Table of contents

OpenID authentication support for the Spring Security plugin.

OpenID authentication support for the Spring Security plugin. - Reference Documentation

Authors: Burt Beckwith

Version: 2.0-RC2

Table of Contents

1 Introduction to the Spring Security OpenID Plugin

The OpenID plugin adds OpenID login support to a Grails application that uses Spring Security. It depends on the Spring Security Core plugin.

Using OpenID authentication frees you from having to maintain passwords for those users, but it also poses some challenges.

In a typical application that uses form-based logins with Spring Security, all of your user information is stored in the database. Since an OpenID user authenticates at an OpenID provider, you don't maintain their password but Spring Security needs information to populate an Authentication - username, roles, and account statuses (enabled, locked, etc.) Only the username is available from the OpenID login (plus optionally some attributes made available by Attribute Exchange), and the rest is provided by your application, usually from the database.

The plugin supports two workflows to integrate OpenID authentication with local user accounts. One is user registration, and the other is linking one or more OpenIDs with a valid local account. Both workflows are triggered by a successful OpenID authentication followed by a `UsernameNotFoundException` indicating that a local user wasn't found. The plugin provides basic implementations of both workflows but each application is different, so you'll most likely need to customize and extend the initial implementation.

1.1 History

- Version 2.0-RC2
  - released October 27, 2013
- Version 2.0-RC1
  - released October 08, 2013
- Version 1.0.4
  - released July 24, 2012
- Version 1.0.3
  - released July 31, 2011
- Version 1.0.2
  - released March 27, 2011
- Version 1.0.1
  - released February 13, 2011
- Version 1.0
  - released July 27, 2010
- Version 0.1.1
  - released May 12, 2010
- Version 0.1
  - released May 03, 2010

2 Usage

The central issue with integrating OpenID logins with local application users is that OpenID providers can allow multiple login names but the provider response will always be whatever their canonical name for your identity is. For example, I can login with my Yahoo address `burtbeckwith@yahoo.com` but what is returned is a URL that looks like `https://me.yahoo.com/a/CkkjY454mYx10td2e05dqasd5Jedt8VAgg--%27`. If I had registered as a regular user on the site with `burtbeckwith@yahoo.com` as my username, my login would fail unexpectedly. To get it to work I'd have to know my `https://me.yahoo.com/...` name to propertly register, which will frustrate users.

When a user authenticates in Spring Security, an `Authentication` is created and stored in a `ThreadLocal` by the `SecurityContextHolder`. Typically the `Authentication`'s principal is an instance of `UserDetails` (in the plugin this is implemented by the `GrailsUser` class) and this is a very simple class. It's really just a POJO with fields for username, password, granted authorities, and status booleans (enabled, account locked, etc.) So when an OpenID user has no associated local account, there's no direct way to specify authorities or the statuses (the password is optional in this case since that's only used for database authentication). We can assume the user is enabled as long as the OpenID authentication succeeded, so we really just need a way to determine what an OpenID user's roles are, otherwise they won't be able to do any more in the application than a non-authenticated user.

Associating OpenIDs with local accounts

This plugin has two features that address this issues. One is the ability to associate multiple OpenIDs with a user record. Recall that the Core plugin generates a user class that looks like this:

```
package com.yourcompany.yourapp

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)
   }
}
```

So to use this plugin you need to add a `hasMany` for a collection of OpenID domain classes (generated by the s2-create-openid script) used to store OpenIDs:

```
package com.yourcompany.yourapp

class User {

transient springSecurityService

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

static hasMany = [openIds: OpenID]

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)
   }
}
```

Now when an existing user authenticates with OpenID, you can detect that there's no local database with that username and display a page where the user can associate that OpenID with an existing account. Subsequent authentication attempts will use the plugin's enhanced `UserDetailsService` that looks for a user not just by username but also by OpenID, so OpenID authenticate attempts work fine, and form-based logins do too if they provide the correct password.

New accounts

That works for existing accounts, but how do we create these in the first place? When Spring Security throws a `UserNotFoundException` after a successful OpenID login, the plugin detects that the authentication is a valid OpenID authentication, and if configured to do so (i.e. if `grails.plugin.springsecurity.openid.registration.autocreate` is `true`) will redirect the user to a signup page. This way you can guide them through the process of creating an account. This is more efficient than presenting a regular registration workflow because their canonical OpenID for that provider will already be known and can be associated with the user record.

3 Tutorials

3.1 User registration and linking

In this tutorial we'll cover

- creating a sample application
- installing the plugin
- configuring the plugin
- using the account linking workflow
- using the registration workflow

First, create a new application:

```
$ grails create-app openidtest
$ cd openidtest
```

Install the OpenID plugin by adding a dependency in BuildConfig.groovy:

```
  plugins {
     …
     runtime ':spring-security-openid:2.0-RC1'
  }
```

This will also install the Spring Security Core plugin since it's a dependency of this one.

Run the `s2-quickstart script` to initialize the core plugin:

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

To support the remember-me checkbox, run the `s2-create-persistent-token` script to generate a domain class for persistent tokens:

```
$ grails s2-create-persistent-token
com.openidtest.PersistentLogin
```

To support linking one or more OpenIDs with local accounts, we need to create an `OpenID` domain class:

```
$ grails s2-create-openid com.openidtest.OpenID
```

and edit the generated user class and add a `hasMany` for an `openIds` property:

```
package com.openidtest

class User {

transient springSecurityService

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

static hasMany = [openIds: OpenID]

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)
   }
}
```

Now create some test users and grant them some roles in `grails-app/conf/BootStrap.groovy`:

```
import com.openidtest.Role
import com.openidtest.User
import com.openidtest.UserRole

class BootStrap {

def init = { servletContext ->
      def roleAdmin = new Role(authority: 'ROLE_ADMIN').save()
      def roleUser = new Role(authority: 'ROLE_USER').save()

def user = new User(username: 'user', password: 'password', enabled: true
).save()
      def admin = new User(username: 'admin', password: 'password', enabled:
true).save()

UserRole.create user, roleUser
      UserRole.create admin, roleUser
      UserRole.create admin, roleAdmin, true
   }
}
```

The plugin contains an `OpenIdController` but it'll be more natural to access its `createAccount` action under /login/ and we also want to use this controller's `auth` action instead of the core plugin's `LoginController.auth` since this one supports both OpenID and regular username/password logins, so add mappings in `grails-app/conf/UrlMappings.groovy` to support these changes:

```
class UrlMappings {

static mappings = {

"/login/auth" {
         controller = 'openId'
         action = 'auth'
      }
      "/login/openIdCreateAccount" {
         controller = 'openId'
         action = 'createAccount'
      }

...

}
}
```

Now create a controller that's secured with annotations for testing:

```
$ grails create-controller secure
```

and add this code:

```
package openidtest

import grails.plugin.springsecurity.annotation.Secured

class SecureController {

@Secured(['ROLE_ADMIN'])
   def admins() {
      render 'Logged in with ROLE_ADMIN'
   }

@Secured(['ROLE_USER'])
   def users() {
      render 'Logged in with ROLE_USER'
   }
}
```

and finally we're ready to run the app:

```
$ grails run-app
```

Navigate to http://localhost:8080/openidtest/secure/admins and you should be prompted with the login screen. Leave the Use OpenID checkbox checked and enter a valid OpenID. Don't check the remember-me checkbox yet (it doesn't work with the extended workflows where you create a new user or link an OpenID) and click the "Log in" button.

After authenticating at the OpenID provider, you'll be redirected to the registration page. Note that there's a link to just associate the current OpenID with a local account - for now click the "link this OpenID" link.

At the next screen enter the username and password for the user we created in BootStrap with ROLE_ADMIN ('admin'/'password') and you will be redirected to http://localhost:8080/openidtest/secure/admins

To test that the OpenID is linked to your account, logout by navigating to http://localhost:8080/openidtest/logout/ and navigate to http://localhost:8080/openidtest/secure/admins. Logging out removes the remember-me cookie, so authenticate again with your OpenID (this time check the remember-me checkbox) and it should skip the register/link step and go directly to the secured page. You can also repeat the process and switch to the username/password login and use that.

To test remember-me, close the browser and re-open it, and navigate to http://localhost:8080/openidtest/secure/admins. It should skip the authentication step entirely and show the page.

User Registration

Now let's create a new user associated with an OpenID. Logout again and navigate to http://localhost:8080/openidtest/secure/users to initiate a login for a resource that requires ROLE_USER.

Since you've already associated the previous OpenID with a user, you either need to use a second OpenID, or restart the application to clear out the in-memory database.

Login using either the new OpenID or the original after restarting, and after authenticating externally you'll be redirected to the registration page. This time create a new user. By default the password validation rules are rather strict and you can change them, but for now enter a password that's at least 8 characters and contains at least one number and at least one symbol (e.g. !, #, $, %, etc.)

You should be redirected to http://localhost:8080/openidtest/secure/users after creating the account. Now logout and log back in, both with the username/password for the account you created and the OpenID you linked and both should work.

3.2 Plain OpenID

In the previous tutorial we went through two workflows to allow linking OpenIDs to local accounts. Another option is to only support OpenID logins but not associate them with local accounts. This might be useful if you want to allow people to perform some basic actions like clicking Like and Don't Like buttons, adding comments, etc.

To support this, rather than showing the registration page when an authenticated OpenID user is redirected to your controller, you could just create an `Authentication` for them with dummy information. Recall that the minimum requirements to populate a `UserDetails` instance are the username, the status booleans (enabled, locked out, etc.) and one or more granted authorities. You could emulate a basic application user by using their OpenID as the username, setting all statuses to `true`, and granting them a virtual role, e.g. `ROLE_OPENID`.

Copy the plugin's `grails.plugin.springsecurity.openid.OpenIdController.groovy` to your application's grails-app/controllers directory and replace the existing `createAccount` action with this:

```
def createAccount() {

def config = SpringSecurityUtils.securityConfig

String openId =
session[OpenIdAuthenticationFailureHandler.LAST_OPENID_USERNAME]
   if (!openId) {
      flash.error = 'Sorry, an OpenID was not found'
      redirect uri: config.failureHandler.defaultFailureUrl
      return
   }

def user = new GrailsUser(openId, 'password', true, true,
         true, true, [new SimpleGrantedAuthority('ROLE_OPENID')], 0)

SCH.context.authentication = new UsernamePasswordAuthenticationToken(
         user, 'password', user.authorities)

session.removeAttribute
OpenIdAuthenticationFailureHandler.LAST_OPENID_USERNAME
   session.removeAttribute
OpenIdAuthenticationFailureHandler.LAST_OPENID_ATTRIBUTES

def savedRequest = requestCache.getRequest(request, response)
   if (savedRequest && !config.successHandler.alwaysUseDefault) {
      redirect url: savedRequest.redirectUrl
   }
   else {
      redirect uri: config.successHandler.defaultTargetUrl
   }
}
```

You'll need to add these imports:

```
import grails.plugin.springsecurity.userdetails.GrailsUser
import org.springframework.security.core.authority.SimpleGrantedAuthority
import org.springframework.security.core.context.SecurityContextHolder as SCH
```

To test this, add a new action to the secure controller that requires the virtual `ROLE_OPENID` role:

```
package openidtest

import grails.plugin.springsecurity.annotation.Secured

class SecureController {

@Secured(['ROLE_ADMIN'])
   def admins() {
      render 'Logged in with ROLE_ADMIN'
   }

@Secured(['ROLE_USER'])
   def users() {
      render 'Logged in with ROLE_USER'
   }

@Secured(['ROLE_OPENID'])
   def openid() {
      render 'Logged in with ROLE_OPENID'
   }
}
```

Then start the server with `grails run-app` and navigate to http://localhost:8080/openidtest/secure/openid, and login using any OpenID. Once you authenticate and get redirected back to your application you should see the text `Logged in with ROLE_OPENID` indicating that you're logged in as a basic OpenID user.

Note that since this is a fake role, there's no need to store it in the database since real application users will never be granted ROLE_OPENID.

4 Configuration

There are a few configuration options for OpenID.

All of these property overrides must be specified in `grails-app/conf/Config.groovy` using the `grails.plugin.springsecurity` suffix, for example

```
grails.plugin.springsecurity.openid.domainClass =
     'com.mycompany.myapp.OpenID'
```

Name	Default	Meaning
openid.claimedIdentity FieldName	'openid_identifier'	the login name form parameter
openid.nonceMaxSeconds	300	maximum life of the generated nonce shared with the OpenID provider; determines how long the authentication is allowed to take
openid.domainClass	'OpenID'	the full class name of the many-to-one OpenID domain class
openid.encodePassword	`false`	whether to encode the password in OpenIdController.createNewAccount (set to `true` if not using the new style of User domain class that auto-encrypts)
openid.registration. autocreate	`true`	if `true` will redirect valid OpenID authentications to the create and link user workflows, otherwise shows the standard login fail page
openid.registration. requiredAttributes	none	required Attribute Exchange attributes
openid.registration. optionalAttributes	[email: 'http://schema.openid.net/ contact/email']	optional Attribute Exchange attributes
openid.registration. createAccountUri	'/login/openIdCreateAccount'	redirect address used when `openid.registration.autocreate` is `true`
openid.registration. roleNames	['ROLE_USER']	a list of names of roles to grant to users who self-register after an OpenID authentication; the roles must already exist
openid.userLookup. openIdsPropertyName	'openIds'	the name of the property in the user class for the OpenID domain class collection

5 Attribute Exchange

TBD