OAuth2 Provider support for the Spring Security plugin.

Spring Security OAuth2 Plugin - Reference Documentation

Authors:

Version: 2.0-RC5

Table of Contents

1 Introduction to the Spring Security OAuth2 Plugin

The OAuth2 plugin adds OAuth 2.0 support to a Grails application that uses Spring Security. It depends or

Under the covers, Spring Security OAuth version 2.0.2.RELEASE is used by the plugin to provide OAuth library.

This plugin provides support for Grails domain classes necessary for providing OAuth 2.0 authorization. ' Spring Security Core's methods, i.e. request maps, annotations, intercept maps and careful configuration of

1.1 Change Log

- 2.0-RC5
  - Upgrade to Spring OAuth 2.0.7.RELEASE for compatibility with Spring Security Core RC5 (iss
  - Resolve minor problems affecting stateless access of OAuth 2.0 resources
  - Remove need to include `clientCredentialsAuthenticationProvider` in `grails.plugin.springsecurit
  - Document using plugin to create only authorization server only or only a resource server (issue #
- 2.0-RC4
  - Fix for Grails 2.5.0 (issue #76)
  - Add support for basic authentication (issue #80)
  - Fix access token header format in the docs (issue #84)
  - Throw exception on validation code save (issue #90)
  - Fixes and enhancements for additional information (issue #87)
  - Add support for unlimited refresh tokens (issue #75)
- 2.0-RC3
  - Upgrade to Spring OAuth 2.0.6.RELEASE (issue #63)
  - Fix problems with updating access tokens (issues #49, #50, and #68)
  - Add TravisCI build
  - Ensure Set-Cookie header is not set in response
  - Fix handling of scope parameter (issue #64)
- 2.0-RC2
  - Resolves session vulnerability (issue #42)
  - Upgrade to Spring Security OAuth2 2.0.4.RELEASE
  - Supports authorization auto-approval
  - Minor tweaks to domain models
- 2.0-RC1

    - Complete overhaul of the plugin
    - Requires/supports Spring Security Core 2.0-RC4
    - Uses Spring Security OAuth2 2.0.2.RELEASE
- 1.0.5.2
    - Fix #13 - Make clientSecret optional in client configuration structure
- 1.0.5.1
    - Merge pull request #21 (Burt's cleanup)
    - Use log wrapper instead of log4j
    - Depends on Grails 2.0 or greater (consistent with core plugin)
- 1.0.5
    - Initial release of plugin compatible with spring security core 2.0-RC2

2 Getting Started

The following assumes that the Spring Security Core plugin has been installed and its required domain clas

2.1 Install Plugin

Install the OAuth2 plugin by adding a dependency in `grails-app/conf/BuildConfig.groovy`:

```
plugins {
    compile ":spring-security-oauth2-provider:2.0-RC3"
}
```

This has a dependency on the Spring Security Core plugin, which will be installed if necessary.

2.2 Create Domain Classes

Run the s2-init-oauth2-provider script to generate the required domain classes.

2.3 Secure Authorization and Token Endpoints

Update the Core plugin's rules for the authorization and token endpoints so they are protected by Spring Se

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
        '/oauth/authorize.dispatch':      ["isFullyAuthenticated() and (request.g
        '/oauth/token.dispatch':          ["isFullyAuthenticated() and request.ge
        ...
```

The endpoints are standard Spring MVC controllers in the underlying Spring Security OAuth2 implementa

The additional restrictions on the allowed HTTP methods are to ensure compliance with the OAuth 2.0 spe

2.4 Exclude client_secret From Logs

Update the params exclusion list in `grails-app/conf/Config.groovy` so client secrets are not log

```
grails.exceptionresolver.params.exclude = ['password', 'client_secret']
```

2.5 (Optional) Customize Error and Confirm Access Views

When the plugin is installed, two views are copied for the error and confirm access pages. They are locatec

2.6 Client Registration

At this point your application is a proper OAuth 2.0 provider. You can now register clients in what ever me

```
def init = { servletContext ->
        new Client(
                clientId: 'my-client',
                authorizedGrantTypes: ['authorization_code', 'refresh_token', 'im
                authorities: ['ROLE_CLIENT'],
                scopes: ['read', 'write'],
                redirectUris: ['http://myredirect.com']
        ).save(flush: true)
    }
```

2.7 Controlling Access to Resources

Access to resources is controlled by the Spring Security Core plugin's access control mechanisms. Addi
OAuth2SecurityExpressionMethods for what is available in the plugin.

Using SPeL is the only tested and confirmed way to enforce OAuth 2.0 specific restrictions on resource ac

The following controller illustrates the use of OAuth 2.0 SPeL:

```
class SecuredOAuth2ResourcesController {

@Secured(["#oauth2.clientHasRole('ROLE_CLIENT')"])
    def clientRoleExpression() {
        render "client role expression"
    }

@Secured(["ROLE_CLIENT"])
    def clientRole() {
        render "client role"
    }

@Secured(["#oauth2.clientHasAnyRole('ROLE_CLIENT', 'ROLE_TRUSTED_CLIENT')"])
    def clientHasAnyRole() {
        render "client has any role"
    }

@Secured(["#oauth2.isClient()"])
    def client() {
        render "is client"
    }

@Secured(["#oauth2.isUser()"])
    def user() {
        render "is user"
    }

@Secured(["#oauth2.denyOAuthClient()"])
    def denyClient() {
        render "no client can see"
    }

@Secured(["permitAll"])
    def anyone() {
        render "anyone can see"
    }

def nobody() {
        render "nobody can see"
    }

@Secured(["#oauth2.clientHasRole('ROLE_TRUSTED_CLIENT') and #oauth2.isClient() an
    def trustedClient() {
        render "trusted client"
    }

@Secured(["hasRole('ROLE_USER') and #oauth2.isUser() and #oauth2.hasScope('trust'
    def trustedUser() {
        render "trusted user"
    }

@Secured(["hasRole('ROLE_USER') or #oauth2.hasScope('read')"])
    def userRoleOrReadScope() {
        render "user role or read scope"
    }
}
```

The filter chains must be configured to ensure stateless access to the token endpoint and any OAuth 2.0 res

```
grails.plugin.springsecurity.filterChain.chainMap = [
        '/oauth/token':
'JOINED_FILTERS,-oauth2ProviderFilter,-securityContextPersistenceFilter,-logoutFi
        '/securedOAuth2Resources/**':
'JOINED_FILTERS,-securityContextPersistenceFilter,-logoutFilter,-authenticationPr
        '/**':
'JOINED_FILTERS,-statelessSecurityContextPersistenceFilter,-oauth2ProviderFilter,
]
```

Please consult the section on Filter Chain Configuration for more information.

2.8 Trouble Shooting

If you encounter a `NullPointerException` while using the OAuth2 plugin, you might have run intc time of this writing (`1.1.6`) seems to have fixed this issue. To resolve the `NullPointerException` i

If an instance of one of the GORM backed classes that the plugin uses cannot be saved, an `OAuth2Val` flexibility to determine how to deal with this type of error. The typical reason for this exception being thro for further information about the `Errors`.

3 Example Flows

The following examples assume you have followed the steps outlined in the Getting Started section for an

```
def init = { servletContext ->

Role roleUser = new Role(authority: 'ROLE_USER').save(flush: true)

User user = new User(
         username: 'my-user',
         password: 'my-password',
         enabled: true,
         accountExpired: false,
         accountLocked: false,
         passwordExpired: false
    ).save(flush: true)

UserRole.create(user, roleUser, true)

new Client(
        clientId: 'my-client',
        authorizedGrantTypes: ['authorization_code', 'refresh_token', 'implicit',
        authorities: ['ROLE_CLIENT'],
        scopes: ['read', 'write'],
        redirectUris: ['http://myredirect.com']
    ).save(flush: true)
}
```

After retrieving an `access_token` via one of the flows, you must include this in the `Authorization`

For example, if you receive `7b9a989e-3702-4621-a631-fbd1a996fc94` as the `access_toke` a protected resource.

The examples below are given using CURL tool to make the requests. The plugin is compliant with RFC by the User-Agent with an HTTP GET.

3.1 Authorization Code Grant

The authorization code grant flow is initiated by directing your browser to the authorization endpoint:

```
http://localhost:8080/oauth2-test/oauth/authorize?response_type=code&client_id=my
```

You will be redirected to the login page. After signing in, you will be prompted to confirm the request. Do

```
http://myredirect.com/?code=139R59
```

The authorization code included in the query can be exchanged for an access token via the token endpoint:

```
curl -X POST \
     -d "client_id=my-client" \
     -d "grant_type=authorization_code" \
     -d "code=139R59" http://localhost:8080/oauth2-test/oauth/token
```

Using HTTP Basic for client authentication:

```
curl -X POST -u my-client: \
     -d "grant_type=authorization_code" \
     -d "code=139R59" http://localhost:8080/oauth2-test/oauth/token
```

You'll receive the `access_token` in the response:

```
{
    "access_token": "a1ce2915-8d79-4961-8abb-2c6f0fdb4aba",
    "token_type": "bearer",
    "refresh_token": "6540222d-0fb9-4b01-8d45-7be2bdfb68f9",
    "expires_in": 43199,
    "scope": "read"
}
```

3.2 Implicit Grant

The implicit grant is similar to the authorization code grant and can be initiated by directing your browser t

```
http://localhost:8080/oauth2-test/oauth/authorize?response_type=token&client_id=m
```

Upon confirmation, your browser will be redirected to the following URL:

```
http://myredirect.com/#access_token=4e22ad4f-08ae-49dc-befb-2c9821af04d1&token_ty
```

The `access_token` can be extracted from the URL fragment.

3.3 Resource Owner Password Credentials Grant

The resource owner password grant is performed by requesting an access token from the token endpoint:

```
curl -X POST \
     -d "client_id=my-client" \
     -d "grant_type=password" \
     -d "username=my-user" \
     -d "password=my-password" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

Using HTTP Basic for client authentication:

```
curl -X POST -u my-client: \
     -d "grant_type=password" \
     -d "username=my-user" \
     -d "password=my-password" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

The `access_token` is included in the response:

```
{
    "access_token": "1d49fc35-2af6-477e-8fd4-ab0353a4a76f",
    "token_type": "bearer",
    "refresh_token": "4996ba33-be3f-4555-b3e3-0b094a4e60c0",
    "expires_in": 43199,
    "scope": "read"
}
```

3.4 Client Credentials Grant

The client credentials grant is performed by authenticating the client via the token endpoint:

```
curl -X POST \
     -d "client_id=my-client" \
     -d "grant_type=client_credentials" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

Using HTTP Basic for client authentication:

```
curl -X POST -u my-client: \
     -d "grant_type=client_credentials" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

The `access_token` can be extracted from the response:

```
{
    "access_token": "7b9a989e-3702-4621-a631-fbd1a996fc94",
    "token_type": "bearer",
    "expires_in": 43199,
    "scope": "read"
}
```

3.5 Refresh Token Grant

The refresh token grant is performed by exchanging a refresh token received during a previous authorizatic

```
curl -X POST \
     -d "client_id=my-client" \
     -d "grant_type=refresh_token" \
     -d "refresh_token=269afd46-0b41-45c2-a920-7d5af8a38d56" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

Using HTTP Basic for client authentication:

```
curl -X POST -u my-client: \
     -d "grant_type=refresh_token" \
     -d "refresh_token=269afd46-0b41-45c2-a920-7d5af8a38d56" \
     -d "scope=read" http://localhost:8080/oauth2-test/oauth/token
```

The above assumes that `269afd46-0b41-45c2-a920-7d5af8a38d56` is the value of the refresh tc

The `access_token` is included in the response:

```
{
    "access_token": "a3da52c7-4bd2-4d42-a58d-efa64b4de453",
    "token_type": "bearer",
    "refresh_token": "6396c283-47ff-41d2-b887-39bde6af5f1e",
    "expires_in": 43199,
    "scope": "read"
}
```

4 Required Domain Classes

The plugin uses regular Grails domain classes backed by GORM. There are four required domain classes r

The s2-init-oauth2-provider script will create the domain classes for you in a specified package and updat change the default property names, you will need to update `grails-app/conf/Config.groovy` so

> ⚠ The `maxSize` constraints in the generated domain classes have been set to reasonable defaul or have many authorities attached to a single user.

The below discussion assumes the s2-init-oauth2-provider script has been run with `com.yourapp` specif

4.1 Client Class

Information from the Grails client domain class will be extracted to create a `ClientDetails` instance fo

The generated class will look like this:

```
package com.yourapp

class Client {

private static final String NO_CLIENT_SECRET = ''

transient springSecurityService

String clientId
    String clientSecret

Integer accessTokenValiditySeconds
    Integer refreshTokenValiditySeconds

Map<String, Object> additionalInformation

static hasMany = [
            authorities: String,
            authorizedGrantTypes: String,
            resourceIds: String,
            scopes: String,
            autoApproveScopes: String,
            redirectUris: String
    ]

static transients = ['springSecurityService']

static constraints = {
        clientId blank: false, unique: true
        clientSecret nullable: true

accessTokenValiditySeconds nullable: true
        refreshTokenValiditySeconds nullable: true

authorities nullable: true
        authorizedGrantTypes nullable: true

resourceIds nullable: true

scopes nullable: true
        autoApproveScopes nullable: true

redirectUris nullable: true
        additionalInformation nullable: true
    }

def beforeInsert() {
        encodeClientSecret()
    }

def beforeUpdate() {
        if(isDirty('clientSecret')) {
            encodeClientSecret()
        }
    }

protected void encodeClientSecret() {
        clientSecret = clientSecret ?: NO_CLIENT_SECRET
        clientSecret = springSecurityService?.passwordEncoder ? springSecuritySer
    }
}
```

The client secret is encoded using the same strategy that is configured by the Core plugin for handling pass

4.2 Access Token Class

This class represents an access token than has been issued to a client on behalf of a user. The authenticatio

```
package com.yourapp

class AccessToken {

String authenticationKey
    byte[] authentication

String username
    String clientId

String value
    String tokenType

Date expiration
    Map<String, Object> additionalInformation

static hasOne = [refreshToken: String]
    static hasMany = [scope: String]

static constraints = {
        username nullable: true
        clientId nullable: false, blank: false
        value nullable: false, blank: false, unique: true
        tokenType nullable: false, blank: false
        expiration nullable: false
        scope nullable: false
        refreshToken nullable: true
        authenticationKey nullable: false, blank: false, unique: true
        authentication nullable: false, minSize: 1, maxSize: 1024 * 4
        additionalInformation nullable: true
    }

static mapping = {
        version false
        scope lazy: false
    }
}
```

4.3 Refresh Token Class

This class represents a refresh token issued as part of one of the grants that supports issuing a refresh toke
more. The authentication object serialized is an instance of `OAuth2Authentication` from Spring Sec

```
package com.yourapp

class RefreshToken {

String value
    Date expiration
    byte[] authentication

static constraints = {
        value nullable: false, blank: false, unique: true
        expiration nullable: true
        authentication nullable: false, minSize: 1, maxSize: 1024 * 4
    }

static mapping = {
        version false
    }
}
```

If a non-expiring refresh token is desired, the client issuing the refresh token should be configured to retur
non-expiring GORM refresh token will be stored with a `null` expiration. When reading a GORM refre
Spring Security OAuth. Otherwise an instance of `OAuth2RefreshToken` will be created and used.

4.4 Authorization Code Class

This class represents an authorization code that has been issued via the authorization endpoint as part of
2.0.

```
package com.yourapp

class AuthorizationCode {

byte[] authentication
    String code

static constraints = {
        code nullable: false, blank: false, unique: true
        authentication nullable: false, minSize: 1, maxSize: 1024 * 4
    }

static mapping = {
        version false
    }
}
```

5 Optional Domain Classes

The plugin provides support for using a GORM backed ApprovalStore with the ApprovalStor
configured to use the UserApprovalSupport.APPROVAL_STORE method of auto-approval.

The s2-init-oauth2-approval script will create the required domain class for you in a specified package and
change the default property names, you will need to update grails-app/conf/Config.groovy so

The below discussion assumes the s2-init-oauth2-approval script has been run with com.yourapp specif

5.1 Approval Class

This class represents a prior scoped approval granted to a client by a user.

```
package com.yourapp

class Approval {

String username
    String clientId

String scope
    boolean approved

Date expiration
    Date lastModified

static constraints = {
        username nullable: false, blank: false
        clientId nullable: false, blank: false
        scope nullable: false, blank: false
        expiration nullable: false
        lastModified nullable: false
    }
}
```

6 Domain Class Properties

No default class name is assumed for the required domain classes. They must be specified in `grails` `grails.plugin.springsecurity.oauthProvider` namespace.

6.1 Client Class Properties

Property	Default Value	M
clientLookup.className	`null`	C
clientLookup.clientIdPropertyName	`'clientId'`	C
clientLookup.clientSecretPropertyName	`'clientSecret'`	C
clientLookup.accessTokenValiditySecondsPropertyName	`'accessTokenValiditySeconds'`	C
clientLookup.refreshTokenValiditySecondsPropertyName	`'refreshTokenValiditySeconds'`	C
clientLookup.authoritiesPropertyName	`'authorities'`	C
clientLookup.authorizedGrantTypesPropertyName	`'authorizedGrantTypes'`	C
clientLookup.resourceIdsPropertyName	`'resourceIds'`	C
clientLookup.scopesPropertyName	`'scopes'`	C
clientLookup.autoApproveScopesPropertyName	`'autoApproveScopes'`	C
clientLookup.redirectUrisPropertyName	`'redirectUris'`	C
clientLookup.additionalInformationPropertyName	`'additionalInformation'`	C

6.2 Access Token Class Properties

Property	Default Value	Meaning
accessTokenLookup.className	`null`	Access tok
accessTokenLookup.authenticationKeyPropertyName	`'authenticationKey'`	Access tok
accessTokenLookup.authenticationPropertyName	`'authentication'`	Access tok
accessTokenLookup.usernamePropertyName	`'username'`	Access tok
accessTokenLookup.clientIdPropertyName	`'clientId'`	Access tok
accessTokenLookup.valuePropertyName	`'value'`	Access tok
accessTokenLookup.tokenTypePropertyName	`'tokenType'`	Access tok
accessTokenLookup.expirationPropertyName	`'expiration'`	Access tok
accessTokenLookup.refreshTokenPropertyName	`'refreshToken'`	Access tok
accessTokenLookup.scopePropertyName	`'scope'`	Access tok
accessTokenLookup.additionalInformationPropertyName	`'additionalInformation'`	Access tok

Currently only `'bearer'` tokens are supported.

6.3 Refresh Token Class Properties

Property	Default Value	Meaning
refreshTokenLookup.className	`null`	Refresh token class name.
refreshTokenLookup.authenticationPropertyName	`'authentication'`	Refresh token class serialized
refreshTokenLookup.valuePropertyName	`'value'`	Refresh token class value fie
refreshTokenLookup.expirationPropertyName	`'expiration'`	Refresh

6.4 Authorization Code Class Properties

Property	Default Value	Meaning
authorizationCodeLookup.className	`null`	Authorization code clas
authorizationCodeLookup.authenticationPropertyName	`'authentication'`	Authorization code clas
authorizationCodeLookup.codePropertyName	`'code'`	Authorization code clas

7 Configuration

The plugin is pessimistic by default, locking down as much as possible to guard against accidental security
in the `grails.plugin.springsecurity.oauthProvider` namespace.

7.1 Plugin

The following properties define whether the plugin is active and where the required filters are registered in

Property	Default Value
active	`true`
filterStartPosition	`SecurityFilterPosition.X509_FILTER.order`
clientFilterStartPosition	`SecurityFilterPosition.DIGEST_AUTH_FILTER.or`
statelessFilterStartPosition	`SecurityFilterPosition.SECURITY_CONTEXT_FILT`
exceptionTranslationFilterStartPosition	`SecurityFilterPosition.EXCEPTION_TRANSLATION`
basicAuthenticationFilterStartPosition	`SecurityFilterPosition.BASIC_AUTH_FILTER.ord`
registerStatelessFilter	`true`
registerExceptionTranslationFilter	`true`
registerBasicAuthenticationFilter	`true`
realmName	`Grails OAuth2 Realm`

7.2 Endpoint URLs

The endpoint URLs used by the underlying Spring Security OAuth 2.0 implementation can be changed usi

Property	Default Value	Meaning
authorizationEndpointUrl	`'/oauth/authorize'`	Authorization endpoint URL.
tokenEndpointUrl	`'/oauth/token'`	Token endpoint URL.
userApprovalEndpointUrl	`'/oauth/confirm_access'`	URL of the view to display for confirming
userApprovalParameter	`'user_oauth_approval'`	The name of the parameter submitted in th
errorEndpointUrl	`'/oauth/error'`	URL of the view to display if an error occ URI. This is usually the case when there is

When changing the URL for the `authorizationEndpointUrl` or `tokenEndpointUrl`, you **mu** `grails-app/conf/Config.groovy` will look like this:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
        '/oauth/authorize.dispatch':      ["isFullyAuthenticated() and (request.g
        '/oauth/token.dispatch':          ["isFullyAuthenticated() and request.ge
        ...
```

To change the `authorizationEndpointUrl` to `/authorize`, you will need to make the following

```
grails.plugin.springsecurity.oauthProvider.authorizationEndpointUrl = '/authorize

grails.plugin.springsecurity.controllerAnnotations.staticRules = [
        '/authorize.dispatch':            ["isFullyAuthenticated() and (request.g
        '/oauth/token.dispatch':          ["isFullyAuthenticated() and request.ge
        ...
```

The URL mapping must include the `.dispatch` suffix in order integrate with the underlying Spring MV

7.3 Token Services

The following properties apply to how tokens are issued and how long they are valid. If a client has define

Property	Default Value	Meaning
tokenServices.registerTokenEnhancers	`true`	Whether registered `TokenEnha`
tokenServices.accessTokenValiditySeconds	`60 * 60 * 12`	The length of time that an access
tokenServices.refreshTokenValiditySeconds	`60 * 60 * 24 * 30`	The length of time that a refresh t
tokenServices.reuseRefreshToken	`false`	Whether a new refresh token shou
tokenServices.supportRefreshToken	`true`	Whether a refresh token can be is

7.4 Token Enhancers Configuration

By default, the plugin will register a `TokenEnhancerChain` with an empty list of `TokenEnhancer` beans implementing the `TokenEnhancer` interface.

If more control over the ordering of the enhancers in the chain is desired, set the `tokenServices.reg` so the plugin consumer can get a handle to it for more fine grained configuration.

This bean is aliased under the name `tokenEnhancer`. This is the bean that is registered with the `toke` bean.

7.5 Supported Grant Types

The following properties determine which of the standard grant types the application can support. Individu

Property	Default Value	Meaning
grantTypes.authorizationCode	`true`	Whether the Authorization Code Grant is supported.
grantTypes.implicit	`true`	Whether the Implicit Grant is supported.
grantTypes.clientCredentials	`true`	Whether the Client Credentials Grant is supported.
grantTypes.password	`true`	Whether the Resource Owner Password Credentials is sup
grantTypes.refreshToken	`true`	Whether Refresh Token Grant is supported.

7.6 Additional Authorization Constraints

The plugin enforces the following restrictions on authorization request params:

Property	Default Value	Meaning
authorization.requireRegisteredRedirectUri	`true`	Whether a client is required to have registered *Manipulation* and *RFC 6749 Section 10.15: Ope*
authorization.requireScope	`true`	Whether the scope for each access token requeste

7.7 User Approval Configuration

The plugin provides support for the three `UserApprovalHandler` implementations provided by the ur by the application. The following properties determine which method of auto-approval to use and how it is

Property	Default Value	Meaning
auto	`EXPLICIT`	Determines which method of auto-approval to use. The be `EXPLICIT`, `TOKEN_STORE` or `APPROVAL_STORE`
handleRevocationAsExpiry	`false`	When configured to use an approval store for auto-ap outright.
approvalValiditySeconds	`60 * 60 * 24 * 30`	When configured to use an approval store for auto-appro
scopePrefix	`'scope.'`	When configured to use an approval store for auto-appro

The `auto` property determines which of the three `UserApprovalHandler` provided by Spring OAuth

The default option is to require explicit approval for every authorization and is equivalent to setting `auto`

```
grails.plugin.springsecurity.oauthprovider.approval.auto = UserApproval.EXPLICIT
```

Auto-approval based on previously issued access tokens is supported via the `TokenStoreUserApprov`

```
grails.plugin.springsecurity.oauthprovider.approval.auto = UserApproval.TOKEN_STO
```

Auto-approval based on prior approvals is supported via the `ApprovalStoreUserApprovalHandle`

```
grails.plugin.springsecurity.oauthprovider.approval.auto = UserApproval.APPROVAL_
```

The plugin will configure the `TokenStoreUserApprovalHandler` and `ApprovalStoreUserAp`

Please consult Spring OAuth directly for more information on the usage of the `TokenStore` and `Appro`

7.8 Default Client Configuration

An application can use the following properties to define the default values that will be used when creat token unless they have explicitly registered support for the requested grant type.

Property	Default Value	Meaning
defaultClientConfig.resourceIds	`[]`	Resources the client is authorized to a
defaultClientConfig.authorizedGrantTypes	`[]`	Grant types the client supports.
defaultClientConfig.scope	`[]`	Scope to use for each access token req
defaultClientConfig.autoApproveScopes	`[]`	Scopes to auto-approve for authorizati
defaultClientConfig.registeredRedirectUri	`null`	URI to redirect the user-agent to durin
defaultClientConfig.authorities	`[]`	Roles and authorities granted to the cl
defaultClientConfig.accessTokenValiditySeconds	`null`	The length of time that an access toke
defaultClientConfig.refreshTokenValiditySeconds	`null`	The length of time that a refresh token
defaultClientConfig.additionalInformation	`[:]`	Additional information about the clien

7.9 Filter Chain Configuration

Spring Security Core plugin's `securityContextPersistenceFilter` stores state in the HTTP ses

By default, the OAuth2 plugin will register the `statelessSecurityContextPersistenceFilt` as a convenience for the plugin consumer, so they can remove one filter or the other to easily ach `registerStatelessFilter` configuration option to `false`.

The plugin registers an `OAuth2AuthenticationProcessingFilter` under the bean name `oauth`

The plugin registers a `ClientCredentialsTokenEndpointFilter` under the bean name `clien` plugin also registers a `BasicAuthenticationFilter` under the bean name `oauth2BasicAuth` method in the OAuth 2.0 specification.

Finally, the plugin registers an ExceptionTranslationFilter under the bean name HttpSessionRequestCache instance that the Spring Security Core plugin provided ExceptionT automatically by the plugin but can be disabled by setting the registerExceptionTranslationFi

The following filter chain configuration is recommended:

```
grails.plugin.springsecurity.filterChain.chainMap = [
        '/oauth/token':
'JOINED_FILTERS,-oauth2ProviderFilter,-securityContextPersistenceFilter,-logoutFi
        '/securedOAuth2Resources/**':
'JOINED_FILTERS,-securityContextPersistenceFilter,-logoutFilter,-authenticationPr
        '/**':
'JOINED_FILTERS,-statelessSecurityContextPersistenceFilter,-oauth2ProviderFilter,
]
```

The oauth2ProviderFilter and stateful securityContextPersistenceFilter and exce removed, the statelessSecurityContextPersistenceFilter will be used to en oauth2ExceptionTranslationFilter to take its place in the filter chain.

The securityContextPersistenceFilter and exceptionTranslationFilter are also responsible for authenticating the OAuth 2.0 access token included in the request.

It is recommend that filter chain(s) for non-OAuth 2.0 resources have all OAuth 2.0 spe clientCredentialsTokenEndpointFilter, basicAuthenticationFilter and rememberMeAuthenticationFilter and authenticationProcessingFilter are remove

7.10 Domain Class Custom Serialization Configuration

The default behavior of the plugin is to serialize the additionalInformation and scope prop s2-init-oauth2-provider script will generate the domain classes. However, this might not be ideal for all plu

To accommodate these users, it is possible to override the default serialization method on a case-by-case ba

For the additionalInformation fields:

```
package grails.plugin.springsecurity.oauthprovider.serialization;

import java.util.Map;

public interface OAuth2AdditionalInformationSerializer {

Object serialize(Map<String, Object> additionalInformation);

Map<String, Object> deserialize(Object additionalInformation);
}
```

For the scope field:

```
package grails.plugin.springsecurity.oauthprovider.serialization;
import java.util.Set;
public interface OAuth2ScopeSerializer {
Object serialize(Set<String> scopes);
Set<String> deserialize(Object scopes);
}
```

By default, the plugin registers implementations that do little more than return the argument provided to ea

Bean Name	Interface Implemented
`clientAdditionalInformationSerializer`	`OAuth2AdditionalInformationS`
`accessTokenAdditionalInformationSerializer`	`OAuth2AdditionalInformationS`
`accessTokenScopeSerializer`	`OAuth2ScopeSerializer`

Overriding these beans in `resources.groovy` will allow the plugin consumer to customize how the change the `AccessToken` to serialized its `additionalInformation` as JSON `String` and its `scc`

First, modify the `AccessToken` class to reflect the change in the storage of these fields:

```
package test.oauth2

class AccessToken {

String authenticationKey
    byte[] authentication

String username
    String clientId

String value
    String tokenType

Date expiration
    String additionalInformation

String scope

static hasOne = [refreshToken: String]

static constraints = {
        username nullable: true
        clientId nullable: false, blank: false
        value nullable: false, blank: false, unique: true
        tokenType nullable: false, blank: false
        expiration nullable: false
        scope nullable: false
        refreshToken nullable: true
        authenticationKey nullable: false, blank: false, unique: true
        authentication nullable: false, minSize: 1, maxSize: 1024 * 4
        additionalInformation nullable: true
    }

static mapping = {
        version false
        scope lazy: false
    }
}
```

Next, implement the earlier described interfaces:

```
package test

import grails.plugin.springsecurity.oauthprovider.serialization.OAuth2ScopeSerial
import org.springframework.security.oauth2.common.util.OAuth2Utils

class WhiteSpaceDelimitedStringScopeSerializer implements OAuth2ScopeSerializer {

@Override
    Object serialize(Set<String> scopes) {
        return OAuth2Utils.formatParameterList(scopes)
    }

@Override
    Set<String> deserialize(Object scopes) {
        return OAuth2Utils.parseParameterList(scopes)
    }
}
```

And:

```
package test

import grails.plugin.springsecurity.oauthprovider.serialization.OAuth2AdditionalI
import groovy.json.JsonOutput
import groovy.json.JsonSlurper

class JsonAdditionalInformationSerializer implements OAuth2AdditionalInformationS

@Override
    Object serialize(Map<String, Object> additionalInformation) {
        JsonOutput.toJson(additionalInformation)
    }

@Override
    Map<String, Object> deserialize(Object additionalInformation) {
        new JsonSlurper().parseText(additionalInformation as String)
    }
}
```

⚠ The `serialize` methods are guaranteed to receive a non-null argument, although they may

Finally, in `resources.groovy`, override the appropriate beans:

```
import test.JsonAdditionalInformationSerializer
import test.WhiteSpaceDelimitedStringScopeSerializer

beans = {
    // Other beans here

accessTokenAdditionalInformationSerializer(JsonAdditionalInformationSerializer)
    accessTokenScopeSerializer(WhiteSpaceDelimitedStringScopeSerializer)
}
```

8 Standalone Resource Server or Authorization Server

By default, the plugin is configured to assume the role of both the Authorization Server and the Resource S

The plugin registers an instance of the Spring OAuth provided `OAuth2AuthenticationProces` `Authorization` header and confirming its authenticity.

8.1 Authorization Server

To create an application that is only an Authorization Server, it is as simple as configuring the `oauth2ProviderFilter`.

So instead of the following filter chain:

```
grails.plugin.springsecurity.filterChain.chainMap = [
        '/oauth/token':
'JOINED_FILTERS,-oauth2ProviderFilter,-securityContextPersistenceFilter,-logoutFi
        '/securedOAuth2Resources/**':
'JOINED_FILTERS,-securityContextPersistenceFilter,-logoutFilter,-authenticationPr
        '/**':
'JOINED_FILTERS,-statelessSecurityContextPersistenceFilter,-oauth2ProviderFilter,
   ]
```

You would have something like this:

```
grails.plugin.springsecurity.filterChain.chainMap = [
            '/oauth/token':
'JOINED_FILTERS,-oauth2ProviderFilter,-securityContextPersistenceFilter,-logoutFi
            '/**':
'JOINED_FILTERS,-statelessSecurityContextPersistenceFilter,-oauth2ProviderFilter,
   ]
```

Where `/**` is any Authorization Server specific functionality.

8.2 Resource Server

To create an application that is only a Resource Server is slightly more involved. The plugin uses an im authenticate the presented `Bearer` token. If the Authorization Server and Resource Server are distinct ap case. To do this, simply implement the aforementioned `ResourceServerTokenServices` interface a

Next you will need to disable access to the authorization and token endpoints. This can be accomplished b the Authorization and Resource Servers are the same application:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    '/oauth/authorize.dispatch':      ["isFullyAuthenticated() and (request.getMe
    '/oauth/token.dispatch':          ["isFullyAuthenticated() and request.getMet
    '/':                              ['permitAll'],
    '/index':                         ['permitAll'],
    '/index.gsp':                     ['permitAll'],
    '/**/js/**':                      ['permitAll'],
    '/**/css/**':                     ['permitAll'],
    '/**/images/**':                  ['permitAll'],
    '/**/favicon.ico':                ['permitAll']
]
```

Splitting out the authorization parts will result in something like this:

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

Any requests to the authorization or token endpoints will be greeted with a 403 response. You should
following:

```
grails.plugin.springsecurity.filterChain.chainMap = [
            '/securedOAuth2Resources/**':
'JOINED_FILTERS,-securityContextPersistenceFilter,-logoutFilter,-rememberMeAuthen
            '/**':
'JOINED_FILTERS,-statelessSecurityContextPersistenceFilter,-oauth2ProviderFilter,
   ]
```

Where `/**` is any Resource Server specific functionality.