# Geomajas pureGWT face

**Geomajas Developers and Geosparc**

# Geomajas pureGWT face

by Geomajas Developers and Geosparc

1.0.0-M6

# Table of Contents

# List of Tables

# Chapter 1. Introduction

The PureGWT face is an AJAX client for the Geomajas Server, based upon the Google Web Toolkit (GWT). The goal of this client is to provide a lean and mean mapping API for the web. This library does not focus on providing GIS oriented widgets, but instead focuses on a central map concept.

The usage of the Google Web Toolkit as a base, let's the user develop his Ajax application 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 PureGWT face in your web application, there are a few steps you need to take. These include adding the correct libraries to your project, configuring the web.xml to set up the Geomajas Communication Service, and setting up the GWT module configuration.

## 1. Dependencies

First things first. Before you can boot up a web application using Geomajas, you need to include the correct jar files. Using maven these would be:

```
<dependency>
    <groupId>org.geomajas</groupId>
    <artifactId>geomajas-puregwt-client-impl</artifactId>
    <version>1.0.0-M6</version>
</dependency>
<dependency>
    <groupId>org.geomajas</groupId>
    <artifactId>geomajas-puregwt-client-impl</artifactId>
    <version>1.0.0-M6</version>
    <classifier>sources</classifier>
</dependency>
```

## 2. GWT Modules

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 compiler knows where to look. You may have noticed in the previous section that we also added the sources jar as a dependency. This is also necessary for the GWT compiler to compile the original 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 PureGWT module definition.

In your .gwt.xml file, add the following line to include the PureGWT face:

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

<!-- Default CSS theme -->
<inherits name="org.geomajas.puregwt.theme.GeomajasDefaultTheme" />
```

## 3. web.xml

This client expects the Geomajas server-side libraries to be present in the backend, along with a correctly configured map. Details on how to set up the Geomajas backend can be found in the backend documentation files.

# Chapter 3. Architecture

# 1. The central Map API

This section describes the the interfaces of the most central object definitions within the API: those that make up the map model. We start by introducing you to the Gin injection framework, 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. GIN: GWT injection

Before we start describing the API, we need to introduce the GIN injection framework. GIN is based upon Google's Guice framework, but written specifically for GWT. By using GIN, Geomajas has a clean separation between interfaces and implementations. It also means that any default implementation that Geomajas provides can be replace by your own implementations.

### 1.1.1. GeomajasGinModule and GeomajasGinjector

The configuration that ties implementations to their interfaces is defined in the `org.geomajas.puregwt.client.GeomajasGinModule`. Next to this configuration, Geomajas also provides the `org.geomajas.puregwt.client.GeomajasGinjector`. The GeomajasGinjector is a service that provides access to singleton services and a way to create Geomajas maps. The GeomajasGinjector is also tied to the GeomajasGinModule. That is how it knows which interface implementations to use.

The GeomajasGinjector is also the only way to correctly create a new Geomajas map:

```
public static final GeomajasGinjector GEOMAJASINJECTOR = GWT.create(GeomajasGin
```

Note that the GeomajasInjector itself is a singleton service, so you only need to create it once in your application. It is therefore recommended to create it in your GWT module's EntryPoint. In the code example above, we have created it as a public static final.

### 1.1.2. Overriding the default GIN implementations

One of the most important aspects of an injection framework is that one can provide alternative implementations to be used for the known interface, effectively overriding the default behaviour.

To have your application use your own implementations instead of the Geomajas defaults, you need to define your own GIN module. It is best to just copy the GeomajasGinModule and only change the implementations you need to have changed. It is critical not to leave any of the required interfaces out. After you have your own GIN module, you will need to extend the GeomajasGinjector and have it point to your own Gin module:

```
@GinModules(MyCustomGinModule.class)
public interface MyCustomGinjector extends GeomajasGinjector {
}
```

Then in your application, make sure to use your own injector:

```
MyCustomGinjector injector = GWT.create(MyCustomGinjector.class);
```

## 1.2. The Geomajas map: MapPresenter

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.

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 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.
- *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.

Before we continue deeper into the map API, let us first show you how to create a new map. The only way to correctly create a map, is to have the GIN injection framework do it for you. We expect you have already created the necessary injector in your application (see `GIN`).

```
org.geomajas.puregwt.client.map.MapPresenter map = GEOMAJASINJECTOR.getMapPrese
```

# 1.3. Map initialization

The first thing you'll want to do after you have created your map, is to initialize it by loading a map configuration, and setting a size. It is important to realize the Geomajas map configuration is stored in XML format on the server. So when you initialize the map on the client, it will fetch it's configuration from the server. This of course takes a bit of time. This means that the map in only initialized when it's configuration has been received from the server.

So, here is an example of how to initialize the map:

```
map.initialize("application-id", "map-id");
map.setSize(640, 480);
```

The 2 parameters refer to an application and map definition as defined in the backend Spring configuration. Next we have set the size for the map.

At this point the map is requesting a configuration from the server. It does not yet know which layers will be present, what it's initial view on the map will be, what it's CRS is, etc. Often you need to know when the map has been initialized because, for example, you need the layer objects for some functionality. For this occasion, there is the MapInitializationEvent:

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

    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.puregwt.client.widget.MapLayoutPanel mapLayout = new MapLayoutPane
```

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

# 1.5. Map configuration & MapHints

Of course the MapPresenter has a configuration. The biggest part of a map configuration comes from the backend. Usually the first task for a newly created map is to fetch such a configuration from the backend. But the client-side map configuration has more to it than just this back-end counterpart: it can also be used to get and set MapHints or adjust generic map options, such as animated zooming.

The MapConfiguration can be retrieved as such:

```
org.geomajas.puregwt.client.map.MapConfiguration mapConfiguration = mapPresente
```

The configuration has the option to enable or disable animated navigation for each layer individually:

```
mapConfiguration.setAnimated(myLayer, false);
```

Next this it's direct getters and setters, the the configuration also has the ability to store and use MapHints. These MapHints are options used within the map implementation. All MapHints are defined to only accept a certain type of value through Java's generic types:

```
mapConfiguration.setMapHintValue(MapHint<T> hint, T value);
```

By default the following MapHints are defined:

- *MapConfiguration.ANIMATION_ENABLED*: This setting completely enables or disabled animated navigation on the map.
- *MapConfiguration.ANIMATION_TIME*: This setting determines how long animated navigation should last (expressed in milliseconds).

This is an example of how one would disable animated navigation:

```
mapConfiguration.setMapHintValue(MapConfiguration.ANIMATION_ENABLED, false);
```

# 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.puregwt.client.map.ViewPort viewPort = mapPresenter.getViewPort();
```

Through the ViewPort one can get the current map position:

```
org.geomajas.geometry.Coordinate position = viewPort.getPosition();
org.geomajas.geometry.Bbox bounds = viewPort.getBounds();
double scale = viewPort.getScale();
String crs = viewPort.getCrs();
```

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

```
viewPort.applyPosition(new Coordinate(0,0));
viewPort.applyScale(0.01);
```

```
viewPort.applyBounds(new Bbox(0,0,100,100));
```

### 1.6.2. 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.puregwt.client.map.layer.LayersModel layersModel = mapPresenter.ge
```

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.puregwt.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. Client layers and server layers

As was just mentioned, multiple interfaces exists that make up a layers functionality. That said, there are 2 different layer definitions though:

- *org.geomajas.puregwt.client.map.layer.Layer*: This is the base layer definition. This definition provides only the most basic layer functionality, such as a unique ID, a readable title, the possibility to select it and mark it as visible.
- *org.geomajas.puregwt.client.map.layer.ServerLayer*: Extension of the layer interface to indicate the layer has actually been defined on the server in the Geomajas map configuration.

### 1.8.2. Supporting interfaces

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

- *org.geomajas.puregwt.client.map.layer.FeaturesSupported*: Extension for layers that contain features. Features are the base vector-objects a layers can consist of. This interface allows filters to be set (CQL), and has support for feature selection management.
- *org.geomajas.puregwt.client.map.layer.LabelsSupported*: Allows labels for a layer to be turned on or off.
- *org.geomajas.puregwt.client.map.layer.OpacitySupported*: Allows one to determine a layers opacity. The opacity determines the transparency on the map.
- *org.geomajas.puregwt.client.map.layer.HasLayerRenderer*: Allows layer implementations to specify their own renderers.

## 1.8.3. Features

In the previous section we briefly mentioned the Feature (`org.geomajas.puregwt.client.map.feature.Feature`) concept. Features are the individual objects that make up vector layers. Examples of vector layers are WFS (Web Feature Service) layers or Shapefile layers. Geomajas represents such layers for example through the FeaturesSupported interface.

A Feature itself contains the following information:

- *A unique ID*: Every feature should have a unique identifier within it's layers.
- *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...

### 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
```

### 1.8.3.2. Searching features

Vector layers that have been defined on the backend will always implement the FeaturesSupported interface. Through the FeatureService, it is possible to search for features by location or through CQL filters.

```
FeatureService featureService = mapPresenter.getFeatureService();

// Get a FeaturesSupported layer:
Layer layer = mapPresenter.getLayersModel().getLayer("someVectorLayer");
final FeaturesSupported fs = (FeaturesSupported) layer;

// Get the map bounds as a polygon:
Geometry mapBounds = GeometryService.toPolygon(mapPresenter.getViewPort().getBo

// Now search:
featureService.search(fs, mapBounds, 0, new FeatureMapFunction() {

    public void execute(Map<FeaturesSupported, List<Feature>> featureMap) {
        // Now do something with the features ....
        List<Feature> features = featureMap.get(fs);
```

```
    }
});
```

# 2. Events

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. The PureGWT face has incorporated this train of thought, and provides a map-centric `EventBus`. In other words, every `MapPresenter` governs it's own `EventBus`.

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.

## 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.
- 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

Instead of randomly providing methods to register handlers (a handler is the GWT version of a Listener) 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, it is not meant to add extra event types to it. 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:

```
MapEventBus mapEventBus = mapPresenter.getEventBus();
```

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

### 2.2.2. GIN EventBus

Next to the map specific event bus, Geomajas also provides an EventBus singleton through the GIN injection framework:

```
EventBus eventBus = GEOMAJASINJECTOR.getEventBus();
```

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 PureGWT API within Geomajas. All event and handler classes can be found in the following package: `org.geomajas.puregwt.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 so that both scaling and translation have occurred. |
| ViewPortScaledEvent | ViewPortChangedHandler | Event that is fired when the map zooms in or out while keeping the same center. |
| ViewPortTranslatedEvent | ViewPortChangedHandler | Event that is fired when the map is being translated, keeping the same scale level. |
| ViewPortChangingEvent | ViewPortChangingHandler | Intermediate event that is fired during the animation/pinching phase when both scaling and translating have occurred |
| ViewPortScalingEvent | ViewPortChangingHandler | Intermediate event that is fired during the animation/pinching phase when the map zooms in or out while keeping the same center. |

| Event | Handler | Description |
|---|---|---|
| ViewPortTranslatingEvent | ViewPortChangingHandler | Intermediate event that is fired when the map is being dragged |

**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 scale 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 scale 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 |

| Event | Handler | Description |
|---|---|---|
| | | 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 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 scale must be between the minimum and maximum allowed scales 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 |
|---|---|---|
| ScaleLevelRenderedEvent | ScaleLevelRenderedHandler | Event that is fired when a scale level has been rendered. This is used by scale-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.

### Note

If you want fine tuned control and attach custom mouse event handlers to custom objects on the map, have a look at the Graphics & Rendering section.

This section here handles the interface that help shield you from such fine tuned but cumbersome mouse event handling.

## 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.puregwt.client.controller.MapController`.

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 controller.

## 3.3. Working with events

Often, MapControllers need to interpret the events in some way to determine their course of action. Let us take the above example again of a MapController that wants to read the mouse position on the map in order to print it out on the GUI. For this case, the MapController also extends the `org.geomajas.gwt.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.

```
// 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.puregwt.client.gfx.VectorContainer*: Depending on the browser used, this container will render in either SVG or VML.
- *org.geomajas.puregwt.client.gfx.CanvasContainer*: This container will make use of the HTML5 canvas construct for drawing.

## 4.2.1. VectorContainers & VectorObjects

For vector object rendering, the PureGWT face 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 with the MapPresenter. This can be done by calling one of the following methods:

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

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

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.addWorldCanvas();
```

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.
    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 scale 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. As 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.addWorldContainer();
```

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

```
// Get the graphics utility from the GIN injector:
GfxUtil gfxUtil = GEOMAJASINJECTOR.getGfxUtil();

// Now transform the geometry into a Shape object:
VectorObject vectorObject = gfxUtil.toShape(feature.getGeometry());
```

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

```
gfxUtil.applyFill(vectorObject, "#0066AA", 0.5);
gfxUtil.applyStroke(vectorObject, "#004499", 0.9, 2, null);
```

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

```
worldContainer.add(vectorObject);
```

# 5. Client/Server communication

Although this chapter of the documentation is about the PureGwt 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

The commands available are always defined on the backend, 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, a CommandService singleton is provided. This service can be accessed through the Gin injection framework:

```
CommandService commandService = GEOMAJASINJECTOR.getCommandService();
```

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");
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

Next to the map, Geomajas provides additional widgets to be placed on top of the map.

## 6.1. Adding widgets on top of the map

By default Geomajas will add a few widgets on top of the map that provide navigation buttons. Of course it is possible to replace any such widget with your own implementation. Actually it is perfectly possible to add any widget you want on top of the map.

The map has a specific panel wherein such widgets can be added:

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

An AbsolutePanel is a GWT layout panel wherein all widgets are positioned absolutely (actually using the CSS position:absolute construct).

# Chapter 4. How-to

## 1. Adding a map to a classic GWT layout

The MapPresenter class not only acts as the Presenter in the MVP (Model-View-Presenter) model but also gives access to the View by implementing the IsWidget interface. As a widget, it can be added to any of the GWT layout classes with some precautions, however:

- The parent widget should implement ProvidesResize
- The size of the map has to be set explicitly

```
ResizeLayoutPanel panel = new ResizeLayoutPanel();
mapPresenter.setSize(100,100);
panel.setWidget(mapPresenter);
```

If the panel is resizable, a custom resize handler should be registered to set the size whenever a resizing event occurs.

## 2. 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
```

## 3. 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).