

Geomajas static security plug-in

Geomajas Developers and Geosparc

Geomajas static security plug-in

by Geomajas Developers and Geosparc

1.9.0

Copyright © 2010-2012 Geosparc nv

Table of Contents

1. Introduction	1
2. Configuration	2
1. Dependencies	2
2. Use staticsecurity security service	2
3. Configure users	3
3.1. Static configuration of users	3
3.2. Dynamic configuration of users	4
3.3. Using LDAP to store user information	4
4. Configure policies	6
4.1. Base layer policies policies	7
4.2. Configure policies based on an area or geometry	8
4.3. Configure policies using a layer filter	9
4.4. Configure policies for individual features	10
4.5. Configure policies for individual attributes	11
5. Configure token validity	12
3. GWT face extensions	14
1. Dependencies	14
2. Using the GWT face widgets	14
2.1. TokenRequestWindow	14
2.2. TokenReleaseButton	15
4. How-to	16
1. Add custom policies with a custom SecurityContext	16
1.1. Define your policy	16
1.2. Implement your policy	16
1.3. Build your security context	18
1.4. Build your security manager	18
1.5. Wire it together	19
2. Using a custom policy client-side	20
3. Evaluating directly configured users first	23

List of Figures

3.1. Token request window	15
---------------------------------	----

List of Examples

2.1. Enabling the use of staticsecurity	3
2.2. Configure users	4
2.3. Configuring a custom authentication service	4
2.4. Configuration to use LDAP to authenticate users	6
2.5. Base layer policies	7
2.6. Command policies	7
2.7. Tool policies	8
2.8. Layer policies	8
2.9. Area policies	9
2.10. Filter on layer policy	10
2.11. Feature specific policy	11
2.12. Attribute specific policy	12
2.13. Schedule cleanup of token list	13
3.1. Include the plug-in in your .gwt.xml file	14
3.2. Set the token request handler	14
4.1. Define a BaseAuthorization for your policy	16
4.2. Configuration for your authorization	17
4.3. Implementation for your authorization	17
4.4. Custom security context, combine the authorizations	18
4.5. Security manager for new security context	19
4.6. Replace security context by version which includes custom policies	19
4.7. A configuration using the custom policy	20
4.8. Example of a client security context	21
4.9. Replace the client configuration loader	21
4.10. Replacement command to read the configuration	22
4.11. Request object for replacement command	22
4.12. Response object for replacement command	23
4.13. Evaluating configured users first	23

Chapter 1. Introduction

The staticsecurity plug-in allows you to define the policies which apply for certain users. The policies are statically defined in a XML file (hence the name). These policies can be linked to users which can also defined in the XML file or which are obtained through code (for example linking to a user database or LDAP store).

This plug-in also includes an example which adds a custom policy and makes that available in the SecurityContext.

Chapter 2. Configuration

The configuration of staticsecurity involves the following elements:

- configure the use of staticsecurity as security service.
- configure the way to access the users.
- configure the policies to use.

1. Dependencies

Make sure you include the correct version of the plug-in in your project. This can be done either by including a reference to geomajas-dep or the following excerpt (with the correct version) in the dependencyManagement section of your project:

```
<dependency>
  <groupId>org.geomajas.plugin</groupId>
  <artifactId>geomajas-plugin-staticsecurity-all</artifactId>
  <version>1.9.0</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

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

You can now include the actual dependency without explicit version. You will probably need the dependency for the face you are using. In case of the GWT face, this is:

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

If there is no specific module for the face you are using, you should include the core module:

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

In case your users are stored in LDAP, you will need to include a dependency on the LDAP module:

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

2. Use staticsecurity security service

To enable the use of the staticsecurity security service, you have to replace the "security.securityInfo" bean with your own version. When such a bean is not defined, nobody is allowed to do anything.

Example 2.1. Enabling the use of staticsecurity

```
<bean name="security.securityInfo" class="org.geomajas.security.SecurityInfo">
  <property name="loopAllServices" value="false"/>
  <property name="securityServices">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.security.StaticSecurityService">
      <bean class="org.geomajas.plugin.staticsecurity.security.LoginAllowedSecurityService">
    </list>
  </property>
</bean>
```

The configuration actually defines two services. The second (LoginAllowedSecurityService) assures that everybody is allowed access to the login command which is provided by the plug-in.

The actual staticsecurity security service requires you to define a bean of the class `org.geomajas.plugin.staticsecurity.configuration.SecurityServiceInfo` to be defined in your application context. All configuration is read from that bean. Details are given in the following sections.

3. Configure users

Users can be configured either statically or dynamically.

3.1. Static configuration of users

In the bean of type `SecurityServiceInfo` you can define the users which exist. This allows only password based authentication.

The `userId` is also used as login. You can add a couple of extra fields to give information about the user.

The password needs to be specified as a base64 encoded MD5 hash. The string to be hashed is a concatenation of

- "Geomajas is a wonderful framework" (without quotes)
- user id
- actual password

There is an online tool which can be used to calculate this has as <http://progs.be/md5.html>.

The `authorizations` property allow you to specify the policies, indicating what is allowed for the user.

Example 2.2. Configure users

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServiceIn
  <property name="users">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.configuration.UserI
        <property name="userId" value="luc"/>
        <property name="password" value="b7NMSP1pZN3Hi6nJGVe9JA"/> <!--
        <property name="userName" value="Luc Van Lierde"/>
        <property name="userOrganization" value="triathlon"/>
        <property name="userDivision" value="all distances"/>
        <property name="userLocale" value="nl_BE"/>
        <property name="authorizations">
          <list>
            <!-- ... include authorizations / policies here -->
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

3.2. Dynamic configuration of users

The user information can also be determined dynamically by using explicitly configuring authentication services. The statically defined users (see previous section) are always automatically included after the configured authentication services.

Below is a simple example from the tests which shows a configuration of an authentication service.

Example 2.3. Configuring a custom authentication service

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServiceIn
  <property name="authenticationServices">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.general.CustomAuther
        <property name="userId" value="custom"/>
        <property name="password" value="custom"/>
        <property name="authorizationInfo">
          <bean class="org.geomajas.plugin.staticsecurity.configurati
            <property name="commandsInclude">
              <list>
                <value>.*</value>
              </list>
            </property>
          </bean>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

3.3. Using LDAP to store user information

Below is an example configuration for obtaining user information from an LDAP store.

You have to configure the host and port for the LDAP server. The template to build the user DN¹ from the user id needs to be given. In the template the pair of braces ("{}") is replaced by the actual login. You can also define the attributes which are used to store the various pieces of user information. The given name and surname are concatenated (separated by a space) when both have a value.

The roles property is used to define the authorizations for the roles which are configured in the LDAP store. The keys are the values of the roles attribute, often DN values. You can also define the defaultRole. This is the set of authorizations which is always assigned to all users who can authenticate using LDAP (the authorizations for the roles are added to that set).

¹distinguished named

Example 2.4. Configuration to use LDAP to authenticate users

```

<bean name="security.securityInfo" class="org.geomajas.security.SecurityInfo"
  <property name="loopAllServices" value="false"/>
  <property name="securityServices">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.security.StaticSecurityService" />
      <bean class="org.geomajas.plugin.staticsecurity.security.LoginAuthService" />
    </list>
  </property>
</bean>

<bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServiceConfiguration"
  <property name="authenticationServices">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.ldap.LdapAuthService"
        <property name="serverHost" value="localhost"/>
        <property name="serverPort" value="3636" />
        <property name="userDnTemplate" value="cn={},dc=staticsecurity,dc=org" />
        <property name="givenNameAttribute" value="givenName" />
        <property name="surNameAttribute" value="sn" />
        <property name="localeAttribute" value="locale" />
        <property name="organizationAttribute" value="o" />
        <property name="divisionAttribute" value="ou" />
        <property name="rolesAttribute" value="memberOf" />
        <property name="defaultRole">
          <list>
            <bean class="org.geomajas.plugin.staticsecurity.configuration.DefaultRole"
              <property name="toolsInclude">
                <list><value>.*</value></list>
              </property>
            </bean>
          </list>
        </property>
        <property name="roles">
          <map>
            <entry key="cn=testgroup,dc=roles,dc=geomajas,dc=org">
              <list>
                <bean class="org.geomajas.plugin.staticsecurity.configuration.DefaultRole"
                  <property name="commandsInclude">
                    <list><value>.*</value></list>
                  </property>
                </bean>
              </list>
            </entry>
          </map>
        </property>
      </bean>
    </list>
  </property>
</bean>

```

4. Configure policies

The policies are configured by providing a list of authentications. Several ways exist, but they all start from the base layer policies.

4.1. Base layer policies policies

The base policies which always need to be provided include

- which commands can be executed
- which tools can be included in the UI
- CRUD rights which are available for the layers

For each of these policies, you can use regular expressions to indicate either what to include or what to exclude. By default you have no rights. You have to indicate what is included and can then refine that list by excluding some stuff from what was included.

The base configuration to allow everything is displayed below:

Example 2.5. Base layer policies

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.LayerAuthorization"
  <property name="commandsInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="toolsInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="visibleLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="updateAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="createAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="deleteAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
</bean>
```

You can explicitly configure which commands can be executed by that user.

Example 2.6. Command policies

```
<property name="commandsInclude">
  <list>
    <value>command.MarinoLoggedIn</value>
  </list>
</property>
```

You can also specify the tools which are allowed to be displayed in the UI.

Example 2.7. Tool policies

```
<property name="toolsInclude">
  <list>
    <value>.*</value>
  </list>
</property>
<property name="toolsExclude">
  <list>
    <value>Zoom.*</value>
  </list>
</property>
```

Importantly, you can configure the CRUD rights for each of the layers. You can specify for each layer, using include and exclude regular expressions, whether the user

- can view the layer
- add features in the layer
- edit features for the layer
- delete features from the layer

Example 2.8. Layer policies

```
<property name="visibleLayersInclude">
  <list>
    <value>roads</value>
    <value>rivers</value>
  </list>
</property>
<property name="createAuthorizedLayersExclude">
  <list>
    <value>.*</value>
  </list>
</property>
<property name="updateAuthorizedLayersExclude">
  <list>
    <value>.*</value>
  </list>
</property>
<property name="deleteAuthorizedLayersExclude">
  <list>
    <value>.*</value>
  </list>
</property>
```

4.2. Configure policies based on an area or geometry

Using `AreaAuthorizationInfo` you can make your policies more granular by defining the area in which the CRUD operations are allowed. You first have to define the normal layer rights. You can then make this more specific by setting the geometry for the rights in the layers property, setting the area for each of the CRUD operations in for the specific layer. The actual geometry is specified using WKT [http://en.wikipedia.org/wiki/Well-known_text].

Example 2.9. Area policies

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.AreaAuthorization"
  <property name="visibleLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="updateAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="createAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="deleteAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="layers">
    <map>
      <entry key="beans">
        <bean class="org.geomajas.plugin.staticsecurity.configuration.L"
          <property name="visibleArea"
            value="MULTIPOLYGON(((1 0,1 10,10 10,10 0,1 0)))"
          <property name="updateAuthorizedArea"
            value="MULTIPOLYGON(((4 0,4 10,10 10,10 0,4 0)))"
          <property name="createAuthorizedArea"
            value="MULTIPOLYGON(((1 0,1 5,10 5,10 0,1 0)))" /
          <property name="deleteAuthorizedArea"
            value="MULTIPOLYGON(((1 0,1 2,10 2,10 0,1 0)))" /
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

4.3. Configure policies using a layer filter

Using the `LayerFilterAuthorizationInfo` authorization bean you can add a ECQL filter to a layer to limit the features which can be read.

This is done using the `filters` property which contains a map where the key is the (server) layers id and the value is the filter string.

Example 2.10. Filter on layer policy

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.LayerFilterAuthor
  <property name="visibleLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="filters">
    <map>
      <entry key="beans" value="stringAttr='bean2'"/>
    </map>
  </property>
</bean>
```

4.4. Configure policies for individual features

You can also be specific about which features in a layer are accessible by the user. This can be done using the `FeatureAuthorizationInfo` authorization bean.

Using the `layers` property, you can determine which CRUD rights are allowed for which features. The property contains a map with the (server) layer id as key and a `LayerFeatureAuthorizationInfo` bean as value. This allows you to select the features using include and exclude rules on the feature id. As this only allows you to select on feature ids, consider this bean as an example for building more powerful feature selection policies.

Example 2.11. Feature specific policy

```

<bean class="org.geomajas.plugin.staticsecurity.configuration.FeatureAuthorizat
  <property name="visibleLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="layers">
    <map>
      <entry key="beans">
        <bean class="org.geomajas.plugin.staticsecurity.configuration.L
          <property name="createAuthorized" value="false"/>
          <property name="visibleIncludes">
            <list>
              <value>.*</value>
            </list>
          </property>
          <property name="visibleExcludes">
            <list>
              <value>2</value>
            </list>
          </property>
          <property name="updateAuthorizedIncludes">
            <list>
              <value>.*</value>
            </list>
          </property>
          <property name="updateAuthorizedExcludes">
            <list>
              <value>2</value>
              <value>3</value>
            </list>
          </property>
          <property name="deleteAuthorizedIncludes">
            <list>
              <value>1</value>
            </list>
          </property>
        </bean>
      </entry>
    </map>
  </property>
</bean>

```

4.5. Configure policies for individual attributes

The most specific access policies can be built using the `AttributeAuthorizationInfo` authorization bean. This allows you to specify which attributes can be used for specific features.

Using the `layers` property, you can determine which CRUD rights are allowed for which features. The property contains a map with the (server) layer id as key and a `LayerAttributeAuthorizationInfo` bean as value. This allows you to select the attributes using include and exclude rules on the attribute name. You can add a `"@featureId"` suffix to the regular expression to limit the attribute to a features (using a regular expression on the feature id). As this only allows you to select on feature ids, consider this bean as an example for building more powerful attribute selection policies.

Example 2.12. Attribute specific policy

```

<bean class="org.geomajas.plugin.staticsecurity.configuration.AttributeAuthoriz
  <property name="visibleLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="updateAuthorizedLayersInclude">
    <list>
      <value>.*</value>
    </list>
  </property>
  <property name="layers">
    <map>
      <entry key="beans">
        <bean class="org.geomajas.plugin.staticsecurity.configuration.L
          <property name="readableIncludes">
            <list>
              <value>.*</value>
            </list>
          </property>
          <property name="readableExcludes">
            <list>
              <value>booleanAttr</value>
              <value>stringAttr@2</value>
            </list>
          </property>
          <property name="writableIncludes">
            <list>
              <value>.*</value>
            </list>
          </property>
          <property name="writableExcludes">
            <list>
              <value>stringAttr</value>
              <value>integerAttr@2</value>
            </list>
          </property>
        </bean>
      </entry>
    </map>
  </property>
</bean>

```

5. Configure token validity

Tokens have a default lifespan of four hours. You can change this using the `tokenLifetime` property in `SecurityServiceInfo`. When a token is used after it has expired, it will be removed from the list of available tokens and the token is invalid. By default the list of available tokens is not automatically purged. However, this can easily be done using a little bit of Spring configuration by adding the following configuration.

Example 2.13. Schedule cleanup of token list

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:task="http://www.springframework.org/schema/task"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/sche
http://www.springframework.org/schema/context http://www.springframework.org/sc
http://www.springframework.org/schema/task http://www.springframework.org/schem

    <task:annotation-driven />

</beans>
```

When this is enabled, the list of available tokens are purged every minute.

Chapter 3. GWT face extensions

The plug-in contains a module which provides some widgets for the GWT face and which allows automatic handling of login using the staticsecurity login window.

1. Dependencies

When using the GWT face extensions, you should depend on the module which provides these. This dependency automatically includes all necessary source code for GWT compilation and the core staticsecurity module.

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

This assumes you are using either geomajas-dep or that you important the staticsecurity-all pom in your dependencyManagement section as discussed in section Section 1, “Dependencies”.

2. Using the GWT face widgets

To make sure the widgets can be used in your GWT client code, you have to inherit the module in you .gwt.xml file.

Example 3.1. Include the plug-in in your .gwt.xml file

```
<inherits name="org.geomajas.plugin.staticsecurity.StaticSecurity"/>
```

You can now configure to use the staticsecurity token request handler. This will assure that credentials are asked when the current token is either invalid or when a security exception occurs if no token was available.

Example 3.2. Set the token request handler

```
GwtCommandDispatcher.getInstance().setTokenRequestHandler(
    new StaticSecurityTokenRequestHandler(
        "Possible users are 'luc' and 'marino'. The password is the same as the
```

2.1. TokenRequestWindow

The token request window allows your users to login to the application. There are two fields to enter the login and password and a login and reset button. When the users enters his or her credentials, the user is logged in. When a problem occurs, this is displayed.

The window can display a custom message. This can be passed in the constructor. The SsecLayout class can be used to customize the layout and look (for example the logo).

Figure 3.1. Token request window

2.2. TokenReleaseButton

The token release button allows the user to logout by pressing the button. This will contact the server to immediately invalidate the current token, thus logging the user out. When the application requires you to login, new credentials will be asked the next time the server is contacted. The button listens to the token changed events to control its state, disabling it when no token is available and enabling it after login.

Chapter 4. How-to

1. Add custom policies with a custom SecurityContext

To add custom policies in the security context, you need the following steps:

- Define your custom policy.
- Provide an implementation for your custom policy.
- Build a security context which can combine different authorizations which contain the new policy.
- Build a security manager which uses the new security context.
- Wire it all together.

Let's go over this using a practical example. In our example application, we want a "blabla" button which should not be visible to all users.

The way it is handled here allows you to query the new policy using your autowired security context.

```
@Autowired
private AppSecurityContext securityContext
```

1.1. Define your policy

You always have to define your policy in as an interface which extends BaseAuthorization. This is the base interface for all types of authorization objects. As you extend BaseAuthorization, you will always have to include handling of basic security on layers.

We just want to add a method to determine if using the "blabla" button is allowed.

Example 4.1. Define a BaseAuthorization for your policy

```
public interface AppAuthorization extends BaseAuthorization {

    /**
     * Does the user have access to the "blabla" button?
     *
     * @return true when button is allowed
     */
    boolean isBlablaButtonAllowed();

}
```

1.2. Implement your policy

To be able to implement our new authorization object we also need to provide a way to configure the value for specific users or roles. For this example, we build on the layer configuration object which is provided by the staticsecurity plug-in.

Apart from the methods which are used to get and set the extra values in the configuration, there is also a getAuthorizatn() method which is overwritten. This is used by the staticsecurity security service to build the actual authorization objects from the configuration.

Example 4.2. Configuration for your authorization

```
public class AppAuthorizationInfo extends LayerAuthorizationInfo {

    private boolean blablaButtonAllowed;

    /**
     * Set whether the blabla button is allowed.
     *
     * @param blablaButtonAllowed is blabla button allowed
     */
    public void setBlablaButtonAllowed(boolean blablaButtonAllowed) {
        this.blablaButtonAllowed = blablaButtonAllowed;
    }

    /**
     * Does the user have access to the "blabla" button?
     *
     * @return true when button is allowed
     */
    boolean isBlablaButtonAllowed() {
        return blablaButtonAllowed;
    }

    @Override
    public BaseAuthorization getAuthorization() {
        return new AppAuthorizationImpl(this);
    }
}
```

The actual implementation is quite easy on our case. Just use the configuration object to return the correct value for the new policy.

It is important to have a no-args constructor in this class as it is needed to deserialize the authorization when needed (which is for example used in rasterizing).

Example 4.3. Implementation for your authorization

```
public class AppAuthorizationImpl extends LayerAuthorization implements AppAutho

    private AppAuthorizationInfo appAuthorizationInfo;

    protected AppAuthorizationImpl() {
        // for deserialization
        super();
    }

    public AppAuthorizationImpl(AppAuthorizationInfo info) {
        super(info);
        appAuthorizationInfo = info;
    }

    public boolean isBlablaButtonAllowed() {
        return appAuthorizationInfo.isBlablaButtonAllowed();
    }
}
```

1.3. Build your security context

We need to build a new security context implementation which also handles our additional policy. We do this by extending the default implementation. Note that you need to add `geomajas-impl` as a project dependency to compile this (but rest assured, being able to extend `DefaultSecurityContext` is assured by the API contract).

Example 4.4. Custom security context, combine the authorizations

```
@Scope(value = "thread", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class AppSecurityContext extends DefaultSecurityContext implements AppAuthenticating {

    // new authorization

    public boolean isBlablaButtonAllowed() {
        boolean allowed = false;
        for (Authentication authentication : getSecurityServiceResults()) {
            for (BaseAuthorization authorization : authentication.getAuthorizations()) {
                if (authorization instanceof AppAuthorization) {
                    allowed |= ((AppAuthorization) authorization).isBlablaButtonAllowed();
                }
            }
        }
        return allowed;
    }
}
```

1.4. Build your security manager

The security context is created and set by the security manager, so you also have to replace the security manager to make sure it uses your security context.

Example 4.5. Security manager for new security context

```
public class AppSecurityManager extends DefaultSecurityManager {

    @Autowired
    private SecurityContext securityContext;

    /** @inheritDoc */
    public boolean setSecurityContext(String authenticationToken, List<Authenticati
        if (!authentications.isEmpty()) {
            // build authorization and build thread local SecurityContext
            ((AppSecurityContext) securityContext).setAuthentications(authentic
            return true;
        }
        return false;
    }

    /** @inheritDoc */
    public void clearSecurityContext() {
        ((AppSecurityContext) securityContext).setAuthentications(null, null);
    }

    /** @inheritDoc */
    public void restoreSecurityContext(SavedAuthorization authorizations) {
        if (null == authorizations) {
            clearSecurityContext();
        } else {
            ((AppSecurityContext) securityContext).restoreSecurityContext(autho
        }
    }
}
```

1.5. Wire it together

You have to make sure that spring know about your new security manager and context. You can do this by including the following excerpt in your application context. It is very important to include the scope and scoped-proxy bits in the security context declaration. That annotations which are included on the class are not used when using the bean style declaration, so you have to add them explicitly. If you forget this, you will get spurious errors using the wrong context.

Example 4.6. Replace security context by version which includes custom policies

```
<bean name="security.SecurityManager"
      class="org.geomajas.plugin.staticsecurity.gwt.example.server.security.App
<bean name="security.SecurityContext" scope="thread"
      class="org.geomajas.plugin.staticsecurity.gwt.example.server.security.App
    <aop:scoped-proxy/>
</bean>
```

You can create a configuration which uses your new policy.

Example 4.7. A configuration using the custom policy

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServiceIn
  <property name="tokenLifetime" value="600" /> <!-- ten minutes -->
  <property name="users">
    <list>
      <bean class="org.geomajas.plugin.staticsecurity.configuration.UserI
        <property name="userId" value="luc"/>
        <property name="password" value="b7NMSP1pZN3Hi6nJGVe9JA"/> <!--
        <property name="userName" value="Luc Van Lierde"/>
        <property name="authorizations">
          <list>
            <bean class="org.geomajas.plugin.staticsecurity.gwt.exa
              <property name="commandsInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="toolsInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="visibleLayersInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="updateAuthorizedLayersInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="createAuthorizedLayersInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="deleteAuthorizedLayersInclude">
                <list>
                  <value>.*</value>
                </list>
              </property>
              <property name="blablaButtonAllowed" value="true" />
            </bean>
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

2. Using a custom policy client-side

If you want to have access to some of your custom policies on the client side, then you have to get these from the server on startup.

The best way to handle this is to replace the command to get the initial configuration with a version which also includes your policy. Then on the response you can set the policy in a class to allow you to access the information.

For starters, let's create the client security context class. This is a static class which just contains accessors for the `blablaButtonAllowed` property.

Example 4.8. Example of a client security context

```
public final class ClientSecurityContext {

    private static boolean blablaButtonAllowed;

    private ClientSecurityContext() {
        // utility class, hide constructor
    }

    /**
     * Is it allowed to push the "blabla" button?
     *
     * @return true when allowed
     */
    public static boolean isBlablaButtonAllowed() {
        return blablaButtonAllowed;
    }

    /**
     * Set whether using the "blabla" button is allowed.
     *
     * @param blablaButtonAllowed allowed?
     */
    public static void setBlablaButtonAllowed(boolean blablaButtonAllowed) {
        ClientSecurityContext.blablaButtonAllowed = blablaButtonAllowed;
    }
}
```

The value will be set when the configuration is read. This is automatically re-read when the security token changes. The advantage of combining this with the get configuration command is that only one command needs to be called. Use code like the following in your entry point class before the map widget is created.

Example 4.9. Replace the client configuration loader

```
ClientConfigurationService.setConfigurationLoader(new ClientConfigurationLoader() {
    public void loadClientApplicationInfo(final String applicationId, final ClientConfigurationRequest commandRequest) {
        GwtCommand commandRequest = new GwtCommand(AppConfigurationRequest.COMMAND_NAME);
        commandRequest.setCommandRequest(new AppConfigurationRequest(applicationId));
        GwtCommandDispatcher.getInstance().execute(commandRequest,
            new AbstractCommandCallback<AppConfigurationResponse>() {
                public void execute(AppConfigurationResponse response) {
                    ClientSecurityContext.setBlablaButtonAllowed(response.isBlablaButtonAllowed());
                    ClientSecurityContext.setBlablaButtonAllowed(response.isBlablaButtonAllowed());
                    setter.set(applicationId, response.getApplication());
                }
            });
    }
});
```

This snippet uses a custom command which extends the standard `GetConfigurationCommand`. The code is shown below.

Example 4.10. Replacement command to read the configuration

```
@Component
public class AppConfigurationCommand implements Command<AppConfigurationRequest>

    @Autowired
    private CommandDispatcher commandDispatcher;

    @Autowired
    private AppSecurityContext securityContext;

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

    public void execute(AppConfigurationRequest request, AppConfigurationResponse response) {
        GetConfigurationResponse original = (GetConfigurationResponse) commandDispatcher.execute(
            GetConfigurationRequest.COMMAND, request, securityContext.getToApplicationId());
        response.setApplication(original.getApplication());

        // Add app specific configuration
        response.setBlablaButtonAllowed(securityContext.isBlablaButtonAllowed());
    }
}
```

The request and response objects also need to be redefined for this command. For the request object this is almost trivial. The most important change is to overwrite the `COMMAND` field.

Example 4.11. Request object for replacement command

```
public class AppConfigurationRequest extends GetConfigurationRequest {

    private static final long serialVersionUID = 100L;

    public static final String COMMAND = "command.AppConfiguration";

    public AppConfigurationRequest(String applicationId) {
        super();
        setApplicationId(applicationId);
    }

    public AppConfigurationRequest() {
        // for deserialization
        super();
    }
}
```

The response object needs some extra extensions to ensure that the custom policy information is included.

Example 4.12. Response object for replacement command

```
public class AppConfiguratioResponse extends GetConfigurationResponse {

    private static final long serialVersionUID = 100L;

    private boolean blablaButtonAllowed;

    public boolean isBlablaButtonAllowed() {
        return blablaButtonAllowed;
    }

    public void setBlablaButtonAllowed(boolean blablaButtonAllowed) {
        this.blablaButtonAllowed = blablaButtonAllowed;
    }
}
```

You can see this in action in the example application.

3. Evaluating directly configured users first

Normal behavior is that the users which are directly configured in the `SecurityServiceInfo` object users field are searched after the services which are configured in the `authenticationServices` field. This can be very useful for testing, for example when the remote service is not available (and the system waits for times). This can be changed by using a configuration like

Example 4.13. Evaluating configured users first

```
<bean class="org.geomajas.plugin.staticsecurity.configuration.SecurityServiceIn
    <!-- LDAP authentication AFTER configured users -->
    <property name="authenticationServices">
        <list>
            <bean class="org.geomajas.plugin.staticsecurity.security.StaticAuth
            <bean class="org.geomajas.plugin.staticsecurity.ldap.LdapAuthentica
                <!-- ..... LDAP settings -->
            </bean>
        </list>
    </property>
</bean>
```