

Spring Security Core Plugin - Reference Documentation

Authors: Burt Beckwith, Beverley Talbott

Version: 2.0-RC2

Table of Contents

- 1** Introduction to the Spring Security Plugin
 - 1.1** Configuration Settings Now in Config.groovy
 - 1.2** Getting Started
- 2** What's New in Version 2.0
- 3** Required and Optional Domain Classes
 - 3.1** Person Class
 - 3.2** Authority Class
 - 3.3** PersonAuthority Class
 - 3.4** Requestmap Class
- 4** Configuring Request Mappings to Secure URLs
 - 4.1** Defining Secured Annotations
 - 4.2** Simple Map in Config.groovy
 - 4.3** Requestmap Instances Stored in the Database
 - 4.4** Using Expressions to Create Descriptive, Fine-Grained Rules
- 5** Helper Classes
 - 5.1** SecurityTagLib
 - 5.2** SpringSecurityService
 - 5.3** SpringSecurityUtils
- 6** Events
 - 6.1** Event Notification
 - 6.2** Registering an Event Listener
 - 6.3** Registering Callback Closures
- 7** User, Authority (Role), and Requestmap Properties
- 8** Authentication
 - 8.1** Basic and Digest Authentication
 - 8.2** Certificate (X509) Login Authentication
 - 8.3** Remember-Me Cookie
 - 8.4** Ajax Authentication
- 9** Authentication Providers
- 10** Custom UserDetailsService
- 11** Password and Account Protection
 - 11.1** Password Hashing

- 11.2** Salted Passwords
- 11.3** Account Locking and Forcing Password Change
- 12** URL Properties
- 13** Hierarchical Roles
- 14** Switch User
- 15** Filters
- 16** Channel Security
- 17** IP Address Restrictions
- 18** Session Fixation Prevention
- 19** Logout Handlers
- 20** Voters
- 21** Miscellaneous Properties
- 22** Tutorials
 - 22.1** Using Controller Annotations to Secure URLs
 - 22.2** Migration From the Acegi Plugin
- 23** Controller MetaClass Methods
- 24** Internationalization
- 25** Migrating from the Acegi plugin
 - 25.1** Differences Between the Spring Security and Acegi Plugins
 - 25.2** Migrating from the Acegi Plugin

1 Introduction to the Spring Security Plugin

The Spring Security plugin simplifies the integration of [Spring Security](#) (formerly Acegi Security) into Grails applications. The plugin provides sensible defaults with many configuration options for customization. Nearly everything is configurable or replaceable in the plugin and in Spring Security itself, which makes extensive use of interfaces.

This guide documents configuration defaults and describes how to configure and extend the Spring Security plugin for Grails applications.

Release History and Acknowledgment

- October 4, 2013
 - 2.0-RC2 release
 - [JIRA Issues](#)
- April 6, 2012
 - 1.2.7.3 release
 - [JIRA Issues](#)
- February 2, 2012
 - 1.2.7.2 release
 - [JIRA Issues](#)
- January 18, 2012
 - 1.2.7.1 release
 - [JIRA Issues](#)
- December 30, 2011
 - 1.2.7 release
 - [JIRA Issues](#)
- December 2, 2011
 - 1.2.6 release
 - [JIRA Issues](#)
- December 1, 2011
 - 1.2.5 release
- October 18, 2011
 - 1.2.4 release
- October 15, 2011
 - 1.2.3 release
- October 15, 2011

- 1.2.2 release
- [JIRA Issues](#)
- August 17, 2011
 - 1.2.1 release
 - [JIRA Issues](#)
- July 31, 2011
 - 1.2 release
 - [JIRA Issues](#)
- May 23, 2011
 - 1.1.3 release
 - [JIRA Issues](#)
- February 26, 2011
 - 1.1.2 release
- February 26, 2011
 - 1.1.1 release
 - [JIRA Issues](#)
- August 8, 2010
 - 1.1 release
 - [JIRA Issues](#)
- August 1, 2010
 - 1.0.1 release
- July 27, 2010
 - 1.0 release
 - [JIRA Issues](#)
- July 16, 2010
 - 0.4.2 release
 - [JIRA Issues](#)
- June 29, 2010
 - 0.4.1 release
 - [JIRA Issues](#)
- June 21, 2010
 - 0.4 release
 - [JIRA Issues](#)


- May 12, 2010
 - 0.3.1 release
 - [JIRA Issues](#)
- May 12, 2010
 - 0.3 release
 - [JIRA Issues](#)
- May 2, 2010
 - 0.2 release
- April 27, 2010
 - initial 0.1 release

This plugin is based on work done for the [Acegi](#) plugin by Tsuyoshi Yamamoto.

1.1 Configuration Settings Now in Config.groovy

The Spring Security plugin maintains its configuration in the standard `Config.groovy` file. Default values are in the plugin's `grails-app/conf/DefaultSecurityConfig.groovy` file, and you add application-specific values to the `grails-app/conf/Config.groovy` file. The two configurations will be merged, with application values overriding the defaults.

This structure enables environment-specific configuration such as, for example, fewer structure-restrictive security rules during development than in production. Like any environment-specific configuration parameters, you wrap them in an `environments` block.

 The plugin's configuration values all start with `grails.plugin.springsecurity` to distinguish them from similarly named options in Grails and from other plugins. You must specify all property overrides with the `grails.plugin.springsecurity` suffix. For example, you specify the attribute `password.algorithm` as:

```
grails.plugin.springsecurity.password.algorithm='bcrypt'
```

in `Config.groovy`

1.2 Getting Started

If you will be migrating from the Acegi to the Spring Security plugin, see [Migrating from the Acegi Plugin](#).

Once you install the plugin, you simply run the initialization script, [s2-quickstart](#), and make any required configuration changes in `Config.groovy`. The plugin registers filters in `web.xml`, and also configures the Spring beans in the application context that implement various pieces of functionality. Ivy determines which jar files to use.

To get started using the Spring Security plugin with your Grails application, see [Tutorials](#).

You do not need to know much about Spring Security to use the plugin, but it can be helpful to understand the underlying implementation. See [the Spring Security documentation](#).

2 What's New in Version 2.0

There are many changes in the 2.x versions of the plugin from the older approaches in 1.x.

Package changes

All classes are now in the `grails.plugin.springsecurity` package or a subpackage. The names tend to correspond to the analogous Spring Security classes where appropriate, for example `MutableLogoutFilter` is in the `grails.plugin.springsecurity.web.authentication.logout` package to correspond with the `org.springframework.security.web.authentication.logout` package.

Some of the changes were more subtle though, for example all classes in the old `grails.plugins.springsecurity` packages and subpackages are now in `grails.plugin.springsecurity`, only one character different. This will result in a non-trivial upgrade process for your applications, but that is a benefit as it will hopefully point you at other important changes you might have otherwise missed.

Configuration prefix changes

The prefix used in `Config.groovy` for the plugin's configuration settings has changed from `grails.plugins.springsecurity` to `grails.plugin.springsecurity`.

More aggressively secure by default

In 1.x it was assumed that defaulting pages to not be secured, and configuring guarded URLs as needed, was a more pragmatic approach. Now however, all URLs are blocked unless there is a request mapping rule. The assumption behind this change is that if you forget to guard a new URL, it can take a long time to discover that users had access, whereas if you forget when using the "pessimistic" approach, nobody can access the URL and the error will be quickly discovered. This approach is more work, but much safer.

This is described in more detail [here](#).

Logout POST only

By default only POST requests are allowed to trigger a logout. To allow GET access, add this

```
grails.plugin.springsecurity.logout.postOnly = false
```

bcrypt by default

The default password hashing algorithm is now bcrypt since it is a very robust hashing approach. [PBKDF2](#) is similar and is also supported. You can still use any message digest algorithm that is supported in your JDK; see [this Java page](#) for the available algorithms.

Session Fixation Prevention by default

Session Fixation Prevention is now enabled by default, but can be disabled with


```
grails.plugin.springsecurity.useSessionFixationPrevention = false
```

@Secured annotation

As of Grails 2.0, controller actions can be defined as closures or methods, with methods being preferred. The @Secured annotation no longer supports being defined on controller action closures, so you will need to convert them to real methods.

You can also specify the HTTP method that an annotation is defined for (e.g. when using REST). When doing this you must explicitly name the `value` attribute, e.g.

```
@Secured(value=["hasRole('ROLE_ADMIN')"], httpMethod='POST')
def someMethod() {
    ...
}
```

In addition, you can define a closure in the annotation which will be called during access checking. The closure must return `true` or `false` and has all of the methods and properties that are available when using SpEL expressions, since the closure's delegate is set to a subclass of `WebSecurityExpressionRoot`, and also the Spring `ApplicationContext` as the `ctx` property:

```
@Secured(closure = {
    assert request
    assert ctx
    authentication.name == 'admin1'
})
def someMethod() {
    ...
}
```

Anonymous authentication

In standard Spring Security and older versions of the plugin, there is support for an "anonymous" authentication. This is implemented by a filter that registers a simple `Authentication` in the `SecurityContext` to remove the need for null checks, since there will always be an `Authentication` available. This approach is still problematic though because the `Principal` of the anonymous authentication is a `String`, whereas it is a `UserDetails` instance when there is a non-anonymous authentication.

Since you still have to be careful to differentiate between anonymous and non-anonymous authentications, the plugin now creates an anonymous `Authentication` which will be an instance of `grails.plugin.springsecurity.authentication`.

`GrailsAnonymousAuthenticationToken` with a standard `org.springframework.security.core.userdetails.User` instance as its `Principal`. The authentication will have a single granted role, `ROLE_ANONYMOUS`.

No HQL

Some parts of the code used HQL queries, for example in the generated `UserRole` class and in `SpringSecurityService.findRequestmapsByRole`. These have been replaced by "where" queries to make data access more portable across GORM implementations.

Changes in generated classes

The `enabled` property in the generated `User` class now defaults to `true`. This will make creating instances a bit more DRY:

```
def u = new User(username: 'me', password: 'itsasecret').save()
```

If you prefer the old approach, change your generated class.

Also, the plugin includes the `grails.plugin.springsecurity.LoginController.groovy` and `grails.plugin.springsecurity.LogoutController.groovy` controllers, and `grails-app/views/auth.gsp` and `grails-app/views/denied.gsp` GSPs. If you had no need previously to change these you can delete your files and the plugins' files will be used instead. If you do want to change them, copy each as needed to your application and make the required changes, and yours will be used instead.

One small change is that there is no longer a default value for the domain class name properties (`userLookup.userDomainClassName`, `authority.className`, `requestMap.className`, `rememberMe.persistentToken.domainClassName`). This was of little use and tended to cause confusing error messages when there was a misconfiguration.

SecurityContextHolder strategy

You can now define the `SecurityContextHolder` strategy. By default it is stored in a `ThreadLocal`, but you can also configure it to use an `InheritableThreadLocal` to maintain the context in new threads, or a custom class that implements the `org.springframework.security.core.context.SecurityContextHolderStrategy` interface. To change the strategy, set the `grails.plugin.springsecurity.sch.strategyName` config property to `"MODE_THREADLOCAL"` (the default) to use a `ThreadLocal`, `"MODE_INHERITABLETHREADLOCAL"` to use an `InheritableThreadLocal`, or the name of a class that implements `SecurityContextHolderStrategy`.

Debug filter

You can enable a "debug" filter based on the `org.springframework.security.config.debug.DebugFilter` class. It will log security information at the "info" level and can help when debugging configuration issues. This should only be enabled in development mode so consider adding the property that enables it inside an `environments` block in `Config.groovy`

```
environments {
  development {
    grails.logging.jul.usebridge = true
    grails.plugin.springsecurity.debug.useFilter = true
  }
  production {
    grails.logging.jul.usebridge = false
  }
}
```

Also add the implementation class name in your Log4j configuration:

```
info 'grails.plugin.springsecurity.web.filter.DebugFilter'
```

Storing usernames in the session

In Spring Security 3.0 and earlier, the username was stored in the HTTP session under the key "SPRING_SECURITY_LAST_USERNAME". This is no longer done, but the plugin will use the old behavior if the `grails.plugin.springsecurity.apf.storeLastUsername` setting is set to `true` (the default is `false`). Further, the name is no longer escaped before storing, it is stored exactly as entered by the user, so you must escape it when redisplaying to avoid XSS attacks.

@Authorities annotation

You can use the new `@Authorities` annotation to make your annotations more DRY. See [this blog post](#) for a description about the motivation and implementation details. Note that the package for the annotation in the plugin is `grails.plugin.springsecurity.annotation`, not `grails.plugins.springsecurity.annotation` as described in the blog post.

Miscellaneous changes

AuthenticationDetailsSource

Previously you could configure the details class that was constructed by the `authenticationDetailsSource` bean by setting the `authenticationDetails.authClass` property. In Spring Security 3.2 this isn't possible because `WebAuthenticationDetailsSource` always returns a `WebAuthenticationDetails`. But you can still customize the details class by creating a class that implements the [AuthenticationDetailsSource](#) interface, e.g.:

```
package com.mycompany;

import javax.servlet.http.HttpServletRequest;

import org.springframework.security.authentication.AuthenticationDetailsSource;

public class MyAuthenticationDetailsSource implements
AuthenticationDetailsSource<HttpServletRequest, MyWebAuthenticationDetails> {

    public MyWebAuthenticationDetails buildDetails(HttpServletRequest context) {
        // build a MyWebAuthenticationDetails
    }
}
```

and registering that as the authenticationDetailsSource bean in resources.groovy

```
import com.mycompany.MyAuthenticationDetailsSource

beans = {
    authenticationDetailsSource(MyAuthenticationDetailsSource) {
        // any required properties
    }
}
```

3 Required and Optional Domain Classes

By default the plugin uses regular Grails domain classes to access its required data. It's easy to create your own user lookup code though, which can access the database or any other source to retrieve user and authority data. See [Custom UserDetailsService](#) for how to implement this.

To use the standard user lookup you'll need at a minimum 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) you need a 'requestmap' domain class. If you use the recommended approach for mapping the many-to-many relationship between 'person' and 'authority,' you 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 unlimited fields, methods, and so on, as long as the core security-related functionality remains.

3.1 Person Class

Spring Security uses an [Authentication](#) object to determine whether the current user has the right to perform a secured action, such as accessing a URL, manipulating a secured domain object, accessing a secured method, and so on. This object is created during login. Typically overlap occurs between the need for authentication data and the need to represent a user in the application in ways that are unrelated 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()`.

By default the plugin uses a Grails 'person' domain class to manage this data. `username`, `enabled`, `password` are the default names of the core required properties. You can easily [plug in your own implementation](#), and rename the class, package, and fields. In addition, you should define an `authorities` property to retrieve roles; this can be a public field or a `getAuthorities()` method, and it can be defined through a traditional GORM many-to-many or a custom mapping.

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 {

    transient springSecurityService

    String username
    String password
    boolean enabled = true
    boolean accountExpired
    boolean accountLocked
    boolean passwordExpired

    static transients = ['springSecurityService']

    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
    }

    def beforeInsert() {
        encodePassword()
    }

    def beforeUpdate() {
        if (isDirty('password')) {
            encodePassword()
        }
    }

    protected void encodePassword() {
        password = springSecurityService.encodePassword(password)
    }
}

```

Optionally, add other properties such as email, firstName, lastName, and convenience methods, and so on:

```

package com.mycompany.myapp

class User {

  transient springSecurityService

  String username
  String password
  boolean enabled = true
  String email
  String firstName
  String lastName
  boolean accountExpired
  boolean accountLocked
  boolean passwordExpired

  static transients = ['springSecurityService']

  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
  }

  def someMethod {
    ...
  }

  def beforeInsert() {
    encodePassword()
  }

  def beforeUpdate() {
    if (isDirty('password')) {
      encodePassword()
    }
  }

  protected void encodePassword() {
    password = springSecurityService.encodePassword(password)
  }
}

```

The `getAuthorities()` method is analagous to defining `static 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
userLookup.userDomainClassName	none	User class name
userLookup.usernamePropertyName	'username'	User class username field
userLookup.passwordPropertyName	'password'	User class password field
userLookup.authoritiesPropertyName	'authorities'	User class role collection field
userLookup.enabledPropertyName	'enabled'	User class enabled field
userLookup.accountExpiredPropertyName	'accountExpired'	User class account expired field
userLookup.accountLockedPropertyName	'accountLocked'	User class account locked field
userLookup.passwordExpiredPropertyName	'passwordExpired'	User class password expired field
userLookup.authorityJoinClassName	'PersonAuthority'	User/Role many-many join class name

3.2 Authority Class

The Spring Security plugin also requires an 'authority' class to represent a user's role(s) in the application. In general this class restricts 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 access only non-restricted resources but can still authenticate is a bit unusual. Spring Security usually functions fine if a user has no granted authorities, but fails in a few places that assume one or more. So if a user authenticates successfully but has no granted roles, the plugin grants the user a 'virtual' role, `ROLE_NO_ROLES`. Thus the user satisfies Spring Security's requirements but cannot access secure resources, as you would not 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 another existing domain class, it simply 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	none	Role class name
authority.nameField	'authority'	Role class role name field

Note that role names must start with "ROLE_". This is configurable in Spring Security, but not in the plugin. It would be possible to allow different prefixes, but it's important that the prefix not be blank as the prefix is used to differentiate between role names and tokens such as `IS_AUTHENTICATED_FULLY`/`IS_AUTHENTICATED_ANONYMOUSLY`/etc., and SpEL expressions.

The role names should be primarily an internal implementation detail; if you want to display friendlier names in a UI, it's simple to remove the prefix first.

3.3 PersonAuthority Class

The typical approach to mapping the relationship between 'person' and 'authority' is a many-to-many. Users have multiple roles, and roles are shared by multiple users. This approach can be problematic in Grails, because a popular role, for example, `ROLE_USER`, will be granted to many users in your application. 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 existing users who have that role because the collection is a `Set`. So even though no uniqueness concerns may exist, Hibernate loads 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 class is 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 {

    private static final long serialVersionUID = 1

    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) {
        UserRole.where {
            user == User.load(userId) &&
            role == Role.load(roleId)
        }.get()
    }

    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 u, Role r, boolean flush = false) {
        int rowCount = UserRole.where {
            user == User.load(u.id) &&
            role == Role.load(r.id)
        }.deleteAll()

        rowCount > 0
    }

    static void removeAll(User u) {
        UserRole.where {
            user == User.load(u.id)
        }.deleteAll()
    }

    static void removeAll(Role r) {
        UserRole.where {
            role == Role.load(r.id)
        }.deleteAll()
    }

    static mapping = {
        id composite: ['role', 'user']
        version false
    }
}

```

The helper methods make it easy to grant or revoke roles. Assuming you have already loaded a user and a role, you grant the role to the user as follows:

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

Or by using the 3-parameter version to trigger a flush:

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

Revoking a role is similar:

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

Or:

```
User user = ...
Role role = ...
UserRole.remove 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

3.4 Requestmap Class

Optionally, use this class to store request mapping entries in the database instead of defining them with annotations or in `Config.groovy`. This option makes the class configurable at runtime; you can add, remove and edit rules without restarting your application.

Property	Default Value	Meaning
requestMap.className	none	requestmap class name
requestMap.urlField	'url'	URL pattern field name
requestMap.configAttributeField	'configAttribute'	authority pattern field name
requestMap.httpMethodField	'httpMethod'	HTTP method field name (optional, does not have to exist in the class if you don't require URL/method security)

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

```
package com.testapp

import org.springframework.http.HttpMethod

class Requestmap {

    String url
    String configAttribute
    HttpMethod httpMethod

    static mapping = {
        cache true
    }

    static constraints = {
        url blank: false, unique: 'httpMethod'
        configAttribute blank: false
        httpMethod nullable: true
    }
}
```

To use Requestmap entries to guard URLs, see [Requestmap Instances Stored in the Database](#).

4 Configuring Request Mappings to Secure URLs

You can choose among the following approaches to configuring request mappings for secure application URLs. The goal is to map URL patterns to the roles required to access those URLs.

- [@Secured annotations \(default approach\)](#)
- [A simple Map in Config.groovy](#)
- [Requestmap domain class instances stored in the database](#)

You can only use one method at a time. You configure it with the `securityConfigType` attribute; the value has to be an `SecurityConfigType` enum value or the name of the enum as a String.

Pessimistic Lockdown

Many applications are mostly public, with some pages only accessible to authenticated users with various roles. In this case, it might make sense to leave URLs open by default and restrict access on a case-by-case basis. However, if your application is primarily secure, you can use a pessimistic lockdown approach to deny access to all URLs that do not have an applicable URL-Role request mapping. But the pessimistic approach is safer; if you forget to restrict access to a URL using the optimistic approach, it might take a while to discover that unauthorized users can access the URL, but if you forget when using the pessimistic approach, no user can access it and the error should be quickly discovered.

The pessimistic approach is the default, and there are two configuration options that apply. If `rejectIfNoRule` is `true` (the default) then any URL that has no request mappings (an annotation, entry in `controllerAnnotations.staticRules` or `interceptUrlMap`, or a `Requestmap` instance) will be denied to all users. The other option is `fii.rejectPublicInvocations` and if it is `true` (the default) then un-mapped URLs will trigger an `IllegalArgumentException` and will show the error page. This is uglier, but more useful because it's very clear that there is a misconfiguration. When `fii.rejectPublicInvocations` is `false` but `rejectIfNoRule` is `true` you just see the "Sorry, you're not authorized to view this page." error 403 message.

Note that the two settings are mutually exclusive. If `rejectIfNoRule` is `true` then `fii.rejectPublicInvocations` is ignored because the request will transition to the login page or the error 403 page. If you want the more obvious error page, set `fii.rejectPublicInvocations` to `true` and `rejectIfNoRule` to `false` to allow that check to occur.

To reject un-mapped URLs with a 403 error code, use these settings (or none since `rejectIfNoRule` defaults to `true`)

```
grails.plugin.springsecurity.rejectIfNoRule = true
grails.plugin.springsecurity.fii.rejectPublicInvocations = false
```

and to reject with the error 500 page, use these (optionally omit `rejectPublicInvocations` since it defaults to `true`):

```
grails.plugin.springsecurity.rejectIfNoRule = false
grails.plugin.springsecurity.fii.rejectPublicInvocations = true
```

Note that if you set `rejectIfNoRule` or `rejectPublicInvocations` to `true` you'll need to configure the `staticRules` map to include URLs that can't otherwise be guarded:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    '/': ['permitAll'],
    '/index': ['permitAll'],
    '/index.gsp': ['permitAll'],
    '/**/js/**': ['permitAll'],
    '/**/css/**': ['permitAll'],
    '/**/images/**': ['permitAll'],
    '/**/favicon.ico': ['permitAll']
]
```

This is needed when using annotations; if you use the `grails.plugin.springsecurity.interceptUrlMap` map in `Config.groovy` you'll need to add these URLs too, and likewise when using `Requestmap` instances.

In addition, when you enable the switch-user feature, you'll have to specify access rules for the associated URLs, e.g.

```
'/j_spring_security_switch_user': ['ROLE_ADMIN'],
'/j_spring_security_exit_user': ['permitAll']
```

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, for example, `/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 an 'anonymous' Authentication with no roles so that every user has an authentication. The token accepts any authentication, even anonymous.
- `IS_AUTHENTICATED_REMEMBERED`
 - requires the user to be authenticated through a remember-me cookie or an explicit login.
- `IS_AUTHENTICATED_FULLY`
 - requires the user to be fully authenticated with an explicit login.

With `IS_AUTHENTICATED_FULLY` you can implement a security scheme whereby users can check a remember-me checkbox during login and be auto-authenticated each time they return to your site, but must still log in with a password for some parts of the site. For example, allow regular browsing and adding items to a shopping cart with only a cookie, but require an explicit login to check out or view purchase history.

For more information on `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY`, see the Javadoc for [AuthenticatedVoter](#)



The plugin isn't compatible with Grails `<g:actionSubmit>` tags. These are used in the autogenerated GSPs that are created for you, and they enable having multiple submit buttons, each with its own action, inside a single form. The problem from the security perspective is that the form posts to the default action of the controller, and Grails figures out the handler action to use based on the `action` attribute of the `actionSubmit` tag. So for example you can guard the `/person/delete` with a restrictive role, but given this typical edit form:

```
<g:form>
  ...
  <g:actionSubmit class="save" action="update"
                 value='Update' />
  <g:actionSubmit class="delete" action="delete"
                 value='Delete' />
</g:form>
```

both actions will be allowed if the user has permission to access the `/person/index` url, which would often be the case.

The workaround is to create separate forms without using `actionSubmit` and explicitly set the `action` on the `<g:form>` tags, which will result in form submissions to the expected urls and properly guarded urls.

Comparing the Approaches

Each approach has its advantages and disadvantages. Annotations and the `Config.groovy` Map are less flexible because they are configured once in the code and you can update them only by restarting the application (in prod mode anyway). In practice this limitation is minor, because security mappings for most applications are unlikely to change at runtime.

On the other hand, storing `Requestmap` entries enables runtime-configurability. This approach gives you a core set of rules populated at application startup that you can edit, add to, and delete as needed. However, 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.

URLs must be mapped in lowercase if you use the `Requestmap` or `grails-app/conf/Config.groovy` map approaches. For example, if you have a `FooBarController`, its urls will be of the form `/fooBar/list`, `/fooBar/create`, and so on, but these must be mapped as `/foobar/`, `/foobar/list`, `/foobar/create`. This mapping is handled automatically for you if you use annotations.

4.1 Defining Secured Annotations

You can use an `@Secured` annotation (either the standard `org.springframework.security.access.annotation.Secured` or the plugin's `grails.plugin.springsecurity.annotation.Secured` which also works on controller closure actions) in your controllers to configure which roles are required for which actions. To use annotations, specify `securityConfigType="Annotation"`, or leave it unspecified because it's the default:

```
grails.plugin.springsecurity.securityConfigType = "Annotation"
```

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.plugin.springsecurity.annotation.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 must 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.

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.plugin.springsecurity.annotation.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 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, for example:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
  '/js/admin/**': ['ROLE_ADMIN'],
  '/someplugin/**': ['ROLE_ADMIN']
]
```

This example maps all URLs associated with `SomePluginController`, which has URLs of the form `/somePlugin/...`, to `ROLE_ADMIN`; annotations are not an option here because you would not edit plugin code for a change like this.



When mapping URLs for controllers that are mapped in `UrlMappings.groovy`, you need to secure the un-url-mapped URLs. For example if you have a `FooBarController` that you map to `/foo/bar/$action`, you must register that in `controllerAnnotations.staticRules` as `/foobar/**`. This is different than the mapping you would use for the other two approaches and is necessary because `controllerAnnotations.staticRules` entries are treated as if they were annotations on the corresponding controller.

4.2 Simple Map in Config.groovy

To use the `Config.groovy` Map to secure URLs, first specify `securityConfigType="InterceptUrlMap"`:

```
grails.plugin.springsecurity.securityConfigType = "InterceptUrlMap"
```

Define a Map in `Config.groovy`:

```
grails.plugin.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'],
```

4.3 Requestmap Instances Stored in the Database

With this approach you use the `Requestmap` domain class to store mapping entries in the database. `Requestmap` has a `url` property that 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`.

To use `Requestmap` entries, specify `securityConfigType="Requestmap"`:

```
grails.plugin.springsecurity.securityConfigType = "Requestmap"
```

You create `Requestmap` entries as you create entries in any Grails domain class:

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

The `configAttribute` value can have a single value or have multiple comma-delimited values. In this example only users with `ROLE_ADMIN` or `ROLE_SUPERVISOR` can access `/admin/user/**` urls, and only users with `ROLE_SWITCH_USER` can access the switch-user url (`/j_spring_security_switch_user`) and in addition must be authenticated fully, i.e. not using a remember-me cookie. Note that when specifying multiple roles, the user must have at least one of them, but when combining `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, or `IS_AUTHENTICATED_ANONYMOUSLY` with one or more roles means the user must have one of the roles and satisfy the `IS_AUTHENTICATED` rule.

Unlike the [Config.groovy Map approach](#), you do not need to revise the Requestmap entry order because the plugin calculates the most specific rule that applies to the current request.

Requestmap Cache

Requestmap entries are cached for performance, but caching affects 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: 'requestmap.label', default: 'Requestmap'),
        requestmapInstance.id])}"
    redirect action: 'show', id: requestmapInstance.id
  }
}
```

4.4 Using Expressions to Create Descriptive, Fine-Grained Rules

Spring Security uses the [Spring Expression Language \(SpEL\)](#), which allows you to declare the rules for guarding URLs more descriptively than does the traditional approach, and also allows much more fine-grained rules. Where you traditionally would specify a list of role names and/or special tokens (for example, `IS_AUTHENTICATED_FULLY`), with [Spring Security's expression support](#), you can instead use the embedded scripting language to define simple or complex access rules.

You can use expressions with any of the previously described approaches to securing application URLs. For example, consider this annotated controller:

```

package com.yourcompany.yourapp

import grails.plugin.springsecurity.annotation.Secured

class SecureController {

    @Secured(["hasRole('ROLE_ADMIN')"])
    def someAction() {
        ...
    }

    @Secured(["authentication.name == 'ralph'"])
    def someOtherAction() {
        ...
    }
}

```

In this example, `someAction` requires `ROLE_ADMIN`, and `someOtherAction` requires that the user be logged in with username 'ralph'.

The corresponding Requestmap URLs would be

```

new Requestmap(url: "/secure/someAction",
               configAttribute: "hasRole('ROLE_ADMIN)').save()

new Requestmap(url: "/secure/someOtherAction",
               configAttribute: "authentication.name == 'ralph']").save()

```

and the corresponding static mappings would be

```

grails.plugin.springsecurity.interceptUrlMap = [
    '/secure/someAction': ["hasRole('ROLE_ADMIN')"],
    '/secure/someOtherAction': ["authentication.name == 'ralph'"]
]

```

The Spring Security docs have a [table listing the standard expressions](#), which is copied here for reference:

Expression	Description
<code>hasRole(role)</code>	Returns true if the current principal has the specified role.
<code>hasAnyRole([role1,role2])</code>	Returns true if the current principal has any of the supplied roles (given as a comma-separated list of strings)
<code>principal</code>	Allows direct access to the principal object representing the current user
<code>authentication</code>	Allows direct access to the current Authentication object obtained from the SecurityContext
<code>permitAll</code>	Always evaluates to true
<code>denyAll</code>	Always evaluates to false
<code>isAnonymous()</code>	Returns true if the current principal is an anonymous user
<code>isRememberMe()</code>	Returns true if the current principal is a remember-me user
<code>isAuthenticated()</code>	Returns true if the user is not anonymous
<code>isFullyAuthenticated()</code>	Returns true if the user is not an anonymous or a remember-me user
<code>request</code>	the HTTP request, allowing expressions such as <code>"isFullyAuthenticated() or request.getMethod().equals('OPTIONS')"</code>

In addition, you can use a web-specific expression `hasIpAddress`. However, you may find it more convenient to separate IP restrictions from role restrictions by using the [IP address filter](#).

To help you migrate traditional configurations to expressions, this table compares various configurations and their corresponding expressions:

Traditional Config	Expression
<code>ROLE_ADMIN</code>	<code>hasRole('ROLE_ADMIN')</code>
<code>ROLE_USER,ROLE_ADMIN</code>	<code>hasAnyRole('ROLE_USER','ROLE_ADMIN')</code>
<code>ROLE_ADMIN,IS_AUTHENTICATED_FULLY</code>	<code>hasRole('ROLE_ADMIN')</code> and <code>isFullyAuthenticated()</code>
<code>IS_AUTHENTICATED_ANONYMOUSLY</code>	<code>permitAll</code>
<code>IS_AUTHENTICATED_REMEMBERED</code>	<code>isAuthenticated()</code> or <code>isRememberMe()</code>
<code>IS_AUTHENTICATED_FULLY</code>	<code>isFullyAuthenticated()</code>

5 Helper Classes

Use the plugin helper classes in your application to avoid dealing with some lower-level details of Spring Security.

5.1 SecurityTagLib

The plugin includes GSP tags to support conditional display based on whether the user is authenticated, and/or has the required role to perform a particular action. These tags are in the `sec` namespace and are implemented in `grails.plugin.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, to show the username property:

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

If you have customized the authentication to add a `fullName` property, you access it as follows:

```
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. (See also [Switch 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.

```

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

```

access

Renders the body if the specified expression evaluates to true or specified URL is allowed.

```

<sec:access expression="hasRole( 'ROLE_USER' )">
You're a user
</sec:access>

```

```

<sec:access url="/admin/user">
<g:link controller='admin' action='user'>Manage Users</g:link>
</sec:access>

```


You can also guard access to links generated from controller and action names or named URL mappings instead of hard-coding the values, for example

```
<sec:access controller='admin' action='user'>
  <g:link controller='admin' action='user'>Manage Users</g:link>
</sec:access>
```

or if you have a named URL mapping you can refer to that:

```
<sec:access mapping='manageUsers'>
  <g:link mapping='manageUsers'>Manage Users</g:link>
</sec:access>
```

For even more control of the generated URL (still avoiding hard-coding) you can use `createLink` to build the URL, for example

```
<sec:access url='${createLink(controller: 'admin', action: 'user', base: "/"
)}'>
  <g:link controller='admin' action='user'>Manage Users</g:link>
</sec:access>
```

Be sure to include the `base: "/"` attribute in this case to avoid appending the context name to the URL.

noAccess

Renders the body if the specified expression evaluates to false or URL isn't allowed.

```
<sec:noAccess expression="hasRole('ROLE_USER')">
  You're not a user
</sec:noAccess>
```

5.2 SpringSecurityService

`grails.plugin.springsecurity.SpringSecurityService` provides security utility functions. It is a regular Grails service, so you use dependency injection to inject it into a controller, service, taglib, and so on:

```
def springSecurityService
```

getCurrentUser()

Retrieves a domain class instance for the currently authenticated user. During authentication a user/person domain class instance is loaded to get the user's password, roles, etc. and the id of the instance is saved. This method uses the id and the domain class to re-load the instance.

Example:

```
class SomeController {
  def springSecurityService
  def someAction() {
    def user = springSecurityService.currentUser
    ...
  }
}
```

isLoggedIn()

Checks whether there is a currently logged-in user.

Example:

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

getAuthentication()

Retrieves the current user's [Authentication](#). If authenticated, this will typically be a [UsernamePasswordAuthenticationToken](#).

If not authenticated and the [AnonymousAuthenticationFilter](#) is active (true by default) then the anonymous user's authentication will be returned. This will be an instance of `grails.plugin.springsecurity.authentication`.

`GrailsAnonymousAuthenticationToken` with a standard `org.springframework.security.core.userdetails.User` instance as its Principal. The authentication will have a single granted role, `ROLE_ANONYMOUS`.

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. If authenticated, the principal will be a `grails.plugin.springsecurity.userdetails.GrailsUser`, unless you have created a custom `UserDetailsService`, in which case it will be whatever implementation of [UserDetails](#) you use there.

If not authenticated and the [AnonymousAuthenticationFilter](#) is active (true by default) then a standard `org.springframework.security.core.userdetails.User` is used.

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()

Hashes a password with the configured hashing scheme. By default the plugin uses bcrypt, but you can configure the scheme with the `grails.plugin.springsecurity.password.algorithm` attribute in `Config.groovy`. The supported values are 'bcrypt' to use bcrypt, 'pbkdf2' to use [PBKDF2](#), or any message digest algorithm that is supported in your JDK; see [this Java page](#) for the available algorithms.



You are **strongly** discouraged from using MD5 or SHA-1 algorithms because of their 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 [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
  }
}

```



If you are hashing the password in the User domain class (using `beforeInsert` and `encodePassword`) then don't call `springSecurityService.encodePassword()` in your controller since you'll double-hash the password and users won't be able to log in. It's best to encapsulate the password handling logic in the domain class.

updateRole()

Updates a role and, if you use Requestmap instances to secure URLs, updates the role name in all affected Requestmap definitions 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 use Requestmap instances to secure URLs, removes the role from all affected Requestmap definitions. If a Requestmap's config attribute is only the role name (for example, `"/foo/bar/**=ROLE_FOO"`), it is 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()

Flushes the Requestmaps cache and triggers a complete reload. If you use Requestmap instances to secure URLs, the plugin loads and caches all Requestmap instances as a performance optimization. This action saves database activity because the requestmaps are checked for each request. Do not allow the cache to become stale. When you create, edit or delete a Requestmap, flush the cache. Both `updateRole()` and `deleteRole()` call `clearCachedRequestmaps()` for you. Call this method when you create a new Requestmap or do other Requestmap work that affects 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. You typically use this method 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 (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
  }
}

```

5.3 SpringSecurityUtils

`grails.plugin.springsecurity.SpringSecurityUtils` is a utility class with static methods that you can call directly without using dependency injection. It is 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. It is empty (but never null) if the user is not logged in.

parseAuthoritiesString()

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

ifAllGranted()

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

ifNotGranted()

Checks whether the current user has none of the specified roles (a comma-delimited String of role names). Primarily used by `SecurityTagLib.ifNotGranted`.

ifAnyGranted()

Checks whether the current user has any of the specified roles (a comma-delimited String of role names). 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 whether the request was triggered by an Ajax call. The standard way is to determine whether `X-Requested-With` request header is set and has the value `XMLHttpRequest`. In addition, you can configure the name of the header with the `grails.plugin.springsecurity.ajaxHeader` configuration attribute, but this is not recommended because all major JavaScript toolkits use the standard name. Further, you can register a closure in `Config.groovy` with the name `ajaxCheckClosure` that will be used to check if a request is an Ajax request. It is passed the request as its single argument, e.g.

```
grails.plugin.springsecurity.ajaxCheckClosure = { request ->
    // return true or false
}
```

You can also 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 whether the current user switched from another user.

getSwitchedUserOriginalUsername()

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

doWithAuth()

Executes a Closure with the current authentication. The one-parameter version which takes just a Closure assumes that there's an authentication in the HTTP Session and that the Closure is running in a separate thread from the web request, so the `SecurityContext` and `Authentication` aren't available to the standard `ThreadLocal`. This is primarily of use when you explicitly launch a new thread from a controller action or service called in request scope, not from a Quartz job which isn't associated with an authentication in any thread.

The two-parameter version takes a username and a Closure to authenticate as. This will authenticate as the specified user and execute the closure with that authentication. It restores the authentication to the one that was active if it exists, or clears the context otherwise. This is similar to `run-as` and `switch-user` but is only local to the Closure.

6 Events

Spring Security fires application events after various security-related actions such as successful login, unsuccessful login, and so on. Spring Security uses two main event classes, [AbstractAuthenticationEvent](#) and [AbstractAuthorizationEvent](#).

6.1 Event Notification

You can set up event notifications in two ways. The sections that follow describe each approach in more detail.

- Register an event listener, ignoring events that do not interest you. Spring allows only partial event subscription; you use generics to register the class of events that interest you, and you are notified of that class and all subclasses.
- Register one or more callback closures in `grails-app/conf/Config.groovy` that take advantage of the plugin's `grails.plugin.springsecurity.SecurityEventListener`. The listener does the filtering for you.

AuthenticationEventPublisher

Spring Security publishes events using an [AuthenticationEventPublisher](#) which in turn fire events using the [ApplicationEventPublisher](#). By default no events are fired since the `AuthenticationEventPublisher` instance registered is a `grails.plugin.springsecurity.authentication.NullAuthenticationEventPublisher`. But you can enable event publishing by setting `grails.plugin.springsecurity.useSecurityEventListener = true` in `grails-app/conf/Config.groovy`.

You can use the `useSecurityEventListener` setting to temporarily disable and enable the callbacks, or enable them per-environment.

UsernameNotFoundException

Most authentication exceptions trigger an event with a similar name as described in this table:

Exception	Event
<code>AccountExpiredException</code>	<code>AuthenticationFailureExpiredEvent</code>
<code>AuthenticationServiceException</code>	<code>AuthenticationFailureServiceExceptionEvent</code>
<code>LockedException</code>	<code>AuthenticationFailureLockedEvent</code>
<code>CredentialsExpiredException</code>	<code>AuthenticationFailureCredentialsExpiredEvent</code>
<code>DisabledException</code>	<code>AuthenticationFailureDisabledEvent</code>
<code>BadCredentialsException</code>	<code>AuthenticationFailureBadCredentialsEvent</code>
<code>UsernameNotFoundException</code>	<code>AuthenticationFailureBadCredentialsEvent</code>
<code>ProviderNotFoundException</code>	<code>AuthenticationFailureProviderNotFoundEvent</code>

This holds for all exceptions except `UsernameNotFoundException` which triggers an `AuthenticationFailureBadCredentialsEvent` just like a `BadCredentialsException`. This is a good idea since it doesn't expose extra information - there's no differentiation between a bad password and a missing user. In addition, by default a missing user will trigger a `BadCredentialsException` for the same reasons. You can configure Spring Security to re-throw the original `UsernameNotFoundException` instead of converting it to a `BadCredentialsException` by setting `grails.plugin.springsecurity.dao.hideUserNotFoundExceptions = false` in `grails-app/conf/Config.groovy`.

Fortunately all subclasses of [AbstractAuthenticationFailureEvent](#) have a `getException()` method that gives you access to the exception that triggered the event, so you can use that to differentiate between a bad password and a missing user (if `hideUserNotFoundExceptions=false`).

6.2 Registering an Event Listener

Enable events with `grails.plugin.springsecurity.useSecurityEventListener = true` and create one or more Groovy or Java classes, for example:

```
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
    }
}
```

Register the class in `grails-app/conf/spring/resources.groovy`:

```
import com.foo.bar.MySecurityEventListener

beans = {
    mySecurityEventListener(MySecurityEventListener)
}
```

6.3 Registering Callback Closures

Alternatively, `enable events` with `grails.plugin.springsecurity.useSecurityEventListener = true` and register one or more callback closure(s) in `grails-app/conf/Config.groovy` and let `SecurityEventListener` do the filtering.

Implement the event handlers that you need, for example:

```

grails.plugin.springsecurity.useSecurityEventListener = true

grails.plugin.springsecurity. onInteractiveAuthenticationSuccessEvent = { e,
appCtx ->
    // handle InteractiveAuthenticationSuccessEvent
}

grails.plugin.springsecurity. onAbstractAuthenticationFailureEvent = { e,
appCtx ->
    // handle AbstractAuthenticationFailureEvent
}

grails.plugin.springsecurity. onAuthenticationSuccessEvent = { e, appCtx ->
    // handle AuthenticationSuccessEvent
}

grails.plugin.springsecurity. onAuthenticationSwitchUserEvent = { e, appCtx ->
    // handle AuthenticationSwitchUserEvent
}

grails.plugin.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: When a user authenticates, Spring Security initially fires an `AuthenticationSuccessEvent`. This event fires before the `Authentication` is registered in the `SecurityContextHolder`, which means that the `springSecurityService` methods that access the logged-in user will not work. Later in the processing a second event is fired, an `InteractiveAuthenticationSuccessEvent`, and when this happens the `SecurityContextHolder` will have the `Authentication`. Depending on your needs, you can implement a callback for either or both events.

7 User, Authority (Role), and Requestmap Properties

Properties you are most likely to be override are the `User` and `Authority` (and `Requestmap` if you use 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.

8 Authentication

The Spring Security plugin supports several approaches to authentication.

The default approach stores users and roles in your database, and uses an HTML login form which prompts the user for a username and password. The plugin also supports other approaches as described in the sections below, as well as add-on plugins that provide external authentication providers such as [OpenID](#), [LDAP](#), and single sign-on using [CAS](#)

8.1 Basic and Digest Authentication

To use [HTTP Basic Authentication](#) in your application, set the `useBasicAuth` attribute to `true`. Also change the `basic.realmName` default value to one that suits your application, for example:

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

Property	Default	Description
<code>useBasicAuth</code>	<code>false</code>	Whether to use basic authentication.
<code>basic.realmName</code>	<code>'Grails Realm'</code>	Realm name displayed in the browser authentication popup.
<code>basic.credentialsCharset</code>	<code>'UTF-8'</code>	The character set used to decode Base64-encoded data

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

If you don't want all of your URLs guarded by Basic Auth, you can partition the URL patterns and apply Basic Auth to some, but regular form login to others. For example, if you have a web service that uses Basic Auth for `/webservice/**` URLs, you would configure that using the `chainMap` config attribute:

```
grails.plugin.springsecurity.filterChain.chainMap = [
  '/webservice/**': 'JOINED_FILTERS,-exceptionTranslationFilter',
  '/**':
  'JOINED_FILTERS,-basicAuthenticationFilter,-basicExceptionTranslationFilter'
]
```

In this example we're using the `JOINED_FILTERS` keyword instead of explicitly listing the filter names. Specifying `JOINED_FILTERS` means to use all of the filters that were configured using the various config options. In each case we also specify that we want to exclude one or more filters by prefixing their names with `-`.

For the `/webservice/**` URLs, we want all filters except for the standard `ExceptionTranslationFilter` since we want to use just the one configured for Basic Auth. And for the `/**` URLs (everything else) we want everything except for the Basic Auth filter and its configured `ExceptionTranslationFilter`.

[Digest Authentication](#) is similar to Basic but is more secure because it does not send your password in obfuscated cleartext. Digest resembles Basic in practice - you get the same browser popup dialog when you authenticate. But because the credential transfer is genuinely hashed (instead of just Base64-encoded as with Basic authentication) you do not need SSL to guard your logins.

Property	Default Value	Meaning
useDigestAuth	false	Whether to use Digest authentication.
digest.realmName	'Grails Realm'	Realm name displayed in the browser popup
digest.key	'changeme'	Key used to build the nonce for authentication; it should be changed but that's not required.
digest.nonceValiditySeconds	300	How long a nonce stays valid.
digest.passwordAlreadyEncoded	false	Whether you are managing the password hashing yourself.
digest.createAuthenticatedToken	false	If true, creates an authenticated UsernamePasswordAuthenticationToken to avoid loading the user from the database twice. However, this process skips the isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired(), isEnabled() checks, so it is not advised.
digest.useCleartextPasswords	false	If true, a cleartext password encoder is used (not recommended). If false, passwords hashed by DigestAuthPasswordEncoder are stored in the database.

Digest authentication has a problem in that by default you store cleartext passwords in your database. This is because the browser hashes your password along with the username and Realm name, and this is compared to the password hashed using the same algorithm during authentication. The browser does not know about your MessageDigest algorithm or salt source, so to hash them the same way you need to load a cleartext password from the database.

The plugin does provide an alternative, although it has no configuration options (in particular the digest algorithm cannot be changed). If `digest.useCleartextPasswords` is false (the default), then the `passwordEncoder` bean is replaced with an instance of `grails.plugin.springsecurity.authentication.encoding`.

`DigestAuthPasswordEncoder`. This encoder uses the same approach as the browser, that is, it combines your password along with your username and Realm name essentially as a salt, and hashes 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 probably also want to change the realm name:

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

Digest authentication cannot be applied to a subset of URLs like Basic authentication can. This is due to the password encoding issues. So you cannot use the `chainMap` attribute here - all URLs will be guarded.



Note that since the Digest auth password encoder is different from the typical encoders you must to pass the username as the "salt" value. The generated User class uses `springSecurityService` which assumes you're not using a salt value. If you use the generated code in the User class to encode your password, change the dependency injection for `springSecurityService` with one for the `passwordEncoder` bean instead:

```
transient passwordEncoder
```

and change the code in `encodePassword()` from

```
password = springSecurityService.encodePassword(password)
```

to

```
password = passwordEncoder.encodePassword(password, username)
```

8.2 Certificate (X509) Login Authentication

Another authentication mechanism supported by Spring Security is certificate-based, or "mutual authentication". It requires HTTPS, and you must configure the server to require a client certificate (ordinarily only the server provides a certificate). Your username is extracted from the client certificate if it is valid, and you are "pre-authenticated". As long as a corresponding username exists in the database, your authentication succeeds and you are not asked for a password. Your `Authentication` contains the authorities associated with your username.

The table describes available configuration options.

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=(.*?)(?:, \$)'	Regular expression (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 a valid Authentication already exists.
x509.invalidateSessionOnPrincipalChange	true	Whether to invalidate the session if the principal changed (based on a checkForPrincipalChanges check).
x509.subjectDnClosure	none	If set, the plugin's ClosureX509PrincipalExtractor class is used to extract information from the X.509 certificate using the specified closure
x509.throwExceptionWhenTokenRejected	false	If true thrown a BadCredentialsException

The details of configuring your server for SSL and configuring browser certificates are beyond the scope of this document. If you use Tomcat, see its [SSL documentation](#). To get a test environment working, see the instructions in [this discussion at Stack Overflow](#).

8.3 Remember-Me Cookie

Spring Security supports creating a remember-me cookie so that users are not required to log in with a username and password for each session. This is optional and is usually implemented as a checkbox on the login form; the default `auth.gsp` supplied by the plugin has this feature.

Property	Default Value	Meaning
<code>rememberMe.cookieName</code>	<code>'grails_remember_me'</code>	remember-me cookie name; should be unique per application.
<code>rememberMe.alwaysRemember</code>	<code>false</code>	If <code>true</code> , create a remember-me cookie even if no checkbox is on the form.
<code>rememberMe.tokenValiditySeconds</code>	1209600 (14 days)	Max age of the cookie in seconds.
<code>rememberMe.parameter</code>	<code>'_spring_security_remember_me'</code>	Login form remember-me checkbox name.
<code>rememberMe.key</code>	<code>'grailsRocks'</code>	Value used to encode cookies; should be unique per application.
<code>rememberMe.useSecureCookie</code>	<code>none</code>	Whether to use a secure cookie or not; if <code>true</code> a secure cookie is created, if <code>false</code> a non-secure cookie is created, and if not set, a secure cookie is created if the request used HTTPS
<code>rememberMe.createSessionOnSuccess</code>	<code>true</code>	Whether to create a session of one doesn't exist to ensure that the Authentication is stored for future requests
<code>rememberMe.persistent</code>	<code>false</code>	If <code>true</code> , stores persistent login information in the database.
<code>rememberMe.persistentToken.domainClassName</code>	<code>none</code>	Domain class used to manage persistent logins.
<code>rememberMe.persistentToken.seriesLength</code>	16	Number of characters in the cookie's <code>series</code> attribute.
<code>rememberMe.persistentToken.tokenLength</code>	16	Number of characters in the cookie's <code>token</code> attribute.
<code>atr.rememberMeClass</code>	RememberMeAuthenticationToken	remember-me authentication class.

You are most likely to change these attributes:

- `rememberMe.cookieName`. Purely aesthetic as most users will not look at their cookies, but you probably want the display name to be application-specific rather than `"grails_remember_me"`.
- `rememberMe.key`. Part of a salt when the cookie is encrypted. Changing the default makes it harder to execute brute-force attacks.
- `rememberMe.tokenValiditySeconds`. Default is two weeks; set it to what makes sense for your application.

Persistent Logins

The remember-me cookie is very secure, but for an even stronger solution you can use persistent logins that store the username in the database. See the [Spring Security docs](#) for a description of the implementation.

Persistent login is also useful for authentication schemes like OpenID and Facebook, where you do not manage passwords in your database, but most of the other user information is stored locally. Without a password you cannot use the standard cookie format, so persistent logins enable remember-me cookies in these scenarios.

To use this feature, run the [s2-create-persistent-token](#) script. This will create the domain class, and register its name in `grails-app/conf/Config.groovy`. It will also enable persistent logins by setting `rememberMe.persistent` to `true`.

8.4 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, for example when using [OpenID](#) or a Single Sign-on implementation such as [CAS](#)), 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 to use Ajax and DHTML to present a login form within the current page in a popup. The form submits the authentication request through Ajax and displays success or error messages as appropriate.

The plugin supports Ajax logins, but you 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 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 is 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>

```

Note these changes:

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

Here is the content of the login form template (grails-app/views/includes/_ajaxLogin.gsp). The CSS and Javascript are shown inline, but you should extract them 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 is replaced with a hyperlink.
- Error messages are displayed within the popup `<div>`.
- Because there is 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; you do this by redirecting the user to `/j_spring_security_logout`.

How Does Ajax login 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 did.

To summarize, the typical flow would be

- click the link to display the login form
- enter authentication details and click login
- the form is submitted using an Ajax request
- if the authentication succeeds:
 - a redirect to `/login/ajaxSuccess` occurs (this URL is configurable)
 - the rendered response is JSON and it contains two values, a boolean value `success` with the value `true` and a string value `username` with the authenticated user's login name
 - the client determines that the login was successful and updates the page to indicate the the user is logged in; this is necessary since there's no page redirect like there would be for a non-Ajax login
- if the authentication fails:
 - a redirect to `/login/authfail?ajax=true` occurs (this URL is configurable)
 - the rendered response is JSON and it contains one value, a string value `error` with the displayable error message; this will be different depending on why the login was unsuccessful (bad username or password, account locked, etc.)
 - the client determines that the login was not successful and displays the error message
- note that both a successful and an unsuccessful login will trigger the `onSuccess` Ajax callback; the `onError` callback will only be triggered if there's an exception or network issue

Triggering an Ajax login

So far we've discussed explicit Ajax logins where the user can view some of the site's pages but you've added a link to an in-page login form. An attempt to load a secure page will trigger a redirect to the standard login page. But if you're using Ajax in your pages you should handle the case where the request is secure and requires being logged in. This will also handle session timeouts where the user doesn't have a remember-me cookie; you can pop up a login dialog in the page.

For example consider this Ajax form:

```

<g:form action="ajaxAdd">
  <g:textArea id='postContent' name="content"
              rows="3" cols="50" onkeydown="updateCounter()" />
  <br/>
  <g:submitToRemote value="Post"
                    url="[controller: 'post', action: 'addPostAjax']"
                    update="[success: 'firstPost']"
                    onSuccess="clearPost(e)"
                    onLoading="showSpinner(true)"
                    onComplete="showSpinner(false)"
                    on401="showLogin();" />
  
</g:form>

```

Most of the attributes are typical, but the `on401` attribute is the key to making Ajax logins work. As long as the `LoginController` sends a 401 error code the need to authenticate can be easily handled.

Note that depending on the version of the plugin that you're using, you may need to add the `authAjax` method to your `LoginController`:

```

def authAjax() {
    response.setHeader 'Location',
    SpringSecurityUtils.securityConfig.auth.ajaxLoginFormUrl
    response.sendError HttpServletResponse.SC_UNAUTHORIZED
}

```

and this requires an import for `javax.servlet.http.HttpServletResponse`.

9 Authentication Providers

The plugin registers authentication providers that perform authentication by implementing the [AuthenticationProvider](#) interface.

Property	Default Value	Meaning
providerNames	['daoAuthenticationProvider', 'anonymousAuthenticationProvider', 'rememberMeAuthenticationProvider']	Bean names of authentication providers.

Use `daoAuthenticationProvider` to authenticate using the User and Role database tables, `rememberMeAuthenticationProvider` to log in with a rememberMe 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`. Suppose you have a custom `MyAuthenticationProvider` in `resources.groovy`:

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

You register the provider in `grails-app/conf/Config.groovy` as:

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


10 Custom UserDetailsService

When you authenticate users from a database using [DaoAuthenticationProvider](#) (the default mode in the plugin if you have not enabled OpenID, LDAP, and so on), an implementation of [UserDetailsService](#) is required. This class is responsible for returning a concrete implementation of [UserDetails](#). The plugin provides `grails.plugin.springsecurity.userdetails.GormUserDetailsService` as its `UserDetailsService` implementation and `grails.plugin.springsecurity.userdetails.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. The plugin uses an extension of `UserDetailsService`, `grails.plugin.springsecurity.userdetails.GrailsUserDetailsService`, which adds the method `UserDetails loadUserByUsername(String username, boolean loadRoles)` to support use cases like in LDAP where you often infer all roles from LDAP but might keep application-specific user details in the database.

In the following example, the `UserDetails` and `GrailsUserDetailsService` implementation 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 are less likely to need to reload the user from the database. Most of your common data can be kept along with your security credentials.

This example adds 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 is usually more efficient because it takes advantage of Hibernate's first-level and second-level caches.

There is not much to implement other than your application-specific lookup code:

```
package com.mycompany.myapp

import grails.plugin.springsecurity.userdetails.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.mycompany.myapp

import grails.plugin.springsecurity.SpringSecurityUtils
import grails.plugin.springsecurity.userdetails.GrailsUser
import grails.plugin.springsecurity.userdetails.GrailsUserDetailsService
import org.springframework.security.core.authority.GrantedAuthorityImpl
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.security.core.userdetails.UsernameNotFoundException

class MyUserDetailsService implements GrailsUserDetailsService {

    /**
     * Some Spring Security classes (e.g. RoleHierarchyVoter) expect at least
     * one role, so we give a user with no granted roles this one which gets
     * past that restriction but doesn't grant anything.
     */
    static final List NO_ROLES = [new
GrantedAuthorityImpl(SpringSecurityUtils.NO_ROLE)]

    UserDetails loadUserByUsername(String username, boolean loadRoles)
        throws UsernameNotFoundException {
        return loadUserByUsername(username)
    }

    UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

        User.withTransaction { status ->

            User user = User.findByUsername(username)
            if (!user) throw new UsernameNotFoundException(
                'User not found', username)

            def authorities = user.authorities.collect {
                new GrantedAuthorityImpl(it.authority)
            }

            return new MyUserDetails(user.username, user.password, user.enabled,
                !user.accountExpired, !user.passwordExpired,
                !user.accountLocked, authorities ?: NO_ROLES, user.id,
                user.firstName + " " + user.lastName)
        }
    }
}

```

The lookup code is wrapped in a `withTransaction` block to avoid lazy loading exceptions when accessing the `authorities` collection. There are obviously no database updates here but this is a convenient way to keep the Hibernate Session open to enable accessing the roles.

To use your implementation, register it in `grails-app/conf/spring/resources.groovy` like this:

```

beans = {
    userDetailsService(com.mycompany.myapp.MyUserDetailsService)
}

```

Another option for loading users and roles from the database is to subclass `grails.plugin.springsecurity.userdetails.GormUserDetailsService` - the methods are all protected so you can override as needed.

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 are insufficient.

Flushing the Cached Authentication

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

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

11 Password and Account Protection

The sections that follow discuss approaches to protecting passwords and user accounts.

11.1 Password Hashing

By default the plugin uses the bcrypt algorithm to hash passwords. You can customize this with the `grails.plugin.springsecurity.password.algorithm` attribute as described below. In addition you can increase the security of your passwords by adding a salt, which can be a field of the `UserDetails` instance, a global static value, or any custom value you want.

[bcrypt](#) is a much more secure alternative to the message digest approaches since it supports a customizable work level which when increased takes more computation time to hash the users' passwords, but also dramatically increases the cost of brute force attacks. Given how easy it is to [use GPUs to crack passwords](#), you should definitely consider using bcrypt for new projects and switching to it for existing projects. Note that due to the approach used by bcrypt, you cannot add an additional salt like you can with the message digest algorithms.

Enable bcrypt by using the 'bcrypt' value for the `algorithm` config attribute:

```
grails.plugin.springsecurity.password.algorithm = 'bcrypt'
```

and optionally changing the number of rekeying rounds (which will affect the time it takes to hash passwords), e.g.

```
grails.plugin.springsecurity.password.bcrypt.logrounds = 15
```

Note that the number of rounds must be between 4 and 31.

[PBKDF2](#) is also supported.

The table shows configurable password hashing attributes.

If you want to use a message digest hashing algorithm, see [this Java page](#) for the available algorithms.

Property	Default	Description
<code>password.algorithm</code>	'bcrypt'	passwordEncoder algorithm; 'bcrypt' to use bcrypt, 'pbkdf2' to use PBKDF2 , or any message digest algorithm that is supported in your JDK
<code>password.encodeHashAsBase64</code>	false	If true, Base64-encode the hashed password.
<code>password.bcrypt.logrounds</code>	10	the number of rekeying rounds to use when using bcrypt
<code>password.hash.iterations</code>	10000	the number of iterations which will be executed on the hashed password/salt.

11.2 Salted Passwords

The Spring Security plugin uses hashed passwords and a digest algorithm that you specify. For enhanced protection against dictionary attacks, you should use a salt in addition to digest hashing.

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.reflectionSaltSourceProperty

Set the `dao.reflectionSaltSourceProperty` configuration property:

```
grails.plugin.springsecurity.dao.reflectionSaltSourceProperty = 'username'
```

This property belongs to the `UserDetails` class. By default it is an instance of `grails.plugin.springsecurity.userdetails.GrailsUser`, which extends the standard Spring Security [User class](#) and not your 'person' domain class. This limits the available fields unless you use a [custom UserDetailsService](#).

As long as the username does not change, this approach works well for the salt. If you choose a property that the user can change, the user cannot log in again after changing it unless you re-hash the 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 approach requires a custom `UserDetailsService` because 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.

SystemWideSaltSource and Custom SaltSource

Spring Security supplies a simple `SaltSource` implementation, [SystemWideSaltSource](#), which uses the same salt for each user. It's less robust than using a different value for each user but still better than no salt at all.

An example override of the salt source bean using `SystemWideSaltSource` would look like this:

```
import org.springframework.security.authentication.dao.SystemWideSaltSource
beans = {
    saltSource(SystemWideSaltSource) {
        systemWideSalt = 'the_salt_value'
    }
}
```

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
    }
}
```

Hashing Passwords

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

```
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 (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
  }
}
```



If you are encoding the password in the `User` domain class (using `beforeInsert` and `encodePassword`) then don't call `springSecurityService.encodePassword()` in your controller since you'll double-hash the password and users won't be able to log in. It's best to encapsulate the password handling logic in the domain class. In newer versions of the plugin (version 1.2 and higher) code is auto-generated in the user class so you'll need to adjust that password hashing for your salt approach.

11.3 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` that uses these accessor methods:

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

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 an accessor returns true for `accountExpired`, `accountLocked`, or `passwordExpired` or returns false for `enabled`, 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, for example:

```
grails.plugin.springsecurity.failureHandler.exceptionMappings = [
    'org.springframework.security.authentication.LockedException':
    '/user/accountLocked',
    'org.springframework.security.authentication.DisabledException':
    '/user/accountDisabled',
    'org.springframework.security.authentication.AccountExpiredException':
    '/user/accountExpired',
    'org.springframework.security.authentication.CredentialsExpiredException':
    '/user/passwordExpired'
]
```

Without a mapping for a particular exception, the user is redirected to the standard login fail page (by default `/login/authfail`), which displays 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."

You can customize these messages by setting the corresponding property in `Config.groovy`, for example:

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

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

Here's an example for a password expired workflow. You'd need a simple action to display a password reset form (similar to the login form):

```
def passwordExpired() {  
    [username: session['SPRING_SECURITY_LAST_USERNAME']]  
}
```

and the form would look something like this:

```
<div id='login'>  
    <div class='inner'>  
        <g:if test='${flash.message}'>  
            <div class='login_message'>${flash.message}</div>  
        </g:if>  
        <div class='fheader'>Please update your password..</div>  
        <g:form action='updatePassword' id='passwordResetForm' class='cssform'  
autocomplete='off'>  
            <p>  
                <label for='username'>Username</label>  
                <span class='text_'>${username}</span>  
            </p>  
            <p>  
                <label for='password'>Current Password</label>  
                <g:passwordField name='password' class='text_' />  
            </p>  
            <p>  
                <label for='password'>New Password</label>  
                <g:passwordField name='password_new' class='text_' />  
            </p>  
            <p>  
                <label for='password'>New Password (again)</label>  
                <g:passwordField name='password_new_2' class='text_' />  
            </p>  
            <p>  
                <input type='submit' value='Reset' />  
            </p>  
        </g:form>  
    </div>  
</div>
```

It's important that you not allow the user to specify the username (it's available in the HTTP session) but that you require the current password, otherwise it would be simple to forge a password reset.

The GSP form would submit to an action like this one:


```

def updatePassword() {
    String username = session['SPRING_SECURITY_LAST_USERNAME']
    if (!username) {
        flash.message = 'Sorry, an error has occurred'
        redirect controller: 'login', action: 'auth'
        return
    }

    String password = params.password
    String newPassword = params.password_new
    String newPassword2 = params.password_new_2
    if (!password || !newPassword || !newPassword2 || newPassword !=
newPassword2) {
        flash.message = 'Please enter your current password and a valid new
password'
        render view: 'passwordExpired', model: [username:
session['SPRING_SECURITY_LAST_USERNAME']]
        return
    }

    User user = User.findByUsername(username)
    if (!passwordEncoder.isPasswordValid(user.password, password, null
/*salt*/)) {
        flash.message = 'Current password is incorrect'
        render view: 'passwordExpired', model: [username:
session['SPRING_SECURITY_LAST_USERNAME']]
        return
    }

    if (passwordEncoder.isPasswordValid(user.password, newPassword, null
/*salt*/)) {
        flash.message = 'Please choose a different password from your current
one'
        render view: 'passwordExpired', model: [username:
session['SPRING_SECURITY_LAST_USERNAME']]
        return
    }

    user.password = newPassword
    user.passwordExpired = false
    user.save() // if you have password constraints check them here

    redirect controller: 'login', action: 'auth'
}

```

User Cache

If the `cacheUsers` configuration property is set to `true`, Spring Security caches `UserDetails` instances to save trips to the database. (The default is `false`.) This optimization is minor, because typically only two small queries occur 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 do not remove cached instances, even though a user's account is locked or disabled, logins succeed because the database is bypassed. By removing the cached data, you force a trip to the database to retrieve the latest updates.

Here is a sample Quartz job that demonstrates how to 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
    }
}
}
}

```

12 URL Properties

The table shows configurable URL-related properties.

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 to allow authentication to create an HTTP session.
apf.postOnly	true	Whether to allow only POST login requests.
apf.continueChainBeforeSuccessfulAuthentication	false	whether to continue calling subsequent filters in the filter chain
apf.storeLastUsername	false	Whether to store the login username in the HTTP session
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	Map of exception class name (subclass of AuthenticationException) to which the URL will redirect for that exception type after authentication failure.
failureHandler.useForward	false	Whether to render the error page (true) or redirect (false).
failureHandler.allowSessionCreation	true	Whether to enable session creation to store the authentication failure exception
successHandler.defaultTargetUrl	'/'	Default post-login URL if there is no saved request that triggered the login.
successHandler.alwaysUseDefault	false	If true, always redirects 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 for redirect after successful Ajax login.
auth.loginFormUrl	'/login/auth'	URL of login page.
auth.forceHttps	false	If true, redirects 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 for redirect after logout.
logout.filterProcessesUrl	'/j_spring_security_logout'	Logout URL, intercepted by Spring Security filter.
logout.handlerNames	['rememberMeServices', 'securityContextLogoutHandler']	Logout handler bean names. See Logout Handlers
logout.clearAuthentication	true	If true removes the Authentication from the SecurityContext to prevent issues with concurrent requests
logout.invalidateHttpSession	true	Whether to invalidate the HTTP session when logging out
logout.targetUrlParameter	none	the querystring parameter name for the post-logout URL
logout.alwaysUseDefaultTargetUrl	false	whether to always use the afterLogoutUrl as the post-logout URL
logout.redirectToReferer	false	whether to use the Referer header value as the post-logout URL
logout.postOnly	true	If true only POST requests will be allowed to logout
adh.errorPage	'/login/denied'	Location of the 403 error page (or set to null to send a 403 error and not render a page).
adh.ajaxErrorPage	'/login/ajaxDenied'	Location of the 403 error page for Ajax requests.
adh.useForward	true	If true a forward will be used to render the error page, otherwise a redirect is used
ajaxHeader	'X-Requested-With'	Header name sent by Ajax library, used to detect Ajax.
ajaxCheckClosure	none	An optional closure that can determine if a request is Ajax

redirectStrategy. contextRelative	false	If true, the redirect URL will be the value after the request context path. This results in the loss of protocol information (HTTP or HTTPS), so causes problems if a redirect is being performed to change from HTTP to HTTPS or vice versa.
switchUser URLs		See Switch User , under Customizing URLs .
fii.alwaysReauthenticate	false	If true, re-authenticates when there is a Authentication in the SecurityContext
fii.rejectPublicInvocations	true	Disallow URL access when there is no request mapping
fii.validateConfigAttributes	true	Whether to check that all ConfigAttribute instances are valid at startup
fii.publishAuthorizationSuccess	false	Whether to publish an AuthorizedEvent after successful access check
fii.observeOncePerRequest	true	If false allow checks to happen multiple times, for example when JSP forwards are being used and filter security is desired on each included fragment of the HTTP request

13 Hierarchical Roles

Hierarchical roles are a convenient way to reduce clutter in your request mappings.

Property	Default Value	Meaning
roleHierarchy	none	Hierarchical role definition.

For example, if you have several types of 'admin' roles that can be used to access a URL pattern and you do not use hierarchical roles, you need to specify all the admin roles:

```
package com.mycompany.myapp

import grails.plugin.springsecurity.annotation.Secured

class SomeController {

    @Secured(['ROLE_ADMIN', 'ROLE_FINANCE_ADMIN', 'ROLE_SUPERADMIN'])
    def someAction() {
        ...
    }
}
```

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

```
grails.plugin.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.plugin.springsecurity.annotation.Secured

class SomeController {

    @Secured(['ROLE_ADMIN'])
    def someAction() {
        ...
    }
}
```

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

14 Switch User

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



This feature is very powerful; it allows full access to everything the switched-to user can access without requiring the user's password. Limit who can use this feature by guarding the user switch URL with a role, for example, `ROLE_SWITCH_USER`, `ROLE_ADMIN`, and so on.

Switching to Another User

To switch to another user, typically you 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>
```

Here the form is guarded by a check that the logged-in user has `ROLE_SWITCH_USER` and is not shown otherwise. You also need to guard the user switch URL, and the approach depends on your mapping scheme. If you use annotations, add a rule to the `controllerAnnotations.staticRules` attribute:

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

If you use `Requestmaps`, create a rule like this (for example, in `BootStrap`):

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

If you use the `Config.groovy` map, add the rule there:

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

Switching Back to Original User

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>
```

Customizing URLs

You can customize the URLs that are used for this feature, although it is rarely necessary:

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

Property	Default	Meaning
useSwitchUserFilter	false	Whether to use the switch user filter.
switchUser. switchUserUrl	<code>/j_spring_security_switch_user'</code>	URL to access (via GET or POST) to switch to another user.
switchUser. exitUserUrl	<code>/j_spring_security_exit_user'</code>	URL to access to switch to another user.
switchUser. targetUrl	Same as <code>successHandler.defaultTargetUrl</code>	URL for redirect after switching.
switchUser. switchFailureUrl	Same as <code>failureHandler.defaultFailureUrl</code>	URL for redirect after an error during an attempt to switch.
switchUser. usernameParameter	<code>SwitchUserFilter. SPRING_SECURITY_SWITCH_USERNAME_KEY</code>	The username request parameter name

GSP Code

One approach to supporting the switch user feature is 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, a form is displayed to allow input of the username 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>
```

15 Filters

There are a few different approaches to configuring filter chain(s).

Default Approach to Configuring Filter Chains

The default is to use configuration attributes to determine which extra filters to use (for example, Basic Auth, Switch User, etc.) and add these to the 'core' filters. For example, setting `grails.plugin.springsecurity.useSwitchUserFilter = true` adds `switchUserProcessingFilter` to the filter chain (and in the correct order). The filter chain built here is applied to all URLs. If you need more flexibility, you can use `filterChain.chainMap` as discussed in **chainMap** below.

filterNames

To define custom filters, to remove a core filter from the chain (not recommended), or to otherwise have control over the filter chain, you can specify the `filterNames` property as a list of strings. As with the default approach, the filter chain built here is applied to all URLs.

For example:

```
grails.plugin.springsecurity.filterChain.filterNames = [
    'securityContextPersistenceFilter', 'logoutFilter',
    'authenticationProcessingFilter', 'myCustomProcessingFilter',
    'rememberMeAuthenticationFilter', 'anonymousAuthenticationFilter',
    'exceptionTranslationFilter', 'filterInvocationInterceptor'
]
```

This example creates a filter chain corresponding to the Spring beans with the specified names.

chainMap

Use the `filterChain.chainMap` attribute to define which filters are applied to different URL patterns. You define a Map that specifies one or more lists of filter bean names, each with a corresponding URL pattern.

```
grails.plugin.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.

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 you need a `/**` catch-all rule at the end for URLs that do not match one of the earlier rules.

There's also a filter negation syntax that can be very convenient. Rather than specifying all of the filter names (and risking forgetting one or putting them in the wrong order), you can use the `JOINED_FILTERS` keyword and one or more filter names prefixed with a `-`. This means to use all configured filters except for the excluded ones. For example, if you had a web service that uses Basic Auth for `/webservice/**` URLs, you would configure that using:

```
grails.plugin.springsecurity.filterChain.chainMap = [
  '/webservice/**': 'JOINED_FILTERS,-exceptionTranslationFilter',
  '/**':
  'JOINED_FILTERS,-basicAuthenticationFilter,-basicExceptionTranslationFilter'
]
```

For the `/webservice/**` URLs, we want all filters except for the standard `ExceptionTranslationFilter` since we want to use just the one configured for Basic Auth. And for the `/**` URLs (everything else) we want everything except for the Basic Auth filter and its configured `ExceptionTranslationFilter`.

clientRegisterFilter

An alternative to setting the `filterNames` property is `grails.plugin.springsecurity.SpringSecurityUtils.clientRegisterFilter()`. This property allows you to add a custom filter to the chain at a specified position. Each standard filter has a corresponding position in the chain (see `grails.plugin.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 grails.plugin.springsecurity.SecurityFilterPosition
import grails.plugin.springsecurity.SpringSecurityUtils

class Bootstrap {
  def init = { servletContext ->
    SpringSecurityUtils.clientRegisterFilter(
      'myFilter', SecurityFilterPosition.OPENID_FILTER.order + 10)
  }
}
```

This bootstrap code registers 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. The Open ID filter position is just an example - add your filter in the position that makes sense.

16 Channel Security

Use channel security to configure which URLs require HTTP and which require HTTPS.

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

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

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

Header checking

The default implementation of channel security is fairly simple; if you're using HTTP but HTTPS is required, you get redirected to the corresponding SSL URL and vice versa. But when using a load balancer such as an F5 BIG-IP it's not possible to just check secure/insecure. In that case you can configure the load balancer to set a request header indicating the current state. To use this approach, set the `useHeaderCheckChannelSecurity` configuration property to `true` and optionally change the header names or values:

```
grails.plugin.springsecurity.secureChannel.useHeaderCheckChannelSecurity =  
true
```

By default the header name is "X-Forwarded-Proto" and the secure header value is "http" (i.e. if you're not secure, redirect to secure) and the insecure header value is "https" (i.e. if you're secure, redirect to insecure). You can change any or all of these default values though:

```
grails.plugin.springsecurity.secureChannel.secureHeaderName = '...'  
grails.plugin.springsecurity.secureChannel.secureHeaderValue = '...'  
grails.plugin.springsecurity.secureChannel.insecureHeaderName = '...'  
grails.plugin.springsecurity.secureChannel.insecureHeaderValue = '...'
```

17 IP Address Restrictions

Ordinarily you can guard URLs sufficiently with roles, but the plugin provides an extra layer of security with its ability to restrict by IP address.

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

For example, make an admin-only part of your site accessible only from IP addresses of the local LAN or VPN, such as 192.168.1.xxx or 10.xxx.xxx.xxx. You can also set this up at your firewall and/or routers, but it is convenient to encapsulate it 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-value strings, or multi-value lists of strings. They can use [CIDR](#) masks, and can specify either IPv4 or IPv6 patterns. For example, given this configuration:

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

`pattern1` URLs can be accessed only from the external address 123.234.345.456, `pattern2` URLs can be accessed only from a 10.xxx.xxx.xxx intranet address, and `pattern3` URLs can be accessed only from 10.10.200.42 or 10.10.200.63. All other URL patterns are accessible from any IP address.

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 the address format that is actually being used. Otherwise the filter throws exceptions. One option is to set the `java.net.preferIPv4Stack` system property, for example, by adding it to `JAVA_OPTS` or `GRAILS_OPTS` as `-Djava.net.preferIPv4Stack=true`.

18 Session Fixation Prevention

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

```
grails.plugin.springsecurity.useSessionFixationPrevention = true
```

Upon successful authentication a new HTTP session is created and the previous session's attributes are copied into it. If you start your session by clicking a link that was generated by someone trying to hack your account, which contained an active session id, you are no longer sharing the previous session after login. You have your own session.

Session fixation is less of a problem 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.

The table shows configuration options for session fixation.

Property	Default Value	Meaning
<code>useSessionFixationPrevention</code>	<code>false</code>	Whether to use session fixation prevention.
<code>sessionFixationPrevention.migrate</code>	<code>true</code>	Whether to copy the session attributes of the existing session to the new session after login.
<code>sessionFixationPrevention.alwaysCreateSession</code>	<code>false</code>	Whether to always create a session even if one did not exist at the start of the request.

19 Logout Handlers

You register a list of logout handlers by implementing the [LogoutHandler](#) interface. The list is called when a user explicitly logs out.

By default, a `securityContextLogoutHandler` bean is registered to clear the [SecurityContextHolder](#). Also, unless you are 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 are 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.

Property	Default Value	Meaning
<code>logout.handlerNames</code>	<code>['rememberMeServices', 'securityContextLogoutHandler']</code>	Logout handler bean names.

The beans must be declared either by the plugin or by you in `resources.groovy` or `resources.xml`. For example, suppose you have a custom `MyLogoutHandler` in `resources.groovy`:

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

You register it in `grails-app/conf/Config.groovy` as:

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

20 Voters

You can register a list of voters by implementing the [AccessDecisionVoter](#) interface. The list confirms whether a successful authentication is applicable for the current request.

Property	Default Value	Meaning
voterNames	['authenticatedVoter', 'roleVoter', 'webExpressionVoter']	Bean names of voters.

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`. Suppose you have a custom `MyAccessDecisionVoter` in `resources.groovy`:

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

You register it in `grails-app/conf/Config.groovy` as:

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


21 Miscellaneous Properties

Property	Default Value	Meaning
active	true	Whether the plugin is enabled.
printStatusMessages	true	Whether to print status messages such as "Configuring Spring Security Core ..."
rejectIfNoRule	true	'strict' mode where a request mapping is required for all resources; if true make sure to allow <code>IS_AUTHENTICATED_ANONYMOUSLY</code> for '/', '/js/**', '/css/**', '/images/**', '/login/**', '/logout/**', and so on.
anon. key	'foo'	anonymousProcessingFilter key.
atr. anonymousClass	grails.plugin.springsecurity.authentication.GrailsAnonymousAuthenticationToken	Anonymous token class.
useHttpSessionEventPublisher	false	If true, an HttpSession EventPublisher will be configured.
cacheUsers	false	If true, logins are cached using an EhCache. See Account Locking and Forcing Password Change, under User Cache .
useSecurityEventListener	false	If true, configure SecurityEventListener. See Events .
dao. reflectionSaltSourceProperty	none	Which property to use for the reflection-based salt source. See Salted Passwords
dao. hideUserNotFoundExceptions	true	if true, throws a new BadCredentialsException if a username is not found or the password is incorrect, but if false re-throws the UsernameNotFoundException thrown by UserDetailsService (considered less secure than throwing BadCredentialsException for both exceptions)
requestCache.createSession	true	Whether caching SavedRequest can trigger the creation of a session.
roleHierarchy	none	Hierarchical role definition. See Hierarchical Role Definition .
voterNames	['authenticatedVoter', 'closureVoter']	'roleVoter', Bean names of voters. See Voters .

providerNames	['daoAuthenticationProvider', 'anonymousAuthenticationProvider', 'rememberMeAuthenticationProvider']	Bean names of authentication providers. See Authentication Providers .
securityConfigType	'Annotation'	Type of request mapping to use, one of "Annotation", "Requestmap", or "InterceptUrlMap" (or the corresponding enum value from SecurityConfigType). See Configuring Request Mappings to Secure URLs .
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 "InterceptUrlMap". See Simple Map in Config.groovy .
registerLoggerListener	false	If true, registers a LoggerListener that logs interceptor-related application events.
s c r . allowSessionCreation	true	Whether to allow creating a session in t h e securityContextRepository bean
s c r . disableUrlRewriting	true	Whether to disable URL rewriting (and the jsessionid attribute)
scr. springSecurity ContextKey	HttpSessionSecurity ContextRepository. SPRING_SECURITY_ CONTEXT_KEY	The HTTP session key to store the SecurityContext under
scpf. forceEager SessionCreation	false	Whether to eagerly create a session in t h e securityContextRepository bean
sch. strategyName	SecurityContextHolder. MODE_THREADLOCAL	The strategy to use for storing the SecurityContext - can be one of MODE_THREADLOCAL, MODE_INHERITABLETHREADLOCAL, or MODE_GLOBAL, or the name of a class implementing SecurityContextHolderStrategy
debug. useFilter	false	Whether to use the DebugFilter to log request debug information to the console

providerManager. eraseCredentials AfterAuthentication	true	Whether to remove the password from the <code>Authentication</code> and its child objects after successful authentication
---	------	---

22 Tutorials

22.1 Using Controller Annotations to Secure URLs

1. Create your Grails application.

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

2. Install the plugin by adding it to BuildConfig.groovy

```
plugins {  
    compile ':spring-security-core:2.0'  
}
```

3. Create the User and Role domain classes.

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

You can choose your names for your domain classes and package; these are just examples.



Depending on your database, some domain class names might not be valid, especially those relating to security. Before you create names like "User" or "Group", make sure they are not reserved keywords in your database.

The script creates this User class:

```

package com.testapp

package test

class User {

    transient springSecurityService

    String username
    String password
    boolean enabled = true
    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
    }

    def beforeInsert() {
        encodePassword()
    }

    def beforeUpdate() {
        if (isDirty('password')) {
            encodePassword()
        }
    }

    protected void encodePassword() {
        password = springSecurityService.encodePassword(password)
    }
}

```



Earlier versions of the plugin didn't include password hashing logic in the domain class, but it makes the code a lot cleaner.

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 {

    private static final long serialVersionUID = 1

    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) {
        UserRole.where {
            user == User.load(userId) &&
            role == Role.load(roleId)
        }.get()
    }

    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 u, Role r, boolean flush = false) {
        int rowCount = UserRole.where {
            user == User.load(u.id) &&
            role == Role.load(r.id)
        }.deleteAll()

        rowCount > 0
    }

    static void removeAll(User u) {
        UserRole.where {
            user == User.load(u.id)
        }.deleteAll()
    }

    static void removeAll(Role r) {
        UserRole.where {
            role == Role.load(r.id)
        }.deleteAll()
    }

    static mapping = {
        id composite: ['role', 'user']
        version false
    }
}

```

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. They are examples to get you started, so you can edit them as you please. They contain the minimum needed for the plugin.

The plugin has no support for CRUD actions and GSPs for your domain classes; the spring-security-ui plugin supplies a UI for those. So for now you will create roles and users in `grails-app/conf/Bootstrap.groovy`. (See step 7.)

4. Create a controller that will be restricted by role.

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

This command creates `grails-app/controllers/com/testapp/SecureController.groovy`. Add some output so you can verify that things are working:

```
package com.testapp

class SecureController {
    def index() {
        render 'Secure access only'
    }
}
```

5. Start the server.

```
$ grails run-app
```

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

7. Shut down the app (using CTRL-C) and edit `grails-app/conf/Bootstrap.groovy` to add the security objects that you need.

```

import com.testapp.Role
import com.testapp.User
import com.testapp.UserRole

class BootStrap {
def init = { servletContext ->
def adminRole = new Role(authority: 'ROLE_ADMIN').save(flush: true)
    def userRole = new Role(authority: 'ROLE_USER').save(flush: true)

def testUser = new User(username: 'me', 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 the preceding `BootStrap.groovy`:

- The example does not use a traditional GORM many-to-many mapping for the User<->Role relationship; instead you are mapping the join table with the UserRole class. This performance optimization helps significantly when many users have one or more common roles.
- We explicitly flushed the creates because BootStrap does not run in a transaction or `OpenSessionInView`.

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

```

package com.testapp

import grails.plugin.springsecurity.annotation.Secured

class SecureController {
@Secured(['ROLE_ADMIN'])
    def index() {
        render 'Secure access only'
    }
}

```

or

```

package com.testapp

import grails.plugin.springsecurity.annotation.Secured

@Secured(['ROLE_ADMIN'])
class SecureController {
    def index() {
        render 'Secure access only'
    }
}

```


You can annotate the entire controller or individual actions. In this case you have only one action, so you can do either.

9. Run `rails run-app` again and navigate to <http://localhost:8080/bookstore/secure>.

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.

10. Test the Remember Me functionality.

Check the checkbox, and once you've tested the secure page, close your browser and reopen it. Navigate again to the secure page. Because a cookie is stored, you should not need to log in again. Logout at any time by navigating to <http://localhost:8080/bookstore/logout>.

11. Optionally, create a CRUD UI to work with users and roles.

Run `rails generate-all` for the domain classes:

```
$ rails generate-all com.testapp.User
```

```
$ rails generate-all com.testapp.Role
```

Since the User domain class handles password hashing, there are no changes required in the generated controllers.

22.2 Migration From the Acegi Plugin

In this tutorial we'll discuss the general steps required to migrate from the Acegi plugin to the Spring Security Core plugin. A lot of the material here comes from [an email](#) that Lubos Pochman sent to the [User mailing list](#) documenting the steps he took to upgrade from the Acegi plugin.

This isn't a standard step-by-step tutorial since every application is different and the steps required will vary from project to project. Instead these are guidelines and things to keep in mind. You should also read [Section 2](#) and [Section 3](#).

The first thing to do is uninstall the Acegi plugin

```
$ rails uninstall-plugin acegi
```

and install Spring Security Core

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

If this were a new project the next step would be to run the [s2-quickstart](#) script but you wouldn't do this for an existing project where you already have a User and Role class, so it's a good idea to work through the [bookstore tutorial](#) and use the files generated in that project. The files that the script generates are

- `grails-app/domain/com/testapp/User.groovy`
- `grails-app/domain/com/testapp/Role.groovy`
- `grails-app/domain/com/testapp/UserRole.groovy`
- `grails-app/controllers/LoginController.groovy`
- `grails-app/controllers/LogoutController.groovy`
- `grails-app/views/login/auth.gsp`
- `grails-app/views/login/denied.gsp`

Migrate any changes you made in `LoginController.groovy`, `LogoutController.groovy`, `auth.gsp` and `denied.gsp`, and overwrite your files with those. Do the same for `User.groovy` and `Role.groovy`, and move `UserRole.groovy` into your project.

User and Role UI

You can use the standard Grails `generate-all` script to create a UI to manage Users and Roles as described in the previous tutorial, or for a more complete solution use the [Spring Security UI](#) plugin.

authenticateService

The utility service in Spring Security Core is `SpringSecurityService`, so you need to replace `def authenticateService` with `def springSecurityService`. Many of the methods have the same names and signatures but there are some differences:

- `principal()` was renamed to `getPrincipal()`
- `ifAllGranted()`, `ifNotGranted()`, and `ifAnyGranted()` were removed; use `grails.plugin.springsecurity.SpringSecurityUtils.ifAllGranted()`, `ifNotGranted()`, and `ifAnyGranted()` instead
- `getSecurityConfig()` was removed, use `SpringSecurityUtils.getSecurityConfig()` instead

One significant change between the plugins is that the `UserDetails` implementation (`GrailsUser`) no longer has a reference to the domain class instance. This was intended to make it easy to access User class data that's not available in the `Principal` but it has frustrating side effects due to being a disconnected Hibernate object. Instead `GrailsUser` stores the user's id so you can conveniently retrieve the instance when needed. So instead of

```
def user = authenticateService.userDomain()
user = User.get(user.id)
```

use this instead:

```
def user = User.get(springSecurityService.principal.id)
```

Role granting

The Acegi plugin uses a standard Grails many-to-many relationship (i.e. using `hasMany` and `belongsTo`) between `User` and `Role` but this will have performance problems if you have many users. Spring Security Core also uses a many-to-many relationship but maps the join table as a domain class instead of using collections. In the Acegi plugin you would grant a role to a user using

```
Role role = ...
User user = ...
role.addToPeople(user)
```

and remove the grant with

```
Role role = ...
User user = ...
role.removeFromPeople(user)
```

In Spring Security Core you use the helper methods in `UserRole`

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

and

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

which directly insert or delete rows in the User/Role join table.

SecurityConfig.groovy

Configuration settings are now stored in `grails-app/conf/Config.groovy` along with the rest of the application configuration. The primary motivation for this change is to easily support environment-specific security settings. Migrate settings from `SecurityConfig.groovy` to `Config.groovy` (see [this summary](#) for the new names).

In particular it's important that the following properties be configured (replace class and package names to match your domain classes):

```
grails.plugin.springsecurity.userLookup.userDomainClassName =
    'com.yourcompany.yourapp.User'
grails.plugin.springsecurity.userLookup.authorityJoinClassName =
    'com.yourcompany.yourapp.UserRole'
grails.plugin.springsecurity.authority.className =
    'com.yourcompany.yourapp.Role'
```

Delete `SecurityConfig.groovy` when you're finished.

Controller annotations

The `Secured` annotation changed from `org.codehaus.groovy.grails.plugins.springsecurity.Secured` to `grails.plugin.springsecurity.annotation.Secured`. Consider using [SpEL expressions](#) since they're a lot more powerful and expressive than simple role names.

Security tags

- tag names now start with 'if' instead of 'is', and the role attribute changed to roles, so for example change `<g:ifAnyGranted role='... '>` to `<sec:ifAnyGranted roles='... '>`
- use `<sec:username/>` instead of `<g:loggedInUserInfo(field:'username')/>` - use `<sec:loggedInUserInfo>` to render other `GrailsUser` attributes

See more details about the taglibs in [Section 6](#).

23 Controller MetaClass Methods

The plugin registers some convenience methods into all controllers in your application. All are accessor methods, so they can be called as methods or properties. They include:

isLoggedIn

Returns true if there is an authenticated user.

```
class MyController {
  def someAction() {
    if (isLoggedIn()) {
      ...
    }
    ...
    if (!isLoggedIn()) {
      ...
    }
    // or
    if (loggedIn) {
      ...
    }
    if (!loggedIn) {
      ...
    }
  }
}
```

getPrincipal

Retrieves the current authenticated user's Principal (a `GrailsUser` instance unless you've customized this) or null if not authenticated.

```
class MyController {
  def someAction() {
    if (isLoggedIn()) {
      String username = getPrincipal().username
      ...
    }
    // or
    if (isLoggedIn()) {
      String username = principal.username
      ...
    }
  }
}
```

getAuthenticatedUser

Loads the user domain class instance from the database that corresponds to the currently authenticated user, or null if not authenticated. This is the equivalent of adding a dependency injection for `springSecurityService` and calling `PersonDomainClassName.get(springSecurityService.principal.id)` (the typical way that this is often done).

```
class MyController {  
  def someAction() {  
    if (isLoggedIn()) {  
      String email = getAuthenticatedUser().email  
      ...  
    }  
  }  
  // or  
  if (isLoggedIn()) {  
    String email = authenticatedUser.email  
    ...  
  }  
}
```

24 Internationalization

Spring Security Core plugin is provided with English, French and German i18n messages.

If you want to customize or translate the texts then add messages for the following keys to your i18n resource bundle(s) for each exception:

Message	Default Value	Exception
springSecurity.errors.login.expired	"Sorry, your account has expired."	AccountExpiredException
springSecurity.errors.login.passwordExpired	"Sorry, your password has expired."	CredentialsExpiredException
springSecurity.errors.login.disabled	"Sorry, your account is disabled."	DisabledException
springSecurity.errors.login.locked	"Sorry, your account is locked."	LockedException
springSecurity.errors.login.fail	"Sorry, we were not able to find a user with that username and password."	Other exceptions

You can customize all messages in `auth.gsp` and `denied.gsp`:

Message	Default Value
springSecurity.login.title	Login
springSecurity.login.header	Please Login
springSecurity.login.button	Login
springSecurity.login.username.label	Username
springSecurity.login.password.label	Password
springSecurity.login.remember.me.label	Remember me
springSecurity.denied.title	Denied
springSecurity.denied.message	Sorry, you're not authorized to view this page.

25 Migrating from the Acegi plugin

If you have an old application that uses the [Acegi plugin](#) you can use the information here to migrate to the newer approaches in this plugin. Note that these guides were written for the 1.x versions of the plugin, so you'll also need to be aware of what is new in 2.0 and above.

25.1 Differences Between the Spring Security and Acegi Plugins

The Spring Security plugin is a successor to the [Acegi plugin](#). The sections that follow compare the two.

Core Similarities and Differences

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

- Form-based authentication
- Storing users, roles, and optionally requestmaps in the database, with access through domain classes
- Guarding URLs with annotations, requestmap domain class, or static configuration
- Security tags
- Security service
- Security events
- Ajax login
- Basic authentication
- Switch User
- Channel security
- IP address restrictions

and adds several new features:

- Digest authentication
- Session Fixation Prevention
- Salted passwords
- Certificate (x509) login
- Hierarchical roles
- Account locking and forcing password change

Features Not Included in the Spring Security Plugin

The following features are not included in the Spring Security plugin, but are (or will be) available in secondary plugins that extend and depend on the core plugin:

- [OpenID](#)
- [ACL](#)
- [LDAP](#)
- [CAS](#)
- User registration
- Facebook
- NTLM
- Kerberos

Script Differences

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

The Spring Security plugin uses only one script, [s2-quickstart](#). It is similar to `create-auth-domains` because it creates domain classes and login and logout bcontrollers, but it appends files to `grails-app/conf/Config.groovy` instead of creating a standalone configuration file. There is no equivalent to `generate-manager` or `generate-registration` because an optional UI plugin generates 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 discuss required changes to the generated source files, for example, hashing passwords before saving or updating a user.

UserDetails Differences

The Acegi plugin extends the `UserDetails` instance and adds an accessor for the person domain class instance that is used to populate the `UserDetails`. Because the `Authentication` is kept in the HTTP session and the `UserDetails` is attached to that, it is easy to access non-security data such as full name, email, and so on without hitting the database.

However, with this approach, if the domain class has a lot of data, you increase the size of the session payload, which is exacerbated by clustered sessions. Further, any lazy-loaded collections fail to load after retrieving the person from the session because it would have become a detached Hibernate object. This problem is addressed by a call to `person.attach()` or by reloading by id, for example:

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

But with this approach, the person class is essentially a very large wrapper around its primary key since that's the real data you're storing.

To resolve this issue, the Spring Security plugin does not store the domain class but instead stores the id so you can retrieve the person easily:

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

The preceding approach works because the UserDetails implementation is an instance of `grails.plugin.springsecurity.userdetails.GrailsUser`, which extends the standard Spring Security [User](#) and adds a `getId()` method.


You can further extend this class if you want to store more data along with the authentication to avoid database access. See [Custom UserDetailsService](#).

25.2 Migrating from the Acegi Plugin

If you formerly used the Acegi plugin, change your application configuration settings as follows.

Setting	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

The table shows names of corresponding configuration properties.

 The plugin's configuration values all start with `grails.plugin.springsecurity` to distinguish them from similarly named options in Grails and from other plugins. You must specify all property overrides with the `grails.plugin.springsecurity` suffix. For example, you specify the attribute `password.algorithm` as:

```
grails.plugin.springsecurity.password.algorithm='bcrypt'
```

in `Config.groovy`

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 = "Requestmap"
requestMapClass	requestMap.className
requestMapPathField	requestMap.urlField
requestMapConfigAttributeField	requestMap.configAttributeField
useControllerAnnotations	securityConfigType = "Annotation"
controllerAnnotationsMatcher	controllerAnnotations.matcher
controllerAnnotationsMatchesLowercase	controllerAnnotations.lowercase
controllerAnnotationStaticRules	controllerAnnotations.staticRules
controllerAnnotationsRejectIfNoRule	rejectIfNoRule

requestMapString	N/A - 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 is supported in the UI plugin
mailHost	N/A - registration is supported in the UI plugin
mailUsername	N/A - registration is supported in the UI plugin
mailPassword	N/A - registration is supported in the UI plugin
mailProtocol	N/A - registration is supported in the UI plugin
mailFrom	N/A - registration is supported in the UI plugin
mailPort	N/A - registration is supported in the UI plugin
defaultRole	N/A - registration is supported in the UI plugin
useOpenId	N/A - supported in the OpenID plugin
openIdNonceMaxSeconds	N/A - supported in the OpenID plugin
useLdap	N/A - supported in the LDAP plugin
ldapRetrieveGroupRoles	N/A - supported in the LDAP plugin
ldapRetrieveDatabaseRoles	N/A - supported in the LDAP plugin
ldapSearchSubtree	N/A - supported in the LDAP plugin
ldapGroupRoleAttribute	N/A - supported in the LDAP plugin
ldapPasswordAttributeName	N/A - supported in the LDAP plugin
ldapServer	N/A - supported in the LDAP plugin
ldapManagerDn	N/A - supported in the LDAP plugin
ldapManagerPassword	N/A - supported in the LDAP plugin
ldapSearchBase	N/A - supported in the LDAP plugin
ldapSearchFilter	N/A - supported in the LDAP plugin
ldapGroupSearchBase	N/A - supported in the LDAP plugin
ldapGroupSearchFilter	N/A - supported in the LDAP plugin
ldapUsePassword	N/A - 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 - supported in the CAS plugin
cas.casServer	N/A - supported in the CAS plugin
cas.casServerPort	N/A - supported in the CAS plugin
cas.casServerSecure	N/A - supported in the CAS plugin
cas.localhostSecure	N/A - supported in the CAS plugin
cas.failureURL	N/A - supported in the CAS plugin
cas.defaultTargetURL	N/A - supported in the CAS plugin
cas.fullLoginURL	N/A - supported in the CAS plugin
cas.fullServiceURL	N/A - supported in the CAS plugin
cas.authenticationProviderKey	N/A - supported in the CAS plugin
cas.userDetailsService	N/A - supported in the CAS plugin
cas.sendRenew	N/A - supported in the CAS plugin
cas.proxyReceptorUrl	N/A - supported in the CAS plugin
cas.filterProcessesUrl	N/A - 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