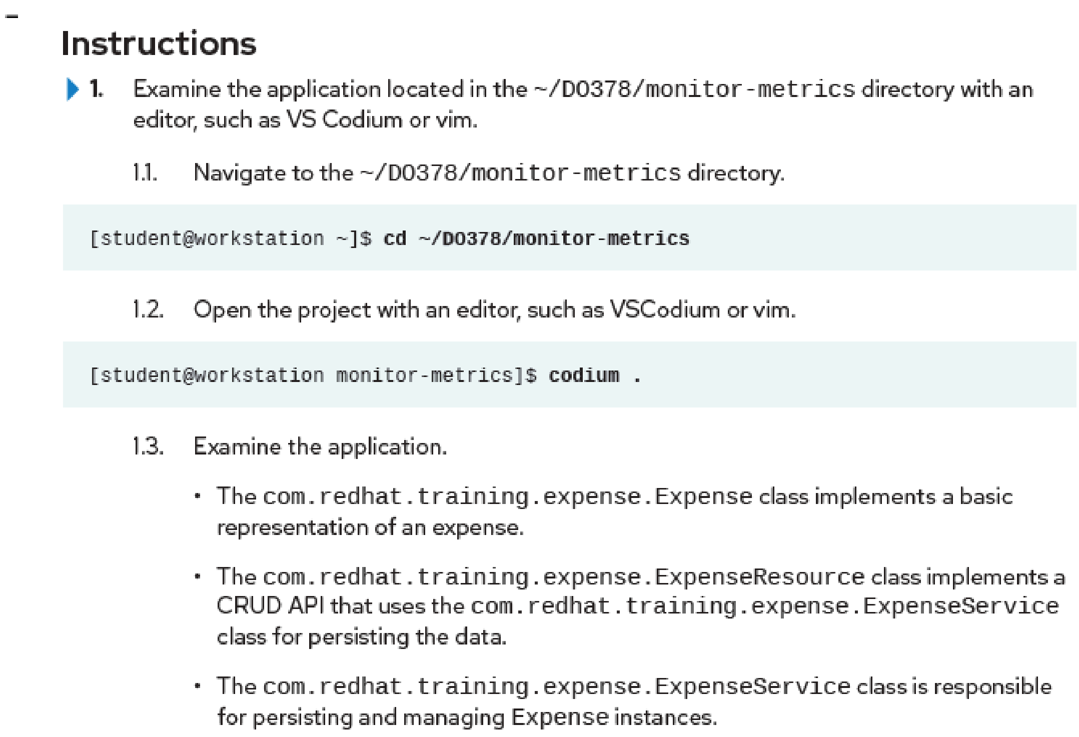
LAB 21: QUARKUS MONITOR METRICS

Autor: José Díaz

Github Repo: https://github.com/joedayz/quarkus-bcp-2025.git

Abre el proyecto **monitor-metrics**



Instructions

1. Examine the application located in the ~/DO378/monitor-metrics directory with an editor, such as VS Codium or vim.

   1.1. Navigate to the ~/DO378/monitor-metrics directory.

```
[student@workstation ~]$ cd ~/DO378/monitor-metrics
```

   1.2. Open the project with an editor, such as VSCodium or vim.

```
[student@workstation monitor-metrics]$ codium .
```

   1.3. Examine the application.

   - The com.redhat.training.expense.Expense class implements a basic representation of an expense.
   - The com.redhat.training.expense.ExpenseResource class implements a CRUD API that uses the com.redhat.training.expense.ExpenseService class for persisting the data.
   - The com.redhat.training.expense.ExpenseService class is responsible for persisting and managing Expense instances.

2. Include the Quarkus extension required to use Micrometer with Prometheus.

   2.1. Return to the terminal window, and use Maven to install the `micrometer-registry-prometheus` extension.

```
[student@workstation monitor-metrics]$ mvn quarkus:add-extension \
-Dextensions=micrometer-registry-prometheus
...output omitted...
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-micrometer-registry-prometheus
 has been installed
...output omitted...
```

   2.2. Use the command `mvn quarkus:dev` to start the application.

```
[student@workstation monitor-metrics]$ mvn quarkus:dev
...output omitted...
... INFO [io.quarkus] ... Listening on: http://localhost:8080
...output omitted...
```

3. Add a metric that counts the number of invocations of the GET and POST endpoints.

   3.1. Open the `ExpenseResource` class, and then inject a `MeterRegistry` instance.

```
package com.redhat.training;

...code omitted...

@Path("/expenses")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ExpenseResource {

    @Inject
    public MeterRegistry registry;

    ...code omitted...
}
```

   3.2. Update the GET endpoint to track the number of invocations of the endpoint. Use the `@Counted` annotation, and set `callsToGetExpenses` as the metric name.

```
@GET
@Counted(value = "callsToGetExpenses")
public Set<Expense> list() {
    return expenseService.list();
}
```

   3.3. Update the POST endpoint to track the number of invocations of the endpoint. Use the `MeterRegistry` instance, and set `callsToPostExpenses` as the metric name.

```
@POST
public Expense create(Expense expense) {
    registry.counter("callsToPostExpenses").increment();

    return expenseService.create(expense);
}
```

3.4. Open a new terminal window, navigate to the project directory, and then execute the `scripts/simulate-traffic.sh` script to generate some requests to the GET and POST endpoints.

```
[student@workstation ~]$ cd ~/DO378/monitor-metrics
[student@workstation monitor-metrics]$ ./scripts/simulate-traffic.sh
GET Response Code: 200
GET Response Code: 200
GET Response Code: 200
POST Response Code: 200
```

3.5. Verify the correctness of the code changes by retrieving the application metrics.

```
[student@workstation monitor-metrics]$ curl http://localhost:8080/q/metrics \
| grep Expenses_total
...output omitted...
# HELP callsToGetExpenses_total
# TYPE callsToGetExpenses_total counter
callsToGetExpenses_total{class=...} 3.0
# HELP callsToPostExpenses_total
# TYPE callsToPostExpenses_total counter
callsToPostExpenses_total 1.0
```

▶ **4.** Add a metric that counts the time consumed by the POST endpoint to persist an expense.

4.1. Open the `ExpenseResource` class, and then update the POST endpoint to track the time consumed persisting expenses. Create a timer called `expenseCreationTime`, and wrap the persistence logic to track the execution time.

```
@POST
public Expense create(Expense expense) {
    registry.counter("callsToPostExpenses").increment();

    return registry.timer("expenseCreationTime")
        .wrap(
            (Supplier<Expense>) () -> expenseService.create(expense)
        ).get();
}
```

4.2. Return to the terminal window, and then execute the `scripts/simulate-traffic.sh` script to generate some requests to the GET and POST endpoints.

```
[student@workstation monitor-metrics]$ ./scripts/simulate-traffic.sh
GET Response Code: 200
GET Response Code: 200
GET Response Code: 200
POST Response Code: 200
```

4.3. Verify the correctness of the code changes by retrieving the application metrics.

```
[student@workstation monitor-metrics]$ curl http://localhost:8080/q/metrics \
| grep expenseCreationTime
...output omitted...
# HELP expenseCreationTime_seconds
# TYPE expenseCreationTime_seconds summary
expenseCreationTime_seconds_count 1.0
expenseCreationTime_seconds_sum 4.00032617
# HELP expenseCreationTime_seconds_max
# TYPE expenseCreationTime_seconds_max gauge
expenseCreationTime_seconds_max 4.00032617
```

The application introduces random delays on the requests processing, and for that reason the output values might be different.

5. Add a metric that monitors the time since the last call to the GET endpoint. Use the `org.apache.commons.lang3.time.StopWatch` class to implement the logic.

5.1. Open the `ExpenseResource` class, and then create a `StopWatch` attribute.

```
package com.redhat.training;

...code omitted...

@Path("/expenses")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ExpenseResource {

    private final StopWatch stopWatch = StopWatch.createStarted();

    ...code omitted...
}
```

5.2. Update the `initMeters` method to initialize a gauge metric.

- Set `timeSinceLastGetExpenses` as the metric name.
- Define a `description` tag with the value `Time since the last call to GET /expenses`.
- Use the `StopWatch` instance as the state object.
- Use the `StopWatch#getTime` method as the value function.

```
@PostConstruct
public void initMeters() {
    registry.gauge(
        "timeSinceLastGetExpenses",
        Tags.of("description", "Time since the last call to GET /expenses"),
        stopWatch,
        StopWatch::getTime
    );
}
```

5.3. Update the GET endpoint to reset and start the time tracking for the gauge metric.

```
@GET
@Counted(value = "callsToGetExpenses")
public Set<Expense> list() {
    stopWatch.reset();
    stopWatch.start();

    return expenseService.list();
}
```

5.4. Return to the terminal window, and then execute the `scripts/simulate-traffic.sh` script to generate some requests to the GET and POST endpoints.

```
[student@workstation monitor-metrics]$ ./scripts/simulate-traffic.sh
GET Response Code: 200
GET Response Code: 200
GET Response Code: 200
POST Response Code: 200
```

5.5. Verify the correctness of the code changes by retrieving the application metrics.

```
[student@workstation monitor-metrics]$ curl http://localhost:8080/q/metrics \
| grep timeSinceLastGetExpenses
...output omitted...
# HELP timeSinceLastGetExpenses
# TYPE timeSinceLastGetExpenses gauge
timeSinceLastGetExpenses{description="Time ... GET /expenses",} 9995.0
```

The gauge value is in milliseconds and, because the application uses random delays, the output might be different.

6. Visualize the metrics data in Grafana.

6.1. Open the web browser and navigate to http://localhost:3000/dashboards.

6.2. Type `admin` as the username and `admin` as the password, and then click Log in.

6.3. Click Skip to omit changing the account password.

6.4. Click the expenses directory, and then click DO378 Expenses Dashboard.

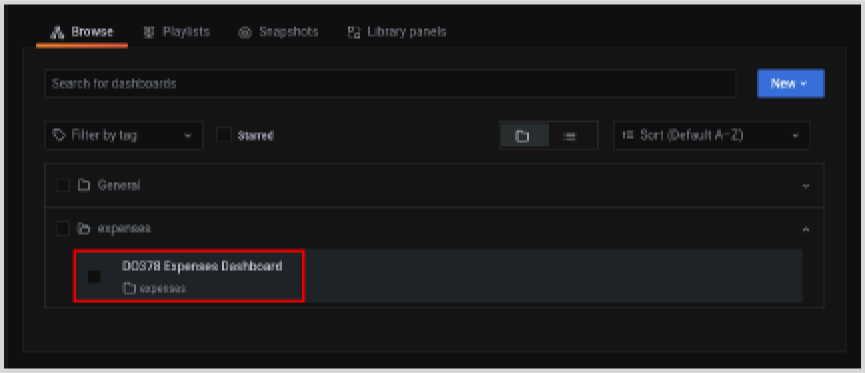


Figure 8.3: Available dashboards

6.5. Observe the dashboard that collects all the metrics added to the application. You can use the `scripts/simulate-traffic.sh` script to generate more metrics and visualize the dashboard update.

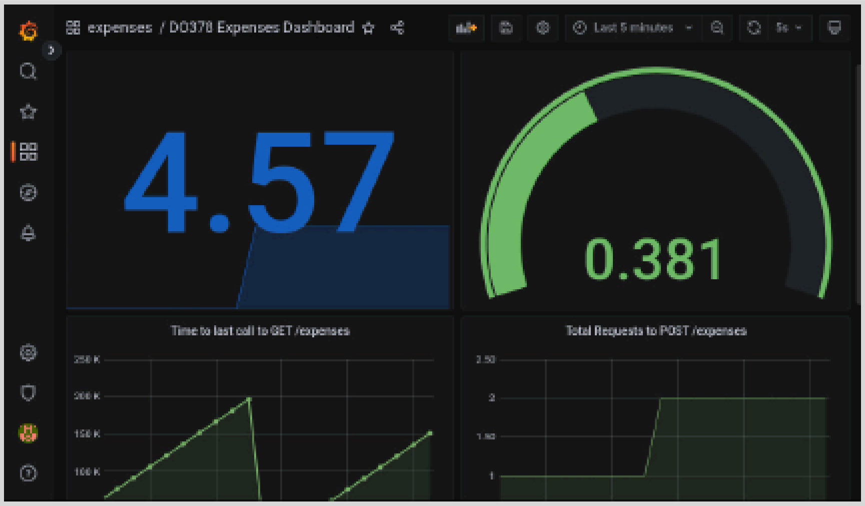

expenses / DO378 Expenses Dashboard
Last 5 minutes
5s
4.57
0.381
Time to last call to GET /expenses
250 K
200 K
150 K
100 K
Total Requests to POST /expenses
2.50
2
1.50
1

Figure 8.4: Application dashboard

6.6. Return to the terminal that runs the Quarkus application, and then press **q** to stop the application.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish monitor-metrics
```

This concludes the section.

enjoy!

Jose