Quarkus Cheat-Sheet

What is Quarkus?

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

Cheat-sheet tested with **Quarkus 0.15.0**.

Getting Started

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

```
mvn io.quarkus:quarkus-maven-plugin:0.15.0:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started \
    -DclassName="org.acme.quickstart.GreetingResource" \
    -Dpath="/hello"
```

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

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

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

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
```

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.
- System properties.
- Resources `src/main/resources/application.properties`.
- External config directory under the current working directory: `config/application.properties`.

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

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 (https://quarkus.io/guides/cdi-reference).

```
@ApplicationScoped
public class GreetingService {

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

Scope annotation is mandatory to make the bean discoverable by CDI.

```
@Inject
GreetingService greetingService;
```

Important Quarkus is designed with Substrate VM in mind. For 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) {}
```

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

Create Your Custom Constraints

First you need to create the custom annotation:

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

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

}
```

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

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

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

And use it normally:

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

Manual Validation

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

```
@Inject
Validator validator;
```

And use it:

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

Logging

You can configure how Quarkus logs:

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

Prefix is `quarkus.log`.

Property	Default	Description
`console.enable`	true	Console logging enabled.
`console.format`	`%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c{3.}] (%t) %s%e%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%e%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 level category.
`level`	INFO	Default minimum level.

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 invokation happens within a JAX-RS resource class, you can propagate headers from incoming request to the outgoing request by using next configuartion property.

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

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

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.

To package and run the application for testing:

```
@QuarkusTest
public class GreetingResourceTest {

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

}
```

Test port can be modified by using `quarkus.http.test-port` configuration property.

You can also inject the URL where Quarkus is started:

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

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.

To test native executables you can 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
```

Database operations:

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

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

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

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

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

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

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

- `order by` which expands to `from EntityName order by …`

- `<columnName>` which expands to `from EntityName where <columnName>=?`

- `<query>` which is expanded to `from EntityName where <query>`

Static Methods

Field	Parameters	Return
findById	Object	Returns object or null if not found.
find	String, [Object…, Map<String, Object>, Parameters]	Lists of entities meeting given query with parameters set.
find	String, Sort, [Object…, Map<String, Object>, Parameters]	Lists of entities meeting given query with parameters set sorted by `Sort` attribute/s.
findAll		Finds all entities.
findAll	Sort	Finds all entities sorted by `Sort` attribute/s.
stream	String, [Object…, Map<String, Object>, Parameters]	java.util.stream.Stream of entities meeting given query with parameters set.
stream	String, Sort, [Object…, Map<String, Object>, Parameters]	java.util.stream.Stream of entities meeting given query with parameters set sorted by `Sort` attribute/s.
streamAll		java.util.stream.Stream of all entities.
streamAll	Sort	java.util.stream.Stream of all entities sorted by `Sort` attribute/s.
count		`Number of entities.
count	String, [Object…, Map<String, Object>, Parameters]	Number of entities meeting given query with parameters set.
deleteAll		Number of deleted entities.
delete	String, [Object…, Map<String, Object>, Parameters]	Number of deleted entities meeting given query with parameters set.
persist	[Iterable, Steram, Object…]	

Tip Panache also supports *DAO* pattern by providing `PanacheRepository<TYPE>` interface to be implemented by your *repository* class.

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
`flyway.migrate-at-start`	`false`	Flyway migration automatically.
`flyway.locations`	`classpath:db/migration`	CSV locations to scan recursively for migrations. Supported prefixes `classpath` and `filesystem`.
`flyway.connect-retries`	`0`	The maximum number of retries when attempting to connect.
`flyway.schemas`	`none`	CSV case-sensitive list of schemas managed.
`flyway.table`	`flyway_schema_history`	The name of Flyway's schema history table.
`flyway.sql-migration-prefix`	`V`	Prefix for versioned SQL migrations.
`flyway.repeatable-sql-migration-prefix`	`R`	Prefix for repeatable SQL migrations.

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 (https://github.com/eclipse/microprofile-reactive-messaging) 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));
    }
}
```

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> randomRumbers;
```

Possible return types:

Kafka

To integrate with Kafka you need to add next extensions:

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

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

Kafka configuration schema: `smallrye.messaging.[sink|source].{stream-name}.<property>=<value>`

```
smallrye.messaging.source.prices.type=
    io.smallrye.reactive.messaging.kafka.Kafka
smallrye.messaging.source.prices.topic=
    prices
smallrye.messaging.source.prices.bootstrap.servers=
    localhost:9092
smallrye.messaging.source.prices.key.deserializer=
    org.apache.kafka.common.serialization.StringDeserializer
smallrye.messaging.source.prices.value.deserializer=
    org.apache.kafka.common.serialization.IntegerDeserializer
smallrye.messaging.source.prices.group.id=
    my-group-id
```

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

> **Important** If the stream is not configured then it is assumed to be an in-memory stream.

Reactive PostgreSQL Client

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

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

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 extension 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.

Fault Tolerance

Quarkus uses https://github.com/eclipse/microprofile-fault-tolerance[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) {
        return "It could be worse";
    }
}
```

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

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

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

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

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

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

Fault tolerance annotations:

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

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

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

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

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

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

Observability

Health Checks

Quarkus relies on **MicroProfile Health** spec to provide health checks.

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

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

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

To create a custom health check you need to implement the `HealthCheck` interface and annotate it with `@Health` annotation.

```java
@Health
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:

```
{
    "outcome": "UP",
    "checks": [
        {
            "name": "Database connection",
            "state": "UP",
            "data": {
                "connection": true
            }
        }
    ]
}
```

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.
@Metered	Method, constructor, or class as metered.

Annotation	Description
@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 to provide tracing support.

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

By default, requests sent to any endpoint are traced without any code changes being required.

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

Amazon Lambda

Quarkus integrates with Amazon Lambda.

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

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

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

        // ...
    }
}
```

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 initializing(@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
camel.routesUris		Uri to an MXL with camel routes to load and initialize at build time.
camel.deferInitPhase	false	Defer context initialization phase until runtime.
camel.disableJaxb	false	Disable jaxb support at runtime. Useful when routes loaded at build time.
camel.disableXml	false	Disable XML support in various parts of Camel.
camel.dumpRoutes	false	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.

Also, in case of starting Quarkus application in `dev` or `test` mode, Swagger UI is accessible at `/swagger-ui`.

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

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

Resources

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

Authors :

@alexsotob
Java Champion and SW Engineer at Red Hat

0.15.0