# OAuth 1 Developers Guide

## Introduction

This is the developers guide for the support for OAuth 1.0. For OAuth 2.0, everything is different, so see it's developers guide (oauth2.html).

This user guide is divided into two parts, the first for the OAuth 1.0 provider, the second for the OAuth 1.0 consumer. Here's a TOC for quick navigation:

## OAuth 1.0 Provider

The OAuth 1.0 provider is responsible for managing the OAuth 1.0 consumers that can access its protected resources on behalf of a user. The provider does this by managing and verifying the OAuth 1.0 tokens that can be used to access the protected resources. Of course, the provider must also supply an interface for the user to confirm that a consumer can be granted access to the protected resources (i.e. a confirmation page).

### Managing Consumers

The entry point into your database of consumers is defined by the `ConsumerDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ConsumerDetailsService.html). You must define your own `ConsumerDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ConsumerDetailsService.html) that will load `ConsumerDetails` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ConsumerDetails.html) by the *consumer key*. Note the existence of an in-memory implementation (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/InMemoryConsumerDetailsService.html) of `ConsumerDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ConsumerDetailsService.html).

When implementing your `ConsumerDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ConsumerDetailsService.html) consider returning instances of BaseConsumerDetails (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/BaseConsumerDetails.html) which contains additional information about the consumer that may be useful when displaying a confirmation screen to the user.

### Managing Tokens

The `OAuthProviderTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/token/OAuthProviderTokenServices.html) interface defines the operations that are necessary to manage OAuth 1.0 tokens. Note the following:

- When a request token is created, care must be taken to ensure that it is not an access token.
- When a request token is authorized, the authentication must be stored so that the subsequent access token can reference it.
- When an access token is created, it must reference the authentication that was used to authorized the request token that is used to create the access token.

When creating your `OAuthProviderTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/token/OAuthProviderTokenServices.html) implementation, you may want to consider extending the `RandomValueProviderTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/token/RandomValueProviderTokenServices.html) which creates tokens via random value and handles everything except for the persistence of the tokens. There is also an in-memory implementation (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/token/InMemoryProviderTokenServices.html) of the `OAuthProviderTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/token/OAuthProviderTokenServices.html) that may be suitable, but note that when using the in-memory implementation a separate thread is spawned to take care of the cleanup of expired tokens.

### OAuth 1.0 Provider Request Filters

The requests for the tokens and for access to protected resources are handled by standard Spring Security request filters. The following filters are required in the Spring Security filter chain in order to implement OAuth 1.0:

- The `UnauthenticatedRequestTokenProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/UnauthenticatedRequestTokenProcessingFilter.html) is used to service the request for an unauthenticated request token. Default URL: `/oauth_request_token`.
- The `UserAuthorizationProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/UserAuthorizationProcessingFilter.html) is used authorize a request token. The user must be authenticated and it is assumed that the user has been presented with the appropriate confirmation page.
- The `AccessTokenProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/AccessTokenProcessingFilter.html) is used to service the request for an OAuth 1.0 access token. Default URL: `/oauth_access_token`.
- The `ProtectedResourceProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/ProtectedResourceProcessingFilter.html) is used to load the Authentication for the request given an authenticated access token.

### Managing Nonces

The OAuth 1.0 spec also recommends that the nonce that is supplied on every OAuth 1.0 request be checked to ensure it isn't used twice for the same timestamp. In order to do this, nonces must be stored and verified on every OAuth 1.0 request. The interface that is used to validate nonces is `OAuthNonceServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/nonce/OAuthNonceServices.html). The default implementation, `ExpiringTimestampNonceServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/nonce/ExpiringTimestampNonceServices.html), does not adhere to this recommendation, but only validates that the timestamp isn't too old. If further assurance is required, you will need to supply your own implementation of `OAuthNonceServices`. Note the existence of an in-memory implementation (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/nonce/InMemoryNonceServices.html).

### Managing Callbacks

With the 1.0a revision of the OAuth 1.0 specification, the callback URL is provided at the time the request is made for a request token and will be used when redirecting the user back to the OAuth 1.0 consumer. Therefore, a means must be provided to persist the callback between requests. The interface that is used to persist callbacks is `OAuthCallbackServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/callback/OAuthCallbackServices.html). The default implementation, `InMemoryCallbackServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/callback/InMemoryCallbackServices.html) persists the callbacks in-memory. You must supply your own implementation of `OAuthCallbackServices` if this is inadequate.

### Managing Verifiers

With the 1.0a revision of the OAuth 1.0 specification, the a verifier is provided to the consumer via the user that must be passed back to the provider when requesting the access token. Therefore, a means must be provided to create and persist the verifier. The interface that is used to this end is `OAuthVerifierServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/verifier/OAuthVerifierServices.html). The default implementation, `RandomValueInMemoryVerifierServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/verifier/RandomValueInMemoryVerifierServices.html), creates a small, user-friendly (6 readable ASCII characters by default) verifier and persists the verifier in memory. You must supply your own implementation of `OAuthVerifierServices` if this is inadequate.

### Authorization By Consumer

It is sometimes required to limit access to a resource to a specific consumer or to a consumer that has specific roles. The classes in the `org.springframework.security.oauth.provider.attributes` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/attributes/package-summary.html) package can be used to do this. Methods can be protected using the annotations in that package, and the `ConsumerSecurityConfig` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/attributes/ConsumerSecurityConfig.html) can be supplied to the standard Spring Security filter interceptor in order to enable the annotations. Finally, the `ConsumerSecurityVoter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/provider/attributes/ConsumerSecurityVoter.html) would need to be supplied to the Spring Security authentication manager.

### Provider Configuration

For the OAuth 1.0 provider, configuration is simplified using the custom spring configuration elements. The schema for these elements rests at http://www.springframework.org/schema/security/spring-security-oauth.xsd (http://www.springframework.org/schema/security/spring-security-oauth.xsd). The namespace is `http://www.springframework.org/schema/security/oauth`.

The following configuration elements are used to supply provider configuration:

## The "provider" element

The `provider` element is used to configure the OAuth 1.0 provider mechanism. The following attributes can be applied to the `provider` element:

- `consumer-details-service-ref`: The reference to the bean that defines the consumer details service. This is required if not autowired.
- `token-services-ref`: The reference to the bean that defines the token services.
- `request-token-url`: The URL at which a request for an unauthenticated request token will be serviced. Default value: "/oauth_request_token"
- `authenticate-token-url`: The URL at which a request to authenticate a request token will be serviced. Default value: "/oauth_authenticate_token"
- `access-token-url`: The URL at which a request for an access token (using an authenticated request token) will be serviced. Default value: "/oauth_access_token"
- `access-granted-url`: The URL to which the user will be redirected upon authenticating a request token, but only if there was no callback URL supplied from the oauth consumer. Default value: "/"
- `user-approval-url`: The URL to which the user will be redirected if for some reason authentication of a request token failed. Default behavior is to just issue a "401: unauthorized" response.
- `nonce-services-ref`: The reference to the bean that defines the nonce services. Default is to supply an instance of `org.springframework.security.oauth.provider.nonce.ExpiringTimestampNonceServices`
- `callback-services-ref`: The reference to the bean that defines the callback services. Default is to supply an instance of `org.springframework.security.oauth.provider.callback.InMemoryCallbackServices`
- `verifier-services-ref`: The reference to the bean that defines the verifier services. Default is to supply an instance of `org.springframework.security.oauth.provider.verifier.RandomValueInMemoryVerifierServices`
- `auth-handler-ref`: The reference to the bean that defines the authentication handler. Default is to supply an instance of `org.springframework.security.oauth.provider.DefaultAuthenticationHandler`
- `support-ref`: The reference to the bean that defines the provider support logic. Default is to supply an instance of `org.springframework.security.oauth.provider.CoreOAuthProviderSupport`
- `token-id-param`: The name of the request parameter that specifies to the 'authenticate-token-url' the id of the token that is to be authenticated. Default value: "requestToken".
- `callback-url-param`: The name of the request parameter that specifies to the 'authenticate-token-url' the callback URL to which the user is to be redirected upon successful authentication. Default value: "callbackURL".

## The "consumer-details-service" element

The `consumer-details-service` element is used to define an in-memory implementation of the consumer details service. It takes an `id` attribute and an arbitrary number of `consumer` child elements that define the following attributes for each consumer:

- `key`: (required) The consumer key.
- `secret`: (required) The consumer secret.
- `name`: The (display) name of the consumer.
- `authorities`: Comma-separated list of authorities (e.g. roles) that are granted to the consumer.
- `resourceName`: The name of the resource.
- `resourceDescription`: The description of the resource.
- `requiredToObtainAuthenticatedToken`: Whether this consumer is required to obtain an authenticated oauth token. If *true*, it means that the OAuth 1.0 consumer won't be granted access to the protected resource unless the user is directed to the token authorization page. If *false*, it means that the provider has an additional level of trust with the consumer. Not requiring an authenticated access token is also known as "2-legged" OAuth or "signed fetch". For more information, see two-legged OAuth (./twolegged.html).

## The "token-services" element

The `token-services` element is a simple element that can be used to provide an in-memory implementation of the provider token services. It supports an *id* attribute (bean id) and a *cleanupInterval* attribute that specifies how often the cleanup thread should wake up (in seconds).

## The "verifier-services" element

The `verifier-services` element is a simple element that can be used to provide an in-memory implementation of the provider verifier services. It supports an `id` attribute (bean id) and a `verifierLengthBytes` attribute that specifies the length of the verifier.

### Configuring An OAuth-Aware Expression Handler

You may want to take advantage of Spring Security's expression-based access control (http://static.springsource.org/spring-security/site/docs/3.0.x/reference/el-access.html). You can register a oauth-aware expression handler with the `expression-handler` element. Use the id of the oauth expression handler to add oauth-aware expressions to the built-in expressions.

The expressions include *oauthConsumerHasRole*, *oauthConsumerHasAnyRole*, and *denyOAuthConsumer* which can be used to provide access based on the role of the oauth consumer.

## OAuth 1.0 Consumer

The OAuth 1.0 consumer logic is responsible for (1) obtaining an OAuth 1 access token and (2) signing requests for OAuth 1 protected resources. OAuth for Spring Security provides a request filter for acquiring the access token, a request filter for ensuring that access to certain URLs is locked down to a set of acquired access token, and utilities for making a request for a protected resource. A consumer must be responsible for maintaing a list of protected resources that can be accessed and, like the provider, a consumer must be responsible for managing the OAuth 1.0 tokens.

If you were discouraged by the complexity of implementing an OAuth 1.0 provider, take heart. Implementation of an OAuth 1.0 consumer is easier, partially because OAuth 1.0 for Spring Security provides suitable defaults for most cases.

### Managing Protected Resources

A database of protected resources that are accessible by a consumer must be provided through the `ProtectedResourceDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/ProtectedResourceDetailsService.html). Each protected resource must provide all information related to obtaining access to it. This includes the URL to obtain a request token, the URL to which to redirect the user for authorization, the URL at which to obtain an access token, etc. It also contains various properties that describe the provider of the protected resource. Consider the existence of the `InMemoryProtectedResourceDetailsService` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/InMemoryProtectedResourceDetailsService.html) and the `BaseProtectedResourceDetails` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/BaseProtectedResourceDetails.html) for help in creating the database of protected resources.

### Managing Provider Tokens

Like the provider, the consumer must be responsible for managing the OAuth tokens. The necessary interface for managing the consumer tokens is `OAuthConsumerTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/token/OAuthConsumerTokenServices.html). Assuming that the consumer can leverage an active HTTP session, the default `HttpSessionBasedTokenServices` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/token/HttpSessionBasedTokenServices.html) might be adequate, but if you'd like to persist access tokens longer than a user session, you'll have to implement your own persistent implementation of the token services.

### OAuth 1.0 Consumer Request Filters

There are two request filters that are applicable to the OAuth consumer logic. The first filter, `OAuthConsumerContextFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/OAuthConsumerContextFilter.html), is responsible for establishing an OAuth-specific security context, very similar to Spring Security's `SecurityContext`. The security context simply contains a set of access tokens that have been obtained for the current user. This security context is leveraged when making requests for protected resources.

There is another request filter, `OAuthConsumerProcessingFilter` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/OAuthConsumerProcessingFilter.html), that can be applied to specific URLs or URL patterns that require access to a remote protected resource. Putting this filter in Spring Security's filter chain will ensure that any access tokens needed for the specified URL patters will be obtained before allowing access to the resources.

### Requesting Protected Resources

The `OAuthRestTemplate` (http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth/consumer/OAuthRestTemplate.html) can be used to make REST-like requests to resources protected by OAuth. It's used just like a standard RestTemplate (new in Spring 3), but is supplied with a specific `ProtectedResourcDetails` so it can sign its requests.

### Consumer Configuration

For the OAuth 1.0 consumer, configuration is simplified using the custom spring configuration elements. The schema for these elements rests at http://www.springframework.org/schema/security/spring-security-oauth.xsd (http://www.springframework.org/schema/security/spring-security-oauth.xsd). The namespace is `http://www.springframework.org/schema/security/oauth`.

Two custom configuration elements are used to supply provider configuration:

### The "consumer" element

The `consumer` element configures the OAuth 1.0 consumer mechanism. This element is used to set up the security filter(s) that will handle the OAuth consumer logic. The OAuth context filter establishes a context for the OAuth consumer logic. The OAuth access filter is used to apply OAuth constraints on specified URLs (request paths) in your application. The access filter is applied by specified one or more `url` child elements to the `consumer` element.

The `url` element supports the following attributes:

- `pattern`: (required) The URL pattern.
- `resources`: (required) Comma-separated list of the ids of the protected resources that the URL requires access to.
- `httpMethod`: The HTTP method that requires access. Default is all methods.

The `consumer` element also supports the following attributes:

- `resource-details-service-ref`: The reference to the resource details service. This is required if not autowired.
- `oauth-failure-page`: The page to which to redirect the user if a problem happens during OAuth 1.0 authentication.
- `entry-point-ref`: Reference to the entry point to use if a problem happens during OAuth 1.0 authentication (overrides *oauth-failure-page*).
- `path-type`: URL path type. Default value: "ant".
- `lowercase-comparisons`: Whether to use lowercase comparisons.
- `support-ref`: Reference to the OAuth 1.0 consumer support logic.
- `token-services-factory-ref`: Reference to the token services factory.

### The "resource-details-service" element

The `resource-details-service` element configures an in-memory implementation of the resource details. It supports an "id" attribute and an arbitrary number of `resource` child elements which are used to define the protected resources and support the following attributes:

- `id`: (required) The resource id.
- `key`: (required) The consumer key.
- `secret`: (required) The shared secret.
- `request-token-url`: (required) The URL to use to get the OAuth 1.0 request token.
- `user-authorization-url`: (required) The URL to which to redirect the user to authorize the request token.
- `access-token-url`: (required) The URL to use to get an OAuth 1.0 access token.
- `signature-method`: The signature method to use (e.g. "HMAC-SHA1", "PLAINTEXT", etc.). Default "HMAC-SHA1".
- `user-authorization-token-param`: Name of the request parameter to use to pass the value of the request token when redirecting the user to the authorization page. Default value: "requestToken"
- `user-authorization-callback-param`: Name of the request parameter to use to pass the value of the callback URL when redirecting the user to the authorization page. Default value: "callbackURL"
- `accepts-authorization-header`: Whether the provider accepts the HTTP authorization header. Default: "true"
- `authorization-header-realm`: The "realm" for the HTTP authorization header.
- `use10a`: Whether the resource is protected using OAuth 1.0a. Default: "true"

## Customizations Not Explicitly Supported by Namespace

The XML DSL has extension points for some of the most common use cases, generally specified through strategies injected through attributes (e.g. the `token-services-ref` in the `<provider/>`), but occasionally you may need to add customizations not supported in this way. Other cases can be handled locally without losing the benefit of the namespace because the bean definitions created are all designed to be easy to override. The namespace parsers create bean definitions with fixed bean definition names (hopefully easy to guess, but it is not hard to verify them by reading the source code of the parsers), and all you need to do to override one part of the namespace support is create a bean definition with the same name. For instance, the `<provider/>` element creates an `OAuthProviderProcessingFilter` which itself has a default `ProtectedResourceProcessingFilter`, but if you wanted to replace it you could override the bean definition:

```
 <oauth:provider .../>

 <bean  id="oauthProtectedResourceFilter" class="org.springframework.security.oauth.provider.filter.ProtectedResourceProcessing
Filter">
    ...
 </bean>
```

In this example, the explicit bean definition overrides the one created by the `<provider/>` because of the ordering in the application context declaration (this is a standard Spring bean factory feature). Bean definitions created by the namespace parsers follow the convention that they start with "oauth" and generally they are the class name of the default implementation provided by the framework.