

LAB 21: QUARKUS MONITOR METRICS

Autor: José Díaz

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

Abre el proyecto **19-monitor-metrics-start**.

1. Abre el proyecto en tu editor favorito.
2. Revisa la aplicación.
 - a. La clase `com.bcp.training.expense.Expense` implementa una representación básica de un `expense`.
 - b. La clase `com.bcp.training.expense.ExpenseResource` implementa una API CRUD que usa `com.bcp.training.expense.ExpenseService` para persistir la data.
 - c. La clase `com.bcp.training.expense.ExpenseService` es responsable por persistir y administrar instancias `Expense`.
3. Incluir la extensión Quarkus requerida para usar Micrometer con Prometheus.
 - a. **`mvn quarkus:add-extension -Dextensions=micrometer-registry-prometheus`**

```
[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...
```

- b. Usar el commando **`mvn quarkus:dev`** para iniciar la aplicación.

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

4. Agrega una métrica que cuenta el número de invocaciones de los endpoints GET y POST.
 - a. Abre la clase **`ExpenseResource`** e inyecta una instancia de **`MeterRegistry`**.

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

```

        private final Stopwatch stopWatch =
StopWatch.createStarted();

        @Inject
        public ExpenseService expenseService;

        @Inject
        public MeterRegistry registry;
...
    }

```

- b. Actualiza el endpoint **GET** para seguir el número de invocaciones del endpoint. Usa la anotación **@Counted**, y establece el nombre de la métrica a **callsToGetExpenses**.

```

@GET
@Counted(value = "callsToGetExpenses")
public Set<Expense> list() {

    return expenseService.list();
}

```

- c. Actualiza el endpoint **POST** para seguir el número de invocaciones del endpoint. Usa la instancia **MeterRegistry**, y establece el nombre de la métrica a **callsToPostExpenses**.

```

@POST
public Expense create(Expense expense) {

    registry.counter("callsToPostExpenses").increment(
    );
    return expenseService.create(expense);
}

```

- d. Abre una nueva terminal windows, navega a el directorio del proyecto, y luego ejecuta el **scripts/simulate-traffic.sh** (Linux o macosx) o **scrips/simulate-traffic.ps1** (windows) para generar algunos requests a los endpoints **GET** y **POST**.

`./scripts/simulate-traffic.sh`

```

[student@workstation ~]$ cd ~/D0378/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

```

e. Verifica que las metricas ya están disponibles invocando:

curl <http://localhost:8080/q/metrics> | grep Expenses_total

```
[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
```

5. Agrega una métrica que cuenta el tiempo consumido por el endpoint POST para persistir un expense.

a. Abre la clase ExpenseResource y luego actualiza el endpoint POST para seguir el tiempo consumido para persistir expenses. Crea un timer llamado expenseCreationTime, y wrapea la logica de persistencia para registrar el tiempo de ejecución.

```
@POST
public Expense create(Expense expense) {

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

b. Retorna a la terminal de windows, y luego ejecuta el scripts/simulate-traffic.sh (linux o macosx) o scripts/simulate-traffic.ps1 (windows) para generar algunos requests para los endpoints GET y POST.

./scripts/simulate-traffic.sh

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

c. Verifica que las métricas son registradas:

curl <http://localhost:8080/q/metrics> | grep expenseCreationTime

```
[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
```

La aplicación introduce algunos random delays en el procesamiento del request, y por esa razón los valores de salida pueden ser distintos.

6. Agrega una métrica que monitorea el tiempo desde la ultima llamada al endpoint **GET**. Usa el **org.apache.commons.lang3.time.StopWatch** para implementar la lógica.

a. Abre la clase **ExpenseResource** y luego crea un atributo **StopWatch**.

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

    private final StopWatch stopWatch =
StopWatch.createStarted();

    @Inject
    public ExpenseService expenseService;

    ...
}
```

b. Actualiza el método **initMeters** para inicializar una métrica gauge.

- Establece el nombre de la métrica a **timeSinceLastGetExpenses**.
- Define un tag de descriptoin con el valor **Time since the last call to GET /expenses**.
- Usa la instancia **StopWatch** como el objeto state.
- Usa el método **StopWatch#getTime** como el valor de la función.

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

```
}  
);  
}
```

- c. Actualiza el endpoint GET para resetear e iniciar el tiempo de seguimiento de la métrica gauge.

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

- d. Retorna a la terminal de windows, y luego ejecuta el scripts/simulate-traffic.sh (linux o macosx) o scripts/simulate-traffic.ps1 (windows) para generar algunos requests para los endpoints GET y POST.

./scripts/simulate-traffic.sh

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

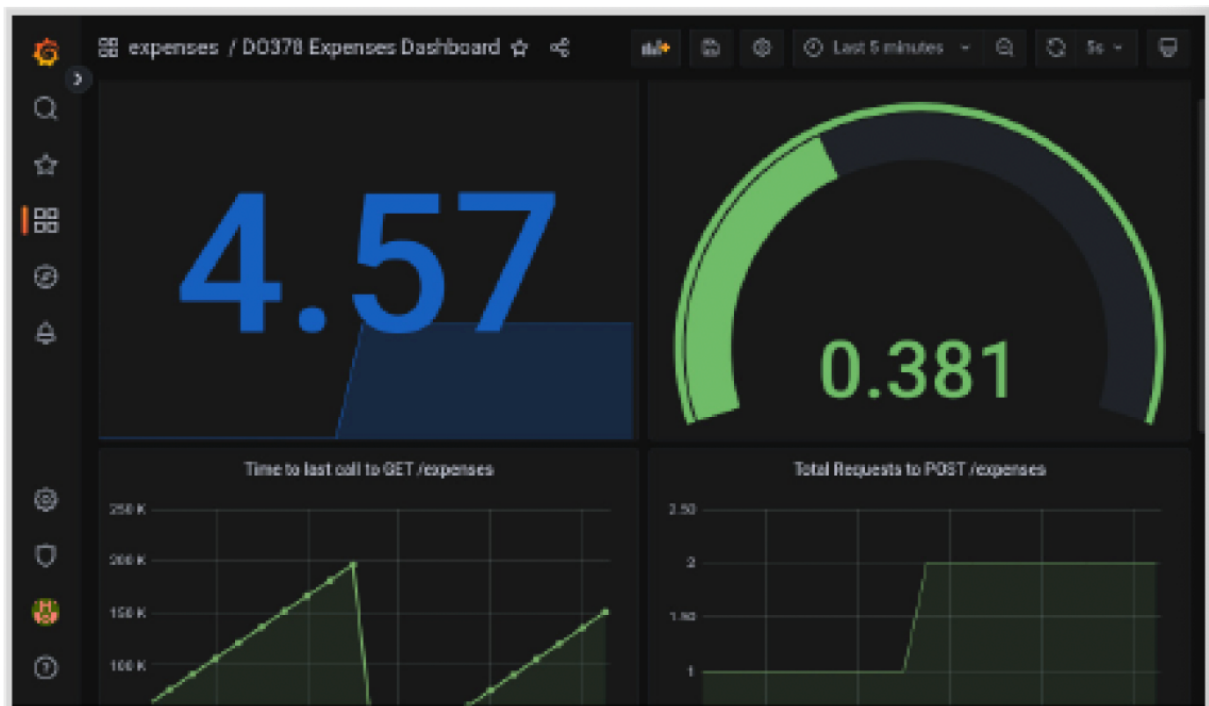
- e. Verifica que las metricas son recuperadas:

curl <http://localhost:8080/q/metrics> | grep timeSinceLastGetExpenses.

```
[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
```

El valor de gauge es en milisegundos, y como la aplicación usa random delays, la salida puede ser diferente.

7. Visualiza las métricas en Grafana.
 - a. Abre el navegador web y navega a <http://localhost:3000/dashboards>
 - b. Tipea admin como username y password como admin y luego clic en Log in.
 - c. Clic Skip para omitir cambiar el password de la cuenta.
 - d. Clic el directorio expenses y luego clic en DO378 Expenses Dashboard.



- e. Observa el dashboard que colecciona todas las métricas agregadas a la aplicación.
- f. Retorna a la terminal que ejecuta la aplicación Quarkus, y luego presiona **q** para detener la aplicación.

El laboratorio ha terminado.

Enjoy!

José