides 2 libraries:

- The central map API and implementation
- Extended functionalities that make use of the Geomajas server project

Both have a specific starting point and both starting points provide important services, or ways to create a new map. The basic Geomajas starting point is:

`org.geomajas.gwt2.client.GeomajasImpl`

When also using the server extension, another starting point becomes available:

`org.geomajas.gwt2.client.GeomajasServerExtension`

In the next paragraph we will use this class to create a new map.

1.2. The Geomajas map aka the MapPresenter

1.2.1. MapPresenter responsibilities

The `MapPresenter` represents the central map interface and as such it determines the map's functionalities. It provides support for many of the topics that are discussed in the following sections, such as MapControllers or an EventBus.

In short, the `MapPresenter` has the following responsibilities:

- *Managing the view on the map*: This is done through the ViewPort definition.
- *Managing the layers*: This is done through the LayersModel definition.
- *Providing user interaction*: Catching native HTML events is done through `MapControllers`. The map has support for one active `MapController` for user interaction, and a set of passive map controllers that are allowed to catch native events, but may not interrupt default event bubbling.
- *Event handling*: The MapPresenter provides an EventBus through which all specific Geomajas events pass. You you can react to changes on the ViewPort, or layer composition changes, or, ... A full list of events is provided in a later section. The events covered here are Geomajas custom events, not native browser events.
- *Rendering and custom drawing*: Next to the automatic rendering of the layers, the MapPresenter also provides API for custom rendering. Custom rendering can occur through HTML, VML, SVG and Canvas.

On top of those responsibilities, a map also has a configuration, a container for widgets, options for custom drawing, etc.

Before we delve deeper into the map API, let us first show you how to create a new map.

1.2.2. Creating a new map

When creating a new map, it is important you provide it with a configuration. Now this configuration can be provided through code, or it can be provided by the Geomajas server project. Let us start with the first option. First of all, you will need to assemble a `MapConfiguration` object, as such:

```
MapConfiguration configuration = new MapConfigurationImpl();
configuration.setCrs("EPSG:4326", CrsType.DEGREES);
configuration.setMaxBounds(new Bbox(-180, -90, 360, 180));
configuration.setMinimumResolution(2.682209014892578E-6);
```

This is your most basic map configuration object. Although it is recommended to provide a list of resolutions, it is not strictly required if you make sure to provide a maximum resolution.

Tip

You must provide a zooming range with the map configuration. The recommended option is to provided a list of resolutions. The alternative is to provided a minimum resolution (=maximum zoom in). If you provide the minimum resolution, the map will automatically assume the zooming range to be between the maximum resolution set out by the maximum bounds and the provided minimum resolution.

Another important factor to note is that we do not simply provide a coordinate reference system (crs), we also say how large it's units are. This is needed for accurate resolution calculations. Currently there are 2 types of CRS supported automatically:

- *CrsType.DEGREES*: The CRS is expressed in degrees. An example is LatLon.
- *CrsType.METRIC*: The CRS is expressed in meters. An example is Mercator.

It the CRS you want to use is neither, then there is the option of manually providing the length of a single unit in meters:

```
configuration.setCrs("MyOwnCRS", 1000);
```

This would tell the map that a single unit of the CRS is a thousand meters in length.

Now that we have our map configuration, it is time to create a new map:

```
MapPresenter mapPresenter = GeomajasImpl.getInstance().createMapPresenter(confi
```

This will create a new map of 480 by 480 pixels in size.

1.2.3. Creating a new map using the Geomajas server

The Geomajas server also knows the concept of maps. Using the server project, it is possible to configure as many maps and layers as you want in XML, using Spring beans. How to configure these maps is a part of the Geomajas server project documentation and outside of the scope for this document.

The client can very easily make use of the map configurations provided by the server:

```
MapPresenter mapPresenter = GeomajasImpl.getInstance().createMapPresenter();
mapPresenter.setSize(480, 480);
GeomajasServerExtension.getInstance().initializeMap(mapPresenter, "gwt-app", "m
```

This example create a new map without a configuration. It this point, the map is useless. We than apply a size and fetch a configuration from the server. Fetching a configuration, requires you to know the name of the map (and it's containing "application"). Those are the IDs of the Spring beans in question.

1.3. Map initialization

When creating a map it might take a while before it's configuration is applied (in case you fetch it from the server). As a result Geomajas has introduced a `MapInitializationEvent`. This event is fired when the map has been initialized. When providing a configuration at the moment you create the map, this event is fired immediately. But if you initialize a map with a server-side configuration, it takes a while.

It must also be noted that use a server configuration object may automatically inject layers into your map, as this may be part of the server-side configuration. Actually, this will most likely be the case.

So if you want to play around with server-side layers or you want to change the view on the map, it's best to wait for the `MapInitializationEvent`:

```
mapPresenter.getEventBus().addMapInitializationHandler(new MapInitializationHand

    public void onMapInitialized(MapInitializationEvent event) {
        // Do something interesting ...
    }
});
```

1.4. Adding the map to the GWT layout

In order for the map to display correctly, it must have a size. You either set a fixed size, like we showed in the previous section, or you let some parent widget determine the size. In any case, the map must know how large it should be in pixels.

To this end Geomajas provides a widget to incorporate the map into the GWT 2.0 layout system, call the MapLayout:

```
org.geomajas.gwt2.client.widget.MapLayoutPanel mapLayout = new MapLayoutPanel(ma
```

Now add this mapLayout widget to any GWT layout panel, to get the map to fill up the available area.

1.5. Map configuration & hints

We briefly touched the map configuration when we were trying to create a new map. What we did not mention is that this configuration object is host to a system of `Hints` that provide all kinds of settings for the map. These hints can change the default behaviour of the map in a lot of different ways. Most hints are defined within the `MapConfiguration` itself.

The `MapConfiguration` can be retrieved as such:

```
org.geomajas.gwt2.client.map.MapConfiguration mapConfiguration = mapPresenter.ge
```

With this configuration it is possible to apply and retrieve `Hints` concerning the map. All `Hints` are defined to only accept a certain type of value through Java's generic types:

```
mapConfiguration.setHintValue(Hint<T> hint, T value);
```

By default the following MapHints are defined:

- *MapConfiguration.ANIMATION_TIME*: Hint used to determine how long the animations should take during navigation (zooming). The value should be expressed in milliseconds.
- *MapConfiguration.FADE_IN_TIME*: Hint used to determine how long fading in of resolution or tiles should take while rendering the map.

- *MapConfiguration.ANIMATION_CANCEL_SUPPORT*: Hint that determines whether or not the ViewPort will cancel the currently running animation in favor of a new animation. If this value is false, the new animation will keep running, and the new animation is discarded. If this value is true, the current animation is discontinued, and the new one is started.
- *MapConfiguration.DPI*: Hint used to determine the DPI on the map.
- .....

This is an example of how one would change the default animation time to last 1 second:

```
mapConfiguration.setMapHintValue(MapConfiguration.ANIMATION_TIME, 1000);
```

1.6. Managing the view on the map

1.6.1. The ViewPort

One of the most important concepts within a map is it's position and how to navigate from one place to another. Most of the time it will be the user that determines navigation through a controller on the map (mouse or touch). Sometimes though it might be necessary to have the map navigate to some pre-defined location through code.

The whole navigation and positioning concept is bundled within the ViewPort. The ViewPort can be accessed directly from the MapPresenter:

```
org.geomajas.gwt2.client.map.ViewPort viewPort = mapPresenter.getViewPort();
```

Through the ViewPort one can get the current map position:

```
org.geomajas.gwt2.client.map.View view = viewPort.getView();
org.geomajas.geometry.Coordinate position = viewPort.getPosition();
org.geomajas.geometry.Bbox bounds = viewPort.getBounds();
double resolution = viewPort.getResolution();
```

Next to acquiring current location, you can also force the map to navigate to a certain location instantly:

```
viewPort.applyPosition(new Coordinate(0,0));
viewPort.applyResolution(0.01);
viewPort.applyBounds(new Bbox(0,0,100,100));
```

1.6.2. Navigating using animations

These days a map must be able to navigate using fluid animations. This too is a part of the Geomajas API in the form of the:

```
org.geomajas.gwt2.client.animation.NavigationAnimation
```

For the most basic animation types (zooming, panning, ...) Geomajas provides a factory:

```
org.geomajas.gwt2.client.animation.NavigationAnimationFactory
```

Using this factory it is possible to create your own animations. Once created though, you still have to register them with the ViewPort.

The following example creates a new zoom in animation and applies it immediately:

```
NavigationAnimation animation = NavigationAnimationFactory.createZoomIn(mapPres
mapPresenter.getViewPort().registerAnimation(animation);
```

1.6.3. Rendering spaces

The `ViewPort` can be seen as the navigator behind the map. It manages the map's navigation (by making it zoom or pan) and sends the required events. On top of that, the `ViewPort` also has a second responsibility in providing transformations between different rendering spaces.

Visit the WorldSpace vs ScreenSpace section for more information.

1.7. Layer composition

Also part of the central map model, is a separate interface for managing all the layers of the map. As is typically the case in GIS, people work not just with one type of data, but with many different types that are all represented by "layers" in a map. These layers always have a certain order in which they are drawn, as they lie on top of each other.

The LayersModel is directly accessible from the MapPresenter:

```
org.geomajas.gwt2.client.map.layer.LayersModel layersModel = mapPresenter.getLa
```

This model has the following responsibilities:

- *Adding and removing layers*: These methods will add layers on top, or remove existing layers from the map.
- *Retrieving layers*: You can retrieve layer objects through their unique ID, or by index. It's also possible to get the total layer count.
- *Moving layers up and down*: Remember that the layers form an ordered list, so these methods will change the layer order.
- *Get the currently selected layer*: The layer API provides the possibility to select one single layer. This option can be used for specific use-cases revolving around a single layer.

Note that almost all changes in the LayersModel will trigger specific events, making it easy to follow up on changes.

1.8. Layer API

As many different types of layers exist all with their own specific set of functionalities, we have decided to reflect this diversity in the layer interface, by splitting it up in multiple 'functional' interfaces. There is still a main `org.geomajas.gwt2.client.map.layer.Layer` interface, which must always be implemented, but layer implementations can decide for themselves which of the 'functional' interfaces they support and which they don't.

1.8.1. Supporting interfaces

On top of the basic layer interface, the following extensions are available:

- *org.geomajas.gwt2.client.map.layer.FeaturesSupported*: Extension for layers that contain features. Features are the base vector-objects a layers can consist of.
- *org.geomajas.gwt2.client.map.layer.LabelsSupported*: Allows labels for a layer to be turned on or off.
- *org.geomajas.gwt2.client.map.layer.LegendUrlSupported*: Extension that can provide URLs to the legend images for this layer. A WMS layer for example will implement this to point to the GetLegendGraphic URL.

1.8.2. Server-side layer integration

As was just mentioned, multiple interfaces exist that make up a layers functionality. The server-extension provides additional layer types that make use of layers that have been defined on the server. There are 2 types:

- *org.geomajas.gwt2.client.map.layer.VectorServerLayer*: A proxy to any vector layer type that has been defined on the server. These can in turn be WFS, Database, Shapefile, ...
- *org.geomajas.gwt2.client.map.layer.RasterServerLayer*: A proxy to any raster layer type that has been defined on the server. These can in turn be WMS, TMS, Google maps, ...

These types of layer are defined on the server. They are usually added to a map configuration on the server as well. When you create a new map using such a server map configuration, these layers will already be present in the client-side map object.

1.8.3. Attributes, descriptors and features

In the previous section we briefly mentioned the Feature (`org.geomajas.gwt2.client.map.feature.Feature`) concept. Features are the individual objects that make up vector layers (layers that implement `org.geomajas.gwt2.client.map.layer.FeaturesSupported`).

A Feature itself contains the following information:

- *A unique ID*: Every feature should have a unique identifier within it's layer.
- *A map of attributes*: A layer usually has a fixed set of attributes configured. For each such attribute, the feature may have a value in it's attribute map.
- *A geometry*: Without a geometry, the feature can not be displayed on a map...

Features have an attribute mapping. For every attribute defined within a layer, the feature has a value. The list of attribute you can expect to find in features, are provided by the FeaturesSupported interface:

```
List<AttributeDescriptor> descriptors = featuresSupportedLayer.getAttributeDesc
```

Every attribute descriptor has a name and a type. Using the name, one can find the values on features:

```
Object attributeValue = feature.getAttributeValue(attributeDescriptor.getName()
```

Many types of attributes exist. As a result, the AttributeType has multiple extensions and implementations. At this time, 2 types are supported:

- *PrimitiveAttributeType*: An attribute whose value holds a primitive Java type, such as Strings, Integers, Double, Dates, ...
- *GeometryAttributeType*: An attribute type that holds a Geometry as value.

The Geomajas server project has support for more complex types of attributes such as many-to-one and one-to-many attributes. Expect these to be supported as well in the future in the Geomajas client 2.x

1.8.3.1. Feature Selection

The FeaturesSupported interface allows for feature selection:

```
FeaturesSupported fs = (FeaturesSupported) layer;
```

```
fs.selectFeature(feature);
boolean selected = fs.isFeatureSelected(feature); // returns true
fs.clearSelectedFeatures(); // Deselect all features within this layer.
selected = fs.isFeatureSelected(feature); // returns false
```

2. Events

2.1. Geomajas events versus native events

When developing GWT or Javascript applications, it is important to be aware of the difference between HTML native events and custom created events.

- *Native events*: Events that are triggered by the browser. They are typically triggered by input devices such as the mouse, the keyboard or touch screens. These events are provided in Geomajas through the `MapController`, which lets you define user interaction on the map. These are not the topic of this section. For more information on native browser events, visit the User Interaction section.
- *Custom events*: These are used for Geomajas specific events, such as the MapInitializationEvent we have covered earlier. You can catch these events through the Geomajas `MapEventBus`.

2.2. EventBus concept

Associated with the functionalities in the central map interfaces, are several events that signal changes in the model to all registered handlers. Note that the term `Handler` is used, not listener, as we try to follow the GWT naming conventions. As of GWT 2.x, the use of a central eventbus has been promoted to work together with an MVP approach.

Instead of randomly providing methods to register handlers for specific events, Geomajas follows the GWT reasoning in that it's much easier to work with a central event service: The EventBus. The idea is that all events are passed through this bus, so that the developer never needs to figure out where to register the handlers. Also, the EventBus can be a singleton service available everywhere in your code.

The Geomajas setup goes a bit further though in that it provides an EventBus for every map plus an application specific EventBus. The application specific event bus is optional, but can be an easy way to add your own events.

2.2.1. Geomajas MapEventBus

We start out be explaining the map centric event bus. For every map you create, there is one such event bus. This eventbus will provide all Geomajas specific events originating from within the map. It is not meant to be extended by adding new event types. Note that if you create multiple maps, you will also have multiple such event busses.

The next piece of code shows you how to get access to it:

```
org.geomajas.gwt2.client.map.MapEventBus mapEventBus = mapPresenter.getEventBus
```

This means that for `Handlers` that are registered on such an eventbus, only events that have originated within that map will reach it. Here is an example of how one can attach a Handler to an EventBus:

```
mapPresenter.getEventBus().addHandler(MapResizedEvent.TYPE, new MapResizedHandl

    public void onMapResized(MapResizedEvent event) {
        // The map has resized. Quick, do something meaningful!
    }
```

```
});
```

In the example above a `MapResizedHandler` was used that listens to `MapResizedEvents`. From the moment the `MapResizedHandler` has been registered, it will receive all events that indicate the map has been resized. This handler will not receive `MapResizedEvents` from other maps, only from this one.

This bus only provides Geomajas specific events. For a list, see event overview.

2.2.2. Application EventBus

Next to the map specific event bus, Geomajas also provides an eventbus singleton through the `GeomajasImpl`:

```
com.google.web.bindery.event.shared.EventBus eventBus = GeomajasImpl.getInstanc
```

This is a default GWT EventBus that is not actually used by Geomajas to provides map specific events, but is here as a singleton for application designers to add application specific events to.

2.3. Event overview

Time to go over all supported events and explain their purpose. Note that every event in this list is part of the GWT client API within Geomajas. All event and handler classes can be found in the following package: `org.geomajas.gwt.client.map.event`.

Map events:

Table 3.1. Map Events

Event	Handler	Description
MapInitializationEvent	MapInitializationHandler	Event that is fired when the map has been initialized. Only after this point will layers be available.
MapResizedEvent	MapResizedHandler	Event that is fired when the map widget has changed size.

ViewPort events:

Table 3.2. ViewPort Events

Event	Handler	Description
ViewPortChangedEvent	ViewPortChangedHandler	Event that is fired when the view on the ViewPort has been changed.
NavigationStartEvent	NavigationStartHandler	Event that is fired when the ViewPort starts an animated navigation sequence.
NavigationUpdateEvent	NavigationUpdateHandler	Event that is fired when the ViewPort reports an update in the navigation animation.
NavigationStopEvent	NavigationStopHandler	Event that is fired when an animated navigation sequence in the ViewPort has just ended

Layer events:

Table 3.3. Layer Events

Event	Handler	Description
LayerAddedEvent	MapCompositionHandler	Event that is fired when a new layer has been added to the map.
LayerRemovedEvent	MapCompositionHandler	Event that is fired when a layer has been removed from the map.
LayerSelectedEvent	LayerSelectionHandler	Event that is fired when a layer is selected. Only one layer can be selected at any time, so these events often go together with layer deselect events.
LayerDeselectedEvent	LayerSelectionHandler	Event that is fired when a layer has been deselected. Only one layer can be selected at any one time.
LayerHideEvent	LayerVisibilityHandler	Event that is fired when a layer disappears from view. This can be caused because some layer are only visible between certain resolution levels, or because the user turned a layer off. This event is often triggered by a `LayerVisibilityMarkedEvent`.
LayerShowEvent	LayerVisibilityHandler	Event that is fired when a layer becomes visible. This can be caused because some layer are only visible between certain resolution levels, or because the user turned a layer on. This event is often triggered by a `LayerVisibilityMarkedEvent`.
LayerLabelHideEvent	LayerLabeledHandler	Event that is fired when the labels of a layer have become invisible.
LayerLabelShowEvent	LayerLabeledHandler	Event that is fired when the labels of a layer have become visible.
LayerLabelMarkedEvent	LayerLabeledHandler	Event that is fired when the labels of a layer have been marked as visible or invisible. Note that when labels have been marked as invisible at a moment when they where actually visible, than you can expect a `LayerLabelHideEvent` shortly. On the other hand marking labels as visible does not necessarily mean that they will become visible. For labels to becomes visible, they must be invisible and their layer must be visible as well. Only if those requirements are met

Event	Handler	Description
		will the labels truly become visible and can you expect a `LayerLabelShowEvent` to follow this event.
LayerOrderChangedEvent	LayerOrderChangedHandler	Event that is fired when the order of a layer is changed within the `LayersModel`. This event contains indices pointing to the original index and the target index for the layer. Of course, changing the index of a single layer, also changes the indices of other layers.
LayerRefreshedEvent	LayerRefreshedHandler	Event that reports a layer has been refreshed. This means it's rendering is completely cleared and redrawn.
LayerStyleChangedEvent	LayerStyleChangedHandler	Event that reports changes in layer style.
LayerVisibilityMarkedEvent	LayerVisibilityHandler	Called when a layer has been marked as visible or invisible. When a layer has been marked as invisible, expect a `LayerHideEvent` very soon. But, when a layer has been marked as visible, that does not necessarily mean it will become visible. There are more requirements that have to be met in order for a layer to become visible: the map's resolution must be between the minimum and maximum allowed resolutions for the layer. If that requirement has been met as well, expect a `LayerShowEvent` shortly.

Feature events:

Table 3.4. Feature Events

Event	Handler	Description
FeatureSelectedEvent	FeatureSelectionHandler	Event that is fired when a feature has been selected.
FeatureDeselectedEvent	FeatureSelectionHandler	Event that is fired when a selected feature has been deselected again.

Other events:

Table 3.5. Other Events

Event	Handler	Description
TileLevelRenderedEvent	TileLevelRenderedHandler	Event that is fired when a resolution level has been rendered. This is used by resolution-based layer renderers, and it is up to them to determine when that is.

3. User interaction

This section will handle the basics of interacting with the map, by listening and responding to native browser events (mouse, touch, keyboard) generated from the map. The notion of native events versus custom Geomajas events was mentioned earlier in the "Events" section.

3.1. MapController definition

The basic definition for map interaction is called the MapController, which is the combination of a set of mouse and touch handlers, with some added utility methods. At least the following handlers must be implemented:

- MouseDownHandler
- MouseUpHandler
- MouseMoveHandler
- MouseOutHandler
- MouseOverHandler
- MouseWheelHandler
- DoubleClickHandler
- TouchStartHandler
- TouchEndHandler
- TouchMoveHandler
- TouchCancelHandler
- GestureStartHandler
- GestureEndHandler
- GestureChangeHandler

On top of handling mouse and touch events, the MapController definition also provides methods that are executed when a MapController becomes active on the map, or when it is deactivated.

3.2. Applying your own MapController on the map

The MapController definition can be used in 2 different ways: as a manipulative event controller, or as a passive listener. The main difference is that the listener is not allowed to manipulate the events, while the controller is free to do as it chooses. A controller could for example stop event propagation,

something a listener is not allowed to do. As a result only one controller is allowed on the map, while multiple listeners are allowed.

We have used the terms 'controller' and 'listener', but in reality both are defined by the same interface: `org.geomajas.gwt2.client.controller.MapController`. It's a difference in semantics.

The following code sample shows how to apply both on the map:

```
// Applying a new MapController on the map:
mapPresenter.setMapController(new MyCustomController());

// Adding an additional listener:
mapPresenter.addMapListener(new MyCustomController());
```

A typical example of an active controller is the NavigationController, which determines map navigation through mouse handling.

A typical example of a passive listener could be a widget that reads in the location of the mouse on the map, and prints out the X,Y coordinates. Such a MapController implementation does not interfere with the normal event flow or the main MapController.

By default, a NavigationController is active on the map. Should you apply another MapController, you will loose your default navigation abilities (unless you implement them in your MapController as well). At any time you can reinstall the default NavigationController like this:

```
mapPresenter.setMapController(null);
```

Tip

When writing your own MapControllers it is recommended to always start from the `AbstractMapController`, or even from the `NavigationController`.

3.3. Working with events on the map

Often, MapControllers need to interpret the events in some way to determine their course of action. Let us take an example where a MapController wants to read the mouse position on the map in order to print it out on the GUI. In order to do this, you will need to know the mouse position at the time of each MouseMoveEvent on the map.

For this case, the MapController also extends the `org.geomajas.gwt2.client.controller.MapEventParser` interface. This interface provides methods for extracting useful information from events:

```
// Get the event location in map CRS:
Coordinate worldPosition = mapController.getLocation(event, RenderSpace.WORLD);
```

This method extracts the location of the event, expressed in one of the rendering spaces (more in this later).

```
// Get the DOM elements that was the target of the event:
Element target = mapController.getTarget(event);
```

This method extracts the target DOM element of the event.

4. Graphics & Rendering

This section will handle all rendering related topics, explaining the different render spaces (WorldSpace versus ScreenSpace), and how to make full advantage of them when trying to render objects on the map.

4.1. WorldSpace vs ScreenSpace

Before trying to render anything on a map, it is crucial you understand the difference between WorldSpace and ScreenSpace. Both represent render spaces wherein the user can render his objects.

- *WorldSpace*: World space describes a rendering space where all objects are expressed in the coordinate reference system of the map. As a result, all objects within world space move about with the view on the map.

  Let's say for example that a rectangle is rendered on a map with CRS lon-lat. The rectangle has origin (118,34) and width and height both equal to 1. Than this rectangle will cover the city of Los Angeles. No matter where the user may navigate, the rectangle will always remain above Los Angeles.

- *ScreenSpace*: Screen space describes a rendering space where all objects are expressed in pixels with the origin in the top left corner of the map. Objects rendered in screen will always occupy a fixed position on the map. They are immobile and are not affected by map navigation.

Caution

Beware that drawing a great many objects in WorldSpace can slow down an application, as their positions need to be recalculated every time the map navigates.

4.2. Rendering containers

Before actually rendering custom objects on the map, you must choose a type of container wherein to draw. This type of container will determine the the format used in HTML:

- *org.geomajas.gwt2.client.gfx.VectorContainer*: Depending on the browser used, this container will render in either SVG or VML.
- *org.geomajas.gwt2.client.gfx.CanvasContainer*: This container will make use of the HTML5 canvas construct for drawing.
- *org.geomajas.gwt2.client.gfx.TransformableWidgetContainer*: A container that is meant to display widget in WorldSpace on the map. For example, this could be used to render image markers on certain locations.

These containers are meant for custom rendering. They have nothing to do with the layer of the map. Containers will always be rendered separately and on top of all the layers. All containers are managed by the `ContainerManager`, which can be acquired through the `MapPresenter`:

```
org.geomajas.gwt2.client.map.ContainerManager containerManager = mapPresenter.g
```

4.2.1. VectorContainers & VectorObjects

For vector object rendering, the GWT client makes use of the Vaadin GwtGraphics library. This library provides all the necessary methods for standard SVG and VML rendering. The main interfaces to note are the `VectorContainer` and the `VectorObject`.

The VectorContainer is a container object as the name implies and provides methods for storing and managing VectorObjects. These VectorObjects in turn are the individual objects (such as Rectangle, Circle, Path, ...) that can be drawn on the map.

In order to acquire a VectorContainer, all one has to do is request such a container from the MapPresenter. This can be done by calling one of the following methods:

```
// Getting a VectorContainer for rendering in WorldSpace:
VectorContainer worldContainer = mapPresenter.getContainerManager().addWorldCon
```

```
// Getting a VectorContainer for rendering in ScreenSpace:
VectorContainer screenContainer = mapPresenter.getContainerManager().addScreenC
```

After acquiring such a container it is possible to add multiple VectorObjects to it.

Note

Be careful to make sure you use the correct coordinate system when adding VectorObjects to your VectorContainer. A container that was added in ScreenSpace, expects it's VectorObjects to be expressed in pixel coordinates (integer values). A container that was added in WorldSpace expexts it's VectorObjects to be expressed in world coordinates or user coordinates (the more general term used in GwtGraphics).

4.2.2. CanvasContainers

There is some experimental support for HTML5 canvas rendering. To create a new the canvas container, call the following method:

```
CanvasContainer canvasContainer = mapPresenter.getContainerManager().addWorldCa
```

The canvas rendering API is entirely different from the SVG/VML rendering APIs. Canvas is not DOM-based, but provides a generic 2D context and pen for stroking and filling primitives like paths, arcs and rectangles, drawing text and images, etc... This will sound familiar to those of you that have used the java Graphics2D API. Canvas provides full control of the pixel-by-pixel appearance of the image that you are drawing. The down-side of canvas is that the responsibility of redrawing the image whenever the state changes (and this means any state change, including simple panning or zooming) is left to the application. Canvas also lacks the concept of objects or event targeting, it is just a dumb image. This can of course be mitigated by keeping track of rendered objects yourself (after all, this is what most drawing software using Graphics2D does), but this is much more complicated than with a DOM-based model.

Our simple container implementation keeps track of a list of CanvasShape objects and will automatically repaint them whenever the map is translated or scaled. The container will react immediately when shapes are added or removed, although the repaint method can be called explicitly as well (e.g. when an object is updated) :

```
public interface CanvasContainer extends Transformable, Transparent, IsWidget {
  void addShape(CanvasShape shape);
  void addAll(List<CanvasShape> shapes);
  void removeShape(CanvasShape shape);
  void clear();
  void repaint();
  void setPixelSize(int width, int height);
}
```

The shape objects themselves have to implement a paint() method to draw themselves on the canvas (in world coordinates). An example of such a drawing method for a simple rectangle (CanvasRect) is shown here:

```
  @Override
  public void paint(Canvas canvas, Matrix matrix) {
    canvas.getContext2d().save();
    canvas.getContext2d().setFillStyle(fillStyle);
    canvas.getContext2d().fillRect(box.getX(), box.getY(), box.getWidth(), box.g
    canvas.getContext2d().setStrokeStyle(strokeStyle);
    canvas.getContext2d().setLineWidth(strokeWidthPixels / matrix.getXx());
    canvas.getContext2d().strokeRect(box.getX(), box.getY(), box.getWidth(), bo
    canvas.getContext2d().restore();
  }
```

Notice that the previous context state is saved at the start and restored at the end. This ensures that we don't propagate context changes between successive shapes. The body of the code is simply drawing a rectangle in the required fill and stroke style. We have to divide the original line width in pixels by the resolution factor (assuming uniform scaling) because it will be multiplied afterwards by the container as part of the world-to-screen transformation.

The container implementation uses a simple form of double buffering to make it fast and can render hunderds of thousands of rectangles at once with an acceptable performance. There is presently no support for handling events on a per-shape basis, though, so this is mostly useful for background rendering.

4.3. Drawing geometries on the map

Often one needs to draw geometries on the map. Say we have a Feature who's geometry we want to render in a specific style. Since a Feature is a part of a `FeaturesSupported` layer, it's geometry is expressed in the map CRS. Hence we will want to render it's geometry in WorldSpace. So we start by creating a `VectorContainer`:

```
// Getting a VectorContainer for rendering in WorldSpace:
VectorContainer worldContainer = mapPresenter.getContainerManager().addWorldCon
```

Next we want to add the geometry as a Path to the `VectorContainer`. First we need to transform the geometry into a VectorObject:

```
// Transform the geometry into a VectorObject that can be rendered in a contain
VectorObject vectorObject = GeomajasImpl.getInstance().getGfxUtil().toShape(fea
```

Before adding the path to the `VectorContainer`, we may want to style it first:

```
GeomajasImpl.getInstance().getGfxUtil().applyFill(vectorObject, "#0066AA", 0.5)
GeomajasImpl.getInstance().getGfxUtil().applyStroke(vectorObject, "#004499", 0.
```

Now it's time to add the path to the `VectorContainer`:

```
VectorContainer worldContainer = mapPresenter.getContainerManager().addWorldCon
worldContainer.add(vectorObject);
```

5. Client/Server communication

This part of the documentation handles the communication with the Geomajas server project. Use of the Geomajas server project is optional but recommended.

Although this chapter of the documentation is about the Gwt client API, Geomajas is at it's heart a client/server based framework. The client needs the server to operate correctly. The client/server communication mechanism used is a command pattern based upon the GWT RPC services.

An example of this communication is the map that fetches it's configuration from the server when it initializes.

5.1. CommandService

Most client/server communication that Geomajas does is through a command pattern based upon the GWT RPC services. The Geomajas server knows many Commands, which are defined as Spring beans. This communication mechanism is used to fetch server-side map configurations or for rendering server-layers.

The commands available are always defined on the server, but can be called from the client. Most commands are used internally by Geomajas, but often,

backend plugins provide additional commands for the client to use. For this, the `org.geomajas.gwt2.client.service.CommandService` singleton is provided. This service can be accessed as follows:

```
CommandService commandService = GeomajasServerExtension.getInstance().getComman
```

Next you can use this service to execute a command:

```
// Prepare a command:
EmptyCommandRequest request = new EmptyCommandRequest();
GwtCommand command = new GwtCommand();
command.setCommandName("command.GetSimpleExceptionCommand"); // Maybe not the be
command.setCommandRequest(request);

// Now execute the command:
commandService.execute(command, new AbstractCommandCallback<CommandResponse>()

    @Override
    public void execute(CommandResponse response) {
        // don't do anything. An Exception will been thrown at server-side
    }
});
```

This example is taken from the showcase, where a command is created that throws an exception. Perhaps not the most useful command, but it's a clear example.

Every command is defined by a request and a response object. We create a client-side GwtCommand object that refers to the backend command implementation through a string identifier, in this case "command.GetSimpleExceptionCommand". Normally this string is defined as a public static string in the request object.

5.2. Custom client/server communication

Although Geomajas uses a command pattern for it's own client/server communication, it is not limited by it. After all, Geomajas uses the GWT framework which has native support for Ajax calls (Json, XML, ...). When creating your own WebServices, you are not bound to extend Geomajas' commands. It is perfectly possible to write your own RESTful service or a custom GWT RPC service instead.

6. Widgets/controls

6.1. The default map control widgets

6.1.1. Overview of the default map controls

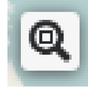
By default Geomajas will add a few control widgets to the map. These provide general navigation functionalities to the user. If you are not satisfied, you can always replace the defaults with others.

The following is a list of widgets that are available out of the box:

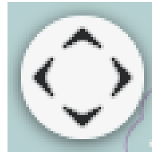
- *org.geomajas.gwt2.client.widget.control.scalebar.Scalebar*: The scalebar shown in the bottom left of the map.

- *org.geomajas.gwt2.client.widget.control.zoom.ZoomControl*: The default zoom control widget, showing a zoom in and a zoom out button.

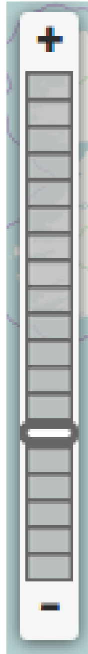
- *org.geomajas.gwt2.client.widget.control.zoomtorect.ZoomToRectangleControl*: Button that allows the user to zoom in to a rectangle. The user needs to drag the rectangle after clicking this button.



- *org.geomajas.gwt2.client.widget.control.pan.PanControl*: A widget that provides 4 buttons to pan the map in 4 directions (north, east, south, west).



- *org.geomajas.gwt2.client.widget.control.zoom.ZoomStepControl*: A widget that provides a zoom step for every resolution available in the map configuration. This way a user can zoom directly to a certain zoom level should he choose to.



By default a Geomajas map will contain the following widgets: `ScaleBar`, `ZoomControl`, `ZoomToRectangleControl`.

6.1.2. Using alternative map controls

Let's say you want to create a new map, and make use of the PanControl and ZoomStepControl instead of the default ZoomControl. For this reason, Geomajas has defined the:

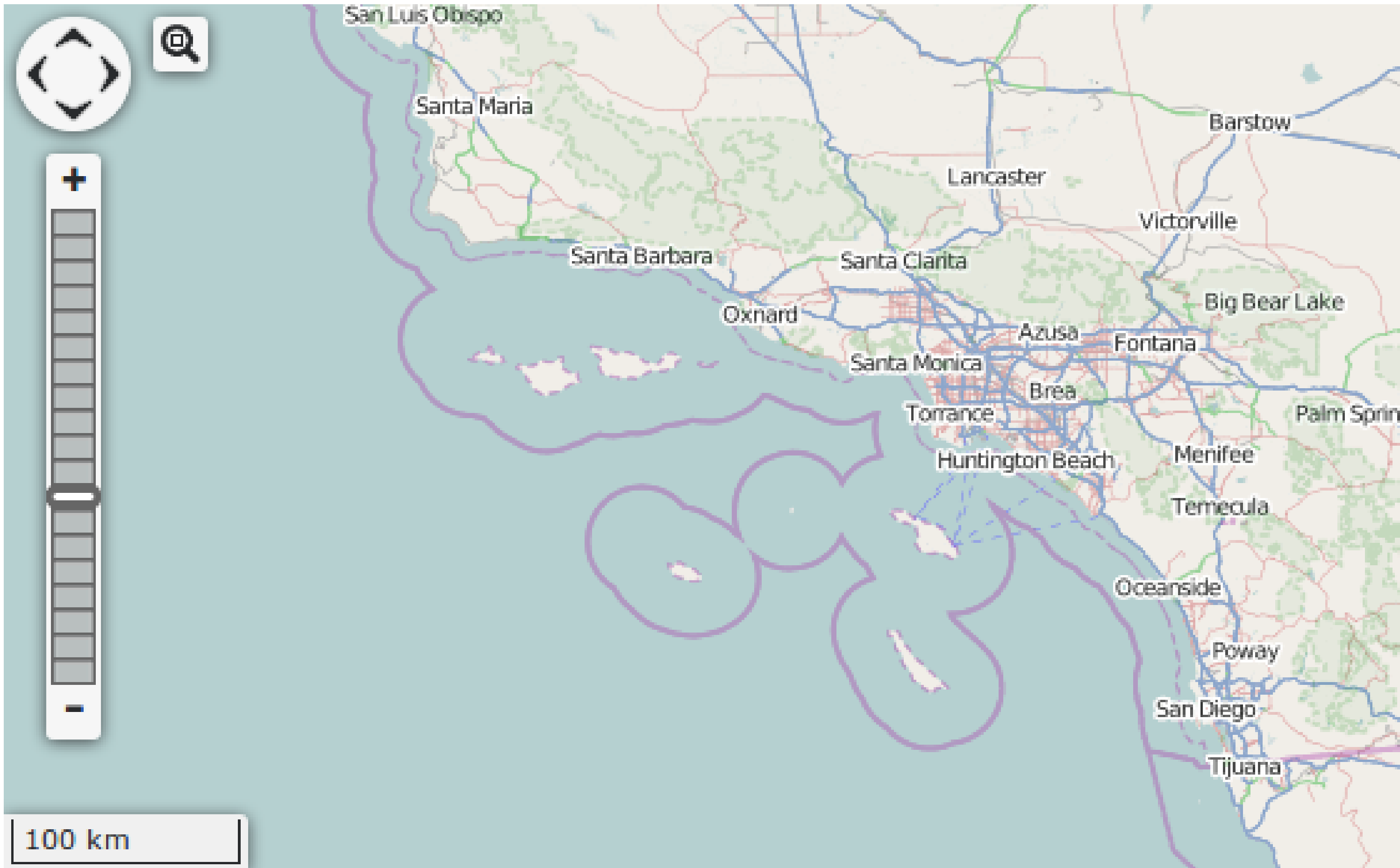
```
org.geomajas.gwt2.client.widget.DefaultMapWidget
```

It is an enumeration that lists the map widgets Geomajas provides by default (the list in the previous section basically). It is possible to use this enumeration to list the types of widget you want on the map at startup. The following piece of code shows you how to do just that:

```
GeomajasServerExtension.getInstance().initializeMap(mapPresenter, "gwt-app", "ma
        new DefaultMapWidget[] { DefaultMapWidget.ZOOM_TO_RECTANGLE_CONTROL, De
```

```
DefaultMapWidget.ZOOM_STEP_CONTROL, DefaultMapWidget.SCALEBAR }
```

This piece of code would result in a map that looks like this:



6.1.3. Styling the map control widgets

One of the strong points of GWT are it's resource bundles. These are bits of content (js, html, css) that are created as part of the GWT compilation process and injected at runtime as they are needed. More information in GWT resource bundles can be found in the official GWT documentation.

Every widget in Geomajas has it's own resource bundle for which Geomajas provides a default style. Most widgets will allow a resource bundle to be provided in it's constructor. For the map control widgets though, there is an easier way to override the default style: by defining a `org.geomajas.gwt2.client.widget.GeomajasImplClientBundleFactory`.

This factory is responsible for providing resource bundles for the widgets. This factory is created in the GeomajasImpl class through deferred binding, and so it is possible to override the implementation to use. Concretely this can be done by providing the new implementation in your .gwt.xml file:

```
<replace-with class="com.mycompany.MyCustomGeomajasImplClientBundleFactory">
    <when-type-is class="org.geomajas.gwt2.client.widget.GeomajasImplClientBund
</replace-with>
```

Of course this explanation assumes you are using GWT to define the style in your application.

However there are situations where a separate CSS file is provided that should provide the style for the Geomajas widgets. Should that be the case, you find that the resource bundles will override the CSS that you have provided in your CSS file. For this case, there exists a GeomajasImplClientBundleFactory that uses empty CSS classes. This way no CSS will be injected at runtime, and you can

provide the CSS yourself. This `GeomajasImplClientBundleFactory` is called the `org.geomajas.gwt2.client.widget.nostyle.GeomajasImplClientBundleFactoryNoStyle`, and can be loaded by adding the following line to your .gwt.xml file:

```
<inherits name="org.geomajas.gwt2.GeomajasClientImplNoStyle" />
```

6.2. Adding widgets on top of the map

As you know, Geomajas will add a few widgets to the map by default. Let's say you want to use your own widgets, or you want the default widgets to appear at a different location. Know that any widget that Geomajas adds to the map, is still available through the maps widget pane. This is a panel onto which one can add any widget. This panel can be retrieved as such:

```
HasWidgets panel = mapPresenter.getWidgetPane();
```

The panel is by default implemented through an AbsolutePanel. This is a GWT layout panel wherein all widgets are positioned absolutely (actually using the CSS position:absolute construct). So when adding widget to it, make sure you set their position to absolute, and give them top, left, bottom or right values (CSS).

So let us go through an example wherein we want to add the PanControl separately, and add it to the upper right corner of the map:

```
mapPresenter.getEventBus().addMapInitializationHandler(new MapInitializationHan

    public void onMapInitialized(MapInitializationEvent event) {
        // Create the PanControl:
        PanControl panControl = new PanControl(mapPresenter);

        // Make the PanControl stick to the upper-right corner, 5 pixels from th
        panControl.getElement().getStyle().clearLeft(); // Just to be sure...
        panControl.getElement().getStyle().setTop(5, Unit.PX);
        panControl.getElement().getStyle().setRight(5, Unit.PX);

        // Now add the PanControl to the map:
        mapPresenter.getWidgetPane().add(panControl);
    }
});
```

Note we have waited for the MapInitializationEvent to add widgets to the map. This is not strictly necessary, but safer. Perhaps some of the widgets you use may need the map to be fully initialized

Chapter 4. How-to

1. Adding a map to a GWT 2.0 layout

A special MapLayout class is provided for adding a map to a GWT 2.0 layout class (class that have Layout in their name and follow a sizing hierarchy). No special resize handling is needed for this case:

```
MapLayoutPanel mapLayout = new MapLayoutPanel();
mapLayout.setPresenter(mapPresenter);
RootLayoutPanel.get().add(layout); // fills the complete browser view
```

2. How to catch the location of mouse events on the map?

An often asked for use-case is to display the location (in World space) of the mouse when it hovers over the map. This is done be adding a listeners to the map that reacts to the onMouseMove events:

```
final Label label = new Label();
mapPresenter.addMapListener(new AbstractMapController() {

    public void onMouseMove(MouseMoveEvent event) {
        Coordinate location = getLocation(event, RenderSpace.WORLD);
        label.setText("Location: " + location.toString());
    };
});
```

Note that we started from an `AbstractMapController` to create a new `MapController` implementation. This is by far the easiest way to create new controllers. From there we used the `getLocation` method to actually acquire the correct location in World space (map CRS).

3. How to enable/disable animation on a layer?

By default only one layer will be animated when navigating on the map: the bottom layer. The other layers will dissappear while navigating and reappear when navigation has finished. Sometimes one may wish to animate more than one layer during navigation (or perhaps all layers). This is actually pretty straightforward. Since it's the renderer thats responsible, it is there we can change this setting:

```
Layer layer = mapPresenter.getLayersModel().getLayer(1);
mapPresenter.getLayersModelRenderer().setAnimated(layer, true);
```