Quarkus First Steps

Table of Contents

Introduction

This tutorial builds two Java services from scratch that incrementally adds functionality. The first is a *Greeting Service* that returns a greeting. The second is a *Training Service* that persists Training courses and students. The *Training Service* invokes the *Greeting Service* for a message to greet students. This domain model is simple, so the focus is spent on the tutorial APIs and concepts.

The documented steps in this tutorial follow a pattern:

1. Write code
2. Test the code (`curl` or `mvn test`)
3. Check output

This pattern flows quickly due to Quarkus' Live Coding feature and makes checking code changes near-instantaneous, so each change is immediately evaluated.

NOTE This tutorial will require three terminal windows. Each command block will display the terminal where a command runs - *Terminal 1*, *Terminal 2*, or *Terminal 3*. It helps to organize the terminals, so they are all visible. In addition, each code block will show the name of the file to be edited (ex: "StudentResource.java")

Setup Local Repository

1. Clone the project to your local system

Terminal 1

```
$ git clone \
  https://github.com/jclingan/oreilly-quarkus-firststeps \
  firststeps
```

Terminal 1 Output

```
Cloning into 'firststeps'...
remote: Enumerating objects: 467, done.
remote: Counting objects: 100% (467/467), done.
remote: Compressing objects: 100% (292/292), done.
remote: Total 467 (delta 224), reused 359 (delta 123), pack-reused 0
Receiving objects: 100% (467/467), 2.55 MiB | 2.36 MiB/s, done.
Resolving deltas: 100% (224/224), done.
```

Create Greeting Service

Create a simple Quarkus project as a starting point - the *Greeting Service*. A default REST endpoint is generated. While this tutorial utilizes the mvn command line tooling, projects can also be generated at code.quarkus.io using a Web UI.

1. Go to the project’s base directory and create project using maven

   Terminal 1

   ```
   cd firststeps/working

   mvn io.quarkus:quarkus-maven-plugin:1.10.2.Final:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=greeting \
       -DclassName="org.acme.GreetingResource" \
       -Dpath="/greeting" \
       -Dextensions=resteasy-jsonb
   ```

2. Optional: To see a list of available extensions, type:

   Terminal 1

   ```
   mvn quarkus:list-extensions
   ```

3. Open project in your favorite IDE. Open GreetingResource.java and peruse the JAX-RS source code

GreetingResource.java

```java
package org.acme;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/greeting")
public class GreetingResource {

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

NOTE Quarkus does not require a JAX-RS Application class, but will use it if provided. In addition, Quarkus RESTful resources are singletons (`@javax.inject.Singleton`). These are developer convenience features. These points are covered in the Quarkus Getting Started Guide.

4. Start Quarkus in developer mode

   Terminal 1

   ```
   $ mvn compile quarkus:dev
   ```

5. Check that the endpoint returns *Hello RESTEasy*

   Terminal 2

   ```
   curl -i localhost:8080/greeting
   ```

   Terminal 2 Output

   ```
   HTTP/1.1 200 OK
   Content-Length: 5
   Content-Type: text/plain;charset=UTF-8

   Hello RESTEasy
   ```

6. Try out live reload. In the hello() method, replace *Hello* with *Howdy* and save the file

   GreetingResource.java

   ```
   public class GreetingResource {

       @GET
       @Produces(MediaType.TEXT_PLAIN)
       public String hello() {
           return "Howdy RESTEasy"; ①
       }
   }
   ```

   ① Replace *Hello* with *Howdy*

   Terminal 2

   ```
   curl -i localhost:8080/greeting
   ```

   Terminal 2 Output

   ```
   HTTP/1.1 200 OK
   Content-Length: 5
   Content-Type: text/plain;charset=UTF-8

   Howdy RESTEasy
   ```

7. Change *Howdy* back to *Hello*

8. In GreetingResource.java, create a list of Strings:

   GreetingResource.java

   ```
   @Path("/greeting")
   public class GreetingResource {
       List<String> greetings = new ArrayList<>(); ①
   ```

   ① Add this line

9. Add a method called `listGreetings()` at the "/list" path that returns the greetings as a JSON array

GreetingResource.java

```
@GET
@Path("/list")
@Produces(MediaType.APPLICATION_JSON)
public List<String> listGreetings() {
    return greetings;
}
```

10. Check the output

    Terminal 2

    ```
    curl -i localhost:8080/greeting/list
    ```

    Terminal 2 Output

    ```
    HTTP/1.1 200 OK
    Content-Length: 2
    Content-Type: application/json

    []
    ```

Quarkus Configuration

This section covers externalizing configuration using MicroProfile. In these instructions, configuration parameters are stored in `application.properties` instead of `microprofile-config.properties`.

NOTE While Quarkus supports MicroProfile APIs, it also supports much more than MicroProfile like Spring APIs and Vert.x APIs. For that reason, the Quarkus guides refer to the more framework-agnostic `src/main/resources/application.properties`.

Change the hard coded value to return a random value.

1. Create greetings property in `application.properties`

   application.properties

   ```
   greetings=Howdy from app.prop,Hola from app.prop,Hello from app.prop
   ```

2. Use MicroProfile Config to inject greetings

   GreetingResource.java

   ```
   @ConfigProperty(name="greetings")          ①
   List<String> greetings = new ArrayList<>();
   ```

   ① Inject the list of greetings from application.proprties into the greetings list

3. Test the `/list` endpoint

   Terminal 2

   ```
   curl -i http://localhost:8080/greeting/list
   ```

   Terminal 2 Output

   ```
   HTTP/1.1 200 OK
   Content-Length: 66
   Content-Type: application/json

   ["Howdy from app.prop","Hola from app.prop","Hello from app.prop"]
   ```

4. Next, let’s get a random greeting from the list. The primary intent is to create a second method that will be used in future sections.

GreetingResource.java

```java
@GET
@Path("/list/random")
@Produces(MediaType.TEXT_PLAIN)
public String randomGreeting() {
    Random r = new Random();
    return greetings.get(r.nextInt(listGreetings().size())); ①
}
```

① Return a random greeting

5. Test the `/list/random` endpoint. Run multiple times to see random output.

   Terminal 2

   ```
   curl -i http://localhost:8080/greeting/list/random
   ```

   Terminal 2 Output

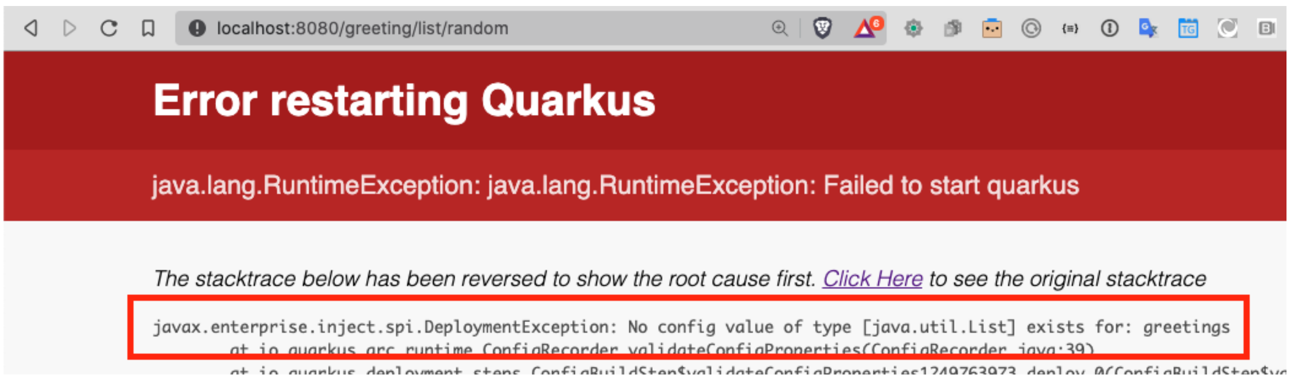
   ```
   HTTP/1.1 200 OK
   Content-Length: 19
   Content-Type: text/plain;charset=UTF-8

   Hello from app.prop
   ```

6. Comment out the greeting property in `application.properties`

   application.propertiese

   ```
   #greetings=Howdy from app.prop,Hola from app.prop,Hello from app.prop
   ```

   Load URL in browser to better see how Quarkus reverses the stack trace to put the primary error at the top of the browsr page:



localhost:8080/greeting/list/random

Error restarting Quarkus

java.lang.RuntimeException: java.lang.RuntimeException: Failed to start quarkus

The stacktrace below has been reversed to show the root cause first. Click Here to see the original stacktrace

```
javax.enterprise.inject.spi.DeploymentException: No config value of type [java.util.List] exists for: greetings
        at io.quarkus.arc.runtime.ConfigRecorder.validateConfigProperties(ConfigRecorder.java:39)
        at io.quarkus.deployment.steps.ConfigBuildStep$validateConfigProperties1249763973.deploy_0(ConfigBuildStep$v
```

7. Uncomment the `greeting` property in `application.properties` that was commented out in the prior step and refresh the browser

---

8. Override the greetings value in the property file with a `GREETINGS` environment variable. Press CTRL-C, set the `GREETINGS` environment variable, and restart the application.

   Terminal 1

   ```
   CTRL-C ①
   export GREETINGS="Howdy from env,Hola from env,Hello from env" ②
   mvn quarkus:dev ③
   ```

   ① Stop the application run with `mvn quarkus:dev`

   ② Override the `greeting` property with an environment variable

   ③ Restart the application in developer mode

   Terminal 2 (Test the application)

   ```
   curl -i http://localhost:8080/greeting/list
   ```

   Terminal 2 Output

   ```
   HTTP/1.1 200 OK
   Content-Length: 51
   Content-Type: application/json

   ["Howdy from env","Hola from env","Hello from env"]
   ```

---

9. Override the `GREETINGS` environment value in the property file with a system property

   Terminal 1

   ```
   CTRL-C  ①
   mvn quarkus:dev -Dgreetings="Howdy from system,Hola from system,Hello from system"
   ②
   ```

   ① Stop application

   ② Start application with system property defined

---

10. Remove the environment variable and restart the application

Terminal 1

```
CTRL-C
unset GREETINGS ①
mvn quarkus:dev ②
```

① Remove the `GREETINGS` environment variable

② Restart the application **without** the greetings system property defined

---

11. Read a configuraton into a configuration property objects. This is useful for consolidating related configuration propertise into a single class.

UnusedConfigProperties.java

```
@ConfigProperties(prefix = "unused")                    ①
public class UnusedConfigProperties {
    int number;                                         ②

    String string="Unused string";                      ②

    Optional<Boolean> flag;                             ③

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

    public boolean getFlag() {
        return flag.isEmpty() ? false : flag.get();
    }

    public void setFlag(Optional<Boolean> flag) {
        this.flag = flag;
    }
}
```

① `@ConfigurationProperties` object will auto-inject property values in fields. The `prefix` specifies that the properties are prefixed with `unused.`.

② Field injection is supported. The property names will be `unused.number` and `unused.string`. If no property value is defined, and no default value is supplieed, a DeploymentException will be thrown. These two fields avoid a DeploymentException when no property value is defined by providing a default field values.

③ Optional fields are supported.

NOTE <a href="https://download.eclipse.org/microprofile/microprofile-config-2.0-RC1/microprofile-config-spec.html">MicroProfile Config 2.0</a>, planning a release in Q4 2020 as a part of MicroProfile 4.0, <a href="https://download.eclipse.org/microprofile/microprofile-config-2.0-RC1/microprofile-config-spec.html#<em>aggregate_related_properties_into_a_cdi_bean">will formally define ConfigProperties</a> where class member fields can be be annotated with <code>@ConfigProperty</code>. Quarkus plans to support _MicroProfile 4.0</em> and <em>MicroProfile Config 2.0</em>.

---

12. Update GreetingResource.java with an endpont to return the values of UnusedConfigProperties.

GreetingResource.java

```
UnusedConfigProperties unused;

public GreetingResource(UnusedConfigProperties unused) {  ①
    this.unused = unused;
}

@GET
@Path("/unused")
@Produces(MediaType.APPLICATION_JSON)
public UnusedConfigProperties getProps() {                ②
    return unused;
}
```

① Inject `UnusedConfigProperties` instance into `unused` field. This approach uses constructor injection. Field injection using `@Inject` is also supported.

② A simple endpont that returns `unused` in JSON format.

---

13. Update `application.properties` with `unused.*` properties

application.properties

```
# Demonstrate @ConfigurationProperties feature, but are not used in application

unused.flag=true
unused.number=10
unused.string=Unused string
```

14. Test `UnusedProperties` using the REST endpoint

    Terminal 2

    ```
    curl -i http://localhost:8080/greeting/unused
    ```

    Terminal 2 output

    ```
    HTTP/1.1 200 OK
    Content-Length: 50
    Content-Type: application/json

    {"flag":false,"number":0,"string":"Unused string"}
    ```

15. Test for updated `unused` property values

    Terminal 2

    ```
    curl -i http://localhost:8080/greeting/unused
    ```

    Terminal 2 output

    ```
    HTTP/1.1 200 OK
    Content-Length: 50
    Content-Type: application/json

    {"flag":true,"number":10,"string":"Unused string"}
    ```

Quarkus Logging

This section covers Quarkus logging. The logging implementation is JBoss Logging, and is configured using `application.properties`.

1. Simplify the logging output to better demonstrate logging. Log formatting symbols are described here.

   Add to application.properties

   ```
   # Simplify console output

   quarkus.log.console.format=%-5p [%c{3.}] %s%e%n   ①

   # Default formatting
   # %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c{3.}] (%t) %s%e%n
   ```

   ① Display Priority (`%-5p`), category (`[%c{3.}]`), and simple message/exception/newline (`%s%e%n`). See Logging Format.

---

2. Create a logging class with a `loggingLevels()` method and REST endpoint to print out logging levels

   LoggingResource.java

   ```java
   @Path("/logging")
   public class LoggingResource {
       private static final
       Logger LOG = Logger.getLogger(LoggingResource.class);                  ①


       @GET
       @Produces(MediaType.TEXT_PLAIN)
       public String getLogging() {                                           ②
           logginglevels();
           return "Check log output";
       }

       void logginglevels() {
         System.out.println("----------------------------------------");
         LOG.fatal("***** " + LOG.getClass().getSimpleName() + ": FATAL *****");③
         LOG.error("***** " + LOG.getClass().getSimpleName() + ": ERROR *****");
         LOG.warn("***** " + LOG.getClass().getSimpleName() + ": WARN *****");
         LOG.info("***** " + LOG.getClass().getSimpleName() + ": INFO *****");
         LOG.debug("***** " + LOG.getClass().getSimpleName() + ": DEBUG *****");
         LOG.trace("***** " + LOG.getClass().getSimpleName() + ": TRACE *****");
       }
   }
   ```

① Get a logger

② Create REST endpoint to print out levels

③ Print output at each logging levels

3. Change logging level in `application.properties`

   application.properties

   ```
   quarkus.log.level=TRACE   ①
   ```

   ① Print all messages.

4. Test TRACE log level

   Terminal 2

   ```
   curl -i http://localhost:8080/logging
   ```

   Terminal 2 output

   ```
   ...

   DEBUG [io.qua.res.runtime] Create resource constructor: public
   org.acme.GreetingResource()
   INFO  [io.quarkus] greeting 1.0-SNAPSHOT on JVM (powered by Quarkus 1.9.0.CR1)
   started in 0.612s. Listening on: http://0.0.0.0:8080
   INFO  [io.quarkus] Profile dev activated. Live Coding activated.
   INFO  [io.quarkus] Installed features: [cdi, resteasy, resteasy-jsonb]
   INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] Hot replace total time: 0.621s
   DEBUG [org.jbo.res.res.i18n] RESTEASY002315: PathInfo: /logging
   ----------------------------------------
   FATAL [org.acm.GreetingResource] ***** JBossLogManagerLogger: FATAL *****
   ERROR [org.acm.GreetingResource] ***** JBossLogManagerLogger: ERROR *****
   WARN  [org.acm.GreetingResource] ***** JBossLogManagerLogger: WARN *****
   INFO  [org.acm.GreetingResource] ***** JBossLogManagerLogger: INFO *****
   DEBUG [org.acm.GreetingResource] ***** JBossLogManagerLogger: DEBUG *****
   TRACE [org.acm.GreetingResource] ***** JBossLogManagerLogger: TRACE *****
   DEBUG [org.jbo.res.res.i18n] MessageBodyWriter:
   org.jboss.resteasy.core.providerfactory.SortedKey
   DEBUG [org.jbo.res.res.i18n] MessageBodyWriter:
   org.jboss.resteasy.plugins.providers.StringTextStar

   ...
   ```

5. Limit console output to INFO

   Add to application.properties

   ```
   quarkus.log.console.level=INFO
   ```

6. Test INFO log level

   Terminal 2

   ```
   curl -i http://localhost:8080/logging
   ```

   Terminal 2 output

   ```
   ----------------------------------------
   FATAL [org.acm.GreetingResource] ***** JBossLogManagerLogger: FATAL *****
   ERROR [org.acm.GreetingResource] ***** JBossLogManagerLogger: ERROR *****
   WARN  [org.acm.GreetingResource] ***** JBossLogManagerLogger: WARN *****
   INFO  [org.acm.GreetingResource] ***** JBossLogManagerLogger: INFO *****
   ```

7. Log all REST log records to a file

   Add to application.properties

   ```
   # Log all REST output to file using a named (REST_TRACE) file handler
   quarkus.log.handler.file."REST_TRACE".enable=true
   quarkus.log.handler.file."REST_TRACE".path=/tmp/rest.log
   quarkus.log.handler.file."REST_TRACE".level=TRACE
   quarkus.log.category."org.jboss.resteasy".handlers=REST_TRACE
   ```

8. Force logging to file

   Terminal 2

   ```
   curl -i http://localhost:8080/logging
   ```

/tmp/rest.log

```
...

DEBUG MessageBodyWriter: org.jboss.resteasy.plugins.providers.StringTextStar
DEBUG MessageBodyWriter: org.jboss.resteasy.plugins.providers.StringTextStar

...
```

9. Update application.properties to configure json logging

   Add to application.properties

   ```
   # JSON configuration settings
   quarkus.log.console.json.pretty-print=true  ①
   ```

   ① Format JSON in a human-readable format

10. Log in JSON output by adding the JSON logging extension

    Terminal 2

    ```
    mvn quarkus:add-extension -Dextensions="logging-json" ①
    ```

    ① Add the quarkus JSON logging extension.

    Terminal 2 output

    ```
    {                                                   ①
    "timestamp": "2020-10-17T22:00:26.63-07:00",
    "sequence": 3168,
    "loggerClassName": "org.jboss.logging.Logger",
    "loggerName": "org.acme.LoggingResource",
    "level": "INFO",
    "message": "***** JBossLogManagerLogger: INFO *****",
    "threadName": "executor-thread-199",
    "threadId": 290,
    "mdc": {
    },
    "ndc": "",
    "hostName": "jclingan-mac",
    "processName": "greeting-dev.jar",
    "processId": 39901
    }

    ...
    ```

① Because the *Greeting Service* is in development mode and the extension enables JSON logging by default, the Live Coding restart will immediately log in JSON format.

11. Disable JSON logging

    Add to application.properties

    ```
    quarkus.log.console.json=false
    ```

12. Start the syslog server

    Terminal 2

    ```
    docker run -d --rm=true -it -p 1514:514/udp \
        --name syslog-ng balabit/syslog-ng:latest    ①
    ```

13. Configure Quarkus to use syslog

    Add to application.properties

    ```
    # Syslog settings

    quarkus.log.syslog.enable=true                ①
    quarkus.log.syslog.endpoint=localhost:1514    ②
    quarkus.log.syslog.protocol=udp               ③
    quarkus.log.syslog.hostname=jclingan-mac      ④
    quarkus.log.syslog.app-name=greeting          ⑤
    quarkus.log.syslog.level=ERROR                ⑥
    ```

    ① More than one logging handler can be specified. There are now three (console, file, syslog)

    ② The host:port of the syslog server

    ③ Log using TCP/IP udp protocol

    ④ Name of the log message originating host. This will be your hostname

    ⑤ The name of the application sending the log message

    ⑥ The syslog log level

14. Log a message and check syslog

Terminal 2

```
curl -i http://localhost:8080/logging          ①
docker exec syslog-ng tail /var/log/messages   ②
```

① Generate a log messsage

② Check the log message on the syslog server

Terminal 2 output

```
...

Oct 18 00:32:16 jclingan-mac greeting[39901]: 2020-10-18 00:32:16,497 FATAL
[org.acm.LoggingResource] (executor-thread-199) ***** JBossLogManagerLogger: FATAL
*****
Oct 18 00:32:16 jclingan-mac greeting[39901]: 2020-10-18 00:32:16,498 ERROR
[org.acm.LoggingResource] (executor-thread-199) ***** JBossLogManagerLogger: ERROR
*****

...
```

15. Start the ELK (Elasticsearch/Logstash/Kibana) stack

Terminal 2

```
docker-compose -f docker/elk.yml up         ①
```

Terminal 2 output

```
Creating network "docker_elk" with driver "bridge"
Creating docker_elasticsearch_1 ... done
Creating docker_logstash_1      ... done
Creating docker_kibana_1        ... done
```

16. Configure GELF (GreyLog Extended Log Format) to use the ELK stack

Add to application.properties

```
# GELF settings
quarkus.log.handler.gelf.enabled=true       ①
quarkus.log.handler.gelf.host=localhost     ②
quarkus.log.handler.gelf.port=12201         ③
```

① Enable centralized logging to Logstash using GELF

② Hostname running Logstash

③ Logstash port

17. Add the GELF extension

    Terminal 3

    ```
    mvn quarkus:add-extension -Dextensions=logging-gelf
    ```

18. View Logstash log output

    Terminal 3

    ```
    curl -i http://localhost:8080/logging
    ```

    Terminal 2 output

    ```
    ...  ①
    logstash_1       |     "SourceSimpleClassName" =>
    "AbstractWriterInterceptorContext",
    logstash_1       |          "SourceMethodName" => "asyncProceed",
    logstash_1       |                    "Thread" => "executor-thread-1",
    logstash_1       |                "LoggerName" =>
    "org.jboss.resteasy.resteasy_jaxrs.i18n",
    logstash_1       |                     "level" => 7,
    logstash_1       |                      "Time" => "2020-10-18 01:44:32,992",
    logstash_1       |           "SourceClassName" =>
    "org.jboss.resteasy.core.interception.jaxrs.AbstractWriterInterceptorContext",
    logstash_1       |                      "host" => "jclingan-mac.local",
    logstash_1       |                  "facility" => "jboss-logmanager",
    logstash_1       |                  "Severity" => "DEBUG",
    logstash_1       |                "@timestamp" => 2020-10-18T08:44:32.992Z,
    logstash_1       |                  "@version" => "1",
    logstash_1       |               "source_host" => "172.27.0.1",
    logstash_1       |             "MessageParam0" =>
    "org.jboss.resteasy.core.interception.jaxrs.ServerWriterInterceptorContext",
    logstash_1       |                   "message" => "Interceptor Context:
    org.jboss.resteasy.core.interception.jaxrs.ServerWriterInterceptorContext,  Method
    : proceed"
    logstash_1       | }
    ...
    ```

    ① It is left up to the student to view data in Kibana

19. Stop logging services and clean up (Optional)

application.properties

```
...
quarkus.log.syslog.enable=false                       ①
...
quarkus.log.handler.gelf.enabled=false                ②
...
quarkus.log.handler.file."REST_TRACE".enable=false    ③
```

① Disable syslog logging

② Disable GELF logging

③ Disable file logging for REST_TRACE named handler

Terminal 3

```
docker-compose -f docker/elk.yml down
docker stop syslog-ng
```

Quarkus Debugging and Command Mode

This section covers Quarkus debugging and command mode. Debugging is straightforward but there are some useful things to know. Command mode allows Quarkus to run command line applications.

1. Disable debugging

   Terminal 1

   ```
   mvn quarkus:dev -Ddebug=false       ①
   ```

   ① -Ddebug=false disables debugging. -Ddebug=true enables debuggingon port 5005 (default)

2. Customize debugging port

   Terminal 1

   ```
   mvn quarkus:dev -Ddebug=5006       ①
   ```

   ① Debug on port 5006

3. Suspend JVM until debugger attached

   Terminal 1

   ```
   mvn quarkus:dev -Dsuspend=y
   ```

4. Create firststeps application

   Terminal 2

   ```
   cd working                                                  ①
   mvn io.quarkus:quarkus-maven-plugin:1.10.2.Final:create \  ②
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=firststeps \
       -DclassName="org.acme.FirstStepsResource" \
       -Dpath="/firststeps"
   ```

   ① Go to the top-level project then working subdirectory

   ② Create the first-steps application

5. Add class FirstStepsMain

FirstStepsMain.java

```
@QuarkusMain                                                    ①
public class FirstStepsMain implements QuarkusApplication {     ②
    @Override
    public int run(String... args) throws Exception {           ③
        System.out.println("******* " + args[0]                 ④
                           + " " + args[1]
                           + " ********");
        return 0;                                               ⑤
    }
}
```

① Application entry point

② QuarkusApplication interface requires a run() method

③ Run method that passes in command line arguments

④ Print out command line arguments

⑤ Return value back to shell (or whatever invoked application)

6. Test the main method

Terminal 2

```
mvn quarkus:dev \
    -Dquarkus.http.port=8088 \          ①
    -Ddebug=5088 \                      ②
    -Dquarkus.args="Hello1 Hello2"      ③
```

① Set a different port to avoid conflict with *Greeting Service*

② Set a different debug port to avoid conflict with *Greeting Service*

③ Define command line arguments to *FirstSteps Service*.

Terminal 2 output

```
...

******* Hello1 Hello2 ********          ①


...

Quarkus application exited with code 0      ②
Press Enter to restart or Ctrl + C to quit  ③
```

① Application output

② The application exited with a return value of 0

③ Quarkus dev mode works with command line applications. Make a change and press <Entr>.

7. Run as a jar file

   Terminal 2

   ```
   mvn clean package
   java -jar \
       -Dquarkus.http.port=8088                         ①
       -Ddebug=5088 \
       target/firststeps-1.0-SNAPSHOT-runner.jar  \
       Hello1 Hello2                                    ②
   ```

   ① System properties defined **before** jar file defined. With developer mode, they are defined after the maven goal

   ② Command line args specified at end of command line

   Terminal 2 output

   ```
   ...

   2020-10-18 13:37:29,552 INFO  [io.quarkus] (main) Installed features: [cdi,
   resteasy]
   ******* Hello1 Hello2 ********
   2020-10-18 13:37:29,610 INFO  [io.quarkus] (main) firststeps stopped in 0.045s

   ...
   ```

8. Add main method so application can be run from IDE

   FirstStepsMain.java

   ```
   public static void main(String... args ) {
       Quarkus.run(FirstStepsMain.class, args);  ①
   }
   ```

   ① Call Quarkus.run on a QuarkusApplication implementation class

9. Add microprofile-rest-client and resteasy-jsonb extensions

   Terminal 2

   ```
   mvn quarkus:add-extension \
       -Dextensions=rest-client,resteasy-jsonb
   ```

Terminal 2 output

```
...

 Extension io.quarkus:quarkus-rest-client has been installed
 Extension io.quarkus:quarkus-resteasy-jsonb has been installed
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------

...
```

10. Create GreetingsRestClient to invoke GreetingService REST endpoints. The MicroProfile Rest Client API is beyond the scope of this tutorial.

    GreetingsRestClient.java

```java
@Path("/greeting")
@RegisterRestClient(baseUri = "http://localhost:8080")
public interface GreetingsRestClient {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello();

    @GET
    @Path("/list")
    @Produces(MediaType.APPLICATION_JSON)
    public List<String> listGreetings();

    @GET
    @Path("/list/random")
    @Produces(MediaType.TEXT_PLAIN)
    public String randomGreeting();
}
```

11. Update FirstStepsMain to invoke *Greeting Service*. A `QuarkusMain` is an `ApplicationScoped` bean so it can inject and invoke other CDI beans.

FirstStepsMain.java

```
@Inject
@RestClient                                                  ①
GreetingsRestClient client;

@Override
public int run(String... args) throws Exception {
    System.out.println("******* " + args[0]
                          + " " + args[1]
                          + " ********");
    System.out.println(client.listGreetings());              ②
    return 0;
}
```

① Inject the `GreetingsRestClient`

② Print the greetings list

12. Run the application from the IDE

IDE output

```
...

******* Hello1 Hello2 ********
[Howdy from app.prop, Hola from app.prop, Hello from app.prop]

...
```

13. Wait for application to exit.

FirstStepsMain.java

```
    @Override
    public int run(String... args) throws Exception {
        System.out.println("******* " + args[0]
                              + " " + args[1]
                              + " ********");
        System.out.println(client.listGreetings());
        Quarkus.waitForExit();                               ①
        return 0;
    }
```

① Wait for the application to exit

NOTE To exit an application using `Quarkus.waitForExit()`, call `Quarkus.asyncExit()`.

14. Stop application started from IDE

15. Run the application from the IDE

    Terminal 2

    ```
    mvn quarkus:dev \
        -Dquarkus.http.port=8088 \
        -Ddebug=5088 \
        -Dquarkus.args="Hello1 Hello2"
    ```

    Terminal 2 output

    ```
    ...

    ******* Hello1 Hello2 ********
    [Howdy from app.prop, Hola from app.prop, Hello from app.prop]

    ... ①
    ```

    ① The application is still running

Quarkus Testing

This section covers testing Quarkus applications.

1. Review generated `FirstStepsResourceTest` class

   FirstStepsResourceTest.java

   ```
   @QuarkusTest                               ①
   public class FirstStepsResourceTest {

       @Test                                  ②
       public void testHelloEndpoint() {
           given()                            ③
             .when().get("/firststeps")       ④
             .then()                          ⑤
                .statusCode(200)              ⑥
                .body(is("Hello RESTEasy"));  ⑦
       }

   }
   ```

   ① `@QuarkusTest` bootstraps Quarkus on port 8081

   ② JUnit test

   ③ RestAssured static method. `given` defines REST call parameters. There are none in this particular test

   ④ Invokes http endpoint

   ⑤ Begin to validate / assert response

   ⑥ Validate the response code of *200*

   ⑦ Validate the response is *Hello RESTEasy*

2. Run the test

   Terminal 3

   ```
   mvn test
   ```

   Terminal 3 output

   ```
   ...

   [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

   ...
   ```

3. Add methods to `FirstStepsResource.java` to invoke the greeting service `/list/random` and `/list` endpoints

   .

```
@Inject
@RestClient
GreetingsRestClient client;                  ①

@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Path("/list")
public List<String> listGreetings() {
    return client.listGreetings();
}

@GET
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@Path("/random")
public String getRandom() {
    return client.randomGreeting();
}
```

   ① Use the Rest Client to invoke the greeting service endoints

4. Update `FirstStepsResourceTest` to test JSON response

   FirstStepsResourceTest.java

```
@Test
public void testRestClient() {
    given()
        .accept(ContentType.JSON)
    .when().get("/firststeps/list")
    .then()
        .body("[0]", is("Howdy from app.prop"))    ①
        .body("[1]", is("Hola from app.prop"))     ①
        .body("[2]", is("Hello from app.prop"));   ①
}
```

   ① Provide a JSON path expresion to match/validate. These expressions gets first, second, and third entry (index starts at zero). Because the matcher is `is()`, the results have to match exactly

5. Run the test

Terminal 3

```
mvn test
```

Terminal 3 output

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

6. Annotate `FirstStepsResourceTest` with @TestHttpEndpoint to abstract top-level PATH from tests

   FirstStepsResourceTest.java

   ```
   @QuarkusTest
   @TestHTTPEndpoint(FirstStepsResource.class)     ①
   public class FirstStepsResourceTest {

   ...
   ```

   ① Use the top-level path defined in the JAX-RS Resource class ("/firststeps")

7. Remove "/firststeps" from test URLs

   FirstStepsResourceTest.java

   ```
   public void testHelloEndpoint() {
       given()
           .when().get()                ①
           .then()
   ...

   public void testRestClient() {
       given()
           .accept(ContentType.JSON)
           .when().get("/list")         ①
   ...

   public void testRestClientAll() {
       given()
           .accept(ContentType.JSON)
           .when().get("/list")
   ...
   ```

   ① Removed the leading "/firststeps"

8. Run tests to validate the change works

Terminal 3

```
mvn test
```

Terminal 3 output

```
...

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

...
```

---

9. Update `FirstStepsResourceTest.java` to test random greeting

   FirstStepsResourceTest.java

```
@Test
public void testRestClientRandom() {
    String string =
            given()
            .accept(ContentType.TEXT)
            .when().get("/random")
            .body().asString();                      ①

    assertThat(string,                               ②
            anyOf(is("Howdy from app.prop"),         ③
                    is("Hola from app.prop"),
                    is("Hello from app.prop")));
}
```

① Get the REST response as a string (it is a text/plain media type)

② Standalone assertion instead of asserting directly in the response

③ anyOf() validates that at least one of multiple conditions is true

---

10. Test testRestClientRandom code

    Terminal 3

```
mvn test
```

Terminal 3 output

```
....

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

....
```

11. Stop the *Greeting Service*

    Terminal 1

    ```
    CTRL-C          ①
    ```

    ① Press CTRL-C in Terminal 1 to stop the greeting service

12. Test the *FirstSteps Service*

    Terminal 3

    ```
    mvn test
    ```

    Terminal 3 output

    ```
    ...     # Lots of Java exceptions because Greetings Service is down

    [INFO]
    [ERROR] Tests run: 3, Failures: 2, Errors: 0, Skipped: 0   ①
    [INFO]

    ...
    ```

    ① Most tests fail

13. Mock the service endoints using CDI alternatives. Create the `CDIMockGreetingService` class that implements the *Greeting Service* functions. Technically, this class is not mocking the greeting service but mocking the *FirstStepsResource* methods that invoke the greeting service. This idea in this step is just to show how methods can be mocked using CDI alternatives.

CDIMockGreetingSerevice.java

```
@Mock
@ApplicationScoped
public class CDIMockGreetingService extends FirstStepsResource {
    @Override
    public List<String> listGreetings() {
       return Arrays.asList("Howdy from app.prop",
               "Hola from app.prop",
               "Hello from app.prop");
    }

    @Override
    public String getRandom() {
        Random r = new Random();
        List<String> greetings = listGreetings();
        return greetings.get(r.nextInt(listGreetings().size()));
    }
}
```

NOTE @Mock is always active. Make sure the mocking class is in the `src/test/java` directory tree so it is only active during tests.

14. Test the mock

Terminal 3

```
mvn test
```

Terminal 3 output

```
...

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

...
```

15. Disable CDI alternative mocking to prepare for using wiremock

CDIMockGreetingService.java

```
//@Mock      ①
```

① The easiest way to disable it is to comment out the @Mock annotation

16. Add wiremock dependency to begin mocking the *Greeting Service*

    pom.xml

    ```
    <dependency>
      <groupId>com.github.tomakehurst</groupId>
      <artifactId>wiremock-jre8</artifactId>
      <version>2.26.3</version>
      <scope>test</scope>
    </dependency>
    ```

17. Create the wiremock class to mock the Greeting service

WiremockGreetingService.java

```
import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static com.github.tomakehurst.wiremock.core.WireMockConfiguration.options;

public class WiremockGreetingService
    implements QuarkusTestResourceLifecycleManager {                        ①

  private WireMockServer wireMockServer;

  @Override
  public Map<String, String> start() {
    wireMockServer = new WireMockServer(options().dynamicPort());           ②
    wireMockServer.start();                                                 ③

    wireMockServer.stubFor(get(urlEqualTo("/greeting"))                     ④
        .willReturn(aResponse()
            .withHeader("Content-Type", "text/plain")                       ⑤
            .withBody("Hello RESTEasy")));                                  ⑥

    wireMockServer.stubFor(get(urlEqualTo("/greeting/list"))                ⑦
        .willReturn(aResponse()
          .withHeader("Content-Type", "application/json")                   ⑧
          .withBody("[\"Howdy from app.prop\"," +
              "\"Hola from app.prop\"," +
              "\"Hello from app.prop\"]")));

    wireMockServer.stubFor(get(urlEqualTo("/greeting/list/random"))         ⑨
        .willReturn(aResponse()
          .withHeader("Content-Type", "text/plain")
          .withBody("Howdy from app.prop")));

    return Collections
        .singletonMap("org.acme.GreetingsRestClient/mp-rest/url",           ⑩
        wireMockServer.baseUrl());
}

    @Override
    public void stop() {
        if (null != wireMockServer) {
            wireMockServer.stop();                                          ⑪
        }
    }
}
```

① Starting/stopping the wiremock server ties into the Quarkus lifecycle of Quarkus. When Quarkus starts/stops, the wiremock server will start/stop first

② Pick a random port for run wiremock to listen on

③ Start the wiremock server when Quarkus lifcycle manager invokes start()

④ Create a stub that responds to the "/greeting" endpoint

⑤ Define a content type header

⑥ Response body is "Hello RESTEasy"

⑦ Stub responds to "/greeting/list" endpoint.

⑧ Content type is "application/json"

⑨ Stub responds to /greeting/list/random. To keep this example simple, it hard codes a response

⑩ Override the GreetingsRestClient URL property to listen to proper endpoint (with dynamic port)

⑪ Stop the wiremock server

---

18. Annotate `FirstStepsResourceTest` with@QuarkusTestResource(WiremockGreetingService.class) to start resources before Quarkus boots the application

    FirstStepsResourceTest.java

    ```
    @QuarkusTest
    @TestHTTPEndpoint(FirstStepsResource.class)
    @QuarkusTestResource(WiremockGreetingService.class)     ①
    public class FirstStepsResourceTest {
    ```

    ① Start Wiremock service before Quarkus application starts

---

19. Test WiremockGreetingService

    Terminal 3

    ```
    mvn test
    ```

Terminal 3 output

```
2020-10-18 20:22:50,259 INFO  [org.ecl.jet.uti.log] (main) Logging initialized
@2076ms to org.eclipse.jetty.util.log.Slf4jLog
2020-10-18 20:22:50,350 INFO  [org.ecl.jet.ser.Server] (main) jetty-
9.4.18.v20190429; built: 2019-04-29T20:42:08.989Z; git:
e1bc35120a6617ee3df052294e433f3a25ce7097; jvm 11.0.8+10-jvmci-20.2-b03
2020-10-18 20:22:50,373 INFO  [org.ecl.jet.ser.han.ContextHandler] (main) Started
o.e.j.s.ServletContextHandler@167279d1{/__admin,null,AVAILABLE}
2020-10-18 20:22:50,376 INFO  [org.ecl.jet.ser.han.ContextHandler] (main) Started
o.e.j.s.ServletContextHandler@730e5763{/,null,AVAILABLE}

...

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

...
```

NOTE TestContainers can be used with Quarkus. See example code here. and the TestResource here.

Quarkus Hibernate ORM with Panache

This section covers Hibernate ORM with Panache, an ease of use layer on top of Hibernate (and JPA).

NOTE This tutorial uses a single persistent unit, but beginning with Quarkus 1.8 multiple persistence units are supported

1. Add necessary extensions

   Terminal 3

   ```
   mvn quarkus:add-extension -Dextensions=hibernate-orm-panache,jdbc-h2  ①
   ```

   ① Add quarkus-hibernate-orm extension provides the Hibernate ORM with Panache functionality. However, a JDBC driver is required so that is added too. This section uses the H2 in-memory database. When packaged as a binary in the *Packaging* section, the PostgreSQL database is used.

2. Add database configuration properties

   application.properties

   ```
   # Database Configuration

   quarkus.datasource.db-kind=h2                                    ①
   quarkus.datasource.jdbc.url=jdbc:h2:mem:test                     ②
   quarkus.hibernate-orm.log.sql=true                               ③
   quarkus.hibernate-orm.sql-load-script=import.sql                 ④
   quarkus.hibernate-orm.database.generation=drop-and-create        ⑤
   ```

   ① Use the H2 database

   ② Database connection string

   ③ Log SQL to the log

   ④ Load the database with "dummy data". `import.sql` is loaded from src/main/resources`

   ⑤ Drop the database table and re-create it on each deployment. When Quarkus is in developer mode, this is done each time a change is made and an (REST, messsaging) endpoint is called.

3. Populate import.sql with "dummy data" to enable basic functionality and testing

src/main/resources/import.sql

```
insert into Training(ID, NAME) values (nextval('hibernate_sequence'), 'Quarkus
First Steps');
insert into Training(ID, NAME) values (nextval('hibernate_sequence'), 'Quarkus and
MicroProfile');
insert into Student(ID, training_id, NAME) values (nextval('hibernate_sequence'),
1,  'John Doe');
insert into Student(ID, training_id, NAME) values (nextval('hibernate_sequence'),
1,  'Jane Doe');
insert into Student(ID, training_id, NAME) values (nextval('hibernate_sequence'),
2,  'John Duke');
insert into Student(ID, training_id, NAME) values (nextval('hibernate_sequence'),
2,  'Jane Quarkus');
```

4. Create Student Entity

Student.java

```
@Entity
public class Student extends PanacheEntity {   ①
    public String name;                         ②
}
```

① Panache entities extend `PanacheEntity`. A `PanacheEntity` provides a unique `id` field`

② Panache entities fields are public. If an entity defines an accessor method, then that method will be called.

5. Create Training entity

Training.java

```
@Entity
public class Training extends PanacheEntity {                    ①
    public String name;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "training_id", referencedColumnName = "id")
    public List<Student> students;

    public Training() {
        students = new ArrayList<Student>();
    }

    public Training(String name) {
        this();
        this.name = name;
    }
}
```

① This entity owns the one-to-many relationship

6. Create REST resource to manage the database

   TrainingResource.java

```
@Path("/training")
public class TrainingResource {                ①
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Training> listTrainings() {
        return Training.listAll();                         ②
    }

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Training postTraining(Training training) {
        training.persist();                                ③

        return training;
    }

}
```

① This class uses the Panaache "active record" API

② Return all trainings

③ Persist the traianing

7. Get all trainings

   Terminal 3

   ```
   curl localhost:8088/training
   ```

   Terminal 3 output

   ```
   [{"id":1,"name":"Quarkus First Steps","students":[{"id":3,"name":"John Doe"},{"id"
   :4,"name":"Jane Doe"}]},{"id":2,"name":"Quarkus and MicroProfile","students":[{"id
   ":5,"name":"John Duke"},{"id":6,"name":"Jane Quarkus"}]}]
   ```

   NOTE Quarkus contains helpful query features like automatic paging and sorting, and simplified queries.

8. Add a training

   Terminal 3

   ```
   curl -i \
        -H"Content-Type: application/json" \
        -X POST \
        -d '{ "name" : "MicroProfile", "students" : [{"name":"John
   Config"},{"name":"Jane Health"}]}' \
        localhost:8088/training
   ```

   Terminal 3 output

   ```
   {"id":7,"name":"MicroProfile","students":[{"id":8,"name":"John Config"},{"id":9,
   "name":"Jane Health"}]} ①
   ```

   ① The ID’s will increment different each time this is run

9. List the trainings

   Terminal 3

   ```
   curl -is localhost:8088/training
   ```

Terminal 3 output

```
[{"id":1,"name":"Quarkus First Steps","students":[{"id":3,"name":"John Doe"},{"id"
:4,"name":"Jane Doe"}]},{"id":2,"name":"Quarkus and MicroProfile","students":[{"id
":5,"name":"John Duke"},{"id":6,"name":"Jane Quarkus"}]},{"id":7,"name":
"MicroProfile","students":[{"id":8,"name":"John Config"},{"id":9,"name":"Jane
Health"}]}] ①
```

① The new training has been addeed.

---

10. Add a data repository class

TrainingRepository.java

```java
@ApplicationScoped
public class TrainingRepository implements PanacheRepository<Training> {
}
```

---

11. Update TrainingResource to use data repository

Add to TrainingResource.java

```java
@Inject
TrainingRepository repository;                              ①

@POST
@Transactional
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Path("/repository")
public Training postRepositoryTraining(Training training) {
    repository.persist(training);                           ②

    return training;
}
```

① Inject a `TrainingRepository` instance

② Persist the training object using the data repository API

---

12. Test postRepositoryTraining

Terminal 3

```
curl -i \
     -H"Content-Type: application/json" \
     -X POST \
     -d '{ "name" : "MicroProfile", "students" : [{"name":"John
Config"},{"name":"Jane Health"}]}' \
     localhost:8088/training/repository
```

Terminal 3 output

```
{"id":7,"name":"MicroProfile","students":[{"id":8,"name":"John Fault Tolerance"},{
"id":9,"name":"Jane JWT Auth"}]}
```

---

13. Add a custom find method to TrainingRepository

    TrainingRepository.java

```java
public Training findByName(String name) {           ①
    return find("name", name).firstResult();        ②
}
```

① Find Training by name

② Find method can take a shortened Hibernate query string as parameter

---

14. Add REST endpoint to access `findByName()`

    TrainingResource.java

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/repository/{name}")
public Training getRepositoryTraining(@PathParam("name") String name) {
    return repository.findByName(name);
}
```

---

15. Re-add the MicroProfile training class. Quarkus will re-start the application after each change, and will import the data from import.sql each time. For that reason, the MicroProfile training is gone and we have to re-add it.

Terminal 3

```
curl -i \
     -H"Content-Type: application/json" \
     -X POST \
     -d '{ "name" : "MicroProfile", "students" : [{"name":"John
Config"},{"name":"Jane Health"}]}' \
     localhost:8088/training/repository
```

Terminal 3 output

```
{"id":7,"name":"MicroProfile","students":[{"id":8,"name":"John Fault Tolerance"},{
"id":9,"name":"Jane JWT Auth"}]}
```

---

16. Get Training by name

    Terminal 3

    ```
    curl -is localhost:8088/training/repository/MicroProfile
    ```

    Terminal 3 output

    ```
    {"id":7,"name":"MicroProfile","students":[{"id":8,"name":"John Fault Tolerance"},{
    "id":9,"name":"Jane JWT Auth"}]}
    ```

    NOTE Quarkus also supports Rest Data Panache, which is similar to Spring Data Rest. This will auto-generate CRUD endpoints on top of a PanacheRepository.

---

17. Test Panache active record pattern. Add quarkus-panache-mock maven dependency. This dependency is not an extension, so it must be added directly to pom.xml. Normally Mockito does not allow testing static methods, but the quarkus-panache-mock libary uses Mockito to mock static methods.

    Add to pom.xml

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

---

18. Test the active record style

TrainingTest.java

```
@QuarkusTest
public class TrainingTest {
    @Test
    public void testActiveRecord() {
        PanacheMock.mock(Training.class);                                   ①

        // Mock a training
        Training training = new Training();
        Mockito.when(Training.findById(10L)).thenReturn(training);
        Mockito.when(Training.count()).thenReturn(1L);
        Mockito.when(Training.listAll())
              .thenReturn(Arrays.asList(training));

        // Make assertions
        Assertions.assertSame(training, Training.findById(10L));
        Assertions.assertSame(Training.listAll().size(), 1);
        Assertions.assertSame(Training.count(), 1L);

        PanacheMock.verify(Training.class).count();                         ②
        PanacheMock.verify(Training.class).listAll();
        PanacheMock.verify(Training.class,
            Mockito.atLeastOnce()).findById(Mockito.any());
    }
}
```

① To mock a Panache active record entity, use `PanacheMock.mock()` instead of `Mockito.mock()`

② `Mockito.verify()` and `Mockito.do*()` must be replaced with `PanacheMock.verify()` and `PanacheMock.do*()`, where * is the full method name

19. Test Panache active record entity

Terminal 3

```
mvn test
```

Terminal 3 output

```
...

[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS

...
```

20. Update test to verify Panache Entity encapsulation

Add lines TrainingTest.java

```
    @Test
    public void testActiveRecord() {
        PanacheMock.mock(Training.class);

        // Mock a training
        Training training = new Training();
        training.name ="Quarkus Deep Dive";                    ①
        Mockito.when(Training.findById(10L)).thenReturn(training);
        Mockito.when(Training.count()).thenReturn(1L);
        Mockito.when(Training.listAll())
            .thenReturn(Arrays.asList(training));

        // Make assertions
        Assertions.assertSame(training, Training.findById(10L));
        Assertions.assertSame(Training.listAll().size(), 1);
        Assertions.assertSame(Training.count(), 1L);
        Assertions.assertTrue("Quarkus Deep Dive!".equals(training.name)); ②

        PanacheMock.verify(Training.class).count();
        PanacheMock.verify(Training.class).listAll();
        PanacheMock.verify(Training.class,
             Mockito.atLeastOnce()).findById(Mockito.any());
    }
```

① Set the training name, which is a public field

② Assert that the training name now has an appended exclamation point.

NOTE Quarkus will call accessor methods on public fields if they exist, even when the field is directly accessed. Behind the scenes, Quarkus (Hibernatee ORM with Panache framework specifically) will create any missing accessor methods and rewrite direct field access to invoke them.

21. Update Training class with a getname() accessor method that appends an exclamation point

Add to Training.java

```
public String getName() {          ①
    return name + "!";
}
```

① When the training.name is referenced directly, the getName() method is called instead.

22. Test Panache entity accessor method

Terminal 3

```
mvn test
```

Terminal 3 output

```
...

[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS

...
```

23. Test data repository method. Added a validation of the custom `findByName()` method that exists in the data repository method. The same method could also be used in the active record method.

Add testDataRepository() to TrainingTest.java

```
@InjectMock                                                          ①
TrainingRepository repository;

@Test
public void testDataRepository() {
    // Mock a training
    Training training = new Training();
    training.name = training.name = "Quarkus Deep Dive";

    Mockito.when(repository.findById(10L)) .thenReturn(training);
    Mockito.when(repository.count()) .thenReturn(1L);
    Mockito.when(repository.listAll())
            .thenReturn(Arrays.asList(training));
    Mockito.when(repository
            .findByName("Quarkus Deep Dive"))                        ②
            .thenReturn(training);

    // Make assertions
    Assertions.assertSame(training, repository.findById(10L));
    Assertions.assertSame(repository.count(), 1L);
    Assertions.assertSame(repository.listAll().get(0), training);
    Assertions.assertSame(repository.findByName("Quarkus Deep Dive"),   ③
            training);
    Assertions.assertTrue("Quarkus Deep Dive!".equals(training.name));

    Mockito.verify(repository).count();                              ④
    Mockito.verify(repository).listAll();
    Mockito.verify(repository).findById(Mockito.any());
    Mockito.verify(repository).findByName(Mockito.any());
}
```

① @InjectMock allows mocked beans to be local to this class instead of for all classes like @Mock (CDI alternative)

② Mock custom method

③ Assertion on custom method

④ Can call `Mockito.verify()` directly using data repository method

24. Test the data repository

Terminal 3

```
mvn test
```

Terminal 3 output

```
...

[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS

...
```

Quarkus Security

This section covers HTTP Basic authentication with authentication using property files, OpenID Connect, and access tokens

1. Add necessary extensions

   Terminal 3

   ```
   mvn quarkus:add-extension -Dextensions=security-properties-file  ①
   ```

   ① The extension is really quarkus-elytron-security-properties-file, but add-extension will attempt to add an extension based on a unique string

   Terminal 3 output

   ```
   ...

    Extension io.quarkus:quarkus-elytron-security-properties-file has been installed

   [INFO] ------------------------------------------------------------------------
   [INFO] BUILD SUCCESS

   ...
   ```

---

2. Enable basic authentication and provide credentials in application.properties

   application.properties

   ```
   # Property file security realm

   quarkus.http.auth.basic=true                                 ①
   quarkus.security.users.embedded.enabled=true                 ②
   quarkus.security.users.embedded.plain-text=true              ③
   quarkus.security.users.embedded.users.john=john              ④
   quarkus.security.users.embedded.users.jane=jane              ⑤
   quarkus.security.users.embedded.roles.john=admin,user        ⑥
   quarkus.security.users.embedded.roles.jane=user              ⑦
   ```

   ① Enable http basic authentication

   ② Enable file-based user credentials

   ③ Passwords are in plain text (vs MD5 hash, for example)

   ④ Define user john

   ⑤ Define user jane

   ⑥ Define john’s roles

⑦ Define jane’s roles

3. Protect the `/training` endpoint (only the top-level path)

   TrainingResource.java

   ```
   @GET
   @Produces(MediaType.APPLICATION_JSON)
   @RolesAllowed("admin")                    ①
   public List<Training> listTrainings() {
       return Training.listAll();
   }
   ```

   ① Only users in the *admin* role can access this resource

4. Test the endpoint with a user in *admin* role

   Terminal 3

   ```
   curl -i --user john:john localhost:8088/training      ①
   ```

   ① `--user john:john` is the format for "user:password"

   Terminal 3 output

   ```
   HTTP/1.1 200 OK
   Content-Length: 223
   Content-Type: application/json

   [{"id":1,"name":"Quarkus First Steps!","students":[{"id":3,"name":"John Doe"},
   {"id":4,"name":"Jane Doe"}]}, {"id":2,"name":"Quarkus and MicroProfile!",
   "students":[{"id":5,"name":"John Duke"},{"id":6,"name":"Jane Quarkus"}]}]
   ```

5. Test endpoint with insufficient privileges

   Terminal 3

   ```
   curl -i --user jane:jane localhost:8088/training
   ```

   Terminal 3 output

   ```
   HTTP/1.1 403 Forbidden
   Content-Length: 0
   ```

6. Add test-security extension to improve security test experience

   pom.xml

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

7. Create a test that provides credentials when running test on protected resource

   Create file SecurityTest.java

   ```
   @QuarkusTest
   @TestSecurity(user="john", roles= {"admin", "user"})                          ①
   //@TestSecurity(authorizationEnabled = false)                                 ②
   @TestHTTPEndpoint(TrainingResource.class)
   public class SecurityTest {
       @Test
       void testTraining() {
           given()
                   .when().get()
                   .then()
                   .statusCode(200)
                   .body("[1].name", is("Quarkus and MicroProfile!"))        ③
                   .body("[1].students[1].id", is(6))                        ④
                   .body("[1].students[1].name", is("Jane Quarkus"));        ⑤
       }
   }
   ```

   ① Use `@TestSecurity` annotation to define credentials the tests are run with

   ② Alternatively, disable authorization entirely. Feel free to try it and comment out the other `@TestSecurity` annotation

   ③ This shows RestAssured using a JSONPath expression to validate the training name

   ④ This shows RestAssured using a JSONPath expression to validate the student id

   ⑤ This shows RestAssured using a JSONPath expression to validate the student name

8. Test @TestSecurity annotation

   Terminal 3

   ```
   mvn test
   ```

Terminal 3 output

```
...

[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS

...
```

---

9. Install Keycloak for authentication and authorization

   Terminal 3

   ```
   # From the top-level project directory
   docker/start-keycloak.sh
   ```

   Terminal 3 output

   ```
   8682008f6df33ff3a78fefb307cf02f3ae47ca9fe1b001c5b78b4e529dbeebee ①
   ```

   ① Container ID will change each time a container is created

---

10. Add the OIDC extension

    Terminal

    ```
    mvn quarkus:add-extension -Dextensions=oidc
    ```

    Terminal output

    ```
    ...

    ✅ Extension io.quarkus:quarkus-oidc has been installed
    [INFO] ------------------------------------------------------------------------
    [INFO] BUILD SUCCESS

    ...
    ```

---

11. Disable http authentication and configure OIDC configuration

application.properties

```
quarkus.http.auth.basic=false                                           ①

# ...

# OIDC configuration for keycloak server.
# Keycloak will inject the jwt info into the access token

quarkus.oidc.enabled=true                                               ②
quarkus.oidc.auth-server-url=http://localhost:8180/auth/realms/quarkus  ③
quarkus.oidc.client-id=firststeps                                       ④
quarkus.oidc.application-type=web-app                                   ⑤
quarkus.oidc.logout.path=/logout                                        ⑥
quarkus.oidc.logout.post-logout-path=/                                  ⑦
quarkus.oidc.roles.source=accesstoken                                   ⑧
```

① Disable HTTP basic authentication (change to false)

② Enable oidc authentication

③ Set the URL to the (keycloak) auth server

④ The ID associated with this application

⑤ Authorization flow - webapp

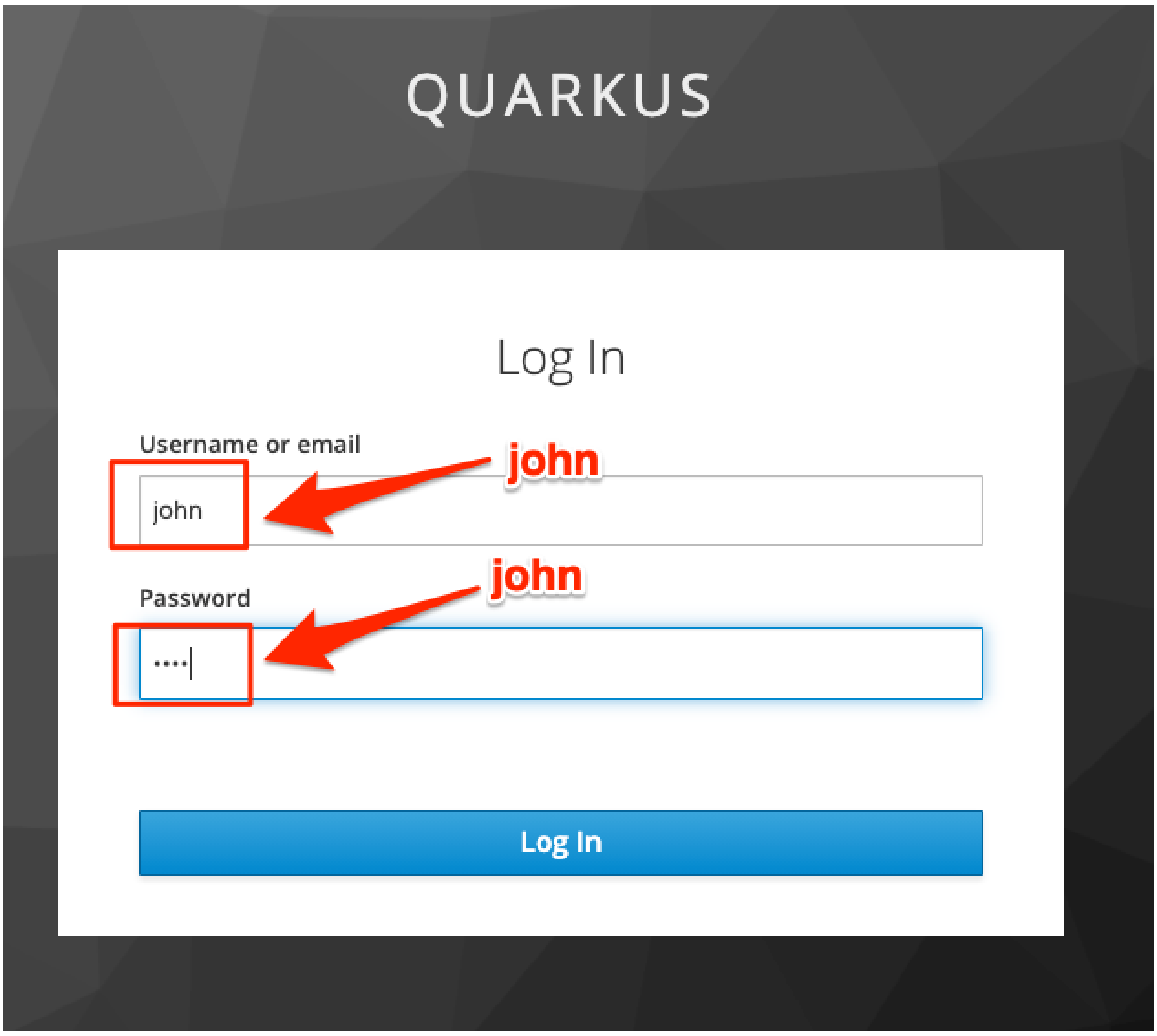
⑥ Logout URL

⑦ Path to go to after logging out

⑧ Source of principle

---

12. Test the oidc authorization flow
    - In a browser, go to http://localhost:8088/training
    - Enter user credential with:
        - User: **john**
        - Password: **john**



Browser output after login

```
[{"id":1,"name":"Quarkus First Steps!","students":[{"id":3,"name":"John Doe
"},{"id":4,"name":"Jane Doe"}]},{"id":2,"name":"Quarkus and MicroProfile!",
"students":[{"id":5,"name":"John Duke"},{"id":6,"name":"Jane Quarkus"}]}]
```

Quarkus Packaging

This section covers how to package and run applications.

1. Create a thinjar

   Terminal 2 (firststeps application)

   ```
   mvn clean package
   ```

   Terminal 2

   ```
   java \
       -Dquarkus.http.port=8088 \                        ①
       -Ddebug=5088 \
       -jar target/firststeps-1.0-SNAPSHOT-runner.jar \      ②
       Hello1 Hello2                                     ③
   ```

   ① When executing a runnable jar file, the system properties belong before the jar file name

   ② Two jar files are generated. `firststeps-1.0-SNAPSHOT.jar` is generated by the maven compiler plugin. The Quarkus plugin generates the `firststeps-1.0-SNAPSHOT-runner.jar`. The runnable Quarkus application is the latter.

   ③ Command line arguments belong after the jar file.

   Feel free to check various endpoints to validate they still work.

2. Create an uber-jar

   Terminal 2

   ```
   mvn clean package -Dquarkus.package.type=uber-jar
   ```

   Terminal 2

   ```
   java \
       -Dquarkus.http.port=8088 \
       -Ddebug=5088 \
       -jar target/firststeps-1.0-SNAPSHOT-runner.jar \
       Hello1 Hello2
   ```

   Feel free to check various endpoints to validate they still work.

3. Quarkus cannot run H2 in-memory database in native mode, even thought it will successfully

compile to native. For this reason, the production binary will use a PostgreSQL database. So, let’s install the PostgreSQL JDBC extension (driver).

Terminal 2

```
mvn quarkus:add-extension -Dextensions=jdbc-postgres
```

4. Start PostgreSQL

   Terminal 3

```
docker run \
    -d \
    --ulimit memlock=-1:-1 \
    -it \
    --rm=true \
    --memory-swappiness=0 \
    --name postgres \
    -e POSTGRES_USER=quarkus \
    -e POSTGRES_PASSWORD=quarkus \
    -e POSTGRES_DB=quarkus \
    -p 5432:5432 \
    postgres:11.2
```

5. Configure PostgreSQL. This will still drop-and-create and import data for a "production" configuration, but hey, it’s only a tutorial.

   application.properties

```
# PostgreSQL "Production" database configuration

%prod.quarkus.datasource.db-kind=postgresql
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/quarkus
%prod.quarkus.datasource.username=quarkus
%prod.quarkus.datasource.password=quarkus
%prod.quarkus.hibernate-orm.sql-load-script=import.sql
```

6. Compile to a native binary

   Terminal 3

```
mvn clean verify -Pnative
```

7. Run the native binary

Terminal 2

```
target/firststeps-1.0-SNAPSHOT-runner \
  -Dquarkus.http.port=8088 \
  -Ddebug=5088 \
  Hello1 Hello2
```

Terminal 2 output

```
...

[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.acme.SecurityTest
2020-10-20 20:52:50,710 INFO  [io.qua.ely.sec.pro.dep.ElytronPropertiesProcessor]
(build-11) Configuring from MPRealmConfig
2020-10-20 20:52:52,087 INFO  [org.ecl.jet.uti.log] (main) Logging initialized
@2988ms to org.eclipse.jetty.util.log.Slf4jLog
2020-10-20 20:52:52,196 INFO  [org.ecl.jet.ser.Server] (main) jetty-
9.4.18.v20190429; built: 2019-04-29T20:42:08.989Z; git:
e1bc35120a6617ee3df052294e433f3a25ce7097; jvm 11.0.8+10-jvmci-20.2-b03
...

[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0                       ①

...

[firststeps-1.0-SNAPSHOT-runner:99017]    classlist:  10,020.47 ms,  1.07 GB
[firststeps-1.0-SNAPSHOT-runner:99017]        (cap):   3,440.47 ms,  1.07 GB
[firststeps-1.0-SNAPSHOT-runner:99017]        setup:   5,313.04 ms,  1.07 GB
20:53:16,064 INFO  [org.hib.Version] HHH000412: Hibernate ORM core version
5.4.22.Final
20:53:16,070 INFO  [org.hib.ann.com.Version] HCANN000001: Hibernate Commons
Annotations {5.1.0.Final}
20:53:16,105 INFO  [org.hib.dia.Dialect] HHH000400: Using dialect:
io.quarkus.hibernate.orm.runtime.dialect.QuarkusPostgreSQL10Dialect
20:53:31,923 INFO  [org.jbo.threads] JBoss Threads version 3.1.1.Final
[firststeps-1.0-SNAPSHOT-runner:99017]     (clinit):     816.13 ms,  4.21 GB
[firststeps-1.0-SNAPSHOT-runner:99017]   (typeflow):  22,936.15 ms,  4.21 GB
[firststeps-1.0-SNAPSHOT-runner:99017]    (objects):  30,186.20 ms,  4.21 GB
[firststeps-1.0-SNAPSHOT-runner:99017]   (features):   1,465.94 ms,  4.21 GB
[firststeps-1.0-SNAPSHOT-runner:99017]     analysis:  58,155.09 ms,  4.21 GB
[firststeps-1.0-SNAPSHOT-runner:99017]     universe:   2,623.30 ms,  4.20 GB
[firststeps-1.0-SNAPSHOT-runner:99017]      (parse):  10,878.09 ms,  5.69 GB
[firststeps-1.0-SNAPSHOT-runner:99017]     (inline):   9,625.10 ms,  7.36 GB
[firststeps-1.0-SNAPSHOT-runner:99017]    (compile):  43,520.46 ms,  7.59 GB
[firststeps-1.0-SNAPSHOT-runner:99017]      compile:  68,837.54 ms,  7.59 GB
```

```
[firststeps-1.0-SNAPSHOT-runner:99017]        image:   8,449.02 ms,  7.50 GB
[firststeps-1.0-SNAPSHOT-runner:99017]        write:   2,346.50 ms,  7.50 GB
[firststeps-1.0-SNAPSHOT-runner:99017]      [total]: 156,113.60 ms,  7.50 GB
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
159737ms  ②

...

[INFO] -------------------------------------------------------
[INFO]  T E S T S
③
[INFO] -------------------------------------------------------

...

2020-10-20 20:55:40,692 INFO  [io.quarkus] (main) firststeps 1.0-SNAPSHOT native
(powered by Quarkus 1.9.0.Final) started in 0.114s. Listening on:
http://0.0.0.0:8081

...

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

① Tests are run in JVM mode as part of a Java build

② Compiled in 159 seconds

③ Tests run against native binary

---

8. Re-run tests on a native binary without having to re-compile the binary. This is helpful since it takes 159s to re-compile a binary!

   Terminal 2

   ```
   ./mvnw test-compile failsafe:integration-test
   ```

Terminal 2 output

```
...

[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
...

[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.153 s
[INFO] Finished at: 2020-10-20T21:09:40-07:00
[INFO] ------------------------------------------------------------------------
```