

# Table of Contents

Consumer guide .....	1
Background .....	1
Overview .....	1
Providing a UI form element to let a user select credentials.....	2
Listing available credentials matching some specific set of criteria .....	5
Persist a reference to a specific credential instance .....	8
Retrieve a previously selected credentials instance .....	9
Retrieve the secret from a specific credentials instance.....	11
Track usage of a credential against specific Jenkins context objects .....	11

# Consumer guide

This document provides guidelines for plugin authors using the Credentials API from within their plugins.

This document is structured as follows:

- ¥ The first section provides some background into the use cases driving the development of the Credentials API.
- ¥ The second section is an overview of all the functionality provided by the Credentials API.
- ¥ The subsequent sections consider each of the API extension points in turn and the recommended usage patterns.

## Background

There are a lot of different use cases where Jenkins requires to access secrets. Even if Jenkins did not directly connect to additional third-party services there are cases, such as code signing, where Jenkins would need secrets.

Prior to the development of the Credentials API, each plugin was responsible for managing these secrets. In some cases this would give rise to user frustration where they would have to provide effectively the same secret multiple times for each plugin that required it.

In the case of username/password secrets, things were even worse. A lot of corporate IT security policies mandate that passwords be changed every 45-90 days. While the current [NIST standards](#) recommend against periodic rotation, the fact remains that a lot of organizations use enforced rotation.

When each plugin was storing the username/password separately, the user would be forced to race to update the password everywhere before Jenkins tried to use the old password three times in a row and trigger a logout of their account.

The Credentials API plugin provides a single centralized store of credentials within Jenkins, which should enable the user to easily update the credentials when they need to change.

## Overview

A plugin that is consuming the Credentials API has a relatively simple set of requirements:

- ¥ Provide a UI form element to let a user select credentials
- ¥ List available credentials matching some specific set of criteria.
- ¥ Persist a reference to a specific credential instance (typically selected by the user)
- ¥ Retrieve a previously selected credentials instance.
- ¥ Retrieve the secret from a specific credentials instance.
- ¥ Track usage of a credential against specific Jenkins context objects.

!

The Credentials API has evolved over time and there are a number of credentials interfaces that have been retained in order to maintain binary compatibility for plugins compiled against prior versions.

The `@Recommended` annotation is used to distinguish credentials interfaces that are recommended to use.

In a lot of cases, consumer plugins will want to use credentials to authenticate against some external service. The recommendation is to use the [Authentication Tokens API](#) to convert the Jenkins Credentials type into a service specific authentication as this allows the conversion logic to be separated cleanly from your connection code.

## Providing a UI form element to let a user select credentials

The Credentials API provides a custom Jenkins `<select>` form control that allows users to select credentials and provides users with appropriate permissions with the ability to add additional credentials.

In the normal case, using this form control is relatively simple:

```
<j:jelly ... xmlns:c="/lib/credentials" ... >
  Ê ..
  Ê <f:entry field="credentialId" title="${%Credentials}">
  Ê   <c:select/>
  Ê </f:entry>
</j:jelly>
```

There are some additional parameters available in this form control:

¥ In the case where the credentials to be used by a job may be selected by a job parameter, you may want to allow the user to choose between providing an expression and selecting the credentials:

```
<j:jelly ... xmlns:c="/lib/credentials" ... >
  Ê ..
  Ê <f:entry field="credentialId" title="${%Credentials}">
  Ê   <c:select expressionAllowed="true"/>
  Ê </f:entry>
</j:jelly>
```

||

The `expressionAllowed` option only makes sense for configuring credentials that will be used in the context of a `hudson.model.Run` (technically it must be a `Run` of a `ParameterizedJob`).

¥ In the case where the credentials will be used immediately in the context of the user selecting

the credentials (as opposed to deferred usage in the context of a queued build job), you may want to allow credentials from the user's per-user credentials store:

```
<j:jelly ... xmlns:c="/lib/credentials" ... >
  ...
  <f:entry field="credentialId" title="${%Credentials}">
    <c:select includeUser="true"/>
  </f:entry>
</j:jelly>
```

¥ In the case of failure to infer the correct context from which the credentials may be sourced, you can supply the correct context:

```
<j:jelly ... xmlns:c="/lib/credentials" ... >
  ...
  <f:entry field="credentialId" title="${%Credentials}">
    <c:select context="${...}"/>
  </f:entry>
</j:jelly>
```

You will need to supply a `doFillXYZItems` method on your descriptor in order to populate the drop-down list:

```
public ListBoxModel doFillCredentialIdItems(
    @AncestorInPath Item item,
    @QueryParameter String credentialId,
    ... !
) {
    StandardListBoxModel result = new StandardListBoxModel();
    if (item == null) {
        if (!Jenkins.getActiveInstance().hasPermission(Jenkins.ADMINISTER)) {
            return result.includeCurrentValue(credentialId);
        }
    } else {
        if (!item.hasPermission(Item.EXTENDED_READ)
            && !item.hasPermission(CredentialsProvider.USE_ITEM)) {
            return result.includeCurrentValue(credentialId);
        }
    }
    return result
        .includeEmptySelection() #
        .includeMatchingAs(...) $
        .includeCurrentValue(credentialId); %
}
```

! Include any additional contextual parameters that you need in order to refine the credentials list. For example, if the credentials will be used to connect to a remote server, you might include

the server URL form element as a `@QueryParam` so that the domain requirements can be built from that URL.

- " We protect against fully populating the drop-down list for users that have no ability to actually make a selection. This is also useful in preventing unwanted requests being made to an external credentials store.
- # If it is valid for the user to select no credentials, then include the empty selection.
- \$ We need to include the matching credentials. In some cases you may have disjoint unions of credentials, in which case you can call this method multiple times, the first credential added for any given ID wins.
- % If you include the current value then the form configuration will remain unchanged in the event that the backing credential is removed. An alternative is to let the form "magically" select a new credential, but typically this will be the wrong credential. The recommendation is to just add the "non-existing" credential and have form validation report the error

It is recommended to have a `doCheckXYZ` method to validate the credentials. Ideally this method will check the credentials against the service that they will be used with. At a minimum the method should verify that the credentials exist.

```
public FormValidation doCheckCredentialsId(
    @AncestorInPath Item item, "
    @QueryParam String value, !
    ... "
) {
    if (item == null) {
        if (!Jenkins.getInstance().hasPermission(Jenkins.ADMINISTER)) {
            return FormValidation.ok(); #
        }
    } else {
        if (!item.hasPermission(item.EXTENDED_READ)
            && !item.hasPermission(CredentialsProvider.USE_ITEM)) {
            return FormValidation.ok(); #
        }
    }
    if (StringUtil.isBlank(value)) { $
        return FormValidation.ok(); $
    }
    if (value.startsWith("${") && value.endsWith("}")) { %
        return FormValidation.warning("Cannot validate expression based credentials");
    }
    if (CredentialsProvider.listCredentials( &
        ...,
        CredentialsMatchers.withId(value) &
    ).isEmpty()) {
        return FormValidation.error("Cannot find currently selected credentials");
    }
    return FormValidation.ok();
}
```

- ! This is a `doCheckXYZ` method so the value to check will be in the parameter called `value`
- " Replicate the other parameters from the `doFindXYZItems` method.
- # Return no-op validation results for users that do not have permission to list credentials.
- \$ If anonymous connection is permitted, we can return OK for an empty selected credentials. Better yet would be to try and ping the remote service anonymously and report success / error if anonymous access is actually enabled on the remote service. *You may want to cache the check result for a short time-span if the remote service has rate limits on anonymous access.*
- % If you have not enabled credentials parameter expressions on the select control then you do not need this test.
- & This example checks that the credentials exist, but does not use them to connect. Alternatively `CredentialsMatchers.firstOrNull(CredentialsProvider.lookupCredentials(É), withId(value))` can be used to retrieve the credentials, a `null` return value would indicate that the error that they cannot be found, while the non-`null` return value could be used to validate the credentials against the remote service. *You may want to cache the check result for a short time-span if the remote service has rate limits.*

## Listing available credentials matching some specific set of criteria

We use the `CredentialsProvider.listCredentials()` overloads to list credentials. An external credentials provider may be recording usage of the credential and as such the `listCredentials` methods are supposed to not access the secret information and hence should not trigger such usage records.

!

If you are listing available credentials in order to populate a drop-down list, then `StandardListBoxModel.includeMatchingAs()` may be a more convenient way to call `CredentialsProvider.listCredentials()`

There are currently two overloads, one taking `Item` as the context and the other taking `ItemGroup` as the context, the other parameters are otherwise identical.

||

A current RFE [JENKINS-39324](#) is looking to replace overloaded methods with a single method taking the more generic `ModelObject`.

The parameters are:

`type`

The type of credentials to list.

`item` or `itemGroup`

The context within which to list available credentials.

`authentication`

The authentication that is listing available credentials.

## domainRequirements

The requirements of the credentials domains appropriate to the credentials.

## matcher

The additional criteria that must be met by the credentials.

Here are some examples of usage:

¥ We want to let the user select the username password credentials to be used to check out the source from source control as part of a build:

```
CredentialsProvider.listCredentials(  
    Ê StandardUsernamePasswordCredentials.class, !  
    Ê job, "  
    Ê job instanceof Queue.Task #  
    Ê ? Tasks.getAuthenticationOf((Queue.Task)job) $  
    Ê : ACL.SYSTEM, %  
    Ê URIRequirementBuilder.fromUri(scmUrl), &  
    Ê null '  
);
```

! We want username password credentials and we want the user to be able to selected them  
We need `IdCredentials.getId()` to allow retrieval when building the job. We need `UsernamePasswordCredentials` to ensure that they are username and password.

" We will be using these credentials in the context of a specific job.

# For almost all implementations of `Job`, this will be `true`. (Note: `external jobs` do not implement `Queue.Task`)

\$ This is important, we must use the authentication that the job is likely to run as.

% If not a `Queue.Task` then use `ACL.SYSTEM`

& We use the requirements builder most idiomatically appropriate to our use case. In most cases, unless `URIRequirementBuilder` can be used to construct at least some domain requirements.

' We do not have any additional requirements to place, so we can specify `null` for the matcher.

¥ We want to let the user select the username password credentials to be used to tag a build:

```
CredentialsProvider.listCredentials(  
    Ê StandardUsernamePasswordCredentials.class,  
    Ê job,  
    Ê Jenkins.getAuthentication(), !  
    Ê URIRequirementBuilder.fromUri(scmUrl),  
    Ê null  
)
```

! This is a UI immediate operation that will be performed as the user not a queued operation that will be performed as the authentication of the `Job`.

¥ We want to let the user select the credentials to update the issue tracker with each build result of a specific job.

```
CredentialsProvider.listCredentials(  
    StandardCredentials.class, !  
    job,  
    job instanceof Queue.Task  
    ? Tasks.getAuthenticationOf((Queue.Task)job)  
    : ACL.SYSTEM,  
    URIRequirementBuilder.fromUri(issueTrackerUrl),  
    AuthenticationTokens.matcher(IssueTrackerAuthentication.class) "  
)
```

! There are different credential types that can be used to authenticate with the issue tracker, e.g. application tokens, OAuth authentication, user/password, etc. The only common feature is that we need each credential to have a `IdCredentials.getId()` to allow retrieval when building the job.

" We need to limit the credentials to those types that we can actually use to authenticate with the issue tracker. We leverage the [Authentication Tokens API](#) to convert the Jenkins credentials types into the issue tracker API's native authentication types. This means that we can use the `AuthenticationTokens.matcher(Class)` to restrict the list of credentials to the subset that can be converted. Alternatively, more complex conversion contexts can be handled with `AuthenticationTokens.matcher(AuthenticationTokenContext)`

¥ We want to let the user select the credentials to toggle the deployment state of a blue/green deployment for a completed build.

```
CredentialsProvider.listCredentials(  
    StandardCredentials.class, !  
    job,  
    Jenkins.getAuthentication(), "  
    URIRequirementBuilder.fromUri(loadBalancerUrl),  
    CredentialMatchers.allOf(  
        AuthenticationTokens.matcher(LoadBalancerAuthentication.class),  
        CredentialMatchers.withProperty("permission", "lb.switch") #  
    )  
)
```

! There are different credential types that can be used to authenticate with the load balancer, this is the common base class.

" This is an immediate action performed by the user.

# In this case there may be multiple credentials available to the user, we only want the ones with `"lb.switch".equals(credential.getPermission())`. Any credentials that do not have a `getPermission()` method will be excluded as well as any that do not have the corresponding return value.

¥ We want to let the user specify the credentials used to update the post commit receive hooks of



a source control system for any corresponding jobs configured in Jenkins.

- || This drop down list would typically be displayed from one of the *Manage Jenkins* screens and limited to Jenkins Administrators.

```
CredentialsProvider.listCredentials(  
    StandardUsernameCredentials.class, !  
    Jenkins.getInstance(), "  
    ACL.SYSTEM, "  
    URIRequirementBuilder.fromUri(scmUrl),  
    AuthenticationTokens.matcher(MySCMAuthentication.class) !  
)
```

- ! For this SCM, management of post commit hooks requires authentication that has specified a username, so even though there are other authentication mechanisms supported by `AuthenticationTokens.matcher(É)` we limit at the type level as that reduces the response that needs to be filtered. The alternative would have been a matcher that combined `CredentialsMatchers.instanceOf(StandardUsernameCredentials.class)` but this reduces the ability of an external credentials provider to filter the query on the remote side.
- " We are doing this operation outside of the context of a single job, rather this is being performed on behalf of the entire Jenkins instance. Thus we should be performing this as `ACL.SYSTEM` and in the context of `Jenkins.getInstance()`. This has the additional benefit that the admin can restrict the high permission hook management credentials to `CredentialsScope.SYSTEM` which will prevent access by jobs.

## Persist a reference to a specific credential instance

You should be using `StandardCredentials` (or a more specific sub-interface) to access the Credentials API. Persist the `IdCredentials.getId()` of a credentials. For any given context and authentication, the `IdCredentials.getId()` should consistently return the same credentials instance.

- ! Do not make any assumptions about the format of a `IdCredentials.getId()` other than it must not contain `{É}`

## Historical evolution of the API

Prior to the introduction of the `StandardCredentials` interface, the assumption was that each credentials type would have its own unique identifier, e.g. a `UsernamePasswordCredentials` would have the username, etc.

This breaks down because the same username may have different passwords for different services, hence the introduction of a separate ID.

Initially the ID was generated using a UUID in order to ensure that it was unique. Also the intent was that the generated ID would be completely hidden from the user.

The introduction of configuration as code, such as `Jenkins pipelines` means that users now need to be able to specify the ID from code, and consequently when users are creating credentials they need to be able to define the id that the credential will be assigned.

## Retrieve a previously selected credentials instance

Assuming you have previously stored the `IdCredentials.getId()`, at some point in time you will want to retrieve it.

If you are working in the context of a `Run` then this is essentially quite simple:

```
StandardCredentials c = CredentialsProvider.findCredentialsById(
    Expression, !
    StandardCredentials.class, "
    run, #
    URIRequirementBuilder.fromUri(...) $
); %
```

- ! This method call will take care of evaluating credentials expressions for you.
- " You can use a more specific type if you need to.
- # You must supply the `hudson.model.Run` instance, the authentication will be determined based on the authentication that the run is using and / or the authentication of the triggering user if the triggering user selected a credential in a credentials parameter and they have permission to supply their own personal credentials.
- \$ Always supply the domain information used when displaying the original drop-down as there could be duplicate IDs between domains.
- % The credential usage will be automatically associated with the `Run` via the fingerprint API for you.

## Why are credentials expressions permissions so complex

If we have a job, "foobar", and we configure a credentials parameter on that job, we can have a number of different ways the credentials get selected:

- ¥ There is the possibility that the parameter uses a default value. The default value should be resolved in the context of the job's authentication from `Tasks.getAuthenticationOf(job)`
- ¥ If the user selects a value, they may have `Credentials/USE_ITEM`, in which case the value should be resolved in the context of the job's authentication from `Tasks.getAuthenticationOf(job)`
- ¥ If the user selects a value, they may have `Credentials/USE_OWN`, in which case the value should be resolved in the context of their own per-user credentials store

If you are working outside the context of a `Run` then you will not have to deal with the complexities of credentials expressions.

In most cases the retrieval will just be a call to one of the `CredentialsProvider.lookupCredentials(É)` wrapped within `CredentialsMatchers.firstOrNull(É, CredentialsMatchers.withId(É))`, for example:

```
StandardCredentials c = CredentialsMatchers.firstOrNull(  
    É CredentialsProvider.lookupCredentials(  
        É StandardCredentials.class, !  
        É job, !  
        É job instanceof Queue.Task !  
        É ? Tasks.getAuthenticationOf((Queue.Task)job)  
        É : ACL.SYSTEM,  
        É URIRequirementBuilder.fromUri(...) !  
    É ),  
    É CredentialsMatchers.withId(credentialsId) "  
);
```

- ! These should be the same as your call to `CredentialsProvider.listCredentials(É)/StandardListBoxModel.includeMatchingAs(É)` in order to ensure that we get the same credential instance back.
- " If you had additional `CredentialsMatcher` expressions in your call to `CredentialsProvider.listCredentials(É)/StandardListBoxModel.includeMatchingAs(É)` then you should merge them here with a `CredentialsMatchers.allOf(É)`

Once you have retrieved a non-null credentials instance, all non-secret properties can be assumed as eager-fetch immutable.

## Retrieve the secret from a specific credentials instance.

All `Secret` and `SecretBytes` properties should be assumed as lazy-fetch and (even if not explicitly declared as such) assume that they could throw either `IOException` or `InterruptedException` on first invocation.

If you access a `Secret` or `SecretBytes` property, you should ensure that the credential usage is tracked against the context object for which it was resolved (*unless you are using it in the context of form validation*).

If you need to send the credentials instance to another node (which is not recommended) then you should use `CredentialsProvider.snapshot()` and send the snapshot. Taking a snapshot is equivalent to accessing a `Secret` or `SecretBytes` property from the point of view of credentials usage tracking.

The recommended way to use a credential is through the [Authentication Tokens API](#):

```
StandardCredentials c = CredentialsMatchers.firstOrNull(!
    CredentialsProvider.listCredentials(
        StandardCredentials.class,
        job,
        job instanceof Queue.Task
        ? Tasks.getAuthenticationOf((Queue.Task)job)
        : ACL.SYSTEM,
        URIRequirementBuilder.fromUri(issueTrackerUrl)
    ),
    CredentialsMatchers.anyOf(
        CredentialsMatchers.withId(credentialsId),
        AuthenticationTokens.matcher(IssueTrackerAuthentication.class)
    )
);
IssueTrackerAuthentication auth = AuthenticationTokens.convert(
    IssueTrackerAuthentication.class,
    c
);
```

! Do not inline the call as we need to hold onto the credentials instance in order to track its usage.

" If you need a more complex mapping then the `AuthenticationTokenContext` version can be used.

# The returned authentication will be `null` if not credentials were found.

## Track usage of a credential against specific Jenkins context objects

Any time you access a credentials secret (*outside of form validation*) you are responsible to ensure that the credentials usage is tracked. If you have used `CredentialsProvider.findCredentialsById` then this obligation has been taken care of for you.

To track usage, just pass the credentials instance and the context object to the corresponding overload of `CredentialsProvider.track(context, credentials)`. If you have multiple credentials to track, you can use `CredentialsProvider.trackAll(context, credentials)`

```
StandardCredentials c = ...;
...
CredentialsProvider.track(job, c);
```

In most cases we can avoid holding object references longer than necessary by combining all these methods together:

```
IssueTrackerAuthentication auth = AuthenticationTokens.convert(
    IssueTrackerAuthentication.class,
    CredentialsProvider.track(
        job,
        CredentialsMatchers.firstOrNull(
            CredentialsProvider.listCredentials(
                StandardCredentials.class,
                job,
                job instanceof Queue.Task
                    ? Tasks.getAuthenticationOf((Queue.Task)job)
                    : ACL.SYSTEM,
                URIRequirementBuilder.fromUri(issueTrackerUrl)
            ),
            CredentialsMatchers.allOf(
                CredentialsMatchers.withId(credentialsId),
                AuthenticationTokens.matcher(IssueTrackerAuthentication.class)
            )
        )
    )
);
```