

Quarkus - HTTP Reference

This document explains various HTTP features that you can use in Quarkus.

HTTP is provided using Eclipse Vert.x as the base HTTP layer. Servlet's are supported using a modified version of Undertow that runs on top of Vert.x, and RESTEasy is used to provide JAX-RS support. If Undertow is present RESTEasy will run as a Servlet filter, otherwise it will run directly on top of Vert.x with no Servlet involvement.

1. Serving Static Resources

To serve static resources you must place them in the `META-INF/resources` directory of your application. This location was chosen as it is the standard location for resources in `jar` files as defined by the Servlet spec. Even though Quarkus can be used without Servlet following this convention allows existing code that places its resources in this location to function correctly.

2. Configuring the Context path

By default Quarkus will serve content from under the root context. If you want to change this you can use the `quarkus.http.root-path` config key to set the context path.

If you are using Servlet you can control the Servlet context path via `quarkus.servlet.context-path`. This item is relative to the http root above, and will only affect Servlet and things that run on top of Servlet. Most applications will want to use the HTTP root as this affects everything that Quarkus serves.

If both are specified then all non-Servlet web endpoints will be relative to `quarkus.http.root-path`, while Servlet's will be served relative to `{quarkus.http.root-path}/{quarkus.servlet.context-path}`.

If REST Assured is used for testing and `quarkus.http.root-path` is set then Quarkus will automatically configure the base URL for use in Quarkus tests, so test URL's should not include the root path.

3. Supporting secure connections with SSL

In order to have Quarkus support secure connections, you must either provide a certificate and associated key file, or supply a keystore.

In both cases, a password must be provided. See the designated paragraph for a detailed description of how to provide it.



To enable SSL support with native executables, please refer to our [Using SSL With Native Executables guide](#).

3.1. Providing a certificate and key file

If the certificate has not been loaded into a keystore, it can be provided directly using the properties listed below. Quarkus will first try to load the given files as resources, and uses the filesystem as a fallback. The certificate / key pair will be loaded into a newly created keystore on startup.

Your `application.properties` would then look like this:

```
quarkus.http.ssl.certificate.file=/path/to/certificate
quarkus.http.ssl.certificate.key-file=/path/to/key
```

3.2. Providing a keystore

An alternate solution is to directly provide a keystore which already contains a default entry with a certificate. You will need to at least provide the file and a password.

As with the certificate/key file combination, Quarkus will first try to resolve the given path as a resource, before attempting to read it from the filesystem.

Add the following property to your `application.properties`:

```
quarkus.http.ssl.certificate.key-store-file=/path/to/keystore
```

As an optional hint, the type of keystore can be provided as one of the options listed. If the type is not provided, Quarkus will try to deduce it from the file extensions, defaulting to type JKS.

```
quarkus.http.ssl.certificate.key-store-file-type=[one of JKS,
JCEKS, P12, PKCS12, PFX]
```

3.3. Setting the password

In both aforementioned scenarios, a password needs to be provided to create/load the keystore with. The password can be set in your `application.properties` (in plain-text) using the following property:

```
quarkus.http.ssl.certificate.key-store-password=your-password
```

However, instead of providing the password as plain-text in the configuration file (which is considered bad practice), it can instead be supplied (using [MicroProfile config](#)) as the environment variable `QUARKUS_HTTP_SSL_CERTIFICATE_KEY_STORE_PASSWORD`. This will also work in tandem with [Kubernetes secrets](#).

Note: in order to remain compatible with earlier versions of Quarkus (before 0.16) the default password is set to "password". It is therefore not a mandatory parameter!

3.4. Disable the HTTP port

It is possible to disable the HTTP port and only support secure requests. This is done via the `quarkus.http.insecure-requests` property in `application.properties`. There are three possible values:

enabled

The default, HTTP works as normal

redirect

HTTP requests will be redirected to the HTTPS port

disabled

The HTTP port will not be opened.



if you use `redirect` or `disabled` and have not added a SSL certificate or keystore, your server will not start!

4. CORS filter

[Cross-origin resource sharing](#) (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Quarkus comes with a CORS filter which implements the `javax.servlet.Filter` interface and intercepts all incoming HTTP requests. It can be enabled in the Quarkus configuration file, `src/main/resources/application.properties`:

```
quarkus.http.cors=true
```

If the filter is enabled and an HTTP request is identified as cross-origin, the CORS policy and headers defined using the following properties will be applied before passing the request on to its actual target (servlet, JAX-RS resource, etc.):

Property Name	Default	Description
<code>quarkus.http.cors.origins</code>		The comma-separated list of origins allowed for CORS. The filter allows any origin if this is not set.
<code>quarkus.http.cors.methods</code>		The comma-separated list of HTTP methods allowed for CORS. The filter allows any method if this is not set.
<code>quarkus.http.cors.headers</code>		The comma-separated list of HTTP headers allowed for CORS. The filter allows any header if this is not set.
<code>quarkus.http.cors.exposed-headers</code>		The comma-separated list of HTTP headers exposed in CORS.

Property Name	Default	Description
<code>quarkus.http.cors.access-control-max-age</code>		The duration (see note below) indicating how long the results of a pre-flight request can be cached. This value will be returned in a <code>Access-Control-Max-Age</code> response header.

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.

Here's what a full CORS filter configuration could look like:

```
quarkus.http.cors=true
quarkus.http.cors.origins=http://foo.com,http://www.bar.io
quarkus.http.cors.methods=GET,PUT,POST
quarkus.http.cors.headers=X-Custom
quarkus.http.cors.exposed-headers=Content-Disposition
quarkus.http.cors.access-control-max-age=24H
```

5. HTTP Limits Configuration

The following properties are supported.

Property Name	Default	Description
<code>quarkus.http.limits.max-body-size</code>	<code>unlimited</code>	The maximum size of request body.
<code>quarkus.http.limits.max-header-size</code>	<code>20K</code>	The maximum length of all headers.



The following config options will recognize sizes expressed as strings in this format (shown as a regular expression): `[0-9]+[KkMmGgTtPpEeZzYy]?`. If no unit suffix is given, bytes are assumed.

- `quarkus.http.limits.max-body-size`,
- `quarkus.http.limits.max-header-size`

6. Servlet Config

To use Servlet you need to explicitly include `quarkus-undertow`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-undertow</artifactId>
</dependency>
```

6.1. undertow-handlers.conf

You can make use of the Undertow predicate language using an `undertow-handlers.conf` file. This file should be placed in the `META-INF` directory of your application jar. This file contains handlers defined using the [Undertow predicate language](#).

6.2. Configuring HTTP Access Logs

You can add HTTP request logging by configuring the `AccessHandler` in the `undertow-handlers.conf` file.

The simplest possible configuration can be a standard Apache `common` Log Format:

```
access-log('common')
```

This will log every request using the standard Quarkus logging infrastructure under the `io.undertow.accesslog` category.

You can customize the category like this:

```
access-log(format='common', category='my.own.category')
```

Finally the logging format can be customized:

```
access-log(format='%h %l %u %t "%r" %s %b %D "%{i,Referer}"
"%{i,User-Agent}" "%{i,X-Request-ID}"', category='my.own.category')
```

6.3. web.xml

If you are using a `web.xml` file as your configuration file, you can place it in the `src/main/resources/META-INF` directory.