Geomajas user guide for developers

Geomajas Developers and Geosparc

Geomajas user guide for developers

by Geomajas Developers and Geosparc

1.9.0

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 for developer who want to use and extend the Geomajas GIS framework.

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 back-end, we only allow dependencies which are freely distributable for commercial puposes (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

Part II. Architecture

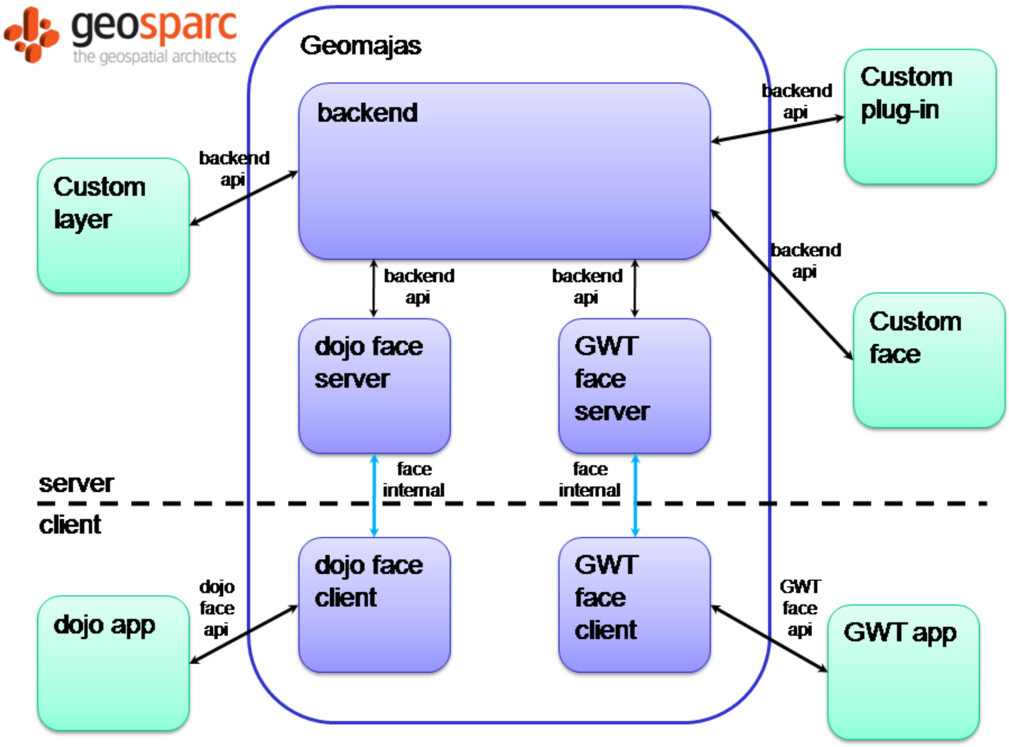
Table of Contents

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 *back-end* and *faces*.

Figure 2.1. Geomajas back-end and faces



The back-end is where you configure your maps, layers and attributes/features. It is always server side. The back-end has an API for interaction with the outside world and for extension using plug-ins. While one of the main purposes of the back-end 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 faces. The back-end is agnostic of web (or other) display frameworks. Faces are often split in two modules, a sever-side module (which directly talks to the back-end using java calls) and serializes data to the client, and a client-side module which only talks to the server side module. The communication between the two modules is private to the face. The face 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).

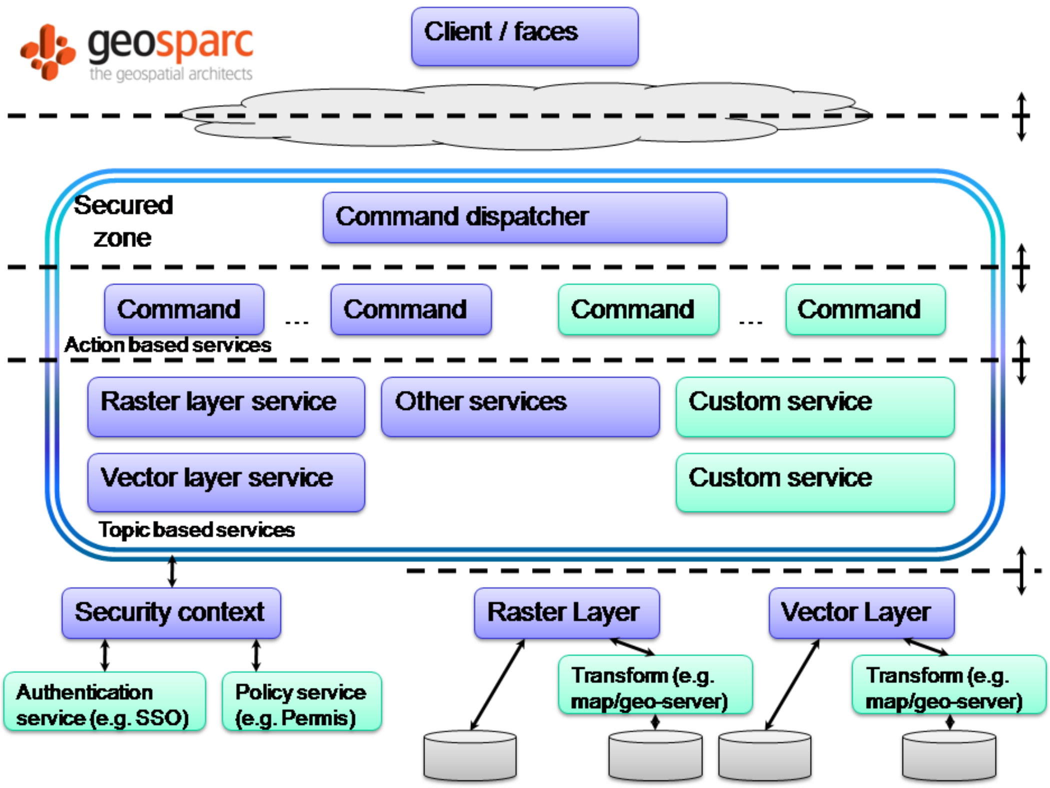
The purpose of Geomajas is to provide rich editing of GIS data in the browser, but the faces also make other applications possible. You could unlock the maps which are configured in Geomajas using a face which makes data available as web services (this would result in a face with only a server-side module).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

Geomajas contains two faces out-of-the-box.

The a dojo face, which uses the dojo toolkit JavaScript widget library in the browser, is mainly provided for backward compatibility. Up until Geomajas 1.4 this was the only face which existed. It integrates well with dojo but has the disadvantage that you need to develop in both java (for the server side) and JavaScript (for the client side) and that debugging can be a challenge.

Since 1.5 we also provide a GWT face. This allows 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



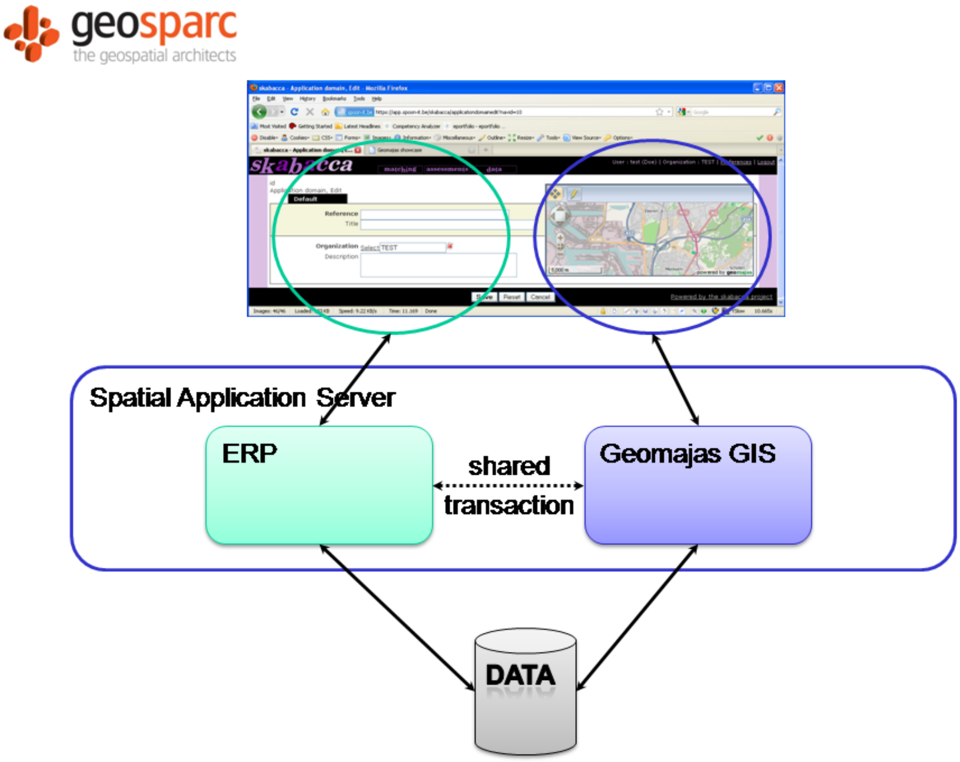
The Geomajas back-end is built from many services which are wired together using dependency injection (DI). This wiring is partly done automatically, and partly through the configuration files. Thanks to the inversion of control (IoC) the back-end is very flexible and can be customized at will.

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).

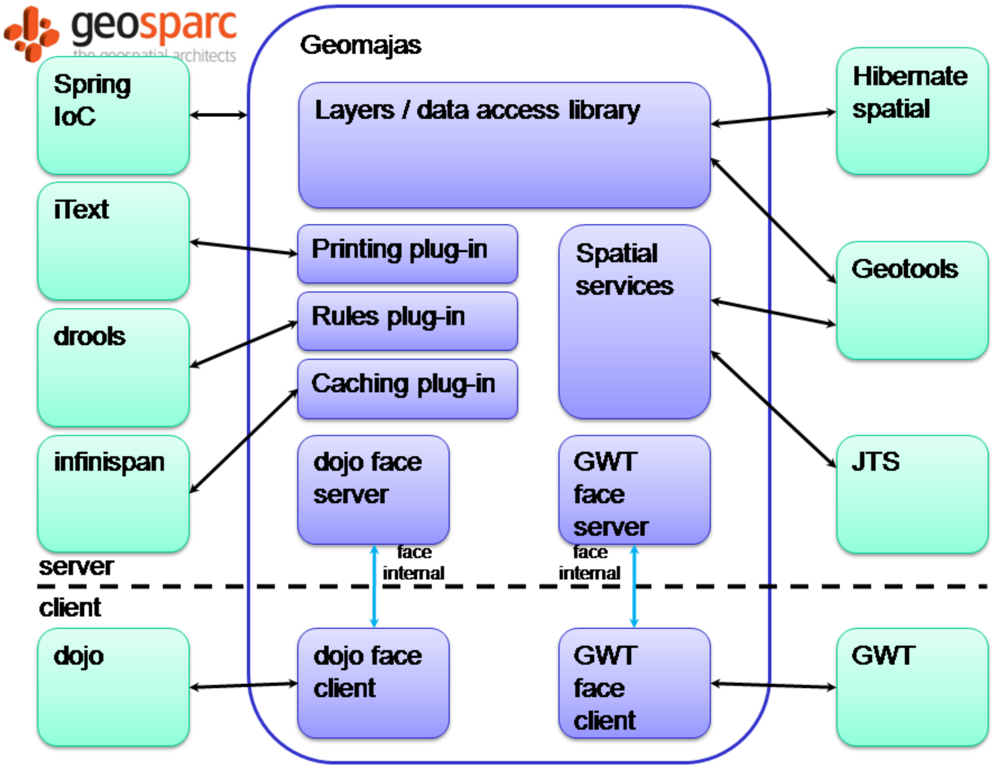
The layers access the actual GIS data, either directly or using some kind of transformation service (for example a GeoServer or MapServer instance).

Figure 2.3. Geomajas for mashups



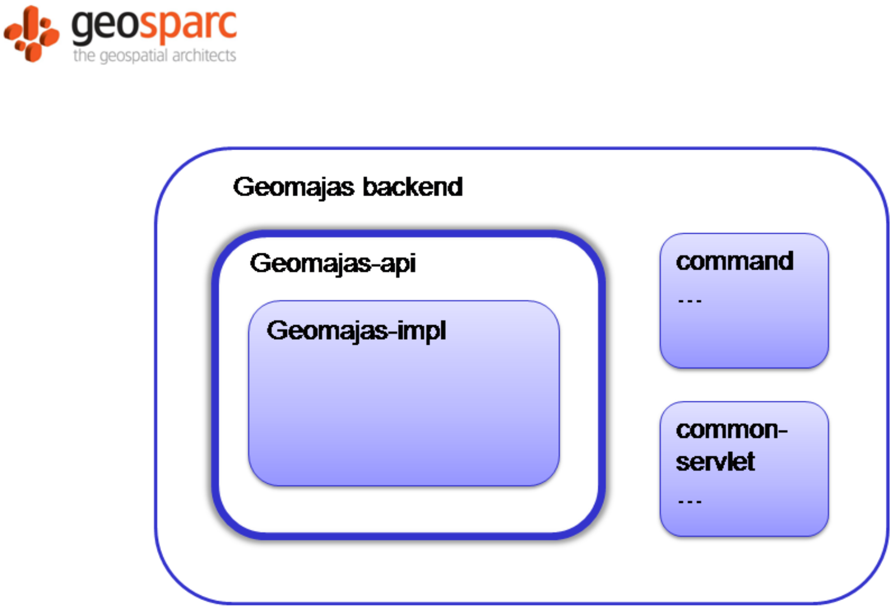
With this advanced configuration, many integration options exist. One example is displayed above, the inclusion of Geomajas in an existing application. On the client side, you just have to include the map widget in your web application. On the server side, there are many options, but you could for example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



The Geomajas back-end 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 back-end.

- *command*: commands are used as main interaction point between the face (client side) and the Geomajas back-end. 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.

- *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.

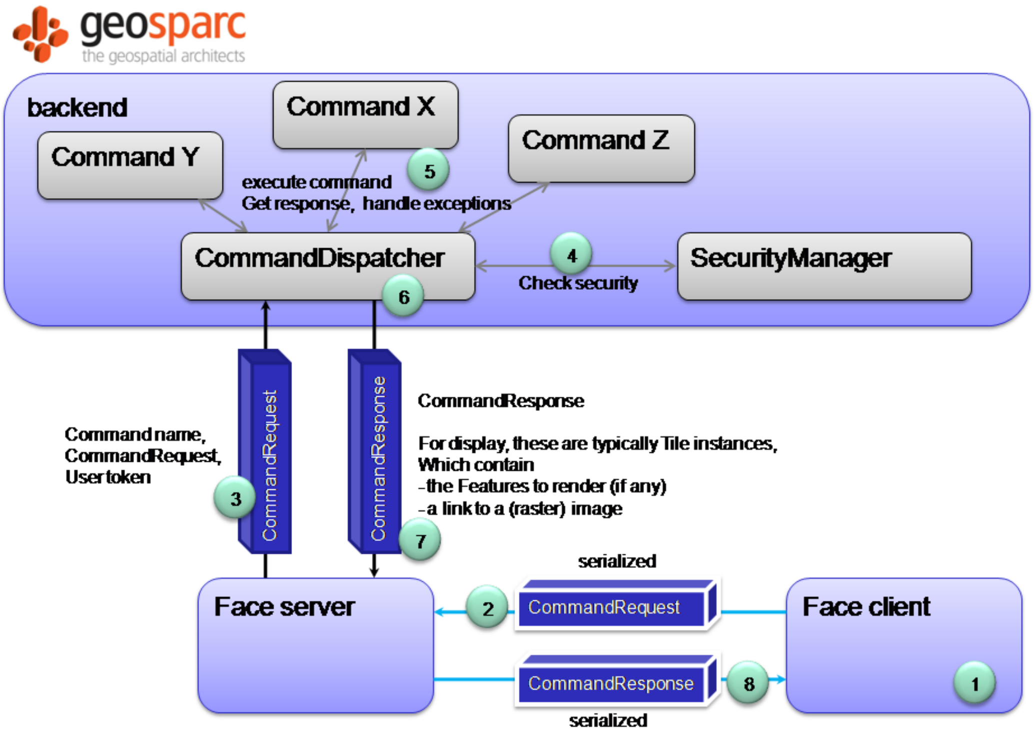
- *pipeline*: all Geomajas back-end services which deal with layers are implemented using pipelines. A pipeline is a list of steps (actions) executed in order. Each pipeline can be overwritten for a layer, or you can change the default which is used when not overwritten for a layer.

  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,...

- *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 face and commands



The interaction of the client faces with the Geomajas back-end 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 face server. For web applications, this will typically be done using an AJAX request.
3. The face server 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 face server.

8. The face server serializes the `CommandResponse` back to the face client.

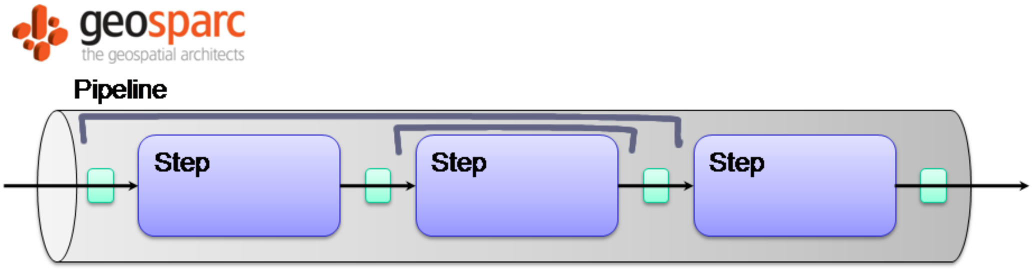
When the command had something to do with rendering, then the response object is likely to contain a `Tile`.

2. Pipelines

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

They are comparable with 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 back-end 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 back-end

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.

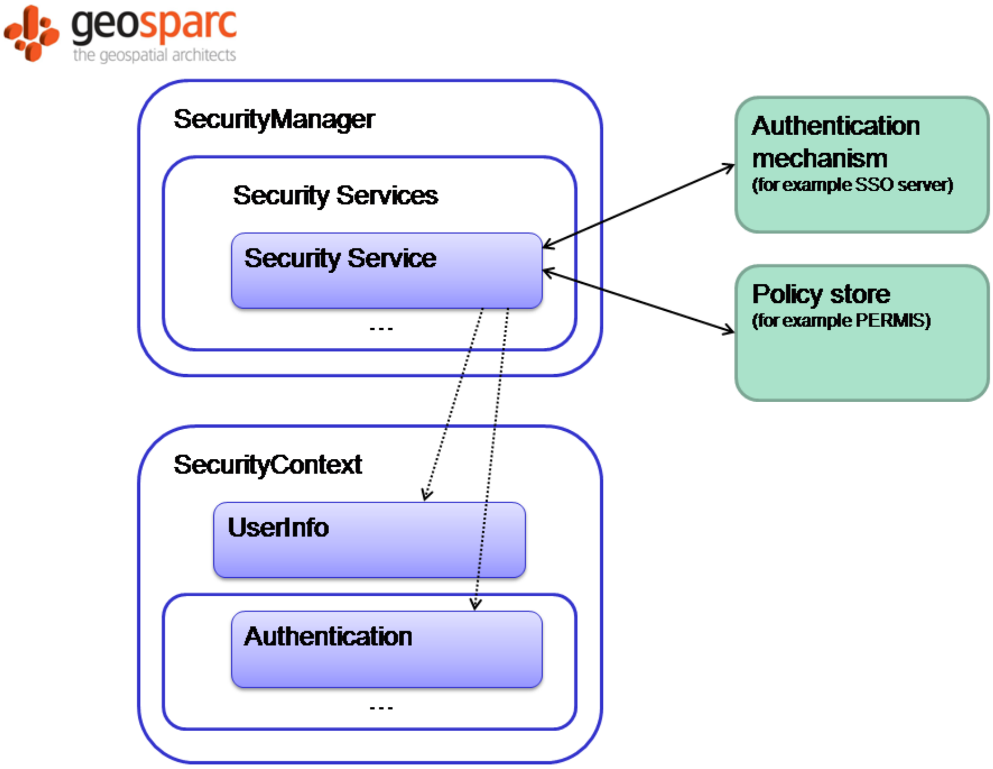
4.1. Security architecture

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 back-end 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 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 faces.

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 (the face probably) to assure that a new authentication token is created.

The authorization have two possible results. When reading or rendering, all unauthorized data should simply 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 back-end

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

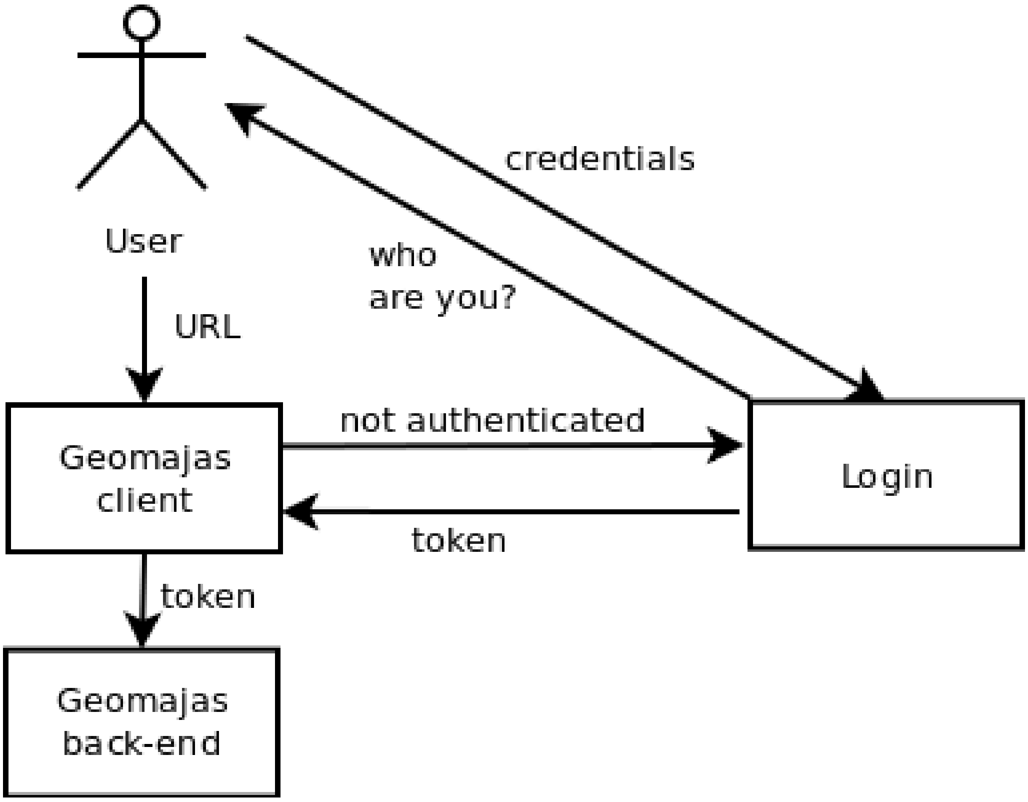
The application will then check whether the user is logged in. If that is not the case, the user is redirected to the login page. This may be an external page as provided by an authentication server (as often used for SSO (Single-Sign-On) solutions), or it could be a login widget. Note that the framework 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 back-end.

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

Figure 2.9. Logging into a Geomajas system



Note

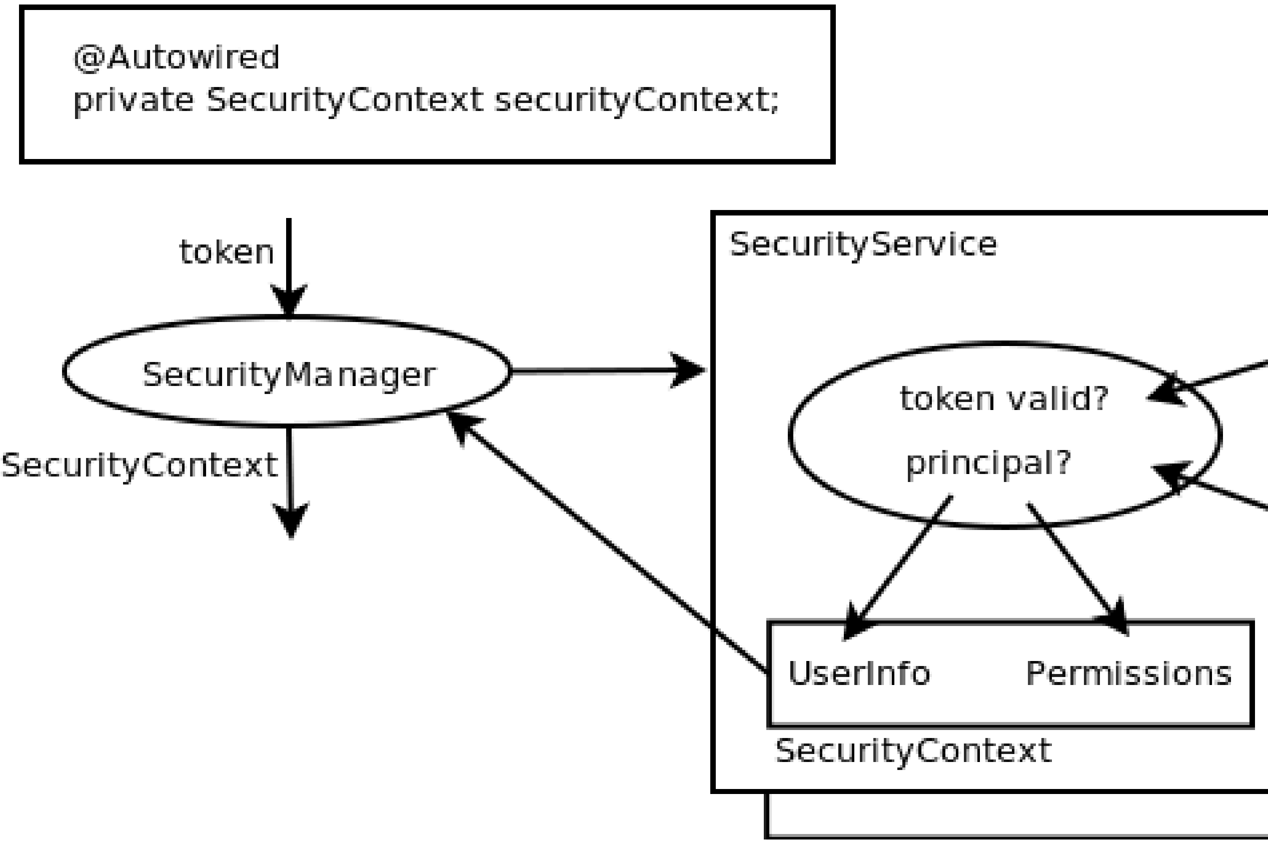
It is important to know that the security token typically has a limited validity. As such, it can happen at any moment that the token is no longer valid and the login screen needs to be presented again. The application author should consider this while developing.

At the back-end, 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.

Back-end 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.

Face:

- The face is responsible for assuring a authentication token is included in all access to the back-end.
- 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 face 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. Plug-ins

Geomajas provides a basic set of functionality as part of the back-end core. This can be extended and made available using plug-ins. One of the functions of the back-end core is to act as a plug-in container. Plug-ins are optional libraries that extend the core functionality by taking advantage of the public API. There are three special types of plug-ins, faces, layers and security plug-ins. which provide extra features, faces, layers and other plu

Faces provide external interfaces for Geomajas. These give access to users or external systems to the configured data. The faces which are included in the Geomajas project are

- GWT face : our recommended face for displaying and editing GIS data in the browser. This allows you to build your web user interface in Java.
- dojo face : a face which allows web display and editing using dojo toolkit. The user interface needs to be developed using JavaScript.

The layer plug-ins provide access to the actual data which needs to be displayed as part of a maps. There are basically two types of layers, providing either raster data (bitmaps) or vector data. The layers which are provided as part of the normal distribution include

- *geomajas-layer-hibernate* (vector): 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-geotools* (vector): access data from any vector data source which has a GeoTools data store defined for it (http://geotools.org/javadocs/org/geotools/data/DataStore.html).
- *geomajas-layer-google* (raster): include Google rasters. 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-openstreetmap* (raster): support for raster data coming from the OpenStreetMap project (http://www.openstreetmap.org/).
- *geomajas-layer-wms* (raster): access data from a WMS server (http://www.opengeospatial.org/standards/wms).
- *geomajas-layer-shapeinmem* (vector): access data from an ESRI shape file which handled in memory. The actual data access if done using GeoTools (http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf).

Other plug-ins allow extensions in functionality, either by providing additional commands or extending the rendering pipelines, or they provide additional security services.

- *geomajas-command*: set of commands which are provided as part of the standard distribution. This is so fundamental to using Geomajas that it is provided as a back-end module.
- *geomajas-plugin-printing*: printing extensions for the framework
- *geomajas-plugin-staticsecurity*: a basic security service which can be configured as part of the Spring configuration and does not use an external source for users or policies, making the security configuration entirely static.

The back-end also contains a set of spatial services. These include services for accessing raster and vector services and a set of utility services.

Chapter 4. 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 back-end module are always required and at least one face and at least one layer plug-in. More plug-ins or faces can be added as needed.

1. Face and plug-in registration

Plug-ins (which include faces) are automatically discovered when available on the classpath. This is done using two files: META-INF/geomajasContext.xml and META-INF/geomajasWebContext.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. Geomajas has defined the following set of module categories (matching the directories in the source):

- *application*: working examples of applications using the Geomajas GIS framework.
- *backend*: these are essential Geomajas modules. Each Geomajas application needs these modules. However, you also need some a face and some plug-ins (like layers) or you won't be able to do much.
- *build-tools*: some modules which are useful for starting or building Geomajas or a Geomajas project.
- *documentation*: documentation modules, specifically the different Geomajas guides. These are the general guides, each of the plug-ins also has a documentation module.
- *face*: faces that present a certain Geomajas client interface to the user.
- *plugin*: modules that extend Geomajas. This can either add new functionality, add support for a certain type of data source, provide a security service or a combination.
- *test*: modules which are used for (integration) testing of Geomajas.

Full list of Geomajas modules:

Table 4.1. List of Geomajas application modules

Name	Purpose
geomajas-dojo-example	Example application using the dojo face. Is more advanced than the dojo-simple demo. Can be useful as template project when starting a new dojo based Geomajas application which uses project specific JavaScript code.
geomajas-dojo-simple	Simple example project using the dojo face. Can be a useful template project when starting a new dojo based Geomajas application.

Name	Purpose
geomajas-gwt-example	Example application using the GWT face which serves both as showcase and test application.
geomajas-gwt-simple	Simple example project using the GWT face. Very similar to the GWT archetype which can be used to start a new GWT based Geomajas project.

Table 4.2. List of Geomajas back-end 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 faces which are servlet based.
geomajas-impl	Main library with default implementations.
geomajas-testdata	Module which contains data which is used for testing Geomajas.

Table 4.3. List of Geomajas build-tools modules

Name	Purpose
geomajas-checkstyle	Module which contains the checkstyle definitions which should be adhered to for all code in the Geomajas source tree.
geomajas-dep	This module can be included in your "dependencyManagement" section to set default versions for many possible dependencies. This includes the current release versions of all Geomajas project modules and their major dependencies. The versions can always be overwritten in your pom. It does not indicate that module versions play well together (though they should if the API contract is adhered). This module should never contain snapshot builds.
geomajas-maven-dojo	Maven plugin which helps to combine all the JavaScript code for dojo, Geomajas and the project itself. This is usually referred to as the "shrink" or "shrinksafe" step.
geomajas-maven-plugin	Maven plugin which is used for generating the documentation. It extracts excerpts from the code to allow inclusion in the docbook guides.
geomajas-parent	parent project which includes some Geomajas specific settings like copyright, java version, checkstyle etc.
geomajas-plugin-archetype	Archetype for starting a new plugin.

Table 4.4. List of Geomajas documentation modules

Name	Purpose
contributorguide	Guide for contributors to the project. Includes information about compilation of the project, coding style, how to contribute to the documentation, JIRA guidelines etc.
devuserguide	Guide for developers who want to use Geomajas in their applications.

Name	Purpose
enduserguide	Guide for end-users who use the Geomajas widgets. This guide should probably be included in the application documentation.
style	Style module for conversion of the docbook files to usable output.
xslt	Transformation module for conversion of the docbook files to usable output.

Table 4.5. List of Geomajas face modules

Name	Purpose
geomajas-face-dojo	Modules for the dojo face, including documentation.
geomajas-face-gwt	Modules for the GWT face, including the documentation.

Table 4.6. List of Geomajas plug-in modules

Name	Purpose
geomajas-layer-geotools	Support for any data format GeoTools supports.
geomajas-layer-google	Support for GoogleMaps raster format.
geomajas-layer-hibernate	Support for database formats through Hibernate.
geomajas-layer-openstreetmap	Support for OpenStreetMap raster format.
geomajas-layer-wms	Support for the WMS raster format.
geomajas-plugin-printing	Adds printing capabilities beyond printing in the browser, by delivering the map as PDF.
geomajas-plugin-staticsecurity	Simple security service which allows including the entire security configuration in the Spring configuration files, making the configuration static.
geomajas-plugin-caching	Caching plug-in which allows improved speed by calculating data only once.

Table 4.7. List of Geomajas test modules

Name	Purpose
geomajas-test-integration	Integrations tests, currently mostly testing the security handling.

Part III. API

Table of Contents

Chapter 5. 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`". 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.

2. Back-end API

The full details about the API can be found in the published javadoc, available on the Geomajas site at http://www.geomajas.org/gis-documentation. There you can find the links for the different versions.

The API for the Geomajas back-end 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 back-end 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 back-end 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 back-end 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. GWT face API

The GWT plug-in also uses the "`@Api`" annotation to indicate classes and methods which are supported to remain stable between minor versions of the face.

You can expect to find this annotation on all widgets, though it is likely that not all public methods will be considered part of the API.

5. API compatibility and Geomajas versions

Versions have the structure "major.minor.revision".

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.

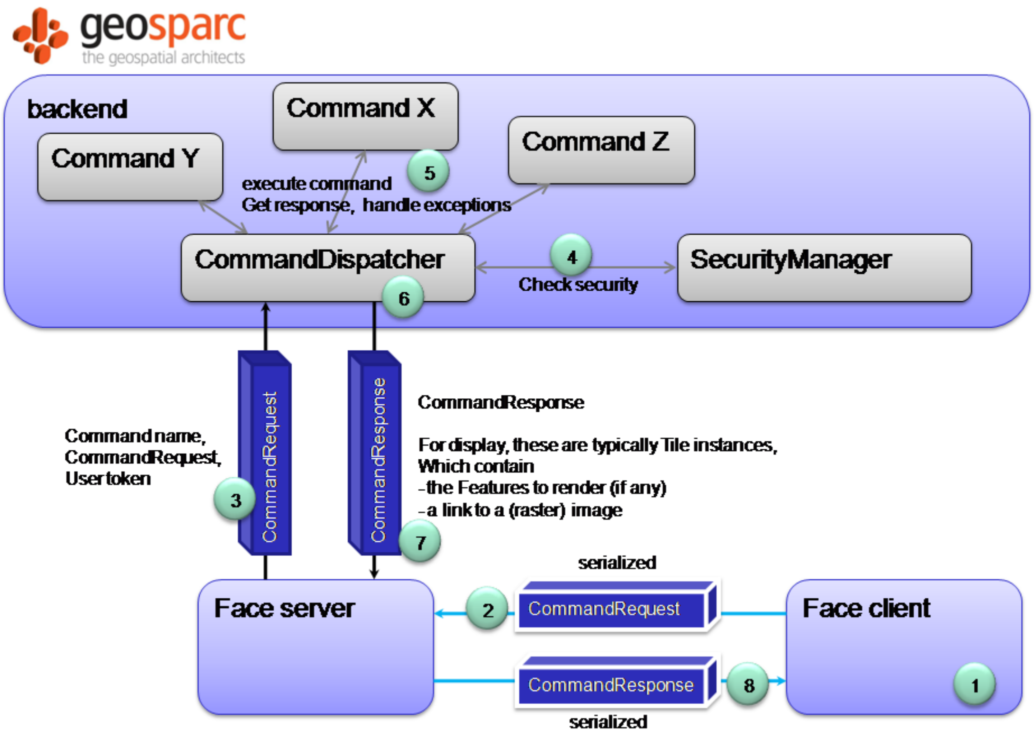
Because of the guarantees about API, the use of the "`@Deprecated`" annotation only indicates that a method or class is not recommended to be used. The method or class will not be removed in future versions with the same major number.

Chapter 6. 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 back-end for use by the faces.

Figure 6.1. Geomajas face 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 6.1. 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 6.2. 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. • *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. • *screenDpi*: screen resolution in dots per inch.

Table 6.3. 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 6.4. 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. • *bbox*: total bounding box wherein to fetch raster tiles. • *scale*: current scale in the client side map. • *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. • *nodeId*: identifier to be used in the DOM tree.

Table 6.5. 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. • *code*: the unique code of the tile to retrieve. • *scale*: the current scale on the map, client side. • *panOrigin*: translation for the tile on the client-side. • *filter*: extra filter that can be used to filter out data. • *crs*: the map's coordinate reference system. • *renderer*: should the server render to SVG or VML? • *styleInfo*: extra styles that can override the originally configured styles. • *paintGeometries*: should the geometries be painted in the tile? This is true by default. • *paintLabels*: should labels be painted in the tile? • *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 6.6. 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. • *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 6.7. MergePolygonCommand

MergePolygonCommand	
Registry key	command.geometry.MergePolygon
Module which provides this command	geomajas-command
Request object class	org.geomajas.command.dto.MergePolygonRequest
Parameters	• *polygons*: array of polygons that need to be merged • *allowMultiPolygon*: is a MultiPolygon allowed when merging multiple polygons?
Description	This command allows the user to merge multiple polygons into a single polygon or multipolygon.
Response object class	org.geomajas.command.dto.MergePolygonResponse
Response values	• *geometry*: the resulting geometry after the merge.

Table 6.8. 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. • *crs*: the map's coordinate reference system.

PersistFeatureTransactionCommand	
Description	Persist a single transaction on the backend (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 6.9. 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. • *attributeName*: the name of the attribute as configured in the feature info. • *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 6.10. 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. • *queryType*: specify exactly whether to search, possible values are `QUERY_INTERSECTS`, `QUERY_TOUCHES`, `QUERY_WITHIN` or `QUERY_CONTAINS`. • *ratio*: if queryType is `QUERY_INTERSECTS`, you can additionally specify what percentage of overlap is enough to be included in the search. • *layerIds*: array of layer ids to search in. • *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. • *crs*: the map's coordinate reference system. The *location* geometry will also be expressed in this crs.

SearchByLocationCommand	
	- *buffer*: before any calculation is made, it is possible to have the location geometry expanded by a buffer of this width (in crs space). - *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 6.11. 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. - *max*: maximum number of features to allow in the result. 0 means unlimited. - *crs*: crs which needs to be used for the geometry in the retrieved features. - *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. - *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". - *filter*: an additional layer filter which needs to be applied when searching. - *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.

SearchFeaturesCommand	
	• *features*: array of features which match the search criteria. Any geometry contained in the features uses the request crs.

Table 6.12. SplitPolygonCommand

SplitPolygonCommand	
Registry key	command.geometry.SplitPolygon
Module which provides this command	geomajas-command
Request object class	org.geomajas.command.dto.SplitPolygonRequest
Parameters	• *geometry*: the geometry that needs splitting. • *splitter*: the splitting geometry (usually a LineString).
Description	Split up a geometry into many pieces by means of a splitting geometry.
Response object class	org.geomajas.command.dto.SplitPolygonResponse
Response values	• *geometries*: the list of resulting geometries after the split.

Table 6.13. 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. • *includeRasterLayers*: true when raster layers should be included. Defaults to false. • *crs* : crs which should be used for the response.
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 7. 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 L

 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` (backend) and `HibernateFeatureModel` (Hibernate layer plugin) implementations.

Chapter 8. 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.

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 back-end 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:

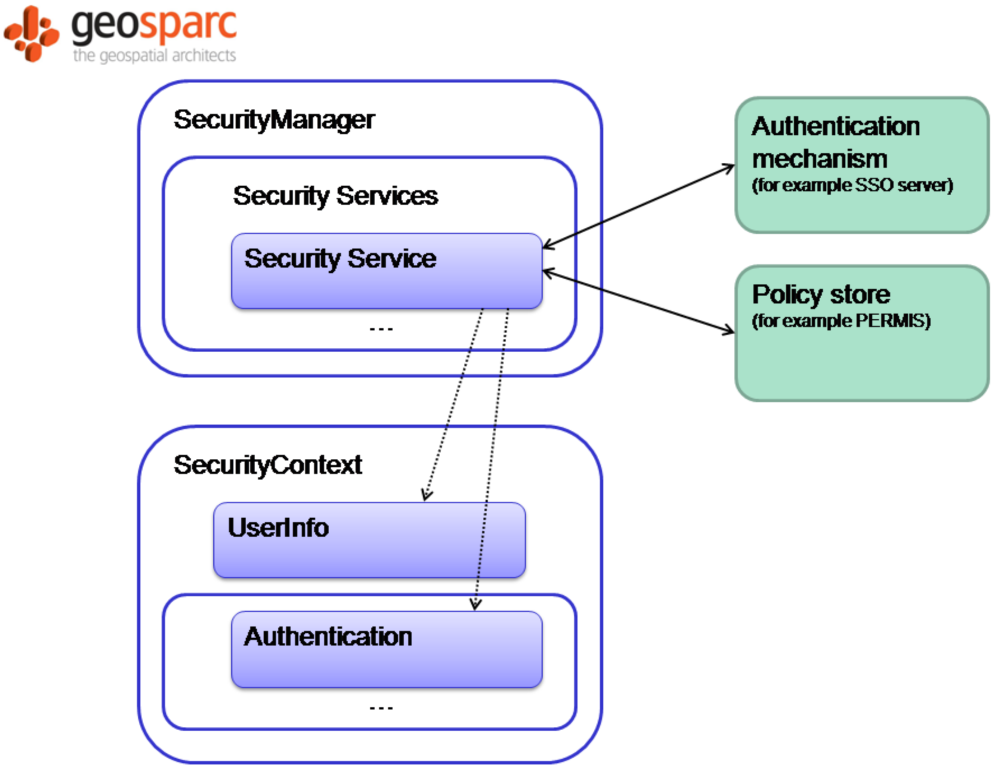
- access rights to a command
- access rights for a layer
- a filter which needs to be applied for a layer
- a region which limits the data which may be accessed for a layer
- access rights on the features
- access rights on the individual attributes of the features

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 Chapter 20, *Create a security plug-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 8.1. 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

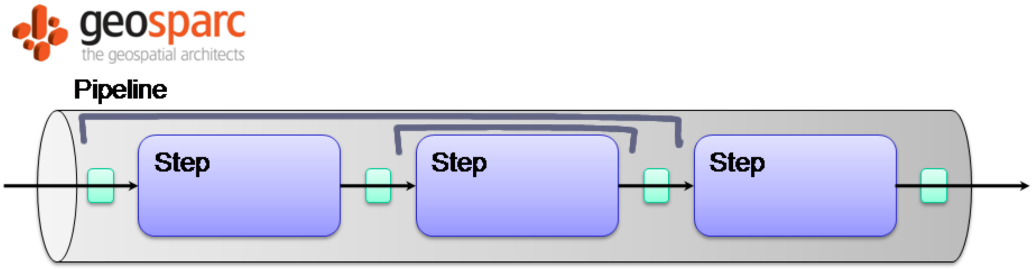
The security context allows access to the currently valid user's policies and some limited information (user id, name and organization). In your code, you just have to inject the security context. The face is responsible for assuring the current thread has the correct security context based on the credentials used when accessing the back-end (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 9. 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 9.1. 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 9.1. 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 9.2. 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 9.3. 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 9.4. 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 9.5. 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 10. Utility Services

The Geomajas back-end 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 faces).

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.

Part IV. Configuration

Table of Contents

Chapter 11. 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 4, “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 11.1. Defining spring configuration locations in web.xml

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

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</liste
</listener>
<listener>
    <listener-class>org.springframework.web.context.request.RequestContextListe
</listener>
    .....
```

root context for geomajas
additional context for your application
assures the application context is available
assures the request can be accessed

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

Example 11.2. 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*:META-INF/geomajasWebContext.xml</param-value>
        <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 face. 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 11.3. 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 11.4. 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
        <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 unt
        <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
        <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 a
        <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
        <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 re
        <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 co
        <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 11.5. 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/schem
http://www.springframework.org/schema/util http://www.springframework.org/schema
http://www.springframework.org/schema/aop http://www.springframework.org/schema
http://www.springframework.org/schema/tx http://www.springframework.org/schema/
```

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 back-end. Only the back-end classes are necessary to configure the back-end, 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 back-end classes exist in the 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 back-end 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. Geomajas configuration

The initial bean which needs to be defined is a bean indicating the *client application info*.

Example 11.6. ClientApplicationInfo definition

```
<bean name="gwt-simple" class="org.geomajas.configuration.client.ClientApplicat
    <property name="maps">
        <list>
            <ref bean="sampleFeaturesMap" />
            <ref bean="sampleOverviewMap" />
        </list>
    </property>
</bean>
```

As you can see, this defines the list of maps for the application. It may (optionally) also define some additional user info and a screen DPI parameter. The DPI refers to the resolution in pixels per inch of your monitor, for a PC its usually 96 (the default) or 72.

There needs to be at least one `ClientApplicationInfo` bean. The bean name is used when requesting the application info.

4. 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 12. Map 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 back-end configuration for the layer. The exact meaning for some of the fields depend on the actual layer, but most important features include:

Table 12.1. 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. 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). 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 12.1. Style info

```
<bean name="airportsInfo" class="org.geomajas.configuration.VectorLayerInfo">
    <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="airportsFeatureInfo" />
    <property name="namedStyleInfos">
        <list>
            <ref bean="airportsStyleInfo" />
        </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 12.2. 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 12.2. Feature info

```
<bean class="org.geomajas.configuration.FeatureInfo" name="airportsFeatureInfo"
    <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 de 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 calculationss 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 face documentation for more information on this.

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

Table 12.3. 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 withe 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.

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 12.4. 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 %')

Type	Operators	Example
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 12.3. Style info

```
<bean class="org.geomajas.configuration.NamedStyleInfo" name="airportsStyleInfo
    <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>
```

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 12.4. 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
        <property name="errorMessage" value="Invalid value: The value must be e
        <property name="constraints">
            <list>
                <bean class="org.geomajas.configuration.validation.NotNullConst
                <bean class="org.geomajas.configuration.validation.PatternConst
                    <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.

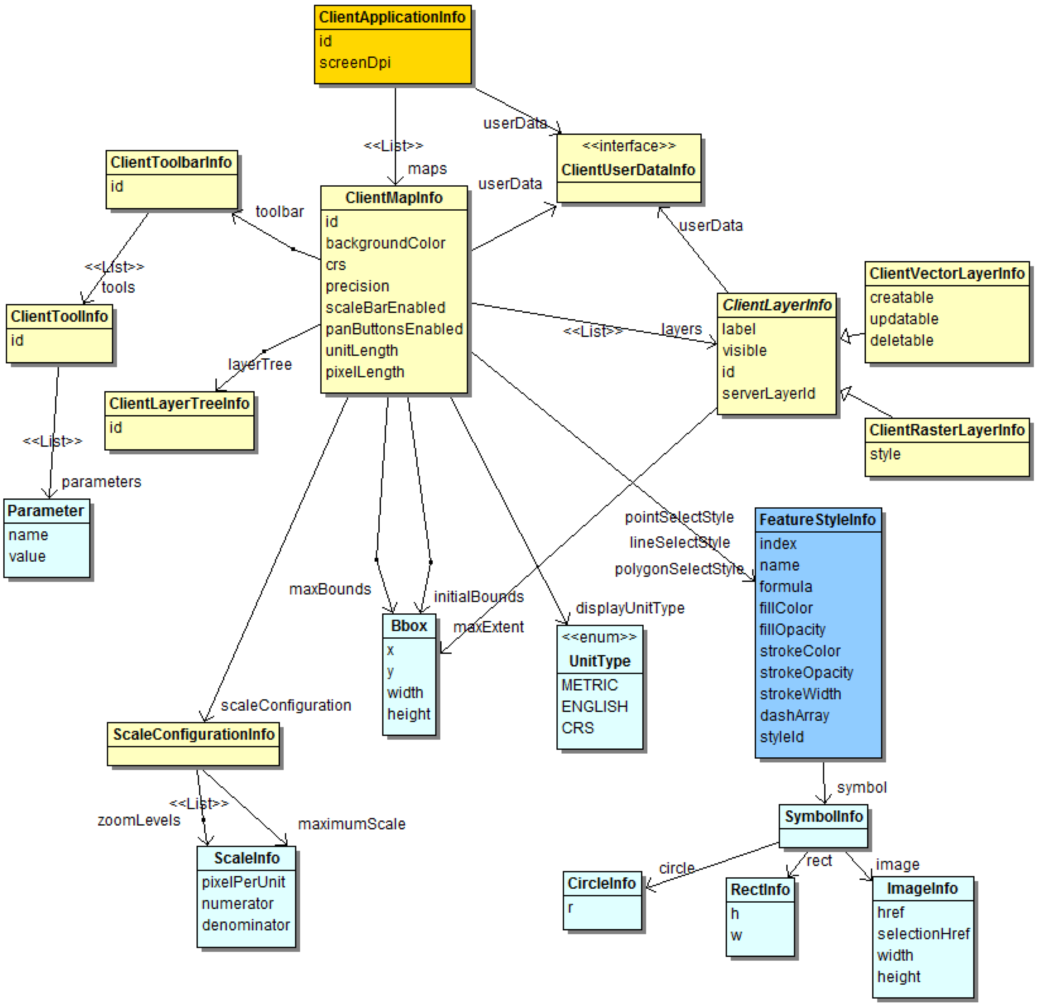
TODO.....

Table 12.5. BeanLayer configuration

Name	Description
features	List of features, which should be `org.geomajas.layer.bean.FeatureBean` instances.

3. Client configuration

Figure 12.1. Geomajas client configuration



3.1. Map configuration

A map is a client side object. The Geomajas back-end works almost exclusively on layers.[1]On the client side however, these layers are combined into maps. In general, the back-end never needs to know which map the layer is displayed in when doing its work. However the back-end does need to know the coordinate reference system which is used.

---

[1] The only current exception is the printing command which converts maps to PDF document. Clearly this also uses the map configuration.

Example 12.5. Client map configuration

```
<bean name="sampleFeaturesMap" class="org.geomajas.configuration.client.ClientMa
    <property name="crs" value="EPSG:4326" />
    <property name="displayUnitType" value="CRS" />
    <property name="initialBounds">
        <bean class="org.geomajas.geometry.Bbox">
            <property name="x" value="-180"/>
            <property name="y" value="-90"/>
            <property name="width" value="360"/>
            <property name="height" value="180"/>
        </bean>
    </property>
    <property name="layers">
        <list>
            <ref bean="wmsLayer" />
            <ref bean="countries110mLayer" />
        </list>
    </property>
```

The crs evidently refers to the map's coordinate reference system. The display unit type determines the unit type of the scale bar (METRIC, ENGLISH or CRS). The initial bounds determine the visible area of the map at startup time. The layers refers to the client layer info objects, not the server layer info or layer instances.

Additionally, a lot of style information can be included in the map configuration. This includes information like background colour, styles which should be used for selected points, lines and polygons and whether scale bare or pan buttons should be enabled.

Example 12.6. Client map configuration

```
<property name="backgroundColor" value="#F0F0F0" />
<property name="lineSelectStyle">
    <bean class="org.geomajas.configuration.FeatureStyleInfo">
        <property name="fillOpacity" value="0" />
        <property name="strokeColor" value="#FF6600" />
        <property name="strokeOpacity" value="1" />
    </bean>
</property>
<property name="pointSelectStyle">
    <bean class="org.geomajas.configuration.FeatureStyleInfo">
        <property name="fillColor" value="#FFFF00" />
    </bean>
</property>
<property name="polygonSelectStyle">
    <bean class="org.geomajas.configuration.FeatureStyleInfo">
        <property name="fillColor" value="#FFFF00" />
        <property name="fillOpacity" value=".5" />
    </bean>
</property>
<property name="scaleBarEnabled" value="true" />
<property name="panButtonsEnabled" value="true" />
```

It is possible to configure the pixels per tile that will be used for rendering and retrieving tiles. These can be defined in two ways:

- The MAP type will base the pixels per tile on the map dimensions.
- The CONFIGURED type you configure the size of the tiles, this defaults to 256x256 pixels.

An other important aspect of the map is the scale configuration. The scale configuration allows to define a maximum scale beyond which the user is not allowed to zoom in. This is not needed for zooming out as there is always a maximum bounds defined for the map (either explicitly or calculated as the union of the layer bounds). Next to that you can define a list of zoom levels. By default, the map will allow zooming to arbitrary scale levels but you may wish to enforce certain scale or zoom levels upon the user (like Google Maps does). By doing so, continuous zooming will no longer be possible and any zooming action will "snap" to the predefined scale levels.

Example 12.7. Client map configuration - scale configuration

```
<property name="scaleConfiguration">
    <bean class="org.geomajas.configuration.client.ScaleConfigurationInfo">
        <property name="maximumScale" value="1:1000" />
        <property name="zoomLevels">
            <list>
                <value>1:128000000</value>
                <value>1:64000000</value>
                <value>1:32000000</value>
                <value>1:16000000</value>
                <value>1:8000000</value>
                <value>1:4000000</value>
                <value>1:2000000</value>
                <value>1:1000000</value>
                <value>1:500000</value>
                <value>1:100000</value>
                <value>1:25000</value>
                <value>1:15000</value>
                <value>1:10000</value>
                <value>1:5000</value>
                <value>1:2500</value>
                <value>1:1000</value>
            </list>
        </property>
    </bean>
</property>
```

Scales can be defined in 2 possible notations:

- the 1 : x notation (see the above listing) is most commonly used in geographics and expresses the ratio between 1 meter on the screen and 1 meter on the earth's sphere
- the floating point notation (e.g. 0.0001) is used by us to express the number of pixels on the screen that correspond to 1 unit on the map (1 pixel per 10000 map units in our example)

Both scale definitions serve a different purpose. The 1 : x scale should give you an idea of what the true scale is at which the map is shown, although in practice this may depend on the DPI (actually PPI) and pixel size of your device. The floating point scale (which has units of pixel/m or pixel/deegree) is used to precisely define the resolution of raster images on the screen. If you use floating point notation, you can make sure that the scales that are being used in an application are the same as those of the raster layer(s) that lies beneath (see raster layer configuration). Otherwise the raster images may get blurry or unreadable when they need to be resized.

A map typically also contains a tool bar. If you want one, you have to specify the tools it should include.

Example 12.8. Client map configuration

```
<property name="toolbar">
    <bean name="sampleFeaturesMapToolbar" class="org.geomajas.configuration.cli
        <property name="tools">
            <list>
                <ref bean="ZoomIn" />
                <ref bean="ZoomOut" />
                <ref bean="ZoomToRectangleMode" />
                <ref bean="PanMode" />
                <ref bean="ToolbarSeparator" />
                <ref bean="ZoomPrevious" />
                <ref bean="ZoomNext" />
                <ref bean="ToolbarSeparator" />
                <ref bean="EditMode" />
                <ref bean="MeasureDistanceMode" />
                <ref bean="SelectionMode" />
            </list>
        </property>
    </bean>
</property>
```

Obviously the tools themselves need to be defined as well. You can pass some parameters to the tools. An example tool definition look like this.

Example 12.9. Tool configuration

```
<bean name="ZoomIn" class="org.geomajas.configuration.client.ClientToolInfo">
    <property name="parameters">
        <list>
            <bean class="org.geomajas.configuration.Parameter">
                <property name="name" value="delta" />
                <property name="value" value="2" />
            </bean>
        </list>
    </property>
</bean>
```

Note that the tool id and the names of the parameters are interpreted by the client, so it is the client face which defines the possible values.

Last but not least, you can also configure the layer tree component which may be connected to the map.

Example 12.10. Client map configuration

```
    <property name="layerTree">
        <bean name="sampleFeaturesTree" class="org.geomajas.configuration.clien
            <property name="tools">
                <list>
                    <ref bean="LayerVisibleTool" />
                    <ref bean="LayerLabeledTool" />
                    <ref bean="ShowTableAction" />
                    <ref bean="LayerRefreshAction" />
                </list>
            </property>
            <property name="treeNode">
                <bean class="org.geomajas.configuration.client.ClientLayerTreeN
                    <property name="label" value="Layers" />
                    <property name="layers">
                        <list>
                            <ref bean="wmsLayer" />
                            <ref bean="countries110mLayer" />
                        </list>
                    </property>
                    <property name="expanded" value="true" />
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

This defines the tools which are available in the layer tree widget, and the tree of layers (as a node, which can contain a list of nodes etc).

Note that the layers are indicated by referring to the client configuration object.

3.2. Client layer configuration

Layer configuration is split in two (linked) parts. You have to create the actual layer which is used in the back-end, and this layer needs to know the configuration information which is also used on the client side. Secondly, there is a distinction between raster and vector layers as they each needs a lot of specific information.

3.2.1. Raster layer

TODO.....

3.2.2. Vector layer

TODO.....

Chapter 13. 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 13.1. 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 13.2. Partial staticsecurity configuration

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

<bean name="security.securityInfo" class="org.geomajas.security.SecurityInf
    <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.SecurityServi
    <property name="users">
        <list>

            <!-- User elvis has restricted attribute editing permissions on
            <bean class="org.geomajas.plugin.staticsecurity.configuration.U
                <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>beans</value>
                                </list>
                            </property>
```

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

Chapter 14. 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 15. Dispatcher servlet configuration

Additional servlet configuration may be needed for any plugin that wants to support its own client-server communication protocol. This is typically the case for faces, but in general any plugin 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 chandler 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 controller
    <bean id="defaultHandlerMapping" class="org.springframework.web.servlet.han
        <!-- need security  -->
        <property name="interceptors">
            <list>
               <ref bean="securityInterceptor" />
            </list>
        </property>
    </bean>

    <bean id="securityInterceptor" class="org.geomajas.servlet.mvc.SecurityInte

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

    <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 autodetects the mapping based on the `@Request`Mapping 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 hashmap 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 16. 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 16.1. 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 the 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 16.2. 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 17. Writing your own commands

A Geomajas command usually consist of three classes, the actual command (which implements the `Command` 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 17.1. 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.command.Command;
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;
import org.springframework.stereotype.Component;

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

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

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

    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 17.1. 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 17.2. Scan to assure command is available

```
<context:component-scan base-package="com.my.program" name-generator="org.geoma
```

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 which are referenced by your request and response objects are actually serializable for the faces in which the commands need to be used. For the dojo face this may require the use of the "`@Json`" annotation to exclude fields. 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 17.3. 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 17.4. 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>
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-staticsecurity-gwt</artifactId>
    <version>${geomajas-plugin-staticsecurity-version}</version>
</dependency>
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-plugin-staticsecurity-gwt</artifactId>
    <version>${geomajas-plugin-staticsecurity-version}</version>
    <classifier>sources</classifier>
</dependency>
```

Chapter 18. Create a plug-in

The general procedure for creating a new plug-in is described here. Additional information for specific types of plug-ins is described in subsequent chapters.

1. Using the plug-in archetype

TODO.....

2. Plug-in structure

TODO.....

3. Plug-in declaration and dependencies

TODO.....

Chapter 19. Create a layer plug-in

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 configuration section above.

1. Writing your own layer

TODO.....

Chapter 20. Create a security plug-in

1. Writing your own security service

TODO.....

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 21. Integrating a Geomajas application in the ZK framework

1. Introduction

ZK [http://www.zkoss.org/] is a general purpose web framework that uses a server-side approach to create RIAs. The framework is Java-based and has a built-in mechanism to synchronize presentation state between client and server based on Ajax requests. Geomajas on the other hand is a GIS application framework based on GWT. GWT compiles java code to javascript and makes use of Ajax requests and a custom serialization mechanism to communicate with the server. There is no presentation state on the server and the client is functionally a thick client (as opposed to ZK).

When mixing two frameworks it is generally advisable to wrap UI components of the foreign framework in such a way that they become indistinguishable from native UI components. This technique is actually mandatory for frameworks that take complete control over the HTML layout, like ZK. While there is not mcuh information available with respect to GWT, we have successfully followed the same technique that was used in ZK for Google Maps and JQuery.

2. Integrating GWT widgets in ZK

ZK allows the creation of custom components. In the 5.0.3 version, we need to prepare four types of files in order to create a component:

- The lang-addon.xml file: Registers the new ZK component
- The component file (*.java): Server-side Java component object
- The widget file (*.js): Client-side JavaScript widget object (there is a one-to-one correspondance with the server component)
- The template files (*.dsp): (optional) molds to generate the HTML representation of the client part. Alternatively, a redraw method can be defined in the .js file.
- The zk.wpd file : Widget Package Descriptor (WPD) file describing the information of a package, such as its widget classes and external JavaScript files

To represent a GWT widget as a ZK component, a wrapper has to be created around the GWT widget. Although both frameworks are Java-based, there is no communication possible between both as GWT is client-side while ZK is server-side. This means that any wrapping has to be done by providing mutual hooks or callback functions in the javascript files.

We will now give a more detailed overview of each file and its functionalities.

The lang-addon.xml file is located in the classpath at /metainfo/zk/lang-addon.xml. In the lang-addon.xml file, the tag name of the new component is determined and the class name of the component and its corresponding widget:

```
<language-addon>
 <addon-name>geomajasmap</addon-name>
 <language-name>xul/html</language-name>
 <component>
  <component-name>geomajasmap</component-name>  <!-- this is the tag name -->
  <component-class>com.xxx.GeomajasComponent</component-class> <!-- this is the
  <widget-class>com.xxx.GeomajasWidget</widget-class> <!-- this the 'javascript
 </component>
```

```
</language-addon>
```

The dsp file is optional and contains some javascript code to write out the HTML representation of the widget. We refer to the general ZK component development documentation [http://books.zkoss.org/wiki/ZK_Component_Development_Essentials] for more information on this file.

The widget file defines a javascript class (or prototype) by extending the ZK widget class and is located in the classpath at web/js/com/xxx/GeomajasWidget.js. This file has the following content:

```
com.xxx.GeomajasWidget = zk.$extends(zk.Widget, {

 bind_ : function(evt) {
  this.$supers('bind_', arguments);
  // save a reference to this instance for later access by GWT
  this.$class._refs[this.uuid] = this;
 },

 unbind_ : function(evt) {
  this.$supers('unbind_', arguments);
 },

 redraw : function(out) {
  out.push('<div', this.domAttrs_(), '></div>');
 }
},  {
  // a static map of (id, instance) pairs, accessible by GWT
  _refs : {}
});
```

The bind_() and unbind_() functions are called to after the widget is attached/detached to the dom tree. It seems a good choice to use the bind_() function as the initialization point of the wrapped GWT component. We would normally just create the GWT component at this point, but, because of a race condition between the ZK and GWT javascript loads, the GWT part may not be loaded yet. For that reason, we register a reference to the component in a static map and let this reference be picked up and initialized later on by the GWT code. The redraw() function simply creates an empty div with the correct id.

The component file is a server side java class that extends the ZK XulElement class. Its main purpose is to act as a relay to the client part:

```
public class GeomajasComponent extends XulElement {

 static {
  addClientEvent(GeomajasComponent.class, FeatureSelectedEvent.NAME, CE_IMPORTA
  ...
 }

 private String featureId;

 // server-to-client communication
 public void panToFeature(String featureId) {
  if (featureId != null && !featureId.equals(this.featureId)) {
   this.featureId = featureId;
   smartUpdate("panToFeature", featureId); // send the update to the client
  }
 }

 @Override
 protected void renderProperties(ContentRenderer renderer) throws java.io.IOExc
  super.renderProperties(renderer);
  render(renderer, "panToFeature", featureId); // mandatory by ZK for batch upd
 }

 // client-to-server communication
 @Override
 public void service(AuRequest request, boolean everError) {
  final String cmd = request.getCommand();

  if (cmd.equals(FeatureDeselectedEvent.NAME)) {
   FeatureDeselectedEvent evt = FeatureDeselectedEvent.getEvent(request);
   Events.postEvent(evt); // post as an application-wide event
  } ...
  else {
   super.service(request, everError);
  }
 }

}
```

The panToFeature() method can be called from other ZK server components to let the map pan to a certain feature. The command is forwarded to the client by using the smartUpdate() method. The way ZK communicates with its client objects or widgets is by invoking setters on the client object. This makes sense for a light weight client object but is rather awkward from a GWT viewpoint. In this case, the setPanToFeature() function will be called on the GeomajasWidget javascript object. We will now make sure that this function directly calls an equivalent function on the GWT component:

```
public class GeomajasMapWrapper {

 private JavaScriptObject object;

 private MapWidget map;

 public GeomajasMapWrapper(JavaScriptObject object, String id) {
  this.object = object;
  ...  // create and attach the map to DOM element with the specified id (using
  initProperties(object);
 }

 public void panToFeature(String args) {
      // pan to feature implementation
 }

 public native void initProperties(JavaScriptObject object) /*-{
  var _this = this;
  object.setPanToFeature = function (featureId) {
   _this.@com.xxx.client.zk.GeomajasMapWrapper::panToFeature(Ljava/lang/String;
  };
 }-*/;

 public native void forwardFeatureSelected(JavaScriptObject object, String feat
  object.fire("onFeatureSelected",{"featureId" : featureId});
 }-*/;

 ...
 public static native void init() /*-{
  var maps = $wnd.com.xxx.GeomajasWidget._refs;
  for(var uuid in maps) {
   var cmp = maps[uuid];
   @com.xxx.client.zk.GeomajasMapWrapper::new(Lcom/google/gwt/core/client/JavaS
  }

  // override the bind function in case GWT was loaded first !
  $wnd.com.keyobs.urbagis.GeomajasWidget.prototype.bind_ = function(evt) {
   this.$supers('bind_', arguments);
   @com.xxx.client.zk.GeomajasMapWrapper::new(Lcom/google/gwt/core/client/JavaS
  };
 }-*/;
```

There are multiple JSNI methods in this wrapper class to communicate between the GWT and ZK worlds:

The init() method should be called once. It initializes registered references in the GeomajasWidget class by calling the constructor of the GeomajasWrapper class itself. It then overrides the bind_() function to make sure that any calls to bind_() will have the same effect. This is important in case the GWT part was loaded before bind_() was called by the ZK framework.

The initProperties() method will add a relay-function to the ZK widget with signature setPanToFeature(). It will forward the smartUpdate() from the ZK framework to the GWT method panToFeature() of our GWT class.

The forwardFeatureSelected() method will fire an event from the ZK client to the ZK server. This method can be called when a feature is selected on the map.

3. Bidirectional communication between the two frameworks

Communicating from GWT to ZK takes the following steps : GWT event -> JSNI call to ZK javascript -> server command -> ZK component -> ZK event.

The following code parts are required:

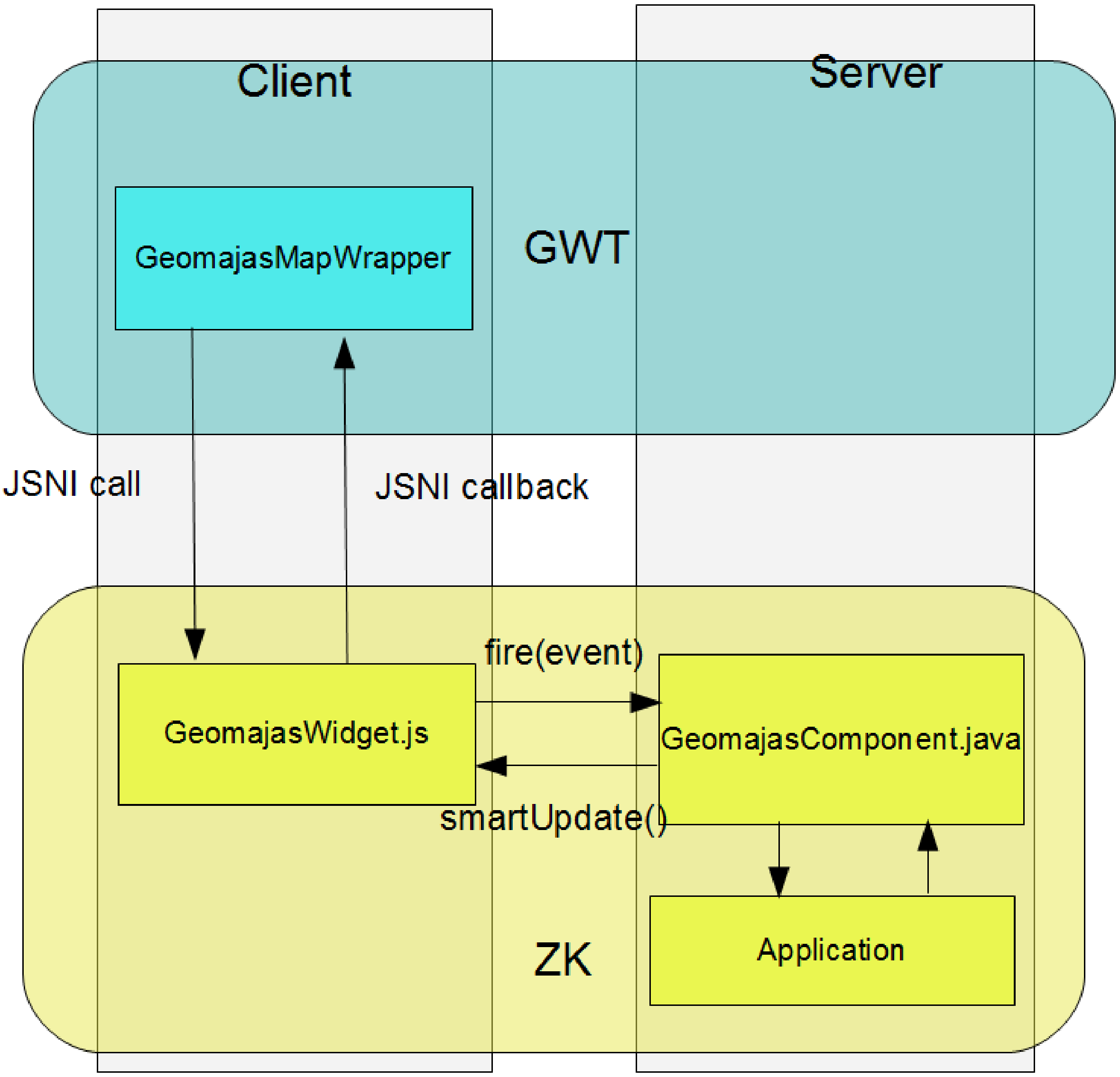
- add event listener to GWT widget
- add JSNI call to GWT widget to forward the event
- add javascript function to ZK javascript to send the ZK command
- process the ZK command in the ZK component
- fire application event in ZK component

Communicating from ZK to GWT takes the following steps : ZK java call -> ZK component -> smartupdate -> ZK JSNI callback -> GWT call.

The following code parts are required:

- add event listener to ZK component
- add update method to ZK component
- add ZK callback function to GWT component
- process the ZK callback in the GWT component

Figure 21.1. Bidirectional communication between ZK and GWT



Part VI. Appendices

Table of Contents

Appendix A. Migrating between Geomajas versions

1. 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>
  ```

2. 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.xml
          WEB-INF/applicationContext.xml
      </param-value>
  </context-param>

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

  root context for geomajas
  additional context for your application
  assures the application context is available
  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.

3. 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*:META-INF/geomajasWebContext.xml</param-value>
          <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.

4. 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`).

5. 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

Name	Property	Description
	layerInfo	Should no longer be set manually, will be set by Spring

6. 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.ShapeIn
      <property name="url" value="classpath:shapes/africa/country.shp"/>
  </bean>
  <bean name="countries" class="org.geomajas.internal.layer.layertree.DefaultVe
      <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.AllowAllSecurityServi
          </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).

6.1. 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`.

6.2. 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)

Name	Property	Action or description
	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 hereafter).
- 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.

The client or face classes are largely new and have been relocated to the `org.geomajas.configuration.client` package. 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.4. Client configuration

Name	Property	Action or description
ClientApplicationInfo	name	removed
ClientMapInfo	maxBounds	replaces MapInfo, optional maximum extent of the map, if present it will be used instead of the union of the layers' maximum extent
ClientLayerInfo	label	moved from LayerInfo
	visible	moved from LayerInfo
	viewScaleMin, viewScaleMax	moved from LayerInfo
	layerInfo	reference to back-end LayerInfo
	maxExtent	transformed extent from back-end
ClientVectorLayerInfo	snappingRules	moved from VectorLayerInfo

Name	Property	Action or description
	namedStyleInfo	The style to apply on the layer. Should be a reference to one of the back-end layer's predefined styles (see VectorLayerInfo).
	creatable, updatable, deletable	moved from VectorLayerInfo
	featureInfo	optional replacement of the back-end layer's FeatureInfo. If present, it is used instead.
ClientRasterLayerInfo	style	moved from ClientRasterLayerInfo
ClientLayerTreeInfo	*	rename of LayerTreeInfo, same properties
ClientLayerTreeNodeInfo	*	rename of LayerTreeNodeInfo
	layers	list of ClientLayerInfo objects, replaces previous list of layer ids
	expanded	changed from string to boolean
ClientToolbarInfo	*	rename of ToolbarInfo
ClientToolInfo	*	rename of ToolInfo

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.

7. Migrating from Geomajas 1.5.1 to Geomajas 1.5.2

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

8. 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.

9. 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.