

# Quarkus - Using a Credentials Provider

Interacting with a datastore typically implies first connecting using credentials. Those credentials will allow the client to be identified, authenticated and eventually authorized. Username/password based authentication is very common, but that is not by any means the only one. Such credentials information may appear in the application configuration, but it is becoming increasingly popular to store this type of sensitive information in secure stores, such as HashiCorp Vault, Azure Key Vault or the AWS Secrets Manager to name just a few.

To bridge datastores that consume credentials, which can take different forms, and secure stores that provide those credentials, Quarkus introduces an intermediate abstraction called **Credentials Provider**, that some extensions may support to consume credentials (e.g. **agroal**), and some others may implement to produce credentials (e.g. **vault**).

This Service Programming Interface (SPI) may also be used by implementers that want to support custom providers not yet implemented in Quarkus (e.g. Azure Key Vault).

Currently, the **Credentials Provider** interface is implemented by the **vault** extension, and is supported by the following credentials consumer extensions:

- **agroal**
- **reactive-db2-client**
- **reactive-mysql-client**
- **reactive-pg-client**

All extensions that rely on username/password authentication also allow setting configuration properties in the **application.properties** as an alternative. But the **Credentials Provider** is the only option if credentials are generated (e.g. **Vault Dynamic DB Credentials**) or if a custom credentials provider is required.

This guide will show how to use the **Credentials Provider** provided in the **vault** extension, then we will look at implementing a custom **Credentials Provider**, and finally we will talk about additional considerations regarding implementing a **Credentials Provider** in a new extension.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

# Prerequisites

To complete this guide, you need:

- roughly 20 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Docker installed

## Vault Credentials Provider

To configure a `Vault Credentials Provider` you need to provide the following properties:

```
quarkus.vault.credentials-provider.<name>.<property>=<value>
```

The `<name>` will be used in the consumer to refer to this provider. The `<property>` and `<value>` fields are specific to the `Vault Credentials Provider`. For complete details, please refer to the [Vault Datasource guide](#).

For instance:

```
quarkus.vault.credentials-provider.mydatabase.kv-path=myapps/vault-quickstart/db
```

Once defined, the `mydatabase` provider can be used in any extension that supports the `Credentials Provider` interface. For instance in `agroal`:

```
# configure your datasource
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = sarah
quarkus.datasource.credentials-provider = mydatabase
quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/mydatabase
```

Note that `quarkus.datasource.username` is the original `agroal` property, whereas the `password` property is not included because the value will come from the `mydatabase` credentials provider we just defined. An alternative is to define both username and password in Vault and drop the `quarkus.datasource.username` property from configuration. All consuming extensions do support the ability to fetch both the username and password from the provider, or just the password.

# Custom Credentials Provider

Implementing a custom credentials provider is the only option when a vault product is not yet supported in Quarkus, or if credentials need to be retrieved from a custom store.

The only interface to implement is:

```
public interface CredentialsProvider {

    String USER_PROPERTY_NAME = "user";
    String PASSWORD_PROPERTY_NAME = "password";

    Map<String, String> getCredentials(String
credentialsProviderName);

}
```

`USER_PROPERTY_NAME` and `PASSWORD_PROPERTY_NAME` are standard properties that should be recognized by any consuming extension that support username/password based authentication.

It is required that implementations be valid `@ApplicationScoped` CDI beans.

Here is a simple example:

```
@ApplicationScoped
@Unremovable
public class MyCredentialsProvider implements CredentialsProvider {

    @Override
    public Map<String, String> getCredentials(String
credentialsProviderName) {

        Map<String, String> properties = new HashMap<>();
        properties.put(USER_PROPERTY_NAME, "hibernate_orm_test");
        properties.put(PASSWORD_PROPERTY_NAME,
"hibernate_orm_test");
        return properties;
    }

}
```

Note that we decided here to return both the username and the password.

This provider may be used in a datasource definition like this:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.credentials-provider=custom
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5431/hibernate_orm_test
```

It is also possible to pass configuration properties to the provider using standard MicroProfile Config injection:

```
custom.foo=bar
```

And in the provider implementation:

```
@Inject
Config config;

@Override
public Map<String, String> getCredentials(String
credentialsProviderName) {

    System.out.println("MyCredentialsProvider called with foo=" +
config.getValue(credentialsProviderName + ".foo", String.class));
    ...
}
```

## New Credentials Provider extension

When creating a custom credentials provider in a new extension, there are a few additional considerations.

First, you need to name it to avoid collisions in case multiple credentials providers are available in the project:

```
@ApplicationScoped
@Unremovable
@Named("my-credentials-provider")
public class MyCredentialsProvider implements CredentialsProvider {
```

It is the responsibility of the consumer to allow a `credentials-provider-name` property:

```
quarkus.datasource.credentials-provider = custom
quarkus.datasource.credentials-provider-name = my-credentials-provider
```

The extension should allow runtime config, such as the `CredentialsProviderConfig` from the `vault` extension to configure any custom property in the provider. For an AWS Secrets Manager extension, this could be:

- `region`
- `credentials-type`
- `secrets-id`

Note also that some consumers such as `agroal` will add to their connection configuration any properties returned by the credentials provider, not just the username and password. So when you design the new credentials provider limit the properties to what would be understood by consumers, or provide appropriate configuration options to support different modes.