



What is Quarkus?

Quarkus is a Kubernetes Native Java stack tailored for GraalVM & OpenJDK HotSpot, crafted from the best of breed Java libraries and standards. Also focused on developer experience, making things just work with little to no configuration and allowing to do live coding.

Cheat-sheet tested with **Quarkus 1.0.0.CR2**.

Getting Started

Quarkus comes with a Maven archetype to scaffold a very simple starting project.

```
mvn io.quarkus:quarkus-maven-plugin:1.0.0.CR2:create \
-DprojectId=org.acme \
-DprojectArtifactId=getting-started \
-DclassName="org.acme.quickstart.GreetingResource" \
-Dpath="/hello"
```

This creates a simple JAX-RS resource called `GreetingResource`.

```
@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

Gradle

There is no way to scaffold a project in Gradle but you only need to do:

```
plugins {
    id 'java'
    id 'io.quarkus' version '0.26.1'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation enforcedPlatform('io.quarkus:quarkus-bom:0.26.1')
    implementation 'io.quarkus:quarkus-resteasy'
}
```

Or in Kotlin:

```
plugins {
    java
}
apply(plugin = "io.quarkus")

repositories {
    mavenCentral()
}

dependencies {
    implementation(enforcedPlatform("io.quarkus:quarkus-bom:0.26.1"))
    implementation("io.quarkus:quarkus-resteasy")
}
```

Extensions

Quarkus comes with extensions to integrate with some libraries such as JSON-B, Camel or MicroProfile spec. To list all available extensions just run:

```
./mvnw quarkus:list-extensions
```



You can use `-DsearchPattern=panache` to filter out all extensions except the ones matching the expression.

And to register the extensions into build tool:

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



`extensions` property supports CSV format to register more than one extension at once.

Application Lifecycle

You can be notified when the application starts/stops by observing `StartupEvent` and `ShutdownEvent` events.

```
@ApplicationScoped
public class ApplicationLifecycle {
    void onStart(@Observes StartupEvent event) {}
    void onStop(@Observes ShutdownEvent event) {}
}
```

Adding Configuration Parameters

To add configuration to your application, Quarkus relies on MicroProfile Config spec.

```
@ConfigProperty(name = "greetings.message")
String message;

@ConfigProperty(name = "greetings.message",
    defaultValue = "Hello")
String messageWithDefault;

@ConfigProperty(name = "greetings.message")
Optional<String> optionalMessage;
```

Properties can be set as:

- Environment variables (`GREETINGS_MESSAGE`).
- System properties (`-Dgreetings.message`).
- Resources `src/main/resources/application.properties`.
- External config directory under the current working directory: `config/application.properties`.



Array, List and Set are supported. The delimiter is comma (,) char and \ is the escape char.

Configuration Profiles

Quarkus allow you to have multiple configuration in the same file (`application.properties`).

The syntax for this is `%{profile}.config.key=value`.

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

HTTP port will be 9090, unless the 'dev' profile is active.

Default profiles are:

- `dev`: Activated when in development mode (`quarkus:dev`).
- `test`: Activated when running tests.
- `prod`: The default profile when not running in development or test mode

You can create custom profile names by enabling the profile either setting `quarkus.profile` system property or `QUARKUS_PROFILE` environment variable.

```
quarkus.http.port=9090
%staging.quarkus.http.port=9999
```

And enable it `quarkus.profile=staging`.

You can also set it in the build tool:

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<version>${surefire-plugin.version}</version>
<configuration>
  <systemPropertyVariables>
    <quarkus.test.profile>foo</quarkus.test.profile>
    <buildDirectory>${project.build.directory}
  </buildDirectory>
  </systemPropertyVariables>
</configuration>
```



Same for `maven-failsafe-plugin`.

```
test {
    useJUnitPlatform()
    systemProperty "quarkus.test.profile", "foo"
}
```

@ConfigProperties

As an alternative to injecting multiple related configuration values, you can also use the `@io.quarkus.arc.config.ConfigProperties` annotation to group properties.

```
@ConfigProperties(prefix = "greeting")
public class GreetingConfiguration {
    private String message;
    // getter/setter
}
```

This class maps `greeting.message` property defined in `application.properties`.

You can inject this class by using CDI `@Inject GreetingConfiguration greeting;`.

Also you can use an interface approach:

```
@ConfigProperties(prefix = "greeting")
public interface GreetingConfiguration {

    @ConfigProperty(name = "message")
    String message();
    String getSuffix();
}
```

If property does not follow getter/setter naming convention you need to use `org.eclipse.microprofile.config.inject.ConfigProperty` to set it.

Nested objects are also supporte:

```
@ConfigProperties(prefix = "greeting")
public class GreetingConfiguration {
    public String message;
    public HiddenConfig hidden;

    public static class HiddenConfig {
        public List<String> recipients;
    }
}
```

And an `application.properties` mapping previous class:

```
greeting.message = hello
greeting.hidden.recipients=Jane,John
```

Bean Validation is also supported so properties are validated at startup time, for example `@Size(min = 20) public String message;`.



`prefix` attribute is not mandatory. If not provided, attribute is determined by class name (ie `GreetingConfiguration` is translated to `greeting` or `GreetingExtraConfiguration` to `greeting-extra`). The suffix of the class is always removed.

YAML Config

YAML configuration is also supported. The configuration file is called `application.yaml` and you need to register a dependency to enable its support:

`pom.xml`

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-config-yaml</artifactId>
</dependency>
```

```
quarkus:
  datasource:
    url: jdbc:postgresql://localhost:5432/some-database
    driver: org.postgresql.Driver
```

Or with profiles:

```
"%dev":
  quarkus:
    datasource:
      url: jdbc:postgresql://localhost:5432/some-database
      driver: org.postgresql.Driver
```

In case of subkeys `~` is used to refer to the unprefixed part.

```
quarkus:
  http:
    cors:
      ~: true
      methods: GET,PUT,POST
```

Is equivalent to:

```
quarkus.http.cors=true
quarkus.http.cors.methods=GET,PUT,POST
```

Custom Loader

You can implement your own `ConfigSource` to load configuration from different places than the default ones provided by Quarkus. For example, database, custom XML, REST Endpoints, ...

You need to create a new class and implement `ConfigSource` interface:

```
package com.acme.config;
public class InMemoryConfig implements ConfigSource {

    private Map<String, String> prop = new HashMap<>();

    public InMemoryConfig() {
        // Init properties
    }

    @Override
    public int getOrdinal() {
        // The highest ordinal takes precedence
        return 900;
    }

    @Override
    public Map<String, String> getProperties() {
        return prop;
    }

    @Override
    public String getValue(String propertyName) {
        return prop.get(propertyName);
    }

    @Override
    public String getName() {
        return "MemoryConfigSource";
    }
}
```

Then you need to register the `ConfigSource` as Java service. Create a file with the following content:

`/META-INF/services/org.eclipse.microprofile.config.spi.ConfigSource`

```
com.acme.config.InMemoryConfig
```

Custom Converters

You can implement your own conversion types from String. Implement `org.eclipse.microprofile.config.spi.Converter` interface:

```
@Priority(DEFAULT_QUARKUS_CONVERTER_PRIORITY + 100)
public class CustomInstantConverter
    implements Converter<Instant> {

    @Override
    public Instant convert(String value) {
        if ("now".equals(value.trim())) {
            return Instant.now();
        }
        return Instant.parse(value);
    }
}
```

`@Priority` annotation is used to override the default `InstantConverter`.

Then you need to register the `Converter` as Java service. Create a file with the following content:

```
/META-INF/services/org.eclipse.microprofile.config.spi.Converter
```

```
com.acme.config.CustomInstantConverter
```

Custom Context Path

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

Injection

Quarkus is based on CDI 2.0 to implement injection of code. It is not fully supported and only a subset of the specification is implemented.

```
@ApplicationScoped
public class GreetingService {

    public String message(String message) {
        return message.toUpperCase();
    }
}
```

Scope annotation is mandatory to make the bean discoverable.

```
@Inject
GreetingService greetingService;
```



Quarkus is designed with Substrate VM in mind. For this reason, we encourage you to use *package-private* scope instead of *private*.

Produces

You can also create a factory of an object by using `@javax.enterprise.inject.Produces` annotation.

```
@Produces
@ApplicationScoped
Message message() {
    Message m = new Message();
    m.setMsn("Hello");
    return m;
}

@Inject
Message msg;
```

Qualifiers

You can use qualifiers to return different implementations of the same interface or to customize the configuration of the bean.

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Quote {
    @Nonbinding String value();
}

@Produces
@Quote("")
Message message(InjectionPoint msg) {
    Message m = new Message();
    m.setMsn(
        msg.getAnnotated()
            .getAnnotation(Quote.class)
            .value()
    );

    return m;
}
```

```
@Inject
@Quote("Aloha Beach")
Message message;
```



Quarkus breaks the CDI spec by allowing you to inject qualified beans without using `@Inject` annotation.

```
@Quote("Aloha Beach")
Message message;
```

JSON Marshalling/Unmarshalling

To work with `JAXB` you need to add a dependency:

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-resteasy-jsonb"
```

Any POJO is marshaled/unmarshalled automatically.

```
public class Sauce {
    private String name;
    private long scovilleHeatUnits;

    // getter/setters
}
```

JSON equivalent:

```
{
  "name": "Blair's Ultra Death",
  "scovilleHeatUnits": 1100000
}
```

In a `POST` endpoint example:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response create(Sauce sauce) {
    // Create Sauce
    return Response.created(URI.create(sauce.getId()))
        .build();
}
```

To work with `Jackson` you need to add:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-resteasy-jackson"
```

If you don't want to use the default `ObjectMapper` you can customize it by:

```
@ApplicationScoped
public class CustomObjectMapperConfig {
    @Singleton
    @Produces
    public ObjectMapper objectMapper() {
        ObjectMapper objectMapper = new ObjectMapper();
        // perform configuration
        return objectMapper;
    }
}
```

XML Marshalling/Unmarshalling

To work with `JAXB` you need to add a dependency:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-resteasy-jaxb"
```

Then annotated POJOs are converted to XML


```
@XmlRootElement
public class Message {
}

@GET
@Produces(MediaType.APPLICATION_XML)
public Message hello() {
    return message;
}
```

Validator

Quarkus uses Hibernate Validator to validate input/output of REST services and business services using Bean validation spec.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-hibernate-validator"
```

Annotate POJO objects with validator annotations such as: `@NotNull`, `@Digits`, `@NotBlank`, `@Min`, `@Max`, ...


```
public class Sauce {

    @NotBlank(message = "Name may not be blank")
    private String name;
    @Min(0)
    private long scovilleHeatUnits;

    // getter/setters
}
```

To validate an object use `@Valid` annotation:

```
public Response create(@Valid Sauce sauce) {}
```



If a validation error is triggered, a violation report is generated and serialized as JSON. If you want to manipulate the output, you need to catch in the code the `ConstraintViolationException` exception.

Create Your Custom Constraints

First you need to create the custom annotation:

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR,
          PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { NotExpiredValidator.class})
public @interface NotExpired {

    String message() default "Sauce must not be expired";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };

}
```

You need to implement the validator logic in a class that implements `ConstraintValidator`.

```
public class NotExpiredValidator
    implements ConstraintValidator<NotExpired, LocalDate>
{

    @Override
    public boolean isValid(LocalDate value,
                          ConstraintValidatorContext ctx) {

        if ( value == null ) return true;
        LocalDate today = LocalDate.now();
        return ChronoUnit.YEARS.between(today, value) > 0;
    }
}
```

And use it normally:

```
@NotExpired
@JsonbDateFormat(value = "yyyy-MM-dd")
private LocalDate expired;
```

Manual Validation

You can call the validation process manually instead of relaying to `@Valid` by injecting `Validator` class.

```
@Inject
Validator validator;
```

And use it:

```
Set<ConstraintViolation<Sauce>> violations =
    validator.validate(sauce);
```

Logging

You can configure how Quarkus logs:

```
quarkus.log.console.enable=true
quarkus.log.console.level=DEBUG
quarkus.log.console.color=false
quarkus.log.category."com.lordofthejars".level=DEBUG
```

Prefix is `quarkus.log`.

<code>category."</code> <code><category-name></code> <code>".level</code>	Minimum level category (default: <code>INFO</code>)
<code>level</code>	Default minimum level (default: <code>INFO</code>)
<code>console.enabled</code>	Console logging enabled (default: <code>true</code>)
<code>console.format</code>	Format pattern to use for logging. Default value: <code>%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c{3.}] (%t) %s%e%n</code>
<code>console.level</code>	Minimum log level (default: <code>INFO</code>)
<code>console.color</code>	Allow color rendering (default: <code>true</code>)
<code>file.enable</code>	File logging enabled (default: <code>false</code>)
<code>file.format</code>	Format pattern to use for logging. Default value: <code>%d{yyyy-MM-dd HH:mm:ss,SSS} %h %N[%i] %-5p [%c{3.}] (%t) %s%e%n</code>
<code>file.level</code>	Minimum log level (default: <code>ALL</code>)
<code>file.path</code>	The path to log file (default: <code>quarkus.log</code>)
<code>file.rotation.max-file-size</code>	The maximum file size of the log file
<code>file.rotation.max-backup-index</code>	The maximum number of backups to keep (default: <code>1</code>)
<code>file.rotation.file-suffix</code>	Rotating log file suffix.
<code>file.rotation.rotate-on-boot</code>	Indicates rotate logs at bootup (default: <code>true</code>)
<code>file.async</code>	Log asynchronously (default: <code>false</code>)
<code>file.async.queue-length</code>	The queue length to use before flushing writing (default: <code>512</code>)

file.async.overflow

Action when queue is full (default: BLOCK)

syslog.enable

syslog logging is enabled (default: false)

syslog.format

The format pattern to use for logging to syslog. Default value:
%d{yyyy-MM-dd HH:mm:ss,SSS} %h %N[%i] %-5p [%c{3.}] (%t) %s%e%n

syslog.level

The minimum log level to write to syslog (default: ALL)

syslog.endpoint

The IP address and port of the syslog server (default: localhost:514)

syslog.app-name

The app name used when formatting the message in RFC5424 format (default: current process name)

syslog.hostname

The name of the host the messages are being sent from (default: current hostname)

syslog.facility

Priority of the message as defined by RFC-5424 and RFC-3164 (default: USER_LEVEL)

syslog.syslog-type

The syslog type of format message (default: RFC5424)

syslog.protocol

Protocol used (default: TCP)

syslog.use-counting-framing

Message prefixed with the size of the message (default false)

syslog.truncate

Message should be truncated (default: true)

syslog.block-on-reconnect

Block when attempting to reconnect (default: true)

syslog.async

Log asynchronously (default: false)

syslog.async.queue-length

The queue length to use before flushing writing (default: 512)

syslog.async.overflow

Action when queue is full (default: BLOCK)

JSON output

You can configure the output to be in *JSON* format instead of plain text.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-logging-json"
```

And the configuration values are prefix with `quarkus.log:`

json

JSON logging is enabled (default: true).

json.pretty-print

JSON output is "pretty-printed" (default: false)

json.date-format

Specify the date format to use (default: the default format)

json.record-delimiter

Record delimiter to add (default: no delimiter)

json.zone-id

The time zone ID

json.exception-output-type

The exception output type: detailed, formatted, detailed-and-formatted (default: detailed)

json.print-details

Detailed caller information should be logged (default: false)

Rest Client

Quarkus implements MicroProfile Rest Client spec:

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

To get content from `http://worldclockapi.com/api/json/cet/now` you need to create a service interface:

```
@Path("/api")
@RegisterRestClient
public interface WorldClockService {

    @GET @Path("/json/cet/now")
    @Produces(MediaType.APPLICATION_JSON)
    WorldClock getNow();

    @GET
    @Path("/json/{where}/now")
    @Produces(MediaType.APPLICATION_JSON)
    WorldClock getSauce(@BeanParam
                        WorldClockOptions worldClockOptions);

}
```

```
public class WorldClockOptions {
    @HeaderParam("Authorization")
    String auth;

    @PathParam("where")
    String where;
}
```

And configure the hostname at `application.properties`:

```
org.acme.quickstart.WorldClockService/mp-rest/url=
http://worldclockapi.com
```

Injecting the client:

```
@RestClient
WorldClockService worldClockService;
```

If invocation happens within JAX-RS, you can propagate headers from incoming to outgoing by using next property.

```
org.eclipse.microprofile.rest.client.propagateHeaders=
Authorization,MyCustomHeader
```



You can still use the JAX-RS client without any problem
`ClientBuilder.newClient().target(...)`

Adding headers

You can customize the headers passed by implementing MicroProfile `ClientHeadersFactory` annotation:

```
@RegisterForReflection
public class BaggageHeadersFactory
    implements ClientHeadersFactory {

    @Override
    public MultivaluedMap<String, String> update(
        MultivaluedMap<String, String> incomingHeaders,
        MultivaluedMap<String, String> outgoingHeaders) {}

}
```

And registering it in the client using `RegisterClientHeaders` annotation.

```
@RegisterClientHeaders(BaggageHeadersFactory.class)
@RegisterRestClient
public interface WorldClockService {}
```

Or statically set:

```
@GET
@ClientHeaderParam(name="X-Log-Level", value="ERROR")
Response getNow();
```

Asynchronous

A method on client interface can return a `CompletionStage` class to be executed asynchronously.

```
@GET @Path("/json/cet/now")
@Produces(MediaType.APPLICATION_JSON)
CompletionStage<WorldClock> getNow();
```

Multipart

It is really easy to send multipart form-data with Rest Client.

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-multipart-provider</artifactId>
</dependency>
```

The model object:

```
import java.io.InputStream;

import javax.ws.rs.FormParam;
import javax.ws.rs.core.MediaType;

import
    org.jboss.resteasy.annotations.providers.multipart.Part
    Type;

public class MultipartBody {

    @FormParam("file")
    @PartType(MediaType.APPLICATION_OCTET_STREAM)
    private InputStream file;

    @FormParam("fileName")
    @PartType(MediaType.TEXT_PLAIN)
    private String name;

    // getter/setters
}
```

And the Rest client interface:

```
import
    org.jboss.resteasy.annotations.providers.multipart.Mult
    ipartForm;

    @Path("/echo")
    @RegisterRestClient
    public interface MultipartService {

        @POST
        @Consumes(MediaType.MULTIPART_FORM_DATA)
        @Produces(MediaType.TEXT_PLAIN)
        String sendMultipartData(@MultipartForm
                                   MultipartBody data);

    }
```

SSL

You can configure Rest Client key stores.

```
org.acme.quickstart.WorldClockService/mp-rest/trustStore=
    classpath:/store.jks
org.acme.quickstart.WorldClockService/mp-rest/trustStorePas
sword=
    supersecret
```

Possible configuration properties:

`%s/mp-rest/trustStore`
Trust store location defined with `classpath:` or `file:` prefix.

`%s/mp-rest/trustStorePassword`
Trust store password.

`%s/mp-rest/trustStoreType`
Trust store type (default: `JKS`)

`%s/mp-rest/hostnameVerifier`
Custom hostname verifier class name.

`%s/mp-rest/keyStore`
Key store location defined with `classpath:` or `file:` prefix.

`%s/mp-rest/keyStorePassword`
Key store password.

`%s/mp-rest/keyStoreType`
Key store type (default: `JKS`)

Timeout

You can define the timeout of the Rest Client:

```
org.acme.quickstart.WorldClockService/mp-rest/connectTimeou
t=
    1000
org.acme.quickstart.WorldClockService/mp-rest/readTimeout=
    2000
```

Testing

Quarkus archetype adds test dependencies with JUnit 5 and Rest-Assured library to test REST endpoints.

```
@QuarkusTest
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/hello")
            .then()
                .statusCode(200)
                .body(is("hello"));
    }
}
```

Test port can be set in `quarkus.http.test-port` property.

You can also inject the URL where Quarkus is started:

```
@TestHTTPResource("index.html")
URL url;
```

Quarkus Test Resource

You can execute some logic before the first test run (`start`) and execute some logic at the end of the test suite (`stop`).

You need to create a class implementing `QuarkusTestResourceLifecycleManager` interface and register it in the test via `@QuarkusTestResource` annotation.


```
public class MyCustomTestResource
    implements QuarkusTestResourceLifecycleManager {

    @Override
    public Map<String, String> start() {
        // return system properties that
        // should be set for the running test
        return Collections.emptyMap();
    }

    @Override
    public void stop() {
    }

    // optional
    @Override
    public void inject(Object testInstance) {
    }

    // optional
    @Override
    public int order() {
        return 0;
    }
}
```



Returning new system properties implies running parallel tests in different JVMs.


And the usage:

```
@QuarkusTestResource(MyCustomTestResource.class)
public class MyTest {
}
```

Mocking

If you need to provide an alternative implementation of a service (for testing purposes) you can do it by using CDI `@Alternative` annotation using it in the test service placed at `src/test/java`:

```
@Alternative
@Priority(1)
@ApplicationScoped
public class MockExternalService extends ExternalService {}
```



This does not work when using native image testing.

A stereotype annotation `io.quarkus.test.Mock` is provided declaring `@Alternative`, `@Priority(1)` and `@Dependent`.

Interceptors

Tests are actually full CDI beans, so you can apply CDI interceptors:

```
@QuarkusTest
@Stereotype
@Transactional
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface TransactionalQuarkusTest {
}

@TransactionalQuarkusTest
public class TestStereotypeTestCase {}
```

Test Coverage Due the nature of Quarkus to calculate correctly the coverage information with JaCoCo, you might need offline instrumentation. I recommend reading this document to understand how JaCoCo and Quarkus works and how you can configure JaCoCo to get correct data.

Native Testing

To test native executables annotate the test with `@NativeImageTest`.

Persistence

Quarkus works with JPA(Hibernate) as persistence solution. But also provides an Active Record pattern implementation under Panache project.

To use database access you need to add Quarkus JDBC drivers instead of the original ones. At this time Apache Derby, H2, MariaDB, MySQL, MSSQL and PostgreSQL drivers are supported.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-hibernate-orm-panache,
            io.quarkus:quarkus-jdbc-mariadb"
```

```
@Entity
public class Developer extends PanacheEntity {

    // id field is implicit

    public String name;
}
```

And configuration in `src/main/resources/application.properties`:

```
quarkus.datasource.url=jdbc:mariadb://localhost:3306/mydb
quarkus.datasource.driver=org.mariadb.jdbc.Driver
quarkus.datasource.username=developer
quarkus.datasource.password=developer
quarkus.hibernate-orm.database.generation=update
```

List of datasource parameters.

`quarkus.datasource` as prefix is skipped in the next table.

Parameter	Type
<code>driver</code>	<code>String</code>
<code>url</code>	<code>String</code>
<code>username</code>	<code>String</code>
<code>password</code>	<code>String</code>
<code>min-size</code>	<code>Integer</code>
<code>max-size</code>	<code>Integer</code>
<code>initial-size</code>	<code>Integer</code>
<code>background-validation-interval</code>	<code>java.time.Duration</code>

Parameter	Type
acquisition-timeout	java.time.Duration
leak-detection-interval	java.time.Duration
idle-removal-interval	java.time.Duration
transaction-isolation-level	io.quarkus.agroal.runtime.TransactionIsolationLevel
enable-metrics	Boolean
xa	Boolean
validation-query-sql	Boolean

Hibernate configuration properties. Prefix `quarkus.hibernate-orm` is skipped.

Parameter	Description	Values[Default]
dialect	Class name of the Hibernate ORM dialect.	Not necessary to set.
dialect.storage-engine	The storage engine when the dialect supports multiple storage engines.	Not necessary to set.
sql-load-script	Name of the file containing the SQL statements to execute when starts. force Hibernate to skip SQL import.	<code>import.sql</code> <code>no-file</code>
batch-fetch-size	The size of the batches.	-1 disabled.
query.query-plan-cache-max-size	The maximum size of the query plan cache.	
query.default-null-ordering	Default precedence of null values in <code>ORDER BY</code> .	<code>[none]</code> , <code>first</code> , <code>last</code> .

Parameter	Description	Values[Default]
database.generation	Database schema is generation.	<code>[none]</code> , <code>create</code> , <code>drop-and-create</code> , <code>drop</code> , <code>update</code> .
database.generation.halt-on-error	Stop on the first error when applying the schema.	<code>[false]</code> , <code>true</code>
database.default-catalog	Default catalog.	
database.default-schema	Default Schema.	
database.charset	Charset.	

jdbc.timezone	Time Zone JDBC driver.	
jdbc.statement-fetch-size	Number of rows fetched at a time.	
jdbc.statement-batch-size	Number of updates sent at a time.	
log.sql	Show SQL logs	<code>[false]</code> , <code>true</code>
log.jdbc-warnings	Collect and show JDBC warnings.	<code>[false]</code> , <code>true</code>
statistics	Enable statiscs collection.	<code>[false]</code> , <code>true</code>

Database operations:

```
// Insert
Developer developer = new Developer();
developer.name = "Alex";
developer.persist();

// Find All
Developer.findAll().list();

// Find By Query
Developer.find("name", "Alex").firstResult();

// Delete
Developer developer = new Developer();
developer.id = 1;
developer.delete();

// Delete By Query
long numberOfDeleted = Developer.delete("name", "Alex");
```

Remember to annotate methods with `@Transactional` annotation to make changes persisted in the database.

If queries start with the keyword `from` then they are treated as *HQL* query, if not then next short form is supported:

- `order by` which expands to `from EntityName order by ...`
- `<columnName>` which expands to `from EntityName where <columnName>=?`
- `<query>` which is expanded to `from EntityName where <query>`

Static Methods

findById: `Object`
Returns object or null if not found. Overloaded version with `LockModeType` is provided.

findByIdOptional: `Optional<Object>`
Returns object or `java.util.Optional`.

find: `String`, [Object..., Map<String, Object>, Parameters]
Lists of entities meeting given query with parameters set.

find: `String`, Sort, [Object..., Map<String, Object>, Parameters]
Lists of entities meeting given query with parameters set sorted by `Sort` attribute/s.

findAll
Finds all entities.

findAll: `Sort`
Finds all entities sorted by `Sort` attribute/s.

stream: `String`, [Object..., Map<String, Object>, Parameters]
`java.util.stream.Stream` of entities meeting given query with parameters set.

stream: `String`, Sort, [Object..., Map<String, Object>, Parameters]

`java.util.stream.Stream` of entities meeting given query with parameters set sorted by `Sort` attribute/s.

streamAll
`java.util.stream.Stream` of all entities.

streamAll: `Sort`
`java.util.stream.Stream` of all entities sorted by `Sort` attribute/s.

count
Number of entities.

count: `String`, [Object..., Map<String, Object>, Parameters]
Number of entities meeting given query with parameters set.

deleteAll
Number of deleted entities.

delete: `String`, [Object..., Map<String, Object>, Parameters]
Number of deleted entities meeting given query with parameters set.

persist: [Iterable, Steram, Object...]
TIP: `find` methods defines a `withLock(LockModeType)` to define the lock type and `withHint(QueryHints.HINT_CACHEABLE, "true")` to define hints.

Pagination

```
PanacheQuery<Person> livingPersons = Person
    .find("status", Status.Alive);
livingPersons.page(Page.ofSize(25));

// get the first page
List<Person> firstPage = livingPersons.list();
// get the second page
List<Person> secondPage = livingPersons.nextPage().list();
```

If entities are defined in external JAR, you need to enable in these projects the `Jandex` plugin in project.

```
<plugin>
  <groupId>org.jboss.jandex</groupId>
  <artifactId>jandex-maven-plugin</artifactId>
  <version>1.0.3</version>
  <executions>
    <execution>
      <id>make-index</id>
      <goals>
        <goal>jandex</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.jboss</groupId>
      <artifactId>jandex</artifactId>
      <version>2.1.1.Final</version>
    </dependency>
  </dependencies>
</plugin>
```

DAO pattern

Also supports *DAO* pattern with `PanacheRepository<TYPE>.`

```
@ApplicationScoped
public class DeveloperRepository
    implements PanacheRepository<Person> {
    public Person findByName(String name){
        return find("name", name).firstResult();
    }
}
```

EntityManager You can inject `EntityManager` in your classes:

```
@Inject
EntityManager em;

em.persist(car);
```

Multiple datasources

You can register more than one datasource.

```
# default
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.url=jdbc:h2:tcp://localhost/mem:default
....
# users datasource
quarkus.datasource.users.driver=org.h2.Driver
quarkus.datasource.users.url=jdbc:h2:tcp://localhost/mem:users
```


Notice that after `datasource` you set the datasource name, in previous case `users`.

You can inject then `AgroalDataSource` with `io.quarkus.agroal.DataSource.`

```
@DataSource("users")
AgroalDataSource dataSource1;
```

Flushing

You can force flush operation by calling `.flush()` or `.persistAndFlush()` to make it in a single call.



This flush is less efficient and you still need to commit transaction.

Testing

There is a Quarkus Test Resource that starts and stops H2 server before and after test suite.

Register dependency `io.quarkus:quarkus-test-h2:test`.

And annotate the test:

```
@QuarkusTestResource(H2DatabaseTestResource.class)
public class FlywayTestResources {
}
```

Transactions

The easiest way to define your transaction boundaries is to use the `@Transactional` annotation.

Transactions are mandatory in case of none idempotent operations.

```
@Transactional
public void createDeveloper() {}
```

You can control the transaction scope:

- `@Transactional(REQUIRED)` (default): starts a transaction if none was started, stays with the existing one otherwise.
- `@Transactional(REQUIRES_NEW)`: starts a transaction if none was started; if an existing one was started, suspends it and starts a new one for the boundary of that method.
- `@Transactional(MANDATORY)`: fails if no transaction was started ; works within the existing transaction otherwise.
- `@Transactional(SUPPORTS)`: if a transaction was started, joins it ; otherwise works with no transaction.
- `@Transactional(NOT_SUPPORTED)`: if a transaction was started, suspends it and works with no transaction for the boundary of the method; otherwise works with no transaction.
- `@Transactional(NEVER)`: if a transaction was started, raises an exception; otherwise works with no transaction.

You can configure the default transaction timeout using `quarkus.transaction-manager.default-transaction-timeout` configuration property. By default it is set to 60 seconds.

You can set a timeout property, in seconds, that applies to transactions created within the annotated method by using `@TransactionConfiguration` annotation.

```
@Transactional
@TransactionConfiguration(timeout=40)
public void createDeveloper() {}
```

If you want more control over transactions you can inject `UserTransaction` and use a programmatic way.

```
@Inject UserTransaction transaction

transaction.begin();
transaction.commit();
transaction.rollback();
```

Infinispan

Quarkus integrates with Infinispan:

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

Serialization uses a library called Protostream.

Annotation based

```
@ProtoFactory
public Author(String name, String surname) {
    this.name = name;
    this.surname = surname;
}

@ProtoField(number = 1)
public String getName() {
    return name;
}

@ProtoField(number = 2)
public String getSurname() {
    return surname;
}
```

Initializer to set configuration settings.

```
@AutoProtoSchemaBuilder(includeClasses =
    { Book.class, Author.class },
    schemaPackageName = "book_sample")
interface BookContextInitializer
    extends SerializationContextInitializer {
}
```

User written based

There are three ways to create your schema:

Protofile

Creates a `.proto` file in the `META-INF` directory.

```
package book_sample;

message Author {
    required string name = 1;
    required string surname = 2;
}
```

In case of having a Collection field you need to use the `repeated` key (ie `repeated Author authors = 4`).

In code

Setting `proto` schema directly in a produced bean.

```
@Produces
FileDescriptorSource bookProtoDefinition() {
    return FileDescriptorSource
        .fromString("library.proto",
            "package book_sample;\n" +
            "message Author {\n" +
            "    required string name = 1;\n" +
            "    required string surname = 2;\n" +
            "}");
}
```

Marshaller

Using `org.infinispan.protostream.MessageMarshaller` interface.

```
public class AuthorMarshaller
    implements MessageMarshaller<Author> {

    @Override
    public String getTypeName() {
        return "book_sample.Author";
    }

    @Override
    public Class<? extends Author> getJavaClass() {
        return Author.class;
    }

    @Override
    public void writeTo(ProtoStreamWriter writer,
        Author author) throws IOException {
        writer.writeString("name", author.getName());
        writer.writeString("surname", author.getSurname());
    }

    @Override
    public Author readFrom(ProtoStreamReader reader)
        throws IOException {
        String name = reader.readString("name");
        String surname = reader.readString("surname");
        return new Author(name, surname);
    }
}
```

And producing the marshaller:

```
@Produces
MessageMarshaller authorMarshaller() {
    return new AuthorMarshaller();
}
```

Infinispan Embedded

```
./mvnw quarkus:add-extension
-Dextensions="infinispan-embeddedy"
```

Configuration in `infinispan.xml`:

```
<local-cache name="quarkus-transaction">
  <transaction
    transaction-manager-lookup=
      "org.infinispan.transaction.lookup.JBossStandaloneJ
TAManagerLookup"/>
</local-cache>
```

Set configuration file location in `application.properties`:

```
quarkus.infinispan-embedded.xml-config=infinispan.xml
```

And you can inject the main entry point for the cache:

```
@Inject
org.infinispan.manager.EmbeddedCacheManager cacheManager;
```

Flyway

Quarkus integrates with Flyway to help you on database schema migrations.

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

Then place migration files to the migrations folder (`classpath:db/migration`).

You can inject `org.flywaydb.core.Flyway` to programmatically execute the migration.

```
@Inject
Flyway flyway;

flyway.migrate();
```

Or can be automatically executed by setting `migrate-at-start` property to `true`.

```
quarkus.flyway.migrate-at-start=true
```

List of Flyway parameters.

`quarkus.` as prefix is skipped in the next table.

flyway.clean-at-start

Execute Flyway clean command (default: `false`)

flyway.migrate-at-start

Flyway migration automatically (default: `false`)

flyway.locations

CSV locations to scan recursively for migrations. Supported prefixes `classpath` and `filesystem` (default: `classpath:db/migration`).

flyway.connect-retries

The maximum number of retries when attempting to connect (default: 0)

flyway.schemas

CSV case-sensitive list of schemas managed (default: none)

flyway.table

The name of Flyway's schema history table (default: `flyway_schema_history`)

flyway.sql-migration-prefix

Prefix for versioned SQL migrations (default: `v`)

`flyway.repeatable-sql-migration-prefix::` Prefix for repeatable SQL migrations (default: `R`)

flyway.baseline-on-migrate

Only migrations above **baseline-version** will then be applied

flyway.baseline-version

Version to tag an existing schema with when executing baseline (default: 1)

flyway.baseline-description

Description to tag an existing schema with when executing baseline (default: `Flyway Baseline`)

Multiple Datasources

To use multiple datasource in Flyway you just need to add the datasource name just after the `flyway` property:

```
quarkus.datasource.users.url=jdbc:h2:tcp://localhost/mem:users
quarkus.datasource.inventory.url=jdbc:h2:tcp://localhost/mem:inventory
# ...

quarkus.flyway.users.schemas=USERS_TEST_SCHEMA
quarkus.flyway.inventory.schemas=INVENTORY_TEST_SCHEMA
# ...
```

Hibernate Search

Quarkus integrates with Elasticsearch to provide a full-featured full-text search using Hibernate Search API.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-hibernate-search-elasticsearch"
```

You need to annotate your model with Hibernate Search API to index it:


```
@Entity
@Indexed
public class Author extends PanacheEntity {

    @FullTextField(analyzer = "english")
    public String bio;

    @FullTextField(analyzer = "name")
    @KeywordField(name = "firstName_sort",
        sortable = Sortable.YES,
        normalizer = "sort")
    public String firstName;

    @OneToMany
    @IndexedEmbedded
    public List<Book> books;

}
```

 It is not mandatory to use Panache.

You need to define the analyzers and normalizers defined in annotations. You only need to implement `ElasticsearchAnalysisConfigurer` interface and configure it.

```
public class MyQuarkusAnalysisConfigurer
    implements ElasticsearchAnalysisConfigurer {

    @Override
    public void configure(
        ElasticsearchAnalysisDefinitionContainerContext ctx)
    {
        ctx.analyzer("english").custom()
            .withTokenizer("standard")
            .withTokenFilters("asciifolding",
                "lowercase", "porter_stem");


        ctx.normalizer("sort").custom()
            .withTokenFilters("asciifolding", "lowercase");
    }
}
```

Use Hibernate Search in REST service:


```
public class LibraryResource {

    @Inject
    EntityManager em;

    @Transactional
    public List<Author> searchAuthors(
        @QueryParam("pattern") String pattern) {
        return Search.getSession(em)
            .search(Author.class)
            .predicate(f ->
                pattern == null || pattern.isEmpty() ?
                    f.matchAll() :
                    f.simpleQueryString()
                        .onFields("firstName",
                            "lastName", "books.title")
                        .matching(pattern)
                )
            .sort(f -> f.byField("lastName_sort"))
            .then().byField("firstName_sort")
            .fetchHits();
    }
}
```



When not using Hibernate ORM, index data using `Search.getSession(em).createIndexer().startAndWait()` at startup time.

Configure the extension in `application.properties`:

```
quarkus.hibernate-search.elasticsearch.version=7
quarkus.hibernate-search.elasticsearch.
    analysis-configurer=MyQuarkusAnalysisConfigurer
quarkus.hibernate-search.elasticsearch.
    automatic-indexing.synchronization-strategy=searchable
quarkus.hibernate-search.elasticsearch.
    index-defaults.lifecycle.strategy=drop-and-create
quarkus.hibernate-search.elasticsearch.
    index-defaults.lifecycle.required-status=yellow
```

List of Hibernate-Elasticsearch properties prefixed with `quarkus.hibernate-search.elasticsearch`:

Parameter	Description
backends	Map of configuration of additional backends.
version	Version of Elasticsearch
analysis-configurer	Class or name of the neab used to configure.
hosts	List of Elasticsearch servers hosts.

Parameter	Description
username	Username for auth.
password	Password for auth.
connection-timeout	Duration of connection timeout.
max-connections	Max number of connections to servers.
max-connections-per-route	Max number of connections to server.
indexes	Per-index specific configuration.
discovery.enabled	Enables automatic discovery.
discovery.refresh-interval	Refresh interval of node list.
discovery.default-scheme	Scheme to be used for the new nodes.
automatic-indexing.synchronization-strategy	Status for which you wait before considering the operation completed (queued,committed or searchable).
automatic-indexing.enable-dirty-check	When enabled, re-indexing of is skipped if the changes are on properties that are not used when indexing.
index-defaults.lifecycle.strategy	Index lifecycle (none, validate, update, create, drop-and-create, drop-abd-create-drop)
index-defaults.lifecycle.required-status	Minimal cluster status (green, yellow, red)
index-defaults.lifecycle.required-status-wait-timeout	Waiting time before failing the bootstrap.
index-defaults.refresh-after-write	Set if index should be refreshed after writes.

Possible annotations:

Parameter	Description
@Indexed	Register entity as full text index
@FullTextField	Full text search. Need to set an analyzer to split tokens.
@KeywordField	The string is kept as one single token but can be normalized.
IndexedEmbedded	Include the Book fields into the Author index.
@ContainerExtraction	Sets how to extract a value from container, e.g from a <code>Map</code> .
@DocumentId	Map an unusual entity identifier to a document identifier.
@GenericField	Full text index for any supported type.
@IdentifierBridgeRef	Reference to the identifier bridge to use for a <code>@DocumentId</code> .
@IndexingDependency	How a dependency of the indexing process to a property should affect automatic reindexing.
@ObjectPath	
@ScaledNumberField	For <code>java.math.BigDecimal</code> or <code>java.math.BigInteger</code> that you need higher precision.

Amazon DynamoDB

Quarkus integrates with Amazon DynamoDB:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-amazon-dynamodb"

@Inject
DynamoDbClient dynamoDB;

@Inject
DynamoDbAsyncClient dynamoDB;
```

```
quarkus.dynamodb.region=
  eu-central-1
quarkus.dynamodb.endpoint-override=
  http://localhost:8000
quarkus.dynamodb.credentials.type=STATIC
quarkus.dynamodb.credentials.static-provider
  .access-key-id=test-key
quarkus.dynamodb.credentials.static-provider
  .secret-access-key=test-secret
```

If you want to work with an AWS account, you’d need to set it with:

```
quarkus.dynamodb.region=<YOUR_REGION>
quarkus.dynamodb.credentials.type=DEFAULT
```

DEFAULT credentials provider chain:

- System properties `aws.accessKeyId`, `aws.secretKey`
- Env. Variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`
- Credentials profile `~/.aws/credentials`
- Credentials through the Amazon EC2 container service if the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` set
- Credentials through Amazon EC2 metadata service.

Configuration parameters prefixed with `quarkus.dynamodb:`

Parameter	Default	Description
<code>enable-endpoint-discovery</code>	<code>false</code>	Endpoint discovery for a service API that supports endpoint discovery.
<code>endpoint-override</code>		Configure the endpoint with which the SDK should communicate.
<code>api-call-timeout</code>		Time to complete an execution.
<code>interceptors</code>		List of class interceptors.

Configuration parameters prefixed with `quarkus.dynamodb.aws:`

Parameter	Default	Description
<code>region</code>		Region that hosts DynamoDB.

Parameter	Default	Description
<code>credentials.type</code>	<code>DEFAULT</code>	Credentials that should be used <code>DEFAULT</code> , <code>STATIC</code> , <code>SYSTEM_PROPERTY</code> , <code>ENV_VARIABLE</code> , <code>PROFILE</code> , <code>CONTAINER</code> , <code>INSTANCE_PROFILE</code> , <code>PROCESS</code> , <code>ANONYMOUS</code>

Credentials specific parameters prefixed with `quarkus.dynamodb.aws.credentials:`

Parameter	Default	Description
<code>DEFAULT</code>		
<code>default-provider.async-credential-update-enabled</code>	<code>false</code>	Should fetch credentials async.
<code>default-provider.reuse-last-provider-enabled</code>	<code>true</code>	Should reuse the last successful credentials.

STATIC		
<code>static-provider.access-key-id</code>		AWS access key id.
<code>static-provider.secret-access-key</code>		AWS secret access key.

PROFILE		
<code>profile-provider.profile-name</code>	<code>default</code>	The name of the profile to use.

PROCESS		
<code>process-provider.command</code>		Command to execute to retrieve credentials.
<code>process-provider.process-output-limit</code>	<code>1024</code>	Max bytes to retrieve from process.

Parameter	Default	Description
<code>process-provider.credential-refresh-threshold</code>	<code>PT15S</code>	The amount of time between credentials expire and credentials refreshed.
<code>process-provider.async-credential-update-enabled</code>	<code>false</code>	Should fetch credentials async.

In case of synchronous client, the next parameters can be configured prefixed by `quarkus.dynamodb.sync-client:`

Parameter	Default	Description
<code>connection-acquisition-timeout</code>	<code>10S</code>	Connection acquisition timeout.
<code>connection-max-idle-time</code>	<code>60S</code>	Max time to connection to be opened.
<code>connection-timeout</code>		Connection timeout.
<code>connection-time-to-live</code>	<code>0</code>	Max time connection to be open.
<code>socket-timeout</code>	<code>30S</code>	Time to wait for data.
<code>max-connections</code>	<code>50</code>	Max connections.
<code>expect-continue-enabled</code>	<code>true</code>	Client send an HTTP <code>expect-continue</code> handshake.
<code>use-idle-connection-reaper</code>	<code>true</code>	Connections in pool should be closed asynchronously.
<code>proxy.endpoint</code>		Endpoint of the proxy server.
<code>proxy.enabled</code>	<code>false</code>	Enables HTTP proxy.
<code>proxy.username</code>		Proxy username.
<code>proxy.password</code>		Proxy password.

Parameter	Default	Description	Parameter	Default	Description	Parameter	Default	Description
proxy.ntlm-domain		For NTLM, domain name.	use-idle-connection-reaper	true	Connections in pool should be closed asynchronously.	event-loop.thread-name-prefix	aws-java-sdk-NettyEventLoop	Prefix of thread names.
proxy.ntlm-workstation		For NTLM, workstation name.	read-timeout	30S	Read timeout.			
proxy.preemptive-basic-authentication-enabled		Authenticate preemptively.	write-timeout	30S	Write timeout.			
proxy.non-proxy-hosts		List of non proxy hosts.	proxy.endpoint		Endpoint of the proxy server.			
tls-managers-provider.type	system-property	TLS manager: none, system-property, file-store	proxy.enabled	false	Enables HTTP proxy.			
tls-managers-provider.file-store.path		Path to key store.	proxy.non-proxy-hosts		List of non proxy hosts.			
tls-managers-provider.file-store.type		Key store type.	tls-managers-provider.type	system-property	TLS manager: none, system-property, file-store			
tls-managers-provider.file-store.password		Key store password.	tls-managers-provider.file-store.path		Path to key store.			
			tls-managers-provider.file-store.type		Key store type.			
			tls-managers-provider.file-store.password		Key store password.			

In case of asynchronous client, the next parameters can be configured prefixed by `quarkus.dynamodb.async-client`:

Parameter	Default	Description	Parameter	Default	Description	Parameter	Default	Description
connection-acquisition-timeout	10S	Connection acquisition timeout.	ssl-provider		SSL Provider (jdk, openssl, openssl-refcnt).	pool.log-leaked-sessions	false	Enable leaked sessions logging.
connection-max-idle-time	60S	Max time to connection to be opened.	protocol	HTTP_1_1	Sets the HTTP protocol.	pool.max-connection-pool-size	100	Max amount of connections.
connection-timeout		Connection timeout.	max-http2-streams		Max number of concurrent streams.	pool.max-connection-lifetime	1H	Pooled connections older will be closed and removed from the pool.
connection-time-to-live	0	Max time connection to be open.	event-loop.override	false	Enable custom event loop conf.	pool.connection-acquisition-timeout	1M	Timeout for connection adquisition.
max-concurrency	50	Max number of concurrent connections.	event-loop.number-of-threads		Number of threads to use in event loop.			

Parameter	Default	Description
pool.idle-time-before-connection-test	-1	Pooled connections idled in the pool for longer than this timeout will be tested before they are used.

As Neo4j uses SSL communication by default, to create a native executable you need to compile with next options GraalVM options:

```
-H:EnableURLProtocols=http,https --enable-all-security-services -H:+JNI
```

And Quarkus Maven Plugin with next configuration:

```
<artifactId>quarkus-maven-plugin</artifactId>
<executions>
  <execution>
    <id>native-image</id>
    <goals>
      <goal>native-image</goal>
    </goals>
    <configuration>
      <enableHttpUrlHandler>true
      </enableHttpUrlHandler>
      <enableHttpsUrlHandler>true
      </enableHttpsUrlHandler>
      <enableAllSecurityServices>true
      </enableAllSecurityServices>
      <enableJni>true</enableJni>
    </configuration>
  </execution>
</executions>
```

Alternatively, and as a not recommended way in production, you can disable SSL and Quarkus will disable Bolt SSL as well.

```
quarkus.ssl.native=false.
```

If you are using Neo4j 4.0, you can use fully reactive. Add next dependency management and dependency:

```
bom:Californium-SR4:pom:import
io.projectreactor:reactor-core.
```

```
public Publisher<String> get() {
    return Flux.using(driver::rxSession, ...);
}
```

MongoDB Client

Quarkus integrates with MongoDB:

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

```
@Inject
com.mongodb.client.MongoClient client;

@Inject
io.quarkus.mongodb.ReactiveMongoClient client;
```

```
quarkus.mongodb.connection-string=mongodb://localhost:27018
quarkus.mongodb.write-concern.journal=false
```

quarkus.mongodb as prefix is skipped in the next table.

Parameter	Type	Description
connection-string	String	MongoDB connection URI.
hosts	List<String>	Addresses passed as host:port.
application-name	String	Application name.
max-pool-size	Int	Maximum number of connections.
min-pool-size	Int	Minimum number of connections.
max-connection-idle-time	Duration	Idle time of a pooled connection.
max-connection-life-time	Duration	Life time of pooled connection.
wait-queue-timeout	Duration	Maximum wait time for new connection.
maintenance-frequency	Duration	Time period between runs of maintenance job.
maintenance-initial-delay	Duration	Time to wait before running the first maintenance job.
wait-queue-multiple	Int	Multiplied with max-pool-size gives max numer of threads waiting.

Parameter	Type	Description
connection-timeout	Duration	
socket-timeout	Duration	
tls-insecure	boolean [false]	Insecure TLS.
tls	boolean [false]	Enable TLS
replica-set-name	String	Implies hosts given are a seed list.
server-selection-timeout	Duration	Time to wait for server selection.
local-threshold	Duration	Minimum ping time to make a server eligible.
heartbeat-frequency	Duration	Frequency to determine the state of servers.
read-preference	primary, primaryPreferred, secondary, secondaryPreferred, nearest	Read preferences.
max-wait-queue-size	Int	Max number of concurrent operations allowed to wait.
write-concern.safe	boolean [true]	Ensures are writes are ack.
write-concern.journal	boolean [true]	Journal writing aspect.
write-concern.w	String	Value to all write commands.
write-concern.retry-writes	boolean [false]	Retry writes if network fails.
write-concern.w-timeout	Duration	Timeout to all write commands.

Parameter	Type	Description
<code>credentials.username</code>	<code>String</code>	Username.
<code>credentials.password</code>	<code>String</code>	Password.
<code>credentials.auth-mechanism</code>	<code>MONGO-CR</code> , <code>GSSAPI</code> , <code>PLAIN</code> , <code>MONGODB-X509</code>	
<code>credentials.auth-source</code>	<code>String</code>	Source of the authentication credentials.
<code>credentials.auth-mechanism-properties</code>	<code>Map<String, String></code>	Authentication mechanism properties.

MongoDB Panache

You can also use the Panache framework to write persistence part when using MongoDB.

```
./mvnw quarkus:add-extension
-Dextensions="mongodb-panache"
```

MongoDB configuration comes from MongoDB Client section.

```
@MongoEntity(collection="ThePerson")
public class Person extends PanacheMongoEntity {
    public String name;

    @BsonProperty("birth")
    public LocalDate birthDate;

    public Status status;
}
```

Possible annotations in fields: `@BsonId` (for custom ID), `@BsonProperty` and `@BsonIgnore`.

Important: `@MongoEntity` is optional.

Methods provided are similar of the ones shown in Persistence section.

```
person.persist();
person.update();
person.delete();

List<Person> allPersons = Person.listAll();
person = Person.findById(personId);
List<Person> livingPersons = Person.list("status", Status.Alive);
List<Person> persons = Person.list(Sort.by("name").and("birth"));

long countAll = Person.count();

Person.delete("status", Status.Alive);
```

All `list` methods have equivalent `stream` versions.

Pagination

You can also use pagination:

```
PanacheQuery<Person> livingPersons =
    Person.find("status", Status.Alive);
livingPersons.page(Page.ofSize(25));

// get the first page
List<Person> firstPage = livingPersons.list();
// get the second page
List<Person> secondPage = livingPersons.nextPage().list();
```

Queries

Native MongoDB queries are supported (if they start with `{` or `org.bson.Document` instance) as well as Panache Queries. Panache Queries equivalence in MongoDB:

- `firstname = ?1 and status = ?2` \rightarrow `{'firstname': ?1, 'status': ?2}`
- `amount > ?1 and firstname != ?2` \rightarrow `{'amount': {'$gt': ?1}, 'firstname': {'$ne': ?2}}`
- `lastname like ?1` \rightarrow `{'lastname': {'$regex': ?1}}`
- `lastname is not null` \rightarrow `{'lastname': {'$exists': true}}`



PanacheQL refers to the Object parameter name but native queries refer to MongoDB field names.

DAO pattern

```
@ApplicationScoped
public class PersonRepository
    implements PanacheMongoRepository<Person> {
}
```

If entities are defined in external JAR, you need to enable in these projects the `Jandex` plugin in project.

```
<plugin>
  <groupId>org.jboss.jandex</groupId>
  <artifactId>jandex-maven-plugin</artifactId>
  <version>1.0.3</version>
  <executions>
    <execution>
      <id>make-index</id>
      <goals>
        <goal>jandex</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.jboss</groupId>
      <artifactId>jandex</artifactId>
      <version>2.1.1.Final</version>
    </dependency>
  </dependencies>
</plugin>
```

Reactive Programming

Quarkus implements MicroProfile Reactive spec and uses RXJava2 to provide reactive programming model.

```
./mvnw quarkus:add-extension
-Dextensions="
    io.quarkus:quarkus-smallrye-reactive-streams-operators"
```

Asynchronous HTTP endpoint is implemented by returning Java `CompletionStage`. You can create this class either manually or using MicroProfile Reactive Streams spec:

```
@GET
@Path("/reactive")
@Produces(MediaType.TEXT_PLAIN)
public CompletionStage<String> getHello() {
    return ReactiveStreams.of("h", "e", "l", "l", "o")
        .map(String::toUpperCase)
        .toList()
        .run()
        .thenApply(list -> list.toString());
}
```

Creating streams is also easy, you just need to return `Publisher` object.

```
@GET
@Path("/stream")
@Produces(MediaType.SERVER_SENT_EVENTS)
public Publisher<String> publishers() {
    return Flowable
        .interval(500, TimeUnit.MILLISECONDS)
        .map(s -> atomicInteger.getAndIncrement())
        .map(i -> Integer.toString(i));
}
```

Reactive Messaging

Quarkus relies on MicroProfile Reactive Messaging spec to implement reactive messaging streams.

```
mvn quarkus:add-extension
-Dextensions="
    io.quarkus:quarkus-smallrye-reactive-messaging"
```

You can just start using in-memory streams by using `@Incoming` to produce data and `@Outgoing` to consume data.

Produce every 5 seconds one piece of data.

```
@ApplicationScoped
public class ProducerData {

    @Outgoing("my-in-memory")
    public Flowable<Integer> generate() {
        return Flowable.interval(5, TimeUnit.SECONDS)
            .map(tick -> random.nextInt(100));
    }
}
```

If you want to dispatch to all subscribers you can annotate the method with `@Broadcast`.

Consumes generated data from `my-in-memory` stream.

```
@ApplicationScoped
public class ConsumerData {
    @Incoming("my-in-memory")
    public void randomNumber(int randomNumber) {
        System.out.println("Received " + randomNumber);
    }
}
```

You can also inject an stream as a field:

```
@Inject
@Stream("my-in-memory") Publisher<Integer> randomNumbers;
```

```
@Inject @Stream("generated-price")
Emitter<String> emitter;
```

Patterns

REST API → Message

```
@Inject @Stream("in")
Emitter<String> emitter;

emitter.send(message);
```

Message → Message

```
@Incoming("in")
@Outgoing("out")
public String process(String in) {
}
```

Message → SSE

```
@Inject @Stream("out")
Publisher<String> result;

@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
public Publisher<String> stream() {
    return result;
}
```

Message → Business Logic

```
@ApplicationScoped
public class ReceiverMessages {
    @Incoming("prices")
    public void print(String price) {
    }
}
```

Possible implementations are:

In-Memory

If the stream is not configured then it is assumed to be an in-memory stream, if not then stream type is defined by `connector` field.

Kafka

To integrate with Kafka you need to add next extensions:

```
mvn quarkus:add-extension
-Dextensions="
    io.quarkus:quarkus-smallrye-reactive-messaging-kafka"
```

Then `@Outgoing`, `@Incoming` or `@Stream` can be used.

Kafka configuration schema: `mp.messaging.[outgoing|incoming].{stream-name}.<property>=<value>`.

The `connector` type is `smallrye-kafka`.

```
mp.messaging.outgoing.generated-price.connector=
    smallrye-kafka
mp.messaging.outgoing.generated-price.topic=
    prices
mp.messaging.outgoing.generated-price.bootstrap.servers=
    localhost:9092
mp.messaging.outgoing.generated-price.value.serializer=
    org.apache.kafka.common.serialization.IntegerSerializer

mp.messaging.incoming.prices.connector=
    smallrye-kafka
mp.messaging.incoming.prices.value.deserializer=
    org.apache.kafka.common.serialization.IntegerDeserializ
er
```


A complete list of supported properties are in Kafka site. For the producer and for consumer

JSON-B Serializer/Deserializer

You can use JSON-B to serialize/deserialize objects.

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

To serialize you can use `io.quarkus.kafka.client.serialization.JsonbSerializer`.

To deserialize you need to extend and provide a type.

```
public class BeerDeserializer
    extends JsonbDeserializer<Beer> {

    public BeerDeserializer() {
        super(Beer.class);
    }

}
```

AMQP

To integrate with AMQP you need to add next extensions:

```
./mvnw quarkus:add-extension
-Dextensions="reactive-messaging-amqp"
```

Then `@Outgoing`, `@Incoming` or `@Stream` can be used.

AMQP configuration schema: `mp.messaging.[outgoing|incoming].{stream-name}.<property>=<value>`. Special properties `amqp-username` and `amqp-password` are used to configure AMQP broker credentials.

The connector type is `smallrye-amqp`.

```
amqp-username=quarkus
amqp-password=quarkus
# write
mp.messaging.outgoing.generated-price.connector=
    smallrye-amqp
mp.messaging.outgoing.generated-price.address=
    prices
mp.messaging.outgoing.generated-price.durable=
    true
# read
mp.messaging.incoming.prices.connector=
    smallrye-amqp
mp.messaging.incoming.prices.durable=
    true
```

A complete list of supported properties for AMQP.

MQTT

To integrate with MQTT you need to add next extensions:

```
./mvnw quarkus:add-extension
-Dextensions="vertx, smallrye-reactive-streams-operator
    smallrye-reactive-messaging"
```

And add `io.smallrye.reactive:smallrye-reactive-messaging-mqtt-1.0:0.0.10` dependency in your build tool.

Then `@Outgoing`, `@Incoming` or `@Stream` can be used.

MQTT configuration schema: `mp.messaging.[outgoing|incoming].{stream-name}.<property>=<value>`.

The connector type is `smallrye-mqtt`.

```
mp.messaging.outgoing.topic-price.type=
    smallrye-mqtt
mp.messaging.outgoing.topic-price.topic=
    prices
mp.messaging.outgoing.topic-price.host=
    localhost
mp.messaging.outgoing.topic-price.port=
    1883
mp.messaging.outgoing.topic-price.auto-generated-client-id=
    true

mp.messaging.incoming.prices.type=
    smallrye-mqtt
mp.messaging.incoming.prices.topic=
    prices
mp.messaging.incoming.prices.host=
    localhost
mp.messaging.incoming.prices.port=
    1883
mp.messaging.incoming.prices.auto-generated-client-id=
    true
```

Kafka Streams

Create streaming queries with the Kafka Streams API.

```
./mvnw quarkus:add-extension
-Dextensions="kafka-streams"
```

All we need to do for that is to declare a CDI producer method which returns the Kafka Streams `org.apache.kafka.streams.Topology`:

```
@ApplicationScoped
public class TopologyProducer {
    @Produces
    public Topology buildTopology() {
        org.apache.kafka.streams.StreamsBuilder.StreamsBuild
        der
            builder = new StreamsBuilder();
            // ...
            builder.stream()
                .join()
                // ...
                .toStream()
                .to();
            return builder.build();
        }
    }
```

Previous example produces content to another stream. If you want to write interactive queries, you can use Kafka streams.

```
@Inject
KafkaStreams streams;

return streams
    .store("stream", QueryableStoreTypes.keyValueStore
());
```

The Kafka Streams extension is configured via the Quarkus configuration file `application.properties`.

```
quarkus.kafka-streams.bootstrap-servers=localhost:9092
quarkus.kafka-streams.application-id=temperature-aggregator
quarkus.kafka-streams.application-server=${hostname}:8080
quarkus.kafka-streams.topics=weather-stations,temperature-v
alues

kafka-streams.cache.max.bytes.buffering=10240
kafka-streams.commit.interval.ms=1000
```

IMPORTANT: All the properties within the `kafka-streams` namespace are passed through as-is to the Kafka Streams engine. Changing their values requires a rebuild of the application.

Reactive PostgreSQL Client

You can use Reactive PostgreSQL to execute queries to PostreSQL database in a reactive way, instead of using JDBC way.

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

Database configuration is the same as shown in Persistence section, but URL is different as it is not a *jdbc*.

```
quarkus.datasource.url=
  vertx-reactive:postgres://host:5431/db
```

Then you can inject `io.vertx.axle.pgclient.PgPool` class.

```
@Inject
PgPool client;

CompletionStage<JSONArray> =
  client.query("SELECT * FROM table")
    .thenApply(rowSet -> {
      JSONArray jsonArray = new JSONArray();
      PgIterator iterator = rowSet.iterator();
      return jsonArray;
    })
```

Reactive MySQL Client

You can use Reactive MySQL to execute queries to MySQL database in a reactive way, instead of using JDBC way.

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

Database configuration is the same as shown in Persistence section, but URL is different as it is not a *jdbc*.

```
quarkus.datasource.url=
  vertx-reactive:mysql://localhost:3306/db
```

Then you can inject `io.vertx.axle.mysqlclient.MySQLPool` class.

```
@Inject
MySQLPool client;

public static CompletionStage<Fruit> findById(
  MySQLPool c, Long id) {
  return c.preparedQuery("SELECT name FROM fruits WHERE i
d = $1",
    Tuple.of(id))
    .thenApply(RowSet::iterator)
    .thenApply(iterator -> iterator.hasNext() ?
      from(iterator.next()) : null);
}
```

ActiveMQ Artemis

Quarkus uses Reactive Messaging to integrate with messaging systems, but in case you need deeper control when using Apache ActiveMQ Artemis there is also an extension:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-artemis-core"
```

And then you can inject `org.apache.activemq.artemis.api.core.client.ServerLocator` instance.

```
@ApplicationScoped
public class ArtemisConsumerManager {

  @Inject
  ServerLocator serverLocator;

  private ClientSessionFactory connection;

  @PostConstruct
  public void init() throws Exception {
    connection = serverLocator.createSessionFactory();
  }
}
```

And configure `ServerLocator` in `application.properties`:

```
quarkus.artemis.url=tcp://localhost:61616
```

You can configure ActiveMQ Artemis in `application.properties` file by using next properties prefixed with `quarkus`:

Parameter	Default	Description
<code>artemis.url</code>		Connection URL
<code>artemis.username</code>		Username for authentication.
<code>artemis.password</code>		Password for authentication.

Artemis JMS

If you want to use JMS with Artemis, you can do it by using its extension:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-artemis-jms"
```

And then you can inject `javax.jms.ConnectionFactory`:

```
@ApplicationScoped
public class ArtemisConsumerManager {

  @Inject
  ConnectionFactory connectionFactory;

  private Connection connection;

  @PostConstruct
  public void init() throws JMSEException {
    connection = connectionFactory.createConnection();
    connection.start();
  }
}
```

INFO: Configuration options are the same as Artemis core.

RBAC

You can set RBAC using annotations or in `application.properties`.

Annotations

You can define roles by using `javax.annotation.security.RolesAllowed` annotation.

```
@RolesAllowed("Subscriber")
```

You can use `io.quarkus.security.Authenticated` as a shortcut of `@RolesAllowed("*")`.

To alter RBAC behaviour there are two configuration properties:

```
quarkus.security.deny-unannotated=true
```

Configuration options:

Parameter	Default	Description
<code>quarkus.jaxrs.deny-uncovered</code>	<code>false</code>	If true denies by default to all JAX-RS endpoints.
<code>quarkus.security.deny-unannotated</code>	<code>false</code>	If true denies by default all CDI methods and JAX-RS endpoints.

File Configuration

Defining RBAC in `application.properties` instead of using annotations.

```
quarkus.http.auth.policy.role-policy1.roles-allowed=user,admin
quarkus.http.auth.permission.roles1.paths=/roles-secured/*,/other/*,/api/*
quarkus.http.auth.permission.roles1.policy=role-policy1

quarkus.http.auth.permission.permit1.paths=/public/*
quarkus.http.auth.permission.permit1.policy=permit
quarkus.http.auth.permission.permit1.methods=GET

quarkus.http.auth.permission.deny1.paths=/forbidden
quarkus.http.auth.permission.deny1.policy=deny
```

You need to provide permissions set by using the `roles-allowed` property or use the built-in ones `deny`, `permit` or `authenticated`.

JWT

Quarkus implements MicroProfile JWT RBAC spec.

```
mvn quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-jwt"
```

Minimum JWT required claims: `typ`, `alg`, `kid`, `iss`, `sub`, `exp`, `iat`, `jti`, `upn`, `groups`.

You can inject token by using `JsonWebToken` or a claim individually by using `@Claim`.

```
@Inject
JsonWebToken jwt;

@Inject
@Claim(standard = Claims.preferred_username)
String name;

@Inject
@Claim("groups")
Set<String> groups;
```

Set of supported types: `String`, `Set<String>`, `Long`, `Boolean`, `javax.json.JsonValue`, `Optional`, `org.eclipse.microprofile.jwt.ClaimValue`.

And configuration in `src/main/resources/application.properties`:

```
mp.jwt.verify.publickey.location=
META-INF/resources/publicKey.pem
mp.jwt.verify.issuer=
https://quarkus.io/using-jwt-rbac
```

Configuration options:

Parameter	Default	Description
<code>quarkus.smallrye-jwt.enabled</code>	<code>true</code>	Determine if the jwt extension is enabled.
<code>quarkus.smallrye-jwt.realm-name</code>	Quarkus-JWT	Name to use for security realm.
<code>quarkus.smallrye-jwt.auth-mechanism</code>	MP-JWT	Authentication mechanism.

Parameter	Default	Description
<code>mp.jwt.verify.publickey</code>	<code>none</code>	Public Key text itself to be supplied as a string.
<code>mp.jwt.verify.publickey.location</code>	<code>none</code>	Relative path or URL of a public key.
<code>mp.jwt.verify.issuer</code>	<code>none</code>	<code>iss</code> accepted as valid.

Supported public key formats:

- PKCS#8 PEM
- JWK
- JWKS
- JWK Base64 URL
- JWKS Base64 URL

To send a token to server-side you should use `Authorization` header: `curl -H "Authorization: Bearer eyJraWQiOi..."`.

To inject claim values, the bean must be `@RequestScoped` CDI scoped. If you need to inject claim values in scope with a lifetime greater than `@RequestScoped` then you need to use `javax.enterprise.inject.Instance` interface.

```
@Inject
@Claim(standard = Claims.iat)
private Instance<Long> providerIAT;
```

RBAC

JWT `groups` claim is directly mapped to roles to be used in security annotations.

```
@RolesAllowed("Subscriber")
```

OpenId Connect

Quarkus can use OpenId Connect or OAuth 2.0 authorization servers such as Keycloak to protect resources using bearer token issued by Keycloak server.

```
mvn quarkus:add-extension
-Dextensions="using-openid-connect"
```

You can also protect resources with security annotations.



```
@GET
@RolesAllowed("admin")
```


Configure application to Keycloak service in `application.properties` file.

```
quarkus.oidc.realm=quarkus
quarkus.oidc.auth-server-url=http://localhost:8180/auth
quarkus.oidc.resource=backend-service
quarkus.oidc.bearer-only=true
quarkus.oidc.credentials.secret=secret
```

Configuration options with `quarkus.oidc` prefix:

Parameter	Default	Description
<code>auth-server-url</code>		The base URL of the OpenID Connect (OIDC) server
<code>introspection-path</code>		Relative path of the RFC7662 introspection service
<code>jwks-path</code>		Relative path of the OIDC service returning a JWK set
<code>public-key</code>		Public key for the local JWT token verification
<code>client-id</code>		The client-id of the application.
<code>credentials.secret</code>		The client secret

- 

With Keycloak OIDC server `https://host:port/auth/realms/{realm}` where `{realm}` has to be replaced by the name of the Keycloak realm.
- 

You can use `quarkus.http.cors` property to enable consuming form different domain.

OAuth2

Quarkus integrates with OAuth2 to be used in case of opaque tokens (none JWT) and its validation against an introspection endpoint.

```
mvn quarkus:add-extension
-Dextensions="security-oauth2"
```

And configuration in `src/main/resources/application.properties`:

```
quarkus.oauth2.client-id=client_id
quarkus.oauth2.client-secret=secret
quarkus.oauth2.introspection-url=http://oauth-server/introspect
```

And you can map roles to be used in security annotations.

```
@RolesAllowed("Subscriber")
```

Configuration options:

Parameter	Default	Description
<code>quarkus.oauth2.enabled</code>	<code>true</code>	Determine if the OAuth2 extension is enabled.
<code>quarkus.oauth2.client-id</code>		The OAuth2 client id used to validate the token.
<code>quarkus.oauth2.client-secret</code>		The OAuth2 client secret used to validate the token.
<code>quarkus.oauth2.introspection-url</code>		URL used to validate the token and gather the authentication claims.
<code>quarkus.oauth2.role-claim</code>	<code>scope</code>	The claim that is used in the endpoint response to load the roles

Authenticating via HTTP

HTTP basic auth is enabled by the `quarkus.http.auth.basic=true` property.

HTTP form auth is enabled by the `quarkus.http.auth.form.enabled=true` property.

Then you need to add `elytron-security-properties-file` or `elytron-security-jdbc`.

Security with Properties File

You can also protect endpoints and store identities (user, roles) in the file system.

```
mvn quarkus:add-extension
-Dextensions="elytron-security-properties-file"
```

You need to configure the extension with users and roles files:

And configuration in `src/main/resources/application.properties`:

```
quarkus.security.users.file.enabled=true
quarkus.security.users.file.users=test-users.properties
quarkus.security.users.file.roles=test-roles.properties
quarkus.security.users.file.auth-mechanism=BASIC
quarkus.security.users.file.realm-name=MyRealm
quarkus.security.users.file.plain-text=true
```

Then `users.properties` and `roles.properties`:

```
scott=jb0ss
jdoe=p4ssw0rd
```

```
scott=Admin,admin,Tester,user
jdoe=NoRolesUser
```

IMPORTANT: If `plain-text` is set to `false` (or omitted) then passwords must be stored in the form MD5 (`username:realm:password`).

Elytron File Properties configuration properties. Prefix `quarkus.security.users` is skipped.

Parameter	Default	Description
<code>file.enabled</code>	<code>false</code>	The file realm is enabled
<code>file.auth-mechanism</code>	<code>BASIC</code>	The authentication mechanism
<code>file.realm-name</code>	<code>Quarkus</code>	The authentication realm name
<code>file.plain-text</code>	<code>false</code>	If passwords are in plain or in MD5
<code>file.users</code>	<code>users.properties</code>	Classpath resource of user/password
<code>file.roles</code>	<code>roles.properties</code>	Classpath resource of user/role

Embedded Realm

You can embed user/password/role in the same application.properties:

```
quarkus.security.users.embedded.enabled=true
quarkus.security.users.embedded.plain-text=true
quarkus.security.users.embedded.users.scott=jb0ss
quarkus.security.users.embedded.roles.scott=admin,tester,user
quarkus.security.users.embedded.auth-mechanism=BASIC
```

IMPORTANT: If plain-text is set to false (or omitted) then passwords must be stored in the form MD5 (username:`realm`:`password`).

Prefix quarkus.security.users.embedded is skipped.

Parameter	Default	Description
file.enabled	false	The file realm is enabled
file.auth-mechanism	BASIC	The authentication mechanism
file.realm-name	Quarkus	The authentication realm name
file.plain-text	false	If passwords are in plain or in MD5
file.users.*		* is user and value is password
file.roles.*		* is user and value is role

Security with a JDBC Realm

You can also protect endpoints and store identities in a database.

```
mvn quarkus:add-extension
-Dextensions="elytron-security-jdbc"
```

You still need to add the database driver (ie jdbc-h2).

You need to configure JDBC and Elytron JDBC Realm:

```
quarkus.datasource.url=
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.username=sa
quarkus.datasource.password=sa

quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=
    SELECT u.password, u.role FROM test_user u WHERE u.user=?
quarkus.security.jdbc.principal-query
    .clear-password-mapper.enabled=true
quarkus.security.jdbc.principal-query
    .clear-password-mapper.password-index=1
quarkus.security.jdbc.principal-query
    .attribute-mappings.0.index=2
quarkus.security.jdbc.principal-query
    .attribute-mappings.0.to=groups
```

You need to set the index (1-based) of password and role.

Elytron JDBC Realm configuration properties. Prefix quarkus.security.jdbc is skipped.

Parameter	Default	Description
auth-mechanism	BASIC	The authentication mechanism
realm-name	Quarkus	The authentication realm name
enabled	false	If the properties store is enabled
principal-query.sql		The sql query to find the password
principal-query.datasource		The data source to use
principal-query.clear-password-mapper.enabled	false	If the clear-password-mapper is enabled
principal-query.clear-password-mapper.password-index	1	The index of column containing clear password
principal-query.bcrypt-password-mapper.enabled	false	If the bcrypt-password-mapper is enabled

Parameter	Default	Description
principal-query.bcrypt-password-mapper.password-index	0	The index of column containing password hash
principal-query.bcrypt-password-mapper.hash-encoding	BASE64	A string referencing the password hash encoding (BASE64 or HEX)
principal-query.bcrypt-password-mapper.salt-index	0	The index column containing the Bcrypt salt
principal-query.bcrypt-password-mapper.salt-encoding	BASE64	A string referencing the salt encoding (BASE64 or HEX)
principal-query.bcrypt-password-mapper.iteration-count-index	0	The index column containing the Bcrypt iteration count

For multiple datasources you can use the datasource name in the properties:

```
quarkus.datasource.url=
quarkus.security.jdbc.principal-query.sql=

quarkus.datasource.permissions.url=
quarkus.security.jdbc.principal-query.permissions.sql=
```

Vault

Quarkus integrates with Vault to manage secrets or protecting sensitive data.

```
mvn quarkus:add-extension
-Dextensions="vault"
```

And configuring Vault in application.properties:

```
# vault url
quarkus.vault.url=http://localhost:8200

quarkus.vault.authentication.userpass.username=
  bob
quarkus.vault.authentication.userpass.password=
  sinclair

# path within the kv secret engine
quarkus.vault.secret-config-kv-path=
  myapps/vault-quickstart/config
```

Then you can inject the value configured at `secret/myapps/vault-quickstart/a-private-key`.

```
@ConfigProperty(name = "a-private-key")
String privateKey;
```

You can access the KV engine programmatically:

```
@Inject
VaultKVSecretEngine kvSecretEngine;

kvSecretEngine.readSecret("myapps/vault-quickstart/" + vaultPath).toString();
```

Fetching credentials DB

With the next `kv vault kv put secret/myapps/vault-quickstart/db password=connor`

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

quarkus.datasource.url=
  jdbc:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver=
  org.postgresql.Driver
quarkus.datasource.username=
  sarah
```

No password is set as it is fetched from Vault.

INFO: dynamic database credentials through the `database-credentials-role` property.

Elytron JDBC Realm configuration properties. Prefix `quarkus.vault` is skipped.

Parameter	Default	Description
<code>url</code>		Vault server URL

Parameter	Default	Description	Parameter	Default	Description
<code>authentication.client-token</code>		Vault token to access	<code>tls.ca-cert</code>		Certificate bundle used to validate TLS communications
<code>authentication.app-role.role-id</code>		Role Id for AppRole auth	<code>tls.use-kubernetes-ca-cert</code>	<code>true</code>	TLS will be active
<code>authentication.app-role.secret-id</code>		Secret Id for AppRole auth	<code>connect-timeout</code>	<code>5s</code>	Tiemout to establish a connection
<code>authentication.userpass.username</code>		Username for userpass auth	<code>read-timeout</code>	<code>1s</code>	Request timeout
<code>authentication.userpass.password</code>		Password for userpass auth	<code>credentials-provider."credentials-provider".database-credentials-role</code>		Database credentials role
<code>authentication.kubernetes.role</code>		Kubernetes authentication role	<code>credentials-provider."credentials-provider".kv-path</code>		A path in vault kv store, where we will find the kv-key
<code>authentication.kubernetes.jwt-token-path</code>		Location of the file containing the Kubernetes JWT token	<code>credentials-provider."credentials-provider".kv-key</code>	<code>password</code>	Key name to search in vault path kv-path
<code>renew-grace-period</code>	<code>1H</code>	Renew grace period duration.			
<code>secret-config-cache-period</code>	<code>10M</code>	Vault config cache period			
<code>secret-config-kv-path</code>		Vault path in kv store			
<code>log-confidentiality-level</code>	<code>medium</code>	Used to hide confidential infos. low, medium, high			
<code>kv-secret-engine-version</code>	<code>1</code>	Kv secret engine version			
<code>kv-secret-engine-mount-path</code>	<code>secret</code>	Kv secret engine path			
<code>tls.skip-verify</code>	<code>false</code>	Allows to bypass certificate validation on TLS communications			

JAX-RS

Quarkus uses JAX-RS to define REST-ful web APIs. Under the covers, Rest-EASY is working with Vert.X directly without using any Servlet.

It is **important** to know that if you want to use any feature that implies a `Servlet` (ie Servlet Filters) then you need to add the `quarkus-undertow` extension to switch back to the `Servlet` ecosystem but generally speaking, you don't need to add it as everything else is well-supported.

```
@Path("/book")
public class BookResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> getAllBooks() {}

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createBook(Book book) {}

    @DELETE
    @Path("/{isbn}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(
        @PathParam("isbn") String isbn) {}

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/search")
    public Response searchBook(
        @QueryParam("description") String description) {}
}
```

To get information from request:

Property	Description	Example
@PathParam	Gets content from request URI.	/book/{id} @PathParam("id")
@QueryParam	Gets query parameter.	/book?desc="" @QueryParam("desc")
@FormParam	Gets form parameter.	
@MatrixParam	Get URI matrix parameter.	/book;author=mkyong;country=malaysia

Property	Description	Example
@CookieParam	Gets cookie param by name.	
@HeaderParam	Gets header parameter by name.	

Valid HTTP method annotations provided by the spec are: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@PATCH`, `@HEAD` and `@OPTIONS`.

You can create new annotations that bind to HTTP methods not defined by the spec.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("LOCK")
public @interface LOCK {

    @LOCK
    public void lockIt() {}
}
```

Injecting

Using `@Context` annotation to inject JAX-RS and Servlet information.

```
@GET
public String getBase(@Context UriInfo uriInfo) {
    return uriInfo.getBaseUri();
}
```

Possible injectable objects: `SecurityContext`, `Request`, `Application`, `Configuration`, `Providers`, `ResourceContext`, `ServletConfig`, `ServletContext`, `HttpServletRequest`, `HttpServletResponse`, `HttpHeaders`, `UriInfo`, `SseEventSink` and `Sse`.

HTTP Filters

HTTP request and response can be intercepted to manipulate the metadata (ie headers, parameters, media type, ...) or abort a request. You only need to implement the next `ContainerRequestFilter` and `ContainerResponseFilter` JAX-RS interfaces respectively.

```
@Provider
public class LoggingFilter
    implements ContainerRequestFilter {

    @Context
    UriInfo info;

    @Context
    HttpServletRequest request;

    @Override
    public void filter(ContainerRequestContext context) {
        final String method = context.getMethod();
        final String path = info.getPath();
        final String address = request.getRemoteAddr();
        System.out.println("Request %s %s from IP %s",
            method, path, address);
    }
}
```

Exception Mapper

You can map exceptions to produce a custom output by implementing `ExceptionHandler` interface:

```
@Provider
public class ErrorMapper
    implements ExceptionMapper<Exception> {

    @Override
    public Response toResponse(Exception exception) {
        int code = 500;
        if (exception instanceof WebApplicationException) {
            code = ((WebApplicationException) exception)
                .getResponse().getStatus();
        }
        return Response.status(code)
            .entity(
                Json.createObjectBuilder()
                    .add("error", exception.getMessage())
                    .add("code", code)
                    .build()
            )
            .build();
    }
}
```

Vert.X Filters and Routes

Programmatically

You can also register Vert.X Filters and Router programmatically inside a CDI bean:

```
import io.quarkus.vertx.http.runtime.filters.Filters;
import io.vertx.ext.web.Router;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

@ApplicationScoped
public class MyBean {

    public void filters(
        @Observes Filters filters) {
        filters
            .register(
                rc -> {
                    rc.response()
                        .putHeader("X-Filter", "filter 1");
                    rc.next();
                },
                10);
    }

    public void routes(
        @Observes Router router) {
        router
            .get("/")
            .handler(rc -> rc.response().end("OK"));
    }
}
```

Declarative

You can use `@Route` annotation to use reactive routes and `@RouteFilter` to sue reactive filters in a declarative way:

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-vertx-web"
```

```
@ApplicationScoped
public class MyDeclarativeRoutes {

    @Route(path = "/hello", methods = HttpMethod.GET)
    public void greetings(RoutingContext rc) {
        String name = rc.request().getParam("name");
        if (name == null) {
            name = "world";
        }
        rc.response().end("hello " + name);
    }

    @RouteFilter(20)
    void filter(RoutingContext rc) {
        rc.response().putHeader("X-Filter", "filter 2");
        rc.next();
    }
}
```

GZip Support

You can configure Quarkus to use GZip in the application.properties file using the next properties with quarkus.resteasy suffix:

Parameter	Default	Description
gzip.enabled	false	EnableGZip.
gzip.max-input	10M	Configure the upper limit on deflated request body.

CORS Filter

Quarkus comes with a CORS filter that can be enabled via configuration:

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

Prefix is `quarkus.http.`

Property	Default	Description
cors	false	Enable CORS.
cors.origins	Any request valid.	CSV of origins allowed.
cors.methods	Any method valid.	CSV of methods valid.
cors.headers	Any requested header valid.	CSV of valid allowed headers.
cors.exposed-headers		CSV of valid exposed headers.

Fault Tolerance

Quarkus uses MicroProfile Fault Tolerance spec:

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-fault-tolerance"
```

MicroProfile Fault Tolerance spec uses CDI interceptor and it can be used in several elements such as CDI bean, JAX-RS resource or MicroProfile Rest Client.

To do automatic **retries** on a method:

```
@Path("/api")
@RegisterRestClient
public interface WorldClockService {
    @GET @Path("/json/cet/now")
    @Produces(MediaType.APPLICATION_JSON)
    @Retry(maxRetries = 2)
    WorldClock getNow();
}
```

You can set fallback code in case of an error by using `@Fallback` annotation:

```
@Retry(maxRetries = 1)
@Fallback(fallbackMethod = "fallbackMethod")
WorldClock getNow() {}

public WorldClock fallbackMethod() {
    return new WorldClock();
}
```

`fallbackMethod` must have the same parameters and return type as the annotated method.

You can also set logic into a class that implements `FallbackHandler` interface:

```
public class RecoverFallback
    implements FallbackHandler<WorldClock> {
    @Override
    public WorldClock handle(ExecutionContext context) {
    }
}
```

And set it in the annotation as value `@Fallback(RecoverFallback.class)`.

In case you want to use **circuit breaker** pattern:

```
@CircuitBreaker(requestVolumeThreshold = 4,
    failureRatio=0.75,
    delay = 1000)
WorldClock getNow() {}
```

If 3 (4 x 0.75) failures occur among the rolling window of 4 consecutive invocations then the circuit is opened for 1000 ms and then be back to half open. If the invocation succeeds then the circuit is back to closed again.

You can use **bulkhead** pattern to limit the number of concurrent access to the same resource. If the operation is synchronous it uses a semaphore approach, if it is asynchronous a thread-pool one. When a request cannot be processed `BulkheadException` is thrown. It can be used together with any other fault tolerance annotation.

```
@Bulkhead(5)
@Retry(maxRetries = 4,
      delay = 1000,
      retryOn = BulkheadException.class)
WorldClock getNow() {}
```

Fault tolerance annotations:


Annotation	Properties
@Timeout	unit
@Retry	maxRetries, delay, delayUnit, maxDuration, durationUnit, jitter, jitterDelayUnit, retryOn, abortOn
@Fallback	fallbackMethod
@Bulkhead	waitingTaskQueue (only valid in asynchronous)
@CircuitBreaker	failOn, delay, delayUnit, requestVolumeThreshold, failureRatio, successThreshold
@Asynchronous	

You can override annotation parameters via configuration file using property `[classname/methodname/]annotation/parameter`:

```
org.acme.quickstart.WorldClock/getNow/Retry/maxDuration=30
# Class scope
org.acme.quickstart.WorldClock/Retry/maxDuration=3000
# Global
Retry/maxDuration=3000
```

You can also enable/disable policies using special parameter enabled.

```
org.acme.quickstart.WorldClock/getNow/Retry/enabled=false
# Disable everything except fallback
MP_Fault_Tolerance_NonFallback_Enabled=false
```



MicroProfile Fault Tolerance integrates with MicroProfile Metrics spec. You can disable it by setting `MP_Fault_Tolerance_Metrics_Enabled` to false.

Observability

Health Checks

Quarkus relies on MicroProfile Health spec to provide health checks.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-health"
```

By just adding this extension, an endpoint is registered to `/health` providing a default health check.

```
{
  "status": "UP",
  "checks": [
  ]
}
```

To create a custom health check you need to implement the `HealthCheck` interface and annotate either with `@Readiness` (ready to process requests) or `@Liveness` (is running) annotations.

```
@Readiness
public class DatabaseHealthCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
        HealthCheckResponseBuilder responseBuilder =
            HealthCheckResponse.named("Database conn");

        try {
            checkDatabaseConnection();
            responseBuilder.withData("connection", true);
            responseBuilder.up();
        } catch (IOException e) {
            // cannot access the database
            responseBuilder.down()
                .withData("error", e.getMessage());
        }

        return responseBuilder.build();
    }
}
```

Builds the next output:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "Database conn",
      "status": "UP",
      "data": {
        "connection": true
      }
    }
  ]
}
```

Since health checks are CDI beans, you can do:

```
@ApplicationScoped
public class DatabaseHealthCheck {

    @Produces
    @Liveness
    HealthCheck check1() {
        return io.smallrye.health.HealthStatus
            .up("successful-live");
    }

    @Produces
    @Readiness
    HealthCheck check2() {
        return HealthStatus
            .state("successful-read", this::isReady)
    }

    private boolean isReady() {}
}
```

You can ping liveness or readiness health checks individually by querying `/health/live` or `/health/ready`.

Quarkus comes with some `HealthCheck` implementations for checking service status.

- **SocketHealthCheck**: checks if host is reachable using a socket.
- **UrlHealthCheck**: checks if host is reachable using a Http URL connection.
- **InetAddressHealthCheck**: checks if host is reachable using `InetAddress.isReachable` method.

```
@Produces
@Liveness
HealthCheck check1() {
    return new UrlHealthCheck("https://www.google.com")
        .name("Google-Check");
}
```

If you want to override or set manually readiness/liveness probes, you can do it by setting health properties:

```
quarkus.smallrye-health.root-path=/hello
quarkus.smallrye-health.liveness-path=/customlive
quarkus.smallrye-health.readiness-path=/customready
```

Automatic readiness probes

Some default *readiness probes* are provided by default if any of the next features are added:

datasource

A probe to check database connection status.

kafka

A probe to check kafka connection status. In this case you need to enable manually by setting `quarkus.kafka.health.enabled` to `true`.

mongoDB

A probe to check MongoDB connection status.

neo4j

A probe to check Neo4J connection status.

artemis

A probe to check Artemis JMS connection status.

kafka-streams

Liveness (for stream state) and Readiness (topics created) probes.

You can disable the automatic generation by setting `<component>.health.enabled` to `false`.

```
quarkus.kafka-streams.health.enabled=false
quarkus.mongodb.health.enabled=false
quarkus.neo4j.health.enabled=false
```

Metrics

Quarkus can utilize the MicroProfile Metrics spec to provide metrics support.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-metrics"
```

The metrics can be read with JSON or the OpenMetrics format. An endpoint is registered automatically at `/metrics` providing default metrics.

MicroProfile Metrics annotations:

Annotation	Description
<code>@Timed</code>	Tracks the duration.
<code>@Metered</code>	Tracks the frequency of invocations.
<code>@Counted</code>	Counts number of invocations.
<code>@Gauge</code>	Samples the value of the annotated object.
<code>@ConcurrentGauge</code>	Gauge to count parallel invocations.

Annotation	Description
<code>@Metric</code>	Used to inject a metric. Valid types <code>Meter</code> , <code>Timer</code> , <code>Counter</code> , <code>Histogram</code> . <code>Gauge</code> only on producer methods/fields.

```
@GET
//...
@Timed(name = "checksTimer",
description = "A measure of how long it takes
               to perform a hello.",
unit = MetricUnits.MILLISECONDS)
public String hello() {}

@Counted(name = "countWelcome",
description = "How many welcome have been performed.")
public String hello() {}
```

`@Gauge` annotation returning a measure as a gauge.

```
@Gauge(name = "hottestSauce", unit = MetricUnits.NONE,
description = "Hottest Sauce so far.")
public Long hottestSauce() {}
```

Injecting a histogram using `@Metric`.

```
@Inject
@Metric(name = "histogram")
Histogram histogram;
```

Tracing

Quarkus can utilize the MicroProfile OpenTracing spec.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-opentracing"
```

Requests sent to any endpoint are traced automatically.

This extension includes OpenTracing support and `Jaeger` tracer.

Jaeger tracer configuration:

```
quarkus.jaeger.service-name=myservice
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
```

`@Traced` annotation can be set to disable tracing at class or method level.

`Tracer` class can be injected into the class.

```
@Inject
Tracer tracer;

tracer.activeSpan().setBaggageItem("key", "value");
```

You can disable `Jaeger` extension by using `quarkus.jaeger.enabled` property.

Additional tracers

JDBC Tracer

Adds a span for each JDBC queries.

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-jdbc</artifactId>
</dependency>
```

Configure JDBC driver apart from tracing properties seen before:

```
# add ':tracing' to your database URL
quarkus.datasource.url=
  jdbc:tracing:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver=
  io.opentracing.contrib.jdbc.TracingDriver
quarkus.hibernate-orm.dialect=
  org.hibernate.dialect.PostgreSQLDialect
```


Cloud

Native

You can build a native image by using GraalVM. The common use case is creating a Docker image so you can execute the next commands:

```
./mvnw package -Pnative -Dquarkus.native.container-build=true

docker build -f src/main/docker/Dockerfile.native
               -t quarkus/getting-started .
docker run -i --rm -p 8080:8080 quarkus/getting-started
```

You can use `quarkus.native.container-runtime` to select the container runtime to use. Now `docker` (default) and `podman` are the valid options.

```
./mvnw package -Pnative -Dquarkus.native.container-runtime=
podman
```

To configure native application, you can create a `config` directory at the same place as the native file and place an `application.properties` file inside. `config/application.properties`.

Kubernetes

Quarks can use Dekorato to generate Kubernetes resources.

```
./mvnw quarkus:add-extensions
      -Dextensions="io.quarkus:quarkus-kubernetes"
```

Running `./mvnw package` the Kubernetes resources are created at `target/kubernetes/` directory.

Property	Default	Description
<code>kubernetes.group</code>	Current username	Set Docker Username.
<code>quarkus.application.name</code>	Current project name	Project name

Generated resource is integrated with MicroProfile Health annotations.

Also, you can customize the generated resource by setting the new values in `application.properties`:

```
kubernetes.replicas=3

kubernetes.labels[0].key=foo
kubernetes.labels[0].value=bar

kubernetes.readiness-probe.period-seconds=45
```

All possible values are explained at <https://quarkus.io/guides/kubernetes#configuration-options>.

Kubernetes Deployment Targets

You can generate different resources setting the property `kubernetes.deployment.target`.

Possible values are `kubernetes`, `openshift` and `knative`. The default value is `kubernetes`.

Kubernetes Client

Quarkus integrates with Fabric8 Kubernetes Client.

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

List of Kubernetes client parameters.

`quarkus.kubernetes-client` as prefix is skipped in the next table.

Property	Default	Description
<code>trust-certs</code>	false	Trust self-signed certificates.
<code>master-url</code>		URL of Kubernetes API server.
<code>namesapce</code>		Default namespace.
<code>ca-cert-file</code>		CA certificate data.
<code>client-cert-file</code>		Client certificate file.
<code>client-cert-data</code>		Client certificate data.
<code>client-key-data</code>		Client key data.
<code>client-key-algorithm</code>		Client key algorithm.
<code>client-key-passphrase</code>		Client key passphrase.
<code>username</code>		Username.
<code>password</code>		Password.

Property	Default	Description
<code>watch-reconnect-interval</code>	PT1S	Watch reconnect interval.
<code>watch-reconnect-limit</code>	-1	Maximum reconnect attempts.
<code>connection-timeout</code>	PT10S	Maximum amount of time to wait for a connection.
<code>request-timeout</code>	PT10S	Maximum amount of time to wait for a request.
<code>rolling-timeout</code>	PT15M	Maximum amount of time to wait for a rollout.
<code>http-proxy</code>		HTTP proxy used to access the Kubernetes.
<code>https-proxy</code>	``	HTTPS proxy used to access the Kubernetes.
<code>proxy-username</code>		Proxy username.
<code>proxy-password</code>		Proxy password.
<code>no-proxy</code>		IP addresses or hosts to exclude from proxying

Or programmatically:

```
@Dependent
public class KubernetesClientProducer {

    @Produces
    public KubernetesClient kubernetesClient() {
        Config config = new ConfigBuilder()
            .withMasterUrl("https://mymaster.com")
            .build();
        return new DefaultKubernetesClient(config);
    }
}
```

And inject it on code:

```
@Inject
KubernetesClient client;

ServiceList myServices = client.services().list();

Service myservice = client.services()
    .inNamespace("default")
    .withName("myservice")
    .get();

CustomResourceDefinitionList crds = client
    .customResourceDefinitions()
    .list();

dummyCRD = new CustomResourceDefinitionBuilder()
    ...
    .build()
client.customResourceDefinitions()
    .create(dummyCRD);
```

Testing

Quarkus provides a Kubernetes Mock test resource that starts a mock of Kubernetes API server and sets the proper environment variables needed by Kubernetes Client.

Register next dependency: `io.quarkus:quarkus-test-kubernetes-client:test`.

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

    @MockServer
    private KubernetesMockServer mockServer;

    @Test
    public void test() {
        final Pod pod1 = ...
        mockServer
            .expect()
            .get()
            .withPath("/api/v1/namespaces/test/pods")
            .andReturn(200,
                new PodListBuilder()
                    .withNewMetadata()
                    .withResourceVersion("1")
                    .endMetadata()
                    .withItems(pod1, pod2)
                    .build())
            .always();
    }
}
```

Amazon Lambda

Quarkus integrates with Amazon Lambda.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-amazon-lambda"
```

And then implement `com.amazonaws.services.lambda.runtime.RequestHandler` interface.

```
public class TestLambda
    implements RequestHandler<MyInput, MyOutput> {
    @Override
    public MyInput handleRequest(MyOutput input,
                                Context context) {

    }
}
```

You can set the handler name by using `quarkus.lambda.handler` property or by annotating the Lambda with the CDI `@Named` annotation.

Test

You can write tests for Amazon lambdas:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-amazon-lambda</artifactId>
  <scope>test</scope>
</dependency>
```

```
@Test
public void testLambda() {
    MyInput in = new MyInput();
    in.setGreeting("Hello");
    in.setName("Stu");
    MyOutput out = LambdaClient.invoke(MyOutput.class, in);
}
```

To scaffold a AWS Lambda run:

```
mvn archetype:generate \
  -DarchetypeGroupId=io.quarkus \
  -DarchetypeArtifactId=quarkus-amazon-lambda-archetype \
  -DarchetypeVersion={version}
```

Azure Functions

Quarkus can make a microservice be deployable to the Azure Functions.

To scaffold a deployable microservice to the Azure Functions run:

```
mvn archetype:generate \
  -DarchetypeGroupId=io.quarkus \
  -DarchetypeArtifactId=quarkus-azure-functions-http-archetype \
  -DarchetypeVersion={version}
```

Apache Camel

Apache Camel Quarkus has its own site:
<https://github.com/apache/camel-quarkus>

WebSockets

Quarkus can be used to handling web sockets.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-undertow-websockets"
```

And web sockets classes can be used:

```
@ServerEndpoint("/chat/{username}")
@ApplicationScoped
public class ChatSocket {

    @OnOpen
    public void onOpen(Session session,
        @PathParam("username") String username) {}

    @OnClose
    public void onClose(..) {}

    @OnError
    public void onError(..., Throwable throwable) {}

    @OnMessage
    public void onMessage(...) {}

}
```

OpenAPI

Quarkus can expose its API description as OpenAPI spec and test it using Swagger UI.

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-smallrye-openapi"
```

Then you only need to access to `/openapi` to get OpenAPI v3 spec of services.

You can update the OpenApi path by setting `quarkus.smallrye-openapi.path` property.

Also, in case of starting Quarkus application in `dev` or `test` mode, Swagger UI is accessible at `/swagger-ui`. If you want to use it in production mode you need to set `quarkus.swagger-ui.always-include` property to `true`.

You can update the Swagger UI path by setting `quarkus.swagger-ui.path` property.

```
quarkus.swagger-ui.path=/my-custom-path
```

You can customize the output by using Open API v3 annotations.

```
@Schema(name="Developers",
    description="POJO that represents a developer.")
public class Developer {
    @Schema(required = true, example = "Alex")
    private String name;
}

@POST
@Path("/developer")
@Operation(summary = "Create deeloper",
    description = "Only be done by admin.")
public Response createDeveloper(
    @RequestBody(description = "Developer object",
        required = true,
        content = @Content(schema =
            @Schema(implementation = Developer.class)))
    Developer developer)
```

All possible annotations can be seen at org.eclipse.microprofile.openapi.annotations package.

You can also serve OpenAPI Schema from static files instead of dynamically generated from annotation scanning.

You need to put OpenAPI documentation under `META-INF` directory (ie: `META-INF/openapi.yaml`).

A request to `/openapi` will serve the combined OpenAPI document from the static file and the generated from annotations. You can disable the scanning documents by adding the next configuration property: `mp.openapi.scan.disable=true`.

Other valid document paths are: `META-INF/openapi.yml`, `META-INF/openapi.json`, `WEB-INF/classes/META-INF/openapi.yml`, `WEB-INF/classes/META-INF/openapi.yaml`, `WEB-INF/classes/META-INF/openapi.json`.

Mail Sender

You can send emails by using Quarkus Mailer extension:

```
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-mailer"
```

You can inject two possible classes `io.quarkus.mailer.Mailer` for synchronous API or `io.quarkus.mailer.ReactiveMailer` for asynchronous API.

```
@Inject
Mailer mailer;

@Inject
ReactiveMailer reactiveMailer;
```

And then you can use them to send an email:

```
mailer.send(
    Mail.withText("to@acme.org", "Subject", "Body")
);

CompletionStage<Void> stage =
    reactiveMailer.send(
        Mail.withText("to@acme.org", "Subject", "Body")
    );
```

Mail class contains methods to add `cc`, `bcc`, headers, bounce address, reply to, attachments, inline attachments and html body.

```
mailer.send(Mail.withHtml("to@acme.org", "Subject", body)
    .addInlineAttachment("quarkus.png",
        new File("quarkus.png"),
        "image/png", "<my-image@quarkus.io>"));
```



If you need deep control you can inject Vert.x mail client


```
@Inject MailClient client;
```

You need to configure SMTP properties to be able to send an email:

```
quarkus.mailer.from=test@quarkus.io
quarkus.mailer.host=smtptest@quarkus.io
quarkus.mailer.port=465
quarkus.mailer.ssl=true
quarkus.mailer.username=...
quarkus.mailer.password=...
```

List of Mailer parameters. `quarkus.` as a prefix is skipped in the next table.

Parameter	Default	Description
mailer.from		Default address.
mailer.mock	false in prod, true in dev and test.	Emails not sent, just printed and stored in a MockMailbox.
mailer.bounce-address		Default address.
mailer.host	mandatory	SMTP host.
mailer.port	25	SMTP port.
mailer.username		The username.
mailer.password		The password.
mailer.ssl	false	Enables SSL.
mailer.trust-all	false	Trust all certificates.
mailer.max-pool-size	10	Max connections . open
mailer.own-host-name		Hostname for and HELO/EHLO Message-ID
mailer.keep-alive	true	Connection pool enabled.
mailer.disable-esmtp	false	Disable ESMTP.
mailer.start-tls	OPTIONAL	TLS security mode. DISABLED, OPTIONAL, REQUIRED.
mailer.login	NONE	Login mode. NONE, OPTIONAL, REQUIRED.
mailer.auth-methods	All methods.	Space-separated list.
mailer.key-store		Path of the key store.

Parameter	Default	Description
mailer.key-store-password		Key store password.
	if you enable SSL for the mailer and you want to build a native executable, you will need to enable the SSL support <code>quarkus.ssl.native=true</code> .	

Testing

If `quarkus.mailer.mock` is set to `true`, which is the default value in `dev` and `test` mode, you can inject `MockMailbox` to get the sent messages.

```
@Inject
MockMailbox mailbox;

@BeforeEach
void init() {
    mailbox.clear();
}

List<Mail> sent = mailbox
    .getMessagesSentTo("to@acme.org");
```

Scheduled Tasks

You can schedule periodic tasks with Quarkus.

```
@ApplicationScoped
public class CounterBean {

    @Scheduled(every="10s")
    void increment() {}

    @Scheduled(cron="0 15 10 * * ?")
    void morningTask() {}
}
```

`every` and `cron` parameters can be surrounded with `{}` and the value is used as config property to get the value.

```
@Scheduled(cron = "{morning.check.cron.expr}")
void morningTask() {}
```

And configure the property into `application.properties`:

```
morning.check.cron.expr=0 15 10 * * ?
```

By default Quarkus expresion is used, but you can change that by setting `quarkus.scheduler.cron-type` property.

```
quarkus.scheduler.cron-type=unix
```

Kogito

Quarkus integrates with Kogito, a next-generation business automation toolkit from Drools and jBPM projects for adding business automation capabilities.

To start using it you only need to add the next extension:

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

Apache Tika

Quarkus integretrs with Apache Tika to detect and extract metadata/text from different file types:

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

```
@Inject
io.quarkus.tika.TikaParser parser;

@POST
@Path("/text")
@Consumes({ "text/plain", "application/pdf",
            "application/vnd.oasis.opendocument.text" })
@Produces(MediaType.TEXT_PLAIN)
public String extractText(InputStream stream) {
    return parser.parse(stream).getText();
}
```

You can configure Apache Tika in `application.properties` file by using next properties prefixed with `quarkus`:

Parameter	Default	Description
tika.tika-config-path	tika-config.xml	Path to the Tika configuration resource.
quarkus.tika.parsers		CSV of the abbreviated or full parser class to be loaded by the extension.
tika.append-embedded-content	true	The document may have other embedded documents. Set if autmatically append.


JGit

Quarkus integrets with JGit to integrate with Git repositories:

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

And then you can start using JGit:

```
try (Git git = Git.cloneRepository()
    .setDirectory(tmpDir)
    .setURI(url)
    .call()) {
    return tmpDir.toString();
}
```

 When running in native mode, make sure to configure SSL access correctly `quarkus.ssl.native=true` (Native and SSL).

Web Resources

You can serve web resources with Quarkus. You need to place web resources at `src/main/resources/META-INF/resources` and then they are accessible (ie `http://localhost:8080/index.html`)

By default static resources as served under the root context. You can change this by using `quarkus.http.root-path` property.

Transactional Memory

Quarkus integrates with the Software Transactional Memory (STM) implementation provided by the Narayana project.

```
./mvnw quarkus:add-extension
-Dextensions="narayana-stm"
```

Transactional objects must be interfaces and annotated with `org.jboss.stm.annotations.Transactional`.

```
@Transactional
@NestedTopLevel
public interface FlightService {
    int getNumberOfBookings();
    void makeBooking(String details);
}
```

The pessimistic strategy is the default one, you can change to optimistic by using `@Optimistic`.

Then you need to create the object inside `org.jboss.stm.Container`.

```
Container<FlightService> container = new Container<>();
FlightServiceImpl instance = new FlightServiceImpl();
FlightService flightServiceProxy = container.create(instance);
```

The implementation of the service sets the locking and what needs to be saved/restored:

```
import org.jboss.stm.annotations.ReadLock;
import org.jboss.stm.annotations.State;
import org.jboss.stm.annotations.WriteLock;

public class FlightServiceImpl
    implements FlightService {
    @State
    private int numberOfBookings;

    @ReadLock
    public int getNumberOfBookings() {
        return numberOfBookings;
    }

    @WriteLock
    public void makeBooking(String details) {
        numberOfBookings += 1;
    }
}
```

Any member is saved/restored automatically (`@State` is not mandatory). You can use `@NotState` to avoid behaviour.

Transaction boundaries

Declarative

- `@NestedTopLevel`: Defines that the container will create a new top-level transaction for each method invocation.
- `@Nested`: Defines that the container will create a new top-level or nested transaction for each method invocation.

Programmatically

```
AtomicAction aa = new AtomicAction();

aa.begin();
{
    try {
        flightService.makeBooking("BA123 ...");
        taxiService.makeBooking("East Coast Taxis ...");

        aa.commit();
    } catch (Exception e) {
        aa.abort();
    }
}
```

Quartz

Quarkus integrates with Quartz to schedule periodic clustered tasks.

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


```
@ApplicationScoped
public class TaskBean {

    @Transactional
    @Scheduled(every = "10s")
    void schedule() {
        Task task = new Task();
        task.persist();
    }
}
```

To configure in clustered mode vida DataSource:

```
quarkus.datasource.url=jdbc:postgresql://localhost/quarkus_test
quarkus.datasource.driver=org.postgresql.Driver
# ...

quarkus.quartz.clustered=true
quarkus.quartz.store-type=db
```

 You need to define the datasource used by clustered mode and also import the database tables following the Quartz schema.

Qute

Qute is a templating engine designed specifically to meet the Quarkus needs. Templates should be placed by default at `src/main/resources/templates` aand subdirectories.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-resteasy-qute"
```

Templates can be defined in any format, in case of HTML:

```
item.html

{@org.acme.Item item}
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title>
</head>
<body>
    <h1>{item.name}</h1>
    <div>Price: {item.price}</div>
    {#if item.price > 100}
    <div>Discounted Price: {item.discountedPrice}</div>
    {/if}
</body>
</html>
```

First line is not mandatory but helps on doing property checks at compilation time.

To render the template:

```
public class Item {
    public String name;
    public BigDecimal price;
}

@Inject
io.quarkus.qute.Template item;

@GET
@Path("/{id}")
@Produces(MediaType.TEXT_HTML)
public TemplateInstance get(@PathParam("id") Integer id) {
    return item.data("item", service.findItem(id));
}

@TemplateExtension
static BigDecimal discountedPrice(Item item) {
    return item.price.multiply(new BigDecimal("0.9"));
}
```

If `@ResourcePath` is not used in `Template` then the name of the field is used as file name. In this case the file should be `src/main/resources/templates/item.{{}}`. Extension of the file is not required to be set.

`discountedPrice` is not a field of the POJO but a method call. Method definition must be annotated with `@TemplateExtension`. First paramter is used to match thebase object.

You can render programmatically too:

```
// file located at src/main/resources/templates/reports/v1/
report_01.{{}}
@ResourcePath("reports/v1/report_01")
Template report;

String output = report
    .data("samples", service.get())
    .render();
```

Sentry

Quarkus integrates with Sentry for logging errors into an error monitoring system.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-logging-sentry"
```

And the configuration to send all errors occuring in the package `org.example` to Sentrty with DSN `https://abcd@sentry.io/1234`:

```
quarkus.log.sentry=true
quarkus.log.sentry.dsn=https://abcd@sentry.io/1234
quarkus.log.sentry.level=ERROR
quarkus.log.sentry.in-app-packages=org.example
```

Full list of configuration properties having `quarkus.log` as prefix:

sentry.enable
Enable the Sentry logging extension (default: false)

sentry.dsn
Where to send events.

sentry.level
Log level (default: `WARN`)

sentry.in-app-packages
Configure which package prefixes your application uses.

Spring DI

Quarkus provides a compatibility layer for Spring dependency injection.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-spring-di"
```

Some examples of what you can do. Notice that annotations are the Spring original ones.

```
@Configuration
public class AppConfiguration {

    @Bean(name = "capitalizeFunction")
    public StringFunction capitalizer() {
        return String::toUpperCase;
    }
}
```

Or as a component:

```
@Component("noopFunction")
public class NoOpSingleStringFunction
    implements StringFunction {
}
```

Also as a service and injection properties from `application.properties`.

```
@Service
public class MessageProducer {

    @Value("${greeting.message}")
    String message;

}
```

And you can inject using `Autowired` or constructor in a component and in a JAX-RS resource too.

```
@Component
public class GreeterBean {

    private final MessageProducer messageProducer;

    @Autowired @Qualifier("noopFunction")
    StringFunction noopStringFunction;

    public GreeterBean(MessageProducer messageProducer) {
        this.messageProducer = messageProducer;
    }
}
```

Spring Web

Quarkus provides a compatibility layer for Spring Web.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-spring-web"
```

Specifically supports the REST related features. Notice that infrastructure things like `BeanPostProcessor` will not be executed.

```
@RestController
@RequestMapping("/greeting")
public class GreetingController {

    private final GreetingBean greetingBean;

    public GreetingController(GreetingBean greetingBean) {
        this.greetingBean = greetingBean;
    }

    @GetMapping("/{name}")
    public Greeting hello(@PathVariable(name = "name")
        String name) {
        return new Greeting(greetingBean.greet(name));
    }
}
```

Supported annotations are: `RestController`, `RequestMapping`, `GetMapping`, `PostMapping`, `PutMapping`, `DeleteMapping`, `PatchMapping`, `RequestParam`, `RequestHeader`, `MatrixVariable`, `PathVariable`, `CookieValue`, `RequestBody`, `ResponseStatus`, `ExceptionHandler` and `RestControllerAdvice`.



If you scaffold the project with `spring-web` extension, then Spring Web annotations are sed in the generated project.

```
mvn io.quarkus:quarkus-maven-plugin:1.0.0.CR2:create ... -Dextensions="spring-web".
```

The next return types are supported: `org.springframework.http.ResponseEntity` and `java.util.Map`.

The next parameter types are supported: An `Exception` argument and `ServletRequest/HttpServletRequest` (adding `quarkus-undertow` dependency).

Spring Data JPA

While users are encouraged to use Hibernate ORM with Panache for Relational Database access, Quarkus provides a compatibility layer for Spring Data JPA repositories.

```
./mvnw quarkus:add-extension
-Dextensions="quarkus-spring-data-jpa"
```

INFO: Of course you still need to add the JDBC driver, and configure it in `application.properties`.

```
public interface FruitRepository
    extends CrudRepository<Fruit, Long> {
    List<Fruit> findByColor(String color);
}
```

And then you can inject it either as shown in Spring DI or in Spring Web.

Interfaces supported:

- `org.springframework.data.repository.Repository`
- `org.springframework.data.repository.CrudRepository`
- `org.springframework.data.repository.PagingAndSortingRepository`
- `org.springframework.data.jpa.repository.JpaRepository`

INFO: Generated repositories are automatically annotated with `@Transactional`.

Repository fragments is also supported:

```
public interface PersonRepository
    extends JpaRepository<Person, Long>, PersonFragment {

    void makeNameUpperCase(Person person);
}
```

User defined queries:

```
@Query("select m from Movie m where m.rating = ?1")
Iterator<Movie> findByRating(String rating);

@Modifying
@Query("delete from Movie where rating = :rating")
void deleteByRating(@Param("rating") String rating);
```

What is currently unsupported:

- Methods `org.springframework.data.repository.query.QueryByExampleExecutor` of
- QueryDSL support
- Customizing the base repository
- `java.util.concurrent.Future` as return type
- Native and named queries when using `@Query`

Spring Security

Quarkus provides a compatibility layer for Spring Security.

```
./mvnw quarkus:add-extension
-Dextensions="spring-security"
```

You need to choose a security extension to define user, roles, ... such as `openid-connect`, `oauth2`, `properties-file` or `security-jdbc` as seen at RBAC.

Then you can use Spring Security annotations to protect the methods:

```
@Secured("admin")
@GetMapping
public String hello() {
    return "hello";
}
```

Quarkus provides support for some of the most used features of Spring Security's `@PreAuthorize` annotation.

Some examples:

hasRole

- `@PreAuthorize("hasRole('admin')")`
- `@PreAuthorize("hasRole(@roles.USER)")` where `roles` is a bean defined with `@Component` annotation and `USER` is a public field of the class.

hasAnyRole

- `@PreAuthorize("hasAnyRole(@roles.USER, 'view')")`

Permit and Deny All

- `@PreAuthorize("permitAll()")`
- `@PreAuthorize("denyAll()")`

Anonymous and Authenticated

- `@PreAuthorize("isAnonymous()")`
- `@PreAuthorize("isAuthenticated()")`

Expressions

- Checks if the current logged in user is the same as the username method parameter:

```
@PreAuthorize("#person.name == authentication.principal.username")
public void doSomethingElse(Person person){}
```

- Checks if calling a method if user can access:

```
@PreAuthorize("@personChecker.check(#person, authentication.principal.username)")
public void doSomething(Person person){}

@Component
public class PersonChecker {
    public boolean check(Person person, String username) {
        return person.getName().equals(username);
    }
}
```

- Combining expressions:

```
@PreAuthorize("hasAnyRole('user', 'admin') AND #user == principal.username")
public void allowedForUser(String user) {}
```

Resources

- <https://quarkus.io/guides/>
- <https://www.youtube.com/user/lordofthejars>

Authors :



@alexsotob

Java Champion and Director of DevExp at Red Hat

1.0.0.CR2

