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 1.13.0.Final**.

Getting Started

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

```mvn
top
dir
```

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

```java
@Path("/hello")
public class GreetingResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

**Packages**

```mvn
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.

```java
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
mvn clean package -Pnative
```

**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=appcds.jar 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.

```java
@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;
    }
}
```

**As Java main**:

```java
@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:

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

**Picocli**

You can use Picocli to implement CLI applications:

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

**Gradle**

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

```gradle
plugins {
    id 'java'
    id 'io.quarkus' version '0.26.1'
}
repositories {
    mavenCentral()
}
dependencies {
    implementation(enforcedPlatform("io.quarkus:quarkus-boo
m:0.26.1"))
    implementation("io.quarkus:quarkus-resteasy")
}
```

**Build**

```gradle
gradlew build
```

**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.jar myapp.jar`.
You can also configure CDI beans with PicoCLI arguments:

```java
public class HelloCommand implements Runnable {
  @CommandLine.Command
  public void run(String... args)
  {
    String name;
    System.out.println(parseResult.matchedOption("-c").getValue().toString());
  }
}
```

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, use `@QuarkusMain`:

```java
@QuarkusMain
@CommandLine.Command(name = "demo", mixinStandardHelpOption = true)
public class ExampleApp implements Runnable, QuarkusApplication {
  private final GreetingService greetingService;

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

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:list-extensions
```

**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
```

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:

```
public class HelloCommand implements Runnable {
  @CommandLine.Command
  public void run(String... args) {
    String name;
    System.out.println(parseResult.matchedOption("-c").getValue().toString());
  }
}
```

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, use `@QuarkusMain`:

```
@QuarkusMain
@CommandLine.Command(name = "demo", mixinStandardHelpOption = true)
public class ExampleApp implements Runnable, QuarkusApplication {
  private final GreetingService greetingService;

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

**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=""
```

**Application Lifecycle**

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

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

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

**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=""
```

**Application Lifecycle**

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

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

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

**Dev Mode**

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.

```
@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 `GREETING_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
```

**Configuration Profiles**

Array, List, and Set are supported. The delimiter is comma `(`), and `\` is the escape char.
Quarkus allow you to have multiple configuration in the same file 
(application.properties).

The syntax for this is @profile).config.key=value.

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

### Additional locations

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

```java
smallrye.config.locations=conf.properties
```

or

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

```java
@ConfigProperties(prefix = "greeting", 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:

```java
@ConfigProperties(prefix = "greeting", 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:
In case of subkeys ~ is used to refer to the unprefix part.

```java
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 the `Converter` interface:

```java
@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.

```java
@ApplicationScoped
public class GreetingService {
    @Inject
    GreetingService greetingService;

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

Scope annotation is mandatory to make the bean discoverable.

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

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

### Qualifiers

You can use qualifiers to return different implementations of the same interface or to customize the configuration of the bean.
Quarkus breaks the CDI spec by allowing you to inject qualified beans without using @Inject annotation.

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.

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.

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

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

Any POJO is marshaled/unmarshalled automatically.

```java
public class Sauce {
    private String name;
    private long scovilleHeatUnits;
    // getter/setters
}
```

JSON equivalent:

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

In a POST endpoint example:

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

To provide custom JsonbConfig object:

```java
@Lock
@ApplicationScoped
class SharedService {
    void addAmount(BigDecimal amount) {
    }

    @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.

To work with Jackson you need to add:

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

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

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

    @Produces
    @IfBuildProperty(name = "some.tracer.enabled", stringValue = "true")
    public Tracer noopTracer() {
        return new NoopTracer();
    }
}
```

```
message = new Message();
message = new Message();
message = new Message();
```
Quarkus uses Hibernate Validator to validate input/output of REST services and business services using Bean validation spec.

```java
public class CustomObjectMapperConfig {
    @ApplicationScoped
    @Produces
    public ObjectMapper objectMapper(Instance<ObjectMapperCustomizer> customizers) {
        ObjectMapper objectMapper = new ObjectMapper();
        // perform configuration
        for (ObjectMapperCustomizer customizer : customizers)
            customizer.customize(mapper);
        return objectMapper;
    }
}
```

Annotate POJO objects with validator annotations such as:

```java
@NotNull
@Digits
@Min
@Max
```

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.

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

You need to add a `ValidationMessages_ca_ES.properties` file:

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

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

```java
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:

```java
quarkus.log.console.enabled = true
quarkus.log.file.path = /path/to/file.log
quarkus.log.file.level = INFO
quarkus.log.console.color = true
quarkus.log.file.format = %d{yyyy-MM-dd HH:mm:ss,SSS} %h %N[%i] %-5p [%c{3.}] (%t) %s%e%n
```

**Prefix is quarkus.log.**

```java
quarkus.log.category."<category-name">.level
```

- **level**
  - Minimum level category (default: INFO)
  - 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} %h %N[%i] %-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**

- **syslog.enable**
  - Syslog logging is enabled (default: false)
- **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.host**
  - The hostname/IP of Logstash/Graylog. Prepend current hostname
- **syslog.app-name**
  - The app name used when formatting the message in RFC5424 format (default: current process name)
- **syslog.facility**
  - The name of the host the messages are being sent from (default: current hostname)
- **syslog.type**
  - Priority of the message as defined by RFC-S424 and RFC-3164 (default: USER_LEVEL)
- **syslog.protocol**
  - The syslog type of format message (default: RFC5424)
- **syslog.use-counting-framing**
  - Protocol used (default: TCP)
- **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:

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

@Inject
Logger log;

@LoggerName("foo")
Logger fLog;

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

### Gelf output

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

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

**handler.gelf.enabled**

- Enable GELF logging handler (default: false)

**handler.gelf.host**


**handler.gelf.port**

- The port. (default: 12201)

**handler.gelf.version**

- GELF version. (default: 1.1)

**handler.gelf.extract-stack-trace**

- Post StackTrace to StackTrace field. (default: true)

**handler.gelf.stack-trace-throwable-reference**

- Gets the cause level to stack trace. 0 is full stack trace. (default: false)
**JSON output**

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

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

**Rest Client**

Quarkus implements MicroProfile Rest spec:

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

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

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

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

And configure the hostname at `application.properties`:

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

**Asynchronous**

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

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

**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 method on a client interface can return a `io.smallrye.mutiny.Uni` instance.

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

A RESTEasy Reactive-based REST Client extension. You only need to replace the `quarkus-rest-client` to `quarkus-rest-client-reactive`. 

**Multipart**

You can still use the JAX-RS client without any problem:

```
ClientBuilder.newClient().target(…)
```
It is really easy to send multipart form-data with Rest Client.

```java
<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.PartType;
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.MultipartForm;
@Path("/echo")
@RegisterRestClient
public interface MultipartService {
  @POST
  @Consumes(MediaType.MULTIPART_FORM_DATA)
  @Produces(MediaType.TEXT_PLAIN)
  String sendMultipartData(@MultipartForm MultipartBody data);
}
```

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.Transaction` annotation.

QuarkusTestProfile

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

⚠️ This implies that the Quarkus service is restarted.

SSL

You can configure Rest Client key stores.

```java
org.acme.quickstart.WorldClockService/mp-rest/trustStore=
  classpath:/store.jks
org.acme.quickstart.WorldClockService/mp-rest/trustStorePassword=
  supersecret
```
public class MyProfile implements io.quarkus.test.junit.QuarkusTestProfile {
    @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 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.

```java
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:

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

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

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

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:

```java
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:

```java
@InjectMock
GreetingService greetingService;
```

```java
Mockito.when(greetingService.greet()).thenReturn("Hi");
```

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

Also spy is supported:

```java
@InjectSpy
GreetingService greetingService;
```

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

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

```java
@ApplicationScoped
@RegisterRestClient
public interface GreetingService {
    // Method definitions...
}
```

```java
@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:

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

```java
@TransactionalQuarkusTest
public class TestStereotypeTestCase {}
```

Test Coverage

Due the nature of Quarkus to calculate correctly the coverage information with JaCoCo, you might need offline instrumentation.

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

Possible configuration parameters prefixed `quarkus.jacoco`:

- `data-file` The jacoco data file. (default: `jacoco-quarkus.exec`)
- `report` If Quarkus should generate the Jacoco report. (default: `true`)
- `output-encoding` Encoding of the generated reports. (default: `UTF-8`)
- `title` Name of the root node HTML report pages.
- `footer` Footer text used in HTML report pages.
- `source-encoding` Encoding of the source files. (default: `UTF-8`)

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.

```bash
./mvnw quarkus:add-extension
-Dextensions="io.quarkus:quarkus-jdbc-mariadb"
```

```java
@Entity
public class Developer extends PanacheEntity {
    // id field is implicit
    public String name;
}
```

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

```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.

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

**Database operations:**

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

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

// Hibernate Filters
Person.findById1().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.deleteBy1(1);

// 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...`
Static Methods

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

findByOptional: 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.

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

Named Queries

@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

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

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.

<dependency>
  <groupId>org.jboss.jandex</groupId>
  <artifactId>jandex</artifactId>
  <version>2.1.1.Final</version>
</dependency>

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:

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

@Test
public void testPanacheMocking()
{
    PanacheMock<Person.class>;
    Mockito.when((Person.class)).thenReturn(Person.class);
    Assertions.assertSame(23, PersonMockMockito.times(1).count());
    PanacheMock.verify(Person.class, Mockito.times(1));
}

DevServices

When testing or running in dev mode Quarkus can even provide you
with a zero config database out of the box. Depending on your
database type you may need docker installed in order to use this feature.

The following open source databases:

- Postresql (container)
You can register more than one datasource.

Multiple datasources

EntityManager

Also supports DAO pattern

Possible configuration values prefixed with `quarkus.datasource:`

`devservices`

If devservices is enabled or not. (default: true)

`devservices.image-name`

The container image name to use instead of the default one.

`devservices.properties`

Generic properties that are added to the database connection URL.

You can inject then

Your class:

`EntityManager` You can inject in your classes:

```
@Inject
EntityManager em;

@Inject
org.hibernate.Session session;

@Inject
org.hibernate.SessionFactory sessionFactory;

em.persist(car);
```

Transactions

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

Transactions are mandatory in case of none idempotent operations.

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

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

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

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

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

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

You can inject then

Quarkus Test Resource

AgroalDataSource

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

Flushing

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

```
flushing
```

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

Testing

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

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

And annotate the test:

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

```

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.

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

Transactions started; if an existing one was started, suspends it and starts a new one for the boundary of that method.

Transactions MANDATORY: fails if no transaction was started; works within the existing transaction otherwise.

Transactions SUPPORTS: if a transaction was started, joins it; otherwise works with no transaction.

Transactions 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.

Transactions NEVER: if a transaction was started, raises an exception; otherwise works with no transaction.
If you need more dynamic approach implement: `@ApplicationScoped
io.quarkus.hibernate.orm.runtime.tenant.TenantConnectionResolver
Hibernate Multitenancy
Multitenancy is supported using Schema or Database approach. First you need to define how tenant is identified:

**Schema approach**

```java
public class CustomTenantResolver implements TenantResolver {
    @Inject
    RoutingContext context;

    @Override
    public String getDefaultTenantId() {
        return "base";
    }

    @Override
    public String resolveTenantId() {
        return "dev";
    }
}
```

**Database approach**

```java
public class CustomTenantResolver implements TenantResolver {
    @Inject
    Config config;

    @ResourceProperties
    TenantResolver yResource = new TenantResolver();

    public interface DeveloperResource extends PanacheEntityResource {
        @MethodProperties(path = "my-developer")
        public Developer add(Developer)
    }

    public interface DeveloperRepository extends PanacheRepositoryBase<Developer, Long> {
        @MethodProperties(path = "my-developer")
        @Override
        public Developer add(Developer)
    }
```

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

You also need to define interfaces for defining endpoints:

In case of Active Record pattern:

```java
public interface DeveloperResource extends PanacheEntityResource {
    @ResourceProperties
    UserResource yResource = new UserResource();

    public interface DeveloperRepository extends PanacheRepositoryBase<Developer, Long> {
        @ResourceProperties
        TenantRepository yRepository = new TenantRepository();

        public interface DeveloperRepository extends PanacheRepositoryBase<Developer, Long> {
        }
```

In case of Repository:

```java
public interface DeveloperResource extends PanacheRepositoryBase<Developer, Long> {
    @ResourceProperties
    TenantRepository yRepository = new TenantRepository();

    public interface DeveloperRepository extends PanacheRepositoryBase<Developer, Long> {
```

Infinispan

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

```java
.quarkus.hibernate-orm.rest-data-panache
```
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;
}
```

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

```java
@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.

```java
package book_sample;

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

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

**In code**

Setting `proto` schema directly in a produced bean.

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

**Marshaller**


```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:

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

**Infinispan Embedded**

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

**Configuration in `infinispan.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;
```
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;
    });
}

@MultipleRedisClients
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.
client-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.

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 programmatically execute the migration.

```java
@Inject
LiquibaseFactory liquibaseFactory;

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

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

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

List of Liquibase parameters.

- `liquibase-catalog-name`: The liquibase tables catalog name.
- `liquibase-schemaname`: The liquibase tables schema name.
- `liquibase-tablename`: The liquibase tables tablename.
- `liquibase-change-log-lock-table-name`: The database change log lock table name. (default: `DATABASECHANGELOGLOCK`)
- `liquibase-change-log-table-name`: The database change log table name. (default: `DATABASECHANGELOG`)
- `liquibase-tablespace-name`: The liquibase tables spaces name.
- `liquibase-schema-name`: The liquibase tables schema name.
- `liquibase-catalog-name`: The liquibase tables catalog name.
- `clean-at-start`: The clean at start flag. (default: `false`)
- `validate-on-migrate`: The validate on update flag. (default: `false`)
- `validate-on-migrate`: The validate on update 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.

List of Liquibase parameters.

- `liquibase-catalog-name`: The liquibase tables catalog name.
- `liquibase-schemaname`: The liquibase tables schema name.
- `liquibase-tablename`: The liquibase tables tablename.
- `liquibase-change-log-lock-table-name`: The database change log lock table name. (default: `DATABASECHANGELOGLOCK`)
- `liquibase-change-log-table-name`: The database change log table name. (default: `DATABASECHANGELOG`)
- `liquibase-tablespace-name`: The liquibase tables spaces name.
- `liquibase-schema-name`: The liquibase tables schema name.
- `liquibase-catalog-name`: The liquibase tables catalog name.
- `clean-at-start`: The clean at start flag. (default: `false`)
- `validate-on-migrate`: The validate on update 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.

**Multiple Datasources**

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

```java
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.

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

```java
@Entity
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
  @KeywordField
  @FullTextField
  public List<Book> books;
}
```

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 context) {
    ctx.analyzer("english").custom()
      .withTokenize("standard")
      .withTokenFilters("asciifolding", "lowercase", "porter_stem");
    ctx.normalizer("sort").custom()
      .withTokenFilters("asciifolding", "lowercase");
  }
}
```

Use Hibernate Search in REST service:

```java
public class LibraryResource {
  @Outline(panache = true)
  public List<Author> searchAuthors(
    @QueryParam("pattern") String pattern) {
    return Search.getSearchSession(em)
      .search(Author.class)
      .predicate(f -> pattern.equals(f))
      .matching(pattern)
      .onFields("firstName", "lastName", "books.title")
      .fetchHits();
  }
}
```

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

Configure the extension in `application.properties`:

```properties
# quarkus.liquibase.inventory.schemas
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
```

It is not mandatory to use Panache.
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 want to consider the operation completed (paused, 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 (some, validate, update, create, drop-and-create, drop-and-create-drop)

index-defaults.lifecycle.required-status
Minimal cluster status (green, yellow, red)

index-defaults.lifecycle.required-status-wait-timeout
Waiting time before failing the bootstrap.

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

Possible annotations:

@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/:

To use asynchronous client with Mutiny:

To use it as a local DynamoDB instance:

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

Configuration parameters prefixed with quarkus.dynamodb:

Parameter | Default | Description
---|---|---
enable-endpoint-discovery | false | Endpoint discovery for a service API that supports endpoint discovery.

default-credentials-provider-chain: | | 

- System properties
- Env. Variables
- AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY
- Credentials profile
- Credentials through the Amazon EC2 container service if the AWS_CONTAINER CREDENTIALS_RELATIVE_URI is set
- Credentials through Amazon EC2 metadata service.
### Configuration parameters prefixed with `quarkus.dynamodb.aws`:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>api-call-timeout</code></td>
<td></td>
<td>Time to complete an execution.</td>
</tr>
<tr>
<td><code>endpoint-override</code></td>
<td></td>
<td>Configure the endpoint with which the SDK should communicate.</td>
</tr>
<tr>
<td><code>interceptors</code></td>
<td></td>
<td>List of class interceptors.</td>
</tr>
</tbody>
</table>

#### Region

Region that hosts DynamoDB.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>region</code></td>
<td></td>
<td>Region that hosts DynamoDB.</td>
</tr>
</tbody>
</table>

#### Credentials

Credentials that should be used:
- DEFAULT
- STATIC
- SYSTEM_PROPERTY
- ENV_VARIABLE
- PROFILE
- CONTAINER
- INSTANCE_PROFILE
- PROCESS
- ANONYMOUS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>credentials.type</code></td>
<td>DEFAULT</td>
<td>Credentials that should be used.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>profile</code></td>
<td></td>
<td>The name of the profile to use.</td>
</tr>
<tr>
<td><code>provider.profile-name</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>provider.profile</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Process

Command to execute to retrieve credentials.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>process</code></td>
<td></td>
<td>Command to execute to retrieve credentials.</td>
</tr>
<tr>
<td><code>provider.command</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>provider.process</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Max bytes to retrieve from process.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>output-limit</code></td>
<td>1024</td>
<td>Max bytes to retrieve from process.</td>
</tr>
</tbody>
</table>

The amount of time between credentials expire and credentials refreshed.

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

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

#### Connection Acquisition

Connection acquisition timeout.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection-acquisition-timeout</code></td>
<td>10S</td>
<td>Connection acquisition timeout.</td>
</tr>
</tbody>
</table>

Max time to connection to be opened.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection-max-idle-time</code></td>
<td>60S</td>
<td>Max time to connection to be opened.</td>
</tr>
<tr>
<td><code>connection-timeout</code></td>
<td></td>
<td>Connection timeout.</td>
</tr>
</tbody>
</table>

Max time connection to be open.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection-time-to-live</code></td>
<td>0</td>
<td>Max time connection to be open.</td>
</tr>
</tbody>
</table>

Time to wait for data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>socket-timeout</code></td>
<td>30S</td>
<td>Time to wait for data.</td>
</tr>
<tr>
<td><code>max-connections</code></td>
<td>50</td>
<td>Max connections.</td>
</tr>
</tbody>
</table>

Client send an HTTP Expect-Continue handshake.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expect-continue-enabled</code></td>
<td>true</td>
<td>Client send an HTTP Expect-Continue handshake.</td>
</tr>
</tbody>
</table>

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

#### TLS Managers

Path to key store.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tls-managers.provider.type</code></td>
<td>system-property</td>
<td>Path to key store.</td>
</tr>
<tr>
<td><code>tls-managers.provider.file.store.type</code></td>
<td></td>
<td>Key store type.</td>
</tr>
<tr>
<td><code>tls-managers.provider.file.store.password</code></td>
<td></td>
<td>Key store password.</td>
</tr>
</tbody>
</table>

Connections in pool should be closed asynchronously.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>use-idle-connection-reaper</code></td>
<td>true</td>
<td>Connections in pool should be closed asynchronously.</td>
</tr>
</tbody>
</table>

#### Proxy

Endpoint of the proxy server.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.endpoint</code></td>
<td></td>
<td>Endpoint of the proxy server.</td>
</tr>
</tbody>
</table>

Enables HTTP proxy.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.enabled</code></td>
<td>false</td>
<td>Enables HTTP proxy.</td>
</tr>
<tr>
<td><code>proxy.username</code></td>
<td></td>
<td>Proxy username.</td>
</tr>
<tr>
<td><code>proxy.password</code></td>
<td></td>
<td>Proxy password.</td>
</tr>
</tbody>
</table>

For NTLM, domain name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.ntlm-domain</code></td>
<td></td>
<td>For NTLM, domain name.</td>
</tr>
</tbody>
</table>

For NTLM, workstation name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.ntlm-workstation</code></td>
<td></td>
<td>For NTLM, workstation name.</td>
</tr>
</tbody>
</table>

Authenticate pre-emptively.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.preemptive-basic-authentication-enabled</code></td>
<td></td>
<td>Authenticate pre-emptively.</td>
</tr>
</tbody>
</table>

List of non proxy hosts.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proxy.non-proxy-hosts</code></td>
<td></td>
<td>List of non proxy hosts.</td>
</tr>
</tbody>
</table>

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

#### Socket Timeouts

Max time connection to be open.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection-time-to-live</code></td>
<td>0</td>
<td>Max time connection to be open.</td>
</tr>
</tbody>
</table>

Time to wait for data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>socket-timeout</code></td>
<td>30S</td>
<td>Time to wait for data.</td>
</tr>
<tr>
<td><code>max-connections</code></td>
<td>50</td>
<td>Max connections.</td>
</tr>
</tbody>
</table>

Client send an HTTP Expect-Continue handshake.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expect-continue-enabled</code></td>
<td>true</td>
<td>Client send an HTTP Expect-Continue handshake.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default-provider.async-credential-update-enabled</code></td>
<td>false</td>
<td>Should fetch credentials async.</td>
</tr>
<tr>
<td><code>default-provider.reuse-last-provider-enabled</code></td>
<td>true</td>
<td>Should reuse the last successful credentials.</td>
</tr>
</tbody>
</table>

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

#### Connection Acquisition

Connection acquisition timeout.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection-acquisition-timeout</code></td>
<td>10S</td>
<td>Connection acquisition timeout.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>connection-max-idle-time</td>
<td>60S</td>
<td>Max time to connection to be opened.</td>
</tr>
<tr>
<td>connection-timeout</td>
<td></td>
<td>Connection timeout.</td>
</tr>
<tr>
<td>connection-time-to-live</td>
<td>0</td>
<td>Max time connection to be open.</td>
</tr>
<tr>
<td>max-concurrency</td>
<td>50</td>
<td>Max number of concurrent connections.</td>
</tr>
<tr>
<td>use-idle-connection-reaper</td>
<td>true</td>
<td>Connections in pool should be closed asynchronously.</td>
</tr>
<tr>
<td>read-timeout</td>
<td>30S</td>
<td>Read timeout.</td>
</tr>
<tr>
<td>write-timeout</td>
<td>30S</td>
<td>Write timeout.</td>
</tr>
<tr>
<td>proxy.endpoint</td>
<td></td>
<td>Endpoint of the proxy server.</td>
</tr>
<tr>
<td>proxy.enabled</td>
<td>false</td>
<td>Enables HTTP proxy.</td>
</tr>
<tr>
<td>proxy.non-proxy-hosts</td>
<td></td>
<td>List of non proxy hosts.</td>
</tr>
<tr>
<td>tls-managers provider.type</td>
<td>system-property</td>
<td>TLS manager: none, system-property, file=store</td>
</tr>
<tr>
<td>tls-managers provider.file.store</td>
<td></td>
<td>Path to key store.</td>
</tr>
<tr>
<td>tls-managers provider.file.store.type</td>
<td></td>
<td>Key store type.</td>
</tr>
<tr>
<td>ssl-provider</td>
<td></td>
<td>SSL Provider ( jdk, openssl, openssl-ref), openssl:</td>
</tr>
</tbody>
</table>

### Amazon S3

- Sets the HTTP protocol.
- Max number of concurrent streams.
- Enable custom event loop conf.
- Number of threads to use in event loop.
- Prefix of thread names.

#### Amazon S3

You can inject asynchronous client too:

```java
@Inject
S3AsyncClient s3AsyncClient;
```

And you need to add the asynchronous Netty client:

```java
<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:

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

Configuration properties:

- `quarkus.neo4j.as` is skipped in the next table.
- Prefix is `quarkus.neo4j`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td></td>
<td>URI of Neo4j. (default: localhost:7687)</td>
</tr>
<tr>
<td>authentication.username</td>
<td></td>
<td>Username. (default: neo4j)</td>
</tr>
<tr>
<td>authentication.password</td>
<td></td>
<td>Password. (default: neo4j)</td>
</tr>
<tr>
<td>authentication.disabled</td>
<td></td>
<td>Disable authentication. (default: false)</td>
</tr>
</tbody>
</table>
As Neo4j uses SSL communication by default, to create a native executable you need to compile with next options GraalVM options:

-Dquarkus.native-image=H:+JNI

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

If you are using Neo4j 4.0, you can use fully reactive. Add the next extension: quarkus-reextasy-mutiny.

And Quarkus Maven Plugin with next configuration:

```xml
<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>
    </configuration>
  </execution>
</executions>
```

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

```java
public Publisher<String> get() {
  return Multi.createFrom().resource(driver::rxSession, 
    session -> session.readTransaction(tx -> {
      RxResult result = tx.run("MATCH (f:Fruit) RETURN f.name as name"); 
      return Multi.createFrom().publisher(result.records().map(record -> record.get("name").asString()).<
    }));
  }
```

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>`.

```java
@Get
Publisher<Optional<String>> get() {
  return Multi.createFrom().resource(driver::rxSession, 
    session -> session.readTransaction(tx -> {
      RxResult result = tx.run("MATCH (f:Fruit) RETURN f.name as name"); 
      return Multi.createFrom().publisher(result.records().map(record -> record.get("name").asString()).asStr
    }));
  }
```

### MongoDB Client

Quarkus integrates with MongoDB:

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

<insert code snippet here>

### Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection-string</td>
<td>String</td>
<td>MongoDB connection URI.</td>
</tr>
<tr>
<td>hosts</td>
<td>List&lt;String&gt;</td>
<td>Addresses passed as host:port.</td>
</tr>
<tr>
<td>application-name</td>
<td>String</td>
<td>Application name.</td>
</tr>
<tr>
<td>max-pool-size</td>
<td>Int</td>
<td>Maximum number of connections.</td>
</tr>
<tr>
<td>min-pool-size</td>
<td>Int</td>
<td>Minimum number of connections.</td>
</tr>
<tr>
<td>max-pool-idle-time</td>
<td>Duration</td>
<td>Idle time of a pooled connection.</td>
</tr>
<tr>
<td>max-connection-life-time</td>
<td>Duration</td>
<td>Life time of pooled connection.</td>
</tr>
<tr>
<td>max-connection-acquisition-time</td>
<td>Duration</td>
<td>Time period between runs of maintenance job.</td>
</tr>
<tr>
<td>max-connection-idle-time</td>
<td>Duration</td>
<td>Time to wait before running the first maintenance job.</td>
</tr>
<tr>
<td>wait-queue-multiple</td>
<td>Int</td>
<td>Multiplied with max pool-size gives max number of threads waiting.</td>
</tr>
<tr>
<td>wait-queue-timeout</td>
<td>Duration</td>
<td>Insecure TLS.</td>
</tr>
<tr>
<td>tls-insecure</td>
<td>boolean [false]</td>
<td>Enable TLS</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>replica-set-name</td>
<td>String</td>
<td>Implies hosts given are a seed list.</td>
</tr>
<tr>
<td>server-selection-timeout</td>
<td>Duration</td>
<td>Time to wait for server selection.</td>
</tr>
<tr>
<td>local-threshold</td>
<td>Duration</td>
<td>Minimum ping time to make a server eligible.</td>
</tr>
<tr>
<td>heartbeat-frequency</td>
<td>Duration</td>
<td>Frequency to determine the state of servers.</td>
</tr>
<tr>
<td>read-preference</td>
<td></td>
<td>Read preferences.</td>
</tr>
<tr>
<td>max-wait-queue-size</td>
<td>Int</td>
<td>Max number of concurrent operations allowed to wait.</td>
</tr>
<tr>
<td>write-concern.safe</td>
<td>boolean [true]</td>
<td>Ensures are writes are acknowledged.</td>
</tr>
<tr>
<td>write-concern.w</td>
<td>String</td>
<td>Value to all write commands.</td>
</tr>
<tr>
<td>write-concern.retrywrites</td>
<td>boolean [false]</td>
<td>Retry writes if network fails.</td>
</tr>
<tr>
<td>write-concern.w.timeout</td>
<td>Duration</td>
<td>Timeout to all write commands.</td>
</tr>
<tr>
<td>credentials.username</td>
<td>String</td>
<td>Username.</td>
</tr>
<tr>
<td>credentials.password</td>
<td>String</td>
<td>Password.</td>
</tr>
<tr>
<td>credentials.auth-source</td>
<td>String</td>
<td>Source of the authentication credentials.</td>
</tr>
<tr>
<td>credentials.auth-mechanism-properties</td>
<td>Map&lt;String, String&gt;</td>
<td>Authentication mechanism properties.</td>
</tr>
</tbody>
</table>

**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.

```java
@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.

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**

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}}

```java
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"), ("birth"));
long updated = Person.update("name", "Mortal").where("status", Status.Alive);
long countAll = Person.count();
Person.deleteByIId(id);
Person.delete("status", Status.Alive);
```
**Project**

Projects can be done for both PanacheQL and native queries.

**Testing**

To mock using active record pattern:

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

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

**Jandex**

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

**Deployment**

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

**MongoDB REST Data Panache**

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

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

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.

```java
<dependency>
 <groupId>com.datastax.oss.quarkus</groupId>
 <artifactId>cassandra-quarkus-client</artifactId>
 <scope>test</scope>
</dependency>
```

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

You need to create a DaoProducer:

```java
@ApplicationScoped
@Produces
public class FruitDaoProducer(QuarkusCqlSession session) {
    FruitMapper mapper = new FruitMapperBuilder(session).build();
    FruitDao = mapper.FruitDao();
}
```

**Cassandra configuration:**

```xml
<properties>
  <quarkus.cassandra.auth.username>s3cr3t</quarkus.cassandra.auth.username>
  <quarkus.cassandra.keyspace>system</quarkus.cassandra.keyspace>
  <quarkus.cassandra.contact-points>127.0.0.1:9042</quarkus.cassandra.contact-points>
</properties>
```

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

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

\[
\text{Reactive}
\]

You can also use Mutiny to define a reactive DAO:

```java
@Dao
public interface FruitDaoReactive {
    @Update
    Uni<Void> update(Fruit fruit);
    @Select
    MutinyMappedReactiveResultSet<Fruit> findById(String stor);
}
```

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

```java
@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.

```java
@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.

```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
@GET
@Produces(MediaType.TEXT_PLAIN)
public Uni<String> hello() {
    return Uni.createFrom().item() -> "hello";
}
```
Uni<Void> uniFromCompletable = Uni.createFrom().converter(UniRxConverters.fromCompletable(), completable);
Uni<String> uniFromSingle = Uni.createFrom().converter(UniRxConverters.fromSingle(), single);
Uni<String> uniFromObserverable = Uni.createFrom().converter(UniRxConverters.fromObserverable(), observable);
Uni<String> uniFromFlowable = Uni.createFrom().converter(UniRxConverters.fromFlowable(), flowable);

Uni<String> uniFromMono = Uni.createFrom().converter(UniReactorConverters.fromMono(), mono);
Uni<String> uniFromFlux = Uni.createFrom().converter(UniReactorConverters.fromFlux(), flux);

Multi<Void> multiFromCompletable = Multi.createFrom().converter(MultiRxConverters.fromCompletable(), completable);
Multi<String> multiFromObservable = Multi.createFrom().converter(MultiRxConverters.fromObserverable(), observable);
Multi<String> multiFromFlowable = Multi.createFrom().converter(MultiRxConverters.fromFlowable(), flowable);

Multi<Void> multiFromMono = Multi.createFrom().converter(MultiRxConverters.fromMono(), mono);
Multi<String> multiFromFlux = Multi.createFrom().converter(MultiRxConverters.fromFlux(), flux);

To RxJava2:
Completable completable = uni.convert().with(UniRxConverters.toCompletable());
Single<Optional<String>> single = uni.convert().with(UniRxConverters.toSingle());
Observable<String> observable = uni.convert().with(UniRxConverters.toObserverable());
Flowable<String> flowable = uni.convert().with(UniRxConverters.toFlowable());

CompletionStage<String> cs = uni.convert().subscribeAsCompletionStage();

To Reactor:
Mono<String> mono = uni.convert().with(UniReactorConverters.toMono());
Flux<String> flux = uni.convert().with(UniReactorConverters.toFlux());

CompletionStageFuture<String> future = Uni.createFrom().completionStage(CompletableFuture.supplyAsync(() -> "hello"));

Multi implements Publisher.

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

To Reactor:
Completable completable = uni.convert().with(UniReactorConverters.toCompletable());
Single<Optional<String>> single = uni.convert().with(UniReactorConverters.toSingle());
Observable<String> observable = uni.convert().with(UniReactorConverters.toObserverable());
Flowable<String> flowable = uni.convert().with(UniReactorConverters.toFlowable());

CompletionStage<String> cs = uni.convert().subscribeAsCompletionStage();

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

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

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 "Yaaaawwwnnnnn…";
    }
}

or in Mutiny:
REST API → Message

Patterns

REST API → Message

Message → Message

Message → SSE

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

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

You can also inject an stream as a field:

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

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

JSON-B Serializer/Deserializer

To customize:

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

To deserialized you need to extend `io.quarkus.kafka.client.serialization.JsonbDeserializer` and provide a type.

To integrate with Kafka you need to add next extensions:

Kafka

To integrate with Kafka you need to add next extensions:

AMQP

To integrate with AMQP you need to add next extensions:

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`
Kafka Streams

Create streaming queries with the Kafka Streams API.

```java
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.Topology;

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

All we need to do for that is to declare a CDI producer method which returns the Kafka Streams.

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

```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-values
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

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

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

---

**MQTT**

The connector type is `smallrye-mqtt`.

```properties
mp.messaging.outgoing.generated-price.connector=smallrye-mqtt
mp.messaging.outgoing.generated-price.address=prices
mp.messaging.outgoing.generated-price.port=1883
mp.messaging.outgoing.generated-price.auto-generated-client-id=true
mp.messaging.outgoing.generated-price.type=smallrye-mqtt
mp.messaging.outgoing.generated-price.topics=weather-stations,temperature-values
mp.messaging.outgoing.generated-price.host=localhost
mp.messaging.outgoing.generated-price.port=1883
mp.messaging.outgoing.generated-price.auto-generated-client-id=true
```

A complete list of supported properties for AMQP.

**AMQP**

```properties
mp.messaging.incoming.generated-price.connector=smallrye-amqp
mp.messaging.incoming.generated-price.address=prices
mp.messaging.incoming.generated-price.durable=true
mp.messaging.incoming.generated-price.type=smallrye-amqp
mp.messaging.incoming.generated-price.host=localhost
mp.messaging.incoming.generated-price.port=1883
mp.messaging.incoming.generated-price.auto-generated-client-id=true
```

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

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

**Reactive PostgreSQL Client**

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

```bash
./mvn quarkus:add-extension
-Dextensions="quarkus-reactive-pg-client"
```
Then you can inject `io.vertx.mutiny.pgsql.PgPool` class.

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.psqlclient.PgPool` class.

```
@Injet
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"
```

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

```
public class ArtemisConsumerManager {

    @PostConstruct
    private void init() throws JMSException {
        connection = serverLocator.createSessionFactory();
    }
    
    @Inject
    ServerLocator serverLocator;
    
    private String connectionURL;
    
    @PostConstruct
    public void init() throws Exception {
        connection = serverLocator.createSessionFactory();
        connection.start();
        
    }
    
    public void start ()
    {
        connectionFactory = connection.createConnectionFactory();
        
    }
```

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"
```

Then you can inject `io.vertx.mutiny.db2client.Db2Pool` class.

```
public void start ()
    {
        connectionFactory = connection.createConnectionFactory();
        
    }
```

### 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.

```
public class ArtemisConsumerManager {

    @PostConstruct
    private void init() throws JMSException {
        connection = serverLocator.createSessionFactory();
    }
    
    @Inject
    ServerLocator serverLocator;
    
    private String connectionURL;
    
    @PostConstruct
    public void init() throws Exception {
        connection = serverLocator.createSessionFactory();
        connection.start();
        
    }
```

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"
```

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

```bash
./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 you need to add the asynchronous Netty client:

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

You can go async by using Mutiny:

```java
@Inject
software.amazon.awssdk.services.sqs.AsyncClient 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:

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

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

Annotations

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

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

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

To alter RBAC behaviour there are two configuration properties:

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

Confuguration options:

```properties
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.

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>
```

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.

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

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

To alter RBAC behaviour there are two configuration properties:

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

Configuration options:

```properties
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.
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 (i.e. `quarkus.http.auth.permission.permitted.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-testing</artifactId>
  <scope>test</scope>
</dependency>
```

**BouncyCastle**

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

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

**JWT**

Quarkus implements MicroProfile JWT RBAC spec.

```java
@Test
void someTestMethod() {
  ...
}
```

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`.

```java
@Inject
JsonWebToken jwt;
```

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

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

Configuration options:

```java
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.verifyIssuer
  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 algorithm. (default: RS256)
smallrye.jwt.token.schemes
  Name of the cookie containing a token.
smallrye.jwt.encrypt.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
- JWKS
- JWKS 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` 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

JWKS groups Claim is directly mapped to roles to be used in security annotations.

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

OpenId Connect

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

```
mvn quarkus:add-extension
-Dextension="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 properties**

Additional properties which will be added as the query parameters.

**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 after logging out.

**Tls verification**
Sets the TLS verification. Possible values: REQUIRED, NONE (default: REQUIRED).

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.core property to enable consuming form different domain.

**Multi-tenancy**
Multi-tenancy is supported by adding a sub-category to OIDC configuration properties (e.g. quarkus.oidc.(tenant_id).property).

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

```
svn 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://auth-server/introspect
```

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

```java
@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)

**Authentication 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.

```
svn 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.roles-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`:

```
svn:
quarkus.security.users.file.enabled=true
quarkus.security.users.file.roles-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
```

**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`:
Security with a JDBC Realm

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

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

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.users.scott=jboss
quarkus.security.users.embedded.roles.scott=admin,tester,us er

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.

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

You need to configure JDBC and Elytron JDBC Realm:

principal-query.datasource
The index column containing the database. (default: none)

principal-query.datasource.driver
A string referencing the database driver. (default: org.h2.Driver)

principal-query.datasource.url
The index column containing the database url. (default: none)

principal-query.datasource.username
The index column containing the database username. (default: sa)

principal-query.datasource.password
The index column containing the database password. (default: sa)

principal-query.datasource.permissions.url
The index column containing the database permissions url. (default: none)

principal-query.datasource.permissions.username
The index column containing the database permissions username. (default: none)

principal-query.datasource.permissions.password
The index column containing the database permissions password. (default: none)

principal-query.datasource.permissions.properties
The index column containing the database permissions properties. (default: none)

principal-query.datasource.permissions.roles
The index column containing the database permissions roles. (default: none)

principal-query.datasource.permissions.roles.properties
The index column containing the database permissions roles properties. (default: none)

principal-query.datasource.roles
The index column containing the database roles. (default: none)

principal-query.datasource.roles.properties
The index column containing the database roles properties. (default: none)

principal-query.datasource.users
The index column containing the database users. (default: none)

principal-query.datasource.users.properties
The index column containing the database users properties. (default: none)

principal-query.datasource.users.roles
The index column containing the database users roles. (default: none)

principal-query.datasource.users.roles.properties
The index column containing the database users roles properties. (default: none)

principal-query.datasource.users.roles.permissions
The index column containing the database users roles permissions. (default: none)

principal-query.datasource.users.roles.permissions.properties
The index column containing the database users roles permissions properties. (default: none)

principal-query.datasource.users.roles.permissions.roles
The index column containing the database users roles permissions roles. (default: none)

principal-query.datasource.users.roles.permissions.roles.properties
The index column containing the database users roles permissions roles properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users
The index column containing the database users roles permissions roles users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.properties
The index column containing the database users roles permissions roles users properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles
The index column containing the database users roles permissions roles users roles. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.properties
The index column containing the database users roles permissions roles users roles properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users
The index column containing the database users roles permissions roles users roles users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.properties
The index column containing the database users roles permissions roles users roles users properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles
The index column containing the database users roles permissions roles users roles users roles. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.properties
The index column containing the database users roles permissions roles users roles users roles properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users
The index column containing the database users roles permissions roles users roles users roles users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.properties
The index column containing the database users roles permissions roles users roles users roles users properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles
The index column containing the database users roles permissions roles users roles users roles users roles. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.properties
The index column containing the database users roles permissions roles users roles users roles users roles properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users
The index column containing the database users roles permissions roles users roles users roles users roles users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.properties
The index column containing the database users roles permissions roles users roles users roles users roles users properties. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles
The index column containing the database users roles permissions roles users roles users roles users roles users roles. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users users. (default: none)

principal-query.datasource.users.roles.permissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users users users users. (default: none)

principal-query.datasource.users.rolespermissions.roles.users.roles.users.roles.users.roles.users.roles.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users.users
The index column containing the database users roles permissions roles users roles users roles users roles users roles users roles users users users users users users users users users users users users users users users. (default: none)

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

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

principal-query.datasource.enabled
The authentication mechanism. (default: BASIC)

principal-query.datasource.permission
The authentication realm name. (default: Quarkus)

principal-query.datasource.permission.enabled
If the properties store is enabled. (default: false)

principal-query.datasource.permission.enabled
The sql query to find the password.

principal-query.datasource.permission.enabled
The data source to use.

principal-query.datasource.permission.enabled
The index column containing clear password. (default:)

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

principal-query.datasource.permission.enabled
The index column containing password hash. (default:)

principal-query.datasource.permission.enabled
A string referencing the password hash encoding (BASE64 or HEX). (default: BASE64)
You can also protect endpoints and store identities in a database using LDAP.

Security with LDAP

You need to configure JDBC:

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

You can access the KV engine programmatically:

```java
@ConfigProperty(name = "a-private-key")
String privateKey;
@ConfigProperty(name = "singer.firstname")
String firstName;
```

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

```java
quarkus.security.ldap.enabled=true
quarkus.security.ldap.enabled=true
quarkus.security.ldap.enabled=true
quarkus.security.ldap.enabled=true
quarkus.security.ldap.enabled=true
```

### 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 `dir=quarkus,dc=io` and binding credentials `{uid=admin,ou=system", "secret"}`. Imports LDIF from a file located at root of the classpath named `quarkus-ldif.ldif`. Register dependency io.quarkus:quarkus-test-ldap:test.

And annotate the test:

```java
@QuarkusTestResource
class ElytronLdapExtensionTestResources {
}
```

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

```java
lime-name
The elytron realm name (default: Quarkus)
```

```java
direct-verification
Provided credentials are verified against LDAP (default: true)
```

```java
dir-context-url
The url of the LDAP server.
```

```java
dir-context-principal
User (bindim) which is used to connect to LDAP server.
```

```java
dir-context-password
The password (bindCredential) which belongs to the principal.
```

You can access the KV engine programmatically:

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

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

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

And configuring Vault in `application.properties`:

```java
# Vault URL
quarkus.vault.url=http://localhost:8200
quarkus.vault.authentication.userpass.username=bob
quarkus.vault.authentication.userpass.password=sinclair
```

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

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

You can also access Vault programmatically:

```java
vault kv put secret/myapps/vault-quickstart/config a-private-key=123456
```
Fetching credentials DB

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

No password is set as it is fetched from Vault.

Dynamic credentials are also supported:

Running the following dynamic database config in Vault:

```
<table>
<thead>
<tr>
<th>Vault server URL</th>
<th>Vault token to access</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication.client-token</td>
<td>Role Id for AppRole auth</td>
</tr>
<tr>
<td>authentication.app-role.secret-id</td>
<td>Secret Id for AppRole auth</td>
</tr>
<tr>
<td>authentication.app-role.secret-id-wrapping-token</td>
<td>Wrapping token containing a Secret Id. secret-id and secret-id-wrapping-token are exclusive.</td>
</tr>
<tr>
<td>authentication.userpass.username</td>
<td>Username for userpass auth</td>
</tr>
<tr>
<td>authentication.userpass.password</td>
<td>Password for userpass auth</td>
</tr>
<tr>
<td>authentication.userpass.password-wrapping-token</td>
<td>Wrapping token containing a password. password and password-wrapping-token are exclusive.</td>
</tr>
<tr>
<td>authentication.kubernetes.role</td>
<td>Kubernetes authentication role</td>
</tr>
<tr>
<td>authentication.kubernetes.jwt-token-path</td>
<td>Location of the file containing the Kubernetes JWT token</td>
</tr>
<tr>
<td>renewal-grace-period</td>
<td></td>
</tr>
</tbody>
</table>
```
Amazon KMS

Renew grace period duration (default: 1m)

**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-key**

A path in vault kv store, where we will find the kv-key

**credentials-provider."credentials-provider".kv-path**

Key name to search in vault path kv-path (default: password)

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>
```

or Apache HTTP:

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

You can go async by using Mutiny:

```java
@Inject
IamAsyncClient async;

@Inject
IamClient client;

@Timeout
async.
```

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 iam.

**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>
```

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

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

```java
@Inject
KmsClient kms;

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

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>
```
**HTTP Configuration**

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cors</code></td>
<td>Enable CORS. (default: false)</td>
</tr>
<tr>
<td><code>cors.origins</code></td>
<td>CSV of origins allowed. (default: Any request valid.)</td>
</tr>
<tr>
<td><code>cors.methods</code></td>
<td>CSV of methods valid. (default: Any method valid.)</td>
</tr>
<tr>
<td><code>cors.headers</code></td>
<td>CSV of valid allowed headers. (default: Any requested header valid.)</td>
</tr>
<tr>
<td><code>cors.exposed-headers</code></td>
<td>CSV of valid exposed headers.</td>
</tr>
<tr>
<td><code>port</code></td>
<td>The HTTP port. (default: 8080)</td>
</tr>
<tr>
<td><code>test-port</code></td>
<td>The HTTP test port. (default: 8081)</td>
</tr>
<tr>
<td><code>host</code></td>
<td>The HTTP host. (default: 0.0.0.0)</td>
</tr>
<tr>
<td><code>host-enabled</code></td>
<td>Enable listening to host. (default: true)</td>
</tr>
<tr>
<td><code>ssl-port</code></td>
<td>The HTTPS port. (default: 8443)</td>
</tr>
<tr>
<td><code>test-ssl-port</code></td>
<td>The HTTPS test port. (default: 8443)</td>
</tr>
<tr>
<td><code>proxy-address-forwarding</code></td>
<td>The address will be set from headers forwarded by the proxy server.</td>
</tr>
<tr>
<td><code>allow-forwarded</code></td>
<td>Proxy address forwarding is enabled when the standard <code>X-Forwarded-For</code> header will be used, rather than the more common but not standard <code>X-Forwarded-For</code>.</td>
</tr>
<tr>
<td><code>insecure-requests</code></td>
<td>If insecure requests are allowed. Possible values: enabled, redirect, disable. (default: enabled)</td>
</tr>
<tr>
<td><code>http2</code></td>
<td>Enables HTTP/2. (default: true)</td>
</tr>
<tr>
<td><code>ssl.port</code></td>
<td>The HTTPS port. (default: 8443)</td>
</tr>
</tbody>
</table>

**SSL Parameters**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssl.certificate.key-file</code></td>
<td>The file path to a service certificate or certificate chain in PEM format. Relative to <code>src/main/resources</code>.</td>
</tr>
<tr>
<td><code>ssl.certificate.key-store-file</code></td>
<td>The file path to the corresponding certificate private key in PEM format. Relative to <code>src/main/resources</code>.</td>
</tr>
<tr>
<td><code>ssl.certificate.key-store-type</code></td>
<td>The key store type. It is automatically detected based on the filename or name or can be set manually. Supported values are: JKS, JCEKS, P12, PKCS12 OF PFX.</td>
</tr>
<tr>
<td><code>ssl.certificate.key-store-password</code></td>
<td>The password to open the key store file.</td>
</tr>
<tr>
<td><code>ssl.certificate.trust-store-file</code></td>
<td>The trust store location which contains the certificate information of the certificates to trust. Relative to <code>src/main/resources</code>.</td>
</tr>
<tr>
<td><code>ssl.certificate.trust-store-type</code></td>
<td>The trust store type. It is automatically detected based on the filename or name or can be set manually.</td>
</tr>
<tr>
<td><code>ssl.certificate.trust-store-password</code></td>
<td>The password to open the trust store file.</td>
</tr>
<tr>
<td><code>ssl.cipher-suites</code></td>
<td>A list of strings of cipher suites to use. If not provided, a reasonable default is selected.</td>
</tr>
<tr>
<td><code>ssl.protocols</code></td>
<td>The list of protocols to explicitly enable. (default: TLSv1.3 and TLSv1.2).</td>
</tr>
<tr>
<td><code>ssl.client-auth</code></td>
<td>Configures the engine to require/request client authentication. Possible values are: none, request, and required. (default: none).</td>
</tr>
<tr>
<td><code>ssl.client-auth-bloodtype</code></td>
<td>The number of IO threads used to perform IO.</td>
</tr>
<tr>
<td><code>limits.max-headers</code></td>
<td>The maximum length of all headers. (default: 20k)</td>
</tr>
<tr>
<td><code>limits.max-body-size</code></td>
<td>The maximum size of a request body. (default: 10M)</td>
</tr>
<tr>
<td><code>limits.max-chunk-size</code></td>
<td>The maximum HTTP chunk size.</td>
</tr>
<tr>
<td><code>limits.max-initial-line-length</code></td>
<td>The maximum length of the initial line. (default: 4096)</td>
</tr>
<tr>
<td><code>idle-timeout</code></td>
<td>If responses should be compressed.</td>
</tr>
<tr>
<td><code>read-timeout</code></td>
<td>Http connection read timeout for blocking IO. (default: 60s)</td>
</tr>
<tr>
<td><code>body.handle-file-uploads</code></td>
<td>If the files sent using multipart/form-data will be stored locally. (default: true)</td>
</tr>
<tr>
<td><code>body.uploads-directory</code></td>
<td>The directory where the files sent using multipart/form-data should be stored. (default: file-uploads)</td>
</tr>
<tr>
<td><code>body.merge-from-attributes</code></td>
<td>If the form attributes should be added to the request parameters. (default: true)</td>
</tr>
<tr>
<td><code>body.delete-uploaded-files-on-end</code></td>
<td>If the uploaded files should be removed after serving the request.</td>
</tr>
<tr>
<td><code>body.preallocate-body-buffer</code></td>
<td>If the body buffer should pre-allocated based on the Content-Length header value. (default: true)</td>
</tr>
<tr>
<td><code>auth.session.encryption-key</code></td>
<td>The encryption key that is used to store persistent logins.</td>
</tr>
<tr>
<td><code>so-reuse-port</code></td>
<td>Enable socket reuse port.</td>
</tr>
<tr>
<td><code>tcp-quick-ack</code></td>
<td>Enable tcp quick ack.</td>
</tr>
<tr>
<td><code>tcp-cork</code></td>
<td>Enable tcp cork.</td>
</tr>
<tr>
<td><code>tcp-fast-open</code></td>
<td>Enable tcp fast open.</td>
</tr>
<tr>
<td><code>domain-socket</code></td>
<td>Path to a unix domain socket. (default: /var/run/io.quarkus.app.socket)</td>
</tr>
<tr>
<td><code>domain-socket-enabled</code></td>
<td>Enables domain socket.</td>
</tr>
<tr>
<td><code>record-request-start-time</code></td>
<td>If enabled then start time will be recorded to enable logging of total request time. (default: false)</td>
</tr>
<tr>
<td><code>access-log.enabled</code></td>
<td>If access logging is enabled. (default: false)</td>
</tr>
<tr>
<td><code>access-log.pattern</code></td>
<td>The access log pattern. (default: common)</td>
</tr>
</tbody>
</table>
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: false)

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 RESTful web APIs. Under the
covers, RestEASY 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.

### JAX-RS

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

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

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

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

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

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

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

    To get information from request:

    @PathParam
    Gets content from request URL. (example: /book/id)
    @PathParam("id")

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

    @FormParam
    Gets form parameter.

    @MatrixParam
    Gets 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.
```

### Implementing Exception Mapper

You can map exceptions to produce a custom output by implementing ExceptionMapper interface:
@Provider
public class ErrorMapper implements ExceptionMapper<Exception> {

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

Caching

Annotations to set Cache-Control headers:

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

Vert.X Filters and Routes

Programmatically

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

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

@ApplicationScoped
public class MyBean {
    public void filters(@Observes Filters filters) {
        filters.register(rc -> {
            rc.response().putHeader("X-Filter", "filter 1");
            rc.next();
        }, 10);
    }
    public void routes(@Observes Router router) {
        router.get("/")
                .handler(rc -> rc.response().end("OK"));
    }
}

Declarative

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

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

GraphQL

Quarkus integrates with GraphQL using MicroProfile GraphQL integration.

./mvnw quarkus:add-extension
-Dextensions="graphql"
Verticles can be:

- **Vert.X** A Verticle is also supported:

```
VerticleDeployer(name)
```

with `quarkus.vertx=verticle`.

**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)

**Enable GZip** (default: false)

**GZip Support**

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

- `gzip.enabled`
- `gzip.max-input`

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

Then you need to configure build tool with gRPC plugins. In the case of Maven, the next configuration properties are provided: one with default gRPC API and other with Mutiny support.

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-starter</artifactId>
    <version>1.44.1</version>
</dependency>
```

**GRPC**

Quarkus integrates with gRPC:

```
@GrpcService
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**
- **Vert.X**

```
 VerticleDeployer{name}
```

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

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

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

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

**sasl.key.store-password** A parameter to specify the password of the key store file. (default: password)

**sasl.trust-store** Trust store which holds the certificate information of the certificates to trust

**sasl.trust-store-type** Parameter to specify type of the trust store file.

**sasl.trust-store-password** A parameter to specify the password of the trust store file.

**sasl.cipher-suites** A list of the cipher suites to use.

**sasl.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;
@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.store=src/main/resources/tls/client.store
```
Fault Tolerance

Quarkus uses MicroProfile Fault Tolerance spec:

```
@Fallback(RecoverFallback.class)
WorldClock getNow(){}
```

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/now")
    @Produces(MediaType.APPLICATION_JSON)
    @Retry(maxRetries = 4, delay = 1000, retryOn = BulkheadException.class)
    WorldClock getNow();
}
```

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

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

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

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

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

And set it in the annotation as value `@Fallback(RecoverFallback.class)`. In case you want to use circuit breaker pattern:

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

If 3 (4 × 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
public WorldClock getNow() {};
```

For fault tolerance, Quarkus 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
public WorldClock getNow() {};
```

### Fault Tolerance Properties

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Timeout</td>
<td><code>unit</code></td>
</tr>
<tr>
<td>@Retry</td>
<td><code>maxRetries</code>, delay, delayUnit, maxDuration, durationUnit, jitter, jitterDelayUnit, retryOn, abortOn</td>
</tr>
<tr>
<td>@Bulkhead</td>
<td><code>waitingTaskQueue</code> (only valid in asynchronous)</td>
</tr>
<tr>
<td>@CircuitBreaker</td>
<td><code>requestVolumeThreshold</code>, <code>failureRatio</code>, <code>successThreshold</code></td>
</tr>
<tr>
<td>@Asynchronous</td>
<td></td>
</tr>
</tbody>
</table>

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/Bulkhead/maxDuration=3000

# Global
Retry/maxDuration=3000
```

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

### 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 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.

```
public class DatabaseHealthCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
        DatabaseHealthCheck 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:

```
org.acme.quickstart.WorldClock/getNow/Retry/enabled=false
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.
You can do it by setting health properties:

```java
public class SimpleHealthGroupCheck implements HealthCheck {
  @Liveness
  HealthCheck check1() {
    return io.smallrye.health.HealthStatus
        .up("successful-live");
  }
  @Readiness
  HealthCheck check2() {
    return HealthStatus
        .state("successful-read", this::isReady)
        .connection(true);
  }
  private boolean isReady() {}
}
```

Since health checks are CDI beans, you can do:

```java
@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)
        .connection(true);
  }
  private boolean isReady() {}
}
```

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

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

Health groups are supported to provide custom health checks groups:

```java
@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:

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

### Metrics

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

```java
./mvnw quarkus:add-extension
<extensions="io.quarkus:quarkus-smallrye-metrics"
```

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

Since health checks are CDI beans, you can do:

```java
@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)
        .connection(true);
  }
  private boolean isReady() {}
}
```

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

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

Health groups are supported to provide custom health checks groups:

```java
@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:

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

### Metrics

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

```java
./mvnw quarkus:add-extension
<extensions="io.quarkus:quarkus-smallrye-metrics"
```

The metrics can be read with JSON or the OpenMetrics format. An endpoint is registered automatically at `/g/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.
You can apply metrics annotations via CDI stereotypes:

```java
@Gauge
public description = "Hottest Sauce so far.",
unit = MetricUnits.NONE,
set to true exposes HottestSauce vendor scope.
```

```java
@Gauge
public description = "How many welcome have been performed.",
unit = MetricUnits.MILLISECONDS)
to perform a hello.
```

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

```java
@GET
@Path("/metrics")
@Timed
@Stereotype
public String hello() {}
```

Add a micrometer dependency for the registry of your choosing:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
  <version>1.7.0</version>
</dependency>
```

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

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

```
@GET
@Path("/metrics")
@Stereotype
public String hello() {}
```

Requests sent to any endpoint are traced automatically.
This extension includes OpenTracing support and Jaeger tracer.

Jaeger tracer configuration:

```java
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.

```java
@Inject
Tracer tracer;
```

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

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

You can log the traceId, spanId and samples in normal log:

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

### Additional Tracers

#### JDBC Tracer

Adds a span for each JDBC queries.

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

Configure JDBC driver apart from tracing properties seen before:

```java
# add 'tracing' to your database URL
quarkus.datasource.url=jdbc:tracing:postgresql://localhost:5432/mydatabase
```

### Kafka Tracer

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

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.

#### AWS XRay

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

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:

```bash
FROM quay.io/quarkus/ubi-quarkus-native-image:{graalvm-version}-java8
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
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
    command=["./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-javax-as-nativebuilder
RUN mkdir -p /tmp/ssl-libs/lib \
    && cp /opt/graalvm/lib/security/cacerts /tmp/ssl-libs/lib/ \
    && cp /opt/graalvm/lib/jre/lib/ssl/cacerts /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
COPY --from=nativebuilder --chown=quarkus:quarkus /tmp/ssl-libs/lib
WORKDIR /work/
FROM registry.access.redhat.com/ubi8/ubi-minimal
COPY src/main/resources/** /work
RUN chmod 775 /work
COPY target/*-runner /work/application
COPY --from=nativebuilder /tmp/ssl-libs /work
WORKDIR /work/
FROM quay.io/quarkus/ubi-quarkus-native-image:graalvm-version-javax-as-nativebuilder
COPY src/main/resources/** /work
EXPOSE 8080
CMD ["./application", 
    
    "-Dquarkus.http.host=0.0.0.0", 
    
    "-Djavax.net.ssl.trustStore=/work/cacerts",
    "-Djavax.net.ssl.trustStore=/work/cacerts"
]
EXPOSE 8080
CMD ["./application", 
    
    "-Dquarkus.http.host=0.0.0.0", 
    
    "-Djavax.net.ssl.trustStore=/work/cacerts",
    "-Djavax.net.ssl.trustStore=/work/cacerts"]

Inclusion of resources
By default, no resources are included in native executable.
quarkus.native.resources.includes = src/main/resources

Given
src/main/resources/foo/**
quarkus.native.resources.includes = foo/**

Exclusion of resources
By default, no resources are excluded.
quarkus.native.resources.excludes = foo/**

Native configuration properties prefixed with quarkus.native:
additional-build-args
Additional arguments to pass to the build process.

enable-http-url-handler
If the HTTP url handler should be enabled. (default: true)

enable-https-url-handler
If the HTTPS url handler should be enabled.

enable-all-security-services
If all security services should be added to the native image.

user-language
Defines the user language used for building the native executable.

user-country
Defines the user country used for building the native executable.

file-encoding
Defines the file encoding.

graalvm-home
The location of the Graal distribution.

java-home
The location of the JDK.

native-image-xmx
The maximum Java heap to be used during the native image generation.

debug-build-process
If the native image build should wait for a debugger to be attached before running.

publish-debug-build-process-port
If the debug port should be published when building with docker and debug-build-process. (default: true)

cleanup-server
If the native image server should be restarted.

enable-isolates
If isolates should be enabled. (default: true)

enable-fallback-images
If a JVM based ‘fallback image’ should be created if native image fails.

enable-server
If the native image server should be used.

debug enabled
If debug is enabled and debug symbols are generated.

auto-service-loader-registration
If all META-INF/services entries should be automatically registered.

dump-proxy
If the bytecode of all proxies should be dumped for inspection.

container-build
If this build should be done using a container runtime.

remote-container-build
If this build is done using a remote docker daemon.

builder-image
The docker image to use to do the image build.

container-runtime
The container runtime that is used to do an image based build.
   (docker, podman)

cleanup-server
If the native image server should be restarted.

debug-enabled
If debug is enabled and debug symbols are generated.

dump-proxy
If the bytecode of all proxies should be dumped for inspection.

graalvm-home
The location of the Graal distribution.

java-home
The location of the JDK.

native-image-xmx
The maximum Java heap to be used during the native image generation.

debug-build-process
If the native image build should wait for a debugger to be attached before running.

publish-debug-build-process-port
If the debug port should be published when building with docker and debug-build-process. (default: true)

cleanup-server
If the native image server should be restarted.

enable-isolates
If isolates should be enabled. (default: true)

enable-fallback-images
If a JVM based ‘fallback image’ should be created if native image fails.

enable-server
If the native image server should be used.

debug-enabled
If debug is enabled and debug symbols are generated.

auto-service-loader-registration
If all META-INF/services entries should be automatically registered.

dump-proxy
If the bytecode of all proxies should be dumped for inspection.

graalvm-home
The location of the Graal distribution.

java-home
The location of the JDK.

native-image-xmx
The maximum Java heap to be used during the native image generation.

debug-build-process
If the native image build should wait for a debugger to be attached before running.

publish-debug-build-process-port
If the debug port should be published when building with docker and debug-build-process. (default: true)

cleanup-server
If the native image server should be restarted.

enable-isolates
If isolates should be enabled. (default: true)

enable-fallback-images
If a JVM based ‘fallback image’ should be created if native image fails.

enable-server
If the native image server should be used.

debug-enabled
If debug is enabled and debug symbols are generated.

auto-service-loader-registration
If all META-INF/services entries should be automatically registered.

dump-proxy
If the bytecode of all proxies should be dumped for inspection.

graalvm-home
The location of the Graal distribution.

java-home
The location of the JDK.

native-image-xmx
The maximum Java heap to be used during the native image generation.

debug-build-process
If the native image build should wait for a debugger to be attached before running.

publish-debug-build-process-port
If the debug port should be published when building with docker and debug-build-process. (default: true)

cleanup-server
If the native image server should be restarted.

enable-isolates
If isolates should be enabled. (default: true)

enable-fallback-images
If a JVM based ‘fallback image’ should be created if native image fails.

enable-server
If the native image server should be used.

debug-enabled
If debug is enabled and debug symbols are generated.

auto-service-loader-registration
If all META-INF/services entries should be automatically registered.

dump-proxy
If the bytecode of all proxies should be dumped for inspection.

quarkus.container-image
A comma separated list of glob expressions to include resources based on src/main/resources path.

resources.includes = src/main/resources/**
quarkus.native.resources.includes = foo/**

quarkus.container-image
A comma separated list of glob expressions to include resources based on src/main/resources path.

resources.includes = src/main/resources/**
quarkus.native.resources.includes = foo/**

Inclusion of resources
By default, no resources are included in native executable.
quarkus.native.resources.includes = src/main/resources

Given
src/main/resources/**
quarkus.native.resources.includes = foo/**

Extra Resources
By default, no resources are excluded.
quarkus.native.resources.excludes = foo/**

Native configuration properties prefixed with quarkus.native:
aditional-build-args
Additional arguments to pass to the build process.

enable-http-url-handler
If the HTTP url handler should be enabled. (default: true)

enable-https-url-handler
If the HTTPS url handler should be enabled.

enable-all-security-services
If all security services should be added to the native image.

user-language
Defines the user language used for building the native executable.

user-country
Defines the user country used for building the native executable.

file-encoding
Defines the file encoding.
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)

Quarkus copies any file under src/main/jib into the built container image.

Prefix is `quarkus.container-image-jib`:

Dockerfile-native-path
Path to the native Dockerfile. (default: $(project.root)/src/main/docker/Dockerfile.native)

S2I
Prefix is `quarkus.container-image-s2i`:

Dockerfile-jvm-path
Path to the JVM Dockerfile. (default: $(project.root)/src/main/docker/Dockerfile.jvm)

Docker

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

Labels and Annotations

The generated manifest use the Kubernetes recommended labels and annotations.

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

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/git/hub
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.ingress.expose=true
quarkus.kubernetes.ingress.host=example.com
quarkus.kubernetes.env.secrets=my-secret
quarkus.kubernetes.secret-volumes.github-token.default-mode=420
```

Kubernetes
Quarkus can use Dekorate to generate Kubernetes resources.

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

Containers
Quarkus copies any file under src/main/jib into the built container image.

Prefix is `quarkus.container-image-jib`:

```
./mvnw quarkus:add-extensions
-Dextensions="quarkus-container-image-jib"
```
You can override the labels by using the next properties:

```java
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:

```java
quarkus.kubernetes.labels.foo-bar
quarkus.kubernetes.annotations.foo-bar
```

```java
metrics
```

```java
When using metrics extension, Prometheus annotations are generated:
```
```java
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 (e.g. `quarkus.kubernetes.readiness-probe.period-seconds` to `quarkus.knative.readiness-probe.period-seconds`).

There are also specific properties for Knative:

```java
quarkus.kubernetes.deployment-target=knative
quarkus.knative.revision-name=my-revision
quarkus.knative.traffic.my-revision.percentage=80
```

List of configuration options:

- kubernetes
- openshift
- knative

You can 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:

```java
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`.

```java
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` API server. (default: `quarkus.kubernetes-config`.)

Data Source Configuration part is absent as it is aut-discovered by Quarkus.

Minikube

Quarkus has a Minikube extension which creates Kubernetes manifests that are tailored for Minikube.

```java
./mvn quarkus:add-extension
-Dextensions="minikube"
```

Remember to execute `eval $(minikube -p minikube docker-env)` to build images directly inside Minikube 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:

```java
./mvn quarkus:add-extension
-Dextensions="openshift"
```

Kubernetes Configuration Extension

Integration between MicroProfile Config spec and ConfigMaps & Secrets:

```java
quarkus.kubernetes-config.enabled=true
quarkus.kubernetes-config.config-maps=cmap1,cmap2
```

If the config key is a Quarkus configuration file `application.properties` and the content is parsed and each key of the configuration file is used as config property.

List of Kubernetes Config parameters.

```java
quarkus.kubernetes-config.enabled as prefix is skipped in the next table.
```

```java
fail-on-missing-config
```

The application will attempt to look up the configuration from the API server. (default: `false`)
The application will not start if any of the configured config sources cannot be located. (default: true)

cfg-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.

List of Kubernetes client parameters.

trust-certs
Trust self-signed certificates. (default: false)

master-url
URL of Kubernetes API server.

namespace
Default namespace.

cacert-file
CA certificate data.

cert-file
Client certificate file.

cert-data
Client certificate data.

key-data
Client key data.

key-algorithm
Client key algorithm.

key-passphrase
Client key passphrase.

username
Username.

password
Password.

watch-reconnect-interval
Watch reconnect interval. (default: PT1S)

watch-reconnect-limit
Maximum reconnect attempts. (default: 5)

connection-timeout
Maximum amount of time to wait for a connection. (default: PT1S)

request-timeout
Maximum amount of time to wait for a request. (default: PT1S)

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 programatically:

```
@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
mockServer.expect()
    .get()
    .withPath("/api/v1/namespaces/test/pods")
    .andReturn(200,
        new PodListBuilder()
            .withNewMetadata()
            .withResourceVersion("1")
            .build()
    ).always();
```

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.


```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()
                    .build()
            ).always();
    }
}
```

AWS Lambda
Quarkus integrates with Amazon Lambda.
To scaffold a deployable microservice to the Azure Functions:

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

## Quarkus - Making a Microservice Deployable to Azure Functions

Quarkus can make a microservice be deployable to Azure Functions.

You can test your implementation by creating a Quarkus project and adding the Azure Functions dependency:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-azure-functions</artifactId>
  <scope>test</scope>
</dependency>
```

You can write tests for Amazon Lambdas:

```
public class TestLambda

implments RequestHandler<MyInput, MyOutput> {
  @Override
  public MyOutput handleRequest(MyInput input, Context context) {
  }
}
```

The `com.amazonaws.services.lambda.runtime.SendResponse` interface 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`

### Test

You can write tests for Amazon Lambdas:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-azure-functions</artifactId>
  <scope>test</scope>
</dependency>
```

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

To scaffold a AWS Lambda run:

```
mvn archetype:generate \\n-DarchetypeGroupid=io.quarkus \\n-DarchetypeArtifactId=quarkus-aws-lambda-archetype \\n-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 \\n-DarchetypeGroupid=io.quarkus \\n-DarchetypeArtifactId=quarkus-azure-functions-http-archetype \\n-DarchetypeVersion={version}
```

## Cloud Events

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.

### Funqy

Funqy is the Quarkus extension that targets Cloud Events (FaaS, KNative, Knative Events) and provides a simple way to run functions on events.

In case of Amazon Lambda, only one Funqy function can be exported per 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 Funqy functions in a pure HTTP environment with simple adding the Funqy HTTP extension.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-funqy-http</artifactId>
</dependency>
```

### Funqy Cloud Events

Add the extension:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-funqy-knative-events</artifactId>
</dependency>
```

```
@CloudEventMapping(trigger = "funqy-knative-events-quickstart"),
@CloudEventMapping(responseType = "annotated")
```

The properties are of form: `quarkus.funqy.knative-events.mapping.<function-name>`.

Also can be overridden with `@io.quarkus.funqy.lambda.Mapping` annotation.

### A K-Native Trigger looks like:

```
apiVersion: serving.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
```

A K-Native Trigger looks like:

```
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:
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="quarkus-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 rst 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.

response-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 requests.

request-curl-options
array of command line options available to the curl command.

request-interceptor
function to intercept remote definition, "Try it out", and OAuth 2.0 requests.

request-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.

presents
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("/developer")
@Operation(summary = "Create developer",
description = "Only be done by admin.")
public Response createDeveloper(  
    @RequestBody(description = "Developer object",  
    required = true,  
    content = {  
        @Content(schema =  
            $ref: "#/components/schemas/Developer"  
        )  
    })  
    Developer developer)
```

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

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

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

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

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.

Possible OpenAPI options with quarkus_swagger-ui.store-schema-directory set:

- path
  the path at which to register the OpenAPI Servlet. (default: openapi)

- store-schema-directory
  The generated OpenAPI schema documents will be stored here on build.

- security-scheme
  add a certain SecurityScheme with config. (basic, jwt, sidr, oauth2Implicit)

- security-scheme-name
  add a SecurityScheme name to the generated OpenAPI document. (default: SecurityScheme)

- security-scheme-description
  add a description to the Security Scheme. (default: Authentication)

- basic-security-scheme-value
  add a scheme value to the Basic HTTP Security Scheme. (default: basic)

- jwt-security-scheme-value
  add a scheme value to the JWT Security Scheme. (default: basic)

- jwt-bearer-format
  add a bearer format to the JWT Security Scheme. (default: bearer)

- oidc-open-id-connect-url
  add a openIdConnectUrl value to the OIDC Security Scheme

- oauth2-implicit-refresh-url
  add an implicit flow refreshUrl value to the OAuth2 Security Scheme

- oauth2-implicit-authorization-url
  add an implicit flow authorizationUrl value to the OAuth2 Security Scheme

- oauth2-implicit-token-url
  add an implicit flow tokenUrl value to the OAuth2 Security Scheme

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(
    Mailer.withText("to@acme.org", "Subject", "Body")
);
```

Reactive Mail client
If you are using quarkus-resteasy-mutiny, you can return @io.smallrye.mutiny.Uni type.

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>miler.from</td>
<td></td>
<td>Default address.</td>
</tr>
<tr>
<td>miler.mock</td>
<td>false in prod, true in dev and test.</td>
<td>Email not sent, just printed and stored in a MockMailbox.</td>
</tr>
<tr>
<td>miler.bounce</td>
<td></td>
<td>Default address.</td>
</tr>
<tr>
<td>miler.host</td>
<td>mandatory</td>
<td>SMTP host.</td>
</tr>
<tr>
<td>miler.port</td>
<td>25</td>
<td>SMTP port.</td>
</tr>
<tr>
<td>miler.username</td>
<td></td>
<td>The username.</td>
</tr>
</tbody>
</table>

### Testing

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

```java
@BeforeEach
MockMailbox mailbox;

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

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

### Scheduled Tasks

You can schedule periodic tasks with Quarkus.

```java
@ApplicationScoped
public class CounterBean {
    @Inject
    private org.quartz.Scheduler quartz;

    @Scheduled(cron = "0 15 10 * *?", delayed = "1s")
    void increment() {}

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

And configure the property into application.properties:

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

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

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

### Kogito

Quarkus integrates with Kogito, a next-generation business automation toolkit from Drools and jBPM projects for adding business automation capabilities.
To start using it you only need to add the next extension:

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

### Apache Tika

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

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tika.tika-config-path</td>
<td>tika-config.xml</td>
<td>Path to the Tika configuration resource. CSV of the abbreviated or full parser class to be loaded by the extension. The document may have other embedded documents. Set if automatically append.</td>
</tr>
<tr>
<td>quarkus.tika.parsers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tika.append-embedded-content</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

### JGit

Quarkus integrates with JGit to integrate with Git repositories:

```java
./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 (e.g., `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.

```java
./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
AtomicAction aa = new AtomicAction();
aa.begin();
    try {
        FlightService.makeBooking("BA123 ...");
        taxiService.makeBooking("East Coast Taxis ...");
        aa.commit();
    } catch (Exception e) {
        aa.abort();
    }
```

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.

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

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

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();
    }
```

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();
    }
```
To configure in clustered mode via DataSource:

```
quarkus.datasource.url=jdbc:postgresql://localhost/quarkus_
```

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

Quartz can be configured using 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.
- `prefix`
  - The prefix for quartz job store tables. (default: `QRTZ_`)
- `trigger-listeners.<name>.class`
  - Class name for the trigger.
- `table-prefix`
  - The prefix for quartz job store tables. (default: `QRTZ_`)
- `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 plugin.

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

```
<html>
<head>
<meta charset="UTF-8"/>
<title>{item.name}</title>
</head>
<body>
<h1>{item.name ?: 'Unknown'}
<h2>{str:reverse('Hello')}</h2>
<div>Price: {item.price}</div>
{#if item.price > 100}
  <div>Discounted Price: {item.discountedPrice}</div>
{/#if}
</body>
</html>
```

To render the template:

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

@Inject
io.quarkus.qute.Template item;

@GET
@Path("/id")
@Produce(MediaType.TEXT_HTML)
public TemplateInstance get(@PathParam("id") Integer id) {
  return item.price.multiply(new BigDecimal("0.9"));
}

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

@TemplateExtension
static String reverse(String val) {
  return new StringBuilder(val).reverse().toString();
}
```

If `@ResourcePath` is not used in `TemplateInstance` then the name of the field is used as file name. In this case the file should be `src/main/resources/templates/item.html`. 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:

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

Methods with multiple parameters are supported too:
The next list methods are supported:

```java
{#list[0]}
```

The next number methods are supported:

```java
{#if counter.mod(5) == 0}
```

Switch/When

```java
{#switch person.name}
   {#case 'John'}
       Hey John!
   {#case 'Mary'}
       Hey Mary!
{/#switch}
```

Switch/When

```java
{#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, &&, &&& , ||, |||, =>, <<=, >, <, <=, <, !=, !in.

Value Resolvers

```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());
}
```

Value Bundling

```java
static BigDecimal discountedPrice(Item item, int scale) {
    return item.price.multiply(scale);
}
```

```java
@io.quarkus.qute.il8n.MessageBundle
public interface AppMessages {
    @io.quarkus.qute.il8n.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

```java
<p>{msg:hello_name('Lucie')}</p>
```

Localization

There are two ways to set localized message:

```java
@io.quarkus.qute.il8n.Localized("de")
public interface GermanAppMessages {
    @Override
    @io.quarkus.qute.il8n.Message("Hallo {name}!")
    String hello_name(String name);
    
    or
    hello_name=Hallo {name}!
    
    You can render programmatically the templates too:
    
    // File located at src/main/resources/templates/reports/v1/report_01.yaml
    #ResourcePath("reports/v1/report_01")
    Template report;
    String output = report
        .data("samples", service.get())
        .render();
```

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.

```bash
./mvnw quarkus:add-extension -Dextensions="quarkus-logging-sentry"
```
And the configuration to send all errors occurring in the package example to Sentry with DSN https://abcd@sentry.io/1234:

```java
quarkus.log.sentry-true
quarkus.log.sentrydsn=https://abcd@sentry.io/1234
quarkus.log.sentrylevel=ERROR
quarkus.log.inapp-packages=org.example
```

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

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

dm
  Where to send events.

dm.level
  Log level (default: WARN)

dm.server-name
  Sets the server name that will be sent with each event.

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

JSch

Quarkus integrates with JSch for SSH communication.

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

Cache

Quarkus can cache method calls by using as key the tuple (method + arguments).

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

Banner

Banner is printed by default. It is not an extension but placed in the core.

```java
@CacheInvalidate
void getDailyForecast(LocalDate date, String city) {}
```

Banner extension

Quarkus integrates with OptaPlanner.

```java
public class TimeTable {
  private SolverJob<TimeTable, UUID> solverJob;
  private UUID problemId = UUID.randomUUID();

  private @Inject SolverManager<TimeTable, UUID> solverManager;
  private TimeTable solution = solverJob.getFinalBestSolution();
```

Impossible configuration options prefixed with quarkus.optaplanner:

```java
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:
  ```<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. (default: 5s)

solver.termination.unimproved-spent-limit
  How long the solver can run without finding a new best solution after finding a new best solution. (default: 2s)

solver-manager.parallel-solver-count
  The number of solvers that run in parallel. (default: PARALLEL_SOLVER_COUNT_AUTO)```
If using netty extension together you already get context propagation for ARC, RESTEasy and transactions. With CompletionStage you need to:

```java
@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 HashiCorp Consul**

You can read runtime configuration from HashiCorp Consul:

```java
./mvnw quarkus:add-extension
-Dextensions="quarkus-smallrye-context-propagation"
```

You need to configure Consul:

```java
quarkus.consul-config.enabled=true
quarkus.consul-config.agent.host-port=localhost:8500
quarkus.consul-config.enabled=false
```

In Consul:

```json
"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

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 content.
- `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:

```java
./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.

```java
./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**

```java
mvn quarkus:add-extension
-Dextensions="amazon-ses"
```

```java
@inject software.amazon.awssdk.services.ses.SesClient sesClient;
```

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

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

or Apache HTTP:

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

```java
quarkus.ses.endpoint-override=http://localhost:8012
quarkus.ses.aws.region=us-east-1
quarkus.ses.aws.credentials.type=static
quarkus.ses.aws.region=us-east-1
quarkus.ses.aws.credentials.static-provider.key-id=test-key
quarkus.ses.aws.credentials.static-provider.secret-access-key=test-secret
```
**Spring DI**

Quarkus provides a compatibility layer for Spring dependency injection.

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

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

```
@Configuration
public class AppConfiguration {

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

Or as a component:

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

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

```
@Service
public class MessageProducer {

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

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

```
@Component
public class GreeterBean {

    private final MessageProducer messageProducer;

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

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

**Spring Boot Configuration**

Quarkus provides a compatibility layer for Spring Boot ConfigurationProperties.

```
./mvn quarkus:add-extension
-Dextensions="quarkus-spring-boot-properties"
```

```
@ConfigurationProperties("example")
public final class ExampleProperties {

    private String value;
    private AnotherClass anotherClass;

    // getters/setters
}
```

You need to configure the extension:

```
quarkus.spring-cloud-config.uri=http://localhost:8089
quarkus.spring-cloud-config.username=user
quarkus.spring-cloud-config.enabled=true
```

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

Prefix is quarkus.spring-cloud-config.

**Spring Cloud Config Client**

Quarkus integrates Spring Cloud Config Client and MicroProfile Config spec.

```
./mvn quarkus:add-extension
-Dextensions="quarkus-spring-cloud-config-client"
```

```
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
```

```
@ConfigurationProperties("example")
public final class ExampleProperties {

    private String value;
    private AnotherClass anotherClass;

    // getters/setters
}
```

```
quarkus.spring-cloud-config.enabled=true
```

**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:

```
@Path("/hello")
@ApplicationScoped
public class Quarkusapp {

    @GET
    public String sayHello() {
        return "hello";
    }
}
```

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
```

**Configuration properties are the same as Amazon DynamoDB but changing the prefix from dynamodb to ses.**

Spring Boot Conguration

Quarkus provides a compatibility layer for Spring Boot ConfigurationProperties.

```
./mvn quarkus:add-extension
-Dextensions="quarkus-spring-boot-properties"
```

```
@ConfigurationProperties("example")
public final class ExampleProperties {

    private String value;
    private AnotherClass anotherClass;

    // getters/setters
}
```

You need to configure the extension:

```
quarkus.spring-cloud-config.enabled=true
```

```
quarkus.spring-cloud-config.uri=http://localhost:8089
quarkus.spring-cloud-config.username=user
quarkus.spring-cloud-config.password=pass
```

Prefix is quarkus.spring-cloud-config.

**Spring Cloud Config Client**

Quarkus integrates Spring Cloud Config Client and MicroProfile Config spec.

```
./mvn quarkus:add-extension
-Dextensions="quarkus-spring-cloud-config-client"
```

```
quarkus.spring-cloud-config.uri=http://localhost:8089
```

You need to configure the extension:

```
quarkus.spring-cloud-config.enabled=true
```

```
quarkus.spring-cloud-config.username=user
quarkus.spring-cloud-config.password=pass
```

Prefix is quarkus.spring-cloud-config.

**Dependency**

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
```
The next return types are supported:
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.

```java
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:

```java
public interface PersonRepository
    extends JpaRepository<Person, Long>, PersonFragment {
    void makeNameUpperCase(Person person);
}
```

**User defined queries:**

```java
@Query("select m from Movie m where m.rating = :rating")
Iterator<Movie> findMyRating(String rating);
```

What is currently unsupported:

- **Methods** of java.util.concurrent.Future
- 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.

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

```java
import java.util.Optional;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.core.annotation.RepositoryRestResource;
import org.springframework.data.repository.data.rest.core.annotation.RestResource;
import org.springframework.data.rest.core.annotation.RestRepository;

@RepositoryRestResource(exported = false, path = "/my-fruit")
public interface FruitsRepository extends CrudRepository<Fruit, Long> {
    Optional<Fruit> findById(Long id);
    @RestResource(exported = true)
    Optional<Fruit> findMyId(Long id);
    @RestResource(exported = true, iterable = 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.

```bash
./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:

```java
@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**
  ```java
  @PreAuthorize("hasRole('admin')")
  @GetMapping
  public String hello() {
      return "hello";
  }
  ```

- **hasRole or hasRole@roles.USER**
  ```java
  @PreAuthorize("hasRole('admin')")
  @GetMapping
  public String hello() {
      return "hello";
  }
  ```

- **hasAnyRole**
  ```java
  @PreAuthorize("hasAnyRole('user', 'admin')")
  @GetMapping
  public String hello() {
      return "hello";
  }
  ```

- **Permit and Deny All**
  ```java
  @PreAuthorize("permitAll()")
  @GetMapping
  public String hello() {
      return "hello";
  }
  ```

- **Anonymous and Authenticated**
  ```java
  @PreAuthorize("isAnonymous()")
  @GetMapping
  public String hello() {
      return "hello";
  }
  ```

- **Checks if the current logged in user is the same as the username method parameter:**
  ```java
  @PreAuthorize("#person.name == authentication.principal.username")
  @GetMapping
  public void doSomethingElse(Person person) {}
  ```

- **Checks if calling a method if user can access:**
  ```java
  @PreAuthorize("@personChecker.check(#person, authentication.principal.username)")
  @GetMapping
  public void doSomething(Person person) {}  
  ```

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

- **Combining expressions:**
  ```java
  @PreAuthorize("hasAnyRole('user', 'admin') AND #user == principal.username")
  @GetMapping
  public void allowedForUser(String user) {}
  ```

### Spring Cache

Quarkus provides a compatibility layer for Spring dependency injection.

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

```java
@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.

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

```java
@org.springframework.scheduling.annotation.Scheduled(cron="*/5 * * * * ?")
void cronJob() {
    System.out.println("Cron expression hardcoded");
}
```

```java
@Scheduled(fixedRate = 1000)
@Scheduled(cron = "(cron.expres)")
```
Resources

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

Authors:

@alexsotob
Java Champion and Director of DevExp at Red Hat