Quarkus Cheat-Sheet

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 1.12.0.Final**.

Getting Started

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

```
mvn io.quarkus:quarkus-maven-plugin:1.12.0.Final:create \
    -DprojectGroupId=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")
}
```

Packaging

```
mvn clean package
```

You need to distribute the `-runner.jar` file together with `quarkus-app` directory.

If `quarkus.package.uber-jar` property is set to true, then a uber-jar is created with all dependencies bundled inside the JAR.

By default, Quarkus uses fast jar packaging, if `quarkus.package.type` property is set to `legacy-jar` then Quarkus creates the old standard jar file.

`application.properties`

```
quarkus.package.uber-jar=true
quarkus.package.type=legacy-jar
```

To compile to native, you need to set `GRAALVM_HOME` environment variable and run the `native` profile.

```
mvn clean package -Pnative

./gradlew build -Dquarkus.package.type=native
```

Possible `quarkus.package.type` are: `jar`, `legacy-jar`, `uber-jar` and `native`.

AppCDS

Automatically generate AppCDS as part of the build process set the next property: `quarkus.package.create-appcds=true`.

To make use of it, just run `java -jar -XX:SharedArchiveFile=app-cds.jsa myapp.jar`.

Command mode

You can define the `main` CLI method to start Quarkus. There are two ways, implementing `io.quarkus.runtime.QuarkusApplication` interface or use the Java main method to launch Quarkus.

```
@io.quarkus.runtime.annotations.QuarkusMain
public class HelloWorldMain implements QuarkusApplication {
    @Override
    public int run(String... args) throws Exception {
        System.out.println("Hello World");
        return 10;
    }
}
```

`run` method called when Quarkus starts, and stops when it finishes.

As Java `main`:

```
@QuarkusMain
public class JavaMain {
    public static void main(String... args) {
        Quarkus.run(HelloWorldMain.class, args);
    }
}
```

❗ Use `@QuarkusMain` in only one place.

Use `Quarkus.waitForExit()` from the main thread if you want to run some logic on startup, and then run like a normal application (i.e. not exit).

You can inject command line arguments by using `@CommandLineArguments` annotation:

```
@CommandLineArguments
String[] args;
```

Picocli

You can use Picocli to implement CLI applications:

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

```java
@CommandLine.Command
public class HelloCommand implements Runnable {

    @CommandLine.Option(names = {"-n", "--name"}, descripti
on = "Who will we greet?", defaultValue = "World")
    String name;

    private final GreetingService greetingService;

    public HelloCommand(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    @Override
    public void run() {
        greetingService.sayHello(name);
    }
}
```

All classes annotated with `picocli.CommandLine.Command` are registered as CDI beans.

If only one class annotated with `picocli.CommandLine.Command` it will be used as entry point. If you want to provide your own `@QuarkusMain`:

```java
@QuarkusMain
@CommandLine.Command(name = "demo", mixinStandardHelpOption
s = true)
public class ExampleApp implements Runnable, QuarkusApplica
tion {

    @Inject
    CommandLine.IFactory factory;

    @Override
    public void run() {
    }

    @Override
    public int run(String... args) throws Exception {
        return new CommandLine(this, factory).execute(arg
s);
    }
}
```

Use `quarkus.picocli.native-image.processing.enable` to `false` to use the `picocli-codegen` annotation processor instead of build steps.

You can also configure CDI beans with PicoCLI arguments:

```java
@CommandLine.Command
public class EntryCommand implements Runnable {
    @CommandLine.Option(names = "-c", description = "JDBC c
onnection string")
    String connectionString;

    @Inject
    DataSource dataSource;
}

@ApplicationScoped
class DatasourceConfiguration {

    @Produces
    @ApplicationScoped
    DataSource dataSource(CommandLine.ParseResult parseResu
lt) {
        System.out.println(parseResult.matchedOption("c").g
etValue().toString());
    }
}
```

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=""

./mvnw quarkus:remove-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.

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

Quarkus supports graceful shutdown. By default there is no timeout but can be set by using the `quarkus.shutdown.timeout` config

Dev Mode

```
./mvnw compile quarkus:dev

./gradlew quarkusDev
```

Endpoints are registered automatically to provide some basic debug info in dev mode:

- `HTTP GET /quarkus/arc/beans`
  - Query Parameters: `scope`, `beanClass`, `kind`.
- `HTTP GET /quarkus/arc/observers`

Dev UI

Quarkus adds a Dev UI console to expose extension features.

The Dev UI is available in dev mode only and accessible at the `/q/dev` endpoint by default.

Adding Configuration Parameters

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

```java
@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 (in decreasing priority) as:

- System properties (`-Dgreetings.message`).
- Environment variables (`GREETINGS_MESSAGE`).
- Environment file named `.env` placed in the current working directory (`GREETING_MESSAGE=`).
- External config directory under the current working directory: `config/application.properties`.
- Resources `src/main/resources/application.properties`.

```
greetings.message = Hello World
```

💡 `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`.

> To get the active profile programmatically use `io.quarkus.runtime.configuration.ProfileManager.getActiveProfile()`.

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"
}
```

Special properties are set in **prod** mode: `quarkus.application.version` and `quarkus.application.name` to get them available at runtime.

```
@ConfigProperty(name = "quarkus.application.name")
String applicationName;
```

Additional locations

You can use `smallrye.config.locations` property to set additional configuration files.

```
smallrye.config.locations=config.properties
```

or

```
java -jar -Dsmallrye.config.locations=config.properties
```

You can embed configuration files inside a dependency by adding `META-INF/microprofile.properties` inside the JAR. When dependency is added to the application, configuration properties are merged with current configuration.

@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", namingStrategy=NamingStrategy.KEBAB_CASE)
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", namingStrategy=NamingStrategy.KEBAB_CASE)
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", namingStrategy=NamingStrategy.KEBAB_CASE)
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 `GreeetingConfiguration` is translated to `greeting` or `GreetingExtraConfiguration` to `greeting-extra`). The suffix of the class is always removed.

Naming strategy can be changed with property `namingStrategy`. `KEBAB_CASE` (whatever.foo-bar) or `VERBATIM` (whatever.fooBar).

`@io.quarkus.arc.config.ConfigIgnore` annotation can be used to ignore the injection of configuration elements.

```
@ConfigIgnore
public Integer ignored;
```

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
```

Undertow Properties

Possible parameters with prefix `quarkus.servlet`:

context-path
   The context path to serve all Servlet context from. (default: `/`)

default-charset
   The default charset to use for reading and writing requests. (default: `UTF-8`)

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;
```

💡 Quarkus breaks the CDI spec by skipping the `@Produces` annotation completely if the producer method is annotated with a scope annotation, a stereotype or a qualifier.

```
@Quote("")
Message message(InjectionPoint msg) {
}

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

Alternatives

It is also possible to select alternatives for an application using `application.properties`.

```
quarkus.arc.selected-alternatives=org.acme.Foo,org.acme.*,Bar
```

Beans by Quarkus Profile

Using `@io.quarkus.arc.profile.IfBuildProfile` and `@io.quarkus.arc.profile.UnlessBuildProfile` annotations, you can conditionally enable a bean.

```
@Dependent
public class TracerConfiguration {
    @Produces
    @IfBuildProfile("prod")
    public Tracer realTracer(Reporter reporter, Configuration configuration) {
        return new RealTracer(reporter, configuration);
    }

    @Produces
    @DefaultBean
    public Tracer noopTracer() {
        return new NoopTracer();
    }
}
```

Using `@io.quarkus.arc.profile.IfBuildProperty` annotation, you can conditionally enable a bean. `@io.quarkus.arc.DefaultBean` sets the default bean.

```
@Dependent
public class TracerConfiguration {
    @Produces
    @IfBuildProperty(name = "some.tracer.enabled", stringValue = "true")
    public Tracer realTracer(Reporter reporter, Configuration configuration) {}

    @Produces
    @DefaultBean
    public Tracer noopTracer() {}
}
```

ℹ️ Properties set at runtime have absolutely no effect on the bean resolution using `@IfBuildProperty`.

Container-managed Concurrency

Quarkus provides `@io.quarkus.arc.Lock` and a built-in interceptor for concurrency control.

```
@Lock
@ApplicationScoped
class SharedService {

    void addAmount(BigDecimal amout) {
    }

    @Lock(value = Lock.Type.READ, time = 1, unit = TimeUnit.SECONDS)
    BigDecimal getAmount() {
    }
}
```

By default the class is in write mode (so no concurrent calls allowed) except when lock type is `READ` where the method can be called concurrently if no write operation in process.

JSON Marshalling/Unmarshalling

To work with `JSON-B` 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;
    }
}
```

❗ Default media type in Quarkus RestEasy is JSON.

XML Marshalling/Unmarshalling

To work with `JAX-B` you need to add a dependency:

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

Then annotated POJOs are converted to XML.

```java
@XmlRootElement
public class Message {
}

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

JAXP

To work with `JAX-P` you need to add a dependency:

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

```java
final DocumentBuilder dBuilder = DocumentBuilderFactory.new
Instance().newDocumentBuilder();
final Document doc = dBuilder.parse(in);
return doc.getDocumentElement().getTextContent();
```

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`, …

```java
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:

```java
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:

```java
@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`.

```java
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:

```java
@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.

```java
@Inject
Validator validator;
```

And use it:

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

Localization

You can configure the based locale for validation messages.

```
quarkus.default-locale=ca-ES
# Supported locales resolved by Accept-Language
quarkus.locales=en-US,es-ES,fr-FR, ca_ES
```

`ValidationMessages_ca_ES.properties`

```
pattern.message=No conforme al patro
```

```java
@Pattern(regexp = "A.*", message = "{pattern.message}")
private String name;
```

Bean Validation can be configured . The prefix is: `quarkus.hibernate-validator`.

fail-fast
  When fail fast is enabled the validation will stop on the first constraint violation detected. (default: `false`)

method-validation.allow-overriding-parameter-constraints
  Define whether overriding methods that override constraints should throw an exception. (default: `false`).

method-validation.allow-parameter-constraints-on-parallel-methods
  Define whether parallel methods that define constraints should throw an exception. (default: `false`).

method-validation.allow-multiple-cascaded-validation-on-return-values
  Define whether more than one constraint on a return value may be marked for cascading validation are allowed. (default: `false`).

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`.

category."<category-name>".level
  Minimum level category (default: `INFO`)

level

Default minimum level (default: `INFO`)

console.enabled

Console logging enabled (default: `true`)

console.format

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

console.level

Minimum log level (default: `INFO`)

console.color

Allow color rendering (default: `true`)

file.enable

File logging enabled (default: `false`)

file.format

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

file.level

Minimum log level (default: `ALL`)

file.path

The path to log file (default: `quarkus.log`)

file.rotation.max-file-size

The maximum file size of the log file

file.rotation.max-backup-index

The maximum number of backups to keep (default: `1`)

file.rotation.file-suffix

Rotating log file suffix.

file.rotation.rotate-on-boot

Indicates rotate logs at bootup (default: `true`)

file.async

Log asynchronously (default: `false`)

file.async.queue-length

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

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`)

You can inject logger instance:

```
import org.jboss.logging.Logger;
import io.quarkus.arc.log.LoggerName;

@Inject
Logger log;

@LoggerName("foo")
Logger fooLog;

public void ping() {
    log.info("Simple!");
}
```

Gelf ouput

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

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

handler.gelf.enabled

Enable GELF logging handler (default: `false`)

handler.gelf.host

Hostname/IP of Logstash/Graylof. Prepend `tcp:` for using TCP protocol. (default: `udp:localhost`)

handler.gelf.port

The port. (default: `12201`)

handler.gelf.version

GELF version. (default: `1.1`)

handler.gelf.extract-stack-trace

Post Stack-Trace to StackTrace field. (default: `true`)

handler.gelf.stack-trace-throwable-reference

Gets the cause level to stack trace. `0` is fulls tack trace. (default: `0`)

handler.gelf.filter-stack-trace

Stack-Trace filtering. (default: `false`)

handler.gelf.timestamp-pattern

Java Date pattern. (default: `yyyy-MM-dd HH:mm:ss,SSS`)

handler.gelf.level

Log level `java.util.logging.Level`. (default: `ALL`)

handler.gelf.facility

Name of the facility. (default: `jboss-logmanage`)

handler.gelf.additional-field.<field>.<subfield>

Post additional fields. `quarkus.log.handler.gelf.additional-field.field1.type=String`

handler.gelf.include-full-mdc
  Include all fields from the MDC.

handler.gelf.maximum-message-size
  Maximum message size (in bytes). (default: `8192`)

handler.gelf.include-log-message-parameters
  Include message parameters from the log event. (default: `true`)

handler.gelf.include-location
  Include source code location. (default: `true`)

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:

```java
@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);

}
```

```java
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:

```java
@RestClient
WorldClockService worldClockService;
```

If invokation 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:

```java
@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.

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

Or statically set:

```java
@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.

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

Reactive

Rest Client also integrates with reactive library named Mutiny. To start using it you need to add the `quarkus-rest-client-mutiny`.

After that, a methodon a client interface can return a `io.smallrye.mutiny.Uni` instance.

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

Multipart

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

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

The model object:

```java
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:

```java
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. To disable SSL verification you can use `io.quarkus.restclient.NoopHostnameVerifier`.

%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
```

Instantiate client programmatically

```java
MovieReviewService reviewSvc = RestClientBuilder.newBuilder
()
            .baseUri(apiUri)
            .build(WorldClockService.class);
```

Testing

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

```java
@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. Timeout can be set in `quarkus.http.test-timeout` property.

You can also inject the URL where Quarkus is started:

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

```java
@TestHTTPEndpoint(GreetingResource.class)
@TestHTTPResource
URL url;
```

```java
@QuarkusTest
@TestHTTPEndpoint(GreetingResource.class)
public class GreetingResourceTest {
    @Test
    public void testHelloEndpoint() {
        given()
          .when().get()
          .then()
             .statusCode(200)
             .body(is("hello"));
    }
}
```

Root path is calculated automatically, not necessary to explicitly set.

If you want any changes made to be rolled back at the end of the test you can use the `io.quarkus.test.TestTransaction` annotation.

QuarkusTestProfile

You can define for each Test class a different configuration options.

⚠️ This implies that the Quarkus service is restarted.

```java
public class MyProfile implements io.quarkus.test.junit.Qua
rkusTestProfile {

    @Override
    public Map<String, String> getConfigOverrides() {
        return Map.of("greetings.message", "This is a Test"
);
    }

    @Override
    public String getConfigProfile() {
        return "my-test-profile";
    }

    @Override
    public Set<String> tags() {
        return Collections.singleton("test1");
    }
}

@QuarkusTest
@TestProfile(MyProfile.class)
public class MyTestClass {
}
```

`quarkus.test.profile.tags` property can be set to limit test execution of test profiles. If not set all tests are executed.

```
quarkus.test.profile.tags=test1
```

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 {
}
```

When using multiple `QuarkusTestResource` you can set `parallel` attribute to `true` to start them concurrently.

Testing Callbacks

You can enrich **all** your `@QuarkusTest` classes by implementing the following callback interfaces:

- `io.quarkus.test.junit.callback.QuarkusTestBeforeClassCallback`

- `io.quarkus.test.junit.callback.QuarkusTestAfterConstructCallback`

- `io.quarkus.test.junit.callback.QuarkusTestBeforeEachCallback`

- `io.quarkus.test.junit.callback.QuarkusTestAfterEachCallback`

```
public class SimpleAnnotationCheckerBeforeClassCallback implements QuarkusTestBeforeClassCallback {
    @Override
    public void beforeClass(Class<?> testClass) {
    }
}
```

And needs to be registered as Java SPI:

META-INF/services/io.quarkus.test.junit.callback.QuarkusTestBeforeClassCallback

```
io.quarkus.it.main.SimpleAnnotationCheckerBeforeClassCallback
```

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`.

Mockito

Instead of creating stubs, you can also create mocks of your services with mockito. Add the following dependency `io.quarkus:quarkus-junit5-mockito`:

```
@InjectMock
GreetingService greetingService;

@BeforeEach
public void setup() {
    Mockito.when(greetingService.greet()).thenReturn("Hi");
}

@Path("/hello")
public class ExampleResource {

    @Inject
    GreetingService greetingService;
}
```

Mock is automatically injected and only valid for the defined test class.

Also `spy` is supported:

```
@InjectSpy
GreetingService greetingService;

Mockito.verify(greetingService, Mockito.times(1)).greet();
```

REST Client

To Mock REST Client, you need to define the interface with `@ApplicationScope`:

```
@ApplicationScoped
@RegisterRestClient
public interface GreetingService {
}

@InjectMock
@RestClient
GreetingService greetingService;

Mockito.when(greetingService.hello()).thenReturn("hello from mockito");
```

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.jdbc.url=jdbc:mariadb://localhost:3306/mydb
quarkus.datasource.db-kind=mariadb
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.

db-kind

Built-in datasource kinds so the JDBC driver is resolved automatically. Possible values: `derby`, `h2`, `mariadb`, `mssql`, `mysql`, `postgresql`, `db2`.

username

Username to access.

password

Password to access.

driver

JDBC Driver class. It is not necessary to set if `db-kind` used.

credentials-provider

Sets a custom credential provider name.

credentials-provider-name

It is the `@Named` value of the credentials provider bean. Not necessary if only one implementation.

jdbc.url

The datasource URL.

jdbc.min-size

The datasource pool minimum size. (default: `0`)

jdbc.max-size

The datasource pool maximum size. (default: `20`)

jdbc.initial-size

The initial size of the pool.

jdbc.background-validation-interval

The interval at which we validate idle connections in the background. (default: `2M`)

jdbc.acquisition-timeout

The timeout before cancelling the acquisition of a new connection. (default: `5`)

jdbc.leak-detection-interval

The interval at which we check for connection leaks.

jdbc.idle-removal-interval

The interval at which we try to remove idle connections. (default: `5M`)

jdbc.max-lifetime

The max lifetime of a connection.

jdbc.transaction-isolation-level

The transaction isolation level. Possible values: `UNDEFINED`, `NONE`, `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`.

jdbc.detect-statement-leaks

Warn when a connection is returned to the pool without the application having closed all open statements. (default: `true`)

jdbc.new-connection-sql

Query executed when first using a connection.

jdbc.validation-query-sql

Query executed to validate a connection.

jdbc.pooling-enabled

Disable pooling to prevent reuse of Connections. (default: `true`)

jdbc.enable-metrics

Enable datasource metrics collection when using `quarkus-smallrye-metrics` extension.

jdbc.additional-jdbc-properties.<extraProperty>

Unspecified properties to be passed to the JDBC driver when creating new connections.

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

dialect

Class name of the Hibernate ORM dialect.

dialect.storage-engine

The storage engine when the dialect supports multiple storage engines.

sql-load-script

Name of the file containing the SQL statements to execute when starts. `no-file` force Hibernate to skip SQL import. (default: `import.sql`)

batch-fetch-size

The size of the batches. (default: -1 disabled)

maxFetchDepth

The maximum depth of outer join fetch tree for single-ended associations.

multitenant

Defines the method for multi-tenancy. Possible values: `DATABASE`, `NONE`, `SCHEMA`. (default: `NONE`)

multitenant-schema-datasource

Defines the name of the data source to use in case of SCHEMA approach.

query.query-plan-cache-max-size

The maximum size of the query plan cache.

query.default-null-ordering

Default precedence of null values in `ORDER BY`. Possible values: `none`, `first`, `last`. (default: `none`)

database.generation

Database schema is generation. Possible values: `none`, `create`, `drop-and-create`, `drop`, `update`. (default: `none`)

database.generation.halt-on-error

Stop on the first error when applying the schema. (default: `false`)

database.generation.create-schemas

Hibernate ORM should create the schemas automatically (for databases supporting them).

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 (default: `false`)

`log.jdbc-warnings`
`statistics`
  Enable statiscs collection. (default: `false`)

`physical-naming-strategy`
  Class name of the Hibernate PhysicalNamingStrategy implementation.

`globally-quoted-identifiers`
  Should quote all identifiers. (default: `false`)

`metrics-enabled`
  Metrics published with smallrye-metrics extension (default: `false`)

second-level-caching-enabled
  Enable/Disable 2nd level cache. (default: `true`)

Database operations:

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

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

// Hibernate Filters
Person.findAll().filter("Person.hasName",  Parameters.with(
"name", "Alex"));

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

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

Person.deleteById(id);
// 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. Returning a `PanacheQuery`.

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

`findAll`
  Finds all entities. Returning a `PanacheQuery`.

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

`list`: `String`, [`Object…`, `Map<String, Object>`, `Parameters`]
  Lists of entities meeting given query with parameters set. Returning a `List`.

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

`listAll`
  Finds all entities. Returning a `List`.

`listAll`: `Sort`
  Finds all entities sorted by `Sort` attribute/s. Returning a `List`.

`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.

Enables a Hibernate filter during fetching of results for this query.

`deleteAll`
  Number of deleted entities.

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

`deleteById`: `boolean`, [`Object`]
  Delete by id. Returns if deleted or not.

`persist`: [`Iterable`, `Steram`, `Object…`]
  Persist object.

In case of using streams, remember to close them or use a `try/catch` block: `try (Stream<Person> persons = Person.streamAll())`.

`find` methods defines a `withLock(LockModeType)` to define the lock type and `withHint(QueryHints.HINT_CACHEABLE, "true")` to define hints.

Named Queries

```java
@Entity
@NamedQuery(name = "Person.getByName", query = "from Person
where name = :name")
public class Person extends PanacheEntity {

    public static Person findByName(String name){
        return find("#Person.getByName", name).firstResult
();
    }
}
```

Pagination

```java
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();
```

Range

```java
PanacheQuery<Person> livingPersons = Person
        .find("status", Status.Alive);
List<Person> secondRange = livingPersons.range(25, 49).list
();
```

You cannot mix pagination and range.

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

```xml
<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>
        <dependencies>
            <dependency>
                <groupId>org.jboss</groupId>
                <artifactId>jandex</artifactId>
                <version>2.1.1.Final</version>
            </dependency>
        </dependencies>
    </executions>
</plugin>
```

Panache includes an annotation processor that enhance your entities. If you disable annotation processors you might need to create a marker file on Panache archives at `META-INF/panache-archive.marker` manually.

Testing

To mock using active record pattern:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-panache-mock</artifactId>
    <scope>test</scope>
</dependency>
```

```java
@Test
public void testPanacheMocking() {
    PanacheMock.mock(Person.class);

    Mockito.when(Person.count()).thenReturn(231);
    Assertions.assertEquals(23, Person.count());
    PanacheMock.verify(Person.class, Mockito.times(1)).count();
}
```

DAO pattern

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

```java
@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:

```java
@Inject
EntityManager em;

@Inject
org.hibernate.Session session;

@Inject
org.hibernate.SessionFactory sessionFactory;

em.persist(car);
```

Multiple datasources

You can register more than one datasource.

```properties
# default
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url=jdbc:h2:tcp://localhost/mem:default
....
# users datasource
quarkus.datasource.users.db-kind=h2
quarkus.datasource.users..jdbc.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`.

```java
@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:

```java
@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.

```java
@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.

```java
@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();
```

You can implement your custom credentials provider (ie Azure KeyVault) to provide a username/password for the database connection. `Name` information is not necessary if there is only one custom credential provider.

```
@ApplicationScoped
@Unremovable
@Named("my-credentials-provider")
public class CustomCredentialsProvider implements Credentia
lsProvider {
    @Inject
    Config config;

    @Override
        public Properties getCredentials(String credentials
ProviderName) {

        properties.put(CredentialsProvider.USER_PROPERTY_NA
ME, "hibernate_orm_test");
                properties.put(CredentialsProvider.PASSWORD
_PROPERTY_NAME, "hibernate_orm_test");

    }
}
```

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

Hibernate Multitenancy

Multitenancy is supported using Schema or Database approach. First you need to define how tenant is identified:

```
@RequestScoped
@Unremovable
public class CustomTenantResolver implements TenantResolver
{

    @Inject
    RoutingContext context;

    @Override
    public String getDefaultTenantId() {

        return "base";

    }

    @Override
    public String resolveTenantId() {

    }

}
```

Schema approach

```
quarkus.hibernate-orm.database.generation=none

quarkus.hibernate-orm.multitenant=SCHEMA
```

Database approach

```
quarkus.hibernate-orm.database.generation=none

quarkus.hibernate-orm.multitenant=DATABASE

# default tenant
quarkus.datasource.base.db-kind=postgresql
quarkus.datasource.base.username=quarkus_test
...
# Tenant 'mycompany'
quarkus.datasource.mycompany.db-kind=postgresql
quarkus.datasource.mycompany.username=mycompany
quarkus.flyway.mycompany.locations=classpath:database/mycom
pany
...
```

If you need more dynamic approach implement: `@ApplicationScoped io.quarkus.hibernate.orm.runtime.tenant.TenantConnectionResolver`

Hibernate Envers

Quarkus supports Hibernate Envers.

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

REST Data Panache

REST Data with Panache extension can generate the basic CRUD endpoints for your entities and repositories.

```
./mvnw quarkus:add-extension
  -Dextensions="hibernate-orm-rest-data-panache"
```

You also need to add the JDBC driver extension and a JSON Marshaller (ie `resteasy-jackson`).

Then you can define interfaces for defining endpoints:

In case of Active Record pattern:

```
public interface DeveloperResource extends PanacheEntityRes
ource<Developer, Long> {
}
```

In case of Repository:

```
public interface DeveloperResource extends PanacheRepositor
yResource<DeveloperRepository, Developer, Long> {
}
```

Quarkus will generate automatically the implementation for you following the next rules:

- Default path is a hyphenated lowercase resource name without a suffix of `resource` or `controller`.
- `get(@PathParam("id"))`, `list`, `add(Developer)`, `update(@PathParam("id"), Developer)`, `delete(@PathParam("id"))`

You can customize these defaults by using `@ResourceProperties` and `@MethodProperties` annotations.

```
@ResourceProperties(hal = true, path = "my-developer", halC
ollectionName = "dev-collections")
public interface DeveloperResource extends PanacheEntityRes
ource<Developer, Long> {
    @MethodProperties(path = "all")
    List<Developer> list();
    @MethodProperties(exposed = false)
    void delete(Long id);
}
```

If `hal` is `true`, you need to send the `Accept: application/hal+json` HTTP header to get the response.

Hibernate Reactive

```
./mvnw quarkus:add-extension
  -Dextensions="quarkus-hibernate-reactive, quarkus-resteas
y-mutiny, "
```

Also you need to add the reactive driver (ie `quarkus-reactive-pg-client`).

You can use: `org.hibernate.reactive.mutiny.Mutiny` or `org.hibernate.reactive.stage.Stage`.

```java
@Entity
@Table(name = "dev")
public class Developer {
}

@Inject
CompletionStage<Stage.Session> stageSession;

@Inject
Uni<Mutiny.Session> mutinySession;

public Uni<Long> reactivePersist() {
    return mutinySession
        .flatMap(s -> s.persist(new Developer(1, "Alex"))
        .flatMap(v -> session.flush())
        ....

}

public CompletionStage<Developer> reactiveFind() {
    return stageSession
        .thenCompose(session -> {
            session.find(Developer.class, 1);
        });
}
```

Infinispan

Quarkus integrates with Infinispan:

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

Serialization uses a library called Protostream.

Annotation based

```java
@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.

```java
@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.

```java
@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.

```java
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:

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

Infinispan Embedded

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

Configuration in `infinispan.xml`:

```xml
<local-cache name="quarkus-transaction">
    <transaction
        transaction-manager-lookup=
        "org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup"/>
</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;
```

Redis

Quarkus integrates with Redis.

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

Configure Redis location:

```
quarkus.redis.hosts=localhost:6379
```

You can use synchronous or reactive clients:

```
@Inject
RedisClient redisClient;

@Inject
ReactiveRedisClient reactiveRedisClient;
```

```java
void increment(String key, Integer incrementBy) {
    redisClient.incrby(key, incrementBy.toString());
}

Uni<List<String>> keys() {
    return reactiveRedisClient
        .keys("*")
        .map(response -> {
            List<String> result = new ArrayList<>();
            for (Response r : response) {
                result.add(r.toString());
            }
            return result;
        });
}
```

Multiple Redis Clients

```
quarkus.redis.hosts = localhost:6379
quarkus.redis.second.hosts = localhost:6379
```

```
@Inject
RedisClient defaultRedisClient;

@Inject
@RedisClientName("second")
RedisClient redisClient2;
```

List of Redis parameters.

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

health.enabled
Health check is published in case the smallrye-health extension is present. (default: `true`)

password
The Redis password.

hosts
The Redis hosts. (default: `localhost:6379`)

database
The Redis database.

timeout
The maximum delay to wait before a blocking command to redis server times out. (default: `10s`)

ssl
Enables or disables the SSL on connect.

clinet-type
The Redis client type. Possible values: `standalone`, `cluster`, `sentinel` (default: `standalone`)

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

clean-at-start
Execute Flyway clean command (default: `false`)

migrate-at-start
Flyway migration automatically (default: `false`)

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

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

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

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

out-of-order
Allows migrations to be run "out of order".

ignore-missing-migrations
Ignore missing migrations when reading the history table.

sql-migration-prefix

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

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

baseline-on-migrate

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

baseline-version

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

baseline-description

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

validate-on-migrate

    Validate the applied migrations against the available ones (default: `true`)

placeholder-prefix

    Prefix of every placeholder (default: `${`)

placeholder-suffix

    Suffix of every placeholder (default: `}`)

callbacks

    Comma-separated list of fully qualified class names of Callback implementations.

Multiple Datasources

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

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

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

Liquibase

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

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

Then place changelog files to the (`src/main/resources/db`) folder.

You can inject `org.quarkus.liquibase.LiquibaseFactory` to

```
@Inject
LiquibaseFactory liquibaseFactory;

try (Liquibase liquibase = liquibaseFactory.createLiquibase()) {
    ...
}
```

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

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

List of Liquibase parameters.

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

change-log

    The change log file. `XML`, `YAML` , `JSON`, `SQL` formats supported. (default: `db/changeLog.xml`)

change-log-parameters."<parameter-name>"

    Liquibase changelog parameters.

migrate-at-start

    The migrate at start flag. (default: `false`)

validate-on-migrate

    The validate on update flag. (default: `false`)

clean-at-start

    The clean at start flag. (default: `false`)

contexts

    The list of contexts.

labels

    The list of labels.

database-change-log-table-name

    The database change log lock table name. (default: `DATABASECHANGELOG`)

database-change-log-lock-table-name

    The database change log lock table name. (default: `DATABASECHANGELOGLOCK`)

default-catalog-name

    The default catalog name.

default-schema-name

    The default schema name.

liquibase-catalog-name

    The liquibase tables catalog name.

liquibase-schema-name

    The liquibase tables schema name.

liquibase-tablespace-name

    The liquibase tables tablespace name.

Multiple Datasources

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

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

quarkus.liquibase.users.schemas=USERS_TEST_SCHEMA
quarkus.liquibase.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.

```java
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:

```java
public class LibraryResource {

    @Inject
    EntityManager em;

    @Transactional
    public List<Author> searchAuthors(
        @QueryParam("pattern") String pattern) {
        return Search.getSearchSession(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.getSearchSession(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`:

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.

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`)

Waiting time before failing the bootstrap.

index-defaults.refresh-after-write
Set if index should be refreshed after writes.

Possible annotations:

@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 `Map`.

@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 `@DocumentId`.

@IndexingDependency
How a dependency of the indexing process to a property should affect automatic reindexing.

@ObjectPath

@ScaledNumberField
For `java.math.BigDecimal` or `java.math.BigInteger` that you need higher precision.

Amazon DynamoDB

Quarkus integrates with https://aws.amazon.com/dynamodb/:

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

```java
@Inject
DynamoDbClient dynamoDB;
```

To use asycnhronous client with Mutiny:

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

```
@Inject
DynamoDbAsyncClient dynamoDB;

Uni.createFrom().completionStage(() -> dynamoDB.scan(scanRe
quest()))....
```

To use it as a local DynamoDB instance:

```
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. Varables `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
enable-endpoint-discovery	false	Endpoint discovery for a service API that supports endpoint discovery.
endpoint-override		Configure the endpoint with which the SDK should communicate.
api-call-timeout		Time to complete an execution.
interceptors		List of class interceptors

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

Parameter	Default	Description
region		Region that hosts DynamoDB.
credentials.type	DEFAULT	Credentials that should be used `DEFAULT`, `STATIC`, `SYSTEM_PROPERTY`, `ENV_VARIABLE`, `PROFILE`, `CONTAINER`, `INSTANCE_PROFILE`, `PROCESS`, `ANONYMOUS`

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

Parameter	Default	Description
DEFAULT		
default-provider.async-credential-update-enabled	false	Should fetch credentials async.
default-provider.reuse-last-provider-enabled	true	Should reuse the last successful credentials.
STATIC		
static-provider.access-key-id		AWS access key id.
static-provider.secret-access-key		AWS secret access key.
PROFILE		
profile-provider.profile-name	default	The name of the profile to use.
PROCESS		

Parameter	Default	Description
process-provider.command		Command to execute to retrieve credentials.
process-provider.process-output-limit	1024	Max bytes to retrieve from process.
process-provider.credential-refresh-threshold	PT15S	The amount of time between credentials expire and credentials refreshed.
process-provider.async-credential-update-enabled	false	Should fetch credentials async.

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

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

Parameter	Default	Description
proxy.enabled	false	Enables HTTP proxy.
proxy.username		Proxy username.
proxy.password		Proxy password.
proxy.ntlm-domain		For NTLM, domain name.
proxy.ntlm-workstation		For NTLM, workstation name.
proxy.preemptive-basic-authentication-enabled		Authenticate pre-emptively.
proxy.non-proxy-hosts		List of non proxy hosts.
tls-managers-provider.type	system-property	TLS manager: none, system-property, file-store
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
connection-acquisition-timeout	10S	Connection acquisation timeout.
connection-max-idle-time	60S	Max time to connection to be opened.
connection-timeout		Connection timeout.

Parameter	Default	Description
connection-time-to-live	0	Max time connection to be open.
max-concurrency	50	Max number of concurrent connections.
use-idle-connection-reaper	true	Connections in pool should be closed asynchronously.
read-timeout	30S	Read timeout.
write-timeout	30S	Write timeout.
proxy.endpoint		Endpoint of the proxy server.
proxy.enabled	false	Enables HTTP proxy.
proxy.non-proxy-hosts		List of non proxy hosts.
tls-managers-provider.type	system-property	TLS manager: none, system-property, file-store
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.
ssl-provider		SSL Provider (jdk, openssl, openssl-refcnt).
protocol	HTTP_1_1	Sets the HTTP protocol.
max-http2-streams		Max number of concurrent streams.

Parameter	Default	Description
event-loop.override	false	Enable custom event loop conf.
event-loop.number-of-threads		Number of threads to use in event loop.
event-loop.thread-name-prefix	aws-java-sdk-NettyEventLoop	Prefix of thread names.

Amazon S3

```
./mvnw quarkus:add-extension
  -Dextensions="quarkus-amazon-s3"
```

```
@Inject
S3Client s3Client;
```

You need to set a HTTP client either `URL Connection`:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>url-connection-client</artifactId>
</dependency>
```

or Apache HTTP:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
</dependency>
```

```
quarkus.s3.sync-client.type=apache
```

And configure it:

```
quarkus.s3.endpoint-override=http://localhost:8008
quarkus.s3.interceptors=io.quarkus.it.amazon.s3.S3ModifyResponse
quarkus.s3.aws.region=us-east-1
quarkus.s3.aws.credentials.type=static
quarkus.s3.aws.credentials.static-provider.access-key-id=test-key
quarkus.s3.aws.credentials.static-provider.secret-access-key=test-secret
```

You can inject asynchronous client too:

```
@Inject
S3AsyncClient s3AsyncClient;
```

And you need to add the asynchronous Netty client:

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```

Configuration properties are the same as Amazon DynamoDB but changing the prefix from `dynamodb` to `s3`.

Neo4j

Quarkus integrates with Neo4j:

```
./mvnw quarkus:add-extension
  -Dextensions="quarkus-neo4j"
```

```
@Inject
org.neo4j.driver.Driver driver;
```

Configuration properties:

`quarkus.neo4j` as prefix is skipped in the next table.

Prefix is `quarkus.neo4j`.

uri
URI of Neo4j. (default: `localhost:7687`)

authentication.username
Username. (default: `neo4j`)

authentication.password
Password. (default: `neo4j`)

authentication.disabled
Disable authentication. (default: `false`)

pool.metrics-enabled
Enable metrics. (default: `false`)

pool.log-leaked-sessions
Enable leaked sessions logging. (default:`false`)

pool.max-connection-pool-size
Max amount of connections. (default: `100`)

pool.max-connection-lifetime
Pooled connections older will be closed and removed from the pool. (default: `1H`)

pool.connection-acquisition-timeout
Timout for connection adquisation. (default: 1M)

pool.idle-time-before-connection-test
Pooled connections idled in the pool for longer than this timeout will be tested before they are used. (default: `-1`)

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 the next extension: `quarkus-resteasy-mutiny`.

```
@GET
public Publisher<String> get() {
    return Multi.createFrom().resource(driver::rxSession,
        session -> session.readTransaction(tx -> {
            RxResult result = tx.run("MATCH (f:Fruit) RETUR
N f.name as name");
            return Multi.createFrom().publisher(result.reco
rds())
                .map(record -> record.get("name").asStr
ing());
        })
    ).withFinalizer(session -> {
        return Uni.createFrom().publisher(session.close());
    });
}
```

MongoDB Client

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

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

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

INFO: Reactive client uses exposes Mutiny API.

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

Multi MongoDB support

You can configure multiple MongoDB clients using same approach as with `DataSource`. The syntax is `quarkus.mongodb.<optional name>.<property>`:

```
quarkus.mongodb.users.connection-string = mongodb://mongo2:
27017/userdb
quarkus.mongodb.inventory.connection-string = mongodb://mon
go3:27017/invdb
```

Inject the instance using `@io.quarkus.mongodb.runtime.MongoClientName` annotation:

```
@Inject
@MongoClientName("users")
MongoClient mongoClient1;
```

`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.

Parameter	Type	Description
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.
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.

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

⛔ `@MongoEntity` is optional.

Multi-tenancy with MongoDB Panache

```
@MongoEntity(collection = "TheBook", clientName = "client2", database = "database2")
```

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 updated = Person.update("name", "Mortal").where("status", Status.Alive);

long countAll = Person.count();

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

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

Pagination

You can also use pagination:

```java
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();
```

Range

```java
PanacheQuery<Person> livingPersons = Person
    .find("status", Status.Alive);
List<Person> secondRange = livingPersons.range(25, 49).list
();
```

You cannot mix pagination and range.

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` → `{'firstname': ?1, 'status': ?2}`

- `amount > ?1 and firstname != ?2` → `{'amount': {'$gt': ?1}, 'firstname': {'$ne': ?2}}`

- `lastname like ?1` → `{'lastname': {'$regex': ?1}}`

- `lastname is not null` → `{'lastname':{'$exists': true}}`

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

Projection

Projection can be done for both PanacheQL and native queries.

```java
import io.quarkus.mongodb.panache.ProjectionFor;

@ProjectionFor(Person.class) (1)
public class PersonName {
    public String name;
}

PanacheQuery<PersonName> shortQuery = Person.find("status "
, Status.Alive).project(PersonName.class);
```

1 Entity class.

Testing

To mock using active record pattern:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-panache-mock</artifactId>
    <scope>test</scope>
</dependency>
```

```java
@Test
public void testPanacheMocking() {
    PanacheMock.mock(Person.class);

    Mockito.when(Person.count()).thenReturn(231);
    Assertions.assertEquals(23, Person.count());
     PanacheMock.verify(Person.class, Mockito.times(1)).coun
t();
}
```

DAO pattern

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

}
```

Jandex

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

```xml
<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>
```

Panache includes an annotation processor that enhance your entities. If you disable annotation processors you might need to create a marker file on Panache archives at `META-INF/panache-archive.marker` manually.

Reactive Panache

MongoDB with Panache allows using reactive implementation too by using `ReactivePanacheMongoEntity` or `ReactivePanacheMongoEntityBase` or `ReactivePanacheMongoRepository` or `ReactivePanacheMongoRepositoryBase` depending on your style.

```java
public class ReactivePerson extends ReactivePanacheMongoEntity {
    public String name;
}

CompletionStage<Void> cs1 = person.persist();
CompletionStage<List<ReactivePerson>> allPersons = ReactivePerson.listAll();
Publisher<ReactivePerson> allPersons = ReactivePerson.streamAll();

Uni<List<PersonName>> persons = ReactivePersonEntity.find(
"lastname", name).project(PersonName.class).list();
```

MongoDB REST Data Panache

MongoDB REST Data with Panache extension can generate the basic CRUD endpoints for your entities and repositories.

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

You also need to add the JDBC driver extension and a JSON Marshaller (ie `resteasy-jackson`).

Then you can define interfaces for defining endpoints:

In case of Active Record pattern:

```java
public interface DeveloperResource extends PanacheMongoEntityResource<Developer, Long> {
}
```

In case of Repository:

```java
public interface DeveloperResource extends PanacheMongoRepositoryResource<DeveloperRepository, Developer, Long> {
}
```

Cassandra

Quarkus integrates with Cassandra and DataStax Object Mapper.

```xml
<dependency>
    <groupId>com.datastax.oss.quarkus</groupId>
    <artifactId>cassandra-quarkus-client</artifactId>
</dependency>
```

Enities and DAOs are generated as you have been doing with DataStax Object Mapper.

You need to create a DaoProducer:

```
@Inject
public FruitDaoProducer(QuarkusCqlSession session) {
    FruitMapper mapper = new FruitMapperBuilder(session).build();
    fruitDao = mapper.fruitDao();
}

@Produces
@ApplicationScoped
FruitDao produceFruitDao() {
    return fruitDao;
}
```

Cassandra configuration:

```
quarkus.cassandra.contact-points=127.0.0.1:9042
quarkus.cassandra.local-datacenter=datacenter1
quarkus.cassandra.keyspace=k1
quarkus.cassandra.auth.username=john
quarkus.cassandra.auth.password=s3cr3t
```

You can configure other Cassandra Java driver settings using `application.conf` or `application.json` files. They need to be located in the classpath of your application. Driver settings reference.

If MicroProfile Metrics extension is registered, the Cassandra extension can provide (if enabled) metrics about the session:

```
quarkus.cassandra.metrics.enabled=true
quarkus.cassandra.metrics.session-enabled=connected-nodes,bytes-sent
quarkus.cassandra.metrics.node-enabled=pool.open-connections
```

Reactive

You can also use Mutiny to define a reactive DAO:

```
@Dao
public interface FruitDaoReactive {

    @Update
    Uni<Void> update(Fruit fruit);

    @Select
    MutinyMappedReactiveResultSet<Fruit> findById(String storeId);
}

@Mapper
public interface FruitMapper {

    @DaoFactory
    FruitDaoReactive fruitDaoReactive();
}
```

Reactive Programming

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

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

Mutiny and JAX-RS

Apart from the `CompletionStage` support, there is also support for Mutiny.

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

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
public Uni<String> hello() {
    return Uni.createFrom().item(() -> "hello");
}

@GET
@Produces(MediaType.TEXT_PLAIN)
public Multi<String> multi() {
    return Multi.createFrom().items("hello", "world");
}
```

Mutiny

Quarkus integrates with **Mutiny** as reactive programming library:

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

```java
@ApplicationScoped
public static class ReactiveHello {

    public Uni<String> greeting() {
        return Uni.createFrom().item(() -> "hello")
                .emitOn(Infrastructure.getDefaultExecutor
());
    }

    public Multi<String> stream() {
        return Multi.createFrom().items("hello", "world")
                .emitOn(Infrastructure.getDefaultExecutor
());
    }
}
```

Converting from/to `RxJava2` or `Reactor` APIs:

RxJava 2

```xml
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>mutiny-rxjava</artifactId>
</dependency>
```

From `RxJava2`:

```java
Uni<Void> uniFromCompletable = Uni.createFrom()
                                  .converter(UniRxConvert
ers.fromCompletable(), completable);
Uni<String> uniFromSingle = Uni.createFrom()
                               .converter(UniRxConverters.
fromSingle(), single);
Uni<String> uniFromObservable = Uni.createFrom()
                                   .converter(UniRxConverters.
fromObservable(), observable);
Uni<String> uniFromFlowable = Uni.createFrom()
                                 .converter(UniRxConverters.
fromFlowable(), flowable);
...

Multi<Void> multiFromCompletable = Multi.createFrom()
                                        .converter(MultiRxC
onverters.fromCompletable(), completable);
Multi<String> multiFromObservable = Multi.createFrom()
                                         .converter(MultiRxC
onverters.fromObservable(), observable);
Multi<String> multiFromFlowable = Multi.createFrom()
                                       .converter(MultiRxC
onverters.fromFlowable(), flowable);
...
```

To `RxJava2`:

```java
Completable completable = uni.convert().with(UniRxConverter
s.toCompletable());
Single<Optional<String>> single = uni.convert().with(UniRxC
onverters.toSingle());
Observable<String> observable = uni.convert().with(UniRxCon
verters.toObservable());
Flowable<String> flowable = uni.convert().with(UniRxConvert
ers.toFlowable());
...

Completable completable = multi.convert().with(MultiRxConve
rters.toCompletable());
Single<Optional<String>> single = multi.convert().with(Mult
iRxConverters.toSingle());
Observable<String> observable = multi.convert().with(MultiR
xConverters.toObservable());
Flowable<String> flowable = multi.convert().with(MultiRxCon
verters.toFlowable());
...
```

Reactor API

```xml
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>mutiny-reactor</artifactId>
</dependency>
```

From `Reactor`:

```java
Uni<String> uniFromMono = Uni.createFrom().converter(UniRea
ctorConverters.fromMono(), mono);
Uni<String> uniFromFlux = Uni.createFrom().converter(UniRea
ctorConverters.fromFlux(), flux);
Multi<String> multiFromMono = Multi.createFrom().converter
(MultiReactorConverters.fromMono(), mono);
Multi<String> multiFromFlux = Multi.createFrom().converter
(MultiReactorConverters.fromFlux(), flux);
```

To `Reactor`:

```java
Mono<String> mono  = uni.convert().with(UniReactorConverter
s.toMono());
Flux<String> flux  = uni.convert().with(UniReactorConverter
s.toFlux());

Mono<String> mono2 = multi.convert().with(MultiReactorConve
rters.toMono());
Flux<String> flux2 = multi.convert().with(MultiReactorConve
rters.toFlux());
```

CompletionStages or *Publisher*

```java
CompletableFuture<String> future = Uni
        .createFrom().completionStage(CompletableFuture.sup
plyAsync(() -> "hello"));

CompletationStage<String> cs = Uni
        .createFrom().subscribeAsCompletionStage();
```

`Multi` implements `Publisher`.

RESTEasy Reactive

RESTEasy Reactive is a new implementation of JAX-RS but fully reactive.

```
mvn quarkus:add-extension
    -Dextensions="quarkus-resteasy-reactive"
```

```java
package org.acme.rest;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
@Path("rest")
public class Endpoint {

    @Path("hello")
    @GET
    public String hello(){
        // executed in event-loop
        return "Hello, World!";
    }

    @GET
    public Uni<Book> culinaryGuide(){
        // executed in event-loop but not blocking
        return Book.findByIsbn("978-2081229297");
    }

    @io.smallrye.common.annotation.Blocking
    @GET
    public String blockingHello() throws InterruptedException {
        // executed in worker-thread
        return "Yaaaawwwwnnnnnn…";
    }

}
```

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.

```java
@ApplicationScoped
public class ProducerData {

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

or in Mutiny:

```java
@ApplicationScoped
public class ProducerData {

    @Outgoing("my-in-memory")
    public Multi<Integer> generate() {
        return Multi.createFrom().ticks().every(Duration.of
Seconds(5))
                    .onItem().apply(n -> 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.

```java
@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:

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

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

Patterns

REST API → Message

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

emitter.send(message);
```

Message → Message

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

Message → SSE

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

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

Message → Business Logic

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

To indicate that the method should be executed on a worker pool you can use `@Blocking`:

```java
@Outgoing("Y")
@Incoming("X")
@Blocking
```

To customize:

```java
@Blocking(value="my-custom-pool", ordered = false)
```

```
smallrye.messaging.worker.my-custom-pool.max-concurrency=3
```

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 `io.quarkus.kafka.client.serialization.JsonbDeserializer` 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
s
        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.StreamsBuil
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 DataSource Properties

Common Reeactive DataSource Client configuration properties prefixed with `quarkus.datasource`:

`reactive.cache-prepared-statements`

Prepared statements should be cached on the client side. (default: `false`)

`reactive.url`

The datasource URL.

`reactive.max-size`

The datasource pool maximum size.

`reactive.trust-all`

All server certificates should be trusted. (default: `false`)

`reactive.trust-certificate-pem`

Trust configuration in the PEM format.

`reactive.trust-certificate-jks`

Trust configuration in the JKS format.

`reactive.trust-certificate-pfx`

Trust configuration in the PFX format.

`reactive.key-certificate-pem`

Key/cert configuration in the PEM format.

`reactive.key-certificate-jks`

Key/cert configuration in the JKS format.

`reactive.key-certificate-pfx`

Key/cert configuration in the PFX format.

`reactive.thread-local`

Use one connection pool per thread.

`reactive.reconnect-attempts`

The number of reconnection attempts when a pooled connection cannot be established on first try. (default: `0`)

`reactive.reconnect-interval`

The interval between reconnection attempts when a pooled connection cannot be established on first try. (default: `PT1S`)

`reactive.idle-timeout`

The maximum time without data written to or read from a connection before it is removed from the pool.

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.db-kind=postgresql
quarkus.datasource.reactive.url=postgresql:///your_database
```

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

```
@Inject
PgPool client;

Uni<List<Fruit>> fruits =
    client.preparedQuery("SELECT * FROM fruits")
    .onItem().apply(rowSet -> {
        JsonArray jsonArray = new JsonArray();
        for (Row row : rowSet) {
            jsonArray.add(from(row));
        }
        return jsonArray;
    })
```

Reactive MySQL Client

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

```
./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.db-kind=mysql
quarkus.datasource.reactive.url=mysql:///your_database
```

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

Reactive DB2 Client

You can use Reactive DB2 to execute queries to DB2 database in a reactive way, instead of using JDBC.

```
./mvnw quarkus:add-extension
  -Dextensions="quarkus-reactive-db2-client"
```

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

```
quarkus.datasource.db-kind=db2
quarkus.datasource.reactive.url=vertx-reactive:db2://localhost:50005/hreact
```

Then you can inject `io.vertx.mutiny.db2client.DB2Pool` class.

Reactive Transactions

`io.vertx.mutiny.sqlclient.SqlClientHelper` is an util class that allows you to run reactive persisten code within a transaction.

```
Uni<Void> r = SqlClientHelper.inTransactionUni(client, tx -> {
        Uni<RowSet<Row>> insertOne = tx.preparedQuery("INSERT INTO fruits (name) VALUES ($1) RETURNING (id)")
                .execute(Tuple.of(fruit1.name));
});
```

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`:

`artemis.url`

Connection URL.

`artemis.username`

Username for authentication.

`artemis.password`

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`:

```java
@ApplicationScoped
public class ArtemisConsumerManager {

    @Inject
    ConnectionFactory connectionFactory;

    private Connection connection;

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

💡 | Configuration options are the same as Artemis core.

Vert.X Reactive Clients

Vert.X Reactive clients in Quarkus, the next clients are supported and you need to add the dependency to use them:

Vert.X Mail Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-mail-client
```

Vert.X MongoDB Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-mongo-client
```

Vert.X Redis Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-redis-client
```

Vert.X Cassandra Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-cassandra-client
```

Vert.X Consul Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-consul-client
```

Vert.X Kafka Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-kafka-client
```

Vert.X AMQP Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-amqp-client
```

Vert.X RabbitMQ Client

```
io.smallrye.reactive:smallrye-mutiny-vertx-rabbitmq-client
```

Example of Vert.X Web Client:

```java
@Inject
Vertx vertx;

private WebClient client;

@PostConstruct
void initialize() {
    this.client = WebClient.create(vertx, ...);
}
```

Amazon SQS Client

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

Injecting the client:

```java
@Inject
software.amazon.awssdk.services.sqs.SqsClient sqs;

SendMessageResponse response = sqs.sendMessage(m -> m.queueUrl(queueUrl).messageBody(message));

List<Message> messages = sqs.receiveMessage(m -> m.maxNumberOfMessages(10).queueUrl(queueUrl)).messages();
```

And configure it:

```
quarkus.sqs.endpoint-override=http://localhost:8010
quarkus.sqs.aws.region=us-east-1
quarkus.sqs.aws.credentials.type=static
quarkus.sqs.aws.credentials.static-provider.access-key-id=test-key
quarkus.sqs.aws.credentials.static-provider.secret-access-key=test-secret
```

You need to set a HTTP client either `URL Connection`:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>url-connection-client</artifactId>
</dependency>
```

or Apache HTTP:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
</dependency>
```

```
quarkus.sqs.sync-client.type=apache
```

You can go async by using Mutiny:

```java
@Inject
software.amazon.awssdk.services.sqs.SqsAsyncClient sqs;

Uni.createFrom()
    .completionStage(
        sqs.sendMessage(m -> m.queueUrl(queueUrl).messageBody(message))
    )
    .onItem()...

return Uni.createFrom()
    .completionStage(
        sqs.receiveMessage(m -> m.maxNumberOfMessages(10).queueUrl(queueUrl))
    )
    .onItem()
```

And you need to add the asynchronous Netty client:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```

Configuration properties are the same as Amazon DynamoDB but changing the prefix from `dynamodb` to `sqs`.

TLS

You can trust all certificates globally by using `quarkus.tls.trust-all=true`

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:

quarkus.jaxrs.deny-uncovered

If true denies by default to all JAX-RS endpoints. (default: `false`)

quarkus.security.deny-unannotated

If true denies by default all CDI methods and JAX-RS endpoints. (default: `false`)

By default in Quarkus, if an incoming request has a credential the request will always be authenticated (even if the target page does not require authentication).

You can change this behaviour by setting `quarkus.http.auth.proactive` property to `false`.

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`.

You can use `enabled` property (ie `quarkus.http.auth.permission.permit1.enabled`) to enable the entire permission set.

Testing

Quarkus provides explicit support for testing with different users, and with the security subsystem disabled.

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-test-security</artifactId>
    <scope>test</scope>
</dependency>
```

```
@Test
@TestSecurity(authorizationEnabled = false)
void someTestMethod() {
...
}

@Test
@TestSecurity(user = "testUser", roles = {"admin", "user"})
void someTestMethod() {
...
}
```

BouncyCastle

Quarkus supports BouncyCastle, you only need to add the BouncyCastle dependency and configure the security provider:

```
quarkus.security.security-providers=BC
quarkus.security.security-providers=BCJSSE
quarkus.security.security-providers=BCFIPS
quarkus.security.security-providers=BCFIPSJSSE
```

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;

@Inject
JWTParser parser;
```

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:

mp.jwt.verify.publickey

Public Key text itself to be supplied as a string.

`mp.jwt.verify.publickey.location` Relative path or URL of a public key.

mp.jwt.verify.issuer

`iss` accepted as valid.

smallrye.jwt.token.header

Sets header such as `Cookie` is used to pass the token. (default: `Authorization`).

smallrye.jwt.token.cookie
Name of the cookie containing a token.

smallrye.jwt.token.schemes
Comma-separated list containing an alternative single or multiple schemes. (default: `Bearer`).

smallrye.jwt.require.named-principal
A token must have a upn or preferred_username or sub claim set if using `java.security.Principal`. `True` makes throw an exception if not set. (default: `false`).

smallrye.jwt.path.sub
Path to the claim with subject name.

smallrye.jwt.claims.sub
Default sub claim value.

smallrye.jwt.path.groups
Path to the claim containing the groups.

smallrye.jwt.groups-separator
Separator for splitting a string which may contain multiple group values. (default. ` `).

smallrye.jwt.claims.groups
Default groups claim value.

smallrye.jwt.jwks.refresh-interval
JWK cache refresh interval in minutes. (default: `60`).

smallrye.jwt.expiration.grace
Expiration grace in seconds. (default: `60`).

smallrye.jwt.verify.aud
Comma separated list of the audiences that a token aud claim may contain.

smallrye.jwt.verify.algorithm
Signature algorith. (defsult: `RS256`)

smallrye.jwt.token.kid
If set then the verification JWK key as well every JWT token must have a matching `kid` header.

smallrye.jwt.time-to-live
The maximum number of seconds that a JWT may be issued for use.

smallrye.jwt.sign.key-location
Location of a private key which will be used to sign the claims when either a no-argument `sign()` or `innerSign()` method is called.

smallrye.jwt.encrypt.key-location

Location of a public key which will be used to encrypt the claims or inner JWT when a no-argument `encrypt()` method is called.

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")
```

Generate tokens

JWT generation API:

```
Jwt.claims()
    .issuer("https://server.com")
    .claim("customClaim", 3)
    .sign(createKey());

JwtSignatureBuilder jwtSignatureBuilder = Jwt.claims("/testJsonToken.json").jws();
jwtSignatureBuilder
    .signatureKeyId("some-key-id")
    .signatureAlgorithm(SignatureAlgorithm.ES256)
    .header("custom-header", "custom-value");
    .sign(createKey());

Jwt.claims("/testJsonToken.json")
    .encrypt(createKey());

JwtEncryptionBuilder jwtEncryptionBuilder = Jwt.claims("/testJsonToken.json").jwe();
jwtEncryptionBuilder
    .keyEncryptionKeyId("some-key-id")
    .keyEncryptionAlgorithm(KeyEncryptionAlgorithm.ECDH_ES_A256KW)
    .header("custom-header", "custom-value");
    .encrypt(createKey());

Jwt.claims("/testJsonToken.json")
   .innerSign(createKey());
   .encrypt(createKey());
```

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:

enabled
The OIDC is enabled. (default: `true`)

tenant-enabled
If the tenant configuration is enabled. (default: `true`)

application-type
The application type. Possible values: `web_app`, `service`. (default: `service`)

connection-delay
The maximum amount of time the adapter will try connecting.

auth-server-url
The base URL of the OpenID Connect (OIDC) server.

introspection-path
Relative path of the RFC7662 introspection service.

jwks-path
Relative path of the OIDC service returning a JWK set.

public-key
Public key for the local JWT token verification

client-id
The client-id of the application.

roles.role-claim-path
Path to the claim containing an array of groups. (`realm/groups`)

roles.role-claim-separator
Separator for splitting a string which may contain multiple group values.

token.issuer
Issuer claim value.

token.audience
Audience claim value.

token.expiration-grace
Expiration grace period in seconds.

token.principal-claim
Name of the claim which contains a principal name.

token.refresh-expired
If property is enabled then a refresh token request is performed.

credentials.secret
The client secret

authentication.redirect-path
Relative path for calculating a `redirect_uri` query parameter.

authentication.restore-path-after-redirect
The original request URI used before the authentication will be restored after the user has been redirected back to the application. (default: `true`)

authentication.scopes
List of scopes.

authentication.extra-params
Additional properties which will be added as the query parameters .

authentication.cookie-path
Cookie path parameter.

proxy.host
The host (name or IP address) of the Proxy.

proxy.port
The port number of the Proxy. (default: `80`)

proxy.username
The username to authenticate.

proxy.password
The password to authenticate.

end-session-path
Relative path of the OIDC `end_session_endpoint`.

logout.path
The relative path of the logout endpoint at the application.

logout.post-logout-path
Relative path of the application endpoint where the user should be redirected to after logging out.

tls.verification
Sets the TLs verification. Possible values: `REQUIRED`, `NONE`. (default: `REQUIRED`).

ℹ 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.

Multi-tenancy

Multi-tenancy is supported by adding a sub-category to OIDC configuration properties (ie `quarkus.oidc.{tenent_id}.property`).

```
quarkus.oidc.auth-server-url=http://localhost:8180/auth/realms/quarkus
quarkus.oidc.client-id=multi-tenant-client
quarkus.oidc.application-type=web-app

quarkus.oidc.tenant-b.auth-server-url=https://accounts.google.com
quarkus.oidc.tenant-b.application-type=web-app
quarkus.oidc.tenant-b.client-id=xxxx
quarkus.oidc.tenant-b.credentials.secret=yyyy
quarkus.oidc.tenant-b.token.issuer=https://accounts.google.com
quarkus.oidc.tenant-b.authentication.scopes=email,profile,openid
```

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:

quarkus.oauth2.enabled
Determine if the OAuth2 extension is enabled. (default: `true`)

quarkus.oauth2.client-id
The OAuth2 client id used to validate the token.

quarkus.oauth2.client-secret
The OAuth2 client secret used to validate the token.

quarkus.oauth2.introspection-url
URL used to validate the token and gather the authentication claims.

quarkus.oauth2.role-claim
The claim that is used in the endpoint response to load the roles ((default: `scope`)

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.

file.enabled
    The file realm is enabled. (default: `false`)

file.auth-mechanism
    The authentication mechanism. (default: `BASIC`)

file.realm-name
    The authentication realm name. (default: `Quarkus`)

file.plain-text

If passwords are in plain or in MD5. (default: `false`)

file.users
    Classpath resource of user/password. (default: `users.properties`)

file.roles
    Classpath resource of user/role. (default: `roles.properties`)

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.

algorithm
    Determine which algorithm to use. Possible values: `DIGEST_MD5`, `DIGEST_SHA`, `DIGEST_SHA_256`, `DIGEST_SHA_384`, `DIGEST_SHA_512`, `DIGEST_SHA_512_256`. (default: `DIGEST_MD5`)

file.enabled
    The file realm is enabled. (default: `false`)

file.auth-mechanism
    The authentication mechanism. (default: `BASIC`)

file.realm-name
    The authentication realm name. (default: `Quarkus`)

file.plain-text
    If passwords are in plain or in MD5. (default: `false`)

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.

auth-mechanism
    The authentication mechanism. (default: `BASIC`)

realm-name
    The authentication realm name. (default: `Quarkus`)

enabled
    If the properties store is enabled. (default: `false`)

principal-query.sql
    The sql query to find the password.

principal-query.datasource
    The data source to use.

principal-query.clear-password-mapper.enabled
    If the clear-password-mapper is enabled. (default: `false`)

principal-query.clear-password-mapper.password-index
    The index of column containing clear password. (default: `1`)

principal-query.bcrypt-password-mapper.enabled
    If the bcrypt-password-mapper is enabled. (default: `false`)

principal-query.bcrypt-password-mapper.password-index
    The index of column containing password hash. (default: `0`)

principal-query.bcrypt-password-mapper.hash-encoding
    A string referencing the password hash encoding (`BASE64` or `HEX`). (default: `BASE64`)

`principal-query.bcrypt-password-mapper.salt-index`

The index column containing the Bcrypt salt. (default: `0`)

`principal-query.bcrypt-password-mapper.salt-encoding`

A string referencing the salt encoding (`BASE64` or `HEX`). (default: `BASE64`)

`principal-query.bcrypt-password-mapper.iteration-count-index`

The index column containing the Bcrypt iteration count. (default: `0`)

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=
```

Security with JPA

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

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

> ℹ️ Also you might require `jdbc-postgresql`, `resteasy`, `hibernate-orm-panache`.

```java
@io.quarkus.security.jpa.UserDefinition
@Table(name = "test_user")
@Entity
public class User extends PanacheEntity {
    @io.quarkus.security.Username
    public String name;

    @io.quarkus.security.Password
    public String pass;

    @ManyToMany
    @Roles
    public List<Role> roles = new ArrayList<>();

    public static void add(String username, String password) {
        User user = new User();
        user.username = username;
        user.password = BcryptUtil.bcryptHash(password);
        user.persist();
    }
}

@Entity
public class Role extends PanacheEntity {

    @ManyToMany(mappedBy = "roles")
    public List<ExternalRolesUserEntity> users;

    @io.quarkus.security.RolesValue
    public String role;
}
```

You need to configure JDBC:

```
quarkus.datasource.url=jdbc:postgresql:security_jpa
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=quarkus
quarkus.datasource.password=quarkus

quarkus.hibernate-orm.database.generation=drop-and-create
```

Security with LDAP

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

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

```
quarkus.security.ldap.enabled=true
quarkus.security.ldap.dir-context.principal=uid=tool,ou=accounts,o=YourCompany,c=DE
quarkus.security.ldap.dir-context.url=ldaps://ldap.server.local
quarkus.security.ldap.dir-context.password=PASSWORD
quarkus.security.ldap.identity-mapping.rdn-identifier=uid
quarkus.security.ldap.identity-mapping.search-base-dn=ou=users,ou=tool,o=YourCompany,c=DE
quarkus.security.ldap.identity-mapping.attribute-mappings."0".from=cn
quarkus.security.ldap.identity-mapping.attribute-mappings."0".to=groups
quarkus.security.ldap.identity-mapping.attribute-mappings."0".filter=(member=uid={0})
quarkus.security.ldap.identity-mapping.attribute-mappings."0".filter-base-dn=ou=roles,ou=tool,o=YourCompany,c=DE
```

Testing

There is a Quarkus Test Resource that starts and stops InMemory LDAP server before and after test suite. It is running in `localhost` with `dc=quarkus,dc=io` and binding credentials (`"uid=admin,ou=system"`, `"secret"`). Imports *LDIF* from a file located at root of the classpath named `quarkus-io.ldif`.

Register dependency io.quarkus:quarkus-test-ldap:test.

And annotate the test:

```java
@QuarkusTestResource(io.quarkus.test.ldap.LdapServerTestResource.class)
public class ElytronLdapExtensionTestResources {
}
```

Elytron LDAP Realm configuration properties. Prefix `quarkus.security.ldap` is skipped.

`enabled`

Enable the LDAP elytron module (default: `false`)

`realm-name`

The elytron realm name (default: `Quarkus`)

`direct-verification`

Provided credentials are verified against LDAP (default: `true`)

`dir-context.url`

The url of the LDAP server.

`dir-context.principal`

User (`bindDn`) which is used to connect to LDAP server.

`dir-context.password`

The password (`bindCredential`) which belongs to the principal.

`identity-mapping.rdn-identifier`

The identifier (`baseFilter`) which correlates to the provided user (default: `uid`)

identity-mapping.search-base-dn

The dn where we look for users.

identity-mapping.attribute-mappings.<id>.from

The `roleAttributeId` from which is mapped

identity-mapping.attribute-mappings.<id>.to

The identifier whom the attribute is mapped to (default: `gropus`)

identity-mapping.attribute-mappings.<id>.filter

The filter (`roleFilter`)

identity-mapping.attribute-mappings.<id>.filter-base-dn

The filter base dn (`rolesContextDn`)

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
quarkus.vault.secret-config-kv-prefix.singer.paths=
    multi/singer1, multi/singer2
```

```
vault kv put secret/myapps/vault-quickstart/config a-private-key=123456
```

```
vault kv put secret/multi/singer1 firstname=paul
```

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

@ConfigProperty(name = "singer.firstname")
String firstName;
```

You can access the KV engine programmatically:

```
@Inject
VaultKVSecretEngine kvSecretEngine;

kvSecretEngine.readSecret("myapps/vault-quickstart/" + vaul
tPath).toString();

Map<String, String> secrets;
kvSecretEngine.writeSecret("myapps/vault-quickstart/crud",
 secrets);

kvSecretEngine.deleteSecret("myapps/vault-quickstart/crud"
);
```

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

No password is set as it is fetched from Vault.

Dynamic credentials are also supported:

Running the following dynamic database config in Vault:

```
vault write database/config/mydb plugin_name=postgresql-database-
plugin …..
```

You can configure as:

```
quarkus.vault.credentials-provider
    .mydatabase.database-credentials-role=mydbrole

quarkus.datasource.db-kind=
    postgresql
quarkus.datasource.credentials-provider=
    mydatabase
quarkus.datasource.jdbc.url=
    jdbc:postgresql://localhost:5432/mydatabase
```

Username and password are fetched from Vault

Transit

```
@Inject
VaultTransitSecretEngine transit;

transit.encrypt("my_encryption", text);
transit.decrypt("my_encryption", text).asString();
transit.sign("my-sign-key", text);
```

Transit Key

```
@Inject
VaultTransitSecretEngine transit;

transit.createKey(KEY_NAME, new KeyCreationRequestDetail().
setExportable(true));
transit.readKey(KEY_NAME);
transit.listKeys();
transit.exportKey(KEY_NAME, VaultTransitExportKeyType.encry
ption, null);
transit.updateKeyConfiguration(KEY_NAME, new KeyConfigReque
stDetail().setDeletionAllowed(true));
transit.deleteKey(KEY_NAME);
```

Vault TOTP

TOTP secret engine is supported by using `io.quarkus.vault.VaultTOTPSecretEngine` class:

```
@Inject
VaultTOTPSecretEngine vaultTOTPSecretEngine;

CreateKeyParameters createKeyParameters = new CreateKeyPara
meters("Google", "test@gmail.com");
createKeyParameters.setPeriod("30m");

/** Generate Key (QR code) */
final Optional<KeyDefinition> myKey = vaultTOTPSecretEngine
                        .createKey("my_
key_2", createKeyParameters);

/** Generate key number to login */
final String keyCode = vaultTOTPSecretEngine.generateCode(
"my_key_2");

/** Login logic */
boolean valid = vaultTOTPSecretEngine.validateCode("my_key_
2", keyCode);
```

Vault Provisioning

Vault extension offers façade classes to Vault provisioning functions:

```java
@Inject
VaultSystemBackendEngine vaultSystemBackendEngine;

@Inject
VaultKubernetesAuthService vaultKubernetesAuthService;

String rules = "path \"transit/*\" {\n" +
        " capabilities = [ \"create\", \"read\", \"updat
e\" ]\n" +
        "}";
String policyName = "sys-test-policy";

vaultSystemBackendEngine.createUpdatePolicy(policyName, rul
es);

vaultKubernetesAuthService
    .createRole(roleName, new VaultKubernetesAuthRole()
        .setBoundServiceAccountNames(boundServiceAccountN
ames)
        .setBoundServiceAccountNamespaces(boundServiceAcc
ountNamespaces)
        .setTokenPolicies(tokenPolicies));
```

Vault configuration properties. Prefix `quarkus.vault` is skipped.

url
    Vault server URL

authentication.client-token
    Vault token to access

authentication.app-role.role-id
    Role Id for AppRole auth

authentication.app-role.secret-id
    Secret Id for AppRole auth

authentication.app-role.secret-id-wrapping-token
    Wrapping token containing a Secret Id. `secret-id` and `secret-id-wrapping-token` are exclusive.

authentication.userpass.username
    Username for userpass auth

authentication.userpass.password
    Password for userpass auth

authentication.userpass.password-wrapping-token
    Wrapping token containing a password. `password` and `password-wrapping-token` are exclusive.

authentication.kubernetes.role
    Kubernetes authentication role

authentication.kubernetes.jwt-token-path
    Location of the file containing the Kubernetes JWT token

renew-grace-period
    Renew grace period duration (default: `1H`)

secret-config-cache-period
    Vault config source cache period (default: `10M`)

secret-config-kv-path
    Vault path in kv store. List of paths is supported in CSV

log-confidentiality-level
    Used to hide confidential infos. `low`, `medium`, `high` (default: `medium`)

kv-secret-engine-version
    Kv secret engine version (default: 1)

kv-secret-engine-mount-path Kv secret engine path (default: `secret`)

tls.skip-verify
    Allows to bypass certificate validation on TLS communications (default: `false`)

tls.ca-cert
    Certificate bundle used to validate TLS communications

tls.use-kubernetes-ca-cert
    TLS will be active (default: `true`)

connect-timeout
    Tiemout to establish a connection (default: `5S`)

read-timeout
    Request timeout (default: `1S`)

credentials-provider."credentials-provider".database-credentials-role
    Database credentials role

credentials-provider."credentials-provider".kv-path
    A path in vault kv store, where we will find the kv-key

credentials-provider."credentials-provider".kv-key
    Key name to search in vault path kv-path (default: `password`)

Amazon KMS

```
mvn quarkus:add-extension
    -Dextensions="amazon-kms"
```

```java
@Inject
KmsClient kms;

kms.encrypt(req -> req.keyId(keyArn).plaintext(
        SdkBytes.fromUtf8String(data))).ciphertextBlob();
```

```
quarkus.kms.endpoint-override=http://localhost:8011
quarkus.kms.aws.region=us-east-1
quarkus.kms.aws.credentials.type=static
quarkus.kms.aws.credentials.static-provider.access-key-id=t
est-key
quarkus.kms.aws.credentials.static-provider.secret-access-k
ey=test-secret
```

You need to set a HTTP client either `URL Connection`:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>url-connection-client</artifactId>
</dependency>
```

or Apache HTTP:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
</dependency>
```

```
quarkus.sqs.sync-client.type=apache
```

You can go async by using Mutiny:

```java
@Inject
software.amazon.awssdk.services.kms.KmsAsyncClient kms;

Uni.createFrom().completionStage(
    kms.encrypt(req -> req.keyId(keyArn).plaintext(SdkByte
s.fromUtf8String(data))
    ))
```

And you need to add the asynchronous Netty client:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```

Configuration properties are the same as Amazon DynamoDB but changing the prefix from `dynamodb` to `kms`.

Amazon IAM

```
mvn quarkus:add-extension
    -Dextensions="quarkus-amazon-iam"
```

You need to set a HTTP client either `URL Connection`:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>url-connection-client</artifactId>
</dependency>
```

or Apache HTTP:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
</dependency>
```

And you need to add the asynchronous Netty client:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```

```java
@Inject
IamClient client;

@Inject
IamAsyncClient async;
```

```
quarkus.iam.endpoint-override=${iam.url}
quarkus.iam.aws.region=us-east-1
quarkus.iam.aws.credentials.type=static
quarkus.iam.aws.credentials.static-provider.access-key-id=test-key
quarkus.iam.aws.credentials.static-provider.secret-access-key=test-secret
```

Configuration properties are the same as Amazon DynamoDB but changing the prefix from `dynamodb` to `iam`.

HTTP Configuration

You can configure HTTP parameters. Using `quarkus.http` prefix:

cors
Enable CORS. (default: `false`)

cors.origins
CSV of origins allowed. (dedault: Any request valid.)

cors.methods
CSV of methods valid. (default: Any method valid.)

cors.headers
CSV of valid allowed headers. (default: Any requested header valid.)

cors.exposed-headers
CSV of valid exposed headers.

port
The HTTP port. (default: `8080`)

test-port
The HTTP test port. (default: `8081`)

host
The HTTP host. (default: `0.0.0.0`)

host-enabled
Enable listening to host:port. (default: `true`)

ssl-port
The HTTPS port. (default `8443`)

test-ssl-port
The HTTPS port used to run tests. (default `8444`)

proxy-address-forwarding
The address, scheme etc will be set from headers forwarded by the proxy server.

allow-forwarded
Proxy address forwarding is enabled then the standard `Forwarded` header will be used, rather than the more common but not standard `X-Forwarded-For`.

insecure-requests
If insecure requests are allowed. Possible values: `enabled`, `redirect`, `disable`. (default: `enabled`)

http2
Enables HTTP/2. (default: `true`)

ssl.port
The *HTTPS* port. (default: `8443`)

ssl.certificate.file
The file path to a service certificate or certificate chain in *PEM* format. Relative to `src/main/resources`.

ssl.certificate.key-file
The file path to the corresponding certificate private key in *PEM* format. Relative to `src/main/resources`.

ssl.certificate.key-store-file
The key store contains the certificate information. Relative to `src/main/resources`.

ssl.certificate.key-store-file-type
The key store type. It is automatically detected based on the file name or can be set manually. Supported values are: `JKS`, `JCEKS`, `P12`, `PKCS12` or `PFX`.

ssl.certificate.key-store-password
The password to open the key store file.

`ssl.certificate.trust-store-file` The trust store location which contains the certificate information of the certificates to trust. Relative to `src/main/resources`.

ssl.certificate.trust-store-file-type
The trust store type. It is automatically detected based on the file name or can be set manually.

ssl.certificate.trust-store-password
The password to open the trust store file.

ssl.cipher-suites
A list of strings of cipher suites to use. If not provided, a reasonable default is selected.

ssl.protocols
The list of protocols to explicitly enable. (default: `TLSv1.3` and `TLSv1.2`).

ssl.client-auth
Configures the engine to require/request client authentication. Possible values are: `none`, `request` and `required`. (default: `none`).

io-threads
The number if IO threads used to perform IO.

limits.max-header-size
The maximum length of all headers. (default: `20k`)

limits.max-body-size
The maximum size of a request body. (default: `10M`)

limits.max-chunk-size
The max HTTP chunk size.

limits.max-initial-line-length
The maximum length of the initial line. (default: `4096`)

idle-timeout

enable-compression
If responses should be compressed.

read-timeout
Http connection read timeout for blocking IO. (default: `60s`)

body.handle-file-uploads
If the files sent using `multipart/form-data` will be stored locally. (default: `true`)

body.uploads-directory
The directory where the files sent using `multipart/form-data` should be stored. (default: `file-uploads`)

body.merge-from-attributes
If the form attributes should be added to the request parameters. (default: `true`)

body.delete-uploaded-files-on-end
If the uploaded files should be removed after serving the request.

body.preallocate-body-buffer
If the body buffer should pre-allocated based on the `Content-Length` header value. (default: `1K`)

auth.session.encryption-key
The encryption key that is used to store persistent logins.

so-reuse-port
Enable socket reuse port.

tcp-quick-ack
Enable tcp quick ack.

tcp-cork
Enable tcp cork.

tcp-fast-open
Enable tcp fast open.

domain-socket
Path to a unix domain socket. (default: `/var/run/io.quarkus.app.socket`)

domain-socket-enabled
Enables domain socket.

record-request-start-time
If enabled then start time will be recorded to enable logging of total request time. (default: `false`)

access-log.enabled
If access logging is enabled. (default: `false`)

access-log.pattern
The access log pattern. (default: `common`)

If logging should be done to a separate file. (default: `false`)

access-log.base-file-name
The access log file base name. (default: `quarkus`)

access-log.log-directory
The log directory to use when logging access to a file.

access-log.log-directory
The log directory to use when logging access to a file.

access-log.log-suffix
The log file suffix. (default: `.log`)

access-log.category
The log category to use if logging is being done via the standard log mechanism. (default: `io.quarkus.http.access-log`)

access-log.rotate
If the log should be rotated daily. (default: `true`)

same-site-cookie.\<name\>.case-sensitive
If the cookie pattern is case sensitive.

same-site-cookie.\<name\>.value
The value to set in the samesite attribute.

same-site-cookie.\<name\>.enable-client-checker
Some User Agents break when sent SameSite=None, this will detect them and avoid sending the value. (default: `true`)

same-site-cookie.\<name\>.add-secure-for-none
If this is true then the 'secure' attribute will automatically be sent on cookies with a SameSite attribute of None. (default: `true`)

If metrics extension is registered, you can enable to get HTTP metrics by setting `quarkus.resteasy.metrics.enabled` to true.

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.

```java
@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:

@PathParam
Gets content from request URI. (example: `/book/{id}` `@PathParam("id")`)

@QueryParam
Gets query parameter. (example: `/book?desc=""` `@QueryParam("desc")`)

@FormParam
Gets form parameter.

@MatrixParam
Get URI matrix parameter. (example: `/book;author=mkyong;country=malaysia`)

@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.

```java
@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.

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

Possible injectable objects: `SecurityContext`, `Request`, `Application`, `Configuration`, `Providers`, `ResourceContext`, `ServletConfig`, `ServletContext`, `HttpServletRequest`, `HttpServletResponse`, `HttpHeaders`, `Urinfo`, `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.

```java
@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 `ExceptionMapper` interface:

```java
@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();
    }

}
```

Caching

Annotations to set Cache-Control headers:

```java
@Produces(MediaType.APPLICATION_JSON)
@org.jboss.resteasy.annotations.cache.NoCache
public User me() {}


@Produces(MediaType.APPLICATION_JSON)
@org.jboss.resteasy.annotations.cache.Cache(
    maxAge = 2000,
    noStore = false
)
public User you() {}
```

Vert.X Filters and Routes

Programmatically

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

```java
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"
```

```java
@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();
    }

    @Route
    String hello(@Param Optional<String> name) {}

    @Route
    String helloFromHeader(@Header("My-Header") String header) {}

    @Route
    String createPerson(@Body Person person) {}

}
```

GraphQL

Quarkus integrates with GraphQL using MicroProfile GraphQL integration.

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

```java
@GraphQLApi
public class FilmResource {

    @Query("allFilms")
    public List<String> films() {
    }

    @Query
    public String getFilm(@Name("filmId") int id)) {}

    @Query
    public List<Hero> getHeroesWithSurname(
        @DefaultValue("Skywalker") String surname) {
    }

    @Mutation
    public Greetings load(Greetings greetings) {
    }
}
```

If name not provided, then query name is resolved from method name.

You can see the full schema at `/graphql/schema.graphql`. Also GraphiQL UI is enabled at dev and test mode at `/graphql-ui/`.

Extension can be configured with the follwoing paramters prefixed with `quarkus.smallrye-graphql`.

root-path

The rootPath under which queries will be served. (default: `/graphql`)

root-path-ui

The path where GraphQL UI is available. (default: `/graphql-ui`)

always-include-ui

The path where GraphQL UI is available. (default: `/graphql-ui`)

root-path-ui

Always include the UI. By default this will only be included in dev and test. (default: `false`)

enable-ui

If GraphQL UI should be enabled. (default: `false`)

metrics.enabled

Enable metrics. (default: `false`)

Vert.X Verticle

Vert.X Verticles are also supported:

```java
@ApplicationScoped
public class VerticleDeployer {

    @Inject
    Vertx vertx;

    public void init(@Observes StartupEvent ev) {
        CountDownLatch latch = new CountDownLatch(1);
        vertx.deployVerticle(BareVerticle::new,
            new DeploymentOptions()
                .setConfig(
                    new JsonObject()
                    .put("id", "bare")
                )
            )
            .thenAccept(x -> latch.countDown());

        latch.countDown();
    }
}
```

Verticles can be:

bare

extending `io.vertx.core.AbstractVerticle`.

mutiny

extendig `io.smallrye.mutiny.vertx.core.AbstractVerticle`.

GZip Support

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

gzip.enabled

EnableGZip. (default: `false`)

gzip.max-input

Configure the upper limit on deflated request body. (default: `10M`)

GRPC

Quarkus integrates with gRPC:

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

Then you need to configure `build tool` with gRPC plugins. In the case of Maven, the `kr.motd.maven:os-maven-plugin` extension and `org.xolstice.maven.plugins:protobuf-maven-plugin`

Protos files are stored at `src/main/proto`.

When `java` files are created two service implementations are provided: one with default gRPC API and other with Mutiny support.

With `quarkus.grpc.server` prefix, the next configuration properties can be set:

port

The gRPC Server port. (default: `9000`)

host

The gRPC server host. (default: `0.0.0.0`)

handshake-timeout

The gRPC handshake timeout.

max-inbound-message-size

The max inbound message size in bytes.

plain-text

Use plain text. (default: `true`)

alpn

TWhether ALPN should be used. (default: `true`)

enable-reflection-service

Enables the gRPC Reflection Service. (default: `false`)

ssl.certificate

The file path to a server certificate or certificate chain in PEM format.

Column 1

ssl.key

The file path to the corresponding certificate private key file in PEM format.

ssl.key-store

An optional key store which holds the certificate information instead of specifying separate files.

ssl.key-store-type

An optional parameter to specify the type of the key store file.

ssl.key-store-password

A parameter to specify the password of the key store file. (default: `password`)

ssl.trust-store

Trust store which holds the certificate information of the certificates to trust

ssl.trust-store-type

Parameter to specify type of the trust store file.

ssl.trust-store-password

A parameter to specify the password of the trust store file.

ssl.cipher-suites

A list of the cipher suites to use.

ssl.protocols

The list of protocols to explicitly enable. (default: `TLSv1.3,TLSv1.2`)

transport-security.certificate

The path to the certificate file.

transport-security.key

The path to the private key file.

To consume the service:

```
@GrpcService("hello")
GreeterGrpc.GreeterBlockingStub client;

@GrpcService("hello")
io.grpc.Channel channel;
```

Some configuration example to set the host and the SSL parameters:

```
quarkus.grpc.clients.hello.host=localhost
quarkus.grpc.clients.hello.plain-text=false
quarkus.grpc.clients.hello.ssl.certificate=src/main/resources/tls/client.pem
quarkus.grpc.clients.hello.ssl.key=src/main/resources/tls/client.key
quarkus.grpc.clients.hello.ssl.trust-store=src/main/resources/tls/ca.pem
```

Column 2

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(){}
```

Column 3

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 `/q/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 {

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

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

    private boolean isReady() {}
}
```

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

```
@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.

vault
A probe to check Vault conection status.

gRPC
A readiness probe for the gRPC services.

Cassandra
A readiness probe to check Cassandra connection status.

Redis
A readiness probe to check Redis connection status.

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
```

In the case of Vault you can pass parameters to modify the call of the `status` endpoint in Vault.

```
quarkus.vault.health.enabled=true
quarkus.vault.health.stand-by-ok=true
quarkus.vault.health.performance-stand-by-ok=true
```

Health groups are supported to provide custom health checks groups:

```
@io.smallrye.health.HealthGroup("mygroup1")
public class SimpleHealthGroupCheck implements HealthCheck
{
}
```

You can ping grouped health checks by querying `/group/mygroup1`.

Group root path can be configured:

```
quarkus.smallrye-health.group-path=/customgroup
```

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 `/q/metrics` providing default metrics.

MicroProfile Metrics annotations:

`@Timed`
Tracks the duration.

`@SimplyTimed`
Tracks the duration without mean and distribution calculations.

`@Metered`
Tracks the frequency of invocations.

`@Counted`
Counts number of invocations.

`@Gauge`
Samples the value of the annotated object.

`@ConcurrentGauge`
Gauge to count parallel invocations.

`@Metric`
Used to inject a metric. Valid types `Meter`, `Timer`, `Counter`, `Histogram`. `Gauge` 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 historgram;
```

You can configure Metrics:

```
quarkus.smallrye-metrics.path=/mymetrics
```

Prefix is `quarkus.smallrye-metrics.`

`path`
The path to the metrics handler. (default: `/q/metrics`)

`extensions.enabled`
Metrics are enabled or not. (default: `true`)

`micrometer.compatibility`
Apply Micrometer compatibility mode. (default: `false`)

`quarkus.hibernate-orm.metrics.enabled` set to `true` exposes Hibernate metrics under `vendor` scope.

`quarkus.mongodb.metrics.enabled` set to `true` exposes MongoDB metrics under `vendor` scope.

You can apply metrics annotations via CDI stereotypes:

```
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD })
@Timed(name = "checksTimer",
description = "A measure of how long it takes
                  to perform a hello.",
unit = MetricUnits.MILLISECONDS)
public @interface TimedMilliseconds {
}
```

There is a tight integration with Micrometer in the form of an extension:

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

Add a micrometer dependency for the registry of your choosing:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

You can configure Micrometer. Prefix is `quarkus.micrometer`:

`enabled`
Micrometer metrics support. (default: `true`)

`registry-enabled-default`
Micrometer MeterRegistry discovery. (default: `true`)

`binder-enabled-default`
Micrometer MeterBinder discovery. (default: `true`)

`binder.vertx.enabled`
Vert.x metrics support.

`binder.mp-metrics.enabled`
Microprofile Metrics support.

`binder.jvm`
Micrometer JVM metrics support. (default: `true`)

`binder.system`
Micrometer System metrics support. (default: `true`)

`export.datadog.enabled`
Support for export to Datadog Support for Datadog.

`export.jmx.enabled`
Support for export to JMX Support for JMX.

`export.prometheus.enabled`
Support for export to Prometheus.

export.prometheus.path

The path for the prometheus metrics endpoint (produces text/plain). (default: `/q/metrics`)

export.azuremonitor.enabled

Support for export to Azure Monitor.

export.azuremonitor.instrumentation-key

The path for the azure monitor instrumentationKey.

export.statsd.enabled

Support for export to StatsD.

export.stackdriver.enabled

Micrometer metrics support. (default: `true`)

export.signalfx.enabled

Micrometer metrics support. (default: `true`)

export.signalfx.uri

Signalfx URI.

export.signalfx.access-token

Access Token.

binder.vertx.match-patterns

Comma-separated case-sensitive list of regular expressions defining Paths that should be matched and used as tags

binder.vertx.ignore-patterns

Comma-separated case-sensitive list of regular expressions defining Paths that should be ignored / not measured.

export.datadog

Datadog MeterRegistry configuration in `Map<String, String>` format.

export.jmx

JMX registry configuration properties in `Map<String, String>` format.

export.prometheus

Prometheus registry configuration properties in `Map<String, String>` format.

export.stackdriver

Stackdriver registry configuration properties in `Map<String, String>` format.

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
quarkus.jaeger.metrics.enabled=true
```

`@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.

You can log the `traceId`, `spanId` and `sampled` in normal log:

```
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId},
                                spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
```

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
```

Kafka Tracer

Adds a span for each message sent to or received from a Kafka topic.

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

And configure it:

```
...
mp.messaging.outgoing.generated-price.topic=prices

# For Produces
mp.messaging.outgoing.generated-price.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
...
# For consumers
mp.messaging.incoming.prices.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor
```

AWS XRay

If you are building native images, and want to use AWS X-Ray Tracing with your lambda you will need to include `quarkus-amazon-lambda-xray` as a dependency in your pom.

Native Executable

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`.

SSL

To create a native image with SSL you need to copy SunEC library and certificates:

Java 8:

```
FROM quay.io/quarkus/ubi-quarkus-native-image:{graalvm-version}-java8 as nativebuilder
RUN mkdir -p /tmp/ssl-libs/lib \
   && cp /opt/graalvm/jre/lib/security/cacerts /tmp/ssl-libs \
   && cp /opt/graalvm/jre/lib/amd64/libsunec.so /tmp/ssl-libs/lib/

FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY --from=nativebuilder /tmp/ssl-libs/ /work/
COPY target/*-runner /work/application
RUN chmod 775 /work /work/application
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0", "-Djava.library.path=/work/lib", "-Djavax.net.ssl.trustStore=/work/cacerts"]
```

Java 11:

```
FROM quay.io/quarkus/ubi-quarkus-native-image:{graalvm-version}-java11 as nativebuilder
RUN mkdir -p /tmp/ssl-libs/lib \
   && cp /opt/graalvm/lib/security/cacerts /tmp/ssl-libs \
   && cp /opt/graalvm/lib/libsunec.so /tmp/ssl-libs/lib/

FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY --from=nativebuilder /tmp/ssl-libs/ /work/
COPY target/*-runner /work/application
RUN chmod 775 /work /work/application
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0", "-Djava.library.path=/work/lib", "-Djavax.net.ssl.trustStore=/work/cacerts"]
```

Inclusion of resources

By default, no resources are included in native executable. `quarkus.native.resources.includes` allows to set glob expressions to include resources based on `src/main/resources` path.

Given `src/main/resources/foo/selected.png`:

```
quarkus.native.resources.includes=foo/**
```

Exclusion of resources

By default, no resources are excluded.

```
quarkus.native.resources.excludes = foo/**
```

Container Images Creation

You can levarage to Quarkus to generation and release of Docker containers. It provides several extensions to make it so.

```
mvn clean package
            -Dquarkus.container-image.build=true
            -Dquarkus.container-image.push=true
            -Dquarkus.container-image.registry=quay.io
```

Prefix is `quarkus.container-image`:

group
The group/repository of the image. (default: the `${user.name}`)

name
The name of the image. (default: the application name)

tag
The tag of the image. (default: the application version)

additional-tags
Additional tags of the container image.

registry
The registry to use for pushing. (default: `docker.io`)

username
The registry username.

password
The registry password.

insecure
Flag to allow insecure registries. (default: `false`)

build
Boolean to set if image should be built. (default: `false`)

push
Boolean to set if image should be pushed. (default: `false`)

Jib

```
./mvnw quarkus:add-extensions
      -Dextensions="quarkus-container-image-jib"
```

Quarkus copies any file under `src/main/jib` into the built container image.

Prefix is `quarkus.container-image-jib`:

base-jvm-image
The base image to use for the jib build. (default: `fabric8/java-alpine-openjdk8-jre`)

base-native-image
The base image to use for the native build. (default: `registry.access.redhat.com/ubi8/ubi-minimal`)

jvm-arguments
The arguments to pass to java. (default: `-Dquarkus.http.host=0.0.0.0,-Djava.util.logging.manager=org.jboss.logmanager.LogManager`)

native-arguments
The arguments to pass to the native application. (default: `-Dquarkus.http.host=0.0.0.0`)

environment-variables
Map of environment variables.

jvm-entrypoint
A custom entry point of the container image in JVM mode.

native-entrypoint
A custom entry point of the container image in native mode.

labels
Custom labels to add to the generated image.

base-registry-username

The username to use to authenticate with the registry used to pull the base JVM image.

base-registry-password

The password to use to authenticate with the registry used to pull the base JVM image.

ports

The ports to expose.

user

The user to use in generated image.

always-cache-base-image

Controls the optimization which skips downloading base image layers that exist in a target registry (default: `false`).

appcds-builder-image

When AppCDS generation is enabled, if this property is set, then the JVM used to generate the AppCDS file will be the JVM present in the container image.

Docker

```
./mvnw quarkus:add-extensions
    -Dextensions="quarkus-container-image-docker"
```

Prefix is `quarkus.container-image-s2i`:

dockerfile-jvm-path

Path to the JVM Dockerfile. (default: `${project.root}/src/main/docker/Dockerfile.jvm`)

dockerfile-native-path

Path to the native Dockerfile. (default: `${project.root}/src/main/docker/Dockerfile.native`)

S2I

```
./mvnw quarkus:add-extensions
    -Dextensions="quarkus-container-image-s2i"
```

Prefix is `quarkus.container-image-docker`:

base-jvm-image

The base image to use for the s2i build. (default: `fabric8/java-alpine-openjdk8-jre`)

base-native-image

The base image to use for the native build. (default: `registry.access.redhat.com/ubi8/ubi-minimal`)

Kubernetes

Quarks can use **Dekorate** to generate Kubernetes resources.

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

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

💡 **Container Images Creation** integrates with Kubernetes extension, so no need of extra Kubernetes properties.

Generated resource is integrated with **MicroProfile Health** annotations.

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

```
quarkus.kubernetes.namespace=mynamespace

quarkus.kubernetes.replicas=3

quarkus.kubernetes.labels.foo=bar

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

quarkus.kubernetes.mounts.github-token.path=/deployment/github
quarkus.kubernetes.mounts.github-token.read-only=true

quarkus.kubernetes.secret-volumes.github-token.volume-name=github-token
quarkus.kubernetes.secret-volumes.github-token.secret-name=greeting-security
quarkus.kubernetes.secret-volumes.github-token.default-mode=420

quarkus.kubernetes.config-map-volumes.github-token.config-map-name=my-secret

quarkus.kubernetes.expose=true

quarkus.kubernetes.env.vars.my-env-var=foobar
quarkus.kubernetes.env.configmaps=my-config-map,another-config-map
quarkus.kubernetes.env.secrets=my-secret,my-other-secret

quarkus.kubernetes.resources.requests.memory=64Mi
quarkus.kubernetes.resources.requests.cpu=250m
quarkus.kubernetes.resources.limits.memory=512Mi
quarkus.kubernetes.resources.limits.cpu=1000m
```

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

Labels and Annotations

The generated manifest use the Kubernetes recommended labels and annotations.

```
"labels" : {
    "app.kubernetes.io/part-of" : "todo-app",
    "app.kubernetes.io/name" : "todo-rest",
    "app.kubernetes.io/version" : "1.0-rc.1"
}

"annotations": {
    "app.quarkus.io/vcs-url" : "<some url>",
    "app.quarkus.io/commit-id" : "<some git SHA>",
}
```

You can override the labels by using the next properties:

```
quarkus.kubernetes.part-of=todo-app
quarkus.kubernetes.name=todo-rest
quarkus.kubernetes.version=1.0-rc.1
```

Or add new labels and/or annotations:

```
quarkus.kubernetes.labels.foo=bar
quarkus.kubernetes.annotations.foo=bar
```

metrics

When using metrics extension, Prometheus annotations are generated:

```
prometheus.io/scrape: "true"
prometheus.io/path: /metrics
prometheus.io/port: "8080"
```

Kubernetes Deployment Targets

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

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

Knative Properties

Most of the Kubernetes properties are valid in Knative output by just changing the `kubernetes` prefix to `knative` prefix (ie `quarkus.kubernetes.readiness-probe.period-seconds` to `quarkus.knative.readiness-probe.period-seconds`).

There are also specific properties for Knative:

```
quarkus.kubernetes.deployment-target=knative
quarkus.knative.revision-name=my-revision
quarkus.knative.traffic.my-revision.percentage=80
```

List of configuration options:

kubernetes

openshift

https://quarkus.io/guides/kubernetes#openshift

Knative

https://quarkus.io/guides/kubernetes#knative

Using Existing Resources

You an provide your Kubernetes resources in form of yaml/json and they will provide additional resources or be used as base for the generation process:

Resources are added in `src/main/kubernetes` directory with the target name (`kubernetes.json`, `openshift.json`, `knative.json`, or the yml equivalents) with one orm ore Kubernetes resources. Any resource found will be added in the generated manifests. If one of the provided resources has the same name as one of the generated ones, then the generated resource will be created on top of the provided resource, respecting existing content.

To override the name of the generated resource you can use: `quarkus.kubernetes.name`, `quarkus.openshift.name` and `quarkus.knative.name`.

Deployment

To deploy automatically the generated resources, you need to set `quarkus.container.deploy` flag to `true`.

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

💡 If you set this flag to `true`, the `build` and `push` flags from container-image are set to `true` too.

To deploy the application, the extension uses the https://github.com/fabric8io/kubernetes-client. All options described at Kubernetes Client are valid here.

Minikube

Quarkus has a Minikube extension which creates Kubernetes manifests that are tailored for Minikube.

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

💡 Remember to execute `eval $(minikube -p minikube docker-env)` to build images directly inside Minkube cluster.

OpenShift

Instead of adding Kubernetes extension, set container image s2i and the `target` to `openshift` for working with OpenShift, an extension grouping all of the is created:

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

Kubernetes Configuration Extension

Integration between MicroProfile Config spec and ConfigMaps & Secrets:

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

```
quarkus.kubernetes-config.enabled=true
quarkus.kubernetes-config.config-maps=cmap1,cmap2
```

```
@ConfigProperty(name = "some.prop1")
String someProp1;

@ConfigProperty(name = "some.prop2")
String someProp2;
```

If the config key is a Quarkus configuration file `application.properties`/`application.yaml`, the content is parsed and each key of the configuration file is used as config property.

List of Kubernetes Config parameters.

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

enabled
The application will attempt to look up the configuration from the API server. (default: `false`)

fail-on-missing-config
The application will not start if any of the configured config sources cannot be located. (default: `true`)

config-maps
ConfigMaps to look for in the namespace that the Kubernetes Client has been configured for. Supports CSV.

namespace
Access to ConfigMaps from a specific namespace.

secrets.enabled
Whether or not configuration can be read from secrets. (default: `false`)

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.

trust-certs
Trust self-signed certificates. (default: `false`)

master-url
URL of Kubernetes API server.

namesapce
Default namespace.

ca-cert-file
CA certificate data.

client-cert-file
Client certificate file.

client-cert-data
Client certificate data.

client-key-data
Client key data.

client-key-algorithm
Client key algorithm.

client-key-passphrase
Client key passphrase.

username
Username.

password
Password.

watch-reconnect-interval
Watch reconnect interval. (default: `PT1S`)

watch-reconnect-limit
Maximum reconnect attempts. (default: `-1`)

connection-timeout
Maximum amount of time to wait for a connection. (default: `PT10S`)

request-timeout
Maximum amount of time to wait for a request. (default: `PT10S`)

rolling-timeout
Maximum amount of time to wait for a rollout. (default: `PT15M`)

http-proxy
HTTP proxy used to access the Kubernetes.

https-proxy
HTTPS proxy used to access the Kubernetes.

proxy-username
Proxy username.

proxy-password

Proxy password.

no-proxy

IP addresses or hosts to exclude from proxying

Or programmatically:

```java
@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:

```java
@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`.

```java
@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();
    }
}
```

AWS 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.

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

    }
}
```

The interface `com.amazonaws.services.lambda.runtime.RequestStreamHandler` is also supported.

The interface `com.amazon.ask.SkillStreamHandler` is also supported.

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:

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

```java
@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}
```

Funqy

Quarkus Funqy is part of Quarkus's serverless strategy and aims to provide a portable Java API to write functions deployable to various FaaS environments like AWS Lambda, Azure Functions, Knative, and Knative events.

```java
public class GreetingFunction {

    @Inject
    GreetingService service;

    @io.quarkus.funqy.Funq
    public String greet(String name) {}

    @io.quarkus.funqy.Funq("HelloCustomer")
    public String greet(Customer name) {}

    @Funq
    public Greeting greet(Friend friend,
        @io.quarkus.funqy.Context AwsContext ctx) {}
}
```

In case of Amazon Lambda, only one Funqy function can be exported per Amazon Lambda deployment. If there is only one method annotated with `@Funq` then no prob, if not, you need to set the function name with `quarkus.funqy.export` property.

Funqy HTTP

You can invoke on Funqy functions in a pure HTTP environment with simple adding the Funqy HTTP extension.

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-funqy-http</artifactId>
</dependency>
```

Funqy Cloud Events

Add the extension:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-funqy-knative-events</artifactId>
</dependency>
```

```java
@Funq
public String defaultChain(String input) {}
```

The Cloud Event type that triggers the function is `defaultChain`. It generates a response that triggers a new Cloud Event whose type is `defaultChain.output` and the event source is `defaultChain`.

It can be changed by using the next properties:

```
quarkus.funq.knative-events.mapping.defaultChain.trigger=configChain.output
quarkus.funq.knative-events.mapping.defaultChain.response-type=annotated
quarkus.funq.knative-events.mapping.defaultChain.response-source=configChain
```

The properties are of form: `quarkus.funqy.knative-events.mapping.{function name}..`

Also can be overriden with `@io.quarkus.funqy.knative.events.CloudEventMapping` annotation.

```java
@Funq
@CloudEventMapping(trigger = "annotated", responseSource = "annotated", responseType = "lastChainLink")
public String annotatedChain(String input) {}
```

`responseType` chains `annotatedChain` response to `lastChainLink` function.

```java
@Funq
public void lastChainLink(String input,
            @Context io.quarkus.funqy.knative.events.CloudEvent event) {}
```

A K-Native Trigger looks like:

```yaml
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: defaultchain
spec:
  filter:
    attributes:
      type: defaultChain
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: funqy-knative-events-quickstart
```

And to `curl` from inside the Kubernetes cluster:

```
curl -v "http://default-broker.knativetutorial.svc.cluster.local" \
-X POST \
-H "Ce-Id: 1234" \
-H "Ce-Specversion: 1.0" \
-H "Ce-Type: defaultChain" \
-H "Ce-Source: curl" \
-H "Content-Type: application/json" \
-d '"Start"'
```

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:

```java
@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
```

Possible Swagger UI options with `quarkus.swagger-ui` prefix:

urls

The urls that will be included as options. (ie `quarkus.swagger-ui.urls.petstore=https://petstore.swagger.io/v2/swagger.json`)

urls-primary-name

If urls option is used, this will be the name of the default selection.

title

The html title for the page.

theme

Swagger UI theme to be used.

footer

A footer for the html page. Nothing by default.

deep-linking

Enables deep linking for tags and operations.

display-operation-id

Controls the display of operationId in operations list.

default-models-expand-depth

The default expansion depth for models.

default-model-expand-depth

The default expansion depth for the model on the model-example section.

default-model-rendering

Controls how the model is shown when the API is first rendered.

display-request-duration

Controls the display of the request duration (in milliseconds) for "Try it out" requests.

doc-expansion

Controls the default expansion setting for the operations and tags.

filter

Enables filtering.

max-displayed-tags

Limits the number of tagged operations displayed to at most this many.

operations-sorter

Apply a sort to the operation list of each API. (`alpha`, `method`, `function`)

show-extensions

Controls the display of vendor extension.

show-common-extensions

Controls the display of extensions.

tag-sorter

Apply a sort to the tag list of each API.

on-complete

Provides a mechanism to be notified when Swagger UI has finished rendering a newly provided definition.

syntax-highlight

Set to false to deactivate syntax highlighting of payloads and cURL command.

oauth2-redirect-url

OAuth redirect URL.

request-interceptor

Function to intercept remote definition, "Try it out", and OAuth 2.0 requests.

request-curl-options

Array of command line options available to the curl command.

response-interceptor

Function to intercept remote definition, "Try it out", and OAuth 2.0 responses.

show-mutated-request

Uses the mutated request returned from a requestInterceptor to produce the curl command in the UI.

supported-submit-methods

List of HTTP methods that have the "Try it out" feature enabled.

validator-url

Swagger UI attempts to validate specs against swagger.io's online validator.

with-credentials

Enables passing credentials, as defined in the Fetch standard, in CORS requests that are sent by the browser.

model-property-macro

Function to set default values to each property in model.

parameter-macro

Function to set default value to parameters.

persist-authorization

It persists authorization data and it would not be lost on browser close/refresh.

layout

The name of a component available via the plugin system to use as the top-level layout for Swagger UI.

plugins

A list of plugin functions to use in Swagger UI.

presets

A list of presets to use in Swagger UI.

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 OpenAPIdocumentation 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`.

You can store generated OpenAPI schema if `quarkus.swagger-ui.store-schema-directory` is set.

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.reactive.ReactiveMailer` for asynchronous/reactive API.

```
@Inject
Mailer mailer;
```

And then you can use them to send an email:

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

Reactive Mail client

```
@Inject
ReactiveMailer reactiveMailer;

CompletionStage<Response> stage =
    reactiveMailer.send(
        Mail.withText("to@acme.org", "Subject", "Body")
    )
    .subscribeAsCompletionStage()
    .thenApply(x -> Response.accepted().build());
```

💡 If you are using `quarkus-resteasy-mutiny`, you can return `io.smallrye.mutiny.Uni` type.

`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=smtp.sendgrid.net
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.

Parameter	Default	Description
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 open connections .
mailer.own-host-name		Hostname for `HELO/EHLO` and 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.
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 `quarkus.ssl.native=true`.

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", delayed="1s")
    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
```

`org.quartz.Scheduler` can be injected as any other bean and scehdule jobs programmatically.

```
@Inject
org.quartz.Scheduler quartz;

quartz.scheduleJob(job, trigger);
```

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

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

```java
@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 integrates with JGit to integrate with Git repositories:

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

And then you can start using JGit:

```java
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`.

```java
@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`.

```java
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:

```java
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

```java
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"
```

```java
@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.

Quartz can be configured usinf the following properties with `quarkus.quartz` prefix:

clustered

Enable cluster mode or not.

store-type

The type of store to use. Possible values: `ram`, `db` (default: `ram`)

datasource

The name of the datasource to use.

table-prefix

The prefix for quartz job store tables. (default: `QRTZ_`)

trigger-listeners.<name>.class

Class name for the trigger.

trigger-listeners.<name>.property-name

The properties passed to the class.

job-listeners.<name>.class

Class name for the job.

job-listeners.<name>.property-name

The properties passed to the class.

plugins.<name>.class

Class name for the plugin.

plugins.<name>.property-name

The properties passed to the class.

instance-name

The name of the Quartz instance. (default: `QuarkusQuartzScheduler`)

thread-count

The size of scheduler thread pool. (default: `25`)

thread-priority

Thread priority of worker threads in the pool. (default: `5`)

force-start

The scheduler will be started even if no scheduled business methods are found.

start-mode

Scheduler can be started in different modes: `normal`, `forced` or `halted`. (default: `normal`)

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

```html
{@org.acme.Item item}
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title>
</head>
<body>
    <h1>{item.name ?: 'Unknown'}</h1>
    <h2>{str:reverse('Hello')}</h2>
    <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.

Including templates passing parameters:

```html
<html>
<head>
<meta charset="UTF-8">
<title>Simple Include</title>
</head>
<body>
  {#include foo limit=10 /}
</body>
</html>
```

To render the template:

```java
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"));
}

@TemplateExtension(namespace = "str")
public static class StringExtensions {
  static String reverse(String val) {
      return new StringBuilder(val).reverse().toString();
  }
}
```

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` and be static method. First parameter is used to match the base object (`Item`). `@TemplateExtension` can be used at class level:

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

Methods with multiple parameters are supported too:

```
{item.discountedPrice(2)}
```

```java
static BigDecimal discountedPrice(Item item, int scale) {
    return item.price.multiply(scale);
}
```

Qute `for` syntax supports any instance of `Iterable`, `Map.EntrySet`, `Stream` or `Integer`.

```
{#for i in total}
  {i}:
{/for}
```

The next map methods are supported:

```
{#for key in map.keySet}
{#for value in map.values}
{map.size}
{#if map.isEmpty}
{map['foo']}
```

The next list methods are supported:

```
{list[0]}
```

The next number methods are supported:

```
{#if counter.mod(5) == 0}
```

Switch/When

```
{#switch person.name}
  {#case 'John'}
    Hey John!
  {#case 'Mary'}
    Hey Mary!
{/switch}
```

```
{#when items.size}
  {#is 1} (1)
    There is exactly one item!
  {#is > 10} (2)
    There are more than 10 items!
  {#else} (3)
    There are 2 -10 items!
{/when}
```

Following operators can be used either in `when` and `switch`: `not`, `ne`, `!=`, `gt`, `>`, `ge`, `>=`, `lt`, `<`, `le`, `<=`, `in`, `ni`, `!in`.

Message Bundling

```java
@io.quarkus.qute.i18n.MessageBundle
public interface AppMessages {
    @io.quarkus.qute.i18n.Message("Hello {name}!")
    String hello_name(String name);
}
```

There are 3 methods to inject the message:

```java
MessageBundles.get(AppMessages.class).hello_name("Lucie");
```

or

```java
@Inject AppMessages app;

app.hello_name("Lucie");
```

or

```html
<p>{msg:hello_name('Lucie')}</p>
```

Localization

There are two ways to set localized message:

```java
@io.quarkus.qute.i18n.Localized("de")
public interface GermanAppMessages {

    @Override
    @io.quarkus.qute.i18n.Message("Hallo {name}!")
    String hello_name(String name);
}
```

or

msg_de.properties

```
hello_name=Hallo {name}!
```

You can render programmaticaly the templates too:

```java
// 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();
```

Value Resolvers

Value resolvers are invoked when evaluating expressions.

```java
void configureEngine(@Observes EngineBuilder builder) {
    builder.addValueResolver(ValueResolver.builder()
        .appliesTo(ctx -> ctx.getBase() instanceof Long && ctx.getName().equals("tenTimes"))
        .resolveSync(ctx -> (Long) ctx.getBase() * 10)
        .build());
}
```

Content Filters

Content filters can be used to modify the template contents before parsing.

```java
void configureEngine(@Observes EngineBuilder builder) {
    builder.addParserHook(new ParserHook() {
        @Override
        public void beforeParsing(ParserHelper parserHelper) {
            parserHelper.addContentFilter(contents -> contents.replace("${", "$\\{"));
        }
    });
}
```

Reactive and Async

```java
CompletionStage<String> async = report.renderAsync();
Multi<String> publisher = report.createMulti();

Uni<String> content = io.smallrye.mutiny.Uni.createFrom()
                        .completionStage(() -> report.renderAsync());
```

Qute Mail Integration

```java
@Inject
MailTemplate hello;

CompletionStage<Void> c = hello.to("to@acme.org")
    .subject("Hello from Qute template")
    .data("name", "John")
    .send();
```

INFO: Template located at `src/main/resources/templates/hello.[html|txt]`.

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.server-name
  Sets the server name that will be sent with each event.

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

JSch

Quarkus integrates with Jsch for SSH communication.

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

```
JSch jsch = new JSch();
Session session = jsch.getSession(null, host, port);
session.setConfig("StrictHostKeyChecking", "no");
session.connect();
```

Cache

Quarkus can cache method calls by using as key the tuple (method + arguments).

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

```
@io.quarkus.cache.CacheResult(cacheName = "weather-cache")
public String getDailyForecast(LocalDate date, String city)
{}
```

`@CacheInvalidate` removes the element represented by the calculated cache key from cache. `@CacheInvalidateAll` removes all entries from the cache. `@CacheKey` to specifically set the arguments

```
@ApplicationScoped
public class CachedBean {

    @CacheResult(cacheName = "foo")
    public Object load(Object key) {}

    @CacheInvalidate(cacheName = "foo")
    public void invalidate(Object key) {}

    @CacheInvalidateAll(cacheName = "foo")
    public void invalidateAll() {}
}
```

You can disable the caching system by setting `quarkus.cache.enabled` property to `false`.

This extension uses Caffeine as its underlying caching provider.

Each cache can be configured individually:

```
quarkus.cache.caffeine."foo".initial-capacity=10
quarkus.cache.caffeine."foo".maximum-size=20
quarkus.cache.caffeine."foo".expire-after-write
quarkus.cache.caffeine."bar".maximum-size=1000
```

Full list of configuration properties having `quarkus.cache.caffeine.[cache-name]` as prefix:

initial-capacity
  Minimum total size for the internal data structures.

maximum-size
  Maximum number of entries the cache may contain.

expire-after-write
  Specifies that each entry should be automatically removed from the cache once a fixed duration has elapsed after the entry's creation, or last write.

expire-after-access
  Specifies that each entry should be automatically removed from the cache once a fixed duration has elapsed after the entry's creation, or last write.

💡 You can multiple cache annotations on a single method.

If you see a `javax.enterprise.context.ContextNotActiveException`, you need to add the `quarkus-smallrye-context-propagation` extension.

Banner

Banner is printed by default. It is not an extension but placed in the core.

quarkus.banner.path
  Path is relative to root of the classpath. (default:

quarkus.banner.enabled
  Enables banner. (default : `true`)

OptaPlanner

Quarkus integrates with OptaPlanner.

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

```
@PlanningSolution
public class TimeTable {
}

@Inject
private SolverManager<TimeTable, UUID> solverManager;

UUID problemId = UUID.randomUUID();
SolverJob<TimeTable, UUID> solverJob = solverManager.solve(problemId, problem);
TimeTable solution = solverJob.getFinalBestSolution();
```

Possible configuration options prefixed with `quarkus.optaplanner`:

solver-config-xml
  A classpath resource to read the solver configuration XML. Not mandatory.

solver.environment-mode
  Enable runtime assertions to detect common bugs in your implementation during development. Possible values: `FAST_ASSERT`, `FULL_ASSERT`, `NON_INTRUSIVE_FULL_ASSERT`, `NON_REPRODUCIBLE`, `REPRODUCIBLE`. (default: `REPRODUCIBLE`)

solver.move-thread-count
  Enable multithreaded solving for a single problem. Possible values: `MOVE_THREAD_COUNT_NONE`, `MOVE_THREAD_COUNT_AUTO`, a number or formula based on the available processors. (default: `MOVE_THREAD_COUNT_NONE`)

solver.termination.spent-limit
  How long the solver can run. (ie `5s`)

solver.termination.unimproved-spent-limit
  How long the solver can run without finding a new best solution after finding a new best solution. (ie `2h`)

solver.termination.best-score-limit
  Terminates the solver when a specific or higher score has been reached. (ie `0hard/-1000soft`)

solver-manager.parallel-solver-count
  The number of solvers that run in parallel. (default: `PARALLEL_SOLVER_COUNT_AUTO`)

Context Propagation

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

If using `mutiny` extension together you already get context propagation for ArC, RESTEasy and transactions. With `CompletionStage` you need to:

```
@Inject ThreadContext threadContext;
@Inject ManagedExecutor managedExecutor;

threadContext.withContextCapture(..)
        .thenApplyAsync(r -> {}, managedExecutor);
```

If you are going to use security in a reactive environment you will likely need Smallrye Content Propagation to propagate the identity throughout the reactive callback.

Configuration from HasiCorp Consul

You can read runtime configuration from HashiCorp Consul.

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

You need to configure Consul:

```
quarkus.consul-config.enabled=true
quarkus.consul-config.agent.host-port=localhost:8500
quarkus.consul-config.properties-value-keys=config/consul-test
```

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

In Consul:

```
"Key": "config/consul-test",
"Value": "greeting.message=Hello from Consul"
```

Possible configuration parameters, prefixed with `quarkus.consul-config`:

enabled
  The application will attempt to look up the configuration from Consul. (default: `false`)

prefix
  Common prefix that all keys share when looking up the keys from Consul. The prefix is **not** included in the keys of the user configuration

raw-value-keys

Keys whose value is a raw string. The keys that end up in the user configuration are the keys specified her with '/' replaced by '.'

properties-value-keys
  Keys whose value represents a properties-like file conttent.

fail-on-missing-key
  If the application will not start if any of the configured config sources cannot be located. (default: `true`)

trust-store
  TrustStore to be used containing the SSL certificate used by Consul agent.

trust-store-password
  Password of TrustStore to be used containing the SSL certificate used by Consul agent.

key-password
  Password to recover key from KeyStore for SSL client authentication with Consul agent.

agent.host-port
  Consul agent host. (default: `localhost:8500`)

agent.use-https
  Use HTTPS when communicating with the agent. (default: `false`)

agent.token
  Consul token to be provided when authentication is enabled.

agent.key-store
  KeyStore (classpath or filesystem) to be used containing the SSL certificate used by Consul agent.

agent.key-store-password
  Password of KeyStore.

agent.trust-certs
  To trust all certificates or not.

agent.connection-timeout
  Connection timeout. (default: `10S`)

agent.read-timeout
  Reading timeout. (default: `60S`)

Amazon Alexa

You can use Amazon Alexa by adding the extension:

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

WebJar Locator

To change how you can refer to webjars skipping the version part you can use WebJars locator extension.

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

Then the JavaScript location is changed from `/webjars/jquery/3.1.1/jquery.min.js` to `/webjars/jquery/jquery.min.js` in your HTML files.

Amazon SES

```
mvn quarkus:add-extension
    -Dextensions="amazon-ses"
```

```
@Inject
software.amazon.awssdk.services.ses.SesClient sesClient;

@Inject
software.amazon.awssdk.services.ses.SesAsyncClient sesClient;
```

```
quarkus.ses.endpoint-override=http://localhost:8012
quarkus.ses.aws.region=us-east-1
quarkus.ses.aws.credentials.type=static
quarkus.ses.aws.credentials.static-provider.access-key-id=test-key
quarkus.ses.aws.credentials.static-provider.secret-access-key=test-secret
```

You need to set a HTTP client either `URL Connection`:

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>url-connection-client</artifactId>
</dependency>
```

or Apache HTTP:

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
</dependency>
```

```
quarkus.ses.sync-client.type=apache
```

Or async:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```

Configuration properties are the same as Amazon DynamoDB but changing the prefix from `dynamodb` to `ses`.

Jbang

Creating an initial script:

```
jbang scripting/quarkusapp.java
```

Adding Quarkus dependencies in script:

```
//DEPS io.quarkus:quarkus-resteasy:{quarkus-version}
```

Put some Quarkus CLI code:

```java
@Path("/hello")
@ApplicationScoped
public class quarkusapp {
    @GET
    public String sayHello() {
        return "hello";
    }
    public static void main(String[] args) {
        Quarkus.run(args);
    }
}
```

To run the script:

```
jbang quarkusapp.java
```

A Maven goal is provided to scaffold a project:

```
mvn io.quarkus:quarkus-maven-plugin:<version>:create-jbang
```

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.

```java
@Configuration
public class AppConfiguration {

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

Or as a component:

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

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

```java
@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.

```java
@Component
public class GreeterBean {

    private final MessageProducer messageProducer;

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

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

Spring Boot Configuration

Quarkus provides a compatibility layer for Spring Boot `ConfigurationProperties`.

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

```java
@ConfigurationProperties("example")
public final class ClassProperties {

    private String value;
    private AnotherClass anotherClass;

    // getters/setters
}
```

```
example.value=class-value
example.anotherClass.value=true
```

Spring Cloud Config Client

Quarkus integrates Spring Cloud Config Client and MicroProfile Config spec.

```
./mvnw quarkus:add-extension
  -Dextensions="quarkus-spring-cloud-config-client"
```

You need to configure the extension:

```
quarkus.spring-cloud-config.uri=http://localhost:8089
quarkus.spring-cloud-config.username=user
quarkus.spring-cloud-config.password=pass
quarkus.spring-cloud-config.enabled=true
```

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

Prefix is `quarkus.spring-cloud-config.`

uri

Base URI where the Spring Cloud Config Server is available. (default: `localhost:8888`)

username

Username to be used if the Config Server has BASIC Auth enabled.

password

Password to be used if the Config Server has BASIC Auth enabled.

enabled

enabled

Enables read configuration from Spring Cloud Config Server. (default: `false`)

fail-fast

True to not start application if cannot access to the server. (default: `false`)

connection-timeout

The amount of time to wait when initially establishing a connection before giving up and timing out. (default: `10s`)

read-timeout

The amount of time to wait for a read on a socket before an exception is thrown. (default: `60s`)

label

The label to be used to pull remote configuration properties.

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:1.12.0.Final: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);

@Query(value = "SELECT COUNT(*), publicationYear FROM Book
 GROUP BY publicationYear")
List<BookCountByYear> findAllByPublicationYear2();

interface BookCountByYear {
    int getPublicationYear();

    Long getCount();
}
```

What is currently unsupported:

- Methods of `org.springframework.data.repository.query.QueryByExampleExecutor`

- QueryDSL support

- Customizing the base repository

- `java.util.concurrent.Future` as return type

- Native and named queries when using `@Query`

Spring Data Rest

While users are encouraged to use REST Data with Panache for the REST data access endpoints generation, Quarkus provides a compatibility layer for Spring Data REST in the form of the `spring-data-rest` extension.

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

```
import java.util.Optional;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

@RepositoryRestResource(exported = false, path = "/my-fruits")
public interface FruitsRepository extends CrudRepository<Fruit, Long> {
    @RestResource(exported = true)
    Optional<Fruit> findById(Long id);
    @RestResource(exported = true)
    Iterable<Fruit> findAll();
}
```

The `spring-data-jpa` extension will generate an implementation for this repository. Then the `spring-data-rest` extension will generate a REST CRUD resource for it.

The following interfaces are supported:

- `org.springframework.data.repository.CrudRepository`

- `org.springframework.data.repository.PagingAndSortingRepository`

- `org.springframework.data.jpa.repository.JpaRepository`

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()")`

- 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) {}
```

Spring Cache

Quarkus provides a compatibility layer for Spring dependency injection.

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

```
@org.springframework.cache.annotation.Cacheable("someCache")
public Greeting greet(String name) {}
```

Quarkus provides compatibility with the following Spring Cache annotations:

- `@Cacheable`

- `@CachePut`

- `@CacheEvict`

Spring Schedule

Quarkus provides a compatibility layer for Spring Scheduled annotation.

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

```
@org.springframework.scheduling.annotation.Scheduled(cron="*/5 * * * ?")
void cronJob() {
    System.out.println("Cron expression hardcoded");
}

@Scheduled(fixedRate = 1000)
@Scheduled(cron = "{cron.expr}")
```

Resources

- https://quarkus.io/guides/

- https://www.youtube.com/user/lordofthejars

Authors :

@alexsotob
Java Champion and Director of DevExp at Red Hat