# Geomajas server documentation

**Geomajas Developers and Geosparc**

# Geomajas server documentation

by Geomajas Developers and Geosparc

1.18.6-SNAPSHOT

# Table of Contents

# List of Figures

# List of Tables

# List of Examples

# Part I. Introduction

# Table of Contents

# Chapter 1. Preface

## 1. About this document

Documentation of the Geomajas Server Project. It also includes some aspects of client-server communication.

## 2. About this project

Geomajas is a free and open source GIS application framework for building rich internet applications. It has sophisticated capabilities for displaying and managing geospatial information. The modular design makes it easily extendable. The stateless client-server architecture guarantees endless scalability. The focus of Geomajas is to provide a platform for server-side integration of geospatial data, allowing multiple users to control and manage the data from within their own browsers. In essence, Geomajas provides a set of powerful building blocks, from which the most complex GIS applications can easily be built. Key features include:

- Modular architecture
- Clearly defined API
- Integrated client-server architecture
- Built-in security
- Advanced geometry and attribute editing with validation
- Custom attribute definitions including object relations
- Advanced querying capabilities (searching, filters, style, ...)

See http://www.geomajas.org/.

## 3. License information



The project also depends on various other open source projects which have their respective licenses.

From the Geomajas source (possibly specific module), the dependencies can be displayed using the "`mvn dependency:tree`" command.

For the dependencies of the Geomajas server, we only allow dependencies which are freely distributable for commercial purposes (this for example excludes GPL and AGPL licensed dependencies).

## 4. Author information

This framework and documentation was written by the Geomajas Developers. If you have questions, found a bug or have enhancements, please contact us through the user fora at http://www.geomajas.org/ .

List of contributors for this manual:

- Pieter De Graef
- Jan De Moerloose
- Joachim Van der Auwera
- Frank Wynants
- Jan Venstermans

# Part II. Architecture

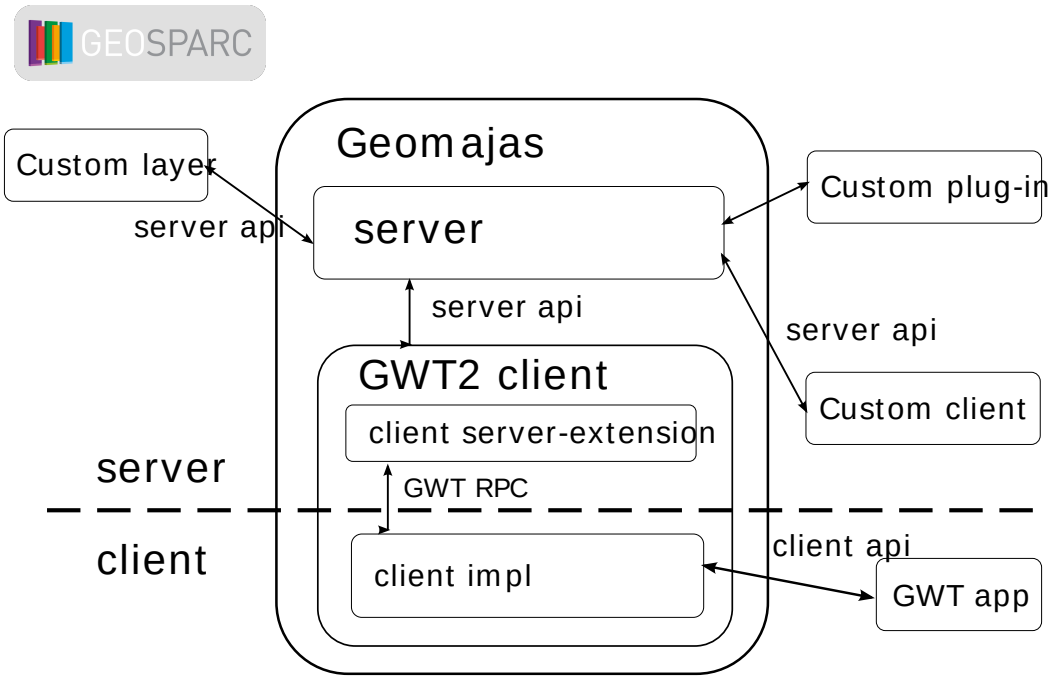
# Table of Contents

# Chapter 2. Architecture

Geomajas is an application framework which allows building powerful GIS application. We will try to look at the architecture starting from a high level overview, drilling down to discover more details.

At the highest level, Geomajas makes a distinction between the *server* and *clients*.

**Figure 2.1. Geomajas server and clients**




The server is where you configure your maps, layers and attributes/features. It is always server side. The server has an API for interaction with the outside world and for extension using plug-ins. While one of the main purposes of the server is to provide bitmaps and vector graphics for the maps and provide data about features to be rendered and edited, it contains no display code.

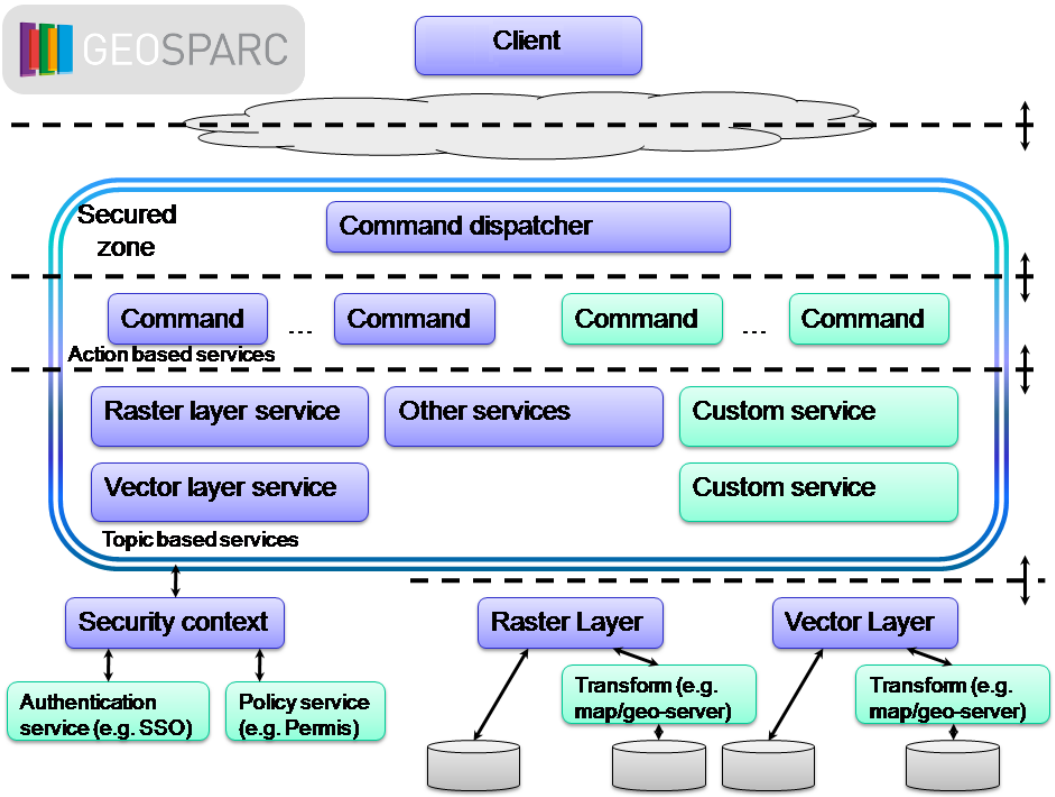
The actual display and editing is done in the clients. The server is agnostic of web (or other) display frameworks. Often, clients will consist of multiple modules. The actual client module(s), that could run standalone without a server, will not contact the server directly. Rather, an extra module in the client, the server-extension module, will act as an intermediate: the client's server-extension talks directly to the server using java calls, while the actual client modules only talks to the client's server-extension module. The communication between the two modules is private to the client. The client itself provides a additional client API. This will typically provide details about available widgets, parameters for these widgets and other possible interactions (like message queues or topics you can subscribe to).

The purpose of Geomajas is to provide rich editing of GIS data in the browser, but the clients also make other applications possible. You could unlock the maps which are configured in Geomajas using a client which makes data available as web services (this would result in a client with only a server-extension module). You could also build a java swing application using the Geomajas server by writing a swing client. This would result in a thick client application (possibly accessible using Java Web Start).

Geomajas contains two GWT based clients. The GWT2 client (from 2.x on) uses GWT libraries only. The older GWT client (1.x versions) also uses SmartGwt widgets. They both allow all development to be done in Java and GWT will handle conversion to Javascript for code which needs to run in the

browser. Obviously this integrates best with GWT based applications, but it can be combined with other web frameworks as well.

### Figure 2.2. Geomajas services



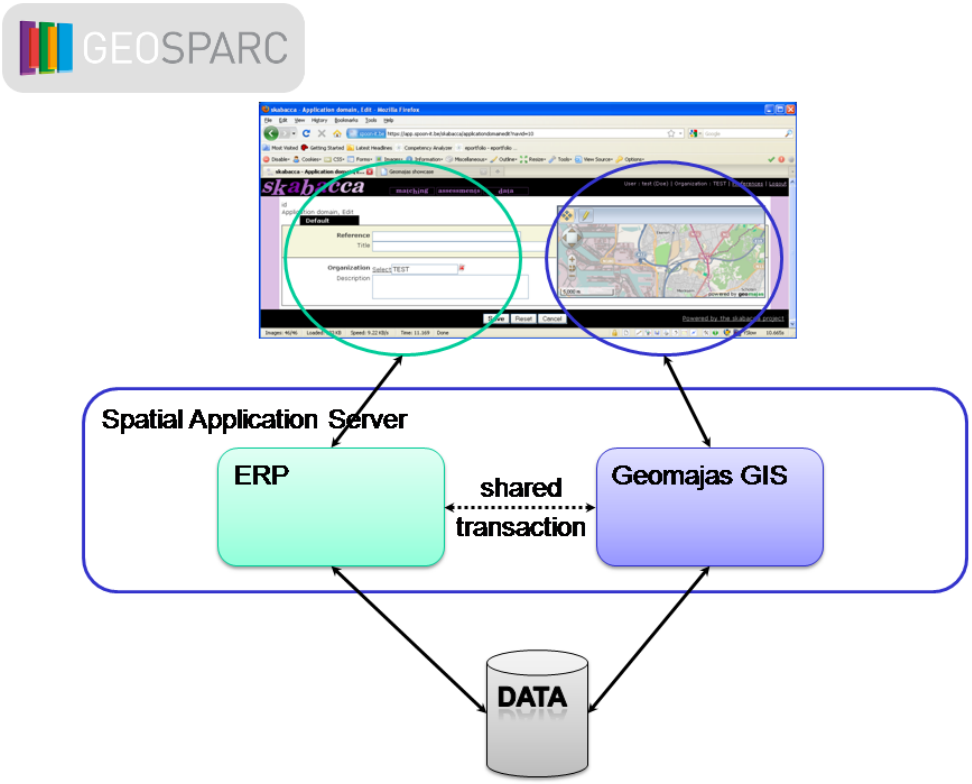

The Geomajas server is built from many services which are wired together using dependency injection (DI). This wiring is partly done automatically, and partly through the configuration files. Thanks to the inversion of control (IoC) the server is very flexible and can be customized at will.

The client-server communication is done through the command dispatcher. This delegates to one of the action based services which handle the command. These typically interact with one or more of the topic based services (though the command could also handle everything directly). The most important built-in topic based services are the raster and vector layer service. They are used to access the GIS data which is stored as either raster or vector layer.

All the services are running in a secured zone and will typically interact with the security context to verify access rights (or policies).

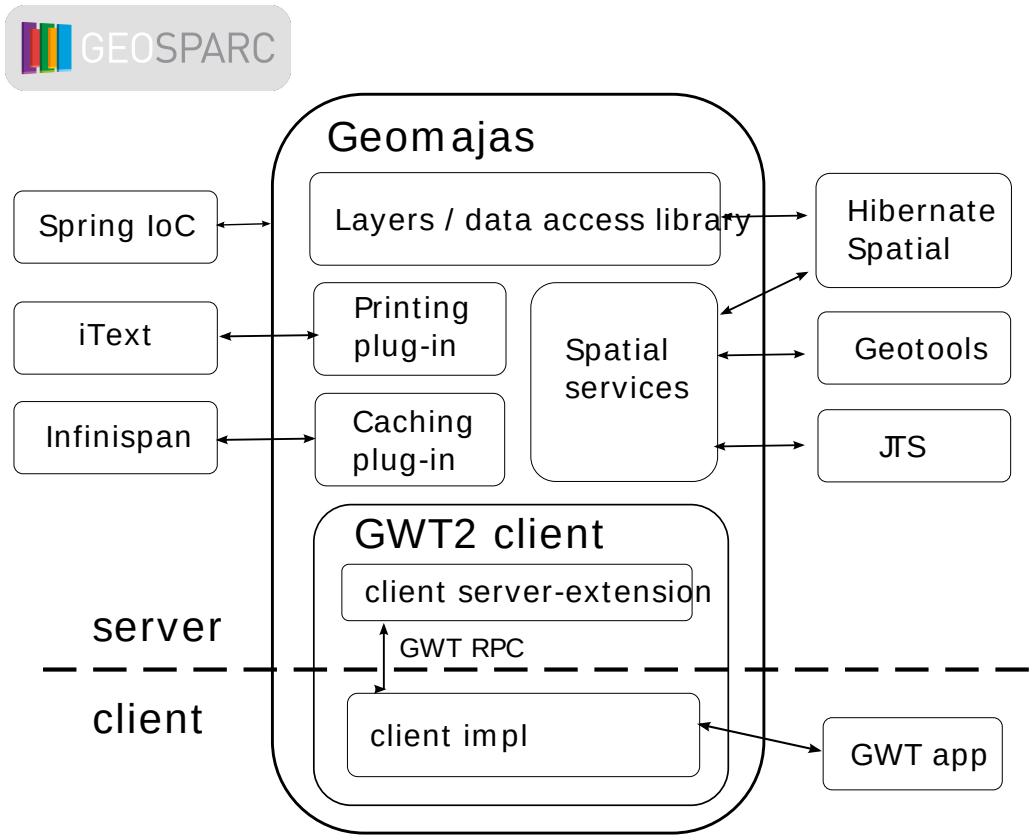
The layers access the actual GIS data, either directly or using some kind of transformation service (for example a GeoServer or MapServer instance).

**Figure 2.3. Geomajas for mashups**



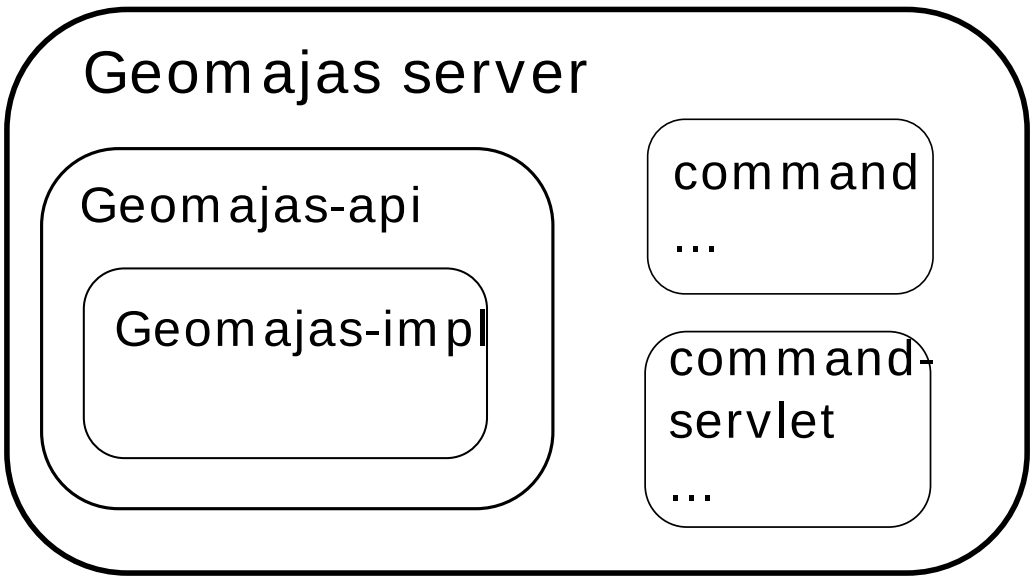

With this advanced configuration, many integration options exist. One example is displayed above, the inclusion of Geomajas in an existing application. On the client side, you just have to include the map widget in your web application. On the server side, there are many options, but you could for example assure that the transactions are shared between your existing application and Geomajas.

**Figure 2.4. Geomajas dependencies**




As is the case for most powerful frameworks, Geomajas stands on the shoulder of giants. We use some of the major open source libraries in our framework (and we integrate with a lot more).

**Figure 2.5. Geomajas server modules**




The Geomajas server is itself built from several modules which are tied together using the Spring framework (http://springframework.org/). The Geomajas-api module is a set of interfaces which

shields implementation details between the different modules. The base plumbing and some generic features are provided by the Geomajas-impl module.

There are four possible ways to extend the server.

1. *command*: commands are used as main interaction point between the client and the Geomajas server. The retrieval of maps and features, calculations, updates on the features and all all other functionalities which are available client-side are done using commands.

2. *layer*: this groups a set of access options for all details of the layers of a map. A layer can be either raster or vector based. A vector layer can be editable. The features describing the objects which are part of the vector layer are accessed through the "feature model" which converts generic feature objects into something Geomajas can use (this way, there is no need for the generic feature objects to implement a "feature" interface, allowing the use of pojos). A feature contains a geometry and can contain attributes, style and labels. Attributes can be complex, including one-to-many and many-to-one relations to other objects.

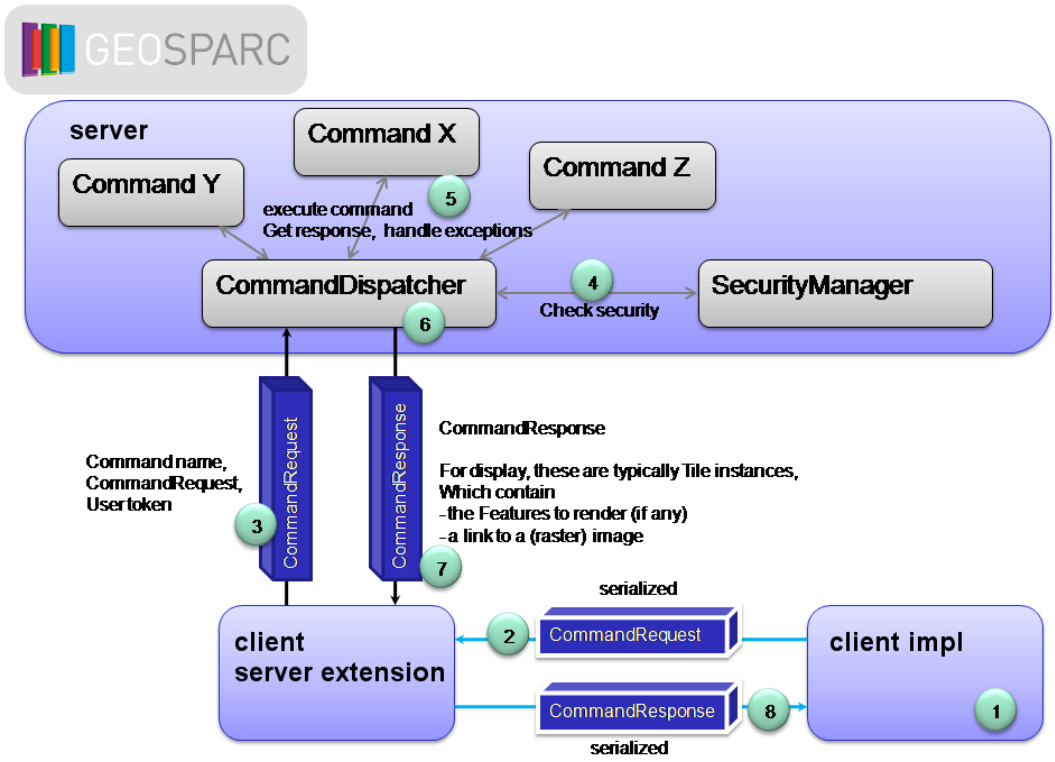
3. *pipeline*: all Geomajas server services which deal with layers are implemented using pipelines. A pipeline is a list of steps (actions) executed in order. Each pipeline can be overwritten for a layer, or you can change the default which is used when not overwritten for a layer.

   Configuring pipelines can be used to change the rendering method, add additional rendering steps (for example marking the editable area on a tile), to introduce caching,...

4. *security*: these modules contain the pluggable security features. You can configure the security services which are used to verify the validity of an authentication token and return the authorization objects based on it. These authorization objects can read the security policies from your (secure) policy store.

# 1. Command

**Figure 2.6. Geomajas client and commands**



The interaction of the client with the Geomajas server is handled using commands.

1. When a command needs to be invoked (probably as result of a user interaction), the client will build a `CommandRequest` object. This contains the name of the command to be used, the parameters for the command, and optionally the user authentication token and language of the user interface.

2. This object is transferred to the client's server-extension. For web applications, this will typically be done using an AJAX request.

3. The client's server-extension will use this `CommandRequest` to invoke the `CommandDispatcher` service, which can be obtained using the Spring context.

4. The `CommandDispatcher` will start by invoking the `SecurityManager` to check whether the execution of the requested command is allowed. If it is allowed, the actual Command is obtained using the Spring context. The `CommandResponse` object is created and the command is executed.

5. The `Command` will now do its job, writing the results in the `CommandResponse` object. When problems occur during execution of the command, it can either write this into the response object or throw an exception.

6. When the command has executed, if it threw an exception, the dispatcher will add this in the response object. It will then convert any exceptions in the response object into some messages which may be sensible to the user (put the message in the correct language in the result object, assuring the "cause" chain is also included). Some extra information is also added in the response object (like command execution time).

7. The `CommandResponse` is returned to the client's server-extension.

8. The client's server-extension serializes the `CommandResponse` back to the client.

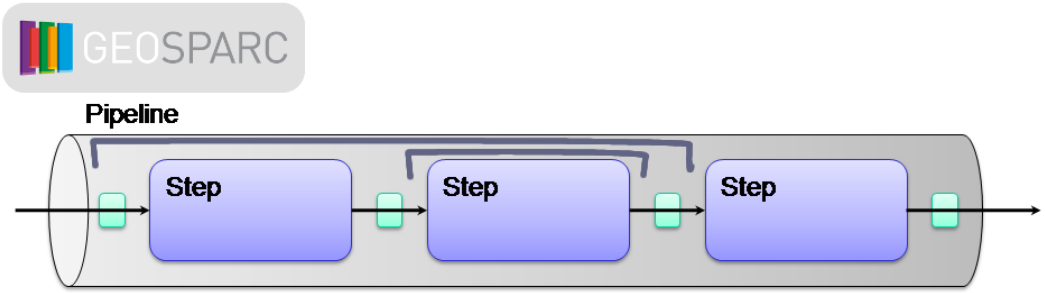
When the command had something to do with rendering, then the response object is likely to contain a `Tile`.

# 2. Pipelines

Pipelines are used in Geomajas to allow extreme configurability of the services which choose to use them.

They are comparable with Business Process Modeling (BPM) processes. At first sight pipelines are much more limited as the steps are always sequential, only allowing each step to either continue to the next stop or stop the pipeline. Nesting pipelines gives back the expressive power of general BPM processes. A step could loop over another pipeline, conditionally execute a pipeline, start several pipelines for parallel processing etc. An important difference is the configurability of pipelines. Pipelines are selected on a combination of pipeline name and the layer on which the pipeline operates. When defining pipelines, you can either define them from scratch, copy an existing pipeline or copy and extend a pipeline. A pipeline can be defined with extension hooks and these hooks can be used to add additional pipeline steps. You can also add interceptors on a pipeline which introduces some code to execute before and after the steps which are intercepted and allows you to skip the execution of the intercepted steps.

**Figure 2.7. Geomajas pipeline architecture**




# 2.1. Pipeline architecture

All the layer access services provided by the Geomajas server are implemented by invoking a pipeline. Using `PipelineService`, blocks of functionality become reusable and highly configurable with limited coupling between the *pipeline step*s.

Some of the services which are implemented as `PipelineService` include:

- `RasterLayerService`: methods for accessing a raster layer, especially getting tiles for a raster layer.
- `VectorLayerService`: methods for accessing a vector layer, for example for getting the features or obtaining vector tiles.

Pipelines can nest. One of the steps in the default "vectorLayer.saveOrUpdate" pipeline will loop over all features to be updated and invoke the "vectorLayer.saveOrUpdateOne" pipeline for each.

Pipelines are server side only, client access is typically made available by invoking a command.

Pipelines are typically invoked for a specific layer. In that case, the default pipeline can be replaced by a layer specific pipeline. This way, layer specific configurations (like optimizations or specific rendering) can be applied. When a layer does not overwrite a pipeline, the default is used. Pipelines are always selected on pipeline name. You can create the layer specific pipeline by setting the layer id for which it applied. When several pipelines have the same steps, you can define the pipeline once, and refer to it later by using a pipeline delegate instead of a list of steps.

A pipeline consists of a number of steps. A pipeline is configured using a `PipelineInfo` object which details the pipeline id and steps. When a pipeline is started (using the `PipelineService`) it executes the pipeline steps until the pipeline is finished (a status which can be set by one of the steps), or until no more steps are available in the pipeline. Each step gets two parameters.

- a context which contains a map of (typed) objects which can be used to pass data between the steps (including initial parameters for the pipeline).
- the result object which can be filled or transformed during the pipeline's execution.

Pipelines can be extended in two ways. When a pipeline is defined, it is possible to include hooks for extensions. These are special no-op steps. When a pipeline is defined, your can either define all the pipeline steps, or refer to a delegate pipeline combined with a map of extension steps. The pipeline will then be based on the delegate pipeline with the extensions steps added after the hooks with matching names.

Pipelines can be also enhanced with interceptors. An interceptor will intercept a block of consecutive steps, allowing to perform an action before and after the block is executed. Depending on the return value of the before action, the steps (and optionally the after action) can be skipped. This can for example be used to implement functionality like caching and auditing.

## 2.2. Application in the server

All the methods in both RasterLayerService and VectorLayerService are implemented using pipelines.

# 3. Layer

The layer extensions allow determining how a layer is built, which data is part of the layer, update and creation of extra data on a layer.

A `Layer` has some metadata (id, coordinate system, label, bbox, stored in the `LayerInfo` object) and allows you get obtain the layer data.

# 4. Security

The data which is accessed using Geomajas can be security sensitive. Geomajas includes all the measures to assure protection of sensitive or private data.

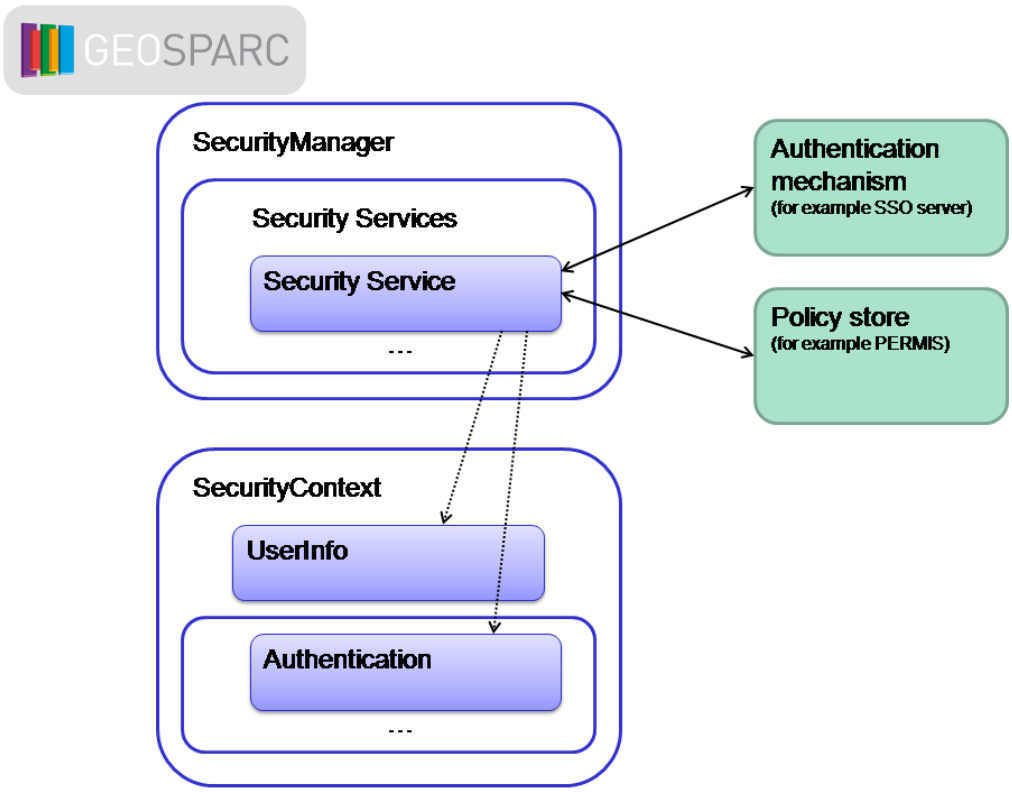
## 4.1. Security architecture

Geomajas is built to be entirely independent of the authentication mechanism and the way to store policies.

Based on the user who is logged into the system, the following restrictions can apply:

- access rights to a command
- access rights for a layer
- a filter which needs to be applied for a layer
- a region which limits the data which may be accessed for a layer
- access rights on the features
- access rights on the individual attributes of the features

**Figure 2.8. Security architecture**




To assure the authentication mechanism is pluggable, an *authentication token* is used. The authentication token is used to determine the *security context*. The security context contains the *policies* which apply and which can be queried.

A list of *security services* can be defined (using Spring bean `security.SecurityInfo`). This list can be overwritten in configuration. By default the list is empty, which prohibits all access to everyone. The server does however include a security service which can be used to allow all access to everyone.

The security service is responsible for converting the authentication token into a list of *authorization objects*. The security manager will loop all configured security services (using Spring bean `security.SecurityInfo`) to find all the authorization objects which apply for the token. By default the security manager will stop looping once one of the security services gave a result. You can change this behaviour to always combine the authorization objects from all security services.

## Note

The system explicitly allows authentication tokens to be generated by different authentication servers. In that case for each authentication server, at least one security service should be configured in Geomajas. However, when using such a configuration, you *have to* verify that the authentication tokens which are generated by the different servers cannot be the same.

In many systems (including Role-Based Access Control (RBAC) systems) an authorization object matches a roles for the authenticated user.

Note that, as the actual authentication mechanisms are handled by the security services, Geomajas does not know any passwords or credentials. Similarly the secure, tamper-proof storage of policies is not handled by Geomajas either.

Details about the current authentication token and access to the policies (using the authorization objects) is available using the `SecurityContext`. The security context is thread specific. When

threads are reused between different users, the security context needs to be cleared at the end of a request (group). This is normally handled by the clients.

The following general authorization checks exist:

- `isToolAuthorized(String toolId)`: true when the tool can be used. The "toolId" matches the "id" parameter which is used in the configuration as specified using the `ToolInfo` class.
- `isCommandAuthorized(String commandName)`: true when the command is allowed to be called. The "commandName" parameter is the same as the command name which is passed to the `CommandDispatcher` service.

And for each layer:

- `isLayerVisible(String layerId)`: true when (part of) the layer is visible.
- `isLayerUpdateAuthorized(String layerId)`: true when (some of) the visible features may be editable.
- `isLayerCreateAuthorized(String layerId)`: true when there is an area in which features can be created.
- `isLayerDeleteAuthorized(String layerId)`: true when (some of) the visible features may be deleted.
- `getVisibleArea(String layerId)`: only the area inside the returned geometry is visible (uses layer coordinate space). All features which fall outside the layer's MaxExtent area are also considered not visible.
- `getUpdateAuthorizedArea(String layerId)`: only the area inside the returned geometry may contain updatable features (uses layer coordinate space). All features which fall outside the layer's MaxExtent area are also considered not updatable.
- `getCreateAuthorizedArea(String layerId)`: only the area inside the returned geometry can new features be created (uses layer coordinate space). All features which fall outside the layer's MaxExtent area are also considered not creatable.
- `getDeleteAuthorizedArea(String layerId)`: only the area inside the returned geometry may contain deletable features (uses layer coordinate space). All features which fall outside the layer's MaxExtent area are also considered not deletable.
- `getFeatureFilter(String layerId)`: get an additional filter which needs to be applied when querying the layer's features.
- `isFeatureVisible(String layerId, InternalFeature feature)`: check the visibility of a feature.
- `isFeatureUpdateAuthorized(String layerId, InternalFeature feature)`: check whether a feature is editable.
- `isFeatureUpdateAuthorized(String layerId, InternalFeature oldFeature, InternalFeature newFeature)`: check whether the update contained in the feature is allowed to be saved.
- `isFeatureCreateAuthorized(String layerId, InternalFeature feature)`: check whether a feature is allowed to be created.
- `isFeatureDeleteAuthorized(String layerId, InternalFeature feature)`: check whether deleting the specific feature is allowed.
- `isAttributeReadable(String layerId, InternalFeature feature, String attributeName)`: check the readability of an attribute. The result can be feature specific.

- `isAttributeWritable(String layerId, InternalFeature feature, String attributeName)`: check whether an attribute is editable. The result can be feature specific.

These authorizations are split in several groups. The security service is not required to provide an implementation of each authorization test (see API), the security context always ensures that all methods are available.

Checking authentication and fetching the authorization details can be time consuming (not to mention the hassle of logging in again). To solve this, the results of the security services can be cached. When a security service can authenticate a token, the reply can contain details about the allowed caching. Three parameters are allowed to be passed, the `validUntil` and `invalidAfter` timestamps and an `extendValid` duration.

The security manager first checks the cache to find (valid) authentication results. A cache entry is only valid until the "validUntil" timestamp. When an entry is valid, validUntil may be extended until "now" plus "extendValid" duration. However, "validUntil" is never extended past "invalidAfter". When no data can be found in the cache, the security services are queried.

### Note

There may be multiple authentications stored for a authentication token. When one of them becomes invalid, they are all considered invalid.

### Note

Entering credentials is never handled by Geomajas. When the authentication token cannot be verified, a security exception is thrown. It is up to the client application to assure that a new authentication token is created.

The authorization have two possible results. When reading or rendering, all unauthorized data should be filtered without warning or exception. When trying to invoke a command or to create, update or delete, any attempt which is not authorized should result in a security exception.

The security uses the approach that all access is forbidden unless is is allowed. Hence, you will always need to configure at least one security service to assure the system is usable.

## 4.2. Interaction between client and server

When a user wants to access a secured Geomajas application, she will normally surf to the URL for the application.

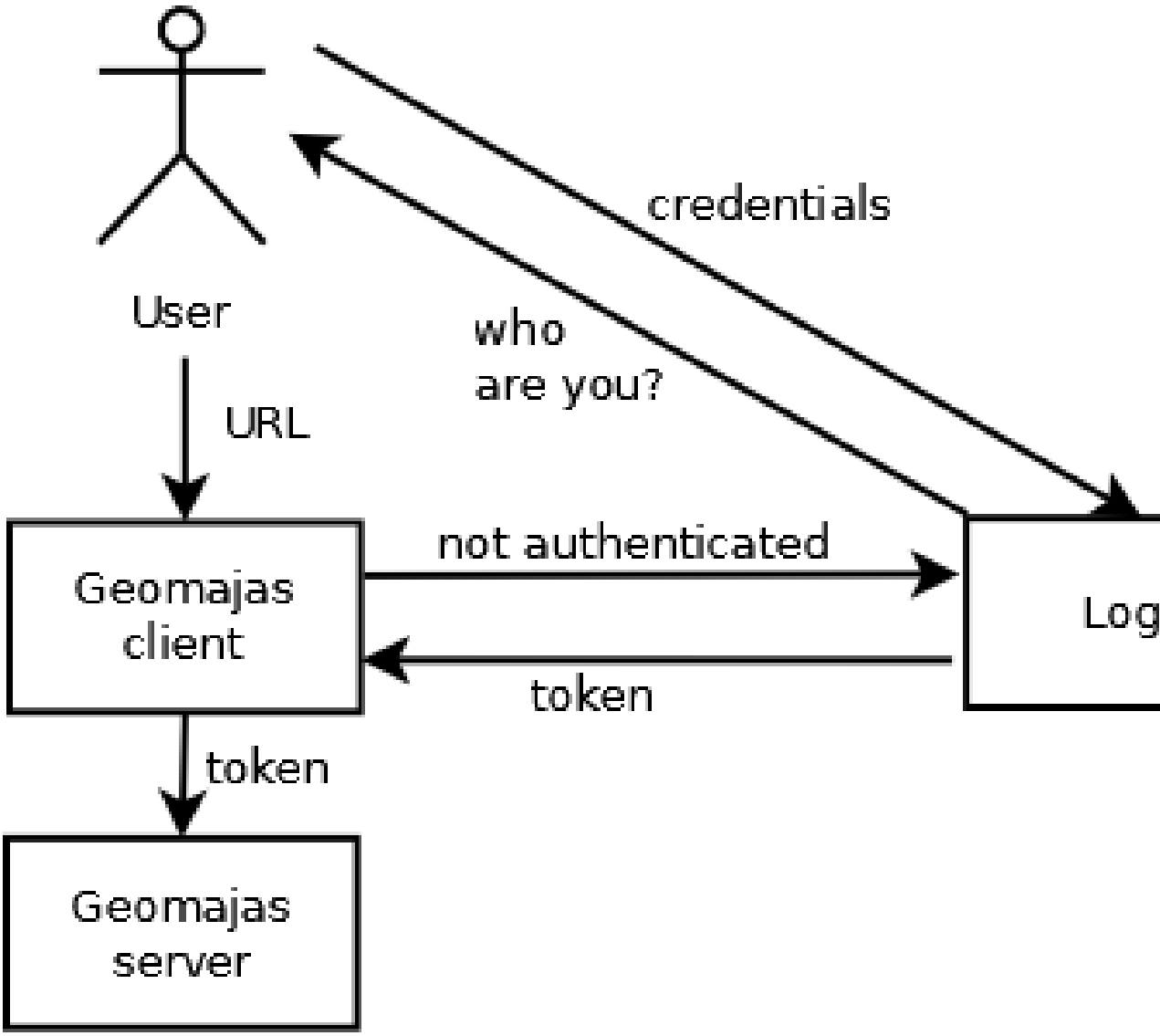
The application will then check whether the user is logged in. If that is not the case, the user is redirected to the login page. This may be an external page as provided by an authentication server (as often used for SSO (Single-Sign-On) solutions), or it could be a login widget. Note that the framework does not handle this redirection itself or even know how the login can be handled. It is up to the application writer to provide this redirection.

The login page will ask the user to provide her credentials. This could be a user name, password pair, a request for a code coming from a hardware device, login using a eID or some other means PKI (Private Key Infrastructure), automatic recognition based on IP address,... When the login handling is satisfied with the provided credentials (the user is really authenticated), it will redirect back to the original application with a security token.

This token is then used by the client to pass authentication information to the server.

As seen from this example, the Geomajas client does not handle the authentication and does not need to know the credentials for the user.

**Figure 2.9. Logging into a Geomajas system**




## Note

It is important to know that the security token typically has a limited validity. As such, it can happen at any moment that the token is no longer valid and the login screen needs to be presented again. The application author should consider this while developing.

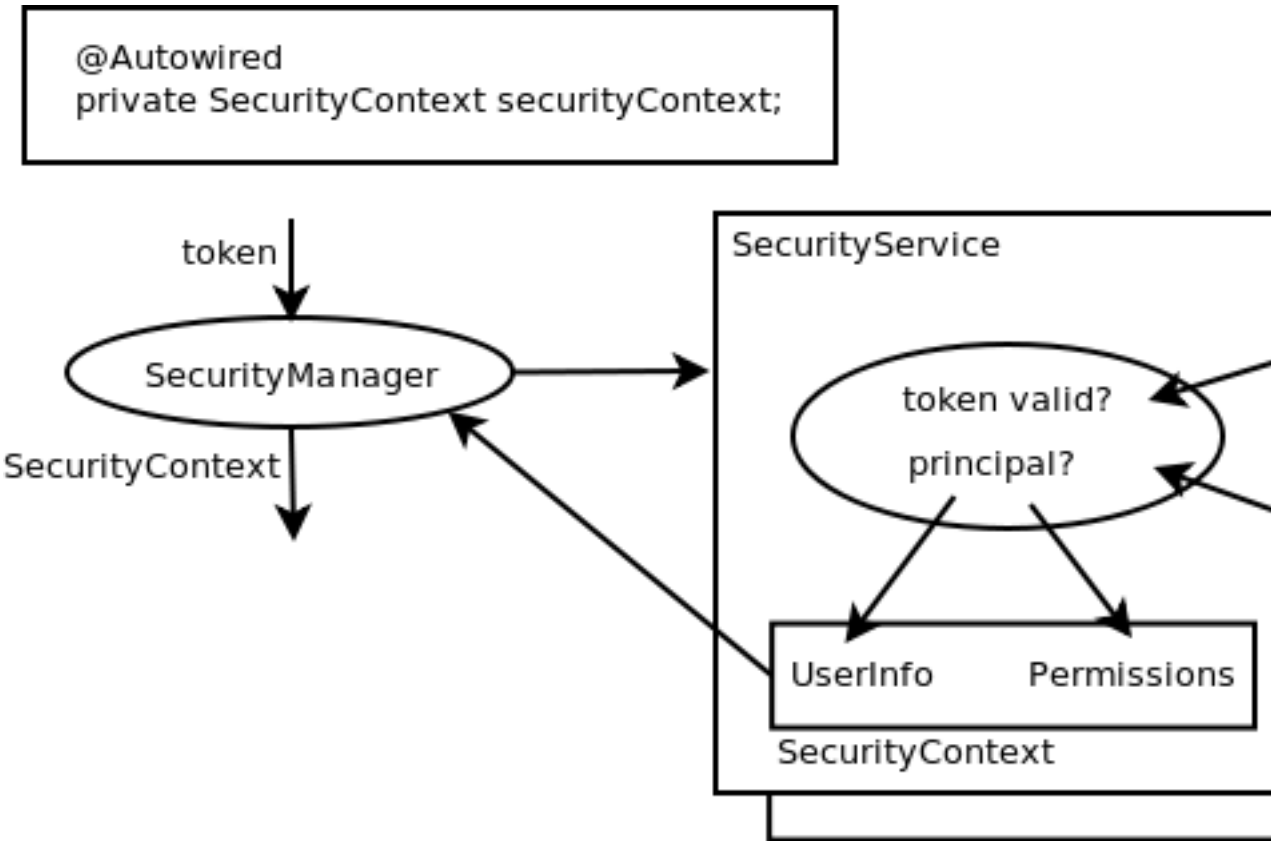
At the server, the programmer has a reasonably simple job. At each policy enforcement point, you need an injected security context which can be used for the policy decisions. This can be included using the following piece of code.

```
@Autowired
private SecurityContext securityContext;
```

The framework handles the instantiation of this security context based on the security token (which is typically received either through the CommandDispatcher's request or a URL parameter). This is done by the SecurityManager. The SecurityManager uses the configuration supplied by Spring bean `security.SecurityInfo`. Each of the security services will check whether the token is valid (which includes checking whether it was supplied by the authentication services which backs the service) and extract the principal from the token. That principal is used to fill some information about

the user (like name, to allow the client to display this) and to read the policies from the policy store. The policies are converted by the service to Authentication objects. The authentication objects are combined in the security context, only allowing things which are allowed by at least one authentication object.

**Figure 2.10. Building the security context**




Both the authentication service and policy store are outside of the Geomajas framework. They can be external services which are accessed by the security services or implemented as part of the security service implementation.

## 4.3. How is this applied ?

The security is applied throughout Geomajas. A list of places (which is not necessarily complete) and some additional ideas for application follow.

Server services:

- `CommandDispatcher` verifies the existence of a `SecurityContext` and creates one if needed.
- `CommandDispatcher` verifies whether the command is allowed to be used.

VectorLayerService:

- Check layer access.
- Handle the "filter" for the layer (if any).
- Filter on visible area as this can increase query speed.
- Post-process features filtering unreadable attributes, set update flags, remove features which are not allowed.

Commands:

- configuration.Get and configuration.GetMap: layers which are invisible should be removed, tools which are not authorized should be removed, "editable" and "deletable" statusses on layers, features, attributes need to be set.
- configuration.UserMaximumExtent: max extent should only consider visible features.
- feature.PersistTransaction: making changes to attributes which are not editable should cause a security exception.
- feature.SearchByLocation: only return visible features and readable attributes.
- feature.SearchFeature: only return visible features and readable attributes.
- geometry.Get: only return the geometry for visible features.
- geometry.MergePolygon: no security implications.
- geometry.SplitPolygon: no security implications.
- render.GetRasterTiles: should only return data for visible layers, ideally post-processing the image to ensure only visible area is included (making the rest transparent).
- render.GetVectorTile: should only return data for visible layers, only display visible features, only return visible features, only render visible features. When attributes need to be included, only readable attributes should be included and the "editable" flag needs to be set.

Rendering:

- The individual rendering steps (especially the layer/feature model) can use the `SecurityContext` to filter the data they produce.
- Images can have areas masked which are not allowed to be seen.
- The rendering pipeline can include steps which check the security. This can make life easier on the layer model which are not guaranteed (or forced) to handle all security aspects. These are active by default but can be removed for speed (when you are sure this is double work).

Cache:

- The caching needs to consider the access rights when storing and retrieving data.

Client:

- The client is responsible for assuring a authentication token is included in all access to the server.
- The "get configuration" commands filter the data to assure invisible layer and attributes and tools which are not allowed are not displayed. No action needed.
- Specific tests on editability of individual features and attributes would be useful to assure the user does try to enter or modify data which cannot be saved.
- The client should ask for credentials again when the token was not available or is no longer valid. Specifically when a `GeomajasSecurityException` is received which code `ExceptionCode.CREDENTIALS_MISSING_OR_INVALID`.

## 4.4. Server configuration

While this is not really touched by description above, the following system configuration issues are likely to be important when you want to secure your Geomajas application.

- Make sure the communication between the client and server uses encryption, possibly by using https. This prevents snooping of your data and/or hijacking the security token.
- Even if your application is using http for some reason, at the very least your authentication method should use https to prevent your passwords from being transmitted on the wire in cleartext. I would expect all authentication servers do this.
- Depending on your needs, it may make sense to store the data encrypted on the server. If you want that, your need a layer model which can access your secured data (possibly passing on the security token).

# Chapter 3. Project structure

The project is built from a large set of modules. A specific application can choose which modules are used or not. In principle, the server module are always required and at least one client and at least one layer plug-in. More plug-ins or clients can be added as needed.

# 1. Plug-in registration

Plug-ins are automatically discovered when available on the classpath. This is done using two files: META-INF/geomajasContext[suffix].xml and META-INF/geomajasWebContext[suffix].xml.

The suffix is meant to deal with classpath issues where context files overwrite each other. Each plug-in can therefore define it's own geomajas(Web)Context.xml file, the classpath is scanned for META-INF/geomajas(Web)Context*.xml

- The geomajasContext.xml file contains information about the plug-in, the dependencies for the plug-in (which are checked when the application context is built, assuring that the set of plug-ins is complete and can be combined) and contains copyright and license information for the plug-in and its dependencies. Additional beans and services can also be defined.

- The geomajasWebContext.xml file is provided to allow additional endpoints to be added in the web tier. Geomajas normally installs a `DispatcherServlet` in the web.xml file to allow additional web endpoints to be added using Spring MVC.

# 2. Module Overview

Different modules have different impacts and different purposes. Therefore different categories of modules are required. In Geomajas server, modules can be categorizes as:

- *core modules*: these are essential Geomajas modules. Each Geomajas application needs these modules. However, you also need some a client and some plug-ins (like layers) or you won't be able to do much.

- *plug-in modules*: modules that extend the Geomajas Server core function. This can either add new functionality, add support for a certain type of data source, provide a security service or a combination.

Full list of Geomajas Server modules:

**Table 3.1. List of Geomajas Server core modules**

| Name | Purpose |
|---|---|
| geomajas-api | Stable interfaces. Reference guide for other modules. |
| geomajas-api-experimental | Experimental interfaces. This contains some experimental stuff which may be promoted to the supported API when useful, or may be changed or dumped. As this is *not* part of the API, it may change between revisions. |
| geomajas-command | Lists all basic commands. |
| geomajas-common-servlet | Code which is shared by the different clients which are servlet based. |
| geomajas-impl | Main library with default implementations. |
| geomajas-testdata | Module which contains data which is used for testing Geomajas. |

### Table 3.2. List of Geomajas Server plug-in modules

| Name | Purpose |
|---|---|
| geomajas-layer-geotools | Support for any data format GeoTools supports and which has a GeoTools data store defined for it (http://geotools.org/javadocs/org/geotools/data/DataStore.html). This also allows access data from an ESRI shape file, handled in memory, using the `ShapeInMemLayer`. |
| geomajas-layer-googlemaps | Support for GoogleMaps raster format. This allows access to the normal and satellite views provided by Google. You still have to make sure you comply with Google terms of use (http://code.google.com/apis/maps/) |
| geomajas-layer-hibernate | Support for database formats through Hibernate. Allow access to any kind of features which are stored in a spatial (relational) database. The data is accessed using the hibernate and hibernate-spatial open source libraries. |
| geomajas-layer-openstreetmap | Support for raster data coming from the OpenStreetMap project (http://www.openstreetmap.org/). |
| geomajas-layer-wms | Support for the WMS raster format. Access data from a WMS server (http://www.opengeospatial.org/standards/wms). |
| geomajas-plugin-cache | Caching plug-in which allows improved speed by calculating data only once. |
| geomajas-plugin-geocoder | Geocoder support, using various geocoding services. |
| geomajas-plugin-print | Adds printing capabilities beyond printing in the browser, by delivering the map as PDF. |
| geomajas-plugin-rasterizing | Rasterize vector layers on the server. |
| geomajas-plugin-reporting | Build reports (using JasperReports) for features, possibly including a map. |
| geomajas-plugin-staticsecurity | Simple security service which allows including the entire security configuration in the Spring configuration files. It does not use an external source for users or policies, making the configuration static. |

# Part III. API

# Table of Contents

# Chapter 4. API contract

## 1. API annotation

As Geomajas is a framework for building enterprise application, it is important to be very accurate about what exactly is considered part of the API, specifically which classes and interfaces and which methods in these classes and interfaces are considered as part of the API.

For this reason, we have introduced the "`@Api`" annotation. A class or interface is only considered part of the public API when it is annotated using "`@Api`" in a final release. When all public methods in the class or interface are considered part of the API, you could use "`@Api(allMethods = true)`". The alternative is to annotate the individual methods.

The API includes many interfaces. These interfaces should only be implemented by client code when they are annotated by "`@UserImplemented`". All other interfaces are provided to indicate the methods available on instances which are obtained through the API or Spring wiring and may have extra methods added in future versions.

All classes and methods which are indicated with "`@Api`" should also have a "`@since`" javadoc comment indicating the version in which the class or method was added to the API.

### Note

Please beware that only the annotations determine whether something is part of the API or not. The manual may discuss things which are not considered API, probably because they are experimental.

### Note

There is also a @FutureApi annotation which indicates that the annotated class/interface/method is very likely to become API in one of the following releases, but we would like some extra validation.

## 2. Server API

The full details about the API can be found in the published javadoc, available on the Geomajas site at http://files.geomajas.org/javadoc/geomajas-project-server/1.18.0/geomajas-project-server-javadoc/ [http://files.geomajas.org/javadoc/geomajas-project-server/1.18.0/geomajas-project-server-javadoc/].

The API for the Geomajas server is contained in the geomajas-api module. This contains only interfaces, exceptions and data transfer objects. The data transfer objects are classes which only contain getters and setters. The server API is divided in the following packages:

- *command*: interfaces, services and data transfer objects related with the command extension points.
- *configuration*: data transfer objects which are used for defining the configuration in Geomajas.
- *geometry*: Geomajas geometry related data transfer objects.
- *global*: some general interfaces, annotations and exceptions which are relevant for a combination of several extension points or the entire API.
- *layer*: interfaces, services, exceptions, data transfer objects and some internal objects related with the layers and objects in a layer. These include the definition of a layer, tiles, features and feature models.

- *security*: interfaces, services and data transfer objects related with the security extension points and security handling.
- *service*: utility services provided by Geomajas.

The server also contains a module geomajas-api-experimental. This contains some experimental stuff which may be promoted to the supported API when useful, or may be changed or dumped. As this is *not* part of the API, it may change between revisions.

# 3. Command and plug-in API

For commands and plug-ins, the same rule applies as for the server API. That means that the "`@Api`" annotation indicates the stability of the interfaces, classes and methods.

These classes can typically be found in packages containing "command.dto" for command request and response objects or packages containing "configuration" for objects which are expected to be defined from the Spring configuration files.

The command name is also considered part of the API when the implementing class in annotated using the "`@Api`" annotation.

# 4. API compatibility and Geomajas versions

Final Releases have a versions with structure "major.minor.revision". Intermediate version (snapshots and milestones) have a structure "major.minor.revision-QUALIFIER".

The major number indicates major changes in the framework and thus gives no guarantee about API compatibility with previous major versions.

Minor versions are used for adding features. Revisions are only produced when bugs need to be fixed which cannot wait for the next minor release (or when the previous revision was rejected in the release vote).

The API for Geomajas needs to be upward compatible for all stable versions with same major number. Specifically this means that

- No API classes or interfaces may be removed.
- No API classes or interfaces may be renamed.
- No API classes or interfaces may have their package name modified.
- No API methods may be removed.
- No API methods may have their signature changed.
- No methods may be added to classes annotated using "`@UserImplemented`".

Additionally, all methods and classes which are added should include an indication of the version in which the class and/or method was added. This is done using the "`@since`" javadoc comment for the methods, class or interface.

Because of the guarantees about API, the use of the "`@Deprecated`" annotation only indicates that a method or class is not recommended to be used. The method or class will not be removed in future versions with the same major number.

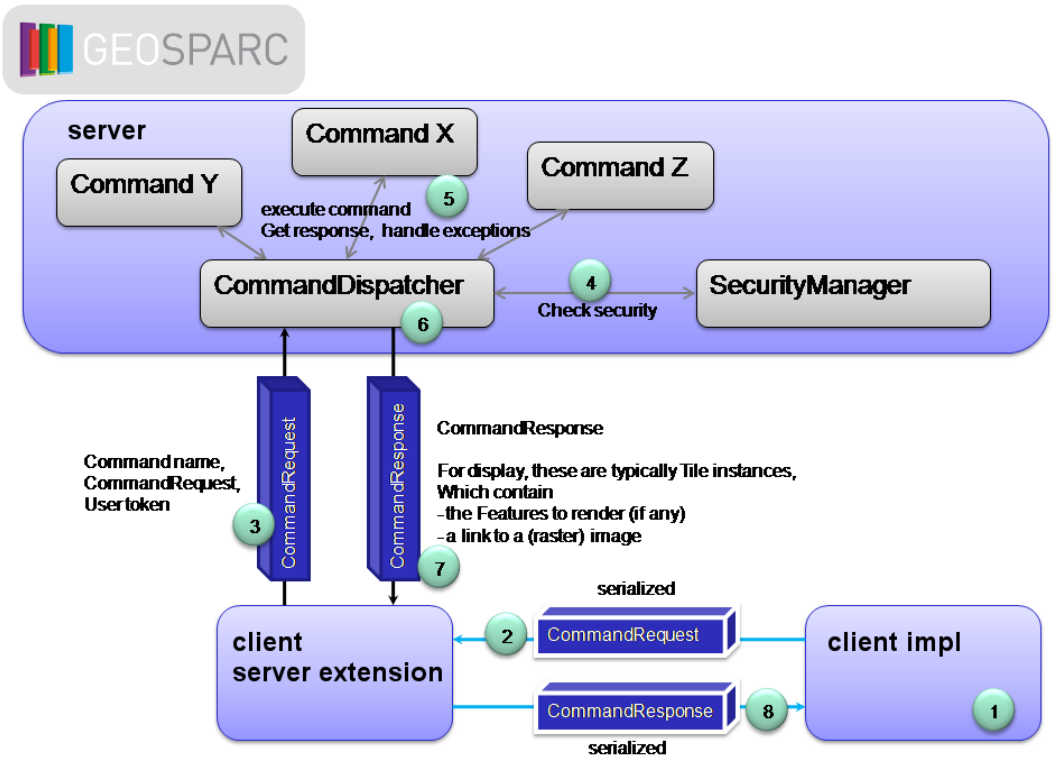
Intermediate releases are ordered (when comparing versions) alphabetically. However, the @Api annotation only becomes effective on a final release. If a milestone is released (e.g. "1.2.0-M1" then this release does not guarantee anything about the @Api annotations).

# Chapter 5. Commands

## 1. CommandDispatcher service

The command dispatcher is the main command execution service. It accepts commands serializable data for executing a command and returns a response which can again be serialized. It is the main entry point into the server for use by the clients.

**Figure 5.1. Geomajas client and commands**




The following methods are provided:

- `CommandResponse execute(String commandName, CommandRequest commandRequest, String userToken, String locale)` : given the command name, request object, user token and locale, try to execute the requested command. The result, including any exception which may have been thrown are included in the returned response object.

## 2. Provided commands

The commands are all registered in the Spring context. The "registry key" as indicated below is used to retrieve the commands. These are services, so a singleton should be sufficient for this.

The default naming for the keys is derived from the fully qualified class name. This is automatically assigned when the command is in a (sub package of) the "command" package. To determine the bean name, all parent packages of the "command" package are removed. Then the name is simplified. It will end up having "command." as prefix, optionally followed by a package, followed by the name. As there already is a "command" prefix, the "Command" suffix is removed from the name if present. When the resulting name starts or end with the sub package, then that is removed as well. For example the "org.geomajas.command.configuration.GetConfigurationCommand" class will get "command.configuration.Get" as registry key.

### Table 5.1. CopyrightCommand

| **CopyrightCommand** | |
|---|---|
| Registry key | command.general.Copyright |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.EmptyCommandRequest |
| Parameters | none |
| Description | This allows you to obtain copyright and license information for Geomajas, it's dependencies, the plg-ins and the dependencies of the plug-ins. This can be used to display that information in a "about" box to assure the copyright and license conditions are adhered. |
| Response object class | org.geomajas.command.dto.CopyrightResponse |
| Response values | List of `CopyrightInfo` objects for the dependencies. Any duplicates are removed based on the copyright info key. |

### Table 5.2. GetConfigurationCommand

| **GetConfigurationCommand** | |
|---|---|
| Registry key | command.configuration.Get |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.EmptyCommandRequest |
| Parameters | none |
| Description | Get the client side configuration information. This returns information about all maps which have been configured. |
| Response object class | org.geomajas.command.dto.GetConfigurationResponse |
| Response values | • *name*: name of the application.<br>• *maps*: list of configured maps for the application. Note that the layer information which is contained in the maps has the coordinates transformed according to the crs of the map.<br>• *screenDpi*: screen resolution in dots per inch. |

### Table 5.3. GetMapConfigurationCommand

| **GetMapConfigurationCommand** | |
|---|---|
| Registry key | command.configuration.GetMap |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.GetMapConfigurationRequest |
| Parameters | • *mapId*: id of map for which the information should be returned. |
| Description | Get the client side configuration information for the specified map. |
| Response object class | org.geomajas.command.dto.GetMapConfigurationResponse |
| Response values | • *mapInfo*: information about the requested map. Note that the layer information which is contained in the maps has the coordinates transformed according to the crs of the map. |

### Table 5.4. GetRasterTilesCommand

| **GetRasterTilesCommand** | |
|---|---|
| Registry key | command.render.GetRasterTiles |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.GetRasterTilesRequest |
| Parameters | • *crs*: coordinate reference system that the map uses.<br>• *bbox*: total bounding box wherein to fetch raster tiles.<br>• *scale*: current scale in the client side map.<br>• *layerId*: the id of the raster layer to fetch tiles for. |
| Description | Retrieve a set of raster tiles as image links for a given layer within a certain bounding box expressed in a certain coordinate reference system. |
| Response object class | org.geomajas.command.dto.GetRasterTilesResponse |
| Response values | • *rasterData*: list of `RasterTile` objects.<br>• *nodeId*: identifier to be used in the DOM tree. |

### Table 5.5. GetVectorTileCommand

| **GetVectorTileCommand** | |
|---|---|
| Registry key | command.render.GetVectorTile |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.GetVectorTileRequest |
| Parameters | • *layerId*: the id of the vector layer to fetch a tile in.<br>• *code*: the unique code of the tile to retrieve.<br>• *scale*: the current scale on the map, client side.<br>• *panOrigin*: translation for the tile on the client-side.<br>• *filter*: extra filter that can be used to filter out data.<br>• *crs*: the map's coordinate reference system.<br>• *renderer*: should the server render to SVG or VML?<br>• *styleInfo*: extra styles that can override the originally configured styles.<br>• *paintGeometries*: should the geometries be painted in the tile? This is true by default.<br>• *paintLabels*: should labels be painted in the tile?<br>• *featureIncludes*: indication of which data to include in the feature. Possible values (add to combine): 1=attributes, 2=geometry, 4=style, 8=label. Default value is to include everything. |

| **GetVectorTileCommand** | |
|---|---|
| Description | Fetches a single tile for a vector layer. The tile can contain both vectors and labels. This command is used to paint vector layers in the map. |
| Response object class | org.geomajas.command.dto.GetVectorTileResponse |
| Response values | • *tile*: the actual resulting tile. |

### Table 5.6. LogCommand

| **LogCommand** | |
|---|---|
| Registry key | command.general.Log |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.LogRequest |
| Parameters | • *level*: log level, 0 for debug, 1 for info, 2 for warn, 3 for error.<br>• *statement*: string which needs to be logged. |
| Description | This allows you to send a statement to the server side which will be logged there. |
| Response object class | org.geomajas.command.CommandResponse |
| Response values | none |

### Table 5.7. PersistTransactionCommand

| **PersistFeatureTransactionCommand** | |
|---|---|
| Registry key | command.feature.PersistTransaction |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.PersistTransactionRequest |
| Parameters | • *featureTransaction*: the actual transaction object. Contains a list of features as they where, and a list of features as they should be.<br>• *crs*: the map's coordinate reference system. |
| Description | Persist a single transaction on the server (create, update, delete of features). |
| Response object class | org.geomajas.command.dto.PersistTransactionResponse |
| Response values | • *featureTransaction*: the same transaction that was sent to the server. Unless something went wrong, in which case this could be null. |

### Table 5.8. SearchAttributesCommand

| **SearchAttributesCommand** | |
|---|---|
| Registry key | command.feature.SearchAttributes |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.SearchAttributesRequest |
| Parameters | • *layerId*: the layer to search in.<br>• *attributeName*: the name of the attribute as configured in the feature info. |

| **SearchAttributesCommand** | |
|---|---|
| | • *filter*: a filter, to limit the list of returned features. |
| Description | Search for attribute possible values for a certain attribute. This command is only used for many-to-one and one-to-many relationships, to search for possible values. |
| Response object class | org.geomajas.command.dto.SearchAttributesResponse |
| Response values | • *attributes*: list of attribute values. |

### Table 5.9. SearchByLocationCommand

| **SearchByLocationCommand** | |
|---|---|
| Registry key | command.feature.SearchByLocation |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.SearchByLocationRequest |
| Parameters | • *location*: geometry which should be used for the searching.<br>• *queryType*: specify exactly whether to search, possible values are `QUERY_INTERSECTS`, `QUERY_TOUCHES`, `QUERY_WITHIN` or `QUERY_CONTAINS`.<br>• *ratio*: if queryType is `QUERY_INTERSECTS`, you can additionally specify what percentage of overlap is enough to be included in the search.<br>• *layerIds*: array of layer ids to search in.<br>• *layerFilters*: allow combination of server layer id and a filter to be specified for searching.<br>• *searchType*: determines whether to stop searching once something in found in one of the layers (in order of course), or whether to continue searching, and include matching features from all layers.<br>• *crs*: the map's coordinate reference system. The *location* geometry will also be expressed in this crs.<br>• *buffer*: before any calculation is made, it is possible to have the location geometry expanded by a buffer of this width (in crs space).<br>• *featureIncludes*: indication of which data to include in the feature. Possible values (add to combine): 1=attributes, 2=geometry, 4=style, 8=label. Default value is to include everything. |
| Description | This command allows you to search for features, based on geographic location. |
| Response object class | org.geomajas.command.dto.SearchByLocationResponse |
| Response values | • *featureMap*: map with layer ids as key and a list of features as value. Only layers in which features were found are included in the map. |

### Table 5.10. SearchFeatureCommand

| **SearchFeaturesCommand** | |
|---|---|
| Registry key | command.feature.Search |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.SearchFeatureRequest |
| Parameters | • *layerId*: id of layer in which features need to be searched.<br>• *max*: maximum number of features to allow in the result. 0 means unlimited.<br>• *crs*: crs which needs to be used for the geometry in the retrieved features.<br>• *criteria*: array of criteria which need to be matched when searching. Each criterion contains the attribute name, the operator (options include "like" and "contains") and the value to compare. Note that the value usually needs to be contained in single quotes.<br>• *booleanOperator*: operator which should be used to combine the different criteria when more than one was specified. Should be either "AND" or "OR". Default value is "AND".<br>• *filter*: an additional layer filter which needs to be applied when searching.<br>• *featureIncludes*: indication of which data to include in the feature. Possible values (add to combine): 1=attributes, 2=geometry, 4=style, 8=label. Default value is to include everything. |
| Description | This command allows you to search for features, based criteria which allow matching on feature attributes. You can specify multiple search criteria and a filter. |
| Response object class | org.geomajas.command.dto.SearchFeatureResponse |
| Response values | • *layerId*: id of the layer which contains the features. Equals the layerId parameter from the request.<br>• *features*: array of features which match the search criteria. Any geometry contained in the features uses the request crs. |

### Table 5.11. UserMaximumExtentCommand

| **UserMaximumExtentCommand** | |
|---|---|
| Registry key | command.configuration.UserMaximumExtent |
| Module which provides this command | geomajas-command |
| Request object class | org.geomajas.command.dto.UserMaximumExtentRequest |
| Parameters | • *layerIds*: list of layers to include.<br>• *includeRasterLayers*: true when raster layers should be included. Defaults to false.<br>• *crs* : crs which should be used for the response. |

| **UserMaximumExtentCommand** | |
|---|---|
| Description | Get the bounding box of the visible features across the requested layers (visible area for the raster layers). |
| Response object class | org.geomajas.command.dto.UserMaximumExtentResponse |
| Response values | • *bounds* : bounding box. |

# Chapter 6. Layers

Layers allow access to data which needs to be displayed in a map.

For the existing layers, the details about configuring you map to include that layer are included in the map configuration chapter.

## 1. RasterLayerService

All access to raster layers should be done using the raster layer service. The following methods exist

- `List<RasterTile> getTiles(String layerId, CoordinateReferenceSystem crs, Envelope bounds, double scale) throws GeomajasException` : this method allows you to obtain the list of raster tiles which need to be displayed for the given bounds at the requested scale.

## 2. VectorLayerService

Vector layers and the data contained within are accessible using the vector layer service. You should not try to access the layers directly. This service assures that the security constraints are adhered. Following access methods are available

- `void saveOrUpdate(String layerId, CoordinateReferenceSystem crs, List<InternalFeature> oldFeatures, List<InternalFeature> newFeatures) throws GeomajasException` : allows creating or updating several features. You have to pass both the old features (null or the feature before it was modified) and the new value of the feature. The two are at compared to determine whether to create, update or delete.

- `List<InternalFeature> getFeatures(String layerId, CoordinateReferenceSystem crs, Filter filter, NamedStyleInfo style, int featureIncludes) throws GeomajasException` : read all features from the layer which match the filter. You can specify which aspects of the feature need to be set.

- `List<InternalFeature> getFeatures(String layerId, CoordinateReferenceSystem crs, Filter filter, NamedStyleInfo style, int featureIncludes, int offset, int maxResultSize) throws GeomajasException` : read a batch of features from the layer which match the filter. You can specify which aspects of the feature need to be set.

- `Envelope getBounds(String layerId, CoordinateReferenceSystem crs, Filter filter) throws GeomajasException` : get the bounds of the visible features which match the filter. This can be useful for fit-to-page like functionality.

- `List<Attribute<?>> getAttributes(String layerId, String attributeName, Filter filter) throws GeomajasException` : get the list of possible attribute values.

- `InternalTile getTile(TileMetadata tileMetadata) throws GeomajasException` : get a vector tile.

## 3. VectorLayer

A vector layer can be considered as a collection of homogeneous features. As Geomajas is a POJO-based framework, features are not required to implement a specific interface or extend from a common parent class. This can be seen from the `VectorLayer` interface, which expects plain Java objects as arguments and return objects:

```
@Api(allMethods = true)
@UserImplemented
public interface VectorLayer extends Layer<VectorLayerInfo> {

 boolean isCreateCapable();

 boolean isUpdateCapable();

 boolean isDeleteCapable();

 FeatureModel getFeatureModel();

 Object create(Object feature) throws LayerException;

 Object saveOrUpdate(Object feature) throws LayerException;

 Object read(String featureId) throws LayerException;

 void delete(String featureId) throws LayerException;

 Iterator<?> getElements(Filter filter, int offset, int maxResultSize) throws La

 Envelope getBounds(Filter filter) throws LayerException;

 Envelope getBounds() throws LayerException;
}
```

To achieve this, we apply a form of indirection. The full knowledge of how to access and modify the state of the layer objects is captured in a separate interface, called `FeatureModel`. This class acts as a sort of gateway between the layer and the internal feature model of Geomajas. This limits the layer's responsibility to the actual persistence of the features, which can be as simple as doing nothing (in the case of updating an existing Hibernate object).

# 4. FeatureModel

The conversion between layer-specific feature objects and `InternalFeature` objects is handled by the vector layer's `FeatureModel` implementation. The FeatureModel provides the following contract:

```
@Api(allMethods = true)
@UserImplemented
public interface FeatureModel {

 void setLayerInfo(VectorLayerInfo vectorLayerInfo) throws LayerException;

 Attribute getAttribute(Object feature, String name) throws LayerException;

 Map<String, Attribute> getAttributes(Object feature) throws LayerException;

 String getId(Object feature) throws LayerException;

 Geometry getGeometry(Object feature) throws LayerException;

 void setAttributes(Object feature, java.util.Map<String, Attribute> attributes

 void setGeometry(Object feature, Geometry geometry) throws LayerException;

 Object newInstance() throws LayerException;
```

```
 Object newInstance(String id) throws LayerException;

 int getSrid() throws LayerException;

 String getGeometryAttributeName() throws LayerException;

 boolean canHandle(Object feature);
}
```

It basically acts as a layer object factory and modifier and prepares layer objects to reflect their new state before they are persisted by the layer itself. A `FeatureModel` is free to interpret attribute changes as it likes, although most `FeatureModel` implementations will adhere to certain assumptions (see next paragraph).

# 5. EntityAttributeService

In most cases, the process of transferring attribute information to layer objects will follow a concise set of rules:

- if a primitive attribute is changed to a new value, the new value will replace the old value
- if any attribute is changed to a null value, it will be deleted
- if a many-to-one attribute is changed to an existing attribute, it will be replaced by the existing attribute value and updated with the new state
- if a one-to-many attribute is changed to an empty collection or null value, it will become an empty collection (whether the resulting orphan attributes should be deleted depends on the `FeatureModel` or `VectorLayer` implementation, though)
- if a one-to-many attribute is changed to a new collection of attributes: existing attributes are updated, new attributes are created and missing attributes become orphans (whether they are deleted depends on the `FeatureModel` or `VectorLayer` implementation, though)
- the previous rules are recursively applied

The graph-merging of object trees has been captured in a separate service that can be used by the `FeatureModel`. This service is called `EntityAttributeService` and assumes the the layer objects can be converted to `Entity` instances. The `Entity` interface is a minimal contract that allows navigation and modification of the object tree to apply the above set of rules:

```
@Api(allMethods = true)
@UserImplemented
public interface Entity {

 Object getId(String name) throws LayerException;

 Entity getChild(String name) throws LayerException;

 void setChild(String name, Entity entity) throws LayerException;

 EntityCollection getChildCollection(String name) throws LayerException;

 void setAttribute(String name, Object value) throws LayerException;

 Object getAttribute(String name) throws LayerException;

}
```

Starting with a root entity, one-to-many and many-to-one attributes can be navigated as entities by using the `getChild()` and `getChildCollection()` accessors, respectively. The `Entitycollection` interface provides iteration, creation and deletion of one-to-many attributes.

Examples of how to implement a `FeatureModel` using the `EntityAttributeService` can be found in the `BeanFeatureModel` (server) and `HibernateFeatureModel` (Hibernate layer plugin) implementations.

# Chapter 7. Security

Geomajas has security built-in. If you don't provide a security configuration, nothing will be authorized. For unsecured access, you can use the AllowAllSecurityService security service. Example for the configuration can be found in the security configuration chapter.

which will allow all access to everybody, including full access to features which are only partly within configured bounds.

It is also possible to configure other security services, to allow authentication and authorization to be done by the services which are configured.

### Note

When configuring security services, it is important to assure that login is possible. Anything which is not explicitly allowed is *not* allowed, which likely includes the command which is used to login. You have to make sure that everybody can access the login command.

Specific configuration depends on the configured security services, details of which can be found in the specific plugin's documentation.

# 1. Authentication versus authorization

The security infrastructure makes a clear distinction between authentication and authorization.

Authentication is the act of identifying the user and user the user is how he/she says he is (whether that person is "authentic"). In Geomajas the authentication will result in a authentication token which encapsulated that a user has provided valid credentials. The token in itself does not contain either information about the user or information about what is allowed or authorized (no policies). These can however be accessed using the token.

The Geomajas server core does not do authentication, though it is likely that your security plug-in either provide commands to allow creation of a token (by supplying user credentials) and invalidating the token (logout), or the plug-in will stipulate where this can be done (possibly supplying a redirect to an SSO service or similar).

Authorization on the other hand reads the policies which are in effect to determine what an authenticated user if allowed or disallowed to do and/or access. Geomajas only uses policies which allow access, Everything which is not explicitly allowed is disallowed.

# 2. What can be authorized

Based on the user who is logged into the system, the following restrictions can apply:

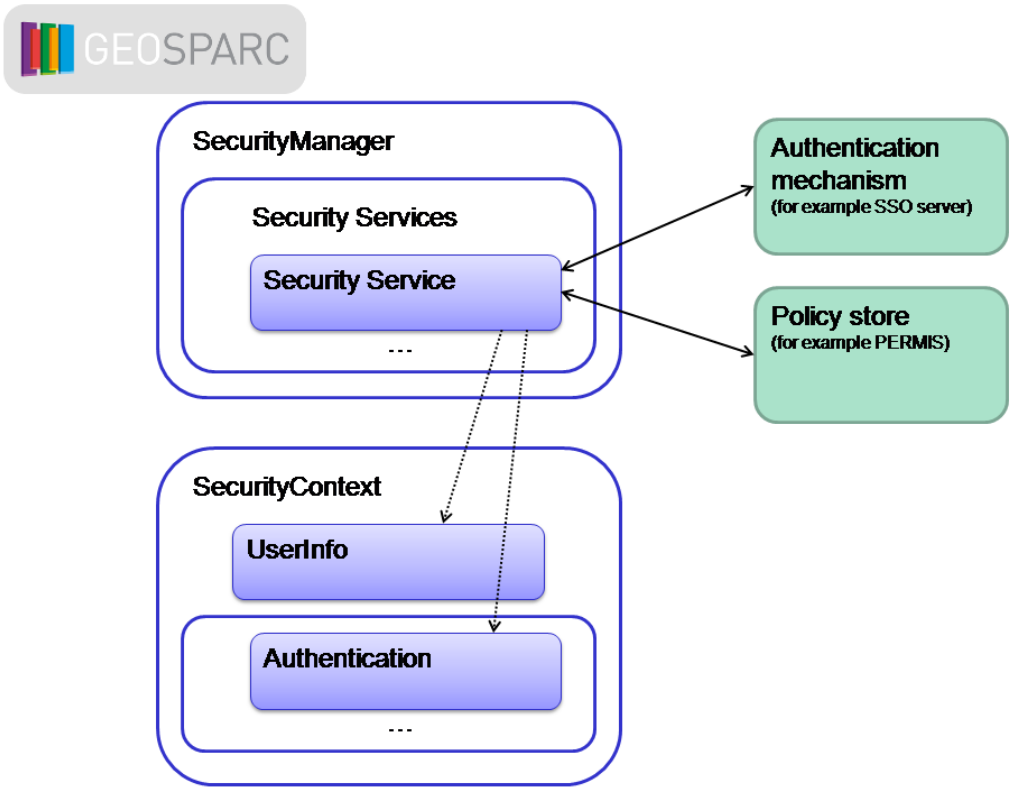
- access rights to a command
- access rights for a layer
- a filter which needs to be applied for a layer
- a region which limits the data which may be accessed for a layer
- access rights on the features
- access rights on the individual attributes of the features

You can extend this by providing additional authentication interfaces which are also implemented by the authentication object returned by your security service. Details can be found in ???.

# 3. SecurityManager service

The security manager manages the (thread-local) security context. It delegates to the available security services to build the authentication objects and get the user information which is then stored in the in the security context. The security services themselves will check with the authentication server or service whether the token is still valid, and will get the policies from a policy server or service to populate the authentication objects with the credentials.

**Figure 7.1. Security architecture**




The SecurityManager service has the following methods:

- `boolean createSecurityContext(String authenticationToken)` : create the security context for this thread, based on the authentication token.
- `void clearSecurityContext()` : clear the security context for this thread.

# 4. SecurityContext service

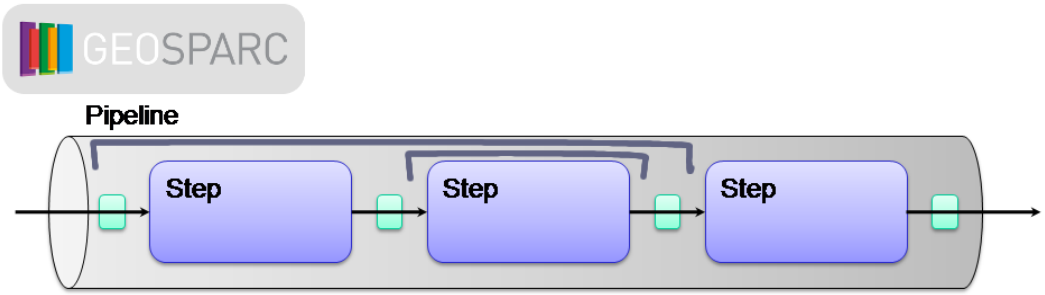
The security context allows access to the currently valid user's policies and some limited information (user id, name and organization). In your code, you just have to inject the security context. The client is responsible for assuring the current thread has the correct security context based on the credentials used when accessing the server (it will use the SecurityManager service to do that).

The security context contains all methods from the UserInfo and Authorization interfaces, plus some methods to get the current token and get the list of authentication objects which have been combined.

# Chapter 8. Pipelines

Pipelines are building blocks which are used in Geomajas to make certain aspects highly extend- and customizable. For more details, see the architecture Section 2, “Pipelines”.

**Figure 8.1. Geomajas pipeline architecture**




# 1. PipelineService

The pipeline service helps you to execute a pipeline. It allows you to fetch a named pipeline which applies for a specific layer (either the layer specific pipeline or the default pipeline). It also has methods to create an empty pipeline context and execute a pipeline.

# 2. Configuration

A pipeline can be defined by specifying the pipeline name and the pipeline steps.

**Example 8.1. Simple pipeline definition**

```
<bean class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName" value="pipelineTest"/>
    <property name="pipeline">
        <list>
            <bean class="org.geomajas.internal.service.pipeline.Step1">
                <property name="id" value="s1"/>
            </bean>
            <bean class="org.geomajas.internal.service.pipeline.Step2">
                <property name="id" value="s2"/>
            </bean>
            <bean class="org.geomajas.internal.service.pipeline.Step3">
                <property name="id" value="s3"/>
            </bean>
        </list>
    </property>
</bean>
```

A pipeline can be layer specific and can refer to a delegate (bean reference). The use of the delegate means that the pipeline definition (list of steps) is copied from the delegate. The following pipeline extends the previous one (the ref value indicates that the pipeline is referenced by bean name/id).

**Example 8.2. Layer specific pipeline which refers to a delegate**

```
<bean id="inter" class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName" value="pipelineTest"/>
```

```
    <property name="layerId" value="inter"/>
    <property name="delegatePipeline" ref="stop" />
</bean>
```

When referring to the pipeline definition using a delegate, the pipeline can also be extended by inserting additional steps at the extension hooks. You can pass a map of "extensions" which are named steps. When a extension hook of the name is found, that step will be included in the pipeline just after the hook definition.

First you have to define the actual extension hooks by adding steps of class `PipelineHook`.

### Example 8.3. Define pipeline extension hooks

```
<bean id="hooked" class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName" value="hookedTest"/>
    <property name="layerId" value="base"/>
    <property name="pipeline">
        <list>
            <bean class="org.geomajas.service.pipeline.PipelineHook">
                <property name="id" value="PreStep1"/>
            </bean>
            <bean class="org.geomajas.internal.service.pipeline.Step1">
                <property name="id" value="s1"/>
            </bean>
            <bean class="org.geomajas.service.pipeline.PipelineHook">
                <property name="id" value="PostStep1"/>
            </bean>
            <bean class="org.geomajas.service.pipeline.PipelineHook">
                <property name="id" value="PreStep2"/>
            </bean>
            <bean class="org.geomajas.internal.service.pipeline.Step2">
                <property name="id" value="s2"/>
            </bean>
            <bean class="org.geomajas.service.pipeline.PipelineHook">
                <property name="id" value="PostStep2"/>
            </bean>
        </list>
    </property>
</bean>
```

The hooks can then be used to add extra steps to the pipeline at the predefined places. Note that you can only add one step per hook in a pipeline definition. If you need to define more, you will have to extend the pipeline more than once.

### Example 8.4. Extending a delegate pipeline

```
<bean id="hooked2" class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName" value="hookedTest"/>
    <property name="layerId" value="delegate"/>
    <property name="delegatePipeline" ref="hooked" />
    <property name="extensions">
        <map>
            <entry key="PreStep2">
                <bean class="org.geomajas.internal.service.pipeline.Step2">
                    <property name="id" value="ps2"/>
                </bean>
            </entry>
        </map>
    </property>
```

```
</bean>
```

Pipelines support the concept of a pipeline interceptor. A pipeline interceptor surrounds a group of steps and allows some action to be performed before the first step of the group and another action after the last step. The group of steps can be skipped depending on the outcome of the first action. The following configuration shows how to add an interceptor to a pipeline:

**Example 8.5. Adding interceptor to delegate pipeline**

```
<bean id="intercept1" class="org.geomajas.service.pipeline.PipelineInfo">
    <property name="pipelineName" value="interceptorTest1"/>
    <property name="layerId" value="base"/>
    <property name="delegatePipeline" ref="intercepted" />
    <property name="interceptors">
        <list>
            <bean class="org.geomajas.internal.service.pipeline.Interceptor">
                <property name="id" value="i1" />
                <property name="fromStepId" value="s1" />
                <property name="toStepId" value="s2" />
            </bean>
        </list>
    </property>
</bean>
```

If the interceptor should be around just one step, you can use the "stepId" property to set the fromStepId and toStepId at once. If you do not define the fromStepId, the pipeline will be intercepted from the start. If you do not define the toStepId the pipeline will be intercepted till the end. Note that you cannot have interceptors which cross each other. For this reason we recommend using interceptor either for individual steps or have them start from the beginning of the pipeline (interceptors which span till the end can often be replaced by a hook which stops pipeline execution).

# 3. Default pipelines

The default pipelines are detailed here. All the steps mentioned here have a hook before and after the step to allow customization of the pipeline. These hooks have the name of the step as mentioned here, with either "pre" or "post" as prefix (note that these keys are case dependent).

## 3.1. RasterLayerService

### 3.1.1. getTiles()

pipeline name "rasterLayer.getTiles", constant PipelineCode.PIPELINE_GET_RASTER_TILES:

- "Get" : get the raster tile.

## 3.2. VectorLayerService

### 3.2.1. saveOrUpdate()

pipeline name "vectorLayer.saveOrUpdate", constant PipelineCode.PIPELINE_SAVE_OR_UPDATE:

- "EqualSize" : verify that the list of old and new features match.
- "SaveOrUpdate" : this handles the save or update for the individual features using the pipeline below.

### 3.2.2. saveOrUpdate each feature

pipeline name "vectorLayer.saveOrUpdateOne", constant PipelineCode.PIPELINE_SAVE_OR_UPDATE_ONE:

- "Delete" : delete the feature if it has been removed.
- "CheckId" : check that the id for the old and new feature match.
- "TransformGeometry" : assure the geometry is transformed to layer coordinate space.
- "Create" : handle the creation of a new feature.
- "Update" : update the feature.
- "UpdateSave" : save it back to the data store.
- "UpdateFeature" : and assure the feature itself reflects the state from the database.

### 3.2.3. getFeatures()

pipeline name "vectorLayer.getFeatures", constant PipelineCode.PIPELINE_GET_FEATURES:

- "LayerFilter" : calculate the correct filter based on security and layer extent.
- "GetFeaturesStyle" : get the styles which are relevant for the features.
- "GetFeatures" : fetch and fill the features.

### 3.2.4. getBounds()

pipeline name "vectorLayer.getBounds", constant PipelineCode.PIPELINE_GET_BOUNDS:

- "LayerFilter" : calculate the correct filter based on security and layer extent.
- "GetBounds" : calculate the bounds for the features which comply with the filter.

### 3.2.5. getAttributes()

pipeline name "vectorLayer.getAttributes", constant PipelineCode.PIPELINE_GET_ATTRIBUTES:

- "LayerFilter" : calculate the correct filter based on security and layer extent.
- "GetAttributes" : get the attributes for the filtered features.

### 3.2.6. getTile()

pipeline name "vectorLayer.getTile", constant PipelineCode.PIPELINE_GET_VECTOR_TILE:

- "TileFilter" : calculate the correct filter based on security and tile extent.
- "GetFeatures" : fetch and fill the features.
- "TileTransform" : transform the tile to the requested coordinate reference system.
- "TileFillStep" : assure features are only rendered in on tile (as needed for SVG and VML rendering).
- "GetStringContent" : render the features to the requested string content.

# Chapter 9. Utility Services

The Geomajas server core also contains a set of utility services.

# 1. ConfigurationService

This service allows you to easily access some of the configuration information.

Provided methods are:

- `VectorLayer getVectorLayer(String id)` : get a vector layer based on the layer id.
- `RasterLayer getRasterLayer(String id)` : get a raster layer based on the layer id.
- `Layer<?> getLayer(String id)` : get a layer (can be either vector or raster), based on the layer id.
- `ClientMapInfo getMap(String mapId, String applicationId)` : get the map with given id for a specific application.
- `void invalidateLayer(String layerId)`: should be called when the configuration of the layer has been changed in a way which may affect the rendering of the layer. It is typically used to invalidate cached. It should for example be used when the layer is deleted or reconfigured, when authorizations for the layer change, etc.
- `void invalidateAllLayers()`: similar to invalidateLayer(), but invalidates all layers. A possible reason to call this is changes in security configuration.

# 2. GeoService

GeoServices provides a set of methods which ease the working with geometries and related objects.

- `CoordinateReferenceSystem getCrs(String crs) throws LayerException` : get the CRS object based on the CRS id (it is advised to use getCrs2() instead of this one).
- `Crs getCrs2(String crs) throws LayerException` : get the CRS object based on the CRS id (preferred, this also contains the Crs id).
- `int getSridFromCrs(String crs)` : attempts to extract the SRID (Spatial Reference Id) from the CRS.
- `int getSridFromCrs(CoordinateReferenceSystem crs)` : attempts to extract the SRID (Spatial Reference Id) from the CRS.
- `String getCodeFromCrs(CoordinateReferenceSystem crs)` : attempts to extract the code (e.g. "EPSG:4326") from the CRS.
- `String getCodeFromCrs(Crs crs)` : get the code (e.g. "EPSG:4326") from the CRS.
- `MathTransform findMathTransform(CoordinateReferenceSystem sourceCrs, CoordinateReferenceSystem targetCrs) throws GeomajasException` : get the transformation which converts between two coordinate systems.
- `CrsTransform getCrsTransform(CoordinateReferenceSystem sourceCrs, CoordinateReferenceSystem targetCrs) throws GeomajasException` : get the transformation which converts between two coordinate systems.
- `CrsTransform getCrsTransform(Crs sourceCrs, Crs targetCrs) throws GeomajasException` : get the transformation which converts between two coordinate systems.

- `CrsTransform getCrsTransform(String sourceCrs, String targetCrs) throws GeomajasException` : get the transformation which converts between two coordinate systems.
- `Geometry transform(Geometry source, CrsTransform transform)` : transform a geometry from source to target CRS.
- `Geometry transform(Geometry source, Crs sourceCrs, Crs targetCrs) throws GeomajasException` : transform a geometry from source to target CRS.
- `Geometry transform(Geometry source, String sourceCrs, String targetCrs) throws GeomajasException` : transform a geometry from source to target CRS.
- `Geometry transform(Envelope source, CrsTransform transform)` : transform an envelope from source to target CRS.
- `Geometry transform(Envelope source, Crs sourceCrs, Crs targetCrs) throws GeomajasException` : transform an envelope from source to target CRS.
- `Geometry transform(Envelope source, String sourceCrs, String targetCrs) throws GeomajasException` : transform an envelope from source to target CRS.
- `Geometry transform(Bbox source, CrsTransform transform)` : transform a bounding box from source to target CRS.
- `Geometry transform(Bbox source, Crs sourceCrs, Crs targetCrs) throws GeomajasException` : transform a bounding box from source to target CRS.
- `Geometry transform(Bbox source, String sourceCrs, String targetCrs) throws GeomajasException` : transform a bounding box from source to target CRS.
- `Coordinate calcDefaultLabelPosition(InternalFeature feature)` : determine a default position for positioning the label for a feature.
- `Geometry createCircle(Point center, double radius, int nrPoints)` : get a geometry which approximates a circle (if only a geometry could contain curves).

# 3. DtoConverterService

This service allows conversion between objects which are used internally (which may contain JTS or GeoTools objects) and data transfer objects which can be used for communication with the outside world (including the clients).

There are two methods which are provided, toInternal() and toDto() and these are overloaded for many different types of objects.

# 4. FilterService

The FilterService allows you to build filters at runtime on the client-side which are applied when requesting vector features. The service is implemented by setting the filter parameter of a vector layer with an ECQL [http://docs.codehaus.org/display/GEOTOOLS/ECQL+Parser+Design] string.

# 5. TextService

Utility functions for calculating text and font related parameters server-side. These parameters could in principle be calculated more accurately on the displaying device itself, but unfortunately there is no support for this in browser environments.

- `Rectangle2D getStringBounds(String text, FontStyleInfo fontStyle)` : get the bounds for the given string.

# 6. ResourceService

This service allows to look up absolute or relative Spring resources in the context of the application. A resource can be found by passing a location string to the service. The default implementation of the service will use the following algorithm to find the resource:

- try to look up the resource by calling the `ApplicationContext.getResource()` method
- try to look up the resource in the classpath. The location is interpreted as a root classpath location (leading `'/'` is optional).
- try to look up the resource in a list of configured root resource locations. The location is interpreted asThe a relative path with respect to the root location.

The ResourceService provides the following methods:

- `Resource find(String location) throws GeomajasException` : lookup method for resources. If a resource is returned it is guaranteed to exist. This method returns null if it cannot find an existing resource.
- `List<String> getRootPaths()` : returns the list of root paths. These paths are location strings that are interpreted by the application context's resource loader. Examples are `'classpath:com/example'` or `'http://www.geomajas.org'` or `'WEB-INF/images'`. A list of root paths can be configured by overriding the bean definition of the ResourceService in your application context:

```
<bean class="org.geomajas.internal.service.ResourceServiceImpl" id="service.Reso
  <property name="rootPaths">
    <list>
      <value>classpath:org/geomajas/internal/service</value>
      <value>WEB-INF/images</value>
    </list>
  </property>
</bean>
```

Additional root paths can also be configured anywhere in the application context by defining a `ResouceInfo` bean:

```
<bean class="org.geomajas.service.resource.ResourceInfo" id="anotherResource">
    <property name="rootPath" value="classpath:org/geomajas/"/>
</bean>
```

# 7. LegendGraphicService

This service allows to obtain a graphical image of each layer style. Such images are typically used to create a map legend, hence the name of the service. The service provides the following method:

- `RenderedImage getLegendGraphic(LegendGraphicMetadata legendMetadata) throws GeomajasException;`

The LegendGraphicMetadata is a metadata object that more or less follows the parameter set used by the SLD-WMS GetLegendGraphic request:

```
public interface LegendGraphicMetadata {
  // server layer id
  String getLayerId();
```

```
  // user defined SLD style
  UserStyleInfo getUserStyle();
  // predefined layer style
  NamedStyleInfo getNamedStyle();
  // user defined SLD rule
  RuleInfo getRule();
  // scale to determine applicable rule
  double getScale();
  // width of the image
  int getWidth();
  // height of the image
  int getHeight();
}
```

The current implementation will return a single image, representing a single rule in SLD. The rule is either passed directly to the service or derived from the predefined styles of the layer (in this case the first rule is taken). A Spring MVC controller `LegendGraphicController` is available to access the service from a REST-like URL:

```
[DISPATCHER_URL]/legendgraphic/{layerId}/{styleName}/{ruleIndex}.{format}
```

In this case the style name (as predefined on the server) and the index of the rule can be passed, as well as the final image format.

# 8. FeatureExpressionService

This service evaluates feature expressions for internal feature objects. Expressions are textual and expressed in a language that depends on the specific implementation of this service.

The default implementation uses the Spring Expression Language (SpEL) [http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/expressions.html], but other implementations may be based on eg. Freemarker/Velocity templates or even a custom language. As a minimal requirement, the language should be able to evaluate simple references to primitive attribute values.

The service provides the following methods:

- `Object evaluate(String expression, InternalFeature feature) throws LayerException`: evaluates the expression for the feature given, evaluation result can be any object type
- `void setVariables(Map<String, Object> variables)`: allows to insert context variables. These are available as variables that can be used in expressions (in the default implementation this is done by prepending the name with a hash tag (#)

Examples of feature expressions for a feature with attributes name = `"Belgium"` and capital = `{name: "Brussels"}` (many-to-one attribute)

### Table 9.1. Feature expression examples

| Expression | Evaluated to: | Comment |
|---|---|---|
| name | Belgium | attributes can be directly referred by name |
| name + ' ( ' + capital.name + ' )' | Belgium ( Brussels ) | concatenation (+), attribute navigation (.), string constants (using single quotes !) |
| #reverser.reverse(name) | miugleB | variables referenced by hash tag (#) |

In the last case, an object with name `reverser` was added to the service with a method `reverse()` that performs the string reversal:

```
<bean class="org.geomajas.internal.service.FeatureExpressionServiceImpl" id="se
 <property name="variables">
  <map>
   <entry key="reverser">
     <bean class="com.myproject.myReverserImpl" scope="thread">
   </entry>
  </map>
 </property>
</bean>
```

### Warning

Custom variables that are added to the service should be made thread-safe (use thread scope if in doubt) !

# 9. DispatcherUrlService

This service gives access to the dispatcher URL, which is the base URL that maps to DispatcherServlet of the Geomajas application. This allows building absolute URLs to services linked in as dispatched controllers (Spring MVC controllers). The absolute URLs should be usable in web clients that access the Geomajas application and may therefore depend on the client or even the individual web request. This is especially relevant for proxied applications, for which the absolute URL is not directly available to the web application but can usually be derived from specific headers of the web request (or configured if that is not the case).

The service provides the following methods:

- `String getDispatcherUrl()`: returns the external base URL of the DispatcherServlet (for use by the web client)
- `String getLocalDispatcherUrl()`: returns the local base URL of the DispatcherServlet (for use by the web application)
- `String localize(String externalUrl)`: converts an external URL that points to the DispatcherServlet to a local URL (implemented as simple string replacement of base URL in most cases)

There are 2 implementations available of this service:

- `org.geomajas.service.impl.StaticUrlDispatcherService`: allows static configuration of both local and external base URL
- `org.geomajas.servlet.AutomaticDispatcherUrlService`: allows dynamic calculation of both local and external base URL (based on the servlet request). The local base URL can also be configured statically.

# Part IV. Configuration

# Table of Contents

# Chapter 10. Configuration basics

Geomajas leverages the Spring framework for configuration. The initial configuration needs to be done using web.xml. There you need to indicate the files which contain the configuration information.

## 1. web.xml

In your `web.xml` file, you need to assure the configuration is made available to the application, and you can indicate which files are used to contain the configuration. Though it is possible to put all configuration information in one file, we recommend splitting your configuration in several files (see Section 3, “Recommended application context structure”).

The listener class initialises the application context as needed for Geomajas. You have to specify the files which contain the application context in the `contextConfigLocation` context parameter. You have to add the Geomajas context file as first item in the list. Each entry can start with a location prefix. When no location prefix is specified, the file is searched in the web context. You can also use location prefixes as defined by Spring, e.g. "classpath:" or "classpath*:". Note that whitespace is used as separator which means that the path itself should not contain spaces. It is possible to use wildcards (e.g. "WEB-INF/*.xml").

These are defined using an excerpt like the following:

### Example 10.1. Defining spring configuration locations in web.xml

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:org/geomajas/spring/geomajasContext.xml  1
    WEB-INF/applicationContext.xml  2
  </param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.
            ContextLoaderListener</listener-class>  3
</listener>
<listener>
  <listener-class>org.springframework.web.context.request.
            RequestContextListener</listener-class>  4
</listener>
.....
```

1 root context for Geomajas
2 additional context for your application
3 assures the application context is available
4 assures the request can be accessed

You also need to define at least the dispatcher servlet and possibly an additional servlet for your client. The dispatcher servlet can be defined as follows.

### Example 10.2. Dispatcher servlet declaration in web.xml

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-c
    <init-param>
        <param-name>contextConfigLocation</param-name>
```

```
        <param-value>classpath:org/geomajas/spring/geomajasWebContext.xml</para
        <description>Spring Web-MVC specific (additional) context files.</descr
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/d/*</url-pattern>
</servlet-mapping>
```

Another option you have in setting up the web.xml file, is a specially designed filter that improves caching behaviour and compresses some when sending them to the browser. The default configuration of this filter is tuned to be used in combination with the GWT client. All files containing ".nocache." in their name will not be cached, while all files containing ".cache." in their name will be cached. It will cache all graphics files, css, html and js files. The JavaScript, HTML and CSS files will also be gzip compressed if the client supports it. The caching will is not enable for requests to localhost, and all handling is disabled for paths which are handled by the dispatcher servlet.

To activate this filter (highly recommended!) add the following to the web.xml:

### Example 10.3. Cache filter declaration in web.xml

```
<filter>
    <filter-name>CacheFilter</filter-name>
    <filter-class>org.geomajas.servlet.CacheFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>CacheFilter</filter-name>
    <url-pattern>*</url-pattern>
</filter-mapping>
```

It is also possible to configure all aspects of this filter. The full (default) configuration is like this:

### Example 10.4. Full cache filter declaration in web.xml

```
<filter>
  <filter-name>CacheFilter</filter-name>
  <filter-class>org.geomajas.servlet.CacheFilter</filter-class>
  <init-param>
    <description>Time that cache stuff should be cached, defaults to 1 year.</d
    <param-name>cacheDurationInSeconds</param-name>
    <param-value>31536000</param-value>
  </init-param>
  <init-param>
    <description>All uris which start with one of these prefixes remain untouch
    <param-name>skipPrefixes</param-name>
    <param-value>/d/</param-value>
  </init-param>
  <init-param>
    <description>When the uri contains one of these, the cache headers are adde
    <param-name>cacheIdentifiers</param-name>
    <param-value>.cache.</param-value>
  </init-param>
  <init-param>
    <description>When the uri ends in one of these, the cache headers are added
    <param-name>cacheSuffixes</param-name>
    <param-value>.js .png .jpg .jpeg .gif .css .html</param-value>
```

```
  </init-param>
  <init-param>
    <description>When the uri contains one of these, the cache headers are remov
    <param-name>noCacheIdentifiers</param-name>
    <param-value>.nocache.</param-value>
  </init-param>
  <init-param>
    <description>When the uri end in one of these, the cache headers are removed
    <param-name>noCacheSuffixes</param-name>
    <param-value></param-value>
  </init-param>
  <init-param>
    <description>When the uri ends in one of these, the response is gzip compres
    <param-name>zipSuffixes</param-name>
    <param-value>.js .css .html</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>CacheFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

# 2. General principles

Each configuration file needs the following header:

### Example 10.5. Spring configuration preamble

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

This defines the most common schemas which are needed. The configuration is built by populating the configuration classes. The configuration classes are split up between client-side and server. Only the server classes are necessary to configure the server, which behaves as a catalog of layers. The client side classes are used to define applications and maps, which are purely client-side concepts in the Geomajas architecture.

The server classes have a class name ending in "Info" and are mostly found in the `org.geomajas.configuration` package. These classes are actually used to represent the DTO part of the server layers, thereby allowing to transfer information or metadata of these layers to the client.

Configuration is done using the Spring Framework. We will give some notions here, but for a full introduction to Spring, please read the reference documentation http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/.

Each configuration file can contain one or more bean definitions, which correspond to actual Java bean instances. You can set all the properties of the objects using this configuration file. Primitive types can be set directly using a string representation of the value. When the value is another bean, then it can either be defined in-line, or you can use a reference. You can choose whether the referenced bean is defined in the same file or a different one. As long as the bean name is unique, and the location is added in the `contextConfigLocation` context parameter in the web.xml file, the reference is resolved.

It is possible to define a bean with the same name (or id) more than once. In that case, the last occurrence will be used.

# 3. Recommended application context structure

Geomajas requires some configuration, especially of the maps and layers, and some additional configuration for security, plug-ins etc. This is typically done using the applicationContext.xml file. There can be quite a lot of configuration, pieces can be split off to make that file less overwhelming. You should still be able to know where to find something without inspecting file contents (this particularly means that a bean which is referenced from more than one file needs a predictable file for finding that bean). We recommend using the following scheme.

The context files are normally put in the WEB-INF directory in the web context (src/main/webapp/WEB-INF in your maven project).

*Application/map configuration:* the configuration can be split at several levels. You can always put stuff in the parent (# indication is used for cases where this will probably be the default)[1].

- application info: appXxx.xml (# typically inside the applicationContext.xml is there is only one application).
- map info: mapYyy.xml (#)
- client layer info: clientLayerZzz.xml (#)
- server layer info: layerUuu.xml

The main object of a file should have the same name or id as the filename. For example the client layer "clientLayerRivers" would be in the file "clientLayerRivers.xml" (if it is in a separate file).

General configuration (security, pipelines, tools for toolbars etc) will be in applicationContext.xml. Extra files with clear names can be created to store configuration for specific plug-ins. For example, when extensive security configuration is needed, there may be a separate security.xml file.

[1] In many configurations, only the applicationContext.xml and server-side layer files will exist.

# Chapter 11. Layer configuration

The central configuration which needs to be done is the map and the collection of layers which are part of that map.

# 1. Raster layer configuration

Raster layers are image-based layers which, depending on the type, may be configured to retrieve their images from WMS, Google Maps or OpenStreetMap (tile) servers. All raster layer implementations implement the `org.geomajas.layer.RasterLayer` interface, which means they provide an accessor for a `RasterLayerInfo` metadata object. The info object configuration is normally defined in the Spring configuration as part of the entire layer configuration. Depending on the type of layer, extra properties are needed to provide a full configuration.

## 1.1. Raster layer info

For all raster layers, you will need to define a raster layer info object to define the server configuration for the layer. The exact meaning for some of the fields depend on the actual layer, but most important features include:

**Table 11.1. Raster Layer info**

| Name | Description |
|---|---|
| dataSourceName | The name of the data source as used by the layer. |
| crs | The coordinate reference system, expressed as "EPSG:<srid>". Caveat: make sure this is the same as the maps' CRS as full raster image reprojection is not supported! If the CRS is not the same, an attempt will be done to rescale and align the center coordinates, though. |
| maxExtent | The bounds of the layer, specified in layer coordinates. After transformation to map coordinates, this determines the locations and absolute size of the tiles. |
| zoomLevels | A list of scale values corresponding to the zoom levels at which the raster data should be fetched.<br><br>An image or tile scale is obtained by dividing the size of the tile in pixels by the size of the tile in map units. For example, if the tile is 256 x 256 pixels and this corresponds to an area of 100 m x 100 m, the scale can be calculated as 256/100 = 2,56 pixels per meter. The inverse value of the scale is more often used and is sometimes called the*resolution*. Images are usually optimized or prerendered for a specific (set of) resolution(s), so it is important to specify these here if they are known. On top of that, some servers provide specific tile caching for these predefined resolutions (for example WMS-T).<br><br>A word of caution concerning zoom levels : setting the zoom levels here will only make sure that tiles will be fetched at predefined levels but does not impose any restrictions on the zoom levels of the map itself. If the zoom levels of the map have different values or are not specified at all (arbitrary zooming), raster images will be stretched on the client side to accommodate for these differences. |
| tileWidth | Width in pixels of the requested images. |
| tileHeight | Height in pixels of the requested images. |

The location of the images or tiles is defined by calculating the real width and height (based on the resolution) and "paving" the maximum extent with tiles starting at the origin (x,y) of the extent. If no resolutions are predefined, the tiles are calculated by dividing the maximum extent by successive powers of 2. Make sure the width/height ratio of the maximum extent corresponds to the width/height ratio of the tile.

# 2. Vector layer configuration

Vector layers contain homogeneous vector based features. All vector layer implementations implement the `org.geomajas.layer.VectorLayer` interface, which means they provide an accessor for a `VectorLayerInfo` metadata object. The info object configuration is normally defined in the Spring configuration as part of the entire layer configuration. Depending on the type of layer, extra properties are needed to provide a full configuration.

The definition of the actual layer is similar to the definition of a raster layer.

## 2.1. Vector layer info

For the layer configuration, you have to create the layer info object.

### Example 11.1. Style info

```
<bean name="layerAirportsInfo" class="org.geomajas.configuration.VectorLayerInf
    <property name="layerType" value="POINT" />
    <property name="crs" value="EPSG:4326" />
    <property name="maxExtent">
        <bean class="org.geomajas.geometry.Bbox">
            <property name="x" value="-87.4" />
            <property name="y" value="24.3" />
            <property name="width" value="8.8" />
            <property name="height" value="6.4" />
        </bean>
    </property>
    <property name="featureInfo" ref="layerAirportsFeatureInfo" />
    <property name="namedStyleInfos">
        <list>
            <ref bean="layerAirportsStyleInfo" />
        </list>
    </property>
</bean>
```

This defines the details common to both raster and vector layers, like layer id, crs, layer type, max extent (bounding box) etc.

The following table describes the properties of the `VectorLayerInfo` object:

### Table 11.2. VectorLayer info

| Property | Description |
|---|---|
| layerType | This property determines the type of the default geometry of the features. The following types are supported: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING and MULTIPOLYGON |
| crs | The coordinate reference system, expressed as "EPSG:<srid>". This is probably determined by the layer, but has to be specified anyhow as we have no auto detection in place yet. |

| Property | Description |
|---|---|
| maxExtent | The bounds of the layer, specified in layer coordinates. After transformation to map coordinates, this determines the locations and absolute size of the tiles. |
| featureInfo | The feature metadata |
| namedStyleInfos | The list of predefined style metadata objects which define the named styles for this layer |

The feature metadata can be found in the `FeatureInfo` object. This objects contains the complete feature type description (id, attributes and geometry) as well as the validation rules for the attributes. An example definition of this object is given below:

### Example 11.2. Feature info

```
<bean class="org.geomajas.configuration.FeatureInfo" name="layerAirportsFeature
    <property name="dataSourceName" value="airprtx020" />
    <property name="identifier">
        <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
            <property name="label" value="Id" />
            <property name="name" value="ID" />
            <property name="type" value="LONG" />
        </bean>
    </property>
    <property name="geometryType">
        <bean class="org.geomajas.configuration.GeometryAttributeInfo">
            <property name="name" value="the_geom" />
            <property name="editable" value="true" />
        </bean>
    </property>
    <property name="attributes">
        <list>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="Name" />
                <property name="name" value="NAME" />
                <property name="editable" value="true" />
                <property name="identifying" value="true" />
                <property name="type" value="STRING" />
            </bean>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="County" />
                <property name="name" value="COUNTY" />
                <property name="editable" value="true" />
                <property name="identifying" value="false" />
                <property name="type" value="STRING" />
            </bean>
        </list>
    </property>
</bean>
```

## Note

Since 1.9.0, it is possible to use the following 2 properties in the attribute definitions:

- *hidden*: true/false. By default this value is false, but it can be set to 'true' to hide it from all client side widgets and/or views. This way such an attribute can still be used on the client for hidden calculations or filters, without users actually seeing it.

- *formInputType*: A text value to indicate that this attribute should be represented in forms by the type represented by the text value. This feature is used in the GWT client from version 1.9.0 and higher when using `FeatureForms`. This way it is possible to use custom form items in the forms used for editing the attribute values.

  See the GWT client documentation for more information on this.

The following table describes the properties of the `FeatureInfo` object:

### Table 11.3. Feature info configuration

| Name | Description |
|---|---|
| dataSourceName | This name is used by the layer to internally reference the source that provides the data. Depending on the type of layer, this could be a table name (geotools-postgis), a shape file name (geotools-shapeinmem, in this case there is a 1-to-1 correspondence with the geotools datastore), a WFS layer name (geotools-wfs) or a java class name (hibernate). |
| identifier | Metadata of the primitive attribute that provides a unique identification of the feature. |
| geometryType | Metadata of the geometrical attribute that provides the default geometry of the feature. |
| attributes | Metadata of all other attributes |

This defines the identifier, geometry object and attributes for the feature.

Attributes can be either primitive attributes or association attributes. Primitive attributes represent primitive Java types as well as some common types like Date and String. The following primitive attribute types are defined: BOOLEAN, SHORT, INTEGER, LONG, FLOAT, DOUBLE, CURRENCY, STRING, DATE, URL and IMGURL. Association attributes represent non-primitive Java types. There are two types of association attributes defined: MANY_TO_ONE and ONE_TO_MANY. These reflect the many-to-one and one-to-many relationships as defined in an entity-relationship model and can only be used in conjunction with the `HibernateLayer`.

Last but not least, you can define one or more named style definitions which should be used for rendering of the layer. The actual style that is being used by the client is determined in the client configuration, but you predefine a number of styles (of type `NamedStyleInfo`) here for later reference in the client configuration. This is similar to a regular WMS server, where you can also define a set of predefined styles for each feature type.

Geomajas provides two ways to configure styles:

- simple configuration based on `FeatureStyleInfo` objects: this configuration is limited to simple stroke and fill styles and a limited number of point symbols (rectangle, circle, image). It uses a single filter for determining the applicable style.

- OGC standards-based SLD [http://docs.geoserver.org/latest/en/user/styling/index.html] (Styled Layer Descriptor, version 1.0) configuration: this configuration uses an SLD file for styling and supports all SLD features that are supported by the underlying geotools library. The file can be directly referenced in the configuration, but the server internally makes use of our own DTO objects. This is needed to allow client-side manipulation of the SLD style and communication across the wire of complete SLD descriptors or SLD excerpts like rules. The geomajas SLD project has been created as a Geomajas independent library for this purpose.

Each style object is itself composed of a number of feature styles (`FeatureStyleinfo`) and a label style (`LabelStyleInfo`). You can define formulas to determine which feature style should be used. Formulas are defined as ECQL [http://docs.codehaus.org/display/GEOTOOLS/ECQL+Parser

+Design] strings that are parsed to OpenGIS Filter Objects [http://geoapi.sourceforge.net/2.0/javadoc/org/opengis/filter/package-summary.html] The first style whose formula passes will be applied for the feature. Note that when applying filters to a style an 'other filter' should be defined to prevent null pointer exceptions for features that are not captured by the filter. The following table describes a subset of the ECQL filter types:

### Table 11.4. OGC ECQL Filter Types

| Type | Operators | Example |
|---|---|---|
| Comparison | =, <, > | (ATTRIBUTE = 'GEORGE') (ATTRIBUTE > 10 AND ATTRIBUTE < 20) |
| Text | LIKE, NOT LIKE | (ATTRIBUTE LIKE 'SAMUEL') (ATTRIBUTE NOT LIKE 'TOM %') |
| Null | IS NULL, IS NOT NULL | (ATTRIBUTE IS NULL) (ATTRIBUTE IS NOT NULL) |
| Exists | EXISTS, DOES-NOT-EXIST | (ATTRIBUTE EXISTS) (ATTRIBUTE DOES-NOT-EXIST) |
| Between | BETWEEN | (ATTRIBUTE BETWEEN 10 AND 20) |

An example definition of this object is below:

### Example 11.3. Style info

```
<bean class="org.geomajas.configuration.NamedStyleInfo" name="layerAirportsStyl
    <property name="featureStyles">
        <list>
            <bean class="org.geomajas.configuration.FeatureStyleInfo">
                <property name="name" value="Airports (Florida)" />
                <property name="fillColor" value="#FF3333" />
                <property name="fillOpacity" value=".7" />
                <property name="strokeColor" value="#333333" />
                <property name="strokeOpacity" value="1" />
                <property name="strokeWidth" value="1" />
                <property name="symbol">
                    <bean class="org.geomajas.configuration.SymbolInfo">
                        <property name="rect">
                            <bean class="org.geomajas.configuration.RectInfo">
                                <property name="w" value="12" />
                                <property name="h" value="12" />
                            </bean>
                        </property>
                    </bean>
                </property>
            </bean>
        </list>
    </property>
    <property name="labelStyle">
        <bean class="org.geomajas.configuration.LabelStyleInfo">
            <property name="labelAttributeName" value="NAME" />
            <property name="fontStyle">
                <bean class="org.geomajas.configuration.FontStyleInfo">
                    <property name="color" value="#FEFEFE" />
                    <property name="opacity" value="1" />
                </bean>
            </property>
```

```
            <property name="backgroundStyle">
                <bean class="org.geomajas.configuration.FeatureStyleInfo">
                    <property name="fillColor" value="#888888" />
                    <property name="fillOpacity" value=".8" />
                    <property name="strokeColor" value="#CC0000" />
                    <property name="strokeOpacity" value=".7" />
                    <property name="strokeWidth" value="1" />
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

The LabelStyleInfo configuration declares a label value, a font style and a background style. This configuration is used to render the labels in the default tile rendering pipeline of the vector layer service. The result is a rectangular label with the specified font and background color, expressed in SVG or VML. If the style is defined in SLD, a reasonable attempt is made to convert the TextSymbolizer of the SLD file to a LabelStyleInfo instance (extracting the halo color as background color and using the same font style).

The label value can be specified in two ways in the `LabelStyleInfo` object:

- `labelAttributeName`: this is the name of the feature attribute value that should be directly used as the label text
- `labelValueExpression`: this a textual expression that may contain by-name references of feature attribute values as well as any other constructs allowed by the `FeatureExpressionService` of the current application context.

## 2.1.1. Validation

Most feature attributes should be validated before they can be saved to a file or database. Validation is a concern that stretches across many layers of a typical application: there is usually a need for client-side validation (making the application more user friendly) , server-side validation (to protect the server from invalid data) as well as database validation (to preserve data integrity). Preferably validation rules should be defined as much as possible in a single place to avoid conflicts and duplication.

Our attribute configuration supports several types of validation by defining a `"validator"` property inside the attribute:

### Example 11.4. Attribute validator configuration

```
<property name="validator">
  <bean class="org.geomajas.configuration.validation.ValidatorInfo">
    <property name="toolTip"
      value="Is this city a capital city or not? (Y or N)" />
    <property name="errorMessage"
      value="Invalid value: The value must be either Y or N." />
    <property name="constraints">
      <list>
        <bean class="org.geomajas.configuration.validation.NotNullConstraintInf
        <bean class="org.geomajas.configuration.validation.PatternConstraintInf
          <property name="regexp" value="[YN]$" />
        </bean>
      </list>
    </property>
  </bean>
</property>
```

This property contains some general validator information and a set of constraints that should be applied to the attribute. The available constraint types have been based on the new JavaBeans standard: JSR-303.

## 2.2. Bean layer configuration

Bean layer provides an in-memory layer which is not persisted in any way. The features can be defined in the configuration file using some specialised beans. It is particularly useful for testing. A BeanLayer object (a kind of VectorLayer) must be defined. An example for defining two polygons:

```
<bean name="beans" class="org.geomajas.layer.bean.BeanLayer">
  <property name="layerInfo" ref="beansInfo" />
  <property name="features">
    <list>
        <bean class="org.geomajas.layer.bean.FeatureBean">
        <property name="id" value="1" />
        <property name="stringAttr" value="bean1" />
        <property name="geometry"
          value="POLYGON(((0 0,1 0,1 1,0 1,0 0)))" />
      </bean>
      <bean class="org.geomajas.layer.bean.FeatureBean">
        <property name="id" value="2" />
        <property name="stringAttr" value="bean2" />
        <property name="geometry"
          value="POLYGON(((4 0,6 0,6 3,4 3,4 0)))" />
      </bean>
    </list>
  </property>
</bean>
```

The following table describes the properties of the `BeanLayer` object:

**Table 11.5. BeanLayer configuration**

| Name | Description |
|---|---|
| layerInfo | A VectorLayerInfo instance, comparable to the layerInfo of a VectorLayer, see above. |
| features | List of features, which should be `org.geomajas.layer.bean.FeatureBean` instances. |

# Chapter 12. Security configuration

To make sure the system can be used, you have to configure the security to allow access. The easiest configuration is to allow access to everybody.

### Example 12.1. Allow full access to everybody

```
<bean name="security.securityInfo" class="org.geomajas.security.SecurityInfo">
    <property name="loopAllServices" value="false"/>
    <property name="securityServices">
        <list>
            <bean class="org.geomajas.security.allowall.AllowAllSecurityService
        </list>
    </property>
</bean>
```

Any other configuration would depend on the available security services. For example, when using the staticsecurity plugin, the following could be defined.

### Example 12.2. Partial staticsecurity configuration

```
    <bean name="SecurityService"
          class="org.geomajas.plugin.staticsecurity.security.StaticSecurityServi

    <bean name="security.securityInfo"
          class="org.geomajas.security.SecurityInfo">
        <property name="loopAllServices" value="true"/>
        <property name="securityServices">
            <list>
                <ref bean="SecurityService"/>
                <bean class="org.geomajas.plugin.staticsecurity.security.LoginA
            </list>
        </property>
    </bean>

    <bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServic
        <property name="users">
            <list>

                <!-- User elvis has restricted attribute editing permissions on
                <bean class="org.geomajas.plugin.staticsecurity.configuration.Us
                    <property name="userId" value="elvis"/>
                    <property name="password" value="BUOMyQ95onvc7gMrMjFtDQ"/> <
                    <property name="userName" value="Elvis Presley"/>
                    <property name="authorizations">
                        <list>
                            <bean class="org.geomajas.plugin.staticsecurity.con
                                <property name="commandsInclude">
                                    <list>
                                        <value>.*</value>
                                    </list>
                                </property>
                                <property name="visibleLayersInclude">
                                    <list>
                                        <value>.*</value>
                                    </list>
                                </property>
```

```
<property name="updateAuthorizedLayersInclude">
    <list>
        <value>layerBeans</value>
        <value>layerBeansEditableGrid</value>
    </list>
</property>
```

Most notable in this example is the inclusion of two security services. The second is provided to allow login and logout (*only*) for everybody. The first defines users and authorizations (only the beginning of the configuration is displayed here).

# Chapter 13. Transaction configuration

Spring has support declarative transaction management, which relieves us from the burden of writing our own transaction demarcation and exception handling code. Of course, Spring transaction management has to be hooked up with the transaction definition and life cycle of the underlying data platform (hibernate, JTA, JDBC) . Each data access technology should provide its own implementation of the Spring class `PlatformTransactionManager`. You should check your plug-in documentation for details about configuring the transaction manager.

Transaction management is typically only needed for editable database layers (although we support and encourage it for read-only layers as well). There is currently no support for having multiple platform transaction managers, although configurations with multiple transaction managers should be possible. This will be investigated and fixed in the future. In practice this means that you currently must not mix editable layers which require a different transaction manager.

# Chapter 14. Dispatcher servlet configuration

Additional servlet configuration may be needed for any plug-in that wants to support its own client-server communication protocol. This is typically the case for clients, but in general any plug-in that needs a form of communication that does not match the default command structure should be able to add its own endpoint to the dispatcher servlet. Fortunately, Spring MVC has a very simple architecture to accomplish this. In general, a single MVC dispatcher branch consists of three elements:

- A handler mapping, whose function it is to map servlet requests to handlers (based on the URL pattern)
- A handler or controller, whose function it is to handle the actual request and - in most cases - decide which of the views will handle the response
- A view, whose function it is to prepare the response data and send them to the client

Our default geomajasWebContext.xml configuration in geomajas-common-servlet looks as follows:

```
<beans ...>
    <!-- we use the default BeanNameUrlHandlerMapping
        for mapping to controllers -->
    <bean id="defaultHandlerMapping"
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"
        <!-- need security  -->
        <property name="interceptors">
            <list>
                <ref bean="securityInterceptor" />
            </list>
        </property>
    </bean>

    <bean id="securityInterceptor"
       class="org.geomajas.servlet.mvc.SecurityInterceptor"></bean>

    <!--  we need a view resolver -->
    <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

    <context:component-scan base-package="org.geomajas.servlet"/>
</beans>
```

It contains a default handler mapping which maps URLs to controller beans based on the name of the bean. This means that the controller's name should actually be the part of the URL that follows the dispatcher servlet's base path (including wild cards if more than one url has to be mapped).

The interceptor property is added to make sure that a secure context is set up when accessing the Geomajas server. The interceptor assumes that a parameter `userToken` will be passed as part of the HTTP request. The value of the parameter should be equal to the user token received from the authentication service.

The bean name view resolver kicks in when the controller returns a string value or sets the view name in the `ModelAndView` object (we will come to this later). It will invoke the correct view based on the bean name specified by the controller.

With the current setup all the wiring between URLs, controllers and views can be done via annotations. Assume the base dispatcher URL is `http://localhost:8080/geomajas/d` and we want to set up a specific end point for all URLs with follow the pattern `http://localhost:8080/`

`geomajas/d/mymodule/**`. It is than sufficient to create a controller component with the name `/mymodule/**` and return the name of the view bean (which itself can be a component) in the controller method:

```
@Controller("/mymodule/**")
public class MyController {
    @RequestMapping(value = "/mymodule/test.html", method = RequestMethod.GET)
    public String doMyStuff(@RequestParam("test") String test, Model model){
        return "MyView";
    }
}
```

Notice that apart from the annotations there is nothing special about this class. Spring MVC auto detects the mapping based on the `@RequestMapping` annotation (which in this case narrows down the URL to a specific one) and will even map request parameters to method arguments if they are annotated with `@RequestParam`. The model argument is basically just a hash map to store the result of the operation as needed by the view. There are actually many more advanced possibilities, for which you may want to consult the Spring documentation. If the method returns a string like above, this string will be used to determine the view object, which could be the following bean:

```
@Component("MyView")
public class MyView extends AbstractView {
    @Override
    protected void renderMergedOutputModel(Map<String, Object> model, HttpServl
        HttpServletResponse response) throws Exception {
        // write response using the model
    }
}
```

Views are generally responsible for encoding the result in a specified format (e.g. JSON, XML,...). The result itself can be retrieved from the model argument, which will have the same contents as the model argument in the controller.

# Chapter 15. Coordinate Reference Systems

Geomajas uses GeoTools' gt-epsg-wkt module to define the coordinate reference systems which are available.

If you want to add extra coordinate reference systems, this can be done by defining them in the configuration. For example, Geomajas itself already defines the "EPSG:900913" crs (which one of the many codes for the Mercator projection used by Google Maps and OpenStreetMap).

### Example 15.1. Custom CRS addition

```
<bean class="org.geomajas.global.CrsInfo">
    <property name="key" value="EPSG:900913" />
    <property name="crsWkt">
        <value>
            PROJCS["Google Mercator",
            GEOGCS["WGS 84",
            DATUM["World Geodetic System 1984",
            SPHEROID["WGS 84", 6378137.0, 298.257223563, AUTHORITY["EPSG","7030
            AUTHORITY["EPSG","6326"]],
            PRIMEM["Greenwich", 0.0, AUTHORITY["EPSG","8901"]],
            UNIT["degree", 0.017453292519943295],
            AXIS["Geodetic latitude", NORTH],
            AXIS["Geodetic longitude", EAST],
            AUTHORITY["EPSG","4326"]],
            PROJECTION["Mercator (1SP)", AUTHORITY["EPSG","9804"]],
            PARAMETER["semi_major", 6378137.0],
            PARAMETER["semi_minor", 6378137.0],
            PARAMETER["latitude_of_origin", 0.0],
            PARAMETER["central_meridian", 0.0],
            PARAMETER["scale_factor", 1.0],
            PARAMETER["false_easting", 0.0],
            PARAMETER["false_northing", 0.0],
            UNIT["m", 1.0],
            AXIS["Easting", EAST],
            AXIS["Northing", NORTH],
            AUTHORITY["EPSG","900913"]]
        </value>
    </property>
</bean>
```

You can add as many of these beans as needed. The keys transformation which are added this way are tested before the GeoTools library, so you can overwrite definitions if needed.

If you don't like the dependency on the gt-epsg-wkt library, then you could exclude this dependency in your maven pom and use a different dependency if needed.

The transformations between coordinate reference systems should be done using `GeoService`. When this is used for transformations, it avoids throwing transformation exceptions by limiting the geometry to the transformable area[1]. The system tries to determine the transformable area automatically, but this is is not always possible and if it is, it can be inaccurate. Therefor, you can also configure the transformable area for a pair of CRSs. You can see an example configuration below.

[1] If an exception does occur, it will be logged as a warning, but the GeoService assures the transform does not fail. Instead, and empty geometry will be returned.

**Example 15.2. Custom CRS transformation addition**

```
<bean class="org.geomajas.global.CrsTransformInfo">
    <property name="source" value="EPSG:4326" />
    <property name="target" value="EPSG:900913" />
    <property name="transformableArea">
        <bean class="org.geomajas.geometry.Bbox">
            <property name="x" value="-180" />
            <property name="y" value="-86" />
            <property name="width" value="360" />
            <property name="height" value="172" />
        </bean>
    </property>
</bean>
```

# Part V. How-to

# Table of Contents

# Chapter 16. Writing your own commands

A Geomajas command usually consist of three classes, the actual command (which implements the `Command` interface, preferable even the `CommandHasRequest` interface), and two data transfer objects, one to pass the request parameters (extending `CommandRequest`, `LayerIdCommandRequest` or `LayerIdsCommandRequest`), and one which carries the response (extending `CommandResponse`).

It is important to assure your request object extends from LayerIdCommandRequest or LayerIdsRequest when one of the parameters is the layer id (or a list thereof). This can be used by the command dispatcher to assure the layer specific (transaction) interceptors are called.

To create a new command we recommend you use a similar package structure as we used in the geomajas-extension-command module. That is to create a "command" package with under that a "dto" package which contains all the request and response objects, and to put the actual commands in sub packages based on some kind of grouping. This helps to automatically determine a sensible command name.

The basic command implementation looks like this:

### Example 16.1. Example command template

```
package com.my.program.command.mysuper;

import com.my.program.command.dto.MySuperDoItRequest;
import com.my.program.command.dto.MySuperDoItResponse;
import org.geomajas.annotation.Api;
import org.geomajas.command.CommandHasRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

/**
 * Simple example command.
 *
 * @author Joachim Van der Auwera
 */
@Api
@Component()
public class MySuperDoItCommand implements CommandHasRequest<MySuperDoItRequest

    private final Logger log = LoggerFactory.getLogger(MySuperDoItCommand.class

    @Override
    public MySuperDoItRequest getEmptyCommandRequest() {
        return new MySuperDoItRequest();
    }

    @Override
    public MySuperDoItResponse getEmptyCommandResponse() {
        return new MySuperDoItResponse();
    }

    @Override
    public void execute(MySuperDoItRequest request, MySuperDoItResponse respons
```

```
        log.debug("called");
        // ..... perform the actual command
    }
}
```

Note the presence of the "@Component" annotation which assures the command is registered. You could add the name under which the command needs to be registered in the annotation, but when that is omitted, the default command name is derived from the fully qualified class name. In the example given here this results in command name "command.mysuper.DoIt".

The default way to determine the command name assumes there is a package named "command" in the fully qualified name of the implementing class. It will remove everything before that. It will then remove a "Command" suffix if any. Lastly, it will remove duplication between the intermediate package (between "command" and the class name) and the class name itself. Some examples:

### Table 16.1. Samples of command name resolution

| Fully qualified class name | Command name |
|---|---|
| my.app.command.DoIt | command.DoIt |
| my.app.command.super.DoIt | command.super.DoIt |
| my.app.command.super.DoItCommand | command.super.DoIt |
| my.app.command.super.SuperDoItCommand | command.super.DoIt |
| my.app.command.super.DoItSuperCommand | command.super.DoIt |
| my.app.command.super.CommandDoIt | command.super.CommandDoIt |
| my.app.command.super.CommandSuperDoIt | command.super.CommandSuperDoIt |
| my.app.command.super.CommandDoItSuper | command.super.CommandDoIt |

You have to include a line in your Spring configuration to scan class files for annotation to make the components available. For the case above, this could be done by including the following XML fragment in one of your Spring configuration files.

### Example 16.2. Scan to assure command is available

```
<context:component-scan base-package="com.my.program"
    name-generator="org.geomajas.spring.GeomajasBeanNameGenerator" />
```

The command will be executed using a singleton. The use of object variables is not recommended. Any object variables will be shared amongst all command invocation, which can be coming from multiple threads at the same time.

Note that it is not mandatory to create your own request and response object classes. If you don't require any parameters you can use `EmptyCommandRequest` as request class. If you only require a layer id, then use `LayerIdCommandRequest`. If you only return a success code, you could use the `SuccessCommandResponse` class.

You have to take care that all objects that are referenced by your request and response objects are actually serializable for the clients in which the commands need to be used. For GWT you have to assure the no-arguments constructor exists and that the class can be compiled by GWT (no Hibernate enhanced classes, no use of "`super.clone()`",...).

When the commands are included in a separate module, you should assure the sources are available as these are needed for GWT compilation. This can easily be done using the Maven source plug-in.

### Example 16.3. Maven source plugin

```
<plugin>
```

```
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.1.2</version>
    <executions>
        <execution>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <includePom>true</includePom>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Actually including the sources can then be done using a dependency like the following (this includes the staticsecurity module, both the actual code and the sources). You could set "provided" scope on the source dependency to exclude it from the war file. However, this may prevent use of GWT development mode.

### Example 16.4. staticsecurity source plugin - including source

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-staticsecurity</artifactId>
    <version>${geomajas-plugin-staticsecurity-version}</version>
</dependency>
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-staticsecurity</artifactId>
    <version>${geomajas-plugin-staticsecurity-version}</version>
    <classifier>sources</classifier>
</dependency>
```

# Chapter 17. Writing your own security service

A security plug-in provides an implementation of the `org.geomajas.security.SecurityService` interface. The work is done in the `getAuthentication` method which returns an `Authentication` object for a token string. The server does not (need to) know how tokens are generated. This is the responsibility of the client. In the case of the GWT2 client, this is done using the `setTokenRequestHandler` method on `GwtCommandDispatcher`.

The task of the security service is to read the policies and convert these into an `Authorization` objects. These contain methods to test whether certain operations are allowed or not. The different `Authorization` objects are combined by the `SecurityManager` into the `SecurityContext`, which can be injected in the code.

The security service which is provided by the plug-in can then be used in the security configuration (see chapter Chapter 12, *Security configuration*).

For the implementation of your Authorization class, it is important to assure that the instances are serializeable and can also be deserialized. This is important to allow the caching to be clustered. While we don't require that the class implements `Serializeable` (thanks to the use of JBoss Serialization), you should think of the following points;

- A no-arguments constructor is needed. This is allowed to be private (though that is not recommended, better make it protected if you don't want it to be generally used).
- If the authorization class is an inner class, do make it static if possible (otherwise the object has an implicit reference to the containing object, assuring it is included in the serialized state). The inner class should not be private as this cannot be deserialized.
- If you need a logger, declare it as

  ```
  private final transient Logger log = LoggerFactory.getLogger(ClassName.class);
  ```

  to ensure that the logger is not serialized.
- You should not use auto-wiring. Instead, make sure your class implements `AuthorizationNeedsWiring`. This defined the `wire()` method which allows you to query the application context. Make sure you declare wired properties as transient. The `wire()` method is called when the authorization is attached to the `SecurityContext` (both for freshly created and deserialized instances).

# Chapter 18. Adding your own GWT service as a MVC controller

When integrating Geomajas with an existing GWT application, you can add your existing GWT-RPC service to the application context as an MVC controller.

## 1. Create the MVC controller

The code for creating an MVC controller can be based on the `GeomajasController` class (see GeomajasController.java [http://grepcode.com/file/repo1.maven.org/maven2/org.geomajas/geomajas-client-common-gwt-command/2.2.0/org/geomajas/gwt/server/mvc/GeomajasController.java]). Start by implementing your service (e.g. `MyService`) as a bean and autowiring it in a new controller:

```
@Controller("/myService")
@Api
public class MyController extends RemoteServiceServlet
              implements MyService, ServletConfigAware {

  private static final long serialVersionUID = 100L;

  @Autowired
  private MyService myService;

  @Autowired(required = false)
  private ServletContext servletContext;

  private SerializationPolicyLocator serializationPolicyLocator;

  @RequestMapping("/myService")
  public ModelAndView handleRequest(HttpServletRequest request,
              HttpServletResponse response) throws Exception {
    doPost(request, response);
    return null;
  }

  // put delegation methods here
  void myMethod() {
    myService.myMethod();
  }

  ....

  public ServletContext getServletContext() {
    if (null == servletContext) {
      throw new IllegalStateException(
              "getServletContext() cannot be used outside web context");
    }
    return servletContext;
  }

  public void setServletConfig(ServletConfig servletConfig) {
    try {
      super.init(servletConfig);
    } catch (ServletException e) {
```

```
      throw new IllegalStateException("init(servletConfig) failed", e);
    }
  }
}
```

Use the url path of your choice in both the controller name and the request mapping value.

# 2. Add the controller to the web context

To add a controller to the web context, create a controller configuration file `mycontroller.xml` with the following content:

```
<beans xmlns="http://www.springframework.org/schema/beans"
              xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:context="http://www.springframework.org/schema/context"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-2.0.xsd">

  <context:component-scan base-package="<path.to.you.controller.package>" />

</beans>
```

Use the correct scanning package here. This file can be put in WEB-INF and added to the config locations of the dispatcher servlet in your `web.xml`:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-cla
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:org/geomajas/spring/geomajasWebContext.xml
      WEB-INF/mycontroller.xml
    </param-value>
    <description>Spring Web-MVC specific (additional) context files.</descripti
  </init-param>
  <load-on-startup>3</load-on-startup>
</servlet>
```

# 3. Access your RPC service from the client

To access your GWT-RPC service, create a proxy as usual:

```
MyServiceAsync service = (MyServiceAsync) GWT.create(MyService.class);
ServiceDefTarget endpoint = (ServiceDefTarget) service;
endpoint.setServiceEntryPoint(GWT.getModuleBaseURL() + "myService");
```

# Chapter 19. Setting up logging to a file

This is not really Geomajas specific, but as a simple guide, a configuration which can be used to setup logging to a file, using logback as the logging framework:

```
<configuration debug="true">

  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>/var/log/ktunaxa/log.txt</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- rollover daily -->
      <fileNamePattern>/var/log/ktunaxa/log-%d{yyyy-MM-dd}.%i.gz</fileNamePatte
      <maxHistory>60</maxHistory> <!-- keep logs two months -->
      <timeBasedFileNamingAndTriggeringPolicy
        class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
        <!-- or whenever the file size reaches 20MB -->
        <maxFileSize>20MB</maxFileSize>
      </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</patter
      <charset>UTF-8</charset>
    </encoder>
  </appender>

  <logger name="org.geomajas" level="DEBUG"/>

  <root level="INFO">
    <appender-ref ref="FILE"/>
  </root>

</configuration>
```

Some comments

- Daily rolling log and also rolls when larger than 20MB.
- Previous logs are zipped using gzip.
- Two months worth or logs are kept (60 days to be precise).
- This example does enable debug logging. This is not really recommended in production!
- The default log level is set to INFO. This is recommended. Do not set the logging OFF as this will also hide all warnings and errors. If you think INFO is too verbose, use WARN (and maybe start a discussion in the project to see whether the log level is correct for the messages).

# Chapter 20. Using snapshots

To be able to use snapshots, you have to include the snapshot repository in your pom, in the repositories section:

### Example 20.1. Including the snapshot repository

```
<repository>
    <id>Geomajas-latest</id>
    <name>Geomajas snapshot repository</name>
    <url>http://repo.geomajas.org/nexus/content/groups/latest/</url>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
</repository>
```

You can now overwrite the version of Geomajas parts by putting them in the dependencyManagement section *before* the general geomajas-project-server dependency. For example, to use the latest version of the staticsecurity plugin, the dependencyManagement would look like this:

### Example 20.2. Overwriting version when using geomajas-project-server pom dependency

```
<dependencyManagement>
  <dependencies>
    ...
    <!-- place overwriting version before pom dependency-->
    <dependency>
      <groupId>org.geomajas.plugin</groupId>
      <artifactId>geomajas-plugin-staticsecurity</artifactId>
      <version>SNAPSHOT.VERSION</version>
    </dependency>
    ...
    <dependency>
      <groupId>org.geomajas.project</groupId>
      <artifactId>geomajas-project-server</artifactId>
      <version>RELEASE.VERSION</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

# Chapter 21. Set up cross-context communication between GWT client and another web application

In some cases it may be necessary to put the Geomajas server on a different server node (B) than the one that contains the HTML page (A). As long as that node B is located in the same domain, such a situation can be handled without taking recourse to cross-domain communication hacks.

Two measures have to be taken to ensure succesful cross-context communication:

1. The GWT client should direct its requests to a different URL. This can be done by calling the following method on the `GwtCommandDispatcher` instance:

   ```
   public void setServiceEndPointUrl(String url)
   ```

   The url should be the absolute service URL of the cross-context server (B)

2. The cross-context server (B) should have a `GeomajasController` in the web constext that is especially configured to accept the gwt.rpc policy file of the compiled GWT client:

   ```
   <bean name="/geomajasService" class="org.geomajas.gwt.server.mvc.GeomajasCont
    <property name="serializationPolicyLocator">
     <bean class="org.geomajas.gwt.server.mvc.ResourceSerializationPolicyLocator
      <property name="policyRoots">
       <list>
        <value>WEB-INF/policies/</value>
       </list>
      </property>
     </bean>
    </property>
   </bean>
   ```

   The policy file is the file with extension `.gwt.rpc` that is obtained after the GWT compilation process. For the above configuration it should be copied to the folder `WEB-INF/policies` in the web root of B.

This solution can also be useful if there is a library conflict between the Geomajas server and other parts of the application. In such cases web service communication or communication via javascript may be your best option.

# Part VI. Appendices

# Table of Contents

# Appendix A. Migrating between Geomajas versions

## 1. Migrating between Geomajas 1.13.0,1.14.0 and Geomajas (back-end core) 1.15.0

- Restructured GeomajasContext.xml and GeomajasWebContext.xml
  - Because of possible classpath collisions, the geomajasContext.xml and geomajasWebContext.xml files are renamed to geomajasContext[pluginname].xml and geomajasWebContext[pluginname].xml. Make sure that your context-param in the web.xml file points to classpath:org/geomajas/spring/geomajasContext.xml and the init-param of the dispatcherservice to classpath:org/geomajas/spring/geomajasWebContext.xml

## 2. Migrating between Geomajas 1.12.0 and Geomajas (back-end core) 1.13.0

- Backend has been upgraded to Geotools version 9.2.
  - You cannot use previous versions of geomajas-backend if you wish to use Geotools 9.x, you must upgrade to this version.
  - gt-epsg-wkt (which has been deprecated for quite some time) has been removed in favour of gt-epsg-hsql. if you have it in your dependencies it is best to remove them.
  - The axisorder handling default has been changed to reflect the standards. System.setProperty("org.geotools.referencing.forceXY", "true"); see: http://docs.geotools.org/latest/userguide/library/referencing/order.html for a lengthy explanation.

    I did not notice any problems with Geomajas changing this, so as long as you stick to using Geomajas functionality and/or Geotools you should be ok.
  - The Unittests with transformations between epsg:4326 and epsg:31370 (Belgium Lambert) have a discrepancy of about 2 meters with the old version. It should be noted that these tests are done with values far outside the Belgian boundaries (world scale). Tests that stay within reasonable bounds have a discrepancy of approx 10cm. I have added a test with a verified point within Belgian boundaries as a sanitycheck.
  - If you get exceptions/weird behaviour check:
    - Your POM: check if all Geotools libraries are the same version (mvn dependency:tree | grep gt-) (put older plugins last so they do not include older versions).
    - Your POM: upgrade plugins that are not compatible with Geotools 9.2:

      geomajas-layer-geotools / geomajas-face-rest / geomajas-layer-hibernate / geomajas-plugin-rasterizing / geomajas-plugin-deskmanager.
    - Your CODE: check the Geotools changes page if you use Geotools directly in your code: http://docs.geotools.org/latest/userguide/welcome/upgrade.html
  - If your unittests fail you might need to disable asserts on geotools:

    ```
    <plugin>
    ```

```
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>-da</argLine>
    </configuration>
</plugin>
```

Geotools assertions can fail when comparing CRS-objects, as these are possibly wrapped by Geomajas (notably CRS.transform()).

# 3. Migrating between Geomajas 1.11.1 and Geomajas (back-end core) 1.12.0

- No known issues.

# 4. Migrating between Geomajas 1.10.0 and Geomajas (back-end core) 1.11.1

- The Geomajas project no longer supports Java5, from now on, all builds allow Java6 source and produce Java6 bytecode.
- Default zoom behaviour for zooming to a selected point has changed: before it was keeping the current scale, now it zooms to the maximum scale (by default). The zoom scale for zooming to points can be changed on a per-layer level by assigning the `zoomToPointScale` property in `ClientLayerInfo`.
- The back-end now strictly adheres to the "editable" capability defined in attribute configuration. When this is false, the attribute cannot be written. As creating features without geometry is not allowed, having this set to false in GeometryAttributeInfo will prohibit the creation of features.

# 5. Migrating between Geomajas 1.9.0 and Geomajas (back-end core) 1.10.0

- No known issues.

# 6. Migrating between Geomajas 1.8.0 and Geomajas (back-end core) 1.9.0

- Pipelines are now fully checked on application startup, any problems in pipeline configuration will now cause application startup to fail (previously this only failed when the pipeline in question was executed).
- In the GetVectorTile pipeline the GetTileTransformStep has been split in the GetTileTransformStep and GetTileFillStep. The postTileTransform hook is still after the GetTileTransformStep, so the filtering of features to ensure they are only included in one tile has not yet happened.
- When defining a pipeline, if you have both an pipeline definition and a delegate reference, an exception will be thrown (previously the pipeline definition had preference - though the javadoc said differently).
- The GWT client no longer inherits the com.smartgwt.SmartGwt, but com.smartgwt.SmartGwtNoTheme. The reason for this is that the SmartGwt module automatically

uses the 'enterprise' theme. This in turn meant that choosing another theme resulted in Geomajas loading both themes at once.

It is now up to the application definition to specify which SmartGwt theme to use. This must be done by inheriting the specific theme module in your .gwt.xml file like this:

```
<module>
   <inherits name="com.smartclient.theme.enterprise.Enterprise"/>
</module>
```

- GeoTools has been upgraded to 2.7.1. This causes some incompatibilities. The GeoTools plug-in also needs to be updated to 1.8.0 to avoid problems.
- Recursive primitive attribute names (`a.b.c`) are not allowed in this release. They should be replaced by equivalent nested attributes (`<a><b><c></c></b></a>`). A custom form/datasource should be created if one wants to use nested attributes as top level attributes in SmartGWT. For the future, we consider improving widget configurations to allow explicitly mentioning the attributes to be displayed (including linked attributes). We would welcome feedback and contributions for this.
- While we have updated to GeoTools 2.7.1 we have not changed the definition of the projections. It may be useful to update the gwt-epsg-wkt dependency to 2.7.1 as well (though this could break tests if you have tests which verify (re)projection. This could be done by adding the following in your dependencies section:

```
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-wkt</artifactId>
    <version>2.7.1</version>
</dependency>
```

# 7. Migrating between Geomajas 1.7.1 and Geomajas (back-end core) 1.8.0

- Geomajas now automatically limits geometries to the transformable area when doind CRS transformations. This can have the effect that geometries are simplified. For example, a MultiPolygon which contains only one polygon may be converted to a Polygon geometry.
- The use of the GeomajasContextListener in web.xml is no longer recommended. We recommend you use the normal spring listeners. This does mean that you should add "classpath:" in front of the locations in contextConfigLocation (the default location is the web application context).

  Note that you should add both the ContextLoaderListener and RequestContextListener (this last one is not included in the GeomajasContextListener but is needed for some services like AutomaticDispatcherUrlService to function).

### Example A.1. Defining spring configuration locations in web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:org/geomajas/spring/geomajasContext.xml1
        WEB-INF/applicationContext.xml2
    </param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</lis
```

```
</listener>
<listener>
    <listener-class>org.springframework.web.context.request.RequestContextList
</listener>
    .....
```

1. root context for geomajas
2. additional context for your application
3. assures the application context is available
4. assures the request can be accessed

- The use of the CacheFilter servlet was introduced in 1.8.0. It is strongly recommended that you include it in your web.xml file to assure correct caching and compression on server-side responses. This will greatly decrease loading times.

```
<filter>
    <filter-name>CacheFilter</filter-name>
    <filter-class>org.geomajas.servlet.CacheFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CacheFilter</filter-name>
    <url-pattern>*</url-pattern>
</filter-mapping>
```

- The GWT version has also been updated from 2.0.3 to 2.1.1. This in turn requires that the maven-gwt-plugin used in the pom.xml is also updated from 1.2-CPFIX to 2.1.0-1.

# 8. Migrating between Geomajas 1.6.0 and Geomajas (back-end core) 1.7.1

- `ApplicationContextUtils` has been renamed to `ApplicationContextUtil` and is now included in the api (this was done to adhere to the coding style).

- When building the dojo face and the dojo-example application, the maven "-Pnoshrink" has been replaced by "-DskipSkhrink".

- The use of the dispatcher servlet was introduced in 1.7.1. It is strongly recommended that you include it in your web.xml file to assure all plug-ins which expect this can function.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:org/geomajas/spring/geomajasWebContext.xml</par
        <description>Spring Web-MVC specific (additional) context files.</desc
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/d/*</url-pattern>
</servlet-mapping>
```

- the "springsecurity" module has be renamed "staticsecurity" to more correctly address the nature of the plug-in and and to avoid possible confusion with Spring's security stuff. Additionally the old

module has been split in two, one part being the back-end/configuration module, and another the gwt module.

- Many of the layers contain a bug in the 1.6.0 version assuming that injected services are fully initialised (and thus usable) while building the application context. Because of changes in the implementation of some services, these bugs become visible when using the 1.7 backe-end. You have to update your layers as well to 1.7+ to avoid these problems.

# 9. Migrating from Geomajas 1.5.4 to Geomajas 1.6.0

- The gwt-client module no longer automatically adds the "nl" locale to the application. This should now be done by the application. You can do this by adding the line

  ```
  <extend-property name="locale" values="nl"/>
  ```

  to your gwt.xml file.

- In the GWT face, you should now use `MapContext` instead of directly accessing `GraphicsContext`.

- `RasterLayer.paint()` now throws `GeomajasException` instead of `RenderException`. The `RenderException` class has been moved to api-experimental.

- `LocaleSelect` now needs a parameter in the constructor. This parameter is the name of the default language.

- The OpenStreetMap layer changes changed groupId from "geomajas-layer-opentreetmaps" to "geomajas-layer-opentreetmap".

- `GeomajasSecurityException` has moved from "`org.geomajas.global`" to "`org.geomajas.security`".

- `AllowAllSecurityService` has moved from "`org.geomajas.internal.security`" to "`org.geomajas.security.allowall`".

- `VectorLayerService` and `RasterLayerService` have moved from "`org.geomajas.service`" to "`org.geomajas.layer`".

- In `LabelStyleInfo` the style for the font is now of type `FontStyleInfo`.

- `LayerIdsCommandRequest` has been introduced and this is now extended by `SearchByLocationRequest` (no change) and `UserMaximumExtentRequest` (changing `includeLayers` to `layerIds`).

# 10. Migrating from Geomajas 1.5.3 to Geomajas 1.5.4

- SuccessCommandResponse class contained typos. The methods `isSucces()` and `getSucces()` have been renamed to `isSuccess()` and `getSuccess()` respectively.

- Changes in pipeline and promotion to stable API.

- The method `getRasterLayer()` has been added in `ConfigurationService`.

- The `findMathTransform()` method in `GeoService` now throws `GeomajasException` instead of `FactoryException`.

- InternalTile changes (should not affect anybody as these are used internally in the back-end).
- Many `DtoConverterService` methods now throw `GeomajasException`.
- The method `getId()` has been added to `Layer`. All server layers should have a unique id. The id is automatically assigned based on the Spring bean name.
- Configuration changes: `maxTileLevel` has been removed as this was not used.
- Configuration changes: the server-side layers are no longer connected to the client-side layer configurations via the layerInfo objects. Instead, client-side layers refer directly to the server layer's id via a serverLayerId property. The references to the layerinfo objects are injected by a configuration postprocessor, so the layerInfo should no longer be set manually.

**Table A.1. Back end configuration changes**

| Name | Property | Description |
|---|---|---|
| LayerInfo | id | Removed, use id property of Layer instead |
| SnappingRuleInfo | layerInfo | Replaced with serverLayerId |
| | serverLayerId | String ,should refer to id of Layer bean |

**Table A.2. Client configuration changes**

| Name | Property | Description |
|---|---|---|
| ClientLayerInfo | serverLayerId | String, should refer to id of Layer bean |
| | layerInfo | Should no longer be set manually, will be set by Spring |

# 11. Migrating from Geomajas 1.5.2 to Geomajas 1.5.3

- The `LayerModel` class has been integrated in `VectorLayer`. This modifies the configuration. Where before you would have written

```
<bean name="countriesModel" class="org.geomajas.layermodel.shapeinmem.ShapeInM
    <property name="url" value="classpath:shapes/africa/country.shp"/>
</bean>
<bean name="countries" class="org.geomajas.internal.layer.layertree.DefaultVec
    <property name="layerInfo" ref="countriesInfo" />
    <property name="layerModel" ref="countriesModel" />
</bean>
```

into

```
<bean name="countries" class="org.geomajas.layer.shapeinmem.ShapeInMemLayer">
    <property name="layerInfo" ref="countriesInfo" />
    <property name="url" value="classpath:shapes/africa/country.shp"/>
</bean>
```

Note that this includes changing "layermodel" to "layer" in all module and package names.

- `FeaturePainter` interface and related stuff has been removed. These are obsolete with the introduction of the `VectorLayerService`.

- `GeotoolsLayer` has been renamed `GeoToolsLayer`.
- With the change in directory structure, the commands have moved from the `org.geomajas.extension.command` package to `org.geomajas.command`. The `LogCommand` has also been moved into the `general` sub-package.
- Security constraints are now applied in Geomajas. By default, nothing is authorized, so you always have to configure at least one security service. To go back to the old (allow-all) behaviour, include the following excerpt in your configuration file.

```
<bean name="security.securityInfo" class="org.geomajas.security.SecurityInfo">
    <property name="loopAllServices" value="false"/>
    <property name="securityServices">
        <list>
            <bean class="org.geomajas.security.allowall.AllowAllSecurityServic
        </list>
    </property>
</bean>
```

- Layers are now more sensitive to the attributes which are defined for the layer. Attributes which have not been defined in the feature info are not accessible this is the result of the refactoring where the `InternalFeature` store attributes as `Attribute` objects).

# 11.1. General API changes

The geomajas-API has been split up in a formal (geomajas-API) and experimental API (geomajas-api-experimental). All interfaces/classes from the cache and rendering packages have been moved to experimental. This means that the rendering pipeline is at the moment not a part of the official API, but instead more of a preview of what's to come. Furthermore, some major changes have been made in many other packages:

- The `org.geomajas.rendering.tile` has been moved to `org.geomajas.layer.tile`
- Introduction of a DtoConverterService that is able to convert DTO objects from and to back-end internal representations.
- All the different feature definitions have been cut down. Only 2 versions remain at the moment: a DTO feature (`org.geomajas.layer.feature.Feature`) and a feature definition used internally in the backed (`org.geomajas.layer.feature.InternalFeature`).
- All the different tile definitions have been cut down. Only 3 remain. 2 DTO tiles: `org.geomajas.layer.tile.VectorTile` - used in vector layers and `org.geomajas.layer.tile.RasterTile` - used in raster layers. The third is the `org.geomajas.tile.InternalTile`. This tile is used internally on the back-end.
- `GeometricAttributeInfo` has been renamed to `GeometryAttributeInfo`.
- `ApplicationService` has been renamed to `ConfigurationService`.

# 11.2. Configuration changes

The configuration API has been split up in a back-end part and a client (or faces) part. The following general rules have been kept in mind:

- Back-end configuration should be restricted to those properties that are functionally needed on the back-end. We essentially regard the back-end as a container of layers or, in WFS terms, feature types. Higher level concepts like map or application should be dealt with at the client (or faces) level.
- Client configuration should not impact the back-end state. In the near future, this will make it possible to reconfigure clients without restarting the server.

The configuration API has profoundly changed. Where possible, the back-end classes have retained their original (before the split) names, after pruning them to remove all client related information. The client classes have been mostly created from scratch and have been named `ClientXxxInfo.java` for consistency. They have been located in a separate package, called `org.geomajas.configuration.client.` The following table gives a top-down overview of the back-end configuration classes (new classes and properties have been marked in **bold**):

## Table A.3. Back end configuration changes

| Name | Property | Action or description |
|---|---|---|
| ApplicationInfo | * | removed |
| LayerInfo | label | moved to ClientLayerInfo |
| | visible | moved to ClientLayerInfo |
| | viewScaleMin, viewScaleMax | moved to ClientLayerInfo |
| VectorLayerInfo | labelAttribute | moved to LabelStyleInfo |
| | snappingRules | moved to ClientVectorLayerInfo |
| | styleDefinitions | replaced by namedStyleInfos |
| | creatable, updatable, deletable | moved to ClientVectorLayerInfo (automatically assigned) |
| | **namedStyleInfos** | list of NamedStyleInfo. Lists the predefined styles available for this layer. Multiple styles are possible so clients can choose a style |
| RasterLayerInfo | style | moved to ClientRasterLayerInfo |
| **NamedStyleInfo** | **featureStyles** | list of FeatureStyleInfo. Ordered list of style definitions with applicable filters. Together with the label style they define a single named layer style. |
| | **labelStyleInfo** | label attribute name and style |
| **FeatureStyleInfo** | * | replaces StyleInfo same properties except for index |
| | index | replaces id (automatically assigned) |
| **LabelStyleInfo** | * | replaces LabelAttribute, same properties |
| ValidatorInfo and XxxConstraintInfo | * | moved to package `org.geomajas.configuration.validat` |

The most important changes are:

- The removal of client-side properties like visible, label, viewScaleMin, viewScaleMax, style and snapping rules. These are moved to the client configuration (see client documentation).
- The replacement of the single style definition list by a set of named styles. These are styles that are preconfigured in the back end.
- Inclusion of the label attribute name and style as part of the named style. This is more logical and in line with the SLD (Styled Layer Descriptor) specification.

Apart from these changes in content, some general technical improvements have been made as well:

- The Spring bean name (or id) is used to set the id property of the class if there is one. This makes it unnecessary to define the id separately. The way this is done is by using a Spring `BeanPostProcessor`. (see `org.geomajas.internal.configuration.ConfigurationBeanPostProcessor`)
- Some calculations that were previously done in the `GetConfigurationCommand` are now done in the `ConfigurationBeanPostProcessor.`
- Cloning of the client configuration classes can be done with general deep cloning techniques like serialization, bypassing the need for custom cloneable implementations.

As usual, example configurations can be found in the application projects.

# 12. Migrating from Geomajas 1.5.1 to Geomajas 1.5.2

- "layerRef" is renamed to "layerIds" in `LayerTreeNodeInfo`.

# 13. Migrating from Geomajas 1.5.0 to Geomajas 1.5.1

- Configuration has changed from the proprietary format to using Spring configuration.
- There is now a `CommandDispatcher` service and official command names and defined request and response objects. Deprecated commands have been removed.

# 14. Migrating from Geomajas 1.4.x to 1.5.0

- In your application.xml, you should change "OSMLayerFactory" to "OsmLayerFactory"
- In your application.xml, you should change "WMSLayerFactory" to "WmsLayerFactory"
- replace package "layermodels" with "layermodel"
- replace "org.geomajas.core.application.DefaultLayerFactory" with "org.geomajas.internal.application.DefaultLayerFactory"
- mapWidget.addController() and mapWidget.removeController() have been removed. They are replaced by mapWidget.setController(). You could only add one controller anyway.