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

Settembre 2025

Indice

1 Introduzione **6**

2 Car Rental Application **7**

2.1 Architettura 7
2.2 Servizi Esterni 8
2.3 Comunicazione 9

3 Deployment **11**

3.1 Environment 11
3.2 Servizi Esterni 13
3.2.1 Helm 13
3.2.2 Deployment con Helm 14
3.3 Health 14
3.3.1 MicroProfile Health 14
3.3.2 Implementazione 15

4 Tracing and Metrics **16**

4.1 Tracing 16
4.1.1 OpenTelemetry 16
4.1.2 Collector 16
4.1.3 Jaeger 17
4.1.4 OTel workflow 18
4.2 Metrics 18
4.2.1 Prometheus 19
4.2.2 Grafana 19

5 Workflow **21**

5.1 Blocchi 21
5.1.1 XOR 22
5.1.2 AND 22
5.2 Implementazione 22
5.2.1 XOR 22
5.2.2 AND 23
5.3 Nuovo workflow 24

6 Load Generator **26**

6.1 K6 Operator 26

Elenco delle figure

2.1 *car-rental* users use cases [9]. 7
2.2 *car-rental* communication [9]. 10

4.1 OpenTelemetry workflow. 18
4.2 Prometheus-Grafana Flow. 19

5.1 Workflow iniziale di Car-Rental. 21
5.2 Nuovo workflow di `reserve` mostrato da Jaeger. 24
5.3 Sequence diagram dei blocchi XOR e AND. 25

Elenco delle tabelle

Listings

3.1 Kubernetes and Docker Configuration 12
3.2 Reference in *users-service* 13
3.3 MySQL Helm chart 14
3.4 JDBC configuration. 15

4.1 Jaeger configuration for *users-service* 17
4.2 Install Jaeger chart for *users-service* 17
4.3 Prometheus configuration for *users-service* 19

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* [9]. 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.

Lato utente l'applicazione permette, come si vede in Figura 2.1, 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.

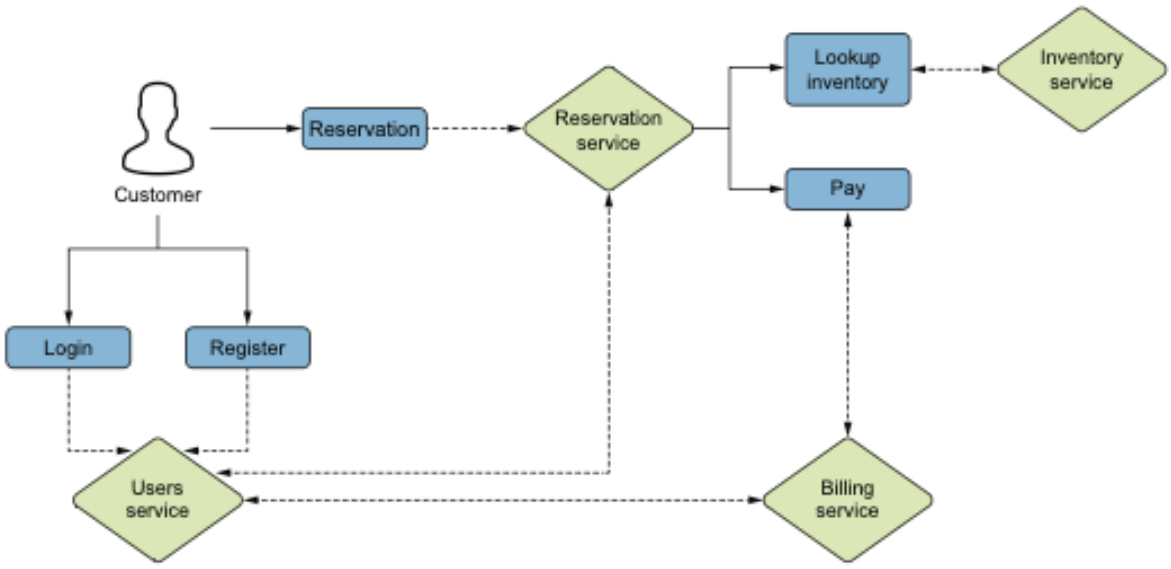

Figura 2.1: *car-rental* users use cases [9].

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 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.

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.

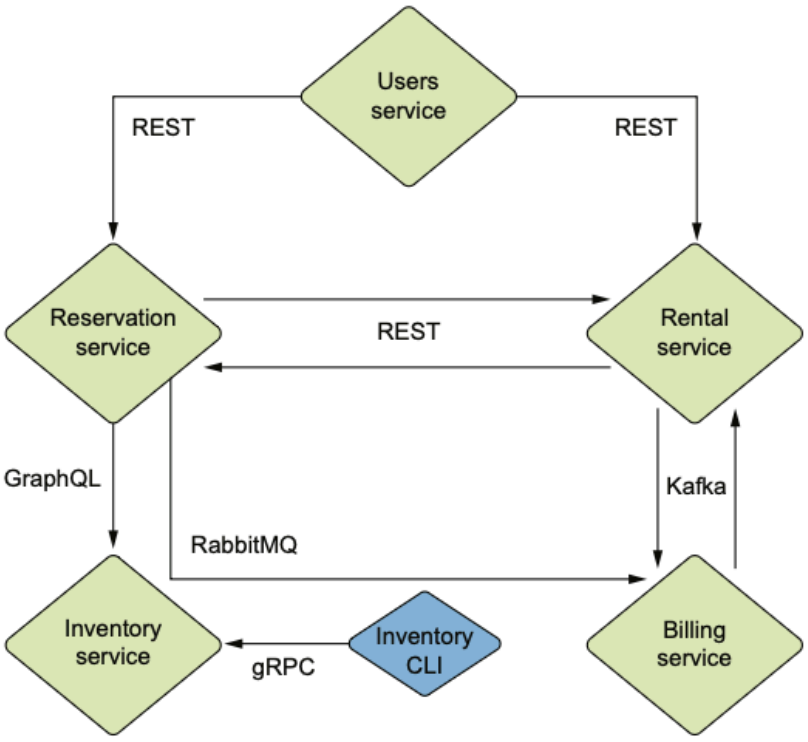

Figura 2.2: *car-rental* communication [9].

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 *users-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=2.0
# 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

```
# reference
%dev.quarkus.rest-client.reservations.url=http://localhost
    :8081
%prod.quarkus.rest-client.reservations.url=http://reservation-
    service
```

Listing 3.2: Reference in *users-service*

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, necessario 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* [5]: 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`[7]: *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 [8]. 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; questo è facilmente configurabile nell'*application.properties*, come si vede dal Listing 3.4. È stata aggiunta anche la configurazione [6] per gestire dopo quanti tentativi falliti riavviare il pod.

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

Listing 3.4: JDBC configuration.

4 Tracing and Metrics

4.1 Tracing

In applicazioni con migliaia di micro servizi, seguire il workflow, cioè la cascata di chiamate, o valutare in che punto si è verificato un errore può essere 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. Quando la chiamata ritorna allora vengono unite le chiamate con lo stesso ID e viene generata la traccia.

4.1.1 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 microservizi. 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).

La vera forza di OpenTelemetry però non è tanto la raccolta di tracce, quando la definizione di OTLP, un protocollo stardard che permette di esportare le telemetrie verso vari e diversi backend (e.g., Jaeger per la visualizzazione).

Per permettere il funzionamento di Opentelemtry con un qualche backend ci sono due possibili:

- Direttamente: OpenTelemetry una volta raccolte le tracce le invia al backend in questione. È una strategia più semplice, ma non scala quando i backend aumentano e se dobbiamo fare un filtraggio delle tracce.
- Collector: I dati vanno all'Otel-Collector, che è il responsabile dell'invio e di operazioni intermedie (e.g., filtraggio, il batching e la gestione dei fallimenti). È la strategia più complesso, però in applicazioni che devono scalare è necessario.

4.1.2 Collector

Nella nostra applicazione è stato usato il collector perché si voleva inviare le tracce al backend di Jaeger e anche salvarle in un file per eseguire ulteriori analisi in un secondo

momento.

La raccolta delle tracce con Quarkus diventa estremamente semplice: aggiungendo l'estensione *quarkus-opentelemetry* il fraework instrumenta il codice e in automatico le telemetrie vengono collezionate. Una volta che le tracce sono raccolte vengono inviate al collector, dopo la configurazione ne Listing 4.1 (relativo a `users-service`), dove l'unica configurazione da spiegare è forse `quarkus.otel.traces.sampler=always_on`: essa permette di non fare nessun campionamento delle chiamate, ma di considerarle tutte.

Il collector è un pod all'interno dell'ambiente Minikube che è stato deployato tramite una configurazione helm. Quando riceve le tracce le invia a due destinazioni: il pod di Jaeger per la visualizzazione e il file */traces/traces.json* che si trova nel PVC (Persinstant Volume Claim) *pvc-traces* sempre all'interno del cluster. Quando eseguiremo l'analisi dei dati allora dovremmo eseguire l'analyser in un pod a cui è collegato questo PVC; in questo modo lo script esegue sui propri dati locali.

```
# jaeger
quarkus.otel.service.name=users-service
quarkus.otel.exporter.otlp.traces.endpoint=http://otel-
    collector:4317
quarkus.otel.traces.sampler=always_on
```

Listing 4.1: Jaeger configuration for *users-service*

4.1.3 Jaeger

Come abbiamo detto Jaeger è un backend per la visualizzazione del work di una chiamata in un'applicazione a microservizi.

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 [4]. 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*

4.1.4 OTel workflow

Fino a questo memomento in qusta sezione abbiamo parlato dei componenti in modo separato; vediamo adesso di mettere tutto insieme per capire come i vari pod interagiscono nella nostra applicazione.

Analiziamo il flusso mostrato in Figura 4.1:

- Viene chiamata una qualche API dell'applicaizone.
- OpenTelemtry raccoglio le telemetri e invia le tracce al collector, che eventualmente fa quale operazione.
- Il collector invia le tracce ai vari backend. Nel nostro caso Jaeger per la visualizzazione e il file system per l'analisi successiva.

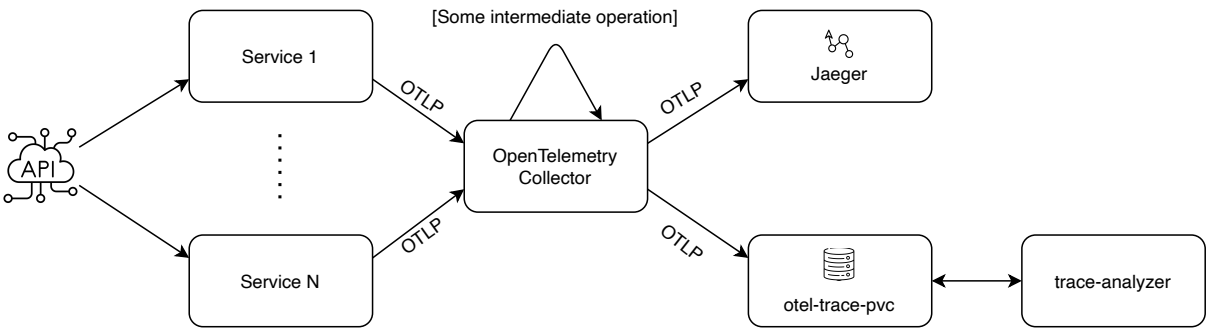

Figura 4.1: OpenTelemetry workflow.

4.2 Metrics

Oltre a tracciare la sequenza delle varie chiamate, come abbiamo mostrato nel Capitolo 4.1, in un applicazione a micro servizi è importante anche essere in grado di catturare metriche, valori reali che espimono una qualche informazione.

In generale possiamo esporre metriche framework o business: le prime sono stardard fornite da un qualche framework e sono generali e valide in un qualunque ambito (e.g., carico della cpu, uso della memoria o richieste per secondo); le seconde sono metriche specifiche per il dominio dell'applicazione.

La specifica MicroProfile per le metriche si chiama Metrics. Nonostante questo solitamente viene usata MicroMeter, un'implementazione che non segue la specifica MicroProfile. Quarkus permette in ogni caso di usare sia implemetazioni della specifica Metrics che MicroMeter, nonostante la seconda sia quella preferibile.

4.2.1 Prometheus

Promethues è un sistama open-source per il monitoraggio e allerta di applicazioni. Svolge tre compiti pricipali: raccoglie metriche, le memorizza e fornisce un endpoint utile per fare querying.

In Quarkus utilizzare MicroMetrics e Prometheus insieme è molto semplice: dobbiamo solo aggiungere l'estensione *quarkus-micrometer-registry-prometheus*: senza strumentalizzare il codice, MicroMeter inizia a raccogliere mentriche standard (e.g., utilizzo CPU e della memoria) nel formato richiesto da Prometheus, mentre quest'ultimo periodicamente interroga gli endpoint creati, raccogliendo e salvando tali metriche.

Dobbiamo a questo punto definire un modo per dire a Prometheus di raccogliere le metriche che MicroMetrics definisce. In Kubernetes questo metodo è detto *ServiceMonitor* e può essere semplicemente implementato nel file di configurazione, come vediamo nel Listing 4.3

```
# prometheus and grafana
quarkus.kubernetes.prometheus.generate-service-monitor=true
quarkus.kubernetes.labels.release=prometheus
```

Listing 4.3: Prometheus configuration for *users-service*

Infine serve installare nel cluster Minikube un container in cui esegue il server Prometheus. Questo è stato fatto, come nel resto del progetto, tramite Helm e il suo chart c. Questo non solo installa Promuethus, ma è un singolo pacchetto che comprende anche Grafana.

4.2.2 Grafana

Il sistema di query di Promethues permette di visualizzaare le metriche raccolte in forma testuale tramite terminale. Quando le metriche sono complesse, ma anche in generale, questo può non essere molto comodo.

Grafana è un backend che permette di visualizzare in modo grafico le metriche raccolte da Promethues, come si può vedere nella Figura 4.2

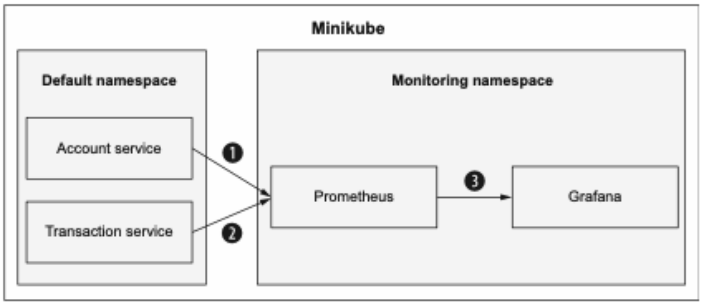

Figura 4.2: Prometheus-Grafana Flow.

Come dette nella Sezione 4.2.1 l'installazione all'interno del cluster Minikube è stata fatta con il chart di Helm *kube-prometheus-stack* [3], che comprende sia Prometheus che Grafana.

5 Workflow

Nei Capitoli 3 e 4 abbiamo visto come l'applicazione car rental, descritta nel Capitolo 2, viene rilasciata su Minikube e come possiamo analizzare metriche e workflow. In questo capitolo ci concentriamo invece sul workflow, cioè su come i microservizi si chiamano durante l'esecuzione di una funzionalità; in particolare la funzionalità che andremo ad esaminare sarà la prenotazione di una macchina.

Prima di qualunque modifica, come si nota dalla Figura 5.1, *Car-Rental* aveva un workflow esclusivamente sequenziale: quando un utente invocava una qualunque operazione, e in particolare `reserve`, tramite *users-service*, la sequenza di chiamate agli altri microservizio era puramente sincrona e determinstica, cioè era sempre la stessa data l'operazione chiamata; ovviamente i tempi di esecuzione variano di chiamata in chiamata.

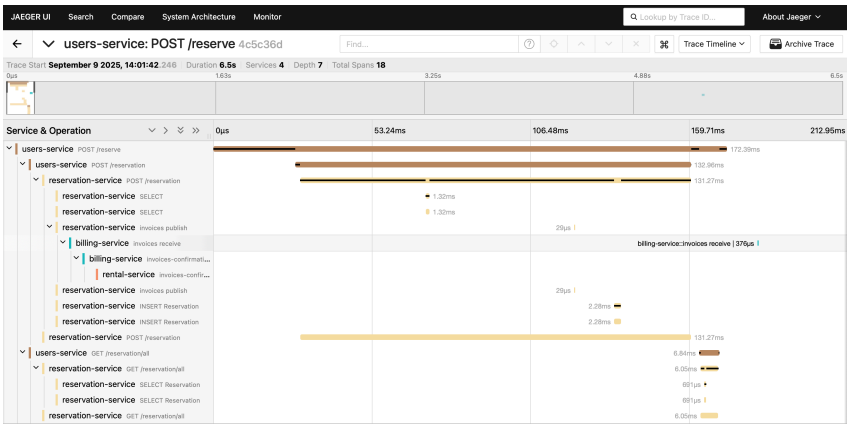

Figura 5.1: Workflow iniziale di Car-Rental.

Il nostro obiettivo è, facendo riferimento sempre all'operazione `reserve`, complicare questo workflow. Facendo riferimento alla terminogia usata nell'articolo del tool Eulero [2], che useremo più avanti, l'applicazione prima della modifica combina solamente blocchi di tipo sequenziale (`SEQ`), mentre a noi interessa aggiungere degli blocchi `XOR` e blocchi `AND`, che sono descritti nella Sezione 5.1 e implementati nella Sezione 5.2.

5.1 Blocchi

Vediamo come prima cosa quali blocchi sono stati aggiunti e in che punto dell'attuale workflow sono stati eseguiti.

5.1.1 XOR

I blocchi XOR rappresentano scelte casuali esclusive definite da una certa probabilità. Questo vuol dire che se abbiamo *n* possibili strade che il flusso può prendere, ne sarà scelta una e una soltanto una in base alla probabilità assegnata.

5.1.2 AND

I blocchi AND rappresentano esecuzioni indipendenti che possono essere eseguite in parallelo. Questo vuol dire che se abbiamo *n* strade, tutte saranno eseguite contemporaneamente e si sincronizzeranno alla fine.

5.2 Implementazione

Adesso che abbiamo visto quali blocchi vogliamo aggiungere, vediamo come sono stati implementati e in che punto del workflow vengono eseguiti.

La cosa importante da osservare è che l'obiettivo di questa modifica non è quello di aggiungere funzionalità all'applicazione, ma solamente quello di complicare il workflow. Per questo motivo ogni blocco aggiunto sarà composto da microservizi che non fanno altro che eseguire una busy wait per un tempo campionato da una distribuzione esponenziale; la busy wait è implementata con un ciclo `while` che controlla il tempo trascorso.

5.2.1 XOR

Il blocco XOR, che si trova nella cartella XOR su GitHub, implementa la scelta esculiva tra tre microservizi. Questo blocco è inserito nel flusso di *users-service*: quest'ultimo microservizio chiama quello responsabile della scelta randomica e una volta che il blocco XOR termina riprende il controllo; in questo modo al codice dell'applicazione precedente è stata solo aggiunta la chiamata a questo blocco.

Il flusso del blocco XOR implementato è il seguente:

- L'inizio del blocco è definito da *start-choice-service*. Questo esegue una busy wait per un tempo campionato da una distribuzione esponenziale con tasso 1. Esegue poi una scelta randomica tra tre microservizio, dove: il primo ha probabilità 0.2, il secondo 0.5 e il terzo 0.3. Questo momento è utile per osservare un altro vantaggio di Quarkus: le probabilità con cui sono definite le scelte non sono build-in nel codice, ma sono definite del file *application.properties*; questo permette di modificarle senza dover mettere mano al codice stesso.

- I tre microservizi chiamati sono *first-choice-service*, *second-choice-service* e *third-choice-service*. Ognuno di questi esegue una busy wait per un tempo campionato

da una distribuzione esponenziale con tasso 1. Al termine di uno di questi il controllo ritorna a *users-service*

5.2.2 AND

Il blocco AND, che si trova nella cartella AND su GitHub, implementa l'esecuzione parallela di due servizi chiamati dallo stesso microservizio client; i due server eseguono contemporaneamente e si sincronizzano nel client.

Anche questo blocco è inserito nel flusso di *users-service*, esattamento dopo la chiamata al servizio *start-choice*: il microservizio chiama *start-parallel* che è il responsabile del fork; una volta che il blocco AND termina riprende il controllo; in questo modo al codice dell'applicazione precedente è stata solo aggiunta la chiamata a questo blocco.

Per l'implementazione del blocco AND è stato usato gRPC, già descritto nella Sezione 2.3. In questo caso ha fatto la differenza l'utilizzo del framework Quarkus, che fornisce l'integrazione nativa con gRPC e permette di definire servizi asincroni in modo semplice.

La pipeline usata per implementare la comunicazione asincrona di gRPC con Quarkus è la seguente:

- Il server dichiara un file *.proto*, che definisce i servizi offerti e i messaggi scambiati. Questo permette di lavorare in modo API-first: si definisce cosa il server espone e si lavora a partire da questo.
- Con l'estensione *quarkus-grpc*, quando si chiama `quarkus build`, il framework genera automaticamente tutto quello che ci serve per implementare il server. In particolare la classe che espone il servizio deve implementare un'interfaccia e sovrascrivere i metodi che nel file *.proto* sono i messaggi scambiati.
- Il client deve memorizzare il file *.proto*, compilarlo in modo che Quarkus generi lo stub del server e intrumentare il codice in modo da poter chiamare i metodi esposti dal server come se fossero locali.

gRPC consente quindi di impelmentare una comunizione asincrona tra client e server tramite canali asincroni. Questo si può fare usando l'interfaccia nativa di gRPC o le API di SmallRye.

In gRPC Java classico questi canali sono gestiti dall'interfaccia `StreamObserver<T>`, dove `T` è il tipo del messaggio; ogni comuniazione è quindi un oggetto che implementa tale interfaccia. Quando si invia un messaggio tramite il canale (usando il metodo `onNext(T message)`), gRPC avvia un thread del suo pool di thread dedicati, il quale si occupa della gestione. Questa interfaccia ha però numerosi limiti: è imperativa e quindi tutto deve essere gestito a mano; si basa su callback, cioè molte funzioni

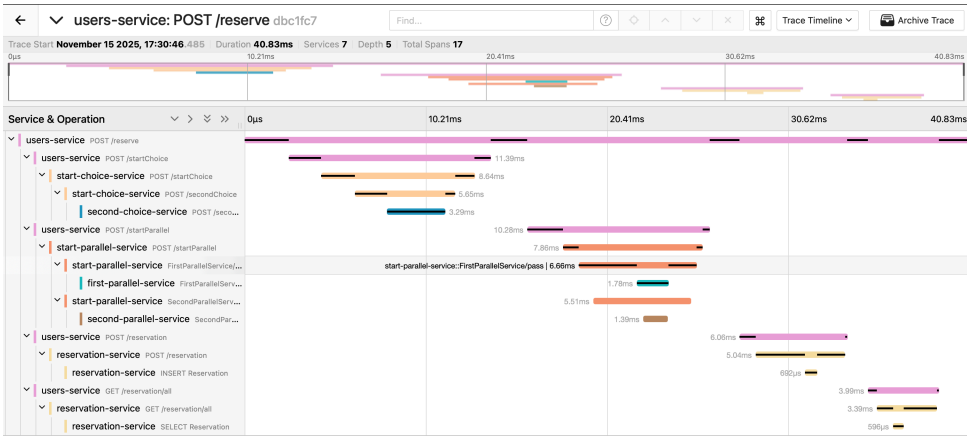

Figura 5.2: Nuovo workflow di `reserve` mostrato da Jaeger.

vengono passate come parametro (uno *StreamObserver* deve implementare tre metodi) e saranno chiamate in un secondo momento; è difficile da testare; non permette di gestire più risponde come un unico oggetto.

Quarkus, ovviamente, non utilizza questa interfaccia, ma sfrutta quella di SmallRye, *Mutiny*. Si tratta di una libreria reattiva e dichiarativa: si deve definire come il sistema si comporta quando i dati arriveranno; non dobbiamo dire come fare le cose ma solo cosa vogliamo ottenere (e.g., nello stesso modo delle lambda). L'interfaccia di *Mutiny* si basa su due tipi, `Uni<T>` e `Multi<T>`: il primo permette l'invio di un solo messaggio, mentre il secondo consente gli stream. I vantaggi che si ottengono sono: codice più leggibile; composizione di chiamate diverse facile da implementare; facile da testare e debuggare.

5.3 Nuovo workflow

Nel nuovo workflow, oltre ad includere i blocchi `XOR` e `AND`, è stata eliminata la parte di fatturazione. Il motivo di questa scelta è in dovuto al nostro obiettivo che ci ha portato a complicare questo workflow: vogliamo valutare il tempo E2E di una prenotazione. La prenotazione e la fatturazione sono in car-rental due operazioni indipendenti: una prenotazione ha successo, e quindi ritorna all'utente, indipendentemente dal completamento della chiamata di `reservation-service` a `billing-service`.

Usando ancora Jeager per visualizzare la traccia della chiamata a `reserve`, possiamo notare dalla Figura 5.2 la presenza dei blocchi `XOR` e `AND` e l'esclusione dei microservizi `billing-service` e `rental-service`. In Figura 5.3 è mostrato invece il sequence diagram UML.

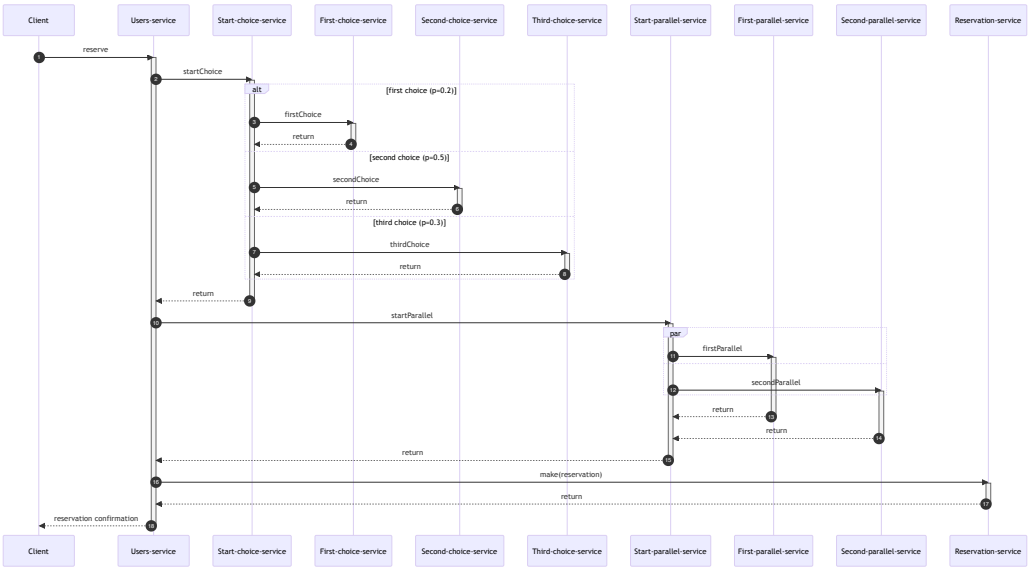

Figura 5.3: Sequence diagram dei blocchi XOR e AND.

6 Load Generator

6.1 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*, in modo da avere un singolo punto dove agire sull'esperimento.

Bibliografia

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

[2] Laura Carnevali et al. «Compositional safe approximation of response time probability density function of complex workflows». In: *ACM Transactions on Modeling and Computer Simulation* 33.4 (2023), pp. 1–26.

[3] Prometheus Community. *Kube Prometheus Stack*. 2025. URL: https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack#configuration (visitato il giorno 04/09/2025).

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

[5] 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).

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

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

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

[9] 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.

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