UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Sofrtware Architecture and Methodologies
&
Quantitative Evaluation of Stochastic Models

Quarkus Car Rental

Edoardo Sarri

7173337

Data

Indice

Elenco delle figure

Elenco delle tabelle

Listings

1 Introduzione

- nel libro si spiega tutto ma per la dev mode. raramente ci sono esempi per la distribuzione dell'app.

2 Car Rental Application

L'applicazione utilizzata durante l'intero progetto è stata *acme-car-rental*. Si tratta di un'applicazione a micro servizi sviluppata all'interno del libro *Quarkus in Action* [7]. Esso mostra le caratteristiche principali di Quarkus e sviluppa le varie funzionalità dell'applicazione capitolo per capitolo, rilasciando su GitHub la versione finale e completa alla fine.

L'applicazione permette di svolgere le attività di base, che ogni applicazione di noleggio auto dovrebbe implementare: ricerca di auto libere in un determinato intervallo di tempo; prenotazione di un auto definito tale intervallo; ricerca delle auto prenotate; aggiunta e rimozione di una'auto dall'inventario della auto disponibili.

Oltre a quanto specificato in questo report e a quanto si trova nella mia repo, la versione originale rilasciata conteneva anche degli elementi che sono stati eliminati visto che per il nostro scopo non erano necessari.

2.1 Architettura

Vediamo come prima cosa l'architettura dell'applicazione. I micro servizi che car-rental definisce, che la Figura 2.1 mette in relazione, sono cinque:

- **Billing-service**:
  Gestisce i pagamenti e le fatture. Viene chiamato da *reservation-service* quando un utente effettua una prenotazione e da *rental-service* quando un'auto viene noleggiata.
- **Inventory-service**:
  Gestisce l'inventario delle auto. Fornisce l'elenco delle auto disponibili e permette agli impiegati di aggiungere o rimuovere veicoli.
- **Rental-service**:
  Gestisce il processo di noleggio effettivo. Un impiegato può avviare un noleggio attraverso questo servizio, che a sua volta interagisce con il *billing-service*.
- **Reservation-service**:
  Gestisce le prenotazioni delle auto. Verifica la disponibilità dei veicoli tramite l'*inventory-service* e, in caso di prenotazione, avvia il processo di pagamento tramite il *billing-service*.
- **Users-service**:
  Fornisce una semplice interfaccia per la gestione delle prenotazioni.

Nel codice originale questo servizio si appoggiava a *Keycloak* per la gestione degli utenti. Per problemi di deployment e gestione, ho preferito rimuovere questa dipendenza; l'applicazione adesso permette a un solo utente, *guest*, di effettuare prenotazioni.

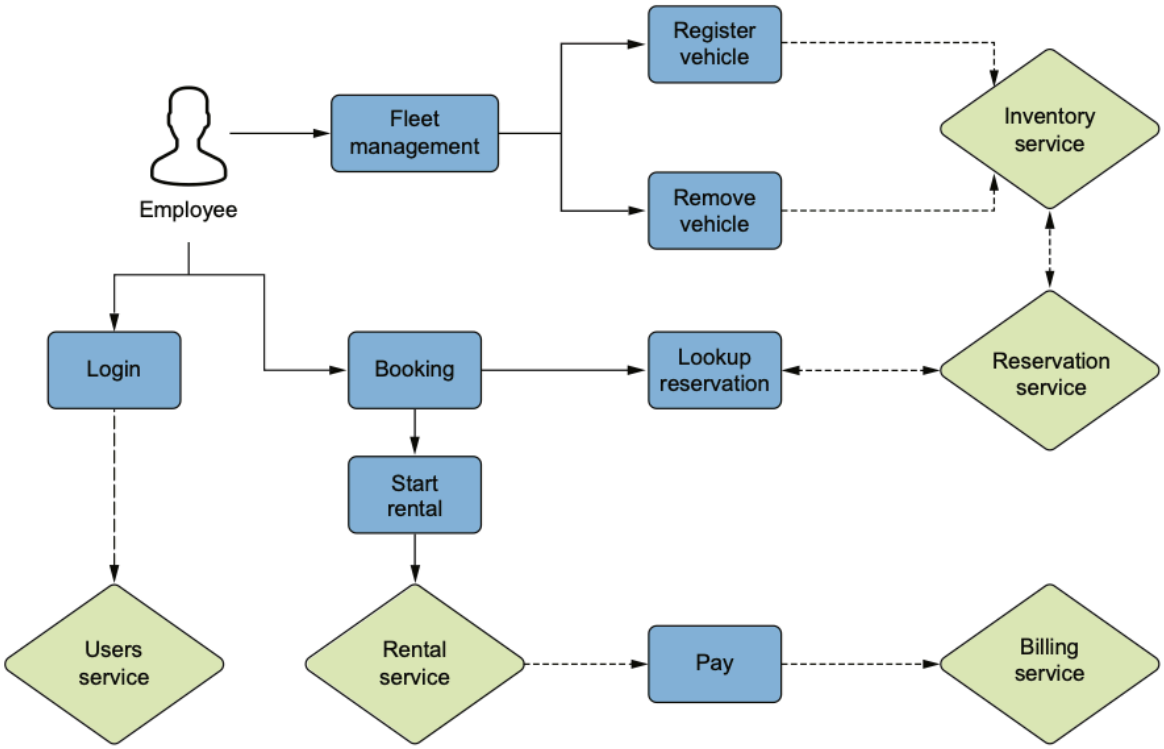

Figura 2.1: *car-rental* architecture [7].

2.2 Servizi Esterni

Oltre ai servizi business dell'applicazione, descritti nella Sezione 2.1, essa si basa anche su dei servizi esterni, non sviluppati dei produttori ma che devono essere configurati e da cui i servizi principali dipendono. Questi sono:

- **Kafka**:
  È una piattaforma di streaming di eventi distribuita, utilizzata per la comunicazione asincrona e mantenuta dalla Apache Software Foundation.
  Gli eventi in Kafka vengono aggiunti in fondo a una coda e sono i subscriber che si devono occupare di gestire i messaggi già letti e quelli da leggere.
  È ideale per streaming di dati di grandi dimensioni.

- **RabbitMQ**:
  È un sistema di message broker che implementa il protocollo AMQP (Advanced

Message Queuing Protocol), usato per la comunicazione asincrona.
Rispetto a Kafka, in RabbitMQ è il publisher che si occupa di distribuire i messaggi e che si assicura della loro ricezione.
È ideale per creare code di lavoro, dove ogni messaggio è un compito da eseguire.

- **MongoDB**:
  È un database NoSQL orientato ai documenti, che memorizza i dati in file simili ai JSON (detti BSON).
  È ideale per quelle applicazioni i cui requisiti sui dati non sono già stati ben definiti e che vogliono scalare orizzontalmente.

- **MySQL**:
  È un database relazionale open-source.
  È ideale per la memorizzazione di dati strutturati, per la rapidità e la semplicità d'uso.

- **PostgreSQL**:
  È un database relazionale object-oriented open-source.
  È più conforme agli standard SQL e offre un set di funzionalità per la gestione di query complesse più ricco rispetto a MySQL.

2.3 Comunicazione

Vediamo infine come i micro servizi, sia di business che esterni, comunicano tra loro. La Figura 2.2 illustra i vari flussi di comunicazione; da notare che *inventory-CLI* è uno di questi servizi rimossi rispetto alla versione rilascaita da Quarkus in Action, vista la sua inutilità nel nostro contesto.

- **REST**:
  È lo stile architetturale di comunicazione più utilizzato nel web e si basa sul protocollo HTTP. È ideale quando si cerca una comunicazione sincrona tra servizi, dove un client invia una richiesta e attende una risposta.
  Nell'applicazione viene usato ad esempio da *reservation-service* per interrogare *inventory-service* sulla disponibilità dei veicoli.

- **GraphQL**:
  È un'alternativa più flessibile a REST, basata sempre sul principio clinet-server, che consente ai client di richiedere esattamente i dati di cui hanno bisogno; in questo modo il client non deve filtrare tutto quello che riceve.

Nell'applicazione viene usato da *users-service* per aggregare dati da diversi servizi con una singola richiesta, semplificando l'interazione dal lato client.

- **gRPC**:
  Protocollo di comunicazione che si basa su HTTP/2, utilizza il formato binario, che permette di inviare più richieste contemporaneamente sullo stesso canale e di ricevere le relative risposte contemporaneamente. È ottimo nelle situazioni in cui si richiede una latenza minima con una comunicazione molto efficiente.
  In questo progetto, *inventory-service* espone un endpoint gRPC che viene utilizzato da *reservation-service* per ottenere informazioni sulle auto in modo performante.

- **Kafka**:
  Nell'applicazione viene usata per notificare eventi come l'inizio o la fine di un noleggio tra il *rental-service* e il *billing-service*. Questo garantisce una comunicazione resiliente anche in caso di fallimenti temporanei di uno dei due.

- **RabbitMQ**:
  Viene usato da *reservation-service* per comunicare la creazione di una nuova prenotazione a *billing-service*, che si occuperà di creare la fattura.

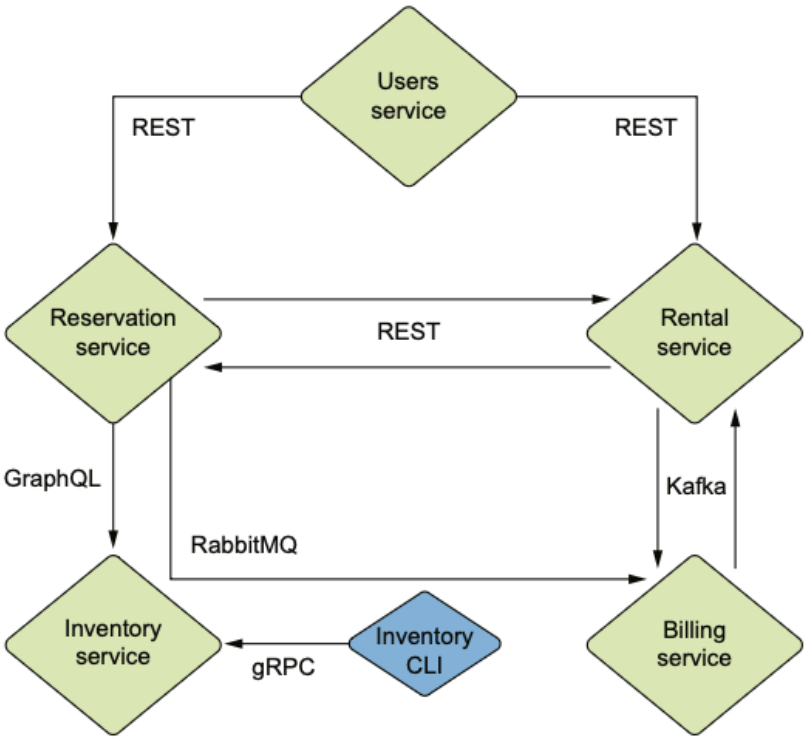

Figura 2.2: *car-rental* communication [7].

3 Deployment

In questo capitolo descriviamo il processo di deployment, più complesso di quello che ci aspettavamo. Nonostante infatti l'applicazione fosse già sviluppata in ogni suo aspetto, essa era predisposta per essere rilasciata su OpenShift e usare Quay.io come registry per le immagini.

Per questi motivi il file di configurazione Quarkus *application.properties* sono stati complettamente rivisti e adattati alla nostra configurazione, che come abbiamo detto nel Capitolo 1 comprende Minikube come ambiente di deployment e Docker come runtime per i container.

Per automattizzare il deployment, nella root directory del progetto è presente una cartella *exec*, all'interno della quale è presente il file *deployment.sh*. Se eseguito a partire dalla root directory e con il Docker demon in esecuzione, allora verrà creato un cluster Minikube con tutti i micro servizi e le loro dipendenze. Il corretto funzionamento di questo script è garantito su MacOS con ARM64.

All'interno dello script viene usato, per ogni micro servizio, il comando `quarkus build`. Questo permette di costruire l'immagine Docker in automatico, senza scrivere a mano il manifesto Kubernetes; l'unico requisito è l'utilizzo dell'estensione *quarkus-container-image-docker*. Inoltre, cosa che in Minikube non è permesso, sarebbe consentito anche eseguire il deployment della nuova immagine appena costruita nell'ambiente di Deployment; questo può essere fatto tramite la configurazione *quarkus.container-image.push=true*

3.1 Environment

L'ambiente per il rilascio dell'applicazione è stato Minikube, un ambiente locale di sviluppo basato su Kubernetes. In questo modo è stato possibile testare il deployment in un ambiente simile a quello di produzione senza la necessità di risorse hardware dedicate. Come abbiamo detto, nonostante l'applicazione rilasciata su GitHub sia completa e configurata per un tipo di deployment, ci sono stati vari aspetti da considerare e che hanno richiesto attenzione e tempo:

- **OpenShift**
  L'applicazione era stata pensata per essere distributa su OpenShift, la piattaforma per container basata su Kubernetes e sviluppata da Red Hat. Nonostante essa sia appunto Kubernetes-based, in Quarkus si richiede di specificare quale ambiente di deployment si intente utilizzare all'interno del file di configurazio-

ne *application.properties*.
Un esempio di configurazione corretta per Kubernetes, usando Docker come runtime di container, è quella mostrata nel Listing 3.1: questo è il contenuto del file di configurazione Quarkus per *inventory-service*, ma generalizzando è lo stesso usato in tutti gli altri microservizi. Oltre ai comandi che sono naturalmente interpretabili, possiamo specificare alcuni concetti più complessi: `quarkus.container-image.build=true` permette di costruire l'immagine in automatico quando eseguiamo il comando `quarkus build`; `quarkus.kubernetes.service-type=NodePort` istruisce Kubernetes a creare un Service di tipo *NodePort* (di default è *ClusterIP*) e quindi accessibile anche dall'esterno del cluster (meno corretto in produzione di *LoadBalancer*, ma giusto per test locali); `quarkus.kubernetes.image-pull-policy=Never` istruisce Kubernetes a far fallire l'avvio del pod se l'immagine Docker su cui si basa non è presente localmente (a differenza di *Always* che la scarica sempre da un registry).

- **Registry**
  Una volta costruita l'immagine docker, il file di configurazione era progettato per eseguire il push su un registry, Quay.io, registry di Red Hat che fa parte dello stesso ecosistema di OpenShift. Siccome il nostro ambiente per il deployment era Minikube, sono state fatte delle modifiche anche in questo senso in modo da rilasciare l'immagine del microservizio all'interno di Minikube stesso.
  In questo scenario è fondamentale la configurazione `quarkus.container-image.push=false` all'interno del Listing 3.1.

- **Riferimenti**
  Usando un ambiente di produzione diversi (OpenShift vs Minikube) rispetto a quello per cui l'applicazione era stata pensata, si sono dovuti cambiare gli URL con cui un microservizio identificava le proprie dipendenze. Un esempio di quello che è stato fatto si trova nel Listing 3.2, relativo al micro servizio *rental-service*: i riferimenti per l'ambiente di sviluppo sono stati lasciati invariati, mentre sono state configurate correttamente le variabili d'ambiente all'interno del container in modo che sia presente il riferimento alla dipendenza usata.

```
# container
quarkus.container-image.build=true
quarkus.container-image.push=false
quarkus.container-image.group=edoardosarri
quarkus.container-image.name=inventory-service
quarkus.container-image.tag=1.1
# kubernetes
quarkus.kubernetes.name=inventory-service
```

```
quarkus.kubernetes.deployment-target=kubernetes
quarkus.kubernetes.service-type=NodePort
quarkus.kubernetes.image-pull-policy=Never
```

Listing 3.1: Kubernetes and Docker Configuration

```
quarkus.rest-client.reservation.url=http://localhost:8081
quarkus.kubernetes.env.vars.quarkus-rest-client-reservation-
    url=http://reservation-service
```

Listing 3.2: Reference in different environments

3.2 Servizi Esterni

Come abbiamo già detto nella Sezione 2.2, i micro servizi business dell'applicazione utilizzano al loro interno altri servizi ausiliari. Questi devono eseguire all'interno dello stesso cluster Minikube e quindi devono essere in un qualche modo rilasciati.

La prima soluzione esplorata è stata farsi scrivere da un LLM il manifesto Kubernetes per l'ambiente Minikube. Nonostante questa fosse una soluzione funzionante, mi sembrava chiaro che ci potesse essere un'alternativa off-the-shelf che mi permettesse di non dover fare configrazioni manuali; l'alternativa adottata è stata Helm.

L'unico servizio esterno per cui è stato generato (da un LLM, in particolare Gemini) il manifesto invece che usare un chart è stato MongoDB. Il problema, che non sono riuscito a risolvere, è stato quella della compatibilità tra la versione di MongoDB e quella di Minikube.

3.2.1 Helm

Helm è un gestore di pacchetti Kubernetes che semplifica il ciclo di vita delle applicazioni all'interno di Kubernetes.

Tramite i Chart possiamo definire, installare e aggiornare anche le applicazioni Kubernetes più complesse. Essi permettono infatti di non generare manifesti complessi e ridondanti a mano, ma di utilizzare una struttura predefinita e riutilizzabile. Nello stesso modo in cui esiste un database di immagini Docker, esiste un repository di Chart Helm, ArtifactHUB; oltre a fornirci molte applicazione, il repository ci fornisce tutte le istruzioni necessarie per configurazioni avanzate [1].

3.2.2 Deployment con Helm

Per capire come Helm semplifica il deployment di servizi terzi da cui un nostro micro servizio dipende, prendiamo come esempio l'installazione del database MySQL, ne-

cessario per *inventory-service*. Tramite un comando, seguito da pochi parametri, come si vede nel Listing 3.3, viene rilasciato un pod MySQL nel cluster Minikube.

```
helm install mysql-inventory bitnami/mysql \
    --set auth.rootPassword=root-pass \
    --set auth.database=mysql-inventory \
    --set auth.username=user \
    --set auth.password=pass
```

Listing 3.3: MySQL Helm chart

3.3 Health

Durante la fase di deployment, e in particolare durante il deployment automazzato con *deployment.sh*, è stato notato che il servizio *reservation-service* non funzionava correttamente. In particolare il pod partiva, andava in stato *running* e *ready*, ma non si riusciva a connetere al suo database PostgreSQL. Dopo numerose e lunghe investigazioni, analizzando i log dei pod, il problema è stato risolto eliminando manualmente il pod: Kubernetes riavvia una nuova istanza e i problemi di connesione vengono risolti.

Questo ci può portare ad affermare che, per qualche motivo, il pod necessità che il database PostgreSQL sia disponibile e pronto per accettare connessioni prima di avviarsi correttamente.

3.3.1 MicroProfile Health

Per evitare di riavviare il pod manualmente o di introdurre nello script del deployment una *wait* è stato deciso di forzare il pod relativo a *reservation-service* ad attendere la disponibilità del database prima di avvarsi.

Questo viene fatto tramite la specifica Health di MicroProfile implementata poi da SmallRye; essa estende le specifiche di Jakarta per adattare quest'ultima all'architettura dei micro servizi. Permette di esporre lo stato di un'applicazione, cioè un valore binario (a differenza delle metriche che sono valori numerici qualunque), che viene solitamente controllato dall'orchestratore che gestisce il ciclo di vita delle applicazione (e.g., Kubernetes) per decidere se un'applicazione è sana e può essere usata.

I tre maggiori controlli di salute che MicroProfile mette a disposizione sono *Startup*, *Readiness* e *Liveness* [3]: la prima verifica che l'applicazione sia stata avviata correttamente; la seconda controlla se l'applicazione è pronta a ricevere traffico; la terza monitora lo stato dell'applicazione durante il suo funzionamento. Se uno di questi controlli fallisce solitamente il pod viene riavviato.

3.3.2 Implementazione

Considerando che il nostro problema viene risolto facendo ripartire il pod del micro servizio una volta che il database è disponibile, l'Health Check utile è lo *StartUp Check*. È stata definita la classe DatabaseConnectionHealthCheck.java, annotata con l'annotazione `org.eclipse.microprofile.health.Startup`, che implementa una `SELECT` sul database: se il comando ha successo allora il check termina con successo e il pod è in esecuzione correttamente, altrimenti il pod viene fatto ripartire.

Nella classe la connessione viene gestita da un oggetto di tipo `AgroalDataSource`[5]: *DataSource* è un'interfaccia che rappresenta una connessione a un database standard in Java (JDBC); *Agroal* è un'implementazione di questa interfaccia ed è quella che funziona meglio con Quarkus.

Quarkus permette con molta facilità di gestire le metriche di Health con le apposite estensioni [6]. Basta infatti aggiungere *quarkus-smallrye-health*, *quarkus-agroal* e *quarkus-jdbc-postgresql* e il tutto funiona completamente. L'unica cosa agiguntiva a cui dobbiamo pensare è a fornire l'url a cui l'oggetto `AgroalDataSource` deve connettersi; quest è facilmente configurabile nell'*application.properties*, come si vede dal Listing 3.4. Sono state anche aggiunte alcune configurazioni [4] per gestire dopo quanti tentativi falliti riavviare il pod.

```
# health
quarkus.kubernetes.liveness-probe.failure-threshold=1
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/
    reservation
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql://postgresql
    -reservation:5432/reservation
```

Listing 3.4: JDBC configuration.

4 Tracing

In applicazioni con migliaia di micro servizi, seguire la cascata di chiamate o valutare in che punto si è verificato un errore è molto complesso. Tramite il tracing non solo possiamo capire quale micro servizio viene chiamato e da chi, ma possiamo anche analizzare i tempi con cui ogni richiesta è stata servita, sia E2E che all'interno del singolo servizio.

Il funzionamento solitamente è abbastanza semplice: durante le varie chiamate viene trasmesso anche un ID (univoco) della traccia; il trasporto di questo ID avviene tramite una qualche funzionalità del protocollo di trasporto, che in HTTP è il relativo header.

4.1 OpenTelemetry

OpenTelemetry, noto anche come OTel, è un insieme di tool e API che permette di collezionare ed esportare le telemetrie di un'applicazione a micro servizi. Oltre alle telemetrie in realtà possono anche essere gestite metriche e logs, ma in questo campo non è stabile e si preferisce usare altro (e.g., MicroMeter).

Fornisce un protocollo, detto OTLP, che permette di esportare le telemetrie dalle applicazioni verso il tool OpenTelemetry Collector. A quest'ultimo si possono collegare vari back end (e.g., Jaeger) per la loro visualizzazione.

Usare il Collector non è necessario: si possono inviare i dati direttamente al backend: in piccole applicazioni è forse una scelta migliore per la facilità di manutenzione e configurazione; se l'applicazione deve scalare allora si hanno numero vantaggi usando il Collector, come il filtraggio, il batching e la riprova in caso di un qualche fallimento.

Per utilizzare OTel nei vari micro servizi della nostra applicazione si deve semplicemente aggiungere l'estensione *quarkus-opentelemetry*. In questo modo Quarkus inizia a collezionare telemetrie in automatico.

4.2 Jaeger

Come abbiamo detto Jaeger è un backend per la visualizzazione della cascate di chiamate all'interno di un applicazione a micro servizi.

Per permettere il suo funzionamento si deve aggiungere delle configurazioni nel file *application.properties*. Queste configurazioni sono abbastanza standard e non dipendono molto dal microservizio specifico, se non per il nome che vogliamo visualizzare

nella UI. Un esempio è quanto configurato per *users-service* all'interno del Listing 4.1. L'unica configurazione da spiegare è forte `quarkus.otel.traces.sampler=always_on`: essa permette di non fare nessun campionamento delle chiamate, ma di considerarle e visualizzarle tutte; in questo modo non si rischia di interrompere una sequenza di chiamate.

```
# jaeger
quarkus.kubernetes.env.vars.otel-service-name=users-service
quarkus.otel.resource.attributes=service.name=users-service
quarkus.kubernetes.env.vars.otel-exporter-otlp-endpoint=http
    ://jaeger-collector:4317
quarkus.otel.exporter.otlp.endpoint=http://jaeger-collector
    :4317
quarkus.otel.traces.sampler=always_on
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{
    traceId}, parentId=%X{parentId}, spanId=%X{spanId}, sampled
    =%X{sampled} [%c{2.}] (%t) %s%e%n
```

Listing 4.1: Jaeger configuration for *users-service*

4.3 Jaeger deployment

Come per i servizi terzi elencati nella Sezione 2.2, anche il backend Jaeger deve essere rilasciato nello stesso cluster Minikube di tutta l'applicazione.

Anche in questo caso è stato utilizzato il chart Helm Jaeger [2]. Si può vedere il comando nel Listing 4.2.

```
helm install jaeger jaegertracing/jaeger \
    --set allInOne.enabled=true \
    --set agent.enabled=false \
    --set collector.enabled=false \
    --set query.enabled=false \
    --set provisionDataStore.cassandra=false \
    --set storage.type=memory
```

Listing 4.2: Install Jaeger chart for *users-service*

5 Load Generator

5.1 K6 Operator

Il generatore di carico K6 può essere installato localmente su una macchina Linux, MacOS o Windows, ma ha anche il grande vantaggio che può esere all'interno di un'infrestruttura cloud. Nel nostro progetto questo è utile e sensato se distruiamo K6 all'interno del cluster Minikube dove è in esecuzione la nostra applicazione; in questo contesto l'istanza K6 prende il nome di K6 Operator.

Per distribuire K6 in un cluster Kubernetes ci sono tre modi, ma quello che abbiamo usato è, come in precedenza, Helm. La documentazione di K6 fornisce istruzioni molto precise su come gestire la configrazione e l'installazione. Nel nostro caso è stato parametrizzato tutto nel file *values.yaml* che possiamo vedere nel Listing 5.1.

```
targetUrl: "http://my-service:80/api/endpoint"
vus: 10
duration: "10s"
script: |
    import http from 'k6/http';
    import { sleep } from 'k6';
    export let options = {
        vus: __ENV.VUS,
        duration: __ENV.DURATION,
    };
    export default function () {
        http.get(__ENV.TARGET_URL);
        sleep(1);
    }
```

Listing 5.1: K6 *values.yaml* file

Bibliografia

[1] *Bitnami Helm Charts for MySQL*. URL: https://artifacthub.io/packages/helm/bitnami/mysql (visitato il giorno 03/09/2025).

[2] JaegerTracing. *Jaeger Helm chart*. URL: https://artifacthub.io/packages/helm/jaegertracing/jaeger (visitato il giorno 03/09/2025).

[3] Kubernetes Authors. *Liveness, Readiness and Startup Probes*. 2025. URL: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/ (visitato il giorno 03/09/2025).

[4] *Quarkus - All configuration options*. URL: https://quarkus.io/guides/all-config (visitato il giorno 04/09/2025).

[5] Quarkus Team. *Configure data sources in Quarkus*. URL: https://quarkus.io/guides/datasource (visitato il giorno 03/09/2025).

[6] Quarkus Team. *SmallRye Health - Quarkus*. URL: https://quarkus.io/guides/smallrye-health (visitato il giorno 03/09/2025).

[7] Martin Štefanko e Jan Martiška. *Quarkus in Action*. https://www.manning.com/books/quarkus-in-action. Data ultima visualizzazione: 2026-07-30. Manning Publications, 2025.

[8] Burr Sutter et al. *Kubernetes Native Microservices*. O'Reilly Media, 2020.