Geomajas rasterizing plug-in guide

Geomajas Developers and Geosparc

Geomajas rasterizing plug-in guide

by Geomajas Developers and Geosparc

1.18.0

Copyright © 2010-2012 Geosparc nv

Table of Contents

1. Introduction 1
   1. Pipeline overview 1
   2. The image and rendering services 2
   3. Rasterizing info classes 4
      3.1. MapRasterizingInfo 4
      3.2. RasterLayerRasterizingInfo 5
      3.3. VectorLayerRasterizingInfo 5
      3.4. ClientGeometryLayerInfo 5
   4. RenderingService and LayerFactoryService 6
2. Configuration 9
   1. Dependencies 9
   2. Pipeline configuration 10
   3. Rasterizing style configuration - SLD (Styled Layer Descriptor) 10
   4. Commands 11
   5. Actions and toolbar configuration 13
3. Controllers 14
   1. RasterizingController 14
   2. TmsController 14
4. How-to 16

List of Figures

1.1. Vector tile pipeline for rasterizing 1
1.2. URL handling for rasterization 2
1.3. Image service 3

List of Tables

1.1. MapRasterizingInfo attributes 4
1.2. RasterLayerRasterizingInfo 5
1.3. VectorLayerRasterizingInfo 5
1.4. ClientGeometryLayerInfo 6
2.1. RasterizeMapCommand 12

List of Examples

1.1. Image service interface definition 3
1.2. Rendering service interface definition 6
1.3. Layer factory interface definition 7
2.1. Applying the rasterized pipeline for getVectorTile() on myLayer 10
2.2. SLD style configuration 11
2.3. SLD style sample 11
2.4. Image URL service definition 12
2.5. Configuration of rasterizing actions 13

Chapter 1. Introduction

The rasterizing plug-in enables the conversion of vector data (coordinate-based geometry definitions) to raster data (images). It allows to extend the vector layer rendering pipeline by introducing an extra rasterizing step. The vector tile response will thereby contain an URL content-type that allows the client to fetch the tile as a normal image.

From a visualization view point, rasterizing tiles are quite comparable to vector tiles for many use cases such as panning and zooming. They have significant advantages when the amount of vector data is high, either in terms of features or in the amount of coordinates for each feature. Especially for web clients, DOM-based vector rendering (SVG/VML) is quite slow and should not be used beyond a couple of thousand features (state of the art as of beginning 2011). In this case rasterizing provides a significant performance boost, especially when combined with server-side caching of the image tiles.

On the other hand, there are some use cases which are evidently difficult to treat without vectorial information at hand:

- editing of geometries.
- snapping to other layers.
- selection of features.

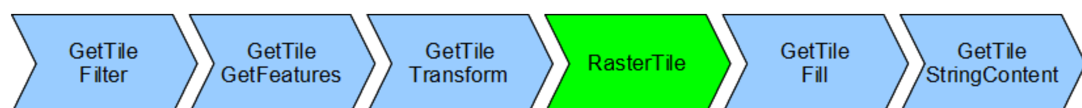
Geomajas provides the ability to load vector data on demand in such cases, thereby combining the best of both worlds.

1. Pipeline overview

The rasterizing plug-in adds a rasterizing service and some pipeline steps to the framework, as well as a controller to serve the tile images.

The normal pipeline for getting the vector tiles is enhanced with an extra rasterizing step which is indicated in green in the following figure:

Figure 1.1. Vector tile pipeline for rasterizing



The rasterizing takes place in the RasterTile step, right after the point where all the features have been collected and transformed to the screen coordinate system. It goes in front of the GetTileFill step because it has to work with the complete set of interacting features of the tile. The GetTileFill step filters the features to assure they are only drawn in one of the tiles[1]. For rasterizing this is not necessary, as drawing cannot span the tile boundary.

The RasterTile step performs the following actions:

- check the rebuild cache and make sure it has the necessary context to rebuild the rasterized tile image.
- if the rasterizing needs to be done now:

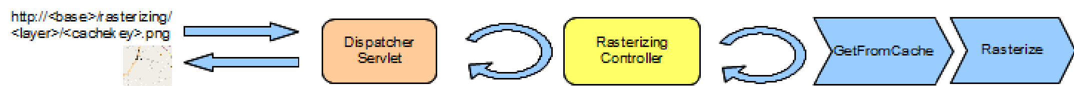
[1]This is needed for SVG and VML rendering to avoid such features being drawn twice. Features are included in the tile which contains the first point of the geometry or (if the first point is not inside the layer maximum bounds) to the super-tile.

    - build the rasterized tile image by calling the ImageService (see next chapter).
    - put the rasterized tile image in the normal cache so that it can be fetched by the controller.
- set the feature content of the tile to a unique URL that contains the cache key.

After this step, the cache must contain both the rebuild information and the rasterized image. The tile that is returned must contain the URL to fetch the tile image. Normally, the client will immediately fetch this URL from the server in a separate client-server interaction.

Our next figure shows how this URL request is handled:

Figure 1.2. URL handling for rasterization



The following series of actions take place:

- the dispatcher servlet dispatches the request to the RasterizingController, based on the URL prefix `/rasterizing/`.
- the RasterizingController invokes a separate rasterizing pipeline, which handles the following steps.
- The rasterizing cache is checked so if it contains the requested image. If successful, the pipeline ends.
- the Rasterize step tries to fetch the rebuild context from the rebuild cache. If successful, the vector tile pipeline is invoked again and the resulting image is returned. If not, an empty image is returned.

Note that you can configure when the initial rasterizing takes place. This can either be done when requesting the vector tile or when requesting the tile image. There is a trade-off, handling the rasterizing at vector tile requests optimizes throughput for the back-end, but seems slower for the client.

2. The image and rendering services

The rasterizing process has an application-level service, called `ImageService` ,which has methods for creating a legend and map image. This service internally calls a pipeline with three steps. The most important step, and the step which is responsible for the actual rendering, is the `RenderMapStep`, This step uses itself a `RenderingService`, which is primarily a wrapper of the GeoTools `StreamRenderer` class.

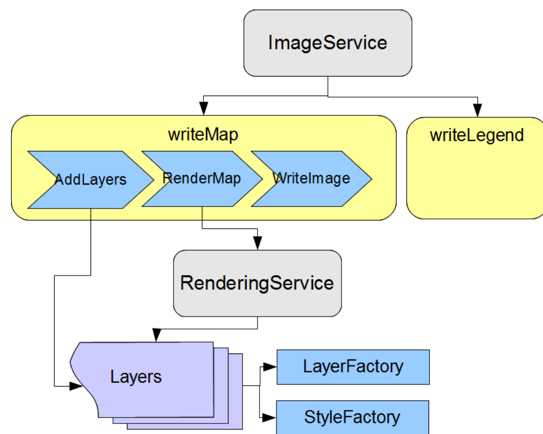
Our rasterizing is based on the GeoTools rasterizing approach. In this approach, a map context is constructed that contains all the necessary information to rasterize a map. A map context contains the following components:

- A list of layers: layers can be ordinary vector layers, raster layers or so-called direct layers, which are responsible for their own rendering.
- A view port: a view port contains an area of interest and a screen area. It provides the transformation between map an screen coordinates.

The actual rasterizing happens by passing the map context to the `RenderingService` along with a `Graphics2D` object for drawing (usually to an internal buffer).

The following picture shows the services and the interaction with pipeline steps and factories:

Figure 1.3. Image service



The image service can be used to rasterize single tiles or complete maps, depending on the composition of the map context. The `ImageService` does not directly take a map context as an argument, but expects a `ClientMapInfo` DTO object instead:

Example 1.1. Image service interface definition

```
public interface ImageService {

    /**
     * Writes a map to the specified output stream.
     *
     * @param stream output stream
     * @param clientMapInfo metadata of the map
     * @throws GeomajasException thrown when the stream could not be written
     */
    void writeMap(OutputStream stream, ClientMapInfo clientMapInfo) throws Geoma

    /**
     * Writes a map to the specified graphics object.
     *
     * @param graphics graphics object
     * @param clientMapInfo metadata of the map
     * @throws GeomajasException thrown when the stream could not be written
     * @since 1.1.0
     */
    void writeMap(Graphics2D graphics, ClientMapInfo clientMapInfo) throws Geoma

    /**
     * Writes a legend to the specified output stream.
     *
     * @param stream output stream
     * @param clientMapInfo metadata of the map
     * @throws GeomajasException thrown when the stream could not be written
     */
    void writeLegend(OutputStream stream, ClientMapInfo clientMapInfo) throws G

}
```

The `ClientMapInfo` DTO object has been extended a bit to include all the rasterizing information needed (see next chapter).

3. Rasterizing info classes

The `ClientMapInfo` DTO object has been extended to pass plugin-specific information to the rasterizing plugin backend. This was done by using the generic `ClientWidgetInfo` mechanism for adding per-map and per-layer information. Three extensions have been defined:

- `MapRasterizingInfo`: this is a `ClientWidgetInfo` extension that is applied at the map level and contains such extra information as scale and bounds, extra layers to be visualized, background transparency and legend information (in case a legend has to be rendered)

- `RasterLayerRasterizingInfo`: this is a `ClientWidgetInfo` extension that is applied at the level of a raster layer and contains client-side dynamic information like visibility and css styling (opacity)

- `VectorLayerRasterizingInfo`: this is a `ClientWidgetInfo` extension that is applied at the level of a vector layer and contains client-side dynamic information at the layer level like visibility, selection and style

The appropriate ClientWidgetInfo object should be set on the map and each of its layers using `getWidgetInfo().put(key,info)` before calling the `ImageService`. The following subsections will explain each of the extension DTO objects.

3.1. MapRasterizingInfo

The `MapRasterizingInfo` DTO object contains the metadata information that is needed to rasterize the map. The following attributes are provided:

Table 1.1. MapRasterizingInfo attributes

Name	Description
bounds	Bounds of the map region that should be visualized
scale	Scale (in pixels per map unit). In combination with the bounds this determines the dimensions of the raster image
transparent	Determines whether the resulting image should be transparent. If false, non of the layers can use transparency !
extraLayers	A list of ClientLayerInfo DTO objects that should be added on top of the configured layers of the map. This opens up the possibility to render additonal layers on top of the map that were not part of the original map configuration. It also allows to add extra objects on top of the map, such as calculated buffers or feature geometries (see ClientGeometryLayerInfo).
legendRasterizingInfo	A DTO object that contains specific information for rendering the legend (in case the `renderLegend()` method is used. It determines the width, height and font properties of the legend.
ClientGeometryLayerInfo	A subclass of the ClientLayerInfo object. This is used to add a geometry to the map that does not have a server side equivalent.

3.2. RasterLayerRasterizingInfo

The `RasterLayerRasterizingInfo` DTO object contains metadata information that is needed to rasterize a raster layer. It contains the following attributes:

Table 1.2. RasterLayerRasterizingInfo

Name	Description
showing	True if the layer should be rasterized. This property is necessary because, as the RasterLayerRasterizingInfo is added to the original layer configuration of the map, skipping of certain layers should be possible by allowing showing = false (leaving the original configuration untouched).
cssStyle	The CSS style property to be set on the raster layer images. This is primarily used to set the opacity.

3.3. VectorLayerRasterizingInfo

The `VectorLayerRasterizingInfo` DTO object contains metadata information that is needed to rasterize a vector layer. It contains the following attributes:

Table 1.3. VectorLayerRasterizingInfo

Name	Description
showing	True if the layer should be rasterized. This property is necessary because, as the VectorLayerRasterizingInfo is added to the original layer configuration of the map, skipping of certain layers should be possible by allowing showing = false (leaving the original configuration untouched).
style	The style to be applied for rasterizing the layer. This may be different from the configured style of the layer.
selectionRule	The style rule to be applied to the selected features. Selected features will be shown in a specific selectiion style defined by this rule.
selectedFeatureIds	The feature identifiers of the selected features.
filter	An extra filter to limit the features that will be rasterized.
paintLabels	Wether labels should be painted.
paintGeometries	Wether geometries should be painted.

3.4. ClientGeometryLayerInfo

The `ClientGeometryLayerInfo` DTO object contains metadata information that is needed to rasterize geometries that are not part of the map configuration and have no server-side layer equivalent. This type of layer can be added as an extra layer to the MapRasterizingInfo DTO to add such objects to the map like user-drawn geometries, calculated buffers, bounds, etcetera. It contains the following attributes:

Table 1.4. ClientGeometryLayerInfo

Name	Description
showing	True if the layer should be rasterized.
style	The style to be applied for rasterizing the geometries.
layerType	The type of the geometries. Should be one of the vector layer types
geometries	The list of geometries to be rendered. All geometries are assumed to be defined in the CRS of the map.

4. RenderingService and LayerFactoryService

As indicated in the image and rendering services section, the actual rendering of the map/legend is based on the Geotools map context and renderer concepts. The RenderingService interface has the following methods:

Example 1.2. Rendering service interface definition

```
public interface RenderingService {

    /**
     * Renders the legend for the specified map context.
     *
     * @param context map context
     * @return the image
     */
    RenderedImage paintLegend(MapContext context);

    /**
     * Renders the map context to the specified Java 2D graphics.
     *
     * @param context map context
     * @param graphics graphics object
     */
    void paintMap(MapContext context, Graphics2D graphics);

    /**
     * Renders the map context to the specified Java 2D graphics using some ext
     *
     * @param context map context
     * @param graphics graphics object
     * @param rendererHints map of renderer hints (see
     *        {@link org.geotools.renderer.lite.StreamingRenderer#setRendererHi
     * @since 1.1.0
     */
    void paintMap(MapContext context, Graphics2D graphics, Map<Object, Object> r
}
```

Both methods take a Geotools `MapContext` object as parameter. The `MapContext` object contains a list of Geotools `Layer` objects, The Geomajas layer DTO objects are internally converted to Geotools layers by using an abstract factory pattern. The abstract factory has a method to check

whether it can perform the layer creation (based on the passed `ClientLayerInfo` object) and an actual factory method:

Example 1.3. Layer factory interface definition

```
public interface LayerFactory {

    /**
     * user data to record the layer id (up to caller to decide what to do with
     */
    String USERDATA_KEY_LAYER_ID = "geomajas.rasterizing.layer"; // String

    /**
     * user data to record if layer is showing (up to caller to decide what to 
     */
    String USERDATA_KEY_SHOWING = "geomajas.rasterizing.showing"; // boolean

    /**
     * user data for the layer styles (DTOs, should eventually become unnecessa
     */
    String USERDATA_KEY_STYLE_RULES = "geomajas.rasterizing.style.rules"; // Li

    /**
     * user data for the map (up to caller to decide what to do with this info)
     */
    String USERDATA_RASTERIZING_INFO = "geomajas.rasterizing.info"; // MapRaste

    /**
     * Returns true if this factory is capable of creating layer instances for 
     *
     * @param mapContext the map context
     * @param clientLayerInfo the client layer metadata
     * @return true if we can create layer instances
     */
    boolean canCreateLayer(MapContext mapContext, ClientLayerInfo clientLayerIn

    /**
     * Creates a layer for the specified metadata.
     *
     * @param mapContext the map context
     * @param clientLayerInfo the client layer metadata
     * @return layer ready for rendering
     * @throws GeomajasException something went wrong
     */
    Layer createLayer(MapContext mapContext, ClientLayerInfo clientLayerInfo) t

    /**
     * Retrieves the userdata for the specified metadata. Especially {@link Lay
     *
     * @param mapContext the map context
     * @param clientLayerInfo the client layer metadata
     * @return the user data key values
     * @since 1.1.0
     */
    Map<String, Object> getLayerUserData(MapContext mapContext, ClientLayerInfo
}
```

The `LayerFactory` also defines a number of user data keys that allow Geotools `Layer` implementations to access Geomajas DTO objects internally, should that be necessary.

The following concrete LayerFactory implementations are available:

- `VectorLayerFactory`: accepts a `ClientVectorLayerInfo` DTO and creates a Geotools `FeatureLayer`
- `RasterLayerFactory`: accepts a `ClientRasterLayerInfo` DTO and creates a `RasterDirectLayer`, which is our own implementation of a Geotools `DirectLayer` for rendering raster layers
- `GeometryLayerFactory`:accepts a `GeometryLayerInfo` DTO and creates a `GeometryDirectLayer`, which is our own implementation of a Geotools `DirectLayer` for rendering geometries

Concrete layer factories are instantiated as singleton components in the application context. A separate `LayerFactoryService` takes care of iterating over the configured layer factories and is the entry point for creating the layers for the map context in the `AddLayersStep` of the rasterizing pipeline. The current system can be extended by creating an additional `ClientLayerInfo` class and a complimentary `LayerFactory`. The custom `LayerFactory` should create a Geotools `Layer` implementation based on `DirectLayer`.

Chapter 2. Configuration

The configuration of the rasterization involves the following elements:

- configure the vector tile pipeline to use rasterization.
- configure the rasterizing service.
- configure the Style2DFactoryService.

1. Dependencies

Make sure you include the correct version of the plug-in in your project. Use the following excerpt (with the correct version) in the dependencyManagement section of your project:

```
 <dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-rasterizing-all</artifactId>
    <version>1.0.0</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

If you are using geomajas-dep, this includes the latest released version of the rasterizing plug-in (at the time of publishing of that version). If you want to overwrite the rasterizing plug-in version, make sure to include this excerpt *before* the geomajas-dep dependency.

You can now include the actual dependency without explicit version.

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-rasterizing</artifactId>
</dependency>
```

If you want to make use of the toolbar actions for exporting map and legend images, you must add the GWT face dependency as well:

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-rasterizing-gwt</artifactId>
</dependency>
```

In this case you should also add the module inheritance to your GWT module descriptor:

```
<module>
    <inherits name="org.geomajas.plugin.rasterizing.Rasterizing"/>
</module>
```

There is also a dependency for the pure GWT face. For the pure GWT face, the toolbar actions have not yet been implemented, though. The dependency just contains a module definition that includes the DTO objects. This dependency is useful for pure GWT plugins that need to communicate with rasterizing backend, like the Simple Printing Plugin:

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-rasterizing-puregwt</artifactId>
</dependency>
```

In this case you should add the same module inheritance to your GWT module descriptor:

```
<module>
    <inherits name="org.geomajas.plugin.rasterizing.Rasterizing"/>
</module>
```

2. Pipeline configuration

The only pipeline that can optionally use rasterization is the GetVectorTile pipeline used by the VectorLayerService. If you want to use rasterization without caching of the rasterized image, you can add the following configuration location to your web.xml file:

```
classpath:org/geomajas/plugin/rasterizing/DefaultRasterizedPipelines.xml
```

If you want to maximize caching (including enabling caching for all layers), add the following location instead:

```
classpath:org/geomajas/plugin/rasterizing/DefaultCachedAndRasterizedPipelines.x
```

Alternatively you can configure for each layer individually which pipeline should be used.

The following rasterized pipelines exist (bean name):

- PIPELINE_GET_VECTOR_TILE_RASTERIZE_BUILD_URL: pipeline to get a tile for a vector layer. This will just generate the raster image URL, the rasterization will take place when the URL is requested.
- PIPELINE_GET_VECTOR_TILE_RASTERIZE: pipeline to rasterize a tile for a vector layer. This image is built but not cached.
- PIPELINE_GET_VECTOR_TILE_RASTERIZE_WITH_CACHING: pipeline to rasterize a tile for a vector layer and cache the image.

A pipeline needs to be defined for generating the raster image itself. You can set the specific pipelines to use for a layer using a configuration like this:

Example 2.1. Applying the rasterized pipeline for getVectorTile() on myLayer

```
<bean class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName">
        <util:constant static-field="org.geomajas.service.pipeline.PipelineCode
    </property>
    <property name="layerId" value="myLayer" />
    <property name="delegatePipeline" ref="PIPELINE_GET_VECTOR_TILE_RASTERIZE_B
</bean>

<bean class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName">
        <util:constant static-field="org.geomajas.plugin.rasterizing.api.Raster
    </property>
    <property name="layerId" value="myLayer" />
    <property name="delegatePipeline" ref="PIPELINE_GET_VECTOR_TILE_RASTERIZE"
</bean>
```

3. Rasterizing style configuration - SLD (Styled Layer Descriptor)

The rendering service is based on the Geotools Rendere and makes use of the SLD standard for style configuration. This means that styles are completely defined through SLD. The Geomajas style

configuration has been extended to support SLD 1.0.0, and more specifically the schema defined in GeoTools (which is also used by Udig and GeoServer). The support of SLD opens up many possibilities for styling that were previously not available. A selection of SLD styles that can be used as a source of inspiration - in addition to the standard [http://www.opengeospatial.org/standards/sld] itself, of course, which is the ultimate reference - can be found in the SLD cookbook [http://docs.geoserver.org/stable/en/user/styling/sld-cookbook/index.html] of GeoServer. The rasterizing showcase demonstrates most cookbook examples.

An example of how to configure the SLD file for a layer style in Geomajas is shown below:

Example 2.2. SLD style configuration

```
<bean class="org.geomajas.configuration.NamedStyleInfo" name="layerPolygonsRast
    <property name="sldLocation" value="classpath:org/geomajas/plugin/rasterizi
 </bean>
```

It is sufficient to let the `sldLocation` property point to the (relative) location of the SLD file.

An example of an SLD style is shown below:

Example 2.3. SLD style sample

```
<StyledLayerDescriptor version="1.0.0"
    xsi:schemaLocation="http://www.opengis.net/sld StyledLayerDescriptor.xsd"
    xmlns="http://www.opengis.net/sld"
    xmlns:ogc="http://www.opengis.net/ogc"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <NamedLayer>
    <Name>Simple Line</Name>
    <UserStyle>
      <Title>SLD Cook Book: Simple Line</Title>
      <FeatureTypeStyle>
        <Rule>
          <LineSymbolizer>
            <Stroke>
              <CssParameter name="stroke">#000000</CssParameter>
              <CssParameter name="stroke-width">3</CssParameter>
            </Stroke>
          </LineSymbolizer>
           </Rule>
      </FeatureTypeStyle>
    </UserStyle>
  </NamedLayer>
</StyledLayerDescriptor>
```

The following limitations hold for the SLD file: .

- the configuration should contain a `NamedLayer` that holds a `UserStyle`
- the user style should contain a single `FeatureStyle` with at least one rule.

4. Commands

Apart from the usage within the map - which is based on the generic `GetVectorTileCommand` - the `RasterizeMapCommand` can be used to separately fetch an image URL to a rasterized map and/or legend:

Table 2.1. RasterizeMapCommand

Id	command.rasterizing.RasterizeMap
request object class	org.geomajas.plugin.rasterizing.command.dto.RasterizeMapReques
request parameters	• clientMapInfo: the map info with the rasterizing extensions
Description	Command that creates an image of the map and the legend, adds it to the cache and returns an URL for both images
response object class	org.geomajas.plugin.rasterizing.command.dto.RasterizeMapRespon
response parameters	• mapKey: cache key for the map image • legendKey: cache key for the legend image • mapUrl: URL of the map image • legendUrl: URL of the legend image

To call the command, an extended `ClientMapInfo` object should be created client-side and passed as a parameter to the request. To facilitate this, an `ImageUrlService` is provided that perfoms this preparation stage in the common use case where the user wants to generate an image of the current screen. The following methods are provided by this service:

Example 2.4. Image URL service definition

```
public interface ImageUrlService {

    /**
     * Create map and legend images for the specified map.
     *
     * @param map the map
     * @param imageCallBack call back function
     * @param makeRasterizable should the service make the map rasterizable ?
     */
    void createImageUrl(MapWidget map, ImageUrlCallback imageCallBack, boolean m

    /**
     * Create map and legend images for the specified map. Preparing for raster
     *
     * @param map the map
     * @param imageCallBack call back function
     */
    void createImageUrl(MapWidget map, ImageUrlCallback imageCallBack);

    /**
     * Prepare the specified map for server-side rasterization.
     *
     * @param map the map
     */
    void makeRasterizable(MapWidget map);
}
```

The basic method to call is the `createImageUrl(MapWidget map, ImageUrlCallback imageCallBack)` method, which fetches the current map and legend image. The result is received asynchronously by passing a special callback interface `ImageUrlCallback`. The map preparation part can be called separately through the `makeRasterizable()` method, which is useful in the case where the user wants to make some additional changes to the rasterizing

extension data. When using this method, the preparation stage can be consequently skipped by calling the `createImageUrl(MapWidget map, ImageUrlCallback imageCallBack, boolean makeRasterizable)` method with `makeRasterizable = false`.

5. Actions and toolbar configuration

For the common use case of making a print of the current map or legend, 2 toolbar actions have been registered:

- `GetMapImageAction`: this action exports an image of the current map screen in a separate browser window
- `GetLegendImageAction`: this action exports an image of the current legend in a separate browser window

The following configuration snippet shows how to configure these actions in the toolbar:

Example 2.5. Configuration of rasterizing actions

```
<bean name="GetMapImage" class="org.geomajas.configuration.client.ClientToo
    <property name="toolId" value="GetMapImage"/>
</bean>

<bean name="GetLegendImage" class="org.geomajas.configuration.client.Client
    <property name="toolId" value="GetLegendImage"/>
</bean>

<bean name="GetLegendImageAll" class="org.geomajas.configuration.client.Cli
    <property name="toolId" value="GetLegendImage"/>
    <property name="parameters">
        <list>
            <bean class="org.geomajas.configuration.Parameter">
                <property name="name" value="showAllLayers" />
                <property name="value" value="true" />
            </bean>
        </list>
    </property>
</bean>
```

`GetLegendImageAction` has a single configuration parameter `showAllLayers`. If set to true, the generated legend image contains all layers of the layer tree. Otherwise, only the visible layers are displayed.

Chapter 3. Controllers

The rasterizing plug-in exposes 2 Spring MVC controllers for accessing rasterized images.

- `RasterizingController`: this controllers returns pre-rendered tiles from the cache, based on the image key. If the image is no longer in the cache, the same key is used to fetch the image from the rebuild cache, re-render the image, put it in the cache and return it.
- `TmsController`: this controller returns pre-rendered tiles from the cache, based on the tile parameters passed to the url. If the image is not in the cache, the url parameters are used to generate rebuild information, render the image, put it in the cache and return it. The controller is called TmsController because it tries to adhere as much as possible to the URL format prescribed by the TMS standard.

1. RasterizingController

The rasterizing controller supports 2 URL configurations:

- **/d/rasterizing/layer/{layerId}/{key}.png**

layerId	id of the server layer
key	key of the image (as returned in the tile URL of the GetVectorTileResponse)

- **`/d/rasterizing/image/{key}.png`**

key	key of the image (as returned in the legend or map URL's of RasterizeMapResponse)

This controller is normally not used outside the scope of the GetVectorTileCommand or RasterizeMapCommand. The key itself is a hash of the tile or map information combined with security information that shouldn't and can't be second guessed by the client (it is the equivalent of a session token for the tile). The fact that it requires an extra command makes it less attractive for clients that can't deal with GWT-RPC or that rely on a predetermined tile grid system like TMS. For this case we have developed the TmsController.

2. TmsController

The TMS controller also supports 2 URL configurations:

- **d/tms/{layerId}@{crs}/{styleKey}/{tileLevel}/{xIndex}/{yIndex}.png?resolution={resolution}&tileOrigin={tileOrigin}&tilewidth={tilewidth}&tileHeight={tileHeight}**

layerId	id of the **vector** server layer
crs	projected CRS of the TMS tile: e.g. "EPSG:3875"
styleKey	key of the SLD style: this is the id of the NamedStyleInfo bean or the registered key of the style when explicitly registering with the StyleService
tileLevel	tile level of the TMS tile: this determines the resolution of the tile in case the resolution is not explicitly passed as a parameter. In this case a quad-pyramid is assumed with top level 0 and width/height equal to the maximum width of the layer. It also assumes that the layer has the same CRS as the projected CRS. Reprojection is only supported if the resolution and tile origin are explicitly passed.
xIndex	index of the x-origin of the tile

yIndex	index of the y-origin of the tile
resolution	resolution of the tile (map units/pixel)
tileOrigin	origin of the tile grid. X-coordinate followed by comma and y-coordinate: <x,y>
tileWidth	tileWidth in pixels
tileHeight	tileHeight in pixels

- **d/tms/{layerId}@{crs}/{tileLevel}/{xIndex}/{yIndex}.png**

layerId	id of the **raster** server layer
crs	projected CRS of the TMS tile: e.g. "EPSG:3875"
tileLevel	tile level of the TMS tile: this determines the resolution of the tile. A quad-pyramid is assumed with top level 0 and width/height equal to the maximum width of the layer. It currently assumes that the layer has the same CRS as the projected CRS. Future implementations may support reprojection.
xIndex	index of the x-origin of the tile
yIndex	index of the y-origin of the tile

This controller does not require a previously sent GWT-RPC command to obtain any keys. As a consequence, security has to be dealt with at a different level. An allow-all security regime could be used on a selected set of layers or one can pass the userToken as a parameter and rely on the Geomajas security interceptor to prepare a security context.

Chapter 4. How-to

This chapter shows some common use cases that apply to the rasterizing plugin.