Frequently Asked Questions:

How do I use KubernetesClient with IPv6 Kubernetes Clusters?

The Kubernetes Client and all its HttpClient provided implementations should be compatible with IPv6 Kubernetes Clusters.

What artifact(s) should my project depend on?

Fabric8 Kubernetes Client version 6 introduced more options with regards to dependencies.

1. Have `compile` dependencies on `kubernetes-client` or `openshift-client` - this is no different than what was done with version 5 and before. If you have done custom development involving effectively internal classes, you'll need to still use this option.
2. Have `compile` dependencies on `kubernetes-client-api` or `openshift-client-api`, and a runtime dependency on `kubernetes-client` or `openshift-client`. This option will provide your application with a cleaner compile time classpath.

Furthermore, you will also have choices in the HttpClient implementation that is utilized.

By default, kubernetes-client has a runtime dependency on Vert.x (`kubernetes-httpclient-vertx`).

If you wish to use another HttpClient implementation typically you will exclude `kubernetes-httpclient-vertx` and include the other runtime or compile dependency instead.

I've tried adding a dependency to kubernetes-client, but I'm still getting weird class loading issues, what gives?

More than likely your project already has transitive dependencies to a conflicting version of the Fabric8 Kubernetes Client. For example spring-cloud-dependencies already depends upon the client. You should fully override the client version in this case via the `kubernetes-client-bom`:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.fabric8</groupId>
      <artifactId>kubernetes-client-bom</artifactId>
      <version>${fabric8.client.version}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  <!-- ... -->
```

```
  </dependencies>
</dependencyManagement>
```

That will align all of the transitive Fabric8 Kubernetes Client dependencies to you chosen version, fabric8.client.version. Please be aware though that substituting a different major version of the client may not be fully compatible with the existing usage in your dependency - if you run into an issue, please check for or create an issue for an updated version of that project's library that contains a later version of the Fabric8 Kubernetes Client.

What threading concerns are there?

There has been a lot of changes under the covers with thread utilization in the Fabric8 client over the 5.x and 6.x releases. So the exact details of what threads are created / used where will depend on the particular release version.

At the core the thread utilization will depend upon the HTTP client implementation. Per client OkHttp maintains a pool of threads for task execution. It will dedicate 2 threads out of that pool per WebSocket connection. If you have a lot of WebSocket usage (Informer or Watches) with OkHttp, you can expect to see a large number of threads in use.

With the JDK, Jetty, or Vert.x http clients they will maintain a smaller worker pool for all tasks that is typically sized by default based upon your available processors, and typically a selector / coordinator thread(s). It does not matter how many Informers or Watches you run, the same threads are shared.

To ease developer burden the callbacks on Watchers and ResourceEventHandlers will not be done directly by an HTTP client thread, but the order of calls will be guaranteed. This will make the logic you include in those callbacks tolerant to some blocking without compromising the on-going work at the HTTP client level. However, you should avoid truly long running operations as this will cause further events to queue and may eventually case memory issues.

> [!NOTE] It is recommended with any HTTP client implementation that logic you supply via Watchers, ExecListeners, ResourceEventHandlers, Predicates, Interceptors, LeaderCallbacks, etc. does not execute long running tasks.

On top of the HTTP client threads the Fabric8 client maintains a task thread pool for scheduled tasks and for tasks that are called from WebSocket operations, such as handling input and output streams and ResourceEventHandler call backs. This thread pool defaults to an unlimited number of cached threads, which will be shutdown when the client is closed - that is a sensible default with as the amount of concurrently running async tasks will typically be low. If you would rather take full control over the threading use KubernetesClientBuilder.withExecutor or KubernetesClientBuilder.withExecutorSupplier - however note that constraining this thread pool too much will result in a build up of event processing queues.

Finally, the fabric8 client will use 1 thread per PortForward and an additional thread per socket connection - this may be improved upon in the future.

What additional logging is available?

Like many Java application the Fabric8 Client utilizes slf4j. You may configure support for whatever underlying logging framework suits your needs. The logging contexts for the Fabric8 Client follow the standard convention of the package structure - everything from within the client will be rooted at the io.fabric8 context. Third-party dependencies, including the chosen HTTP client implementation, will have different root contexts.

If you are using Pod exec, which can occur indirectly via pod operations like copying files, and not seeing the expected behavior - please enable debug logging for the io.fabric8.kubernetes.client.dsl.internal context. That will provide the stdErr and stdOut as debug logs to further diagnose what is occurring.

How to use KubernetesClient in OSGi?

Fabric8 Kubernetes Client provides `ManagedKubernetesClient` and `ManagedOpenShiftClient` as OSGi Declarative Service. In order to use it, you must have Service Component Runtime (SCR) feature enabled in your OSGi runtime. In Apache Karaf, you can add this to the Karaf Maven Plugin configuration:

```
<plugin>
  <groupId>org.apache.karaf.tooling</groupId>
  <artifactId>karaf-maven-plugin</artifactId>
  <version>${karaf.version}</version>
  <configuration>
    <startupFeatures>
      <feature>scr</feature>
    </startupFeatures>
  </configuration>
</plugin>
```

You need to provide component configuration files for the client you're using. For example, in case of Apache Karaf, place configuration files in this directory:

```
src/
   main
       resources
           assembly
             etc
                 io.fabric8.kubernetes.client.cfg
                 io.fabric8.openshift.client.cfg
```

Once added KubernetesClient declarative services would be exposed automatically on startup. Then you can inject it in your project. Here is an example

using Apache camel's `@BeanInject` annotation:

```
@BeanInject
private KubernetesClient kubernetesClient;
```

Why am I getting Exception when using * in NO_PROXY ?

Starting Fabric8 Kubernetes Client v6.1.0, we've changed `NO_PROXY` matching as simple as possible and not support any meta characters. It honors the GNU WGet Spec.

So instead of providing `NO_PROXY` like this:

(Unsupported) :x:

```
NO_PROXY: localhost,127.0.0.1,*.google.com, *.github.com
```

we should provide it like this:

(Supported) :heavy_check_mark:

```
NO_PROXY: localhost,127.0.0.1,.google.com,.github.com
```

How does KubernetesClient load proxy URL from various sources?

KubernetesClient loads proxy URL from the following sources (in decreasing order of precedence): - `ConfigBuilder.withHttpProxy` / `ConfigBuilder.withHttpsProxy` - Cluster's `proxy-url` in `~/.kube/config` - System Properties or Environment Variables - `HTTP_PROXY` : Should be used for HTTP requests (when Kubernetes ApiServer is serving plain HTTP requests) - `HTTPS_PROXY` : Should be used for HTTPS requests (when Kubernetes ApiServer is serving HTTPS)

URLs with `http`, `https`, and `socks5` schemes are supported.

Optimistic Locking Behavior

Unfortunately it's a little complicated as it depends on what operation you are doing - we'll work towards ensuring the Javadocs are as informative as possible. Here is quick overview:

- Basic mutative operations such as update and all variations of patch (patch, edit, accept, serverSideApply) that operate on a given item - are all locked to the resourceVersion on that item. If you don't want this behavior then set the resourceVersion to null:

```
resource.accept(x -> {
  resource.getMetadata().setResourceVersion(null);
  resource... // some other modifications
});
```

When the resourceVersion is null for an update the client obtains the latest resourceVersion prior to attempting the `PUT` operation - in a rare circumstance this may still fail due to a concurrent modification.

When the resourceVersion is null for a patch the server will always attempt to perform the patch - but of course there may be conflicts to deal with if there has been an intervening modification.

> [!NOTE] When using informers - do not make modifications to the resources obtained from the cache - especially to the resourceVersion.
>
> [!NOTE] It is not recommended to use serverSideApply directly against modified existing resources - the intent is to apply only the desired state owned by the applier. See the next topic for more.

- Delete is not locked by default, you may use the applicable lockResourceVersion method if you want the delete to apply only to a specific resourceVersion
- Specialized mutative operations are generally unlocked - this includes scale, rolling operations (resume, pause, restart), and others. The rationale is that you want the narrow operation to succeed even if there has been an intervening change. If you encounter a situation where you require these operations be locked, please raise an issue so that we can see about making the locking function applicable.
- A handful of additional operations, such as undo, updateImage, and others are currently locked by default. This may not be intentional - under the covers this is apply a json patch; older versions of json patching that created the resource diff were unlocked by default. If you encounter an exception due to a concurrent modification performing an operation that seems like it should ignore that possibility by default please raise an issue.
- Legacy operations such as createOrReplace or replace were effectively unlocked - they would repeatedly retry the operation with the freshest resourceVersion until it succeeded. These methods have been deprecated because of the complexity of their implementation and the broad unlocking behavior by default could be considered unsafe.

Alternatives to createOrReplace and replace

`createOrReplace` was introduced as an alternative to the kubectl apply operation. Over the years there were quite a few issues highlighting where the behavior was different, and there were only limited workarounds and improvements offered. Given the additional complexity of matching the kubectl client side apply behavior, that was never offered as an option. Now that there is first class support for serverSideApply it can be used, but it comes with a couple of caveats:

- You will want to use forceConflicts - `resource.forceConflicts().serverSideApply()` - this is especially true when acting as a controller updating a resource that manages
- For some resource types serverSideApply may cause vacuous revisions - see https://github.com/kubernetes/kubernetes/issues/118519 - if you are being informed of modifications on those resources you must either filter those out, don't perform the serverSideApply, or use the java-operator-sdk that should handle that possibility already.
- Keep in mind that serverSideApply is not the same as client side apply. In particular serverSideApply does not treat list merging the same, which may lead to unexpected behavior when list item merge keys are modified by actors other than the manager of that field. See the upstream issue: https://github.com/kubernetes/kubernetes/issues/118725
- A common pattern for createOrReplace was obtaining the current resource from the api server (possibly via an informer cache), making modifications, then calling createOrReplace. Doing something similar with serverSideApply will fail as the managedFields will be populated. While you may be tempted to simply clear the managedFields and the resourceVersion, this is generally not what you should be doing unless you want your logic to assume ownership of every field on the resource. Instead you should construct the object with the desired state and invoke serverSideApply(), letting the server figure how to merge the changes. If you have a more involved situation please read the upstream server side apply documentation to understand topics like transferring ownership and partial patches.

If the limitations / changes necessary to use serverSideApply are too much, you may also use the createOr method. Instead of:

```
resource.createOrReplace()
```

you would use:

```
resource.unlock().createOr(NonDeletingOperation::update)
```

Or `NonDeletingOperation::patch`. The use of the unlock function is optional and is only needed if you are starting with a item that has the resourceVersion populated.
If you have any concern over replacing concurrent changes you should omit the usage of the unlock function.

The alternative to replace is either serverSideApply - with the same caveats as above - or to use update, but with resourceVersion set to null or usage of the unlock function.

> [!NOTE] When using informers - do not make modifications to the resources obtained from the cache - especially to the resourceVersion. If you use the unlock function it will make changes to a copy of your item.

What credentials does the client use for authentication?

By default, KubernetesClient tries to look up for Kubernetes Cluster information in the following sources: - `kubeconfig` file (`~/.kube/config` or `kubeconfig` environment variable) - Currently mounted ServiceAccount inside Pod (`/var/run/secrets/kubernetes.io/serviceaccount/`) - System properties - Environment variables

Be aware that the cluster autoconfiguration sets up more than just authentication information; see the implementation of `io.fabric8.kubernetes.client.Config#configFromSysPropsOr` for details.

Running KubernetesClient from outside a Kubernetes cluster: When running KubernetesClient outside a Kubernetes cluster, KubernetesClient tries looking into user's `~/.kube/config` (or file set by `kubeconfig` environment variable), System properties and Environment variables for fetching Kubernetes Cluster information.

Running KubernetesClient from within a Pod in Kubernetes cluster: Kubernetes supports service account automounting, a feature that automatically provisions the authentication information of a namespace's default service account in every pod. As part of the auto-configuration process. Fabric8 Kubernetes Client integrates with auto mounting and will use these credentials by default if available.

This can be inappropriate in a few cases, such as when you wish to perform kubernetes operation on behalf of third parties in their namespaces. You can disable auto mounting on the service account object, or on the pod object, as explained here. You can also disable the kubernetes client automatic configuration.

Disable KubernetesClient automatic configuration: In order to disable the kubernetes client automatic configuration, you can use any of the following mechanisms:

- Set the system property `kubernetes.disable.autoConfig`
- Set the environment variable `KUBERNETES_DISABLE_AUTOCONFIG` (overridden by the system property)
- Pass `Config.empty()` explicitly as an argument to the Kubernetes configuration builder like so:

```
var client = new KubernetesClientBuilder(Config.empty())
                    .withMasterUrl(...)
                    .withOauthToken(...)
                    // etc.
                    .build()
```