



SPRING-SECURITY-CORE

spring-security-core - Reference Documentation

Authors: Burt Beckwith

Version: 0.1

Table of Contents

1. Introduction	3
1.1 Configuration	3
1.2 Migration from the Acegi plugin	3
2. Domain Classes	9
2.1 Person	9
2.2 Authority	10
2.3 PersonAuthority	11
2.4 Requestmap	12
3. Securing URLs	14
3.1 Annotations	15
3.2 Config.groovy	16
3.3 Requestmap	17
4. Helper Classes	18
4.1. Security Tags	18
4.2. SpringSecurityService	19
4.3. SpringSecurityUtils	22
5. Events	24
6. Configuration	25
7. Custom UserDetailsService	31
8. Ajax Authentication	33
9. Tutorials	36
9.1. Using Controller annotations to secure URLs	36
10. Extending and configuring the plugin	41
10.1. Filters	41
10.2. Basic and Digest Auth	42
10.3. Switch User	43
10.4. Session Fixation	45
10.5. Salted passwords	45
10.6. Certificate (X509) login	46
10.7. Channel security	47
10.8. IP Address Restrictions	47
10.9. Logout Handlers	48
10.10. Voters	48
10.11. Authentication Providers	49
10.12. Hierarchical Roles	49
10.13. Account Locking and Forcing Password Change	50

1. Introduction

The Spring Security plugin simplifies the work involved in integrating [Spring Security](#) (formerly called Acegi Security) into Grails applications. Spring Security versions 2 and 3 have made configuration a lot easier than it used to be with Acegi, but it's still a somewhat complex process. The plugin takes an approach similar to Grails in that it provides sensible defaults with many configuration options for customization. Nearly everything is configurable or replaceable in the plugin and in Spring Security itself (Spring Security makes extensive use of interfaces) so you can easily make whatever changes you need to support various options.

The plugin handles the steps required to register filters in `web.xml` for you, and also configures the Spring beans in the application context that implement various pieces of functionality. In addition, you don't need to deal with figuring out which jar files to use since that's handled by Ivy. So all you need to do is install the plugin, run the initialization script, and make any required configuration changes in `Config.groovy`.


One goal of the plugin is that you shouldn't need to know much about Spring Security to use it, but it can be helpful to understand the underlying implementation, so refer to [the documentation](#) if something doesn't make sense.

Getting started

See the [tutorials](#) section for details on getting started.

1.1 Configuration

The plugin is highly configurable, but hopefully most of the default settings should be fine. If you need to override a property, you can do that under `grails-app/conf/Config.groovy`. The earlier Acegi plugin used its own configuration file, `SecurityConfig.groovy` but this plugin maintains its configuration in the standard `Config.groovy` file. This enables environment-specific configuration, e.g. if you want less restrictive security rules when developing than when you deploy to production. Like any environment-specific config parameters, just wrap them in an `environments` block.



Note that the plugin's configuration values all start with `grails.plugins.springsecurity` to keep them separate from similarly named options in Grails or other plugins. In the documentation if you see an attribute such as `password.algorithm` remember that this would be specified as

```
grails.plugins.springsecurity.password.algorithm='SHA-512'
```

in `Config.groovy`

See [Chapter 6](#) for a detailed discussion about the various general configuration options and [Chapter 10](#) for options for specific features.

1.2 Migration from the Acegi plugin

This plugin is a successor to the original plugin that provided support for Spring Security, the [Acegi plugin](#). It's a new plugin that doesn't depend on that plugin but there are many similarities, so migrating is fairly straightforward.

Core differences

The Spring Security plugin retains many of the core features of the Acegi plugin:

- form-based authentication
- storing users, roles, and optionally requestmaps in the database and accessing via domain classes
- guarding URLs with annotations, requestmap domain class, or static configuration
- security tags
- security service
- security events
- Ajax login
- Basic auth
- Switch User
- Channel security
- IP Address Restrictions

and in addition adds several new features:

- Digest Auth
- Session Fixation
- Salted passwords
- Certificate (x509) login
- Hierarchical Roles
- Account Locking and Forcing Password Change

There are a few core concepts that have changed and will require configuration changes in your application:

Spring Security plugin		Acegi plugin
enabled by default	true	false
cache UserDetails by default	false	true
configuration location	grails-app/conf/Config.groovy	grails-app/conf/SecurityConfig.groovy
security service	springSecurityService	authenticateService

There are features that are not included but which will be available in secondary plugins that will extend and depend on the core plugin:

- Facebook
- OpenID
- LDAP
- CAS
- NTLM
- Kerberos
- User registration

Configuration differences

This table summarizes the configuration attribute names in both plugins:

Acegi plugin	Spring Security plugin
active	active
loginUserDomainClass	userLookup.userDomainClassName
userName	userLookup.usernamePropertyName
enabled	userLookup.enabledPropertyName
password	userLookup.passwordPropertyName
relationalAuthorities	userLookup.authoritiesPropertyName
getAuthoritiesMethod	N/A
authorityDomainClass	authority.className
authorityField	authority.nameField
authenticationFailureUrl	failureHandler.defaultFailureUrl
ajaxAuthenticationFailureUrl	failureHandler.ajaxAuthFailUrl
defaultTargetUrl	successHandler.defaultTargetUrl
alwaysUseDefaultTargetUrl	successHandler.alwaysUseDefault
filterProcessesUrl	apf.filterProcessesUrl
key	anon.key
userAttribute	anon.userAttribute

loginFormUrl	auth.loginFormUrl
forceHttps	auth.forceHttps
ajaxLoginFormUrl	auth.ajaxLoginFormUrl
afterLogoutUrl	logout.afterLogoutUrl
errorPage	adh.errorPage
ajaxErrorPage	adh.ajaxErrorPage
ajaxHeader	ajaxHeader
algorithm	password.algorithm
encodeHashAsBase64	password.encodeHashAsBase64
cookieName	rememberMe.cookieName
alwaysRemember	rememberMe.alwaysRemember
tokenValiditySeconds	rememberMe.tokenValiditySeconds
parameter	rememberMe.parameter
rememberMeKey	rememberMe.key
useLogger	registerLoggerListener
useRequestMapDomainClass	securityConfigType = SecurityConfigType.Requestmap
requestMapClass	requestMap.className
requestMapPathField	requestMap.urlField
requestMapConfigAttributeField	requestMap.configAttributeField
useControllerAnnotations	securityConfigType = SecurityConfigType.Annotation
controllerAnnotationsMatcher	controllerAnnotations.matcher
controllerAnnotationsMatchesLowercase	controllerAnnotations.lowercase
controllerAnnotationStaticRules	controllerAnnotations.staticRules
controllerAnnotationsRejectIfNoRule	rejectIfNoRule
requestMapString	N/A - securityConfigType = SecurityConfigType.InterceptUrlMap is very similar
realmName	basic.realmName
basicProcessingFilter	useBasicAuth
switchUserProcessingFilter	useSwitchUserFilter
swswitchUserUrl	switchUser.switchUserUrl
swexitUserUrl	switchUser.exitUserUrl
swtargetUrl	switchUser.targetUrl
useMail	N/A - registration will be supported in the UI plugin
mailHost	N/A - registration will be supported in the UI plugin
mailUsername	N/A - registration will be supported in the UI plugin
mailPassword	N/A - registration will be supported in the UI plugin
mailProtocol	N/A - registration will be supported in the UI plugin
mailFrom	N/A - registration will be supported in the UI plugin
mailPort	N/A - registration will be supported in the UI plugin
defaultRole	N/A - registration will be supported in the UI plugin
useOpenId	N/A - will be supported in the OpenID plugin

openIdNonceMaxSeconds	N/A - will be supported in the OpenID plugin
useLdap	N/A - will be supported in the LDAP plugin
ldapRetrieveGroupRoles	N/A - will be supported in the LDAP plugin
ldapRetrieveDatabaseRoles	N/A - will be supported in the LDAP plugin
ldapSearchSubtree	N/A - will be supported in the LDAP plugin
ldapGroupRoleAttribute	N/A - will be supported in the LDAP plugin
ldapPasswordAttributeName	N/A - will be supported in the LDAP plugin
ldapServer	N/A - will be supported in the LDAP plugin
ldapManagerDn	N/A - will be supported in the LDAP plugin
ldapManagerPassword	N/A - will be supported in the LDAP plugin
ldapSearchBase	N/A - will be supported in the LDAP plugin
ldapSearchFilter	N/A - will be supported in the LDAP plugin
ldapGroupSearchBase	N/A - will be supported in the LDAP plugin
ldapGroupSearchFilter	N/A - will be supported in the LDAP plugin
ldapUsePassword	N/A - will be supported in the LDAP plugin
useKerberos	N/A - will be supported in a secondary plugin
kerberosLoginConfigFile	N/A - will be supported in a secondary plugin
kerberosRealm	N/A - will be supported in a secondary plugin
kerberosKdc	N/A - will be supported in a secondary plugin
kerberosRetrieveDatabaseRoles	N/A - will be supported in a secondary plugin
useHttpSessionEventPublisher	useHttpSessionEventPublisher
cacheUsers	cacheUsers
useCAS	N/A - will be supported in the CAS plugin
cas.casServer	N/A - will be supported in the CAS plugin
cas.casServerPort	N/A - will be supported in the CAS plugin
cas.casServerSecure	N/A - will be supported in the CAS plugin
cas.localhostSecure	N/A - will be supported in the CAS plugin
cas.failureURL	N/A - will be supported in the CAS plugin
cas.defaultTargetURL	N/A - will be supported in the CAS plugin
cas.fullLoginURL	N/A - will be supported in the CAS plugin
cas.fullServiceURL	N/A - will be supported in the CAS plugin
cas.authenticationProviderKey	N/A - will be supported in the CAS plugin
cas.userDetailsService	N/A - will be supported in the CAS plugin
cas.sendRenew	N/A - will be supported in the CAS plugin
cas.proxyReceptorUrl	N/A - will be supported in the CAS plugin
cas.filterProcessesUrl	N/A - will be supported in the CAS plugin
useNtlm	N/A - will be supported in a secondary plugin
ntlm.stripDomain	N/A - will be supported in a secondary plugin
ntlm.retryOnAuthFailure	N/A - will be supported in a secondary plugin
ntlm.forceIdentification	N/A - will be supported in a secondary plugin
ntlm.defaultDomain	N/A - will be supported in a secondary plugin

ntlm.netbiosWINS	N/A - will be supported in a secondary plugin
httpPort	portMapper.httpPort
httpsPort	portMapper.httpsPort
secureChannelDefinitionSource	N/A, use secureChannel.definition
channelConfig	secureChannel.definition
ipRestrictions	ipRestrictions
useFacebook	N/A - will be supported in the Facebook plugin
facebook.filterProcessesUrl	N/A - will be supported in the Facebook plugin
facebook.authenticationUrlRoot	N/A - will be supported in the Facebook plugin
facebook.apiKey	N/A - will be supported in the Facebook plugin
facebook.secretKey	N/A - will be supported in the Facebook plugin

Script differences

In the Acegi plugin you run the `create-auth-domains` script to initialize the plugin. This creates `grails-app/conf/SecurityConfig.groovy` to allow configuration customization, and creates the `User`, `Role`, and `Requestmap` domain classes, along with the `Login` and `Logout` controllers and views. In addition there's the `generate-manager` script which creates CRUD pages for the domain classes (earlier version of Grails didn't scaffold many-to-many relationships well, so these GSPs were necessary), and a `generate-registration` script which installs a basic user registration controller.

In the Spring Security plugin, there's just one script, [s2-quickstart](#). It's most similar to `create-auth-domains` since it creates domain classes and login/logout controllers, but it appends to `grails-app/conf/Config.groovy` instead of creating a standalone configuration file. There's no equivalent to `generate-manager` or `generate-registration` since there will be an optional UI plugin that will generate domain class management screens, an admin console, and forgot password and registration workflows. If you want to create your own CRUD pages you can use the standard Grails `generate-all` script. Various sections of this documentation discusses the changes you'll need to make in the generated source files, e.g. encrypting passwords before saving or updating a user.

domainClass

The Acegi plugin extended the `UserDetails` instance and added an accessor for the person domain class instance that was used to populate the `UserDetails`. This is convenient because since the `Authentication` is kept in the HTTP session and the `UserDetails` is attached to that, so it was easy to access non-security data such as full name, email, etc. without hitting the database.

This caused problems however. One is that if the domain class has a lot of data, you increase the size of the session payload and this is even worse if you have clustered sessions. Further, any lazy-loaded collections would fail to load after retrieving the person from the session since it would have become a detached Hibernate object. This caused confusion but is easily fixed, either by calling `person.attach()` or by reloading by id, i.e.

```
def userDetails = authenticateService.principal()
def person = userDetails.domainClass
person = Person.get(person.id)
```

but then the person has essentially become a very large wrapper around its primary key since that's the real data you're storing.

So the approach that this plugin takes is to not store the domain class but instead store the id so you can retrieve the person easily:

```
def userDetails = springSecurityService.principal
person = Person.get(userDetails.id)
```

This works because the `UserDetails` implementation is an instance of `org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser` which extends the standard Spring Security [User](#) and adds a `getId()` method.

You're encouraged to further extend this class if you want to store more data along with the authentication to avoid database access - see [Chapter 7](#) for details on this.

2. Domain Classes

The plugin uses regular Grails domain classes to access its required data. At a minimum you'll need a 'person' and an 'authority' domain class. In addition, if you want to store URL<->Role mappings in the database (this is one of multiple approaches for defining the mappings) then you'll need a 'requestmap' domain class, and if you use the recommended approach for mapping the many-to-many relationship between 'person' and 'authority' then you'll also need a domain class to map the join table.

The [s2-quickstart](#) script creates initial domain classes for you. You specify the package and class names, and it creates the corresponding domain classes. After that you can customize them as you like. You can add as many other fields, methods, etc. as you like, as long as the core security-related functionality remains.

2.1 Person

Spring Security uses an [Authentication](#) object to determine whether the current user has the right to perform a secured action, i.e. accessing a URL, manipulate a secured domain object, access a secured method, etc. This is created during login, and typically there's overlap between the need for the data that's required to populate the authentication (username, password, granted authorities, etc.) and the need to represent a user in the application in ways that aren't related to security. The mechanism for populating the authentication is completely pluggable in Spring Security - you only need to provide an implementation of [UserDetailsService](#) and implement its one method, `loadUserByUsername()`.

You can easily [plug in your own implementation](#) but by default the plugin uses a Grails 'person' domain class to manage this data. The class name and package can be named whatever you want, and so can the fields. By default the class name is `Person`, and `username`, `enabled`, `password` are the default names of the required properties. In addition it's expected that there is an `authorities` property to retrieve roles; this can either be a public field or a `getAuthorities()` method, and it can either be defined via a traditional GORM many-to-many or via a custom mapping (more [HERE](#)).

Assuming you choose `com.mycompany.myapp` as your package, and `User` as your class name, you'll generate this class:

```
package com.mycompany.myapp
class User {
    String username
    String password
    boolean enabled
    boolean accountExpired
    boolean accountLocked
    boolean passwordExpired
    static constraints = {
        username blank: false, unique: true
        password blank: false
    }
    static mapping = {
        password column: '`password`'
    }
    Set<Role> getAuthorities() {
        UserRole.findAllByUser(this).collect { it.role } as Set
    }
}
```

and of course if you like you can add other properties, e.g. `email`, `firstName`, `lastName`, and convenience methods, etc.:

```

package com.mycompany.myapp
class User {
    String username
    String password
    boolean enabled
    String email
    String firstName
    String lastName
    static constraints = {
        username blank: false, unique: true
        password blank: false
    }
    Set<Role> getAuthorities() {
        UserRole.findAllByUser(this).collect { it.role } as Set
    }
    def someMethod {
        ...
    }
}

```

The `getAuthorities()` method is analagous to defining `hasMany = [authorities: Authority]` in a traditional many-to-many mapping. This way `GormUserDetailsService` can call `user.authorities` during login to retrieve the roles without the overhead of a bidirectional many-to-many mapping.

The class and property names are configurable using these configuration attributes:

Property	Default Value	Meaning
<code>userLookup.userDomainClassName</code>	'Person'	User class name
<code>userLookup.usernamePropertyName</code>	'username'	User class username field
<code>userLookup.passwordPropertyName</code>	'password'	User class password field
<code>userLookup.authoritiesPropertyName</code>	'authorities'	User class role collection field
<code>userLookup.enabledPropertyName</code>	'enabled'	User class enabled field
<code>userLookup.accountExpiredPropertyName</code>	'accountExpired'	User class account expired field
<code>userLookup.accountLockedPropertyName</code>	'accountLocked'	User class account locked field
<code>userLookup.passwordExpiredPropertyName</code>	'passwordExpired'	User class password expired field
<code>userLookup.authorityJoinClassName</code>	'PersonAuthority'	User/Role many-many join class name

2.2 Authority

The plugin also requires an 'authority' class to represent a user's role(s) in the application, used in general to restrict URLs to users who have been assigned the required access rights. A user can have multiple roles to indicate various access rights in the application, and should have at least one. A basic user who can only access non-restricted resources but can still authenticate is a bit unusual. Spring Security will for the most part function fine if a user has no granted authorities, but will fail in a few places that assume one or more. So if a user authenticates successfully but has no granted roles, the plugin will grant the user a 'virtual' role, `ROLE_NO_ROLES` to work around this limitation. This way the user will satisfy all of Spring Security's requirements but not be able to do anything since you wouldn't associate any secure resources with this role.

Like the 'person' class, the 'authority' class has a default name, `Authority`, and a default name for its one required property, `authority`. If you want to use an existing domain class, it just has to have a property for name. As with the name of the class, the names of the properties can be whatever you want - they're specified in `grails-app/conf/Config.groovy`.

Assuming you choose `com.mycompany.myapp` as your package, and `Role` as your class name, you'll generate this class:

```

package com.mycompany.myapp
class Role {
    String authority
    static mapping = {
        cache true
    }
    static constraints = {
        authority blank: false, unique: true
    }
}

```

The class and property names are configurable using these configuration attributes:

Property	Default Value	Meaning
authority.className	'Authority'	Role class name
authority.nameField	'authority'	Role class role name field

2.3 PersonAuthority

The typical approach for mapping the relationship between 'person' and 'authority' is a many-to-many; users have multiple roles, and roles are shared by multiple users. This can be problematic in Grails however since a popular role, e.g. `ROLE_USER`, will be granted to many users in your application. Since GORM uses collections to manage adding and removing related instances and maps many-to-many relationships bidirectionally, granting a role to a user requires loading all of the existing users who have that role because the collection is a `Set`. So even though there may be no uniqueness concerns, Hibernate will still load them all to enforce uniqueness. The recommended approach in the plugin is to map a domain class to the join table that manages the many-to-many, and using that to grant and revoke roles to users.

Like the other domain classes, this will be generated for you, so you don't need to deal with the details of mapping it. Assuming you choose `com.mycompany.myapp` as your package, and `User` and `Role` as your class names, you'll generate this class:

```

package com.testapp
import org.apache.commons.lang.builder.HashCodeBuilder
class UserRole implements Serializable {
    User user
    Role role
    boolean equals(other) {
        if (!(other instanceof UserRole)) {
            return false
        }
        other.user?.id == user?.id &&
        other.role?.id == role?.id
    }
    int hashCode() {
        def builder = new HashCodeBuilder()
        if (user) builder.append(user.id)
        if (role) builder.append(role.id)
        builder.toHashCode()
    }
    static UserRole get(long userId, long roleId) {
        find 'from UserRole where user.id=:userId and role.id=:roleId',
            [userId: userId, roleId: roleId]
    }
    static UserRole create(User user, Role role, boolean flush = false) {
        new UserRole(user: user, role: role).save(flush: flush, insert: true)
    }
    static boolean remove(User user, Role role, boolean flush = false) {
        UserRole instance = UserRole.findByUserAndRole(user, role)
        instance ? instance.delete(flush: flush) : false
    }
    static void removeAll(User user) {
        executeUpdate 'DELETE FROM UserRole WHERE user=:user', [user: user]
    }
    static mapping = {
        id composite: ['role', 'user']
        version false
    }
}

```

The helper methods make it easy to grant or revoke roles. Assuming you've already loaded a user and a role, you grant the role to the user with

```
User user = ...
Role role = ...
UserRole.create user, role
```

or using the 3-paramter version to trigger a flush:

```
User user = ...
Role role = ...
UserRole.create user, role, true
```

Revoking a role is similar:

```
User user = ...
Role role = ...
UserRole.revoke user, role
```

or

```
User user = ...
Role role = ...
UserRole.revoke user, role, true
```

The class name is the only configurable attribute:

Property	Default Value	Meaning
userLookup.authorityJoinClassName	'PersonAuthority'	User/Role many-many join class name

2.4 Requestmap

This class is optionally used to store request mapping entries in the database instead of defining them with annotations or in `Config.groovy`. This has the advantage of being configurable at runtime; you can add, remove and edit rules without restarting your application.

Assuming you choose `com.mycompany.myapp` as your package, and `Requestmap` as your class name, you'll generate this class:

```
package com.testapp
class Requestmap {
    String url
    String configAttribute
    static mapping = {
        cache true
    }
    static constraints = {
        url blank: false, unique: true
        configAttribute blank: false
    }
}
```

The class and property names are configurable using these configuration attributes:

Property	Default Value	Meaning
requestMap.className	'Requestmap'	requestmap class name
requestMap.urlField	'url'	URL pattern field name
requestMap.configAttributeField	'configAttribute'	authority pattern field name

See [this section](#) on URL mapping for details on using Requestmap entries to guard URLs.

3. Securing URLs

There are three ways to configure request mappings to secure application URLs. The goal is to create a mapping of URL patterns to the roles required to access those URLs. Use whichever approach makes the most sense for you. The three approaches are:

- @Secured annotations
- a simple Map in Config.groovy
- Requestmap domain class instances stored in the database

and the default approach is to use annotations. You can only use one method at a time, and it's configured with the securityConfigType attribute; the value has to be an SecurityConfigType enum value.

To use annotations, specify SecurityConfigType.Annotation (or leave it unspecified since it's the default):

```
import grails.plugins.springsecurity.SecurityConfigType
...
grails.plugins.springsecurity.securityConfigType = SecurityConfigType.Annotation
```

To use the Config.groovy Map, specify SecurityConfigType.InterceptUrlMap:

```
import grails.plugins.springsecurity.SecurityConfigType
...
grails.plugins.springsecurity.securityConfigType = SecurityConfigType.InterceptUrlMap
```

To use Requestmap entries, specify SecurityConfigType.Requestmap:

```
import grails.plugins.springsecurity.SecurityConfigType
...
grails.plugins.springsecurity.securityConfigType = SecurityConfigType.Requestmap
```

In addition, you can use a pessimistic 'lockdown' approach if you like. Most applications are mostly public, with some pages only accessible to authenticated users with various roles. Here it makes more sense to leave URLs open by default and restrict access one a case-by-case basis. But if your app is primarily secure, you can deny access to all URLs that don't have an applicable URL-Role configuration.

To use the pessimistic approach, add this to grails-app/conf/Config.groovy:

```
grails.plugins.springsecurity.rejectIfNoRule = true
```

and any requested URL that doesn't have a corresponding rule will be denied to all users.

URLs and authorities

In each approach you configure a mapping for a URL pattern to the role(s) that are required to access those URLs, e.g. /admin/user/** requires ROLE_ADMIN. In addition, you can combine the role(s) with tokens such as IS_AUTHENTICATED_ANONYMOUSLY, IS_AUTHENTICATED_REMEMBERED, and IS_AUTHENTICATED_FULLY. One or more Voters will process any tokens and enforce a rule based on them:

- IS_AUTHENTICATED_ANONYMOUSLY
 - signifies that anyone can access this URL; by default the AnonymousAuthenticationFilter ensures that there's an 'anonymous' Authentication with no roles so every user has an authentication, so this token just requires any authentication, even anonymous
- IS_AUTHENTICATED_REMEMBERED
 - signifies that the user is authenticated via a remember-me cookie or an explicit login
- IS_AUTHENTICATED_FULLY
 - requires that the user be fully authenticated via an explicit login

IS_AUTHENTICATED_FULLY is useful to implement a security scheme where you allow users to check a

remember-me checkbox during login, and auto-authenticate them each time they come back to your site, but require them to login with a password for some parts of the site. For example regular browsing might be allowed and even adding items to a shopping cart with only a cookie, but checking out or viewing purchase history would require an explicit login.

Advantages and disadvantages

Each approach has its advantages and disadvantages. Annotations and the `Config.groovy` Map are less flexible since they're configured once in the code and can only be updated by restarting the application (in prod mode anyway). In practice this isn't that serious a concern since for most applications security mappings are unlikely to change at runtime.

If you want runtime-configurability then storing `Requestmap` entries enables this. This allows you to have a core set of rules populated at application startup and to edit, add, and delete them whenever you like. But it separates the security rules from the application code, which is less convenient than having the rules defined in `grails-app/conf/Config.groovy` or in the applicable controllers using annotations.

Some notes

- to understand the meaning of `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY`, see the Javadoc for [AuthenticatedVoter](#)
- URLs must be mapped in lowercase if using the `Requestmap` or `grails-app/conf/Config.groovy` map approaches, so for example if you have a `FooBarController`, its urls will be of the form `/fooBar/list`, `/fooBar/create`, etc. but these must be mapped as `/foobar/`, `/foobar/list`, `/foobar/create`. This is handled automatically for you if you use annotations.

3.1 Annotations

The plugin supplies an `@Secured` annotation that you can use in your controllers to configure which roles are required for which actions.

You can define the annotation at the class level, meaning that the specified roles are required for all actions, or at the action level, or both. If the class and an action are annotated then the action annotation values will be used since they're more specific.

For example, given this controller:

```
package com.mycompany.myapp
import grails.plugins.springsecurity.Secured
class SecureAnnotatedController {
    @Secured(['ROLE_ADMIN'])
    def index = {
        render 'you have ROLE_ADMIN'
    }
    @Secured(['ROLE_ADMIN', 'ROLE_SUPERUSER'])
    def adminEither = {
        render 'you have ROLE_ADMIN or SUPERUSER'
    }
    def anybody = {
        render 'anyone can see this'
    }
}
```

you'd need to be authenticated and have `ROLE_ADMIN` to see `/myapp/secureAnnotated` (or `/myapp/secureAnnotated/index`) and be authenticated and have `ROLE_ADMIN` or `ROLE_SUPERUSER` to see `/myapp/secureAnnotated/adminEither`. Any user can access `/myapp/secureAnnotated/anybody`.

Quite often most actions in a controller require similar access rules, so you can also define annotations at the class level:

```

package com.mycompany.myapp
import grails.plugins.springsecurity.Secured
@Secured(['ROLE_ADMIN'])
class SecureClassAnnotatedController {
    def index = {
        render 'index: you have ROLE_ADMIN'
    }
    def otherAction = {
        render 'otherAction: you have ROLE_ADMIN'
    }
}
@Secured(['ROLE_SUPERUSER'])
def super = {
    render 'super: you have ROLE_SUPERUSER'
}
}

```

Here you'd need to be authenticated and have `ROLE_ADMIN` to see `/myapp/secureClassAnnotated` (or `/myapp/secureClassAnnotated/index`) or `/myapp/secureClassAnnotated/otherAction`. However you must have `ROLE_SUPERUSER` to access `/myapp/secureClassAnnotated/super` - the action-scope annotation overrides the class-scope annotation.

controllerAnnotations.staticRules

You can also define 'static' mappings that cannot be expressed in the controllers, such as '/' or for JavaScript, CSS, or image URLs. Use the `controllerAnnotations.staticRules` property, e.g.

```

grails.plugins.springsecurity.controllerAnnotations.staticRules = [
    '/js/admin/**': ['ROLE_ADMIN'],
    '/somePlugin/**': ['ROLE_ADMIN']
]

```

In this example we've mapped all URLs associated with 'somePlugin' to `ROLE_ADMIN`; annotations aren't an option here since you wouldn't want to edit plugin code for a change like this.

3.2 Config.groovy

To use this approach, just define a Map in `Config.groovy`:

```

grails.plugins.springsecurity.interceptUrlMap = [
    '/secure/**': ['ROLE_ADMIN'],
    '/finance/**': ['ROLE_FINANCE', 'IS_AUTHENTICATED_FULLY'],
    '/js/**': ['IS_AUTHENTICATED_ANONYMOUSLY'],
    '/css/**': ['IS_AUTHENTICATED_ANONYMOUSLY'],
    '/images/**': ['IS_AUTHENTICATED_ANONYMOUSLY'],
    '/**': ['IS_AUTHENTICATED_ANONYMOUSLY'],
    '/login/**': ['IS_AUTHENTICATED_ANONYMOUSLY'],
    '/logout/**': ['IS_AUTHENTICATED_ANONYMOUSLY']
]

```

When using this approach, make sure that you order the rules correctly. The first applicable rule is used, so for example if you have a controller that has one set of rules but an action that has stricter access rules, e.g.

```

'/secure/**': ['ROLE_ADMIN', 'ROLE_SUPERUSER'],
'/secure/reallysecure/**': ['ROLE_SUPERUSER']

```

then this would fail - it wouldn't restrict access to `/secure/reallysecure/list` to a user with `ROLE_SUPERUSER` since the first URL pattern matches, so the second would be ignored. The correct mapping would be


```
'/secure/reallysecure/**': ['ROLE_SUPERUSER']
'/secure/**':               ['ROLE_ADMIN', 'ROLE_SUPERUSER'],
```

3.3 Requestmap

With this approach you store mapping entries in the database, using the Requestmap domain class. Requestmap has a url property which contains the secured URL pattern and a configAttribute property containing a comma-delimited list of required roles and/or tokens such as IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_REMEMBERED, and IS_AUTHENTICATED_ANONYMOUSLY. Creation of Requestmap entries is the same as for any Grails domain class:

```
new Requestmap(url: '/js/**', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/css/**', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/images/**', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/login/**', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/logout/**', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/*', configAttribute: 'IS_AUTHENTICATED_ANONYMOUSLY').save()
new Requestmap(url: '/profile/**', configAttribute: 'ROLE_USER').save()
new Requestmap(url: '/admin/**', configAttribute: 'ROLE_ADMIN').save()
new Requestmap(url: '/admin/user/**', configAttribute: 'ROLE_SUPERVISOR').save()
```

Unlike the Config.groovy Map approach above, you don't need to worry about Requestmap entry order since the plugin calculates the most specific rule that applies to the current request.

Requestmap cache

Requestmap entries are cached for performance, but this has an impact on runtime configurability. If you create, edit, or delete an instance, the cache must be flushed and repopulated to be consistent with the database. You can call `springSecurityService.clearCachedRequestmaps()` to do this. For example, if you create a RequestmapController the save action should look like this (and the update and delete actions should similarly call `clearCachedRequestmaps()`):

```
class RequestmapController {
    def springSecurityService
    ...
    def save = {
        def requestmapInstance = new Requestmap(params)
        if (!requestmapInstance.save(flush: true)) {
            render view: 'create', model: [requestmapInstance: requestmapInstance]
            return
        }
        springSecurityService.clearCachedRequestmaps()
        flash.message = "${message(code: 'default.created.message', args: [message(code: 'requ
        redirect action: show, id: requestmapInstance.id
    }
}
```

4. Helper Classes

The plugin has a few helper classes that you can use in your application to avoid having to deal with the lower-level details of Spring Security.

4.1. Security Tags

The plugin comes with a few GSP tags to support conditional display based on whether the user is authenticated, and/or has the required role to perform some action. All of the tags are in the `sec` namespace and are implemented in `grails.plugins.springsecurity.SecurityTagLib`.

ifLoggedIn

Displays the inner body content if the user is authenticated.

Example:

```
<sec:ifLoggedIn>
Welcome Back!
</sec:ifLoggedIn>
```

ifNotLoggedIn

Displays the inner body content if the user is not authenticated.

Example:

```
<sec:ifNotLoggedIn>
<g:link controller='login' action='auth'>Login</g:link>
</sec:ifNotLoggedIn>
```

ifAllGranted

Displays the inner body content only if all of the listed roles are granted.

Example:

```
<sec:ifAllGranted roles="ROLE_ADMIN,ROLE_SUPERVISOR">secure stuff here</sec:ifAllGranted>
```

ifAnyGranted

Displays the inner body content if at least one of the listed roles are granted.

Example:

```
<sec:ifAnyGranted roles="ROLE_ADMIN,ROLE_SUPERVISOR">secure stuff here</sec:ifAnyGranted>
```

ifNotGranted

Displays the inner body content if none of the listed roles are granted.

Example:

```
<sec:ifNotGranted roles="ROLE_USER">non-user stuff here</sec:ifNotGranted>
```

loggedInUserInfo

Displays the value of the specified authentication field if logged in. For example this will show the username property:

```
<sec:loggedInUserInfo field="username"/>
```

and if you have customized the authentication to add a `fullName` property, you would access it using

```
Welcome Back <sec:loggedInUserInfo field="fullName"/>
```

username

Displays the value of the authentication username field if logged in.

```
<sec:ifLoggedIn>
Welcome Back <sec:username/>
</sec:ifLoggedIn>
<sec:ifNotLoggedIn>
<g:link controller='login' action='auth'>Login</g:link>
</sec:ifNotLoggedIn>
```

ifSwitched

Displays the inner body content only if the current user switched from another user.

```
<sec:ifLoggedIn>
Logged in as <sec:username/>
</sec:ifLoggedIn>
<sec:ifSwitched>
<a href='${request.contextPath}/j_spring_security_exit_user'>
  Resume as <sec:switchedUserOriginalUsername/>
</a>
</sec:ifSwitched>
<sec:ifNotSwitched>
  <sec:ifAllGranted roles='ROLE_SWITCH_USER'>
    <form action='${request.contextPath}/j_spring_security_switch_user' method='POST'>
      Switch to user: <input type='text' name='j_username'/><br/>
      <input type='submit' value='Switch'/> </form>
    </sec:ifAllGranted>
  </sec:ifNotSwitched>
```

ifNotSwitched

Displays the inner body content only if the current user has not switched from another user.

switchedUserOriginalUsername

Renders the original user's username if the current user switched from another user.

4.2. SpringSecurityService

`grails.plugins.springsecurity.SpringSecurityService` provides security utility functions. It's a regular Grails service, so you can use dependency injection to inject it into a controller, service, taglib, etc.:

```
def springSecurityService
```

isLoggedIn()

Checks to see if there's a currently logged-in user.

Example:

```
class SomeController {
    def springSecurityService
    def someAction = {
        if (springSecurityService.isLoggedIn()) {
            ...
        }
        else {
            ...
        }
    }
}
```

getAuthentication()

Retrieves the current user's [Authentication](#) if logged in, or null otherwise.

Example:

```
class SomeController {
    def springSecurityService
    def someAction = {
        def auth = springSecurityService.authentication
        String username = auth.username
        def authorities = auth.authorities // a Collection of GrantedAuthority
        boolean authenticated = auth.authenticated
        ...
    }
}
```

getPrincipal()

Retrieves the currently logged in user's Principal, or null if not logged in. This will be a `org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser` unless you've created a custom `UserDetailsService`, in which case it'll be whatever implementation of [UserDetails](#) you use there.

Example:

```
class SomeController {
    def springSecurityService
    def someAction = {
        def principal = springSecurityService.principal
        String username = principal.username
        def authorities = principal.authorities // a Collection of GrantedAuthority
        boolean enabled = principal.enabled
        ...
    }
}
```

encodePassword()

Encrypts a password using the configured encryption scheme. By default the plugin uses SHA-256, but this is configurable using the `grails.plugins.springsecurity.password.algorithm` attribute in `Config.groovy`. You can use any message digest algorithm that's supported in your JDK; see [this page](#) for information on what's available. In particular you are **strongly** discouraged from using MD5 or SHA-1 algorithms since they are rather weak and have well-known vulnerabilities. You should also use a salt for your passwords, which greatly increases the computational complexity of decrypting passwords if your database gets compromised. See [here](#) for details on using salted passwords.

Example:

```

class PersonController {
  def springSecurityService
  def updateAction = {
    def person = Person.get(params.id)
    params.salt = person.salt
    if (person.password != params.password) {
      params.password = springSecurityService.encodePassword(password, salt)
      def salt = ... // e.g. randomly generated using some utility method
      params.salt = salt
    }
    person.properties = params
    if (!person.save(flush: true)) {
      render view: 'edit', model: [person: person]
      return
    }
    redirect action: show, id: person.id
  }
}

```

updateRole()

Updates a role and if you're using Requestmap instances to manage securing URLs, will replace the new role name in all Requestmap definitions that use it if the name was changed.

Example:

```

class RoleController {
  def springSecurityService
  def update = {
    def roleInstance = Role.get(params.id)
    if (!springSecurityService.updateRole(roleInstance, params)) {
      render view: 'edit', model: [roleInstance: roleInstance]
      return
    }
    flash.message = "The role was updated"
    redirect action: show, id: roleInstance.id
  }
}

```

deleteRole()

Deletes a role and if you're using Requestmap instances to manage securing URLs, will remove the role from all Requestmap definitions. If a Requestmap's config attribute is just this role's name (e.g. "/foo/bar/=ROLE_FOO") it will be deleted.

Example:

```

class RoleController {
  def springSecurityService
  def delete = {
    def roleInstance = Role.get(params.id)
    try {
      springSecurityService.deleteRole (roleInstance)
      flash.message = "The role was deleted"
      redirect action: list
    }
    catch (DataIntegrityViolationException e) {
      flash.message = "Unable to delete the role"
      redirect action: show, id: params.id
    }
  }
}

```

clearCachedRequestmaps()

If you're using Requestmap instances to manage securing URLs, the plugin will load and cache all Requestmap instances as a performance optimization. This saves a lot of database activity since the requestmaps are checked for each request. But you can't allow the cache to become stale, so when you create, edit or delete a Requestmap you should flush the cache to trigger a complete reload. Both `updateRole()` and `deleteRole()` call this method for you, so you should call this when you create a new Requestmap or if you do some other Requestmap work that would affect the cache.

Example:

```
class RequestmapController {
    def springSecurityService
    def save = {
        def requestmapInstance = new Requestmap(params)
        if (!requestmapInstance.save(flush: true)) {
            render view: 'create', model: [requestmapInstance: requestmapInstance]
            return
        }
        springSecurityService.clearCachedRequestmaps()
        flash.message = "Requestmap created"
        redirect action: show, id: requestmapInstance.id
    }
}
```

reauthenticate()

Rebuilds an [Authentication](#) for the given username and registers it in the security context. This is typically used after updating a user's authorities or other data that is cached in the Authentication or Principal. It also removes the user from the user cache to force a refresh at next login.

Example:

```
class UserController {
    def springSecurityService
    def update = {
        def userInstance = User.get(params.id)
        params.salt = person.salt
        if (userInstance.password != params.password) {
            params.password = springSecurityService.encodePassword(params.password, salt)
            def salt = ... // e.g. randomly generated using some utility method
            params.salt = salt
        }
        userInstance.properties = params
        if (!userInstance.save(flush: true)) {
            render view: 'edit', model: [userInstance: userInstance]
            return
        }
        if (springSecurityService.loggedIn &&
            springSecurityService.principal.username == userInstance.username) {
            springSecurityService.reauthenticate userInstance.username
        }
        flash.message = "The user was updated"
        redirect action: show, id: userInstance.id
    }
}
```

4.3. SpringSecurityUtils

`org.codehaus.groovy.grails.plugins.springsecurity.SpringSecurityUtils` is a utility class with static methods that can be called directly without using dependency injection. It's primarily an internal class but can be called from application code.

authoritiesToRoles()

Extracts role names from an array or Collection of [GrantedAuthority](#).

getPrincipalAuthorities()

Retrieves the currently logged-in user's authorities. Will be empty (but never null) if not logged in.

parseAuthoritiesString()

Splits a comma-delimited String containing role names into a List of [GrantedAuthority](#)

ifAllGranted()

Checks if the current user has all of the specified roles (a comma-delimited String of role names). This is primarily used by `SecurityTagLib.ifAllGranted`

ifNotGranted()

Checks if the current user has none of the specified roles (a comma-delimited String of role names). This is primarily

used by `SecurityTagLib.ifNotGranted`

ifAnyGranted()

Checks if the current user has any of the specified roles (a comma-delimited String of role names). This is primarily used by `SecurityTagLib.ifAnyGranted`

getSecurityConfig()

Retrieves the security part of the Configuration (from `grails-app/conf/Config.groovy`).

loadSecondaryConfig()

Used by dependent plugins to add configuration attributes.

reloadSecurityConfig()

Forces a reload of the security configuration.

isAjax()

Checks if the request was triggered by an Ajax call. The standard way to determine this is to see if `X-Requested-With` request header is set and has the value `XMLHttpRequest`. The plugin relaxes this a bit and only checks if the header is set to any value. In addition, you can configure the name of the header using the `grails.plugins.springsecurity.ajaxHeader` configuration attribute, but this shouldn't be done in general since all of the major JavaScript toolkits use the standard name.

In addition, you can force the request to be treated as Ajax by appending `&ajax=true` to your request query string.

registerProvider()

Used by dependent plugins to register an [AuthenticationProvider](#) bean name.

registerFilter()

Used by dependent plugins to register a filter bean name in a specified position in the filter chain.

isSwitched()

Checks if the current user switched from another user.

getSwitchedUserOriginalUsername()

Gets the original user's username if the current user switched from another user.

5. Events

Spring Security fires application events after various security-related actions such as successful login, unsuccessful login, etc. There are two hierarchies of events, [AbstractAuthenticationEvent](#) and [AbstractAuthorizationEvent](#). There are two main ways of being notified of these events when using the plugin:

- register an event listener, ignoring events that you're not interested in (Spring only allows partial event subscription; you use generics to register the class of events you want to be notified of and it notifies you of those and all subclasses)
- register one or more callback closures in `grails-app/conf/Config.groovy` that take advantage of the plugin's `org.codehaus.groovy.grails.plugins.springsecurity.SecurityEventListener` which does the filtering for you

The first approach involves creating one or more Groovy or Java classes, e.g.

```
package com.foo.bar
import org.springframework.context.ApplicationListener
import org.springframework.security.authentication.event.AuthenticationSuccessEvent
class MySecurityEventListener implements ApplicationListener<AuthenticationSuccessEvent> {
    void onApplicationEvent(AuthenticationSuccessEvent event) {
        // handle the event
    }
}
```

registering them in `grails-app/conf/spring/resources.groovy`:

```
beans = {
    mySecurityEventListener(com.foo.bar.MySecurityEventListener)
}
```

Alternatively, you can just register one or more callback closure(s) in `grails-app/conf/Config.groovy` and let `SecurityEventListener` do all of the work for you, and you can just handle any event you like, e.g.:

```
grails.plugins.springsecurity.useSecurityEventListener = true
grails.plugins.springsecurity.onInteractiveAuthenticationSuccessEvent = { e, appCtx ->
    // handle InteractiveAuthenticationSuccessEvent
}
grails.plugins.springsecurity.onAbstractAuthenticationFailureEvent = { e, appCtx ->
    // handle AbstractAuthenticationFailureEvent
}
grails.plugins.springsecurity.onAuthenticationSuccessEvent = { e, appCtx ->
    // handle AuthenticationSuccessEvent
}
grails.plugins.springsecurity.onAuthenticationSwitchUserEvent = { e, appCtx ->
    // handle AuthenticationSwitchUserEvent
}
grails.plugins.springsecurity.onAuthorizationEvent = { e, appCtx ->
    // handle AuthorizationEvent
}
```


None of these closures are required; if none are configured, nothing will be called. Just implement the event handlers that you need.

Note that when a user authenticates, Spring Security initially fires an `AuthenticationSuccessEvent` but this happens before the `Authentication` is registered in the `SecurityContextHolder`. This means that the `springSecurityService` methods that access the logged-in user won't work. Later in the processing a second event is fired, an `InteractiveAuthenticationSuccessEvent`, and when this happens the `SecurityContextHolder` will have the `Authentication`. So depending on your needs you can choose to implement a callback for either or both events.

Also note that your event callback(s) will be ignored unless you set the `useSecurityEventListener` property to true. This allows you to temporarily disable/enable them or enable them per-environment.

6. Configuration

Much of the Spring Security configuration is user-configurable. The configuration has sensible default values, but each application has special needs. Default values are in the plugin's `grails-app/conf/DefaultSecurityConfig.groovy` file and you can put application-specific values in `grails-app/conf/Config.groovy`. The two configurations will be merged with application values overriding the defaults.

 All of these property overrides must be specified in `Config.groovy` using the `grails.plugins.springsecurity` suffix, for example

```
grails.plugins.springsecurity.userLookup.userDomainClassName =  
    'com.mycompany.myapp.User'
```

Properties that are most likely to be overridden are the `User` and `Role` (and `Requestmap` if using the database to store mappings) class and field names:

Property	Default Value	Meaning
<code>userLookup.userDomainClassName</code>	<code>'Person'</code>	User class name
<code>userLookup.usernamePropertyName</code>	<code>'username'</code>	User class username field
<code>userLookup.passwordPropertyName</code>	<code>'password'</code>	User class password field
<code>userLookup.authoritiesPropertyName</code>	<code>'authorities'</code>	User class role collection field
<code>userLookup.enabledPropertyName</code>	<code>'enabled'</code>	User class enabled field
<code>userLookup.accountExpiredPropertyName</code>	<code>'accountExpired'</code>	User class account expired field
<code>userLookup.accountLockedPropertyName</code>	<code>'accountLocked'</code>	User class account locked field
<code>userLookup.passwordExpiredPropertyName</code>	<code>'passwordExpired'</code>	User class password expired field
<code>userLookup.authorityJoinClassName</code>	<code>'PersonAuthority'</code>	User/Role many-many join class name
<code>authority.className</code>	<code>'Authority'</code>	Role class name
<code>authority.nameField</code>	<code>'authority'</code>	Role class role name field
<code>requestMap.className</code>	<code>'Requestmap'</code>	Requestmap class name
<code>requestMap.urlField</code>	<code>'url'</code>	Requestmap class URL pattern field
<code>requestMap.configAttributeField</code>	<code>'configAttribute'</code>	Requestmap class role/token field

To customize the login error messages that are displayed for the various error conditions:

Property	Default Value	Meaning
<code>errors.login.fail</code>	"Sorry, we were not able to find a user with that username and password."	message displayed when authentication is successful because of missing user or bad password
<code>errors.login.disabled</code>	"Sorry, your account is disabled."	message displayed when authentication is successful but user is not enabled

rememberMeServices bean (cookie management)

Property	Default Value	Meaning
rememberMe.cookieName	'grails_remember_me'	remember-me cookie name - should be unique per application
rememberMe.alwaysRemember	false	create a remember-me cookie even if there's no checkbox on the form if true
rememberMe.tokenValiditySeconds	1209600 (14 days)	max age of the cookie in seconds
rememberMe.parameter	'_spring_security_remember_me'	Login form remember-me checkbox name
rememberMe.key	'grailsRocks'	a value used to encode cookies - should be unique per application
atr.rememberMeClass	RememberMeAuthenticationToken	remember-me authentication class

URL attributes

Property	Default Value	Meaning
apf.filterProcessesUrl	'/j_spring_security_check'	login form post URL, intercepted by Spring Security filter
apf.usernameParameter	'j_username'	login form username parameter
apf.passwordParameter	'j_password'	login form password parameter
apf.allowSessionCreation	true	whether or not to allow authentication to create an HTTP session
apf.postOnly	true	whether to only allow POST login requests
failureHandler.defaultFailureUrl	'/login/authfail?login_error=1'	redirect URL for failed logins
failureHandler.ajaxAuthFailUrl	'/login/authfail?ajax=true'	redirect URL for failed Ajax logins
failureHandler.exceptionMappings	none	a map of exception class name (subclass of AuthenticationException) to URL to redirect to for that exception type after authentication failure
failureHandler.useForward	false	whether to render the error page (true) or redirect (false)
successHandler.defaultTargetUrl	'/'	default post-login URL if there's no saved request that triggered the login
successHandler.alwaysUseDefault	false	if true, always redirect to the value of successHandler.defaultTargetUrl after successful authentication, otherwise redirects to to originally-requested page
successHandler.targetUrlParameter	'spring-security-redirect'	name of optional login form parameter that specifies destination after successful login
successHandler.useReferer	false	whether to use the HTTP Referer header to determine post-login destination
successHandler.ajaxSuccessUrl	'/login/ajaxSuccess'	URL to redirect to after successful Ajax login
auth.loginFormUrl	'/login/auth'	URL of login page
auth.forceHttps	false	if true, will redirect login page requests to HTTPS
auth.ajaxLoginFormUrl	'/login/authAjax'	URL of Ajax login page
auth.useForward	false	whether to render the login page (true) or redirect (false)
logout.afterLogoutUrl	'/'	URL to redirect to after logout
logout.filterProcessesUrl	'/j_spring_security_logout'	logout URL, intercepted by Spring Security filter
logout.handlerNames	'rememberMeServices', 'securityContextLogoutHandler'	logout handler bean names; more details are here
adh.errorPage	'/login/denied'	location of the 403 error page
adh.ajaxErrorPage	'/login/ajaxDenied'	location of the 403 error page for Ajax requests
ajaxHeader	'X-Requested-With'	header name sent by Ajax library, used to detect Ajax
redirectStrategy.contextRelative	false	if true, the redirect URL will be the value after the request context path; this will result in the loss of protocol information (HTTP or HTTPS), so will cause problems if a redirect is being performed to change from HTTP to HTTPS or vice versa

Channel security (declaring which URLs must use HTTPS or HTTP)

More configuration details are [here](#)

Property	Default Value	Meaning
portMapper.httpPort	8080	the HTTP port your app uses
portMapper.httpsPort	8443	the HTTPS port your app uses
secureChannel.definition	none	Map of URL pattern to channel rule

IP address restrictions

More configuration details are [here](#)

Property	Default Value	Meaning
ipRestrictions	none	a Map of URL patterns to IP address patterns

Password encryption attributes

Property	Default	Description
password.algorithm	'SHA-256'	passwordEncoder Message Digest algorithm, see this page for options
password.encodeHashAsBase64	false	if true, Base64-encode the hashed password

HTTP Basic Authentication

More configuration details are [here](#)

Digest Authentication

More configuration details are [here](#)

Property	Default	Description
useBasicAuth	false	whether to use basic auth
basic.realmName	'Grails Realm'	the realm name displayed in the browser authentication popup

Switch User

More configuration details are [here](#)

Property	Default	Meaning
useSwitchUserFilter	false	whether to use the switch user filter
switchUser.switchUserUrl	'/j_spring_security_switch_user'	url to access (via GET or POST) to switch to another user
switchUser.exitUserUrl	'/j_spring_security_exit_user'	url to access to switch to another user
switchUser.targetUrl	the same as successHandler.defaultTargetUrl	the URL to redirect to after switching
switchUser.switchFailureUrl	the same as failureHandler.defaultFailureUrl	the URL to redirect to after an error attempting to switch

Session Fixation

More configuration details are [here](#)

Property	Default Value	
useSessionFixation	false	whether to use session fixation
sessionFixation.migrate	true	whether to copy the session attributes of the existing session to the new session after login
sessionFixation.alwaysCreateSession	false	whether to always create a session even if one didn't exist at the start of the request

Certificate (X509) login

More configuration details are [here](#)

Property	Default Value	Meaning
useX509	false	whether to support certificate-based logins
x509.continueFilterChainOnUnsuccessfulAuthentication	true	whether to proceed when an authentication attempt fails to allow other authentication mechanisms to process the request
x509.subjectDnRegex	'CN=(.*?),'	the regex for extracting the username from the certificate's subject name
x509.checkForPrincipalChanges	false	whether to re-extract the username from the certificate and check that it's still the current user when there's a valid Authentication already
x509.invalidateSessionOnPrincipalChange	true	whether to invalidate the session if the principal changed (based on a checkForPrincipalChanges check)

Other miscellaneous attributes

Property	Default Value	Meaning
active	true	whether the plugin is enabled
rejectIfNoRule	false	'strict' mode where an explicit grant is required to access any resource; if true make sure to add <code>IS_AUTHENTICATED_ANONYMOUSLY</code> for <code>/js/**</code> , <code>/css/**</code> , <code>/images/**</code> , <code>/login/**</code> , <code>/logout/**</code> , etc.
anon.key	'foo'	anonymousProcessingFilter key
anon.userAttribute	'anonymousUser, ROLE_ANONYMOUS'	anonymousProcessingFilter username and roles
atr.anonymousClass	AnonymousAuthenticationToken	anonymous token class
useHttpSessionEventPublisher	false	if true, an HttpSessionEventPublisher will be configured
cacheUsers	false	if true, logins are cached using an EhCache
useSecurityEventListener	false	if true, configure <code>SecurityEventListener</code> ; more details here
dao.reflectionSaltSourceProperty	none	which property to use for the reflection-based salt source; more details here
requestCache.onlyOnGet	false	whether to only cache a SavedRequest on GET requests
requestCache.createSession	true	whether caching <code>SavedRequest</code> can trigger creation of a session
authenticationDetails.authClass	WebAuthenticationDetails	the <code>Authentication</code> details class to use
roleHierarchy	none	hierarchical role definition; more details here
voterNames	'authenticatedVoter', 'roleVoter'	bean names of voters; more details here
providerNames	'daoAuthenticationProvider', 'anonymousAuthenticationProvider', 'rememberMeAuthenticationProvider'	bean names of authentication providers; more details here
securityConfigType	type of request mapping to use	one of <code>SecurityConfigType.Annotation</code> , <code>SecurityConfigType.RequestMapping</code> , <code>SecurityConfigType.InterceptUrlMap</code> ; more details here
controllerAnnotations.matcher	'ant'	whether to use an Ant-style URL matcher ('ant') or a Regexp ('regex')
controllerAnnotations.lowercase	true	whether to do URL comparisons using lowercase
controllerAnnotations.staticRules	none	extra rules that cannot be mapped using annotations
interceptUrlMap	none	request mapping definition when using <code>SecurityConfigType.InterceptUrlMap</code> ; more details here
registerLoggerListener	false	if true registers a LoggerListener which logs interceptor-related application events

7. Custom UserDetailsService

Hopefully the default configuraton plus the configurability exposed in `DefaultSecurityConfig.groovy` and `grails-app/conf/Config.groovy` enable most customization needs for your applications. However security is a large topic and there are many possible ways to secure an application.

When authenticating users from a database using [DaoAuthenticationProvider](#) (the default mode in the plugin if you haven't enabled OpenID, LDAP, etc.), an implementation of [UserDetailsService](#) is required. This class is responsible for returning a concrete implementation of [UserDetails](#). The plugin provides

`org.codehaus.groovy.grails.plugins.springsecurity.GormUserDetailsService` as its `UserDetailsService` implementation and

`org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser` (which extends Spring Security's [User](#)) as its `UserDetails` implementation.

You can extend or replace `GormUserDetailsService` with your own implementation by defining a bean in `grails-app/conf/spring/resources.groovy` (or `resources.xml`) with the same bean name, `userDetailsService`. This works because application beans are configured after plugin beans and there can only be one bean for each name.

Here's an example `UserDetails` and `UserDetailsService` implementation that adds the full name of the user domain class in addition to the standard information. If you extract extra data from your domain class, you'll be less likely to need to reload the user from the database - most of your common data can be kept along with your security credentials.

In this example we're adding in a `fullName` field. Keeping the full name cached avoids hitting the database just for that lookup. `GrailsUser` already adds the `id` value from the domain class to so we can do a more efficient database load of the user. If all you have is the username, then you need to call

`User.findByUsername(principal.username)`, but if you have the `id` you can call

`User.get(principal.id)`. Even if you have a unique index on the username database column, loading by primary key will usually be more efficient since it can take advantage of Hibernate's first-level and second-level caches.

There's really not much to implement other than your application-specific lookup code:

```
package com.foo.bar
import org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser
import org.springframework.security.core.GrantedAuthority
import org.springframework.security.core.userdetails.User
class MyUserDetails extends GrailsUser {
    final String fullName
    MyUserDetails(String username, String password, boolean enabled,
        boolean accountNonExpired, boolean credentialsNonExpired,
        boolean accountNonLocked,
        Collection<GrantedAuthority> authorities,
        long id, String fullName) {
        super(username, password, enabled, accountNonExpired,
            credentialsNonExpired, accountNonLocked, authorities, id)
        this.fullName = fullName
    }
}
```

```
package com.foo.bar
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.security.core.userdetails.UserDetailsService
class MyUserDetailsService implements UserDetailsService {
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // lookup user and data
        return new MyUserDetails(username, password, enabled,
            accountNonExpired, credentialsNonExpired,
            accountNonLocked, authorities, id, fullName)
    }
}
```

and to use it, register it in `grails-app/conf/spring/resources.groovy` like this:

```
beans = {  
    userDetailsService(com.foo.bar.MyUserDetailsService)  
}
```

Another option if you want to load users and roles from the database is to subclass

`org.codehaus.groovy.grails.plugins.springsecurity.GormUserDetailsService` - the methods are all protected so you can override whatever you want.

Also note that this approach works with all beans defined in

`SpringSecurityCoreGrailsPlugin.doWithSpring()` - you can replace or subclass any of the Spring beans to provide your own functionality when the standard extension mechanisms aren't sufficient.

Flushing the cached Authentication

If you store mutable data in your custom `UserDetails` implementation (for example full name like in this example), be sure to rebuild the `Authentication` if it changes. `springSecurityService` has a `reauthenticate` method that will do this for you:

```
class MyController {  
    def springSecurityService  
    def someAction {  
        def user = ...  
        // update user data  
        user.save()  
        springSecurityService.reauthenticate user.username  
        ...  
    }  
}
```


8. Ajax Authentication

The typical pattern of using web site authentication to access restricted pages involves intercepting access requests for secure pages, redirecting to a login page (possibly off-site) and redirecting back to the originally-requested page after a successful login. Each page can also have a login link to allow explicit logins at any time.

Another option is to also have a login link on each page and use Ajax and DHTML to present a login form within the current page in a popup. The form submits the authentication request via Ajax and displays success or error messages as appropriate.

The plugin has support for Ajax logins but you'll need to create your own GSP code. There are only a few necessary changes, and of course the sample code here is pretty basic so you should enhance it for your needs.

The approach I'll show here involves editing your template page(s) to show "You're logged in as ..." text if logged in and a login link if not, along with a hidden login form that's shown using DHTML.

Here's the updated `grails-app/views/layouts/main.gsp`:

```
<html>
<head>
<title><g:layoutTitle default="Grails" /></title>
<link rel="stylesheet" href="${resource(dir:'css',file:'main.css')}}" />
<link rel="shortcut icon" type="image/x-icon"
      href="${resource(dir:'images',file:'favicon.ico')}}" />
<g:layoutHead />
</head>
<body>
  <div id="spinner" class="spinner" style="display:none;">
    
  </div>
  <div id="grailsLogo" class="logo">
    <a href="http://grails.org">
      
    </a>
    <span id='loginLink' style='position: relative; margin-right: 30px; float: right'>
      <sec:ifLoggedIn>
        Logged in as <sec:username/> (<g:link controller='logout'>Logout</g:link>)
      </sec:ifLoggedIn>
      <sec:ifNotLoggedIn>
        <a href="#" onclick='showLogin(); return false;'>Login</a>
      </sec:ifNotLoggedIn>
    </span>
  </div>
  <g:javascript src='application.js' />
  <g:javascript library='prototype' />
  <g:javascript src='prototype/scriptaculous.js?load=effects' />
  <g:render template='/includes/ajaxLogin' />
  <g:layoutBody />
</body>
</html>
```

The changes to note here include:

- the prototype and scriptaculous libraries are included for Ajax support and to hide and show the login form
- there's an include of the template `/includes/ajaxLogin` (see the code below)
- there's a `` positioned in the top-right which shows the username and a logout link when logged in, and a login link otherwise

Here's the content of the login form template (`grails-app/views/includes/_ajaxLogin.gsp`) - note that the CSS and Javascript are shown inline, but should be extracted to their own static files:

```

<style>
#ajaxLogin {
    margin: 15px 0px; padding: 0px;
    text-align: center;
    display: none;
    position: absolute;
}
#ajaxLogin .inner {
    width: 260px;
    margin: 0px auto;
    text-align: left;
    padding: 10px;
    border-top: 1px dashed #499ede;
    border-bottom: 1px dashed #499ede;
    background-color: #EEF;
}
#ajaxLogin .inner .fheader {
    padding: 4px; margin: 3px 0px 3px 0; color: #2e3741; font-size: 14px; font-weight: bold;
}
#ajaxLogin .inner .cssform p {
    clear: left;
    margin: 0;
    padding: 5px 0 8px 0;
    padding-left: 105px;
    border-top: 1px dashed gray;
    margin-bottom: 10px;
    height: 1%;
}
#ajaxLogin .inner .cssform input[type='text'] {
    width: 120px;
}
#ajaxLogin .inner .cssform label {
    font-weight: bold;
    float: left;
    margin-left: -105px;
    width: 100px;
}
#ajaxLogin .inner .login_message {color: red;}
#ajaxLogin .inner .text_ {width: 120px;}
#ajaxLogin .inner .chk {height: 12px;}
.errorMessage { color: red; }
</style>
<div id='ajaxLogin'>
    <div class='inner'>
        <div class='fheader'>Please Login..</div>
        <form action='${request.contextPath}/j_spring_security_check' method='POST'
            id='ajaxLoginForm' name='ajaxLoginForm' class='cssform'>
            <p>
                <label for='username'>Login ID</label>
                <input type='text' class='text_' name='j_username' id='username' />
            </p>
            <p>
                <label for='password'>Password</label>
                <input type='password' class='text_' name='j_password' id='password' />
            </p>
            <p>
                <label for='remember_me'>Remember me</label>
                <input type='checkbox' class='chk' id='remember_me'
                    name='_spring_security_remember_me' />
            </p>
            <p>
                <a href='javascript:void(0)' onclick='authAjax(); return false;'>Login</a>
                <a href='javascript:void(0)' onclick='cancelLogin(); return false;'>Cancel</a>
            </p>
        </form>
        <div style='display: none; text-align: left;' id='loginMessage'></div>
    </div>
</div>
<script type='text/javascript'>
// center the form
Event.observe(window, 'load', function() {
    var ajaxLogin = $('ajaxLogin');
    $('ajaxLogin').style.left = ((document.body.getDimensions().width -
        ajaxLogin.getDimensions().width) / 2) + 'px';
    $('ajaxLogin').style.top = ((document.body.getDimensions().height -
        ajaxLogin.getDimensions().height) / 2) + 'px';
});
function showLogin() {
    $('ajaxLogin').style.display = 'block';
}
function cancelLogin() {
    Form.enable(document.ajaxLoginForm);
    Element.hide('ajaxLogin');
}

```

```

function authAjax() {
    Form.enable(document.ajaxLoginForm);
    Element.update('loginMessage', 'Sending request ...');
    Element.show('loginMessage');
    var form = document.ajaxLoginForm;
    var params = Form.serialize(form);
    Form.disable(form);
    new Ajax.Request(form.action, {
        method: 'POST',
        postBody: params,
        onSuccess: function(response) {
            Form.enable(document.ajaxLoginForm);
            var responseText = response.responseText || '[]';
            var json = responseText.evalJSON();
            if (json.success) {
                Element.hide('ajaxLogin');
                $('loginLink').update('Logged in as ' + json.username +
                                     ' (<%=link(controller: 'logout') { 'Logout' }%>));'');
            }
            else if (json.error) {
                Element.update('loginMessage', "<span class='errorMessage'>" +
                                             json.error + '</error>');
            }
            else {
                Element.update('loginMessage', responseText);
            }
        }
    });
}
</script>

```

The important aspects of this code are:

- the form posts to the same url as the regular form, `j_spring_security_check`; in fact the form is identical including the Remember Me checkbox, except that the submit button has been replaced with a hyperlink
- error messages are displayed within the popup `<div>`
- since there's no page redirect after successful login, the Javascript replaces the login link to give a visual indication that the user is logged in
- details of logout are not shown, but this is achieved by redirecting the user to `/j_spring_security_logout`

So how does it work?

Most Ajax libraries (Prototype, JQuery, and Dojo as of v2.1) include an `X-Requested-With` header that indicates that the request was made by `XMLHttpRequest` instead of being triggered by clicking a regular hyperlink or form submit button. The plugin uses this header to detect Ajax login requests, and uses subclasses of some of Spring Security's classes to use different redirect urls for Ajax requests than regular requests. Instead of showing full pages, `LoginController` has JSON-generating methods `ajaxSuccess()`, `ajaxDenied()`, and `authfail()` that generate JSON that the login Javascript code can use to appropriately display success or error messages.

You can see the Ajax-aware actions in `LoginController`, specifically `ajaxSuccess` and `ajaxDenied`, which send JSON responses that can be used by client JavaScript code. Also `authfail` will check whether the authentication request used Ajax and will render a JSON error response if it was.

9. Tutorials

9.1. Using Controller annotations to secure URLs

Create your Grails application

```
$ grails create-app bookstore
$ cd bookstore
```

Install the plugin

```
$ grails install-plugin spring-security-core
```

Create the User and Role domain classes

```
$ grails s2-quickstart com.testapp User Role
```

You can choose whatever names you like for your domain classes and the package they're in - these are just examples.



Depending on your database, some names might not be valid. This goes for any domain classes you create, but names for security seem to have an affinity towards trouble. So before you use names like "User" or "Group", make sure they are not reserved keywords in your database.

The script will create this User class:

```
package com.testapp
class User {
    String username
    String password
    boolean enabled
    boolean accountExpired
    boolean accountLocked
    boolean passwordExpired
    static constraints = {
        username blank: false, unique: true
        password blank: false
    }
    static mapping = {
        password column: 'password'
    }
    Set<Role> getAuthorities() {
        UserRole.findAllByUser(this).collect { it.role } as Set
    }
}
```

and this Role class:

```

package com.testapp
class Role {
    String authority
    static mapping = {
        cache true
    }
    static constraints = {
        authority blank: false, unique: true
    }
}

```

and a domain class that maps the many-to-many join class, UserRole:

```

package com.testapp
import org.apache.commons.lang.builder.HashCodeBuilder
class UserRole implements Serializable {
    User user
    Role role
    boolean equals(other) {
        if (!(other instanceof UserRole)) {
            return false
        }
        other.user?.id == user?.id &&
        other.role?.id == role?.id
    }
    int hashCode() {
        def builder = new HashCodeBuilder()
        if (user) builder.append(user.id)
        if (role) builder.append(role.id)
        builder.toHashCode()
    }
    static UserRole get(long userId, long roleId) {
        find 'from UserRole where user.id=:userId and role.id=:roleId',
            [userId: userId, roleId: roleId]
    }
    static UserRole create(User user, Role role, boolean flush = false) {
        new UserRole(user: user, role: role).save(flush: flush, insert: true)
    }
    static boolean remove(User user, Role role, boolean flush = false) {
        UserRole instance = UserRole.findByUserAndRole(user, role)
        instance ? instance.delete(flush: flush) : false
    }
    static void removeAll(User user) {
        executeUpdate 'DELETE FROM UserRole WHERE user=:user', [user: user]
    }
    static mapping = {
        id composite: ['role', 'user']
        version false
    }
}

```

It also creates some UI controllers and GSPs:

- grails-app/controllers/LoginController.groovy
- grails-app/controllers/LogoutController.groovy
- grails-app/views/auth.gsp
- grails-app/views/denied.gsp

Note that the script has edited grails-app/conf/Config.groovy and added the configuration for your domain classes. Make sure that the changes are correct.



These generated files are not part of the plugin - these are your application files. So you're free to edit them however you like - they're examples to get you started. They only contain the minimum needed for the plugin, but you're free to add whatever extra fields and methods you like.

The plugin has no support for CRUD actions and GSPs for your domain classes, the spring-security-ui plugin will supply a UI for those. So for now we'll create roles and users in

grails-app/conf/BootStrap.groovy

Create a controller that will be restricted by role

```
$ grails create-controller com.testapp.Secure
```

This will create grails-app/controllers/com/testapp/SecureController.groovy - add some output so we can verify that things are working:

```
package com.testapp
class SecureController {
    def index = {
        render 'Secure access only'
    }
}
```

Start the server

```
$ grails run-app
```

Before we secure the page, navigate to <http://localhost:8080/bookstore/secure> to verify that you can see the page without being logged in.

Shut down the app (using CTRL-C) and edit grails-app/conf/BootStrap.groovy to add the security objects that we need:

```
import com.testapp.Role
import com.testapp.User
import com.testapp.UserRole
class BootStrap {
    def springSecurityService
    def init = { servletContext ->
        def adminRole = new Role(authority: 'ROLE_ADMIN').save(flush: true)
        def userRole = new Role(authority: 'ROLE_USER').save(flush: true)
        String password = springSecurityService.encodePassword('password')
        def testUser = new User(username: 'me', enabled: true, password: password)
        testUser.save(flush: true)
        UserRole.create testUser, adminRole, true
        assert User.count() == 1
        assert Role.count() == 2
        assert UserRole.count() == 1
    }
}
```

Some things to note about what we did in BootStrap.groovy:

- we use springSecurityService to encrypt the password
- we're not using a traditional GORM many-to-many mapping for the User<->Role relationship, instead we're mapping the join table with the UserRole class. This is a performance optimization that will help significantly if many users have one or more common roles
- we're explicitly flushing the creates since BootStrap doesn't run in a transaction or OpenSessionInView

Edit grails-app/controllers/SecureController.groovy to import the annotation class and apply the annotation to restrict access:

```

package com.testapp
import grails.plugins.springsecurity.Secured
class SecureController {
    @Secured(['ROLE_ADMIN'])
    def index = {
        render 'Secure access only'
    }
}

```

or

```

package com.testapp
import grails.plugins.springsecurity.Secured
@Secured(['ROLE_ADMIN'])
class SecureController {
    def index = {
        render 'Secure access only'
    }
}

```

You can annotate the entire controller or individual actions. In this case since we only have one action we can do either.

Now run `grails run-app` again and navigate to <http://localhost:8080/bookstore/secure> and this time, you should be presented with the login page. Log in with the username and password you used for the test user, and you should again be able to see the secure page.

When logging in, you can test the Remember Me functionality. Check the checkbox, and once you've tested the secure page close your browser and re-open it. Navigate again to the secure page, and since you have a cookie stored, you shouldn't need to log in again. Logout at any time by navigating to <http://localhost:8080/bookstore/logout>

Creating a UI

If you would like to have a CRUD UI to work with users and roles, there are a few things you need to do beyond running `grails generate-all`.

The generated `UserController.save` action will look something like this:

```

def save = {
    def userInstance = new User(params)
    if (userInstance.save(flush: true)) {
        flash.message = "${message(code: 'default.created.message', args: [message(code: 'user', id: userInstance.id)])}"
        redirect(action: "show", id: userInstance.id)
    }
    else {
        render(view: "create", model: [userInstance: userInstance])
    }
}

```

This will store cleartext passwords and you won't be able to authenticate, so add a call to encrypt the password with `springSecurityService`:

```

class UserController {
    def springSecurityService
    ...
    def save = {
        def userInstance = new User(params)
        userInstance.password = springSecurityService.encodePassword(params.password)
        if (userInstance.save(flush: true)) {
            flash.message = "${message(code: 'default.created.message', args: [message(code: 'user', id: userInstance.id)])}"
            redirect(action: "show", id: userInstance.id)
        }
        else {
            render(view: "create", model: [userInstance: userInstance])
        }
    }
}

```

Similarly when updating you'll need to encrypt the password if it changes. Change this:

```
def update = {
  def userInstance = User.get(params.id)
  if (userInstance) {
    if (params.version) {
      def version = params.version.toLong()
      ...
    }
    userInstance.properties = params
    if (!userInstance.hasErrors() && userInstance.save(flush: true)) {
      flash.message = "${message(code: 'default.updated.message', args: [message(code: 'u
      redirect(action: "show", id: userInstance.id)
    }
    else {
      render(view: "edit", model: [userInstance: userInstance])
    }
  }
  else {
    flash.message = "${message(code: 'default.not.found.message', args: [message(code: 'us
    redirect(action: "list")
  }
}
```

to

```
def update = {
  def userInstance = User.get(params.id)
  if (userInstance) {
    if (params.version) {
      def version = params.version.toLong()
      ...
    }
    if (userInstance.password != params.password) {
      params.password = springSecurityService.encodePassword(params.password)
    }
    userInstance.properties = params
    if (!userInstance.hasErrors() && userInstance.save(flush: true)) {
      if (springSecurityService.loggedIn &&
          springSecurityService.principal.username == userInstance.username) {
        springSecurityService.reauthenticate userInstance.username
      }
      flash.message = "${message(code: 'default.updated.message', args: [message(code: 'u
      redirect(action: "show", id: userInstance.id)
    }
    else {
      render(view: "edit", model: [userInstance: userInstance])
    }
  }
  else {
    flash.message = "${message(code: 'default.not.found.message', args: [message(code: 'us
    redirect(action: "list")
  }
}
```

Note that there's also a call to `springSecurityService.reauthenticate()` to ensure that the cached Authentication stays current.

10. Extending and configuring the plugin

10.1. Filters

There are a few different approaches to configuring filter chain(s). The default way is to use configuration attributes to determine which extra filters to use (e.g. Basic Auth, Switch User, etc.) and add these to the 'core' filters. For example, setting `grails.plugins.springsecurity.useSwitchUserFilter = true` adds `switchUserProcessingFilter` to the filter chain (and in the correct order). The filter chain built here is applied to all URLs, so if you need more flexibility then you further refine it using `filterChain.chainMap` as discussed below.

filterNames

To define custom filters, remove a core filter from the chain (not recommended), or otherwise have control over the filter chain, you can specify the `filterNames` property as a list of strings, e.g.

```
grails.plugins.springsecurity.filterNames = [
    'httpSessionContextIntegrationFilter', 'logoutFilter', 'authenticationProcessingFilter',
    'myCustomProcessingFilter', 'rememberMeProcessingFilter', 'anonymousProcessingFilter',
    'exceptionTranslationFilter', 'filterInvocationInterceptor'
]
```

This will create a filter chain corresponding to the Spring beans with the specified names. As with the default approach, the filter chain built here is applied to all URLs.

chainMap

You can also define which filters to applied to different URL patterns using the `filterChain.chainMap` attribute. This involves defining a Map which specifies one or more lists of filter bean names, each with a corresponding URL pattern, e.g.:

```
grails.plugins.springsecurity.filterChain.chainMap = [
    '/urlpattern1/**': 'filter1,filter2,filter3,filter4',
    '/urlpattern2/**': 'filter1,filter3,filter5',
    '/**': 'JOINED_FILTERS',
]
```

In this example, four filters are applied to URLs matching `/urlpattern1/**` and three different filters are applied to URLs matching `/urlpattern2/**`. In addition the special token `JOINED_FILTERS` is applied to all URLs. This is a convenient way to specify that all defined filters (configured either with configuration rules like `useSwitchUserFilter` or explicitly using `filterNames`) should apply to this pattern.

Note that the order of the mappings is important. Each URL will be tested in order from top to bottom to find the first matching one. So we need a `/**` catch-all rule at the end for URLs that aren't don't match one of the earlier rules.

clientRegisterFilter

An alternative to setting the `filterNames` property is

`org.codehaus.groovy.grails.plugins.springsecurity.SpringSecurityUtils.clientRegisterFilter`. This allows you to add a custom filter to the chain at a specified position. Each of the standard filters has a corresponding position in the chain (see

`org.codehaus.groovy.grails.plugins.springsecurity.SecurityFilterPosition` for details). So if you have created an application-specific filter, register it in `grails-app/conf/spring/resources.groovy`:

```
beans = {
    myFilter(com.mycompany.myapp.MyFilter) {
        // properties
    }
}
```

and then register it in `grails-app/conf/BootStrap.groovy`:

```
import org.codehaus.groovy.grails.plugins.springsecurity.SecurityFilterPosition
import org.codehaus.groovy.grails.plugins.springsecurity.SpringSecurityUtils
class BootStrap {
    def init = { servletContext ->
        SpringSecurityUtils.clientRegisterFilter(
            'myFilter', SecurityFilterPosition.OPENID_FILTER.order + 10)
    }
}
```

This bootstrap code will register your filter just after the Open ID filter (if it's configured). You cannot register a filter in the same position as another, so it's a good idea to add a small delta to its position to put it after or before a filter that it should be next to in the chain. Note that the Open ID filter position is just an example - add your filter in whatever position makes sense.

10.2. Basic and Digest Auth

Basic Auth

To use [HTTP Basic Authentication](#) in your application set the `useBasicAuth` attribute to `true`. You should also change the `basic.realmName` from its default value to one that's appropriate for your application, e.g.

```
grails.plugins.springsecurity.useBasicAuth = true
grails.plugins.springsecurity.basic.realmName = "Ralph's Bait and Tackle"
```

With this in place, users will be prompted with the standard browser login dialog instead of being redirected to a login page.

Digest Auth

[Digest Authentication](#) is similar to Basic auth but is more secure in that it doesn't send your password in obfuscated cleartext. It looks just like Basic auth in practice - you get the same browser popup dialog when you authenticate. But since the credential transfer is genuinely encrypted (instead of just Base64-encoded as with Basic auth) you don't need to use SSL to guard your logins.

There is one issue with using Digest auth - by default you must store cleartext passwords in your database. This is because the browser encrypts your password along with the username and Realm name, and this is compared to the password encrypted using the same algorithm during authentication. The browser doesn't know about your MessageDigest algorithm or salt source, so to encrypt them the same way you need to load a cleartext password from the database.

However the plugin does provide an alternative, but it has no configuration options (in particular the digest algorithm cannot be changed). If `digest.useCleartextPasswords` is `false` (the default) then the `passwordEncoder` bean will be replaced with an instance of

`grails.plugins.springsecurity.DigestAuthPasswordEncoder`. This encoder uses the same approach as the browser, i.e. it combines your password along with your username and Realm name essentially as a salt, and encrypting with MD5. MD5 is not recommended in general but given the typical size of the salt it is reasonably safe to use.

The only required attribute is `useDigestAuth` which you must set to `true`, but you'll probably also want to change the realm name too:


```
grails.plugins.springsecurity.useDigestAuth = true
grails.plugins.springsecurity.digest.realmName = "Ralph's Bait and Tackle"
```

There are a few other configuration options that you'll be less likely to need to change.

Property	Default Value	Meaning
digest.realmName	'Grails Realm'	The realm name that's displayed in the browser popup
digest.key	'changeme'	A key that's used to build the nonce for authentication; should be changed but that's not required
digest.nonceValiditySeconds	300	the duration that a nonce stays valid
digest.passwordAlreadyEncoded	false	whether you're managing the password encryption yourself
digest.createAuthenticatedToken	false	if true creates an authenticated UsernamePasswordAuthenticationToken to avoid loading the user from the database twice (but this skips the isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired(), isEnabled() checks so this is not advised)
digest.useCleartextPasswords	false	if true then a cleartext password encoder will be used (not recommended); if false then passwords encrypted by DigestAuthPasswordEncoder will be stored in the database

10.3. Switch User

To enable a user to switch from their current Authentication to another user's, set the `useSwitchUserFilter` attribute to `true`. This feature is similar to the 'su' command in Unix, for example to allow an admin to act as a regular user to perform some actions, and then switch back.

 This is a very powerful feature since it allows you full access to whatever the switched-to user can access without knowing their password, so it's very important that you limit who can use this feature. The best way to do this is to guard the user switch URL with a role, e.g. `ROLE_SWITCH_USER`, `ROLE_ADMIN`, etc.

Switching to another user

To switch to another user, typically you would create a form that submits to `/j_spring_security_switch_user`:

```
<sec:ifAllGranted roles='ROLE_SWITCH_USER'>
  <form action='/j_spring_security_switch_user' method='POST'>
    Switch to user: <input type='text' name='j_username' /> <br />
    <input type='submit' value='Switch' />
  </form>
</sec:ifAllGranted>
```

Note that here the form is guarded by a check that the logged-in user has `ROLE_SWITCH_USER` and isn't shown otherwise. In addition you'll want to guard the user switch URL and the approach depends on which mapping scheme you're using. If you're using annotations, add a rule to the `controllerAnnotations.staticRules` attribute:

```
grails.plugins.springsecurity.controllerAnnotations.staticRules = [
  ...
  '/j_spring_security_switch_user': ['ROLE_SWITCH_USER', 'IS_AUTHENTICATED_FULLY']
]
```

If you're using Requestmaps, create one like this (e.g. in BootStrap):

```
new Requestmap(url: '/j_spring_security_switch_user',
               configAttribute: 'ROLE_SWITCH_USER,IS_AUTHENTICATED_FULLY').save(flush: true)
```

and if you're using the `Config.groovy` map, add it there:

```
grails.plugins.springsecurity.interceptUrlMap = [
    ...
    '/j_spring_security_switch_user': ['ROLE_SWITCH_USER', 'IS_AUTHENTICATED_FULLY']
]
```

Switching back

To resume as the original user, navigate to `/j_spring_security_exit_user`.

```
<sec:ifSwitched>
<a href='${request.contextPath}/j_spring_security_exit_user'>
    Resume as <sec:switchedUserOriginalUsername/>
</a>
</sec:ifSwitched>
```

Configuration

In addition you can customize the URLs that are used for this feature, although this is rarely necessary:

```
grails.plugins.springsecurity.switchUser.switchUserUrl = ...
grails.plugins.springsecurity.switchUser.exitUserUrl = ...
grails.plugins.springsecurity.switchUser.targetUrl = ...
grails.plugins.springsecurity.switchUser.switchFailureUrl = ...
```

Name	Default	Meaning
<code>switchUser.switchUserUrl</code>	<code>'/j_spring_security_switch_user'</code>	url to access (via GET or POST) to switch to another user
<code>switchUser.exitUserUrl</code>	<code>'/j_spring_security_exit_user'</code>	url to access to switch to another user
<code>switchUser.targetUrl</code>	the same as <code>successHandler.defaultTargetUrl</code>	the URL to redirect to after switching
<code>switchUser.switchFailureUrl</code>	the same as <code>failureHandler.defaultFailureUrl</code>	the URL to redirect to after an error attempting to switch

Example code

One approach to supporting this feature would be to add code to one or more of your GSP templates. In this example the current username is displayed, and if the user has switched from another (using the `sec:ifSwitched` tag) then a 'resume' link is displayed. If not, and the user has the required role, then a form is displayed to allow input of the username of the user to switch to:

```

<sec:ifLoggedIn>
Logged in as <sec:username/>
</sec:ifLoggedIn>
<sec:ifSwitched>
<a href='${request.contextPath}/j_spring_security_exit_user'>
  Resume as <sec:switchedUserOriginalUsername/>
</a>
</sec:ifSwitched>
<sec:ifNotSwitched>
  <sec:ifAllGranted roles='ROLE_SWITCH_USER'>
    <form action='${request.contextPath}/j_spring_security_switch_user' method='POST'>
      Switch to user: <input type='text' name='j_username' /><br/>
      <input type='submit' value='Switch' />
    </form>
  </sec:ifAllGranted>
</sec:ifNotSwitched>

```

10.4. Session Fixation

To guard against [session-fixation attacks](#) set the `useSessionFixation` attribute to `true`:

```
grails.plugins.springsecurity.useSessionFixation = true
```

When this is active, on successful authentication a new HTTP session will be created and the previous session's attributes will be copied into it. This way, if you were to start your session by clicking a link that was generated by someone trying to hack your account which contained an active session id, you would no longer be sharing the previous session after login - you'd have your own.

This is less of an issue now that Grails by default does not include `jsessionid` in URLs (see [this JIRA issue](#)) but it's still a good idea to use this feature regardless.

There are a couple of configuration options:

Name	Default Value	
<code>sessionFixation.migrate</code>	<code>true</code>	whether to copy the session attributes of the existing session to the new session after login
<code>sessionFixation.alwaysCreateSession</code>	<code>false</code>	whether to always create a session even if one didn't exist at the start of the request

10.5. Salted passwords

The plugin uses encrypted passwords using whatever digest algorithm you specify, but for enhanced protection against dictionary attacks you should use a salt in addition to digest encryption.

There are two approaches to using salted passwords in the plugin - defining a field in the `UserDetails` class to access by reflection, or by directly implementing [SaltSource](#) yourself.

dao.reflectionSaltSourceUserProperty

For the first approach, you need to set the `dao.reflectionSaltSourceUserProperty` configuration property, e.g.

```
grails.plugins.springsecurity.dao.reflectionSaltSourceUserProperty = 'username'
```

Note that this is a property of the `UserDetails` class, which by default is an instance of `org.codehaus.groovy.grails.plugins.springsecurity.GrailsUser` which extends the standard Spring Security [User](#), and not your 'person' domain class. This limits the available fields unless you use a custom `UserDetailsService` as described [here](#).

As long as the username won't change, it's a good choice for the salt. If you choose a property that the user can change, then they won't be able to log in again after changing it unless you re-encrypt their password with the new

value, so it's best to use a property that doesn't change.

Another option is to generate a random salt when creating users and store this in the database by adding a new field to the 'person' class. This requires a custom `UserDetailsService` since you need a custom `UserDetails` implementation that also has a 'salt' property, but this is more flexible and works in cases where users can change their username.

Custom SaltSource

To have full control over the process, you can implement the `SaltSource` interface and replace the plugin's implementation with your own by defining a bean in `grails-app/conf/spring/resources.groovy` with the name `saltSource`:

```
beans = {
    saltSource(com.foo.bar.MySaltSource) {
        // set properties
    }
}
```

Encrypting passwords

Regardless of the implementation, you need to be aware of what value to use for a salt when creating or updating users, e.g. in a `UserController`'s save or update action. Then encrypting the password, you use the two-parameter version of `springSecurityService.encodePassword()`, e.g.

```
class UserController {
    def springSecurityService
    def save = {
        def userInstance = new User(params)
        userInstance.password = springSecurityService.encodePassword(
            params.password, userInstance.username)
        if (!userInstance.save(flush: true)) {
            render view: 'create', model: [userInstance: userInstance]
            return
        }
        flash.message = "The user was created"
        redirect action: show, id: userInstance.id
    }
    def update = {
        def userInstance = User.get(params.id)
        if (userInstance.password != params.password) {
            params.password = springSecurityService.encodePassword(
                params.password, userInstance.username)
        }
        userInstance.properties = params
        if (!userInstance.save(flush: true)) {
            render view: 'edit', model: [userInstance: userInstance]
            return
        }
        if (springSecurityService.loggedIn &&
            springSecurityService.principal.username == userInstance.username) {
            springSecurityService.reauthenticate userInstance.username
        }
        flash.message = "The user was updated"
        redirect action: show, id: userInstance.id
    }
}
```

10.6. Certificate (X509) login

Another authentication mechanism supported by Spring Security is certificate-based, or "mutual authentication". To use this you must use HTTPS and configure the server to require a client certificate (ordinarily only the server provides a certificate). Your username will be extracted from the client certificate if it's valid, and you'll be considered "pre-authenticated". As long as there is a corresponding user in the database with that username, your authentication will succeed and you won't be asked for a password, and the your `Authentication` will contain the authorities associated with your username.

There are a few configuration options available for this feature:

Property	Default Value	Meaning
useX509	false	whether to support certificate-based logins
x509.continueFilterChainOnUnsuccessfulAuthentication	true	whether to proceed when an authentication attempt fails to allow other authentication mechanisms to process the request
x509.subjectDnRegex	'CN=(.*?),'	the regex for extracting the username from the certificate's subject name
x509.checkForPrincipalChanges	false	whether to re-extract the username from the certificate and check that it's still the current user when there's a valid Authentication already
x509.invalidateSessionOnPrincipalChange	true	whether to invalidate the session if the principal changed (based on a checkForPrincipalChanges check)

The details around configuring your server for SSL and configuring browser certificates are beyond the scope of this document, but if you're using Tomcat you should look at its [SSL documentation](#). Also, one quick way to get a test environment working is to use the instructions from [this discussion at Stack Overflow](#).

10.7. Channel security

If you are using SSL for some or all of the URLs in your app, you can configure which require HTTP and which require HTTPS using channel security.

To configure this, build a Map under the `secureChannel.definition` key, where the keys are URL patterns, and the values are one of `REQUIRES_SECURE_CHANNEL`, `REQUIRES_INSECURE_CHANNEL`, or `ANY_CHANNEL`:

```
grails.plugins.springsecurity.secureChannel.definition = [
  '/login/**': 'REQUIRES_SECURE_CHANNEL',
  '/maps/**': 'REQUIRES_INSECURE_CHANNEL',
  '/images/login/**': 'REQUIRES_SECURE_CHANNEL'
  '/images/**': 'ANY_CHANNEL'
]
```

URLs will be checked in order, so be sure to put more specific rules before less specific. In the example above `/images/login/**` is more specific than `/images/**` so it appears first in the configuration.

10.8. IP Address Restrictions

Ordinarily it's sufficient to guard URLs with roles, but the plugin provides an extra layer of security with its ability to restrict by IP address. One use for this would be to guard an admin-only part of your site to only be accessible from IP addresses of the local LAN or VPN, e.g. 192.168.1.xxx or 10.xxx.xxx.xxx. This can also be done at your firewall and/or routers, but it can be convenient to have this encapsulated within your application.

To use this feature, specify an `ipRestrictions` configuration map, where the keys are URL patterns, and the values are IP address patterns that can access those URLs. The IP patterns can be single-valued strings, or multi-valued lists of strings and can use [CIDR](#) masks, and can specify either IPv4 or IPv6 patterns. For example, given this configuration:

```
grails.plugins.springsecurity.ipRestrictions = [
  '/pattern1/**': '123.234.345.456',
  '/pattern2/**': '10.0.0.0/8',
  '/pattern3/**': ['10.10.200.42', '10.10.200.63']
]
```

then `pattern1` URLs can only be access from the external address 123.234.345.456, `pattern2` URLs can only

be accessed from a 10.xxx.xxx.xxx intranet address, and pattern3 URLs can only be accessed from 10.10.200.42 or 10.10.200.63. All other URL patterns are accessible from any IP address.

Note that all addresses can always be accessed from localhost regardless of IP pattern, primarily to support local development mode.



You cannot compare IPv4 and IPv6 addresses, so if your server supports both, you need to specify the IP patterns using whichever address format is actually being used, otherwise the filter will throw exceptions. One option is to set the `java.net.preferIPv4Stack` system property, e.g. by adding it to `JAVA_OPTS` or `GRAILS_OPTS` as `-Djava.net.preferIPv4Stack=true`

10.9. Logout Handlers

Spring Security allows you to register a list of logout handlers (implementing the [LogoutHandler](#) interface) that will be called when a user explicitly logs out.

By default, a `securityContextLogoutHandler` bean is registered to clear the [SecurityContextHolder](#). Also, if you're not using Facebook or OpenID, `rememberMeServices` bean is registered to reset your cookie (Facebook and OpenID authenticate externally so we don't have access to the password to create a remember-me cookie). If you're using Facebook, a `facebookLogoutHandler` is registered to reset its session cookies.

To customize this list, you define a `logout.handlerNames` attribute with a list of bean names. The beans must be declared either by the plugin or by you in `resources.groovy` or `resources.xml`. So if you have a custom `MyLogoutHandler` in `resources.groovy`, e.g.

```
beans = {
    myLogoutHandler(com.foo.MyLogoutHandler) {
        // attributes
    }
}
```

then you'd register it in `grails-app/conf/Config.groovy` as:

```
grails.plugins.springsecurity.logout.handlerNames = [
    'rememberMeServices', 'securityContextLogoutHandler', 'myLogoutHandler'
]
```

10.10. Voters

Spring Security allows you to register a list of voters (implementing the [AccessDecisionVoter](#) interface) that check that a successful authentication is applicable for the current request. By default a `roleVoter` bean is registered to ensure users have the required roles for the request, and an `authenticatedVoter` bean is registered to support `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY` tokens.

To customize this list, you define a `voterNames` attribute with a list of bean names. The beans must be declared either by the plugin, or yourself in `resources.groovy` or `resources.xml`. So if you have a custom `MyAccessDecisionVoter` in `resources.groovy`, e.g.

```
beans = {
    myAccessDecisionVoter(com.foo.MyAccessDecisionVoter) {
        // attributes
    }
}
```

then you'd register it in `grails-app/conf/Config.groovy` as:


```
grails.plugins.springsecurity.voterNames = [
    'authenticatedVoter', 'roleVoter', 'myAccessDecisionVoter'
]
```

10.11. Authentication Providers

The plugin registers authentication providers (implementing the [AuthenticationProvider](#) interface) that perform authentication. By default, three are registered: `daoAuthenticationProvider` to authenticate using the User and Role database tables, `rememberMeAuthenticationProvider` to login with a remember-me cookie, and `anonymousAuthenticationProvider` to create an 'anonymous' authentication if no other provider authenticates.

To customize this list, you define a `providerNames` attribute with a list of bean names. The beans must be declared either by the plugin, or yourself in `resources.groovy` or `resources.xml`. So if you have a custom `MyAuthenticationProvider` in `resources.groovy`, e.g.

```
beans = {
    myAuthenticationProvider(com.foo.MyAuthenticationProvider) {
        // attributes
    }
}
```

then you'd register it in `grails-app/conf/Config.groovy` as:

```
grails.plugins.springsecurity.providerNames = ['myAuthenticationProvider',
                                              'anonymousAuthenticationProvider',
                                              'rememberMeAuthenticationProvider']
```

10.12. Hierarchical Roles

Hierarchical roles are a convenient way to reduce some clutter in your request mappings. For example, if you have several types of 'admin' roles and any one of them can be used to access some URL pattern, then without using hierarchical roles you need to specify all of them:

```
package com.mycompany.myapp
import grails.plugins.springsecurity.Secured
class SomeController {
    @Secured(['ROLE_ADMIN', 'ROLE_FINANCE_ADMIN', 'ROLE_SUPERADMIN'])
    def someAction = {
        ...
    }
}
```

But if you have a business rule that says that being granted `ROLE_FINANCE_ADMIN` implies being granted `ROLE_ADMIN`, and that being granted `ROLE_SUPERADMIN` implies being granted `ROLE_FINANCE_ADMIN`, then you can express that hierarchy as

```
grails.plugins.springsecurity.roleHierarchy = '''
    ROLE_SUPERADMIN > ROLE_FINANCE_ADMIN
    ROLE_FINANCE_ADMIN > ROLE_ADMIN
'''
```

Then you can simplify your mappings by specifying only the roles that are required:

```

package com.mycompany.myapp
import grails.plugins.springsecurity.Secured
class SomeController {
    @Secured(['ROLE_ADMIN'])
    def someAction = {
        ...
    }
}

```

You can also reduce the number of granted roles in the database using this approach - where previously you had to grant `ROLE_SUPERADMIN`, `ROLE_FINANCE_ADMIN`, and `ROLE_ADMIN`, now you only need to grant `ROLE_SUPERADMIN`.

10.13. Account Locking and Forcing Password Change

Spring Security supports four ways of disabling a user account. When you attempt to log in, the `UserDetailsService` implementation creates an instance of `UserDetails` which has these accessors:

- `isAccountNonExpired()`
- `isAccountNonLocked()`
- `isCredentialsNonExpired()`
- `isEnabled()`

and if you use the [s2-quickstart](#) script to create a user domain class, it creates a class with corresponding properties to manage this state.

When one of these accessors returns `false` (i.e. `accountExpired`, `accountLocked`, or `passwordExpired` is `true` or `enabled` is `false`) a corresponding exception is thrown:

Accessor	Property	Exception
<code>isAccountNonExpired()</code>	<code>accountExpired</code>	AccountExpiredException
<code>isAccountNonLocked()</code>	<code>accountLocked</code>	LockedException
<code>isCredentialsNonExpired()</code>	<code>passwordExpired</code>	CredentialsExpiredException
<code>isEnabled()</code>	<code>enabled</code>	DisabledException

You can configure an exception mapping in `Config.groovy` to associate a URL to any or all of these exceptions to determine where to redirect after a failure, e.g.

```

grails.plugins.springsecurity.failureHandler.exceptionMappings = [
    'org.springframework.security.authentication.LockedException': '/user/account',
    'org.springframework.security.authentication.DisabledException': '/user/account',
    'org.springframework.security.authentication.AccountExpiredException': '/user/account',
    'org.springframework.security.authentication.CredentialsExpiredException': '/user/password'
]

```

Without a mapping for a particular exception, the user will be redirected to the standard login fail page (by default `/login/authfail`) like they would if they had entered a bad password, but they'll see an error message from this table:

Property	Default
<code>errors.login.disabled</code>	"Sorry, your account is disabled."
<code>errors.login.expired</code>	"Sorry, your account has expired."
<code>errors.login.passwordExpired</code>	"Sorry, your password has expired."
<code>errors.login.locked</code>	"Sorry, your account is locked."
<code>errors.login.fail</code>	"Sorry, we were not able to find a user with that username and password."

Any of these can be customized by setting the corresponding property in `Config.groovy`, e.g.

```
grails.plugins.springsecurity.errors.login.locked = "None shall pass."
```

You can use this functionality to manually lock a user's account or expire the password, but another option would be to automate the process. For example you could use the [Quartz plugin](#) to periodically expire everyone's password and force them to go to a page where they update it. You could also keep track of the date when the users change their passwords and use a Quartz job to expire their passwords once the password is older than some fixed max age.

User cache

If configured, Spring Security will cache `UserDetails` instances to save trips to the database. This is managed by the `cacheUsers` configuration property, and it defaults to `false` in the plugin but you can enable it if you wish. In general this is a minor optimization since there will most likely be only two small queries during login; one to load the user, and one to load the authorities.

If you enable this feature, you must remove any cached instances after making a change that affects login. If you don't, even though a user's account is locked or disabled, logins will still succeed since the database will be bypassed. By removing the cached data, you force them to go to the database and retrieve the latest updates.

Here is a sample Quartz job that demonstrates how you might find and disable users with passwords that are too old:

```
package com.mycompany.myapp
class ExpirePasswordsJob {
    static triggers = {
        cron name: 'myTrigger', cronExpression: '0 0 0 * * ?' // midnight daily
    }
    def userCache
    void execute() {
        def users = User.executeQuery(
            'from User u where u.passwordChangeDate <= :cutoffDate',
            [cutoffDate: new Date() - 180])
        for (user in users) {
            // flush each separately so one failure doesn't rollback all of the others
            try {
                user.passwordExpired = true
                user.save(flush: true)
                userCache.removeUserFromCache user.username
            }
            catch (e) {
                log.error "problem expiring password for user $user.username : $e.message", e
            }
        }
    }
}
```