Table of Contents

Implementation Guide

This document provides the guidelines for plugin authors seeking to implementation extensions to the Credentials API.

This document is structured as follows:

- The first section is an overview of all the areas of the extension points provided by the Credentials API.
- The subsequent sections consider each of the API extension points in turn.

Overview

There are three main facets of the Credentials API that are designed for extension by plugin authors:

- Credential types themselves - if you need a new type of credential.
- Credential domains - if you need to provide users with a new way of categorising their credentials.
- Credential providers - if you are exposing a new credential store.

The expectation is that most plugin authors requiring to implement a Credentials API extension point will be implementing new credential types.

When implementing a service specific credential type, it may be useful to provide a credential domain specific to that service. An other use-case for implementing a custom credential domains would be if the credentials are being used over a specific protocol.

Implementing a credentials provider should be reserved to integration of Jenkins with external credentials stores although there may be some exotic use-cases around exposing a credential from one authentication scope and context into another authentication scope and context.

Implementing a new Credentials type

The first question you should ask yourself is *"Should I implement a new credentials type?"*

There are a number of existing credentials types, when should you reuse an existing credential and when should you create a new one. The criteria for making this decision is centred around the possibility for re-use of those credentials in another context.

First rule for implementing a new credential type

When there is no possibility of the credentials being used in another context outside of your plugin, implement a new type.

If the credentials may be reused in a different context, then the credentials type should be available to both contexts.

- If I am using a username and password to connect to a service, we have the choice of using the `StandardUsernamePasswordCredentials` interface or creating our own type.
  - *If the service provides a SSO or LDAP integration*, then we can expect that some users will have enabled the integration and thus their corporate username and password will be the same as the username and password they use when connecting to the service. *This means that it is reasonable to expect the correct password to be changed in sync across all these services*. If we implemented a custom credential type, when the 90 day reset of their LDAP password comes around, the user will have to race to update two or more credentials in Jenkins to prevent a lockout of LDAP. *Therefore we reuse the `StandardUsernamePasswordCredentials` credentials type*.
  - *If the service does not provide any SSO or LDAP integration and does not expose such to other services*, then we can expect that the password will be changed without correlation to any other service's passwords. This could be a case where it makes sense to implement a custom credentials type, but we would need to be certain that the passwords would never be used against other services.

    Yes the foolish user may actually change all their passwords to the same password across all these different services, but they will be manually logging into each service one at a time and changing the passwords as they go.

  - *If the service is actually using something that only **looks like** a username and password*, then there may be utility in implementing a custom credentials type. The authentication might be by username and application token or by application token and application secret. These things look like username/password from a certain perspective, but *there is no expectation that they would be valid against any other service*. In fact there may even be the possibility of providing a Jenkins specific integration to obtain / refresh these credentials, for example we could trigger a web flow that would allow the user to connect to the service and request a token/secret that could then be fed back into Jenkins removing the need for the user to copy & paste. *This is a perfect case for implementing a custom credential type*.
- If I am using a token to authenticate with the service, we have the choice of using something like `StringCredentials` our creating our own type of credential.
  - *If the service uses some form of OAuth or OAuth-like authentication*, then:
    - the token will never be valid against any other service.
    - we probably want to provide a custom web flow to make it easier for the user to get the token to Jenkins without resorting to copy&paste

    So, *we implement a service specific credential type*.
  - *If the token is going to be used across multiple Jenkins plugins*, then we need to seek to find the common denominator credentials type that is valid for all these plugins. It may make sense for our plugin to be the provider of the common denominator credential type, say if we can be certain that any other Jenkins plugins will use our plugin as a dependency. Or it may be that some other plugin has already won the battle, in which case add that plugin as a dependency. Or it may not make sense for your plugin to depend on this other plugin, in

which case you should seek to extract the common credential type into a third plugin that both plugins can then depend on.

Only in the very last resort should we use `StringCredentials` as that using that credential type would throw away all contextual information from the credential type and therefore would force users to use credentials domains to restore contextual information.

It may indeed be correct to use `StringCredentials` as the credentials type, but you should think long and hard and be sure you have a valid reason to require the generic type.

`StandardUsernamePasswordCredentials` has it easier as the potential for SSO or LDAP integration will typically drive the choice to the common credential type.

The `StringCredentials` type use case is for injecting secrets into builds, not for connecting to third party services using a plugin native integration.

So we will assume, if you have got this far, that you have convinced yourself of the need to implement a custom credentials type.

At this point, the lazy developer will do something like:

Do not do what Johnny Dont does

```
public class DoNotDoThis extends BaseStandardCredentials {
  ...
  public String getApplicationToken() {...}
  public Secret getApplicationSecret() {...}
  ...
  @Extension
  public class DescriptorImpl extends BaseStandardCredentialsDescriptor { ... }
}
```

The issue here is that by implementing the custom credential type as a `class` we prevent any `CredentialsProvider` extensions that store credentials externally from using the `java.lang.reflect.Proxy` mechanism to lazily fetch the secrets from an external store.

Second rule for implementing a new credential type

Define and access the credential through an interface.

Provide users with a default implementation class.

Do not assume the credentials will be using your implementation class.

So we should have:

Use an interface

```
public interface DoThis extends StandardCredentials {
  String getApplicationToken();
  Secret getApplicationSecret() throws IOException, InterruptedException;
}

public class DoThisImpl extends BaseStandardCredentials implements DoThis {
  ...
  @Override
  public String getApplicationToken() {...}
  @Override
  public Secret getApplicationSecret() {...}
  ...
  @Extension
  public class DescriptorImpl extends BaseStandardCredentialsDescriptor { ... }
}
```

You should note that we have used `Secret` to define the secret. And furthermore, we have added the `throws IOException, InterruptedException` to the getter! Again this is so that a new `CredentialsProvider` can lazily fetch the secret.

Third rule for implementing a new credential type

The credentials interface shall follow the JavaBeans conventions.

The getters that retrieve the actual secrets shall use either `Secret` or `SecretBytes` as the return type and shall throw `IOException` and `InterruptedException`

It was a mistake in the original API to not have the throws on `PasswordCredentials.getPassword()`, do not repeat that mistake!

Walk-through

At this point it is probably easier to provide a walk-through for implementing a custom credentials type.

We start with the scenario:

Scenario:

We are implementing a Jenkins plugin to integrate with Acme Corp’s on-line ordering service.

Our plugin will scan the blueprints from the workspace and submit a purchase order with Acme Corp.

This enables users like Wile E. Coyote to run simulations of the blueprints and only submit orders for tested designs.

Acme Corp’s on-line services are protected by Two-Factor Authentication, so username password authentication will not work, instead the user creates an application token for each authorized application and that token can be used in place of their password. Users are encouraged to use a different application token for each application.

So, we start by seeing if we need to implement a custom credentials type:

- The credentials we will be using look like `UsernamePasswordCredentials` in that there is a username and a secret that acts like a password.
- The credentials are actually not a username and password because we have the explicit expectation that the passwords we will be provided with will only ever be valid against Acme Corp’s on-line services.

So we are going to create a custom credentials type.

We start with the interface:

- We need to extend `StandardUsernameCredentials` because the username is actually the user’s username. If Wile E. Coyote and Roadrunner are sharing the same Jenkins server, they will expect to see their username as the identifier to use when selecting the credentials to use.
- We do not extend `PasswordCredentials` because this is not the password but actually an Acme Corp specific application token. If there were use cases where this application token needs to be used by other plugins and those other plugins are somewhat more generic then we might consider extending `StandardUsernamePasswordCredentials` but it is hard to see that need and use of the Authentication Tokens API plugin would probably be a better solution for those use cases.

AcmeApplicationTokenCredentials.java

```
@NameWith( ①
  value = AcmeApplicationTokenCredentials.NameProvider.class,
  priority = 32 ②
)
public interface AcmeApplicationTokenCredentials extends StandardUsernameCredentials {
  Secret getApplicationToken() throws IOException, InterruptedException; ③
  class NameProvider extends CredentialsNameProvider<AcmeApplicationTokenCredentials>
{
    @NonNull
    @Override
    public String getName(@NonNull AcmeApplicationTokenCredentials c) {
      String description = Util.fixEmptyAndTrim(c.getDescription());
      return c.getUsername()
          + "/*acme*" ④
          + (description != null ? " (" + description + ")" : "");
    }
  }
}
```

① We want to override the default naming strategy from the parent interface. (This may not always be required, for example if one of the name provider from the implemented interfaces provides a good enough name)

② We need to specify a priority that is higher than that of the parent interface.

③ We ensure that the getter follows JavaBeans conventions and uses `Secret` or `SecretBytes` as the return type. Because this getter returns a secret, we also declare `throws IOException, InterruptedException` so that consumers can take reasonable defenses against `java.lang.reflect.Proxy` implementations that make remote calls to obtain the secret.

④ We want the name of the credential to indicate that this is an Acme Corp application token, so rather than `wecoyote/` which is what `StandardUsernamePasswordCredentials` would generate or `wecoyote` which we would get from `StandardUsernameCredentials`, we generate the name `wecoyote/*acme*`

Next we need to provide users with an implementation of this credentials type so that they can create them in the UI.

AcmeApplicationTokenCredentialsImpl.java

```
public class AcmeApplicationTokenCredentialsImpl
    extends BaseStandardCredentials ①
    implements AcmeApplicationTokenCredentials { ②
  private final String username;
  private final Secret applicationToken;
  @DataBoundConstructor ③
  public AcmeApplicationTokenCredentialsImpl(
      @CheckForNull CredentialsScope scope, ④
      @CheckForNull String id, ④
      @NonNull String username, ⑤
      @NonNull String applicationToken, ⑥
      @CheckForNull String description) { ④
    super(scope, id, description);
    this.username = username;
    this.applicationToken = Secret.fromString(applicationToken); ⑦
  }
  /*
  public AcmeApplicationTokenCredentialsImpl( ⑧
  @CheckForNull String id,
  @NonNull String username,
  @NonNull Secret applicationToken) {
    super(null, id, null);
    this.username = username;
    this.applicationToken = applicationToken;
  }
  */
  @NonNull
  @Override
  public String getUsername() { return username; }
  @NonNull
  @Override
  public Secret getApplicationToken() { ⑨
    return applicationToken;
  }
  @Extension
  public static class DescriptorImpl extends BaseStandardCredentialsDescriptor {
    @Override
    public String getDisplayName() {
        return "Acme Corp Application Token"; ⑩
    }
    @Override
    public String getIconClassName() {
        return "icon-acmecorp-credentials"; ⑪
    }
    static { ⑫
      IconSet.icons.addIcon(new Icon(
          "icon-acmecorp-credentials icon-sm",
          "acmecorp-order-step/images/16x16/credentials.png",
          Icon.ICON_SMALL_STYLE,
          IconType.PLUGIN
      ));
      IconSet.icons.addIcon(new Icon(
          "icon-acmecorp-credentials icon-md",
          "acmecorp-order-step/images/24x24/credentials.png",
          Icon.ICON_SMALL_STYLE,
          IconType.PLUGIN
      ));
      IconSet.icons.addIcon(new Icon(
          "icon-acmecorp-credentials icon-lg",
          "acmecorp-order-step/images/32x32/credentials.png",
          Icon.ICON_SMALL_STYLE,
          IconType.PLUGIN
      ));
      IconSet.icons.addIcon(new Icon(
          "icon-acmecorp-credentials icon-xlg",
          "acmecorp-order-step/images/48x48/credentials.png",
          Icon.ICON_SMALL_STYLE,
          IconType.PLUGIN
      ));
    }
  }
}
```

① Always extend from the most specific `BaseStandardCredentials` subclass available.

② Do not forget to implement your actual interface.

③ We need a `@DataBoundConstructor` or users will not be able to create the credentials.

④ Pass-through the `scope`, `id` and `description`, because this is a `@DataBoundConstructor` these names must match.

⑤ The username field must be called `username` because we are inheriting the contract from `UsernameCredentials`.

⑥ Use a `String` type in the `@DataBoundConstructor` for the `Secret` parameter (same would apply for `SecretBytes`).

⑦ We need to convert from the `String` from the submitted web form into the `Secret`. With this approach we can have the test cases just provide a raw unencrypted string value to the constructor and Jenkins will encrypt it for us.

⑧ If you need a simplified programmatic constructor - for example when converting credentials or for use from plugin unit / integration testing, by all means add them, the only minimum requirement is that there be one (and only one) `@DataBoundConstructor`.

⑨ We can remove the exceptions from our implementation as we will not throw these exceptions from the default implementation. The exceptions are on the interface to assist consumers.

⑩ Ideally we would use Jenkins' I18N support and put this string in a `Messages.properties` resource, just showing the text here to keep the example more self-contained.

⑪ While not strictly required to provide a custom icon, it is recommended to provide one to give users additional visual queues.

⑫ You need to register the custom icon somewhere in a class that is guaranteed to be loaded before the icon class name is referenced. You can either do this in a `hudson.Plugin` or in a non-optional `Descriptor` or `@Extension`.

Finally we need the configuration Stapler facet fragment / tear-off. The example here uses Jelly, but any of the supported tear-off frameworks can be used.

credentials.jelly

```
<j:jelly xmlns:j="jelly:core" xmlns:f="/lib/form" xmlns:st="jelly:stapler">
  <st:include page="id-and-description" class="${descriptor.clazz}"/> ①
  <f:entry title="${%Username}" field="username">
    <f:textbox/>
  </f:entry>
  <f:entry title="${%Application Token}" field="applicationToken">
    <f:password/> ②
  </f:entry>
</j:jelly>
```

① Because we inherit from `BaseStandardCredentials` we need to include the `id-and-description` tear-off. (At least until JENKINS-45540 has been resolved in the base version of the Credentials API plugin that your plugin depends on.)

② We use a `<f:password/>` input to hold the application token. This is the most basic implementation of a UI. A more complicated UI (which would have an improved UX for users) could use an invisible entry to round-trip the application token and display a "Authenticate / Reauthenticate" button that would open a new browser window, initiate a request for generating an application token and then pass the resulting token back into the hidden field once the request was fulfilled.

Our plugin should look something like this:

```
pom.xml
src/
    main/
        java/
            org/
                jenkinsci/
                    plugins/
                        acmecorp/
                            credentials/
                                AcmeApplicationTokenCredentials.java
                                AcmeApplicationTokenCredentialsImpl.java
        resources/
            org/
                jenkinsci/
                    plugins/
                        acmecorp/
                            credentials/
                                AcmeApplicationTokenCredentialsImpl/
                                    credentials.jelly
                                    help.html ①
                                    help-username.html ①
                                    help-applicationToken.html ①
                                Messages.properties ②
        webapp/
            images/
                16x16/
                    credentials.png
                24x24/
                    credentials.png
                32x32/
                    credentials.png
                48x48/
                    credentials.png
```

① You will want to provide in-line help for the user.

② You should use Jenkins I18N facilities to localize strings such as the return value from `DescriptorImpl.getDisplayName()`

Additional concerns

There may be some additional concerns that you need to address:

- If your credential type keeps secrets external from the CredentialsProvider, for example the `SSHUserPrivateKey` default implementation can store the key either within the CredentialsProvider or on disk. If we need to transport the credential to an agent, the on-disk file will not be available from the agent. The solution to this issue is to implement a `CredentialsSnapshotTaker` extension, e.g. `BasicSSHUserPrivateKey.CredentialsSnapshotTakerImpl`
- If you need to migrate from one credential type to another **and** there are existing plugins that are depending on your legacy type, you may want to implement a `CredentialsResolver` to

instantiate the legacy type from instances of the new type. See `BasicSSHUserPassword` for an example of how to perform this type of migration. Key points to note:

- The class has a `@ResolveWith` annotation to trigger the resolution process.
- There is a readResolve() that returns the new credential type.
- There is no `DescriptorImpl` because this should not be a user visible type. Also there is no `@DataBoundConstructor`.
- The resolver instantiates an equivalent legacy credential instance from the new credential type

Implementing new Domain specification / requirement types

When a Jenkins instance has lots of credentials, it can become confusing for users to determine which credentials are supposed to be used against each different service.

Where a service requires a specific type of credential, the consuming plugin can just limit the available options to those that are of the required type.

However, often times the type of credential can be somewhat generic, for example `StandardUsernamePasswordCredentials` or `SSHUserPrivateKey`. The user needs some way to indicate that credentials X are for use with service A and credentials Y are for use with service B.

The Credentials API plugin provides for this categorization through the concept of Credentials Domains.

The user defines different credentials domains and puts the appropriate credentials into those domains. The consuming plugin builds up the domain requirements, and then the Credentials API returns a list of all credentials from domains having compatible specifications.

> In general, domain specifications are lenient matching, in other words, if a request does not have the matching requirements then the specification is deemed to have been met.
>
> This does not have to be the case, but if you choose to define a strict matching domain specification you should be aware that this will:
>
> - force users to use credentials domains as credentials in the global domain will never match; and
> - prevent the use of credentials providers that do not support domains.

Technically, you do not need to implement a new `DomainRequirement` class when providing a new `DomainSpecification` as the specification could test for pre-existing requirements.

If you implement a new `DomainRequirement` class, in general you will need to implement a `DomainSpecification` that tests for this requirement.

You should provide some utility methods to assist consuming plugins to instantiate the

DomainRequirement collection for any requests that they make, see URIRequirementBuilder

Walk-through

We will continue the previous scenario for creating a custom credential.

In this case, as the new credential type is specific to Acme Corp we want to allow users to segregate the credentials for use against the production Acme Corp service from the credentials for use against their test server.

While users could use hostname specifications to segregate credentials between test.acme.example.com and prod.acme.example.com, in this case we are going to provide an simplified user experience.

We start with a DomainRequirement:

AcmeRequirement.java

```java
public class AcmeRequirement extends DomainRequirement {
  private static final long serialVersionUID = 1L; ①
  private final boolean test;
  public AcmeRequirement(boolean test) {
    this.test = test;
  }
  public boolean isTest() { return test; }
}
```

① DomainRequirement is Serializable so ensure you define a serialVersionUID. Convention is to start newly created classes off with the value 1L. If you forgot and have to add the value after you released your plugin, you will need to calculate the effective value of the version you released.

Now we need to create a DomainSpecification to test against this requirement:

AcmeSpecification.java

```
public class AcmeSpecification extends DomainSpecification {
  private final boolean test;
  @DataBoundConstructor ①
  public AcmeSpecification(boolean test) {
    this.test = test;
  }
  public boolean isTest() { return test; }
  @NonNull
  @Override
  public Result test(@NonNull DomainRequirement r) {
    if (r instanceof AcmeRequirement) {
      if (this.test == ((AcmeRequirement)r).isTest()) {
        return Result.POSITIVE; ②
      } else {
        return Result.NEGATIVE; ③
      }
    } else if (r instanceof HostnameRequirement) {
      String hostname = ((HostnameRequirement) r).getHostname();
      if (test) {
        if ("test.acme.example.com".equalsIgnoreCase(hostname)) {
          return Result.PARTIAL; ④
        } else {
          return Result.NEGATIVE; ⑤
        }
      } else {
        if ("prod.acme.example.com".equalsIgnoreCase(hostname)) {
          return Result.PARTIAL; ④
        } else {
          return Result.NEGATIVE; ⑤
        }
      }
    }
    return Result.UNKNOWN; ⑥
  }
  @Extension
  public static class DescriptorImpl extends DomainSpecificationDescriptor {
      @Override
      public String getDisplayName() {
          return "Acme Corp On-line Store"; ⑦
      }
  }
}
```

① We need a `@DataBoundConstructor` or users will not be able to create the specification.

② If we see an `AcmeRequirement` and it matches we know the specification has matched exactly, no other specification is expected to match an `AcmeRequirement` so we return the short-circuit result.

③ If we see an `AcmeRequirement` and it doesn’t match, we know this is a miss.

④ If we see a `HostnameRequirement` and it matches, that means that we do not have a miss. We return the `Result.PARTIAL` to indicate that the remaining specifications should be checked against this requirement.

⑤ If we see a `HostnameRequirement` and it doesn't match, we know this is a miss.

⑥ All other requirements are unknown, so we signal checking the other specifications against the requirement.

⑦ Ideally we would use Jenkins' I18N support and put this string in a `Messages.properties` resource, just showing the text here to keep the example more self-contained.

We will need a `config` facet fragment:

`config.jelly`

```
<j:jelly xmlns:j="jelly:core" xmlns:f="/lib/form" xmlns:st="jelly:stapler">
  <f:entry title="${%Test server}" field="test">
    <f:checkbox/>
  </f:entry>
</j:jelly>
```

Finally, we will create a builder to assist consumer plugins:

`AcmeRequirementBuilder.java`

```
public static class AcmeRequirementBuilder {
  private boolean test;
  private AcmeRequirementBuilder() {} ①
  @NonNull
  public AcmeRequirementBuilder create() { ①
    return new AcmeRequirementBuilder();
  }
  @NonNull
  public AcmeRequirementBuilder withTestServer(boolean test) { ①
    this.test = test;
    return this;
  }
  @NonNull
  public List<DomainRequirement> build() {
    List<DomainRequirement> result = new ArrayList<>();
    result.add(new AcmeRequirement(test)); ②
    result.addAll(URIRequirementBuilder.create() ③
        .withUri(test
            ? "https://test.acme.example.com/"
            : "https://prod.acme.example.com/")
        .build();
    );
    return result;
  }
}
```

① This is a builder so we follow the builder pattern.

② Add in our `AcmeRequirement`

③ Because we know this is a requirement against Acme Corp, we can limit the other standard requirements also. This allows the user to create a domain using the standard `HostnameSpecification` and have that work for segregation.

Our plugin should look something like this:

```
pom.xml
src/
    main/
        java/
            org/
                jenkinsci/
                    plugins/
                        acmecorp/
                            domains/
                                AcmeRequirement.java
                                AcmeRequirementBuilder.java
                                AcmeRequirementSpecification.java
        resources/
            org/
                jenkinsci/
                    plugins/
                        acmecorp/
                            domains/
                                AcmeRequirementSpecification/
                                    config.jelly
                                    help.html ①
                                    help-test.html ①
                                Messages.properties ②
```

① You will want to provide in-line help for the user.

② You should use Jenkins I18N facilities to localize strings such as the return value from `DescriptorImpl.getDisplayName()`

Implementing a new CredentialsProvider

The `CredentialsProvider` extension point is perhaps one of the more complicated extension points:

- Where the backing store is remote from Jenkins then:
  - potentially has to be able to either instantiate `java.lang.reflect.Proxy` implementations for credentials, or create on-demand implementation classes using ASM (or similar).
  - potentially has to deal with parsing the `CredentialsMatcher` query language in order to minimize transfer of information over the network.
  - may need to store Jenkins specific state in Jenkins in order to provide credentials domain support.

- Where the backing store is local to Jenkins but contextual to a specific Jenkins model object and not covered by the three existing credentials providers: System, User and Folder, then replication of that code will likely be required.
- Where the backing store is another credentials provider in Jenkins, is probably the simplest case.

Walk-through

Contributors welcome

This walk-through is incomplete and needs the following additions:

- [ ] Provide links to a reference implementation of an external credentials provider.
- [ ] Provide links to a reference implementation of a Jenkins native credentials provider that exposes credentials in a Jenkins context other than the SystemCredentialsProvider, UserCredentialsProvider or FolderCredentialsProvider.

  These existing examples are probably not good as reference examples as they have distractions to do with the evolution of their implementations with the API.

  A good reference implementation would be clean of such distractions.
- [ ] Provide links to some other implementations of credentials providers for other use cases.
- [ ] Provide some details on how the Credentials Query Language can be used to limit querying credentials from the remote service

We will need a new scenario:

Scenario:

Acme Corp has written a secure credentials storage service and wants to write a plugin to integrate that service with Jenkins. The service offers six different higher level functions:

- Search - returns a list of the non-secret portions of those credentials that have properties matching any given query
- Create - adds a credential with the supplied properties
- Update - modifies the properties associated with a credential
- Retrieve - returns the non-secret properties of a specified credential
- Retrieve Secret - returns the secret value of a specified credential
- Delete - removes a specified credential

There are a number of different ways we can integrate this service with Jenkins:

- Read-only, explicitly exposed - in this integration we only use the "Retrieve" and "Retrieve Secret" functionality from within Jenkins.

  The Jenkins admin will define which credentials from the external system are exposed to Jenkins by providing Jenkins with the IDs of the credentials to expose in each Jenkins context.

  At the system level, the Jenkins admin will select an Acme credentials property to be used to map to the Jenkins credential type.

  The Jenkins admin will then define how the Acme credentials properties of each type get mapped to that type's Jenkins credentials properties.

- Read-only, implicitly exposed - in this integration we add the "Search" functionality.

  The Jenkins admin will define a property (or properties) that can be used to determine the Jenkins context that the credential should be exposed in.

- Read-Write, explicitly exposed - in this integration we use the CRUD functionality but not the search functionality.

  Jenkins is using the Acme credentials service as the backing store, but it is not intended that Jenkins be able to discover existing credentials.

- Read-Write, implicitly exposed - this integration essentially makes Jenkins a rich client for the credentials service.

  Jenkins will only manage those credentials that have the correct properties to associate them with the corresponding Jenkins contexts, but if a credential is created externally with those properties it will be automatically available to Jenkins.

We will also need to consider how the availability of the Acme service affects Jenkins.

- If the service is off-line, how should we respond to requests for:
  - Listing credentials available within a specific Jenkins context
  - Querying non-secret properties of credentials
  - Retrieving the secret of the credentials
- If the service is on-line but over-loaded, how should we respond to requests for:
  - Listing credentials available within a specific Jenkins context
  - Querying non-secret properties of credentials
  - Retrieving the secret of the credentials

Due to how Jenkins works, there is actually only one way to solve the above choices without risking breakage of Job configuration when users are modifying jobs.

- Retrieval of the secret of a credential **must** always be live. If the service is off-line or if the credential has been removed from the backing service, throw an `IOException`. If the response takes too long (ideally let the admin define a timeout), throw an `InterruptedException`.

It is critical that the secret only be retrieved at the point in time where it is required.

This enables the backing service to identify and track those credentials that are actually being used.

Additionally, some users may have security requirements that mandate credentials not be stored outside of their secure "vault" so Jenkins **must** not make any attempt at caching the secret portion of a credential.

- Querying non-secret properties of credentials should use short time-window cache with background update. If the cache is stale, block for a short time before falling back to returning the stale values. The Jenkins admin should be able to configure the various timeouts.

Consuming plugins will repeatedly query the non-secret properties of credentials.

While it would be ideal to have all these properties live, in practical terms doing so would negatively visibly impact the performance of the Jenkins UI as well as the non-visible impact on the overall Jenkins performance.

For this reason we want to use a cache with a short time window (admin configurable, default to 5 minutes). Thus repeatedly accessing properties when the cache is within this short time window will respond immediately using the cached value. An access near the end of the cache time window (e.g. in the last 20% of the window) should initiate a background update (unless one is already in flight) An access after the end of the cache time window should block for no more than 1-2 seconds before returning the cached value.

It is an implementation choice whether to repeatedly block access after the cache time window expires if no response is forthcoming. The recommendation is that for any given credential:

- There should only ever be one update request in flight at any point in time.
- If the update request fails, the failure should be cached for a period of time (in effect making the previous cached value "live" again, but perhaps for a shorter time period)

- Listing credentials available within a specific Jenkins context should use a live query with short timeout. If the timeout expires before the response arrives, then attempt to reconstruct the response from the cache.

Listing credentials operations are normally restricted to the population of credentials selection drop-down lists.

Such requests are AJAX requests, so we have the option to block without affecting the rest of the Jenkins UI.

Blocking for more than between 5 and 10 seconds, however, will cause user frustration, thus for this type of request we try to serve the response live and fall-back to the cache if the live response takes too long.

These different caching concerns are addresses at different points in the credentials API:

- A `CredentialsProvider` implementation is expected to return their own implementation of each `Credentials` subclass that they support. When instantiated, the non-secret properties should be populated, but the secret properties will be deferred until accessed.

  Ideally, this should be achieved using either `java.lang.reflect.Proxy` or byte-code generation, something like:

```
public static class CredentialProxy implements Serializable, InvocationHandler {
  private final Map<String,Object> properties;
  private final String secretName;
  private final transient AcmeConnection connection;
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
    if (args != null || args.length > 0) {
      return null; ①
    }
    String n = method.getName();
    if (n.startsWith("get")) {
      n = n.substring(3,4).toLowerCase() + n.substring(4);
    } else if (n.startsWith("is")) {
      n = n.substring(2,3).toLowerCase() + n.substring(3);
    } else {
      return null; ①
    }
    if (secretName.equals(n)) {
      if (connection != null) {
        return connection.getSecret(...);
      } else {
        throw new IOException("No connection"); ②
      }
    } else {
      return properties.get(n);
    }
  }
}
```

  ① Should be unreachable code, as you should refuse to create a proxy for any `Credentials` interface that has anything other than simple JavaBean style getters. If you have implemented that check before creating the proxy then you could be more explicit and throw an exception.

② Any consumer plugin that is transferring a credential to another JVM is supposed to call `CredentialsProvider.snapshot(credential)` and send the return value. The `CredentialsSnapshotTaker` is supposed to fetch the secret as part of the snapshotting, so a proper consumer will never be at risk of this `IOException`.

- The `CredentialsProvider.getCredentials(…)` methods should instantiate the proxies, so these methods will operate from the cache while initiate background refresh. Where the cache is a miss or where the cache is stale, a short term block is acceptable.
- The `CredentialsProvider.getCredentialIds(…)` methods are used to list credentials for drop-down list population, so these methods should use a live request with a fall-back to the cache where the live request takes too long.

When implementing a credentials non-secret properties cache, ideally the cache should survive Jenkins restarts so that users do not end up breaking job configuration if Jenkins starts up while the backing service is off-line.

The main work in an implementation will be the mapping to `CredentialStore` instances.

- Any "explicitly exposed" style implementation will have `CredentialsStore` instance for each context that persists the IDs of the credentials to be exposed and the credentials domains with which those credentials are to be associated.
- A "read-only, implicitly exposed" style implementation can semi-dynamically create `CredentialsStore` instances for each context.

  Technically, the "read-only, implicitly exposed" style credentials provider implementation does not need to interact with the `CredentialsStore` portion of the API as it can expose credentials directly using just the `CredentialsProvider.getCredentials(…)` and `CredentialsProvider.getCredentialIds(…)`, however, implementing the `CredentialsStore` contract is required in order for the credentials to be visible to users via the Credentials side action on the different Jenkins context objects.

- A "read-write, implicitly exposed" style implementation will need to semi-dynamically create `CredentialsStore` instances for each context in order to integrate with the Jenkins credentials management UI.