



## 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 0.18.0**.

## Getting Started

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

```
mvn io.quarkus:quarkus-maven-plugin:0.18.0:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=getting-started \
  -DclassName="org.acme.quickstart.GreetingResource" \
  -Dpath="/hello"
```

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

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

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

## 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
```

**Tip** 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=""
```

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

## Application Lifecycle

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

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

## Adding Configuration Parameters

To add configuration to your application, Quarkus relies on `MicroProfile Config` spec (<https://github.com/eclipse/microprofile-config>).

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

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

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

Properties can be set as:

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

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

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

## Configuration Profiles

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

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

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

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

Default profiles are:

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

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

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

And enable it `quarkus-profile=staging`.

## Custom Loader

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

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

```
package com.acme.config;

public class InMemoryConfig implements ConfigSource {

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

    public InMemoryConfig() {
        // Init properties
    }

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

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

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

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

Then you need to register the ConfigSource as Java service. Create a file /META-INF/services/org.eclipse.microprofile.config.spi.ConfigSource with next content:

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

Custom Converters

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

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

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

@Priority annotation is used to override the default InstantConverter.

Then you need to register the Converter as Java service. Create a file /META-INF/services/org.eclipse.microprofile.config.spi.Converter with next content:

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

Injection

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

```
@ApplicationScoped
public class GreetingService {

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

Scope annotation is mandatory to make the bean discoverable.

```
@Inject
GreetingService greetingService;
```

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

JSON Marshalling/Unmarshalling

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

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

Any POJO is marshaled/unmarshalled automatically.

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

    // getter/setters
}
```

JSON equivalent:

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

In a POST endpoint example:

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

Validator

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

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

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

```
public class Sauce {

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

    // getter/setters
}
```

To validate that an object is valid you need to annotate where is used with @Valid annotation:

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

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

Create Your Custom Constraints

First you need to create the custom annotation:

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

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

}
```

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

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

    @Override
    public boolean isValid(LocalDate value,
                          ConstraintValidatorContext ctx) {
        if ( value == null ) return true;
        LocalDate today = LocalDate.now();
        return ChronoUnit.YEARS.between(today, value) > 0;
    }
}
```

And use it normally:

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

Manual Validation

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

```
@Inject
Validator validator;
```

And use it:

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

Logging

You can configure how Quarkus logs:

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

Prefix is `quarkus.log`.

Property	Default	Description
console.enable	true	Console enabled. logging
console.format	%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c{3.}] (%t) %s%n	Format pattern to use for logging.
console.level	INFO	Minimum log level.
console.color	INFO	Allow color rendering.
file.enable	false	File logging enabled.
file.format	%d{yyyy-MM-dd HH:mm:ss,SSS} %h %N[%i] %-5p [%c{3.}] (%t) %s%n	Format pattern to use for logging.
file.level	ALL	Minimum log level.
file.path	quarkus.log	The path to log file.
category."<category-name>".level	INFO	Minimum category. level
level	INFO	Default level. minimum

Rest Client

Quarkus implements `MicroProfile Rest Client` spec:

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

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

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

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

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

}
```

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

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

And configure the hostname at `application.properties`:

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

Injecting the client:

```
@Inject
@RestClient
WorldClockService worldClockService;
```

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

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

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

Adding headers

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

```
@RegisterForReflection
public class BaggageHeadersFactory
    implements ClientHeadersFactory {
    @Override
    public MultivaluedMap<String, String> update(
        MultivaluedMap<String, String> incomingHeaders,
        MultivaluedMap<String, String> outgoingHeaders) {}
}
```

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

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

Or statically set:

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

### Asynchronous

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

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

## Testing

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

```
@QuarkusTest
public class GreetingResourceTest {

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

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

You can also inject the URL where Quarkus is started:

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

### Quarkus Test Resource

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

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

```
public class MyCustomTestResource
    implements QuarkusTestResourceLifecycleManager {

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

    @Override
    public synchronized void stop() {
    }
}
```

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

And the usage:

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

### Mocking

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

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

**Important** This does not work when using native image testing.

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

### Native Testing

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



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 H2, MariaDB, MSSQL and PostgreSQL drivers are supported.

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

```
@Entity
public class Developer extends PanacheEntity {

    // id field is implicit

    public String name;
}
```

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

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

List of datasource parameters.

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

Parameter	Type
driver	String
url	String
username	String
password	String
min-size	Integer
max-size	Integer
initial-size	Integer

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

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

Parameter	Description	Values[Default]
dialect	Class name of the Not necessary Hibernate to set. ORM dialect.	
dialect.storage-engine	The storage engine when the dialect supports multiple storage engines.	Not necessary to set.
sql-load-script	Name of the file containing the SQL statements to execute when starts.	import.sql

Parameter	Description	Values[Default]
batch-fetch-size	The size of the batches.	-1 disabled.
query.query-plan-cache-max-size	The maximum size of the query plan cache.	
query.default-null-ordering	Default precedence of null values in ORDER BY. none, first, last.	[none], first, last.
database.generation	Database schema generation.	[none], create, drop-and-create, drop, update.
database.generation.halt-on-error	Stop on the first error when applying the schema.	[false], true
database.default-catalog	Default catalog.	
database.default-schema	Default Schema.	
database.charset	Charset.	
jdbc.timezone	Time Zone JDBC driver.	
jdbc.statement-fetch-size	Number of rows fetched at a time.	
jdbc.statement-batch-size	Number of updates sent at a time.	
log.sql	Show SQL logs	[false], true



And annotate the test:

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

## Flyway

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

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

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

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

```
@Inject
Flyway flyway;

flyway.migrate();
```

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

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

List of Flyway parameters.

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

Parameter	Default	Description
<code>flyway.migrate-at-start</code>	<code>false</code>	Flyway migration automatically.
<code>flyway.locations</code>	<code>classpath:db/migration</code>	CSV locations to scan recursively for migrations. Supported prefixes <code>classpath</code> and <code>filesystem</code> .

Parameter	Default	Description
<code>flyway.connect-retries</code>	<code>0</code>	The maximum number of retries when attempting to connect.
<code>flyway.schemas</code>	<code>none</code>	CSV case-sensitive list of schemas managed.
<code>flyway.table</code>	<code>flyway_schema_history</code>	The name of Flyway's schema history table.
<code>flyway.sql-migration-prefix</code>	<code>V</code>	Prefix for versioned SQL migrations.
<code>flyway.repeatable-sql-migration-prefix</code>	<code>R</code>	Prefix for repeatable SQL migrations.

## Hibernate Search

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

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

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

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

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

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

    @OneToMany
    @IndexedEmbedded
    public List<Book> books;

}
```

**Important** It is not mandatory to use Panache.

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

```
public class MyQuarkusAnalysisConfigurer
    implements ElasticsearchAnalysisConfigurer {

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

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

Use Hibernate Search in REST service:

```
public class LibraryResource {

    @Inject
    EntityManager em;

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

**IMPORTANT** If you are importing data without using Hibernate ORM, you need to index data by using `Search.getSearchSession(em).createIndexer().startAndWait()` at startup time.

You need to configure the extension in `application.properties`:

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

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

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

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

Parameter	Description
index-defaults.lifecycle.required-status-wait-timeout	Waiting time before failing the bootstrap.
index-defaults.refresh-after-write	Set if index should be refreshed after writes.
Possible annotations:	
Parameter	Description
@Indexed	Register entity as full text index
@FullTextField	Full text search. Need to set an analyzer to split tokens.
@KeywordField	The string is kept as one single token but can be normalized.
IndexedEmbedded	Include the Book fields into the Author index.
@ContainerExtraction	Sets how to extract a value from container, e.g from a 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.



# Reactive Programming

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

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

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

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

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

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

# Reactive Messaging

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

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

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

Produce every 5 seconds one piece of data.

```
@ApplicationScoped
public class ProducerData {

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

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

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

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

You can also inject an stream as a field:

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

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

## Patterns

### REST API → Message

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

emitter.send(message);
```

### Message → Message

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

### Message → SSE

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

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

### Message → Business Logic

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

Possible implementations are:

### In-Memory

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

### Kafka

To integrate with **Kafka** you need to add next extensions:

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

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

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

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

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

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

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

*JSON-B Serializer/Deserializer*

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

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

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

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

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

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

}
```

**AMQP**

To integrate with AMQP you need to add next extensions:

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

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

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

The connector type is `smallrye-amqp`.

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

A complete list of supported properties for AMQP.

**MQTT**

To integrate with MQTT you need to add next extensions:

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

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

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

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

The connector type is `smallrye-mqtt`.

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

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

**Kafka Streams**

Create streaming queries with the Kafka Streams API.

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

You need to initialize `org.apache.kafka.streams.KafkaStreams` on startup:

```
private KafkaStreams streams;

void onStart(@Observes StartupEvent ev) {
    Properties props = new Properties();
    // ...
    StreamsBuilder builder = new StreamsBuilder();
    // ...
    streams = new KafkaStreams(builder.build(), props);

    executor = Executors.newSingleThreadExecutor();
    executor.execute(() -> {
        waitForTopicsToBeCreated(bootstrapServers);
        streams.start();
    });
}

void onStop(@Observes ShutdownEvent ev) {
    streams.close();
    executor.shutdown();
}
```

And then you can use it.

```
public List getData() {
    return streams.allMetadataForStore("")
        .stream()
        .map()
        .collect()
}
```

**Reactive PostgreSQL Client**

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

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

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

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

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

```
@Inject
PgPool client;

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

## JWT

Quarkus implements MicroProfile JWT RBAC spec.

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

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

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

```
@Inject
JsonWebToken jwt;

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

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

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

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

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

Configuration options:

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

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

Supported public key formats:

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

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

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

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

## RBAC

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

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

## Keycloak

Quarkus can use Keycloak to protect resources using bearer token issued by Keycloak server.

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

You can get token information by injecting `KeycloakSecurityContext` object.

```
@Inject
KeycloakSecurityContext keycloakSecurityContext;
```

You can also protect resources with security annotations.

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

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

```
quarkus.keycloak.realm=quarkus
quarkus.keycloak.auth-server-url=http://localhost:8180/auth
quarkus.keycloak.resource=backend-service
quarkus.keycloak.bearer-only=true
quarkus.keycloak.credentials.secret=secret
quarkus.keycloak.policy-enforcer.enable=true
quarkus.keycloak.policy-enforcer.enforcement-mode=PERMISSIVE
```

You can see all possible Configuration parameters [here](#).

**Tip** you can also use `src/main/resources/keycloak.json` standard Keycloak configuration file.

## JAX-RS

Quarkus uses JAX-RS to define REST-ful web APIs.

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

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

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

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

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

To get information from request:

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

Property	Description	Example
@HeaderParam	Gets header parameter by name.	

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

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

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

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

### Injecting

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

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

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

### HTTP Filters

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



```
@Provider
public class LoggingFilter
    implements ContainerRequestFilter {

    @Context
    UriInfo info;

    @Context
    HttpServletRequest request;

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

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

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

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

## CORS Filter

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

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

Prefix is `quarkus.http`.

Property	Default	Description
----------	---------	-------------

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

## Fault Tolerance

Quarkus uses MicroProfile Fault Tolerance spec:

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

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

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

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

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

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

public String fallbackMethod() {
    return "It could beworse";
}
```

`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<String> {
    @Override
    public String 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 ( $4 \times 0.75$ ) 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 **bulkahead** pattern to limit the number of concurrent access to the same resource. If the operation is synchronous it uses a semaphore approach, if it is asynchronous a thread-pool one. When a request cannot be processed `BulkheadException` is thrown. It can be used together with any other fault tolerance annotation.

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

Fault tolerance annotations:

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

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

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

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

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

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

## Observability

### Health Checks

Quarkus relies on MicroProfile Health spec to provide health checks.

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

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

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

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

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

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

Builds the next output:

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

Since health checks are CDI beans, you can do:

```
@ApplicationScoped
public class DatabaseHealthCheck {

    @Produces
    @ApplicationScoped
    @Liveness
    HealthCheck check1() {
        return () -> HealthCheckResponse
            .named("successful-live").up()
            .build();
    }

    @Produces
    @ApplicationScoped
    @Readiness
    HealthCheck check2() {
        return () -> HealthCheckResponse
            .named("successful-read").up()
            .build();
    }
}
```

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

### Metrics

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

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

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

MicroProfile Metrics annotations:

Annotation	Description
@Timed	Method, constructor, or class as timed.

Annotation	Description
@Metered	Method, constructor, or class as metered.
@Counted	Method, constructor, or class as counted.
@Gauge	Method or field as a gauge.
@Metric	Requesting that a metric be injected or registered.

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

@Gauge annotation returning a measure as a gauge.

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

Injecting a histogram using @Metric.

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

Tracing

Quarkus can utilize the MicroProfile OpenTracing spec.

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

Requests sent to any endpoint are traced automatically.

This extension includes OpenTracing support and Jaeger tracer.

Jaeger tracer configuration:

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

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

Tracer class can be injected into the class.

```
@Inject
Tracer tracer;

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

Cloud

Native

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

```
./mvnw package -Pnative -Dnative-image.docker-build=true

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

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

Kubernetes

Quarks can use ap4k to generate Kubernetes resources.

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

Running ./mvnw package the Kubernetes resources are created at target/wiring-classes/META-INF/kubernetes/ directory.

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

Generated resource is integrated with MicroProfile Health annotations.

Kubernetes Client

Quarkus integrates with Fabric8 Kubernetes Client.

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

Currently you need to create a Kubernates Client **Important** producer by yourself, but this is gonna be improved in near future versions:

```
@Dependent
public class KubernetesClientProducer {

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

And inject it on code:

```
@Inject
KubernetesClient client;

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

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

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

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

## Amazon Lambda

Quarkus integrates with Amazon Lambda.

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

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

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

### Test

You can write tests for Amazon lambdas:

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

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

## Apache Camel

Quarkus integrates wih Apache Camel.

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

You can inject `CamelRuntime` and `CamelContext` instances:

```
@Inject
CamelRuntime runtime;

@Inject
CamelContext context;
```

And also observing Camel events `InitializingEvent`, `InitializedEvent`, `StartingEvent`, `StartedEvent`, `StoppingEvent`, `StoppedEvent`.

```
public void init(@Observes InitializingEvent event) {
    runtime.getContext()
        .setUuidGenerator(new DefaultUuidGenerator());
}
```

You can create `RouteBuilder` for routing rules:

```
public class CamelRoute extends RouteBuilder {

    @Override
    public void configure() {
        from("timer:keep-alive?" +
            "period={{camel.timer-route.period}}")
            .id("timer-route")
            .setBody(constant(""))
            .to("log:keep-alive");
    }
}
```

Custom properties are set in `application.properties`.

```
camel.timer-route.period=1s
```

List of Camel parameters.

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

Parameter	Default	Description
<code>camel.routesUri</code>		Uri to an XML with camel routes.



Parameter	Default	Description
<code>camel.deferInitPhase</code>	<code>false</code>	Defer context initialization phase until runtime.
<code>camel.disableJaxb</code>	<code>false</code>	Disable support at runtime. Useful when routes loaded at build time.
<code>camel.disableXml</code>	<code>false</code>	Disable XML support in various parts of Camel.
<code>camel.dumpRoutes</code>	<code>false</code>	Dump loaded routes.

Quarkus also comes with support for Camel in form of next extensions: `camel-aws-s3`, `camel-aws-sqs`, `camel-infinispan`, `camel-netty4-http`, `camel-salesforce`.

## WebSockets

Quarkus can be used to handling web sockets.

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

And web sockets classes can be used:

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

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

    @OnClose
    public void onClose(..) {}

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

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

}
```

## OpenAPI

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

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

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

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

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

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

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

## Mail Sender

You can send emails by using Quarkus Mailer extension:

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

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

```
@Inject
Mailer mailer;

@Inject
ReactiveMailer reactiveMailer;
```

And then you can use them to send an email:

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

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

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

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

**Tip** If you need deep control you can inject Vert.x mail client `@Inject MailClient client;`

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

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

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

Parameter	Default	Description
<code>mailer.from</code>		Default address.
<code>mailer.mock</code>	<code>false</code> in prod, <code>true</code> in dev and test.	Emails not sent, just printed and stored in a <code>MockMailbox</code> bean.
<code>mailer.bounce-address</code>		Default address.
<code>mailer.host</code>	<i>mandatory</i>	SMTP host.
<code>mailer.port</code>	25	SMTP port.
<code>mailer.username</code>		The username.
<code>mailer.password</code>		The password.
<code>mailer.ssl</code>	<code>false</code>	Enables SSL.
<code>mailer.trust-all</code>	<code>false</code>	Trust all certificates.

Parameter	Default	Description
mailer.max-pool-size	10	Maximum number of open connections .
mailer.own-host-name		Hostname HELO/EHLO Message-ID for and
mailer.keep-alive	true	Connection pool enabled.
mailer.disable-esmtp	false	Disable ESMTP.
mailer.start-tls	OPTIONAL	TLS security mode. NONE, OPTIONAL, REQUIRED.
mailer.login	NONE	Login mode. NONE, OPTIONAL, REQUIRED.
mailer.auth-methods	All methods.	Space-separated list.
mailer.key-store		Path of the key store.
mailer.key-store-password		Key store password.

Testing

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

```
@Inject
MockMailbox mailbox;

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

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

Scheduled Tasks

You can schedule periodic tasks with Quarkus.

```
@ApplicationScoped
public class CounterBean {

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

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

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

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

And configure the property into `application.properties`:

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

Kogito

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


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

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

Resources

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

Authors :



**@alexsotob**  
Java Champion and SW Engineer at Red Hat

0.18.0

