

Quarkus - Kubernetes Client

Quarkus includes the `kubernetes-client` extension which enables the use of the [Fabric8 Kubernetes Client](#) in native mode while also making it easier to work with.

Having a Kubernetes Client extension in Quarkus is very useful in order to unlock the power of Kubernetes Operators. Kubernetes Operators are quickly emerging as a new class of Cloud Native applications. These applications essentially watch the Kubernetes API and react to changes on various resources and can be used to manage the lifecycle of all kinds of complex systems like databases, messaging systems and much much more. Being able to write such operators in Java with the very low footprint that native images provide is a great match.

Configuration

Once you have your Quarkus project configured you can add the `kubernetes-client` extension to your project by running the following command in your project base directory.

```
./mvnw quarkus:add-extension -Dextensions="kubernetes-client"
```

This will add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-kubernetes-client</artifactId>
</dependency>
```

Usage

Quarkus configures a Bean of type `KubernetesClient` which can be injected into application code using the well known CDI methods. This client can be configured using various properties as can be seen in the following example:

```
quarkus.kubernetes-client.trust-certs=false
quarkus.kubernetes-client.namespace=default
```

Note that the full list of properties is available in the [KubernetesClientBuildConfig](#) class.

Overriding

The extension also allows application code to override either of `io.fabric8.kubernetes.client.Config`

`io.fabric8.kubernetes.client.KubernetesClient` which are normally provided by the extension by simply declaring custom versions of those beans.

An example of this can be seen in the following snippet:

```
@ApplicationScoped
public class KubernetesClientProducer {

    @Produces
    public KubernetesClient kubernetesClient() {
        // here you would create a custom client
        return new DefaultKubernetesClient();
    }
}
```

Testing

To make testing against a mock Kubernetes API extremely simple, Quarkus provides the `KubernetesMockServerTestResource` which automatically launches a mock of the Kubernetes API server and sets the proper environment variables needed so that the Kubernetes Client configures itself to use said mock. Tests can inject the mock and set it up in any way necessary for the particular testing using the `@MockServer` annotation.

Let's assume we have a REST endpoint defined like so:

```
@Path("/pod")
public class Pods {

    private final KubernetesClient kubernetesClient;

    public Pods(KubernetesClient kubernetesClient) {
        this.kubernetesClient = kubernetesClient;
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{namespace}")
    public List<Pod> pods(@PathParam("namespace") String namespace)
    {
        return
        kubernetesClient.pods().inNamespace(namespace).list().getItems();
    }
}
```

We could write a test for this endpoint very easily like so:

```

@QuarkusTestResource(KubernetesMockServerTestResource.class)
@QuarkusTest
public class KubernetesClientTest {

    @MockServer
    KubernetesMockServer mockServer;

    @BeforeEach
    public void before() {
        final Pod pod1 = new
PodBuilder().withNewMetadata().withName("pod1").withNamespace("test
").and().build();
        final Pod pod2 = new
PodBuilder().withNewMetadata().withName("pod2").withNamespace("test
").and().build();

mockServer.expect().get().withPath("/api/v1/namespaces/test/pods")
                .andReturn(200,
                        new
PodListBuilder().withNewMetadata().withResourceVersion("1").endMeta
data().withItems(pod1, pod2)
                                .build())
                .always();
    }

    @Test
    public void testInteractionWithAPIServer() {
        RestAssured.when().get("/pod/test").then()
                .body("size()", is(2));
    }
}

```

Note that to take advantage of these features, the `quarkus-test-kubernetes-client` dependency needs to be added, for example like so:

```

<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-kubernetes-client</artifactId>
  <scope>test</scope>
</dependency>

```

Note on implementing the Watcher interface

Due to the restrictions imposed by GraalVM, extra care needs to be taken when implementing a `io.fabric8.kubernetes.client.Watcher` if the application is intended to work in native mode. Essentially every `Watcher` implementation needs to specify the Kubernetes model class that it handles via the `Watcher`'s generic type at class definition time. To better understand this, suppose we want to watch for changes to Kubernetes `Pod` resources. There are a couple ways to write such a `Watcher` that are guaranteed to work in native:

```
client.pods().watch(new Watcher<Pod>() {  
    @Override  
    public void eventReceived(Action action, Pod pod) {  
        // do something  
    }  
  
    @Override  
    public void onClose(KubernetesClientException e) {  
        // do something  
    }  
});
```

or

```
public class PodResourceWatcher implements Watcher<Pod> {  
    @Override  
    public void eventReceived(Action action, Pod pod) {  
        // do something  
    }  
  
    @Override  
    public void onClose(KubernetesClientException e) {  
        // do something  
    }  
}  
  
...  
  
client.pods().watch(new PodResourceWatcher());
```

Note that defining the generic type via a class hierarchy similar to the following example will also work correctly:

```
public abstract class MyWatcher<S> implements Watcher<S> {
}

...

client.pods().watch(new MyWatcher<Pod>() {
    @Override
    public void eventReceived(Action action, Pod pod) {
        // do something
    }
});
```



The following example will **not** work in native mode because the generic type of watcher cannot be determined by looking at the class and method definitions thus making Quarkus unable to properly determine the Kubernetes model class for which reflection registration is needed:

```
public class ResourceWatcher<T extends HasMetadata> implements
Watcher<T> {
    @Override
    public void eventReceived(Action action, T resource) {
        // do something
    }

    @Override
    public void onClose(KubernetesClientException e) {
        // do something
    }
}

client.pods().watch(new ResourceWatcher<Pod>());
```

Access to the Kubernetes API

In many cases in order to access the Kubernetes API server a **ServiceAccount**, **Role** and **RoleBinding** will be necessary. An example that allows listing all pods could look something like this:


```

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: <applicationName>
  namespace: <namespace>
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: <applicationName>
  namespace: <namespace>
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: <applicationName>
  namespace: <namespace>
roleRef:
  kind: Role
  name: <applicationName>
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: ServiceAccount
    name: <applicationName>
    namespace: <namespace>

```
















Replace `<applicationName>` and `<namespace>` with your values. Have a look at [Configure Service Accounts for Pods](#) to get further information.

Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.kubernetes-client.trust-certs</code> Whether or not the client should trust a self signed certificate if so presented by the API server	boolean	<code>false</code>

 <code>quarkus.kubernetes-client.master-url</code>		
URL of the Kubernetes API server	string	
 <code>quarkus.kubernetes-client.namespace</code>		
Default namespace to use	string	
 <code>quarkus.kubernetes-client.ca-cert-file</code>		
CA certificate file	string	
 <code>quarkus.kubernetes-client.ca-cert-data</code>		
CA certificate data	string	
 <code>quarkus.kubernetes-client.client-cert-file</code>		
Client certificate file	string	
 <code>quarkus.kubernetes-client.client-cert-data</code>		
Client certificate data	string	
 <code>quarkus.kubernetes-client.client-key-file</code>		
Client key file	string	
 <code>quarkus.kubernetes-client.client-key-data</code>		
Client key data	string	
 <code>quarkus.kubernetes-client.client-key-algo</code>		
Client key algorithm	string	
 <code>quarkus.kubernetes-client.client-key-passphrase</code>		
Client key passphrase	string	
 <code>quarkus.kubernetes-client.username</code>		
Kubernetes auth username	string	
 <code>quarkus.kubernetes-client.password</code>		
Kubernetes auth password	string	

 <code>quarkus.kubernetes-client.watch-reconnect-interval</code> Watch reconnect interval	Duration 	PT1S
 <code>quarkus.kubernetes-client.watch-reconnect-limit</code> Maximum reconnect attempts in case of watch failure By default there is no limit to the number of reconnect attempts	int	-1
 <code>quarkus.kubernetes-client.connection-timeout</code> Maximum amount of time to wait for a connection with the API server to be established	Duration 	PT10S
 <code>quarkus.kubernetes-client.request-timeout</code> Maximum amount of time to wait for a request to the API server to be completed	Duration 	PT10S
 <code>quarkus.kubernetes-client.rolling-timeout</code> Maximum amount of time in milliseconds to wait for a rollout to be completed	Duration 	PT15M
 <code>quarkus.kubernetes-client.http-proxy</code> HTTP proxy used to access the Kubernetes API server	string	
 <code>quarkus.kubernetes-client.https-proxy</code> HTTPS proxy used to access the Kubernetes API server	string	
 <code>quarkus.kubernetes-client.proxy-username</code> Proxy username	string	
 <code>quarkus.kubernetes-client.proxy-password</code> Proxy password	string	
 <code>quarkus.kubernetes-client.no-proxy</code> IP addresses or hosts to exclude from proxying	list of string	
 <code>quarkus.kubernetes-config.secrets.enabled</code> Whether or not configuration can be read from secrets. If set to <code>true</code> , Kubernetes resources allowing access to secrets (role and role binding) will be generated.	boolean	false

<code>quarkus.kubernetes-config.enabled</code>		
If set to true, the application will attempt to look up the configuration from the API server	boolean	<code>false</code>
<code>quarkus.kubernetes-config.fail-on-missing-config</code>		
If set to true, the application will not start if any of the configured config sources cannot be located	boolean	<code>true</code>
<code>quarkus.kubernetes-config.config-maps</code>		
ConfigMaps to look for in the namespace that the Kubernetes Client has been configured for	list of string	
<code>quarkus.kubernetes-config.secrets</code>		
Secrets to look for in the namespace that the Kubernetes Client has been configured for. If you use this, you probably want to enable <code>quarkus.kubernetes-config.secrets.enabled</code> .	list of string	
<code>quarkus.kubernetes-config.namespace</code>		
Namespace to look for config maps and secrets. If this is not specified, then the namespace configured in the kubectl config context is used. If the value is specified and the namespace doesn't exist, the application will fail to start.	string	



About the Duration format

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the [Duration#parse\(\) javadoc](#).

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.