Quarkus

What is Quarkus?

Quarkus (https://quarkus.io/) 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.

Getting Started

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

```
mvn io.quarkus:quarkus-maven-plugin:0.14.0: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.

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 variable, system property or in `src/main/resources/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();
}
```

Persistence

Quarkus works with JPA(Hibernate) as persistence solution. But also provides an Active Record pattern (https://en.wikipedia.org/wiki/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` 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.

Rest Client

Quarkus implements MicroProfile Rest Client (https://github.com/eclipse/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 worldClockOption

}
```

```
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=
```

Injecting the client:

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

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> clientOutgoingHeaders) {}
}
```

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();
```

Validator

Quarkus uses Hibernate Validator (https://hibernate.org/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 context
        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);
```

Reactive Programming

Quarkus implements MicroProfile Reactive spec (https://github.com/eclipse/microprofile-reactive-streams-operators) 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));
}
```

Observability

Health Checks

Quarkus relies on MicroProfile Health spec (https://github.com/eclipse/microprofile-health) 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.

```
@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 (https://github.com/eclipse/microprofile-metrics) to provide metrics support.

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

The metrics can be read remotely using JSON format or the OpenMetrics format. By just adding the extension, an endpoint is registered at `/metrics` providing default metrics.

MicroProfile Metrics comes with some annotations to add specific monitoring values:

Annotation	Description
`@Timed`	Marking a method, constructor, or class as timed.
`@Metered`	Marking a method, constructor, or class as metered.
`@Counted`	Marking a method, constructor, or class as counted.
`@Gauge`	Marking a 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 (https://github.com/eclipse/microprofile-opentracing) 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
```

Kubernetes

Quarks can use ap4k (https://github.com/ap4k/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

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.

Testing

When you generate the Quarkus project with the archetype, test dependencies with JUnit 5 are registered automatically, but also the 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`.

Authors :

@alexsotob
Java Champion and SW Engineer at Red Hat

v0.14.0