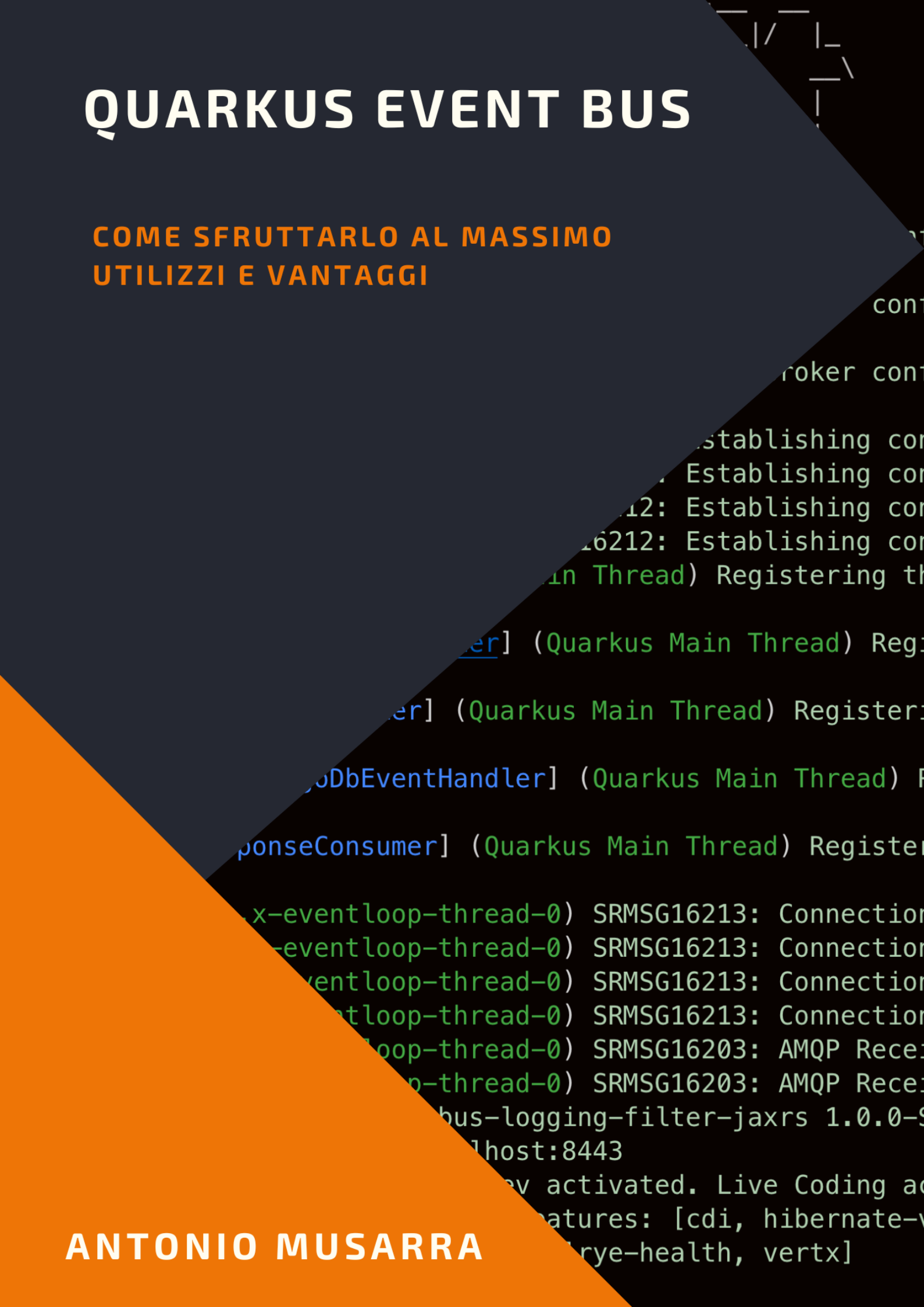
QUARKUS EVENT BUS

COME SFRUTTARLO AL MASSIMO
UTILIZZI E VANTAGGI



ANTONIO MUSARRA

Table of Contents

Informazioni sulla Guida

Antonio Musarra, Sfruttare al massimo l'Event Bus di Quarkus: Utilizzi e Vantaggi

Copertina e impaginazione di Antonio Musarra

Prima edizione digitale Aprile 2024

Serie: Quarkus Dev Guide (#quarkusdevguide)

Promosso da: TheRedCode.it (https://theredcode.it) e Antonio Musarra's Blog (https://www.dontesta.it)

Profilo LinkedIn https://www.linkedin.com/in/amusarra/

Il progetto di esempio realizzato per quest'opera è disponibile su di un repository GitHub all'indirizzo https://github.com/amusarra/eventbus-logging-filter-jaxrs

Quest'opera è stata realizzata usando l'approccio *doc-as-code* e il testo è stato scritto in formato *AsciiDoc*. Il repository GitHub https://github.com/amusarra/eventbus-logging-filter-jaxrs-docs contiene il progetto in formato Maven style da cui è possibile generare la documentazione in formato HTML e PDF di quest'opera.

Nel caso di segnalazioni di errori, suggerimenti o richieste di chiarimenti, si prega di aprire una issue sul rispettivo repository GitHub.

1. Progetto Quarkus:
    a. Issues: https://github.com/amusarra/eventbus-logging-filter-jaxrs/issues
    b. Discussioni: https://github.com/amusarra/eventbus-logging-filter-jaxrs/discussions
2. Progetto Documentazione: https://github.com/amusarra/eventbus-logging-filter-jaxrs-docs/issues

Nell'ambito del social coding e del contributo alla comunità, sono ben accetti *pull request* per migliorare il contenuto di quest'opera e del progetto di esempio.

Note sul Copyright

1. Introduzione

Quarkus, il framework Java progettato per il cloud nativo, offre una vasta gamma di funzionalità per semplificare lo sviluppo delle applicazioni. Una di queste funzionalità è l'Event Bus, un componente fondamentale per la comunicazione asincrona all'interno delle applicazioni. In questo articolo, esploreremo il concetto di Event Bus di Quarkus, i suoi utilizzi e i vantaggi/svantaggi che offre agli sviluppatori.

Nel corso di questa guida entreremo più nel mondo dell'asincronia e della programmazione reattiva, motivo per cui alcune parti di codice potrebbero risultare più complesse rispetto a quelle che siamo abituati a vedere.

Quarkus è reattivo. Ma c'è di più: Quarkus unifica la programmazione reattiva e imperativa (vedi Quarkus Reactive Architecture). Non devi nemmeno scegliere: puoi implementare componenti reattivi e componenti imperativi e combinarli all'interno della stessa applicazione. Non è necessario utilizzare stack, strumenti o API diversi; Quarkus collega entrambi i mondi.

Cercherò di spiegare ogni passaggio nel modo più semplice e chiaro possibile, in modo che anche i lettori meno esperti possano comprendere i concetti e le tecniche presentate. Per coloro che volessero affrontare uno "scontro amichevole tra paradigmi", vi lascio la lettura e visione di Reactive VS Imperative - Primo Episodio di Cristian Bianco e Mauro Celani.

2. Cos'è l'Event Bus di Quarkus?

L'Event Bus di Quarkus è un sistema di messaggistica asincrona (basato su Eclipse Vert.x) che consente alle diverse parti di un'applicazione di comunicare tra loro in modo efficiente e scalabile. È basato sul pattern **publish/subscribe**, in cui i produttori (o publisher) inviano messaggi a un bus centrale e i consumatori (o subscriber) ricevono i messaggi di loro interesse. A seguire gli utilizzi dell'Event Bus di Quarkus.

1. **Integrazione di servizi**: l'Event Bus può essere utilizzato per integrare servizi all'interno di un'architettura a **microservizi**. Ogni microservizio può produrre e consumare eventi attraverso l'Event Bus, consentendo una comunicazione decentralizzata e scalabile.
2. **Notifiche e aggiornamenti**: è possibile utilizzare l'Event Bus per inviare notifiche e aggiornamenti in tempo reale all'interno dell'applicazione. Ad esempio, un servizio può inviare un evento quando avviene una determinata azione e altri servizi possono essere configurati per ricevere e gestire questi eventi di conseguenza.
3. **Elaborazione di flussi di dati**: l'Event Bus può essere utilizzato per gestire flussi di dati in tempo reale. Ad esempio, in un'applicazione di analisi dei dati, i dati possono essere inviati all'Event Bus mentre vengono raccolti e i servizi possono elaborarli in modo asincrono per generare report o avviare azioni specifiche.
4. **Reattività**: utilizzando l'Event Bus, è possibile implementare modelli reattivi all'interno dell'applicazione. I servizi possono essere progettati per reagire in modo dinamico agli eventi che ricevono, migliorando la scalabilità e la capacità di gestire carichi di lavoro variabili.

A seguire i vantaggi dell'Event Bus di Quarkus

1. **Scalabilità**: l'Event Bus consente una comunicazione asincrona e decentralizzata tra i diversi componenti dell'applicazione, consentendo una maggiore scalabilità orizzontale.
2. **Disaccoppiamento**: utilizzando un modello publish/subscribe, l'Event Bus favorisce il disaccoppiamento tra i diversi servizi dell'applicazione, consentendo una maggiore flessibilità e manutenibilità del codice.
3. **Resilienza**: l'Event Bus può gestire automaticamente la perdita di connessione e la disponibilità intermittente dei servizi, garantendo una maggiore resilienza nell'architettura dell'applicazione.
4. **Semplificazione dello sviluppo**: utilizzando l'Event Bus, gli sviluppatori possono concentrarsi sulla logica di business dell'applicazione senza doversi preoccupare della gestione delle comunicazioni asincrone.

Sebbene l'Event Bus di Quarkus offra numerosi vantaggi, è importante considerare anche i possibili svantaggi o sfide associate al suo utilizzo.

1. **Complessità aggiuntiva**: introdurre un Event Bus all'interno di un'applicazione può aumentare la complessità complessiva del sistema. È necessario comprendere il funzionamento dell'Event Bus e gestire correttamente la produzione e il consumo di eventi.
2. **Overhead di gestione**: se non gestito correttamente, l'Event Bus potrebbe aggiungere un overhead significativo alla gestione delle comunicazioni asincrone. È importante ottimizzare l'utilizzo dell'Event Bus per evitare problemi di prestazioni.
3. **Potenziali problemi di coerenza**: l'uso estensivo dell'Event Bus potrebbe portare a potenziali problemi di coerenza dei dati, specialmente in scenari distribuiti. È importante progettare attentamente la gestione degli eventi per garantire la coerenza dei dati nell'intero sistema.
4. **Complessità di debug**: in ambienti distribuiti, il debug delle comunicazioni tramite l'Event Bus potrebbe essere complicato. È importante utilizzare strumenti adeguati per monitorare e tracciare il flusso degli eventi attraverso l'intero sistema.
5. **Possibili ritardi**: l'utilizzo dell'Event Bus introduce un'asincronicità nella comunicazione tra i diversi componenti dell'applicazione. Ciò potrebbe portare a ritardi nella consegna degli eventi e richiedere una gestione appropriata della sincronizzazione dei dati.
6. **Dipendenza dall'infrastruttura**: l'Event Bus dipende dall'infrastruttura sottostante per la sua operatività. Problemi di rete o di disponibilità dei servizi possono influenzare le prestazioni e la disponibilità complessiva dell'Event Bus.

Nonostante questi potenziali svantaggi, l'Event Bus di Quarkus rimane uno strumento potente per la comunicazione asincrona all'interno delle applicazioni cloud native. Con una corretta progettazione e implementazione, è possibile mitigare questi rischi e sfruttare appieno i vantaggi; gli sviluppatori possono implementare architetture scalabili, reattive e resilienti, semplificando lo sviluppo e migliorando le prestazioni delle proprie applicazioni.

3. Come funziona l'Event Bus di Quarkus

L'Event Bus di Quarkus funziona come un sistema di messaggistica asincrona che consente ai diversi componenti di un'applicazione di comunicare tra loro attraverso il meccanismo di pubblicazione e la sottoscrizione.

I messaggi vengono inviati a **indirizzi virtuali** e l'Event Bus offre tre tipi di meccanismi di consegna:

1. **punto-a-punto**: il messaggio viene inviato e un singolo consumatore lo riceve. Se più consumatori ascoltano lo stesso indirizzo, viene applicato un classico algoritmo di *round-robin*;
2. **pubblica/sottoscrivi**: il messaggio viene pubblicato e tutti i consumatori che ascoltano l'indirizzo ricevono il messaggio;
3. **richiesta/risposta**: il mittente invia il messaggio e si aspetta una risposta. Il ricevente può rispondere in modo asincrono.

Tutti questi meccanismi di consegna sono **non bloccanti** e costituiscono uno dei mattoni fondamentali per costruire applicazioni reattive. La caratteristica di scambio di messaggi asincroni consente di rispondere agli stessi, tuttavia, è limitata al comportamento di singolo evento (senza flusso continuo o stream) e ai messaggi locali.

Vediamo in dettaglio come funziona iniziando dai componenti di cui è costituito e poi dalla dinamica.

3.1. Componenti dell'Event Bus

Nel contesto di Quarkus, gli ***eventi*** sono spesso usati come una forma di messaggi (dati o segnali) che vengono pubblicati sull'Event Bus. Questi messaggi possono contenere informazioni pertinenti all'interno dell'applicazione e possono essere utilizzati per comunicare tra diverse parti del sistema. Quindi, nel contesto specifico di Quarkus, "evento" e "messaggio" possono essere considerati interscambiabili e si riferiscono entrambi ai dati trasmessi attraverso l'Event Bus.

1. **Producer (Produttore)**: un produttore è un componente dell'applicazione che pubblica eventi sull'Event Bus. Quando un evento significativo si verifica nell'applicazione, il produttore lo pubblica sull'Event Bus in modo che altri componenti possano riceverlo e reagire di conseguenza.
2. **Consumer (Consumatore)**: un consumatore è un componente dell'applicazione che sottoscrive specifici tipi di eventi sull'Event Bus. Quando un evento di interesse per il consumatore viene pubblicato sull'Event Bus, il consumatore lo riceve e può eseguire le azioni necessarie in risposta all'evento.
3. **Event Bus (Bus degli Eventi)**: è il canale di comunicazione centrale attraverso il quale i produttori pubblicano eventi e i consumatori li ricevono. L'Event Bus gestisce la distribuzione

degli eventi ai consumatori interessati e garantisce l'affidabilità della comunicazione asincrona.

3.2. Funzionamento dell'Event Bus

Il funzionamento dell'Event Bus può essere descritto in cinque punti che sono:

1. **pubblicazione degli eventi**: quando si verifica un evento significativo nell'applicazione, il produttore pubblica l'evento sull'Event Bus utilizzando un meccanismo di pubblicazione. L'evento può contenere dati pertinenti associati all'azione che ha causato la pubblicazione dell'evento;
2. **sottoscrizione agli eventi**: I consumatori interessati agli eventi di un determinato tipo si sottoscrivono all'Event Bus specificando i tipi di eventi di loro interesse. In questo modo, i consumatori riceveranno solo gli eventi rilevanti per le loro funzionalità;
3. **distribuzione degli eventi**: l'Event Bus gestisce la distribuzione degli eventi (in base ai meccanismi di consegna supportati) ai consumatori sottoscritti. Quando un evento viene pubblicato sull'Event Bus, questo invia l'evento a tutti i consumatori interessati ai tipi di eventi corrispondenti;
4. **gestione degli eventi**: i consumatori ricevono gli eventi e li gestiscono in base alle logiche di business specifiche dell'applicazione. Possono elaborare gli eventi, aggiornare lo stato dell'applicazione o avviare azioni specifiche in risposta agli eventi ricevuti;
5. **affidabilità e scalabilità**: l'Event Bus di Quarkus gestisce in modo affidabile la comunicazione asincrona tra i componenti dell'applicazione, garantendo la consegna degli eventi ai consumatori interessati. È inoltre progettato per essere scalabile, consentendo una gestione efficiente di grandi volumi di eventi in applicazioni distribuite.

Il diagramma di sequenza semplificato (vedi Diagramma 1) illustra l'interazione tra **Producer**, **Consumer** e l'**Event Bus**.

1. **Producer**
   - Il Producer genera un evento o un messaggio.
   - Questo evento viene inviato all'Event Bus.
   - Nel diagramma, rappresentiamo il Producer come una freccia che punta verso l'Event Bus.
2. **Event Bus**
   - L'Event Bus riceve l'evento dal Producer.
   - L'Event Bus inoltra l'evento a tutti i Consumer interessati.
3. **Consumer**
   - I Consumer sono i destinatari dell'evento.
   - Ogni Consumer riceve l'evento dall'Event Bus.

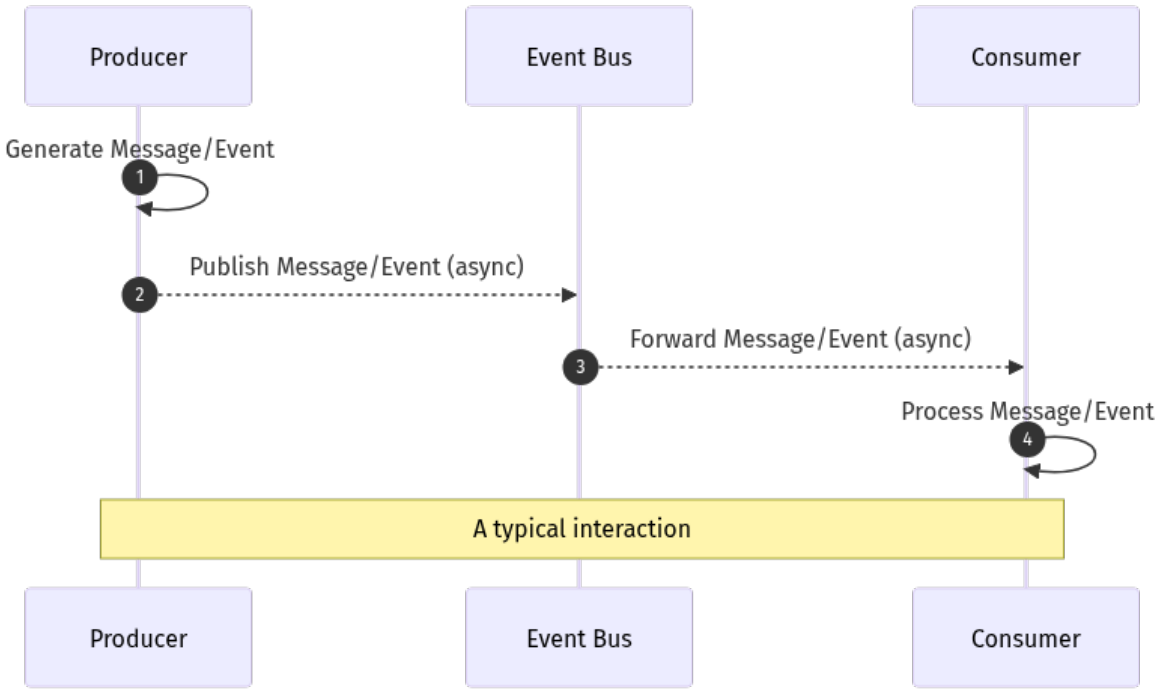


Diagramma 1 - *Interazione tra Producer, Consumer e l'Event Bus*

4. Progettiamo qualcosa di concreto

Ipotizziamo che per la nostra applicazione Quarkus sia richiesta la capacità di poter memorizzare le richieste/risposte **JAX-RS** (Java API for RESTful Web Services) che essa riceve. È inoltre richiesto che queste informazioni possano essere memorizzate su diversi supporti, come per esempio database SQL, database NoSQL o una coda AMQP (Advanced Message Queuing Protocol).

Al fine di rendere più chiaro il requisito, a seguire è illustrato il diagramma che mostra come dovrà essere il flusso dell'applicazione, i componenti interessati e loro relazioni.

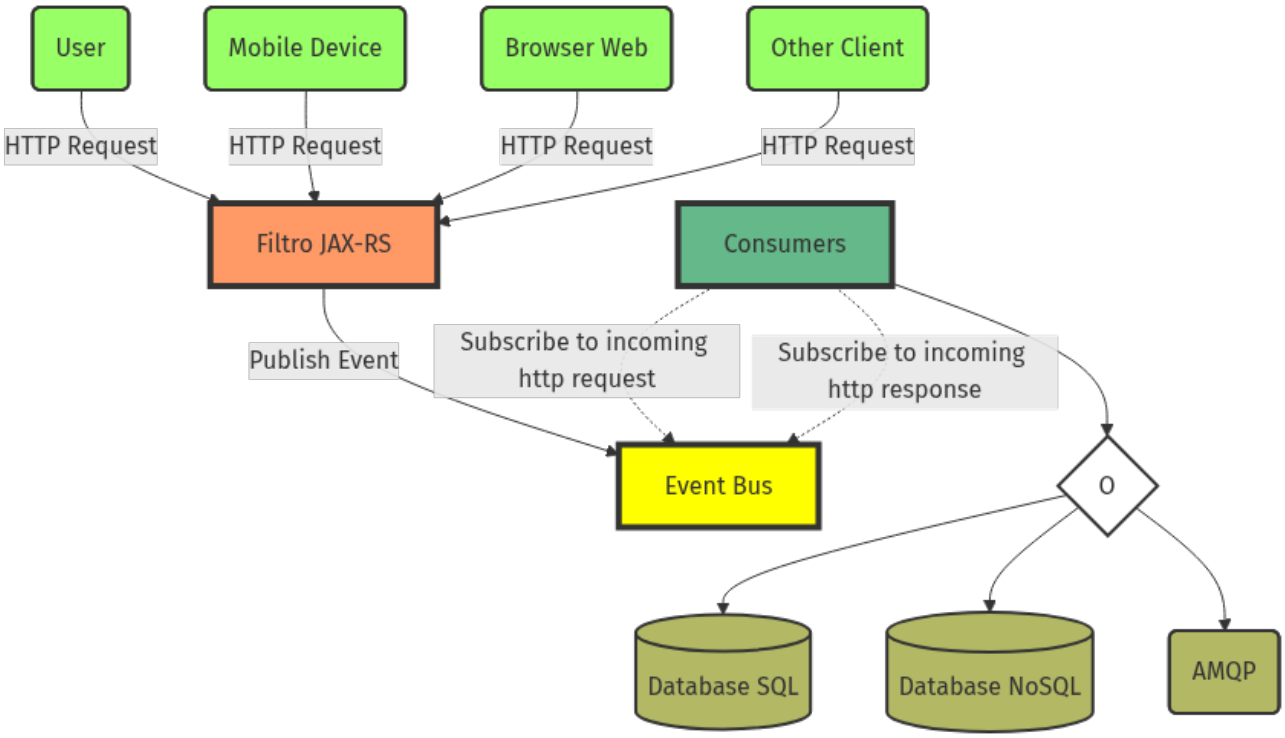


Diagramma 2 - *Flusso dell'applicazione Quarkus per lo storage delle richieste JAX-RS*

Stando al Diagramma 2, le richieste HTTP (verso i servizi REST) provenienti per esempio da utenti, dispositivi mobili, browser web o qualunque altro tipo di client, sono intercettate ed elaborate dal filtro JAX-RS e successivamente pubblicate sull'Event Bus. I Consumer ricevono gli eventi per cui hanno la sottoscrizione, inviandoli poi, a seconda della configurazione e delle esigenze dell'applicazione, verso il Database SQL, Database NoSQL e la coda AMQP.

Ricordiamo che il filtro JAX-RS oltre a processare la request, processerà anche la response, motivo per cui gli eventi pubblicati sull'Event Bus saranno sia per la request sia per la response, di conseguenza ci saranno i rispettivi consumer. Il diagramma di sequenza a seguire aiuterà a capire meglio il ruolo del filtro JAX-RS.

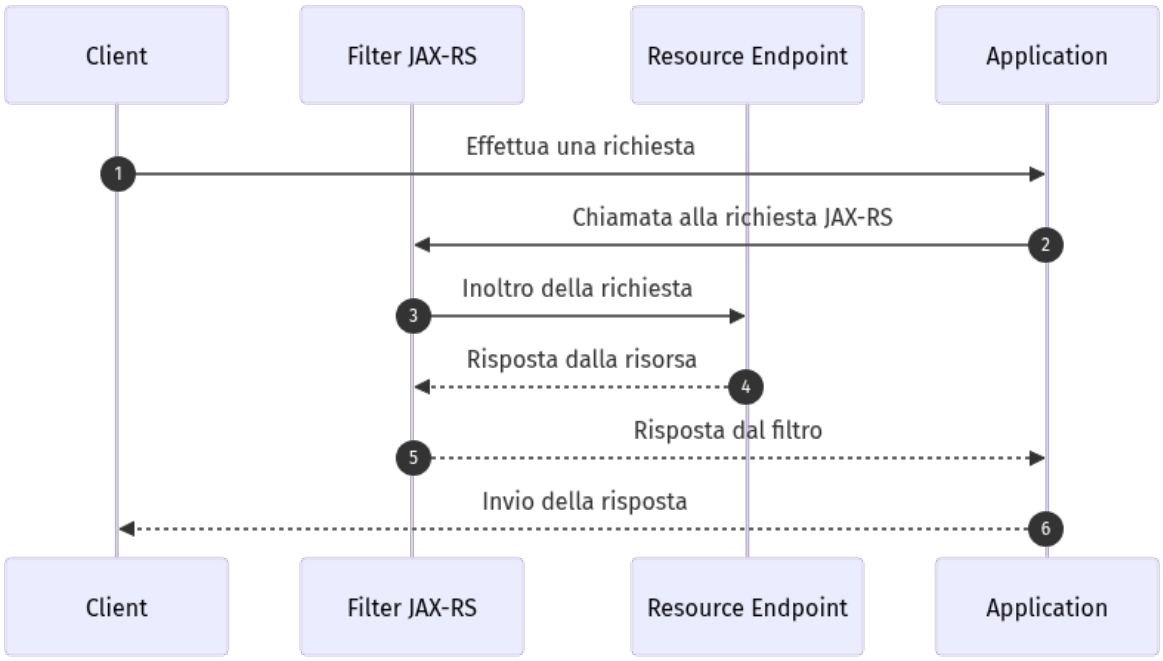


Diagramma 3 - *Sequenza che illustra il flusso di dati attraverso un filtro JAX-RS all'interno dell'applicazione quando viene effettuata una richiesta da un client*

Il diagramma di sequenza precedente mostra il flusso di dati attraverso un filtro JAX-RS all'interno dell'applicazione quando viene effettuata una richiesta da un client.

- Il Client effettua una richiesta all'applicazione.
- L'applicazione, ricevendo la richiesta, la passa attraverso il Filter JAX-RS.
- Il Filter JAX-RS inoltra la richiesta al Resource Endpoint appropriato.
- Il Resource Endpoint elabora la richiesta e invia una risposta.
- Il Filter JAX-RS riceve la risposta dal Resource Endpoint e la passa indietro all'applicazione.
- Infine, l'applicazione invia la risposta al Client.

Il diagramma di sequenza a seguire aggiunge l'Event Bus e su questo sono pubblicati dal filtro JAX-RS i messaggi di richiesta e i messaggi di risposta (vedi step 4 e step 8) del servizio JAX-RS chiamato.

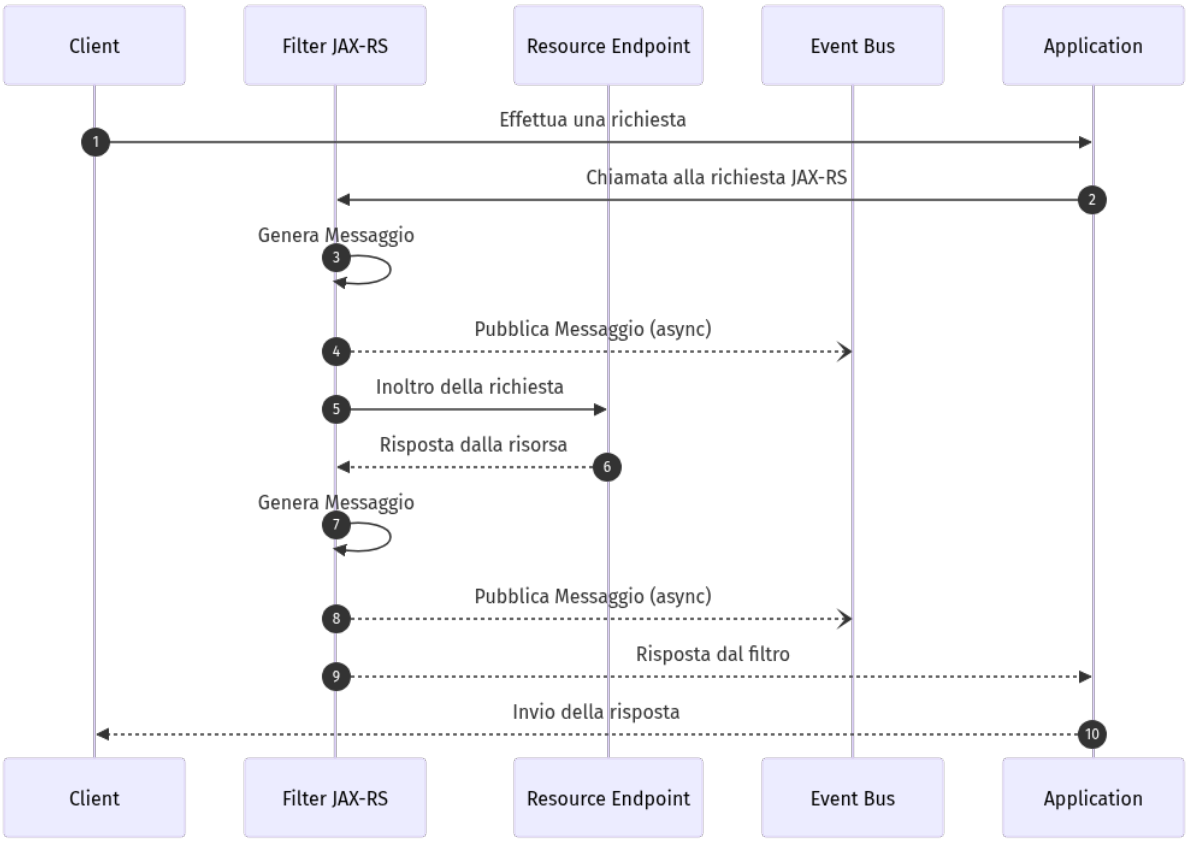


Diagramma 4 - *Estensione del diagramma precendente con l'aggiunta delle interazioni con l'Event Bus*

Quando i messaggi arrivano sul consumer, questi saranno elaborati dallo stesso e successivamente inviati al **Dispatcher** che sarà responsabile di veicolare il messaggio verso il corretto gestore (**Event Handler**) e quest'ultimo sarà l'effettivo responsabile dello store sul database SQL, NoSQL o sulla coda AMQP.

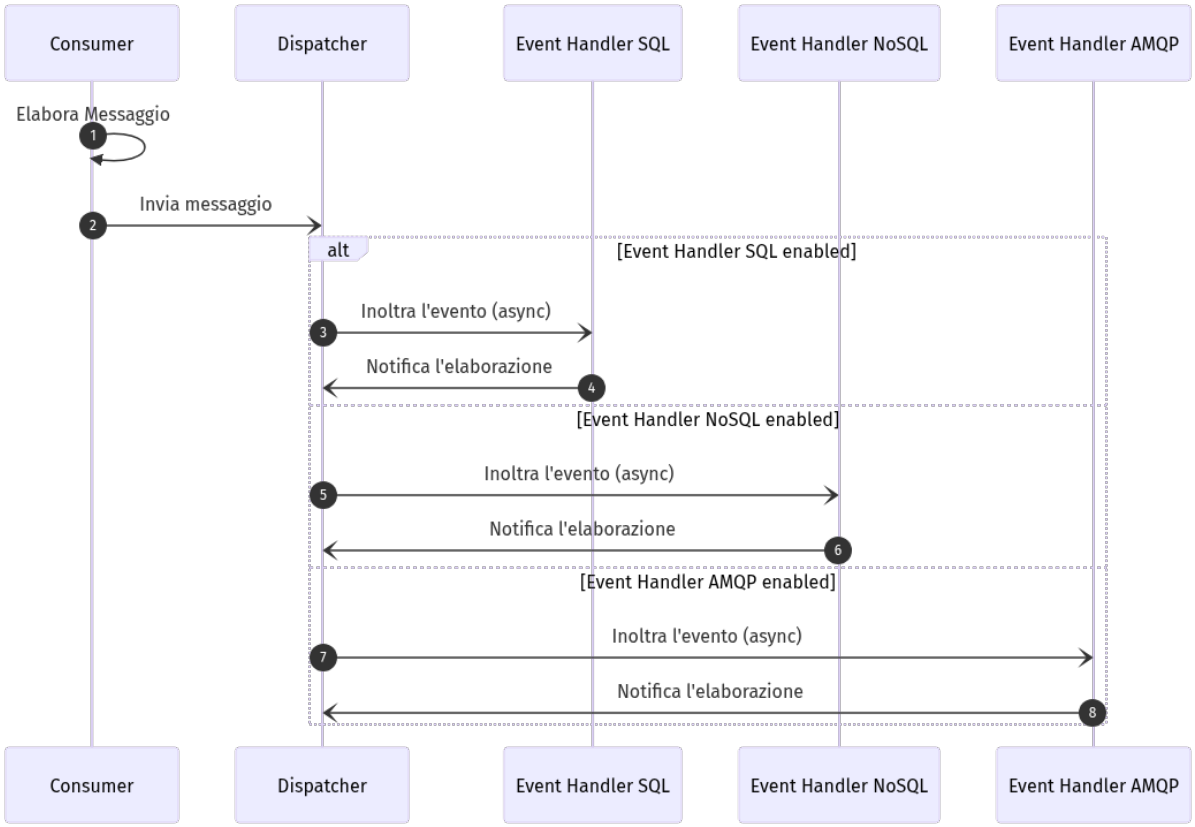


Diagramma 5 - *Utilizzo del Dispatcher per veicolare la richiesta di store delle richieste/risposte JAX-RS verso il sistema configurato.*

Il blocco *alt (abbreviazione di "alternative")* del diagramma di sequenza precedente mostra come il Dispatcher inoltra l'evento all'Event Handler SQL, NoSQL o AMQP in base alla configurazione dell'applicazione. Nel diagramma di flusso precedente (diagramma 2), il Dispatcher è rappresentato dall'Inclusive Gateway (notazione BPMN). Le operazioni del blocco alt sono asincrone, per cui l'ordine l'ordine di esecuzione delle operazioni non è garantito. Una volta che l'Event Handler ha elaborato l'evento, notifica il Dispatcher dell'esito dell'operazione.

4.1. La lista della spesa

Dopo aver visto cosa voler realizzare sfruttando l'Event Bus di Quarkus e quali componenti sono coinvolti e le interazioni/relazioni tra loro grazie all'ausilio dei vari diagrammi, è arrivato il momento di fare quella che personalmente chiamo "lista della spesa" dove andremo a elencare i componenti essenziali da implementare al fine di rendere operativo il progetto.

Nome	Descrizione
Resource Endpoint	È il componente che implementa il punto di accesso o un'interfaccia attraverso cui è possibile accedere o manipolare una risorsa specifica all'interno di un'applicazione o di un sistema. Per lo scopo di questo progetto, l'implementazione di questo componente sarà abbastanza semplice.

Nome	Descrizione
Filtro JAX-RS	Come mostrato dal sequence diagram (diagramma 4). Questo componente sarà responsabile di filtrare richieste e risposte con l'obiettivo di preparare un messaggio che conterrà alcune informazioni che saranno poi pubblicate sull'Event Bus. Per ulteriori informazioni sui filtri JAX-RS, fare riferimento alla specifica Jakarta EE.
Http Request Consumer	Questo componente è il consumer che si registrerà all'indirizzo virtuale dove saranno pubblicati i messaggi contenenti le informazioni che riguardano le richieste JAX-RS ricevute dall'applicazione e opportunamente filtrate. L'indirizzo virtuale sarà configurato sul file di configurazione dell'applicazione Quarkus.
Http Response Consumer	Questo componente è il consumer che si registrerà all'indirizzo virtuale dove saranno pubblicati i messaggi contenenti le informazioni che riguardano le risposte JAX-RS ricevute dall'applicazione e opportunamente filtrate. L'indirizzo virtuale sarà configurato sul file di configurazione dell'applicazione Quarkus.
Dispatcher	Così come indicato nel diagramma 5, questo componente sarà il responsabile della gestione del flusso degli eventi verso gli Event Handler.
Event Handlers	Quei componenti che sono sempre dei consumer, saranno responsabili della gestione degli eventi inoltrati dal Dispatcher e una volta elaborati inviati verso il sistema esterno (SQL, NoSQL e AMPQ). Così come indicato del diagramma 5, ogni Event Handler dovrà notificare l'esito dell'operazione.

Tabella 1 - Lista dei componenti da implementare per il progetto Quarkus

Dopo aver descritto i requisiti che l'applicazione Quarkus deve soddisfare, e quali componenti sono necessari, è possibile procedere con l'implementazione del progetto mettendo le mani in pasta al codice.

5. Implementiamo step-by-step

Dopo aver stilato i requisiti minimi per la nostra applicazione e fatta la necessaria progettazione della soluzione, siamo nelle condizioni di poter procedere con la realizzazione vera e propria, cercando di seguire step-by-step i vari stadi riassunti a seguire.

1. Creazione del progetto Quarkus
2. Realizzazione del Resource Endpoint (servizio Rest)
3. Realizzazione del filtro JAX-RS
4. Definizione degli indirizzi virtuali dell'Event Bus
5. Registrazione dei Consumer per gli eventi dell'Event Bus
6. Adeguamento del filtro JAX-RS per inviare l'evento/messaggio sull'Event Bus
7. Realizzazione del Dispatcher e degli Event Handler

Prima d'iniziare con qualunque attività d'implementazione, è necessario che i requisiti indicati nella tabella a seguire come non opzionali siano soddisfatti. In tabella non sono indicati il terminale (o console del sistema operativo) e l'IDE di cui avete ampia libertà di scelta.

Nome	Opzionale	Descrizione
Java JDK 17/21	NO	Implementazione di OpenJDK 17/21. È possibile usare qualunque delle implementazioni disponibili. Per questo articolo è stata usata la versione 21 di OpenJDK e l'implementazione di Amazon Corretto 21.0.2.
Git	NO	Tool di versioning.
Maven 3.9.6	NO	Tool di build per i progetti Java e di conseguenza Quarkus.
Quarkus 3.9.2	NO	Framework Quarkus 3.9.2 la cui release note è disponibile qui https://quarkus.io/blog/quarkus-3-9-2-released/. Per maggiori informazioni per le release LTS fare riferimento all'articolo Long-Term Support (LTS) for Quarkus.

Nome	Opzionale	Descrizione
Quarkus CLI	SI	Tool a linea di comando che consente di creare progetti, gestire estensioni ed eseguire attività essenziali di creazione e sviluppo. Per ulteriori informazioni su come installare e utilizzare la CLI (Command Line Interface) di Quarkus, consulta la guida della CLI di Quarkus.
Docker v26 o Podman v4/5	NO	Tool per la gestione delle immagini e l'esecuzione dell'applicazione in modalità container. La gestione delle immagini/container sarà necessaria nel momento in cui saranno sviluppati gli Event Handler che dovranno comunicare con i servizi esterni all'applicazione (vedi NoSQL, SQL, AMQP). La gestione delle immagini necessarie e container, sarà totalmente trasparente per noi sviluppatori in quanto a carico dei Dev Services di Quarkus.
GraalVM	SI	Per la build dell'applicazione in modalità nativa. Per maggiori informazioni fare riferimento alla documentazione Building a Native Executable.
Ambiente di sviluppo C	SI	Richiesto da GraalVM per la build dell'applicazione nativa. Per maggiori informazioni fare riferimento alla documentazione Building a Native Executable.
cURL 7.x/8.x	SI	Tool per il test dei Resource Endpoint (servizi REST)

Tabella 2 - Requisiti (anche opzionali) necessari per l'implementazione del progetto Quarkus

L'intero progetto a cui ho dato il nome Event Bus Logging filter JAX-RS è disponibile su GitHub e può essere clonato o scaricato per essere esaminato e testato. Inoltre, il progetto è stato realizzato usando la CLI di Quarkus e Maven, quindi è possibile eseguire il progetto in modalità sviluppo, testarlo e, se necessario, eseguire la build in modalità nativa. All'interno del progetto è presente un file `README_it.md` che contiene tutte le informazioni necessarie per la compilazione, l'esecuzione e il test del progetto.

Nota: Nel caso in cui abbiate Podman al posto di Docker, potreste incontrare in fase di build, test e avvio dell'applicazione l'errore: **Could not find a valid Docker environment**.
Per risolvere il problema fare riferimento al README.md del progetto e in particolare la sezione *Qualche nota sulla configurazione di Podman*.

Per ogni step di realizzazione del progetto (precedentemente indicato) esiste il relativo tag, così che sia possibile seguire passo-passo l'evoluzione del progetto. Per esempio, nel caso in cui volessimo visionare il progetto al termine del primo step, sarà possibile eseguire il comando `git checkout step-1`, e così via per gli altri step. Quando inizieremo ogni step di realizzazione del progetto, sarà indicato il tag di riferimento.

5.1. Creazione del progetto Quarkus

In questo primo step, il cui tag di riferimento è step-1, procederemo con la creazione del progetto Quarkus predisposto per l'utilizzo dell'Event Bus. È possibile creare il progetto usando Maven o Quarkus CLI. L'uso di Quarkus CLI rende più semplice la gestione di progetti Quarkus, per cui ne consiglio l'utilizzo.

Console 1 - *Creazione del progetto Quarkus*

```
# Creazione del progetto utilizzando Quarkus CLI
quarkus create app it.dontesta.eventbus:eventbus-logging-filter-jaxrs \
    --extension='vertx' \
    --no-code

# Creazione del Progetto utilizzando Maven
mvn io.quarkus.platform:quarkus-maven-plugin:3.9.2:create \
    -DprojectGroupId=it.dontesta.eventbus \
    -DprojectArtifactId=eventbus-logging-filter-jaxrs \
    -Dextensions='vertx' \
    -DnoCode
```

Eseguendo i due comandi indicati in precedenza dovreste ottenere il rispettivo output illustrato dalle due immagini a seguire. Il risultano ottenuto è identico per entrambe i comandi.

```
❯ quarkus create app it.dontesta.eventbus:eventbus-logging-filter-jaxrs \
    --extension='vertx' \
    --no-code
Looking for the newly published extensions in registry.quarkus.io
------------
selected extensions:
- io.quarkus:quarkus-vertx


applying codestarts...
📚 java
🔨 maven
📦 quarkus
📝 config-properties
🔧 tooling-dockerfiles
🔧 tooling-maven-wrapper

------------
[SUCCESS] ✅  quarkus project has been successfully generated in:
--> /Users/amusarra/dev/github/amusarra/eventbus-logging-filter-jaxrs
------------
Navigate into this directory and get started: quarkus dev
```

Figura 1 - *Esempio di esecuzione della creazione del progetto Quarkus usando la CLI*

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] --- quarkus:3.9.2:create (default-cli) @ standalone-pom ---
[INFO] -----------
[INFO] selected extensions:
- io.quarkus:quarkus-vertx

[INFO]
applying codestarts...
[INFO] 📚 java
🔨 maven
📦 quarkus
📝 config-properties
🔧 tooling-dockerfiles
🔧 tooling-maven-wrapper
[INFO]
------------
[SUCCESS] ✅  quarkus project has been successfully generated in:
--> /Users/amusarra/dev/github/amusarra/eventbus-logging-filter-jaxrs
-----------
[INFO]
[INFO] ========================================================================================
[INFO] Your new application has been created in /Users/amusarra/dev/github/amusarra/eventbus-logging-filter-jaxrs
[INFO] Navigate into this directory and launch your application with mvn quarkus:dev
[INFO] Your application will be accessible on http://localhost:8080
[INFO] ========================================================================================
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.000 s
[INFO] Finished at: 2024-04-08T14:21:49+02:00
[INFO] ------------------------------------------------------------------------
```

Figura 2 - *Esempio di esecuzione della creazione del progetto Quarkus usando Maven*

Il GAV (Group Id, Artifact Id e Version) della nostra applicazione è it.dontesta.eventbus:eventbus-logging-filter-jaxrs:1.0.0-SNAPSHOT e la directory del progetto per impostazione predefinita è eventbus-logging-filter-jaxrs il cui contenuto è quello mostrato a seguire.

Console 2 - *Struttura del progetto Quarkus dopo la creazione*

```
.
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
└── main
├── docker
│   ├── Dockerfile.jvm
│   ├── Dockerfile.legacy-jar
│   ├── Dockerfile.native
│   └── Dockerfile.native-micro
├── java
└── resources
└── application.properties

6 directories, 9 files
```

L'applicazione è stata creata specificando l'estensione Vertx; ricordo che l'Event Bus di Quarkus è basato su Eclipse Vert.x e in particolare sulla versione 4.5.7 (io.vertx:vertx-core). I più curiosi potrebbero verificarlo usando il comando `mvn dependency:tree -Dverbose|grep io.vertx` eseguito dalla directory del progetto.

5.2. Realizzazione del Resource Endpoint

In questo secondo step, il tag di riferimento è step-2.4. Prima d'iniziare a tracciare le richieste JAX-RS, dobbiamo creare almeno un endpoint REST o (resource endpoint). Per gli scopi del progetto creeremo un endpoint cosiddetto di "echo", termine utilizzato comunemente nel contesto delle API e delle comunicazioni di rete. L'endpoint di echo è un endpoint di servizio che restituisce indietro (o "eco") i dati inviati a esso. In altre parole, qualsiasi richiesta inviata a un endpoint di echo viene restituita esattamente allo stesso modo in cui è stata ricevuta, senza alcuna elaborazione o modifica.

Al fine di poter implementare questo endpoint, nel progetto Quarkus dobbiamo aggiungere il modulo Quarkus REST la cui documentazione di riferimento è Writing Rest Services with Quarkus REST (formerly Rest Easy Reactive).

Per abilitare il modulo è sufficiente usare la CLI di Quarkus eseguendo il comando `quarkus ext add io.quarkus:quarkus-rest` o aggiungere la dipendenza al `pom.xml` del progetto.

Configurazione 1 - *Dipendenza del modulo Quarkus Rest da aggiungere al pom.xml del progetto*

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest</artifactId>
</dependency>
```

Installando il modulo tramite la CLI di Quarkus, l'output atteso e indicato a seguire.

Console 3 - *Output del comando quarkus ext add che informa l'avvenuta aggiunta del modulo al progetto*

```
[SUCCESS] Extension io.quarkus:quarkus-rest has been installed
```

Oltre ad aggiungere il modulo Quarkus REST aggiungiamo anche i moduli:

1. Quarkus REST Jackson: un'estensione di Quarkus che fornisce un'integrazione predefinita tra Quarkus, JAX-RS (per la creazione di servizi REST) e Jackson (per la serializzazione e la deserializzazione degli oggetti JSON);
2. Quarkus Hibernate Validator: integra Hibernate Validator in un'applicazione Quarkus. Hibernate Validator è un'implementazione di riferimento di JSR 380 (Jakarta Bean Validation 2.0) ed è ampiamente utilizzato per la validazione dei bean nelle applicazioni Java.

Per l'abilitazione dei due moduli è sufficiente usare la CLI di Quarkus eseguendo i comandi `quarkus ext add io.quarkus:hibernate-validator` e `quarkus ext add io.quarkus:quarkus-rest-jackson` o aggiungere le due dipendenze al pom.xml del progetto.

Configurazione 2 - *Dipendenza del modulo Hibernate Validator e Quarkus REST Jackson da aggiungere al pom.xml del progetto*

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-validator</artifactId>
</dependency>

<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-jackson</artifactId>
</dependency>
```

Nota: È possibile aggiungere le dipendenze di moduli Quarkus sfruttando anche il plugin Maven di Quarkus eseguendo il comando `mvn quarkus:add-extension -Dextensions='<nome-estensione>'` o `./mvnw quarkus:add-extension -Dextensions='<nome-estensione>'`.

A questo punto passiamo alla creazione dell'Application JAX-RS e del Resource Endpoint Echo che si traduce rispettivamente nella scrittura delle due classi `EventBusApplication` e `EchoResourceEndPoint` mostrate a seguire.

Code 1 - *Application JAX-RS dove viene impostato il contest root per le API*

```
package it.dontesta.eventbus.ws;

import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

@ApplicationPath("/api")
public class EventBusApplication extends Application {
}
```

Code 2 - *Resource Endpoint per il servizio Rest di echo*

```
package it.dontesta.eventbus.ws.resources.endpoint;

import jakarta.validation.constraints.Size;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

@Path("rest")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class EchoResourceEndPoint {

  @Path("echo")
  @POST
  public Response echo(
      @Size(min = 32, max = 4096,
          message = "Il parametro di input deve essere compreso tra 32 byte e 4 KB")
      String input) {
    return Response.ok(input).build();
  }
}
```

Una volta aggiunte le due classi al progetto, sarà possibile avviare l'applicazione ed eseguire un semplice test attraverso l'ausilio del comando cURL. Rispetto a prima, il progetto sarà così strutturato.

Console 4 - *Struttura del progetto Quarkus dopo la creazione delle classi EventBusApplication e EchoResourceEndPoint*

```
.
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── main
        ├── docker
        │   ├── Dockerfile.jvm
        │   ├── Dockerfile.legacy-jar
        │   ├── Dockerfile.native
        │   └── Dockerfile.native-micro
        ├── java
        │   └── it
        │       └── dontesta
        │           └── eventbus
        │               └── ws
        │                   ├── EventBusApplication.java
        │                   └── resources
        │                       └── endpoint
        │                           └── EchoResourceEndPoint.java
        └── resources
            └── application.properties

12 directories, 11 files
```

Per avviare il progetto ed eseguire il test del servizio che risponde alla URL `/api/rest/echo` è sufficiente eseguire i seguenti comandi.

Console 5 - *Esecuzione dell'applicazione Quarkus in Dev Mode*

```
# Esecuzione dell'applicazione in Dev Mode utilizzando Quarkus CLI
quarkus dev

# Esecuzione dell'applicazione in Dev Mode utilizzando Maven
mvn clean quarkus:dev
```

Se tutto è andato per il verso giusto, dovreste ottenere in console un output simile a quello mostrato a seguire. Dall'output sono disponibili diverse informazioni e una di nostra interesse è la porta di ascolto per i servizi HTTP, in questo caso http://localhost:8080 (URL di default).

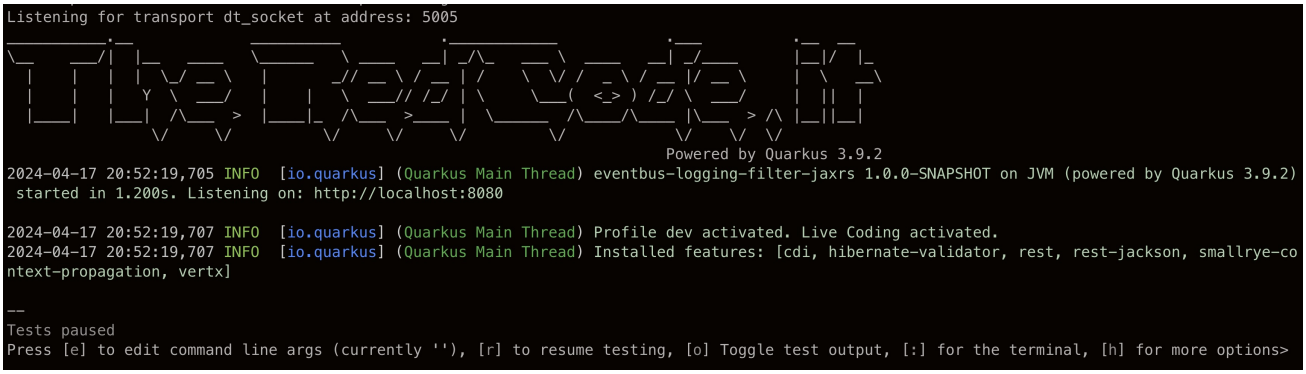


Figura 3 - *Output del primo avvio dell'applicazione Quarkus*

Per testare il servizio di echo è sufficiente eseguire il comando cURL che punti all'indirizzo http://localhost:8080/api/rest/echo passando un payload JSON che sia compreso tra 32 byte e 4096 byte (vedi riga 19 di code 2). Il risultato atteso in risposta e appunto il riflesso dell'input, ovvero, il payload fornito in input.

Console 6 - *Test del servizio* `/api/rest/echo` *attraverso il tool cURL (richiesta e risposta)*

```
# Esecuzione della chiamata JAX-RS verso l'endpoint /api/rest/echo
curl -v \
    -H "Content-Type: application/json" \
    -d '{"id": 123,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "age": 30,
  "address": {
    "street": "123 Main Street",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "555-1234"
    },
    {
      "type": "work",
      "number": "555-5678"
    }
  ]
}
' \
http://localhost:8080/api/rest/echo
```

```
# Risposta del servizio /api/rest/echo

* Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080
> POST /api/rest/echo HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: _/_
> Content-Type: application/json
> Content-Length: 345
>
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
< content-length: 345
<
{"id": 123,
"name": "John Doe",
"email": "john.doe@example.com",
"age": 30,
"address": {
  "street": "123 Main Street",
  "city": "Anytown",
  "state": "CA",
  "zip": "12345"
},
"phoneNumbers": [
  {
    "type": "home",
    "number": "555-1234"
  },
  {
    "type": "work",
    "number": "555-5678"
  }
]
}
* Connection #0 to host localhost left intact
```

Se ricordate, l'implementazione del servizio fa uso delle Validation Bean API (vedi code 2) e in particolare per il parametro di input, cui è richiesta una dimensione tra 32 byte e 4096 byte. Nel caso in cui il parametro di input non dovesse rispettare questa regola, il servizio risponderà con un codice HTTP/400 (o Bad Request) e un JSON con il dettaglio dell'errore, così come indicato a seguire.

Console 7 - *Test del servizio* `/api/rest/echo` *attraverso il tool cURL (richiesta e risposta) e in particolare della validazione dell'input*

```
# Esecuzione della chiamata JAX-RS verso l'endpoint /api/rest/echo
# con un payload che non rispecchia la regola di validazione.
# In pipe al comando cURL è presente il comando jq allo scopo
# di fare il lint del JSON restituito dal servizio.

curl -v \
    -H "Content-Type: application/json" \
    -d '{"message": "Hello, world!"}' \
http://localhost:8080/api/rest/echo | jq

# Output del servizio che risponde con un messaggio di errore

# circa la validazione del parametro di input.
# Dall'output sono state eliminate le informazioni superflue

* Connected to localhost (127.0.0.1) port 8080
> POST /api/rest/echo HTTP/1.1
> Content-Type: application/json
> Content-Length: 28
>
} [28 bytes data]
< HTTP/1.1 400 Bad Request
< validation-exception: true
<
* Connection #0 to host localhost left intact
{
"title": "Constraint Violation",
"status": 400,
"violations": [
  {
    "field": "echo.input",
    "message": "Il parametro di input deve essere compreso tra 32 byte e 4 KB"
  }
]
}
```

All'interno del progetto GitHub è possibile trovare la classe di test `EchoResourceEndPointTest` che contiene i test per il servizio di echo e che potete eseguire per verificare il corretto funzionamento del servizio. Per lanciare i test è sufficiente eseguire il comando `mvn test` o `quarkus test` dalla directory del progetto

Dopo aver appurato il corretto funzionamento del servizio di echo, possiamo proseguire con il prossimo passo d'implementazione, ovvero, la realizzazione del filtro JAX-RS.

6. Aggiornamento v1.4.1 — Endpoint reattivo

Quanto descritto in questa sezione si riferisce alla versione **1.4.1** del progetto software.

A partire dalla versione **1.4.1**, `EchoResourceEndPoint` include un secondo endpoint reattivo `POST /api/rest/echo/reactive` che restituisce `Uni<Response>` anziché `Response`, abilitando il percorso non-bloccante (event-loop) di Quarkus REST. La stessa regola di validazione bean (`@Size(min = 32, max = 4096)`) si applica all'input.

Code 2b - *Endpoint reattivo aggiunto in v1.4.1*

```
@Path("echo/reactive")
@POST
public Uni<Response> echoReactive(
    @Size(min = 32, max = 4096,
        message = "Il parametro di input deve essere compreso tra 32 byte e 4 KB")
    String input) {
  return Uni.createFrom().item(Response.ok(input).build());
}
```

L'endpoint reattivo è utile per:

- verificare il percorso **event-loop** nel filtro (`TraceJaxRsRequestResponseFilter#readBodyAsync`);
- confrontare le prestazioni tra endpoint bloccante e non-bloccante sotto carico.

Console 6b - *Test dell'endpoint reattivo* `/api/rest/echo/reactive`

```
curl -v \
  -H "Content-Type: application/json" \
  -d '{"message": "Test endpoint reattivo con Uni<Response> in Quarkus REST"}' \
  http://localhost:8080/api/rest/echo/reactive
```

La classe di test `EchoResourceEndPointTest` è stata aggiornata con i metodi `testEchoReactiveSuccess`, `testEchoReactiveValidation` e `testEchoReactiveBodyEmpty` che coprono i principali scenari dell'endpoint reattivo.

6.1. Realizzazione del filtro JAX-RS

In questo terzo step, il tag di riferimento è step-3. Al capitolo *3 Progettiamo qualcosa di concreto* è stato introdotto brevemente il filtro JAX-RS come componente della soluzione che stiamo implementando e i diagrammi di sequenza 2 e 3 ne mostrano l'uso nel flusso dell'applicazione. Prima di scrivere qualche riga di codice, è utile fare un breve riassunto su cosa sono questi filtri e

come funzionano.

In JAX-RS, un filtro è un componente che consente di intercettare e modificare le richieste in arrivo e le risposte in uscita prima che vengano gestite dai resource endpoint. I filtri vengono utilizzati per implementare logiche di pre-elaborazione e post-elaborazione, come autenticazione, autorizzazione, logging, manipolazione dei dati e altro ancora.

Ci sono due tipi principali di filtri in JAX-RS.

1. **Filtro di richiesta (Request Filter)**: questo tipo di filtro intercetta le richieste in arrivo prima che vengano instradate ai resource endpoint. Può essere utilizzato per eseguire operazioni come l'autenticazione dell'utente, la validazione dei parametri di input, la registrazione delle richieste e altro ancora.
2. **Filtro di risposta (Response Filter)**: questo tipo di filtro intercetta le risposte in uscita prima che vengano restituite ai client. Può essere utilizzato per aggiungere intestazioni HTTP, trasformare la risposta, eseguire operazioni di logging e altro ancora.

Ecco come funziona il ciclo di vita di un filtro in JAX-RS.

1. **Registrazione del filtro**: il filtro viene registrato all'interno dell'applicazione JAX-RS utilizzando l'annotazione `@Provider`. Questo indica al framework JAX-RS che il componente è un filtro e deve essere gestito come tale.
2. **Fase di invocazione del filtro**: quando una richiesta viene ricevuta dall'applicazione JAX-RS, il framework instrada la richiesta ai filtri di richiesta registrati prima di passarla al resource endpoint corrispondente. Allo stesso modo, quando viene generata una risposta dall'applicazione, il framework instrada la risposta ai filtri di risposta registrati prima di restituirla al client.
3. **Esecuzione del filtro**: ogni filtro ha la possibilità di eseguire le proprie operazioni durante l'invocazione. Un filtro può accedere alla richiesta o alla risposta, eseguire le operazioni necessarie e, se necessario, interrompere il flusso delle richieste o delle risposte.
4. **Chiamata al filtro successivo**: dopo che un filtro ha completato le sue operazioni, il framework JAX-RS continua a passare la richiesta o la risposta al filtro successivo nella catena dei filtri.
5. **Restituzione al client**: dopo che tutte le operazioni dei filtri sono state completate, il framework restituisce la risposta finale al client.

In breve, i filtri in JAX-RS forniscono un meccanismo potente per eseguire operazioni di pre-elaborazione e post-elaborazione sulle richieste e sulle risposte dei servizi, consentendo un maggiore controllo e flessibilità nel gestire il flusso.

Dato che un diagramma è più efficace di mille parole, consiglio di fare riferimento all'Appendice Processing Pipeline delle specifiche Jakarta RESTful Web Services 3.1.0.

Se avete visto il diagramma della *Processing Pipeline*, il nostro interesse è il lato server (non il client) e il diagramma a seguire mostra i cosiddetti **punti di estensione**. Il nostro lavoro consiste nell'implementazione delle due interfacce `ContainerRequestFilter` e `ContainerResponseFilter` che

intercettano le richieste in arrivo e le risposte in uscita rispettivamente. Un punto da tenere in considerazione è l'ordine di esecuzione dei filtri che dipende dalla priorità loro assegnata (per maggiori dettagli fare riferimento Priorities delle specifiche Jakarta RESTful Web Services).

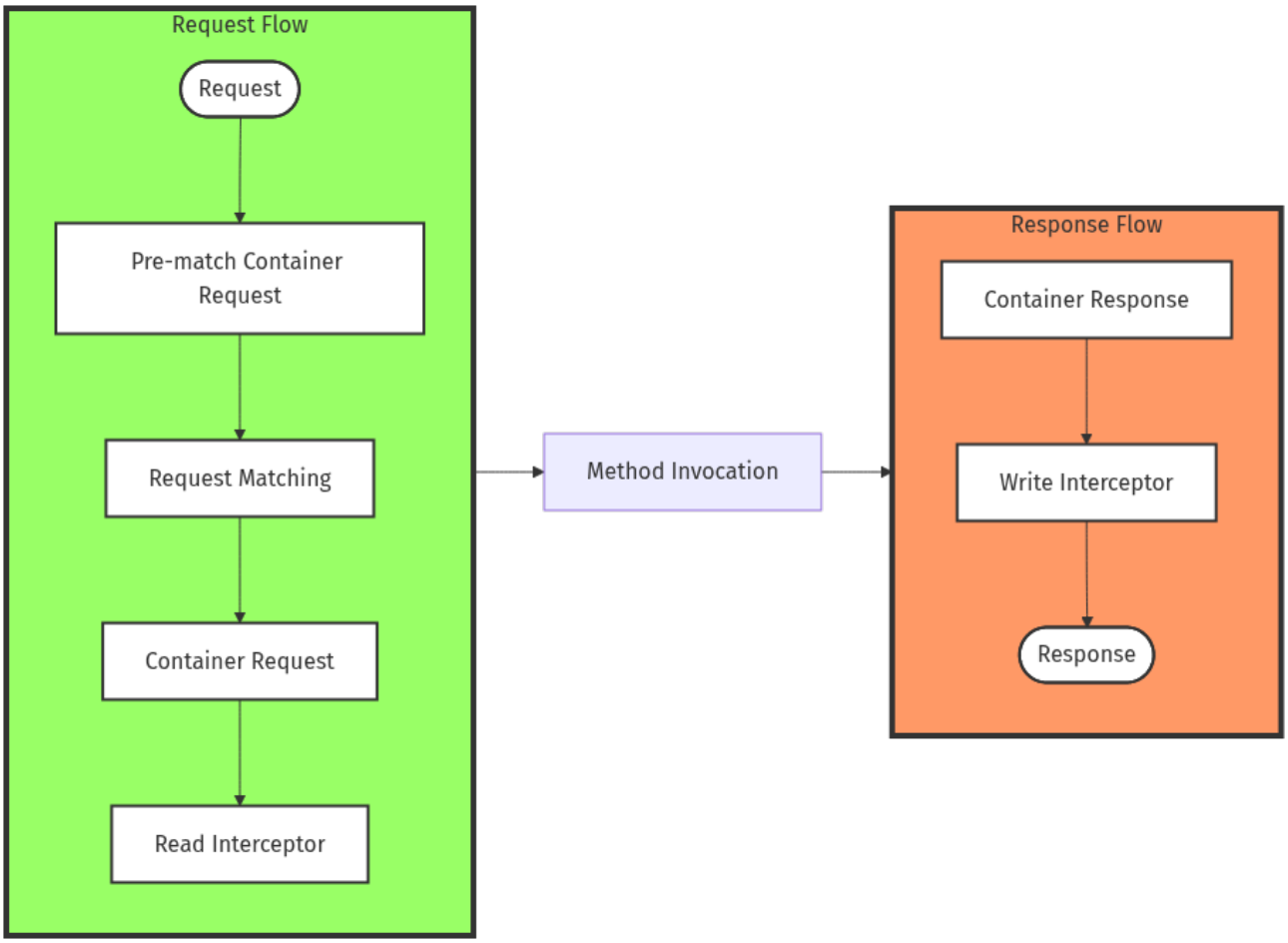


Diagramma 6 - *Punti di estensione del filtro e dell'interceptor lato server*

Il diagramma di classe a seguire mostra la classe `TraceJaxRsRequestResponseFilter` (il nostro filtro JAX-RS) che implementa le due interfacce `ContainerRequestFilter` e `ContainerResponseFilter` e in particolare i due metodi `filter()`.

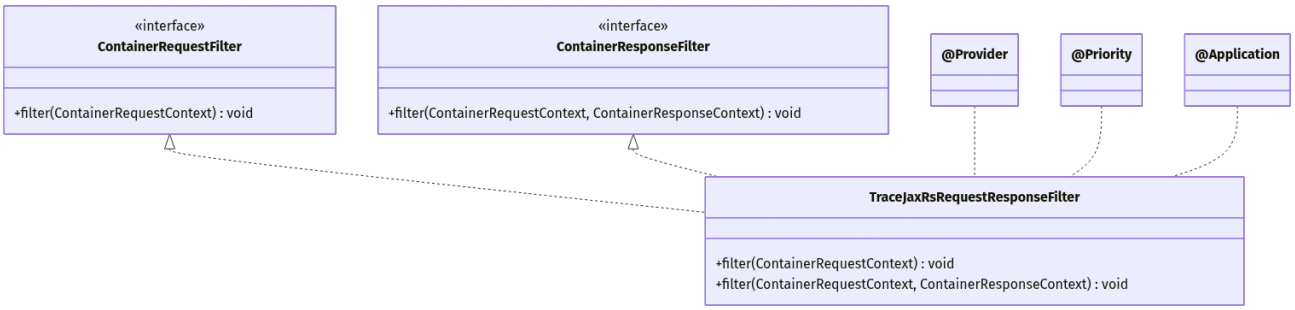


Diagramma 7 - *Relazioni del filtro JAX-RS TraceJaxRsRequestResponseFilter*

Il corpo dei due metodi `filter()` deve contenere la logica necessaria per prepare i messaggi (di request e response) da pubblicare sull'Event Bus di Quarkus.

Il codice del filtro `TraceJaxRsRequestResponseFilter` mostrato a seguire, riporta l'implementazione parziale dei due metodi; manca la parte di codice responsabile della preparazione e invio dei messaggi di richiesta e risposta sull'Event Bus che vedremo successivamente.

Code 3 - *Classe TraceJaxRsRequestResponseFilter che implementa il filtro JAX-RS*

```
package it.dontesta.eventbus.ws.filter;

import jakarta.annotation.Priority;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.ws.rs.Priorities;
import jakarta.ws.rs.container.ContainerRequestContext;
import jakarta.ws.rs.container.ContainerRequestFilter;
import jakarta.ws.rs.container.ContainerResponseContext;
import jakarta.ws.rs.container.ContainerResponseFilter;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.UriInfo;
import jakarta.ws.rs.ext.Provider;
import java.util.List;
import java.util.UUID;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;

@Provider
@Priority(Priorities.USER)
public class TraceJaxRsRequestResponseFilter implements ContainerRequestFilter,
    ContainerResponseFilter {

  @Inject
  Logger log;

  @Context
  UriInfo uriInfo;

  @ConfigProperty(name = "app.filter.enabled", defaultValue = "false")
  boolean filterEnabled;

  @ConfigProperty(name = "app.filter.uris")
  List<String> uris;

  private static final String CORRELATION_ID_HEADER = "X-Correlation-ID";

  @Override
  public void filter(ContainerRequestContext requestContext) {
    // Se il filtro non è abilitato, esci
    if (!filterEnabled) {
      return;
    }

    // Ottieni l'URI della richiesta
    String requestUri = uriInfo.getRequestUri().getPath();
```

```
    String correlationId = getCorrelationId(requestContext.getHeaderString
(CORRELATION_ID_HEADER));

    // Aggiungi l'ID di correlazione alla richiesta
    requestContext.setProperty(CORRELATION_ID_HEADER, correlationId);

    // Applica la logica del filtro in base all'URI
    if (requestUriIsFiltered(requestUri)) {
      /*
       @TODO: Se l'URI richiesto è presente nell'elenco delle URI da filtrare
       * prepara e invia il messaggio della richiesta verso la destinazione
       * dell'Event Bus.
       */
      log.debug("Pubblicazione del messaggio della richiesta HTTP sull'Event Bus");
    }
  }

  @Override
  public void filter(ContainerRequestContext requestContext,
                     ContainerResponseContext responseContext) {
    // Se il filtro non è abilitato, esci
    if (!filterEnabled) {
      return;
    }

    // Ottieni l'URI della richiesta
    String requestUri = uriInfo.getRequestUri().getPath();

    // Recupera l'ID di correlazione dalla richiesta
    String correlationId =
        getCorrelationId((String) requestContext.getProperty(
CORRELATION_ID_HEADER));

    // Aggiungi l'ID di correlazione alla risposta
    responseContext.getHeaders().add(CORRELATION_ID_HEADER, correlationId);

    // Applica la logica del filtro in base all'URI
    if (requestUriIsFiltered(requestUri)) {
      /*
       @TODO: Se l'URI richiesto è presente nell'elenco delle URI da filtrare
       * prepara e invia il messaggio della richiesta verso la destinazione
       * dell'Event Bus.
       */
      log.debug("Pubblicazione del messaggio della risposta HTTP sull'Event Bus");
    }
  }

  /**
   * Ottiene l'ID di correlazione.
```

```
    * Se l'ID di correlazione è nullo, genera un nuovo ID di correlazione.
    * Questo metodo è utilizzato per garantire che l'ID di correlazione sia sempre
presente,
    * sia nella richiesta che nella risposta. Il formato dell'ID di correlazione è un
UUID.
    *
    * @param correlationId L'ID di correlazione
    * @return L'ID di correlazione
    */
   private String getCorrelationId(String correlationId) {
     // Genera un nuovo ID di correlazione se quello attuale è nullo
     if (correlationId == null) {
       correlationId = UUID.randomUUID().toString();
     }
     return correlationId;
   }

   /**
    * Verifica se la Request URI è tra quelle che devono essere filtrate.
    * Il parametro di configurazione app.filter.uris contiene l'elenco delle URI.
    *
    * @param requestUri La Request URI da verificare
    * @return true se la Request URI è tra quelle che devono essere filtrate, false
altrimenti
    */
   private boolean requestUriIsFiltered(String requestUri) {
     log.debug("La Request URI %s è tra quelle che devono essere filtrate".formatted
(requestUri));

     return uris.stream().anyMatch(item -> requestUri.startsWith(item));
   }
}
```

Adesso che abbiamo implementato il filtro JAX-RS (anche se completo in parte), siamo nelle condizioni di poter eseguire un test. Guardando con attenzione l'implementazione attuale, quali sono i punti salienti?

1. Il filtro è influenzato da due parametri di configurazione.
   a. Il parametro `app.filter.enabled` consente di abilitare o disabilitare il processo di elaborazione delle richieste e risposte.
   b. Il parametro `app.filter.uris` consente di specificare quali siano le URI che devono essere sottoposte a operazioni di filtraggio.
2. Il filtro fa in modo di generare il cosiddetto ***correlationId*** che consente di legare richiesta e risposta. Questo viene poi impostato come valore dell'http header (custom) X-Correlation-ID.

A seguire è possibile vedere i parametri di configurazione menzionati in precedenza e i rispettivi valori. In particolare il parametro `app.filter.uris` è difatti un array che al momento contiene un solo elemento.

Configurazione 2 - *Contenuto del file di configurazione application.properties*

```
# The path of the banner (path relative to root of classpath) which could be
provided by user
quarkus.banner.path=quarkus-banner.txt

# Logging configuration
quarkus.log.category."it.dontesta.eventbus.ws.filter.TraceJaxRsRequestResponseFilter
".level=DEBUG

##
## Application configuration properties
##

# Enable or disable the JX-RS filter. Default is true.
app.filter.enabled=true

# The URIs that the filter should be applied to.
app.filter.uris[0]=/api/rest
```

Eseguendo un test allo stato attuale dell'implementazione del progetto, quale dovrebbe essere il risultato atteso?

Effettuando una chiamata verso l'endpoint http://localhost:8080/api/rest/echo, dovremmo vedere tra gli header HTTP di risposta quello il cui nome è X-Correlation-ID e il valore è una stringa in formato UUID Type 4.

Console 8 - *Esecuzione del test sul filtro JAX-RS implementato*

```
# Esecuzione del test sul filtro JAX-RS implementato
# eseguendo la chiamata verso l'endpoint http://localhost:8080/api/rest/echo
curl -v \
  -H "Content-Type: application/json" \
  -d '{"message": "Primo test del filtro JAX-RS che aggiunge l'\''header X-
Correlation-ID"}' \
http://localhost:8080/api/rest/echo
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080
> POST /api/rest/echo HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 82
>
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
< content-length: 82
< X-Correlation-ID: 5f106bed-2d6a-40de-8936-d1644c1c5b85
<
* Connection #0 to host localhost left intact
{"message": "Primo test del filtro JAX-RS che aggiunge l'header X-Correlation-ID"}
```

Sul repository GitHub del progetto è disponibile lo unit test `CorrelationIdHttpHeaderTest` che si accerta che a fronte di una chiamata verso l'endpoint `/api/rest/echo` di ottenere in risposta l'header X-Correlation-ID il cui contenuto sia una stringa in formato UUID Type 4.

7. Aggiornamento v1.4.0 — Filtro annotation-based e pattern capture-only

Quanto descritto in questa sezione si riferisce alla versione **1.4.0** del progetto software.

A partire dalla versione **1.4.0**, il filtro `TraceJaxRsRequestResponseFilter` è stato riscritto adottando due importanti cambiamenti architetturali.

Annotazioni `@ServerRequestFilter` / `@ServerResponseFilter`

Le interfacce `ContainerRequestFilter` e `ContainerResponseFilter` (con `@Provider` e `@Priority`) sono state sostituite dalle annotazioni RESTEasy Reactive `@ServerRequestFilter` e `@ServerResponseFilter`. Questo approccio consente l'iniezione diretta di parametri Vert.x/RESTEasy Reactive (es. `RoutingContext`) senza necessità di registrazione esplicita via `@Provider`.

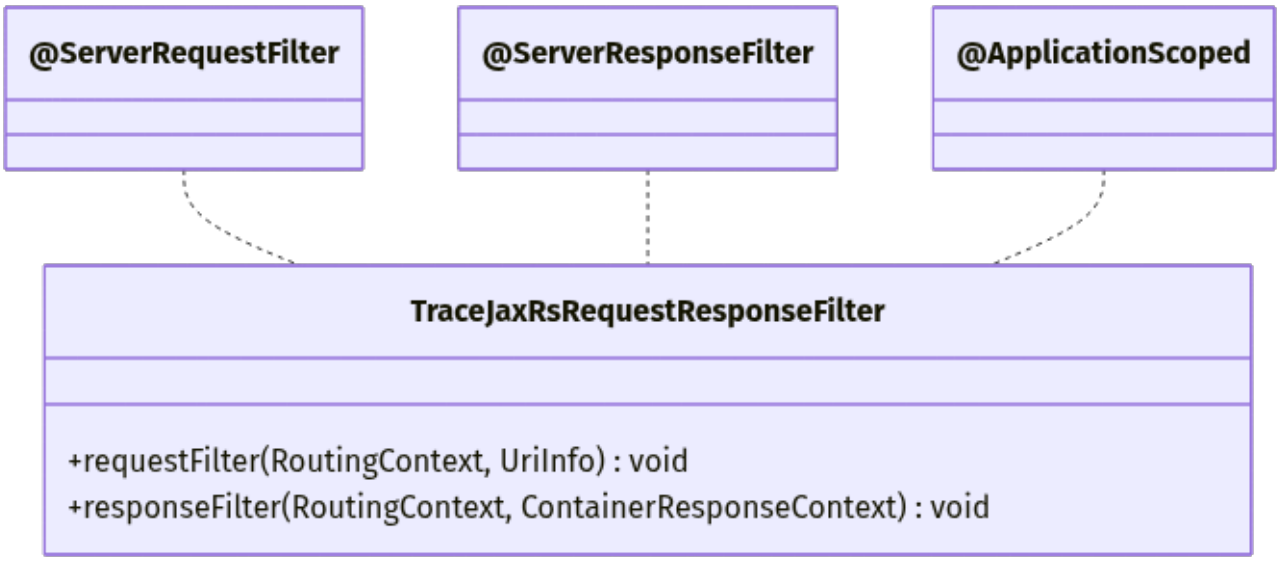


Diagramma 7b - *TraceJaxRsRequestResponseFilter con annotazioni RESTEasy Reactive (v1.4.0)*

Pattern capture-only

Il filtro adotta ora il pattern **capture-only**: invece di costruire il JSON e chiamare `EventBus.publish()` direttamente, il filtro legge il body e accoda record leggeri (`RequestTrace` / `ResponseTrace`, costituiti da soli tipi primitivi/String) in O(1) verso il `TraceEventDispatcher`. Tutta la costruzione JSON, serializzazione e pubblicazione sull'Event Bus sono delegate al dispatcher, eliminando qualsiasi impatto sul ciclo di vita HTTP.

Code 3b - *Struttura semplificata del filtro v1.4.0 con pattern capture-only*

```
@ApplicationScoped
public class TraceJaxRsRequestResponseFilter {
```

```
  @Inject
  TraceEventDispatcher dispatcher;

  @ServerRequestFilter
  public void requestFilter(RoutingContext routingContext, UriInfo uriInfo) {
    if (!filterEnabled) return;
    if (!requestUriIsFiltered(uriInfo.getRequestUri().getPath())) return;

    String correlationId = getOrCreateCorrelationId(routingContext);
    String body = readBodyAsync(routingContext);           // lettura body async
    dispatcher.enqueueRequest(new RequestTrace(...));      // O(1) – nessuna
serializzazione
  }

  @ServerResponseFilter
  public void responseFilter(RoutingContext routingContext,
                             ContainerResponseContext responseContext) {
    if (!filterEnabled) return;
    dispatcher.enqueueResponse(new ResponseTrace(...));    // O(1)
  }
}
```

Per i dettagli sull'implementazione di `TraceEventDispatcher` fare riferimento alla sezione *Realizzazione del Dispatcher e Event Handler*.

7.1. Definizione degli indirizzi virtuali dell'Event Bus

In questo quarto step, il tag di riferimento è step-4. Affinché sia possibile consumare i messaggi/eventi pubblicati sul Event Bus di Quarkus, è necessario definire i cosiddetti virtual address che nel nostro caso sono sei.

1. L'indirizzo per cui registrare il consumer che processerà i messaggi/eventi per le richieste JAX-RS.
2. L'indirizzo per cui registrare il consumer che processerà i messaggi/eventi per le risposte JAX-RS.
3. L'indirizzo per cui registrare il consumer che processerà i messaggi/eventi che devono essere tracciati su un database SQL.
4. L'indirizzo per cui registrare il consumer che processerà i messaggi/eventi che devono essere tracciati su un database NoSQL.
5. L'indirizzo per cui registrare il consumer che processerà i messaggi/eventi che devono essere tracciati su una coda AMQP.
6. L'indirizzo per cui registrare il Dispatcher che sarà responsabile di inoltrare i messaggi/eventi ricevuti verso il corretto gestore (Event Handler) e quest'ultimo sarà l'effettivo responsabile dello store sul database SQL, NoSQL o sulla coda AMQP.

La documentazione di Quarkus Using Event Bus fa uso dell'annotazione `@ConsumeEvent` dov'è possibile specificare l'indirizzo virtuale in questo modo: `@ConsumeEvent("nome-indirizzo-virtuale")`. Nel caso in cui questo non fosse specificato, l'indirizzo virtuale di default è il *fully qualified name* del nome del bean che implementa il consumer.

In questo progetto ho voluto usare un diverso approccio, ovvero, definire gli indirizzi virtuali sulla configurazione dell'applicazione e poi gestire in modo fine la gestione della registrazione dei consumer. A seguire è mostrata la sezione della configurazione dei sei indirizzi virtuali sul file `application.properties`.

Configurazione 3 - *Configurazione degli indirizzi virtuali dell'Event Bus su cui registrare i relativi consumer*

```
# Define the Event Bus virtual address for the HTTP request event
app.eventbus.consumer.http.request.address=http-request

# Define the Event Bus virtual address for the HTTP response event
app.eventbus.consumer.http.response.address=http-response

# Define the Event Bus virtual address for event handler SQL, NoSQL and Queue
app.eventbus.consumer.event.handler.addresses[0]=sql-trace
app.eventbus.consumer.event.handler.addresses[1]=nosql-trace
app.eventbus.consumer.event.handler.addresses[2]=queue-trace

# Define the Event Bus virtual address for the Dispatcher
app.eventbus.consumer.dispatcher.address=dispatcher-event
```

7.2. Registrazione dei Consumer per gli eventi dell'Event Bus

In questo quinto step, il tag di riferimento è step-5.3. Una volta definiti gli indirizzi virtuali per la nostra applicazione Quarkus, vediamo come implementare la registrazione dei consumer e il metodo da utilizzare per realizzare la logica di elaborazione del messaggio ricevuto dal consumer stesso.

Prenderemo in esame la registrazione di un solo consumer perché il processo resta invariato anche per il resto dei consumer. Il consumer in questione è quello che sarà sottoscritto all'indirizzo virtuale `http-request`. A seguire il codice del consumer `HttpRequestConsumer`.

Code 4 - *Implementazione del consumer HttpRequestConsumer e registrazione sull'Event Bus*

```
package it.dontesta.eventbus.consumers.http;

import io.quarkus.runtime.StartupEvent;
import io.vertx.core.json.JsonObject;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;

@ApplicationScoped
public class HttpRequestConsumer {

  @Inject
  EventBus eventBus;

  @Inject
  Logger log;

  @ConfigProperty(name = "app.eventbus.consumer.http.request.address")
  String httpRequestVirtualAddress;

  void onStart(@Observes StartupEvent ev) {
    log.debugf(
        "Registering the consumers to the event bus for HTTP request at addresses:
{%s}",
        httpRequestVirtualAddress);

    eventBus.consumer(httpRequestVirtualAddress, this::handleEvent);
  }

  // Method to handle the event
  public void handleEvent(Message<JsonObject> message) {
    log.debug("Received HTTP request message: " + message.body());
  }
}
```

Analizzando il codice del consumer, attraverso l'annotazione `@ConfigProperty` è letto il valore dell'indirizzo virtuale dalla proprietà di configurazione `app.eventbus.consumer.http.request.address`.

Il metodo `onStart()` esegue la registrazione del consumer sull'Event Bus attraverso il metodo `eventBus.consumer()` specificando l'indirizzo virtuale e il metodo che sarà responsabile dell'elaborazione dei messaggi. La fase di registrazione avviene nel momento in cui si avvierà l'applicazione (vedi l'annotazione `@Obeserves` e per approfondimenti Listening for startup and shutdown events).

Nota: Così come il consumer è stato registrato, è importante assicurarsi che i consumer vengano de-registrati correttamente per evitare perdite di memoria o altri problemi che potrebbero verificarsi se i consumer rimangono registrati sull'Evento Bus anche dopo che l'applicazione è stata arrestata. Per fare ciò, è possibile utilizzare il metodo `eventBus.consumer().unregister()` all'interno del metodo `onStop()` che è responsabile della terminazione dell'applicazione. Quando è utilizzata l'annotazione `@ConsumeEvent`, Quarkus si occupa automaticamente della registrazione e della de-registrazione dei consumer. Fare riferimento al tag step-5.3.1 per vedere come implementare la de-registrazione dei consumer (`HttpRequestConsumer`).

Il metodo `handleEvent()` al momento contiene solamente un'istruzione di log allo scopo di verificare che questo sia correttamente chiamato. Vedremo l'implementazione completa più avanti e nel frattempo eseguiamo un test allo scopo di verificare che:

1. allo start dell'applicazione il consumer venga correttamente registrato sull'Event Bus;
2. eseguendo lo unit test `PublishMessageOnEventBusTest#testPublishMessageOnEventBus` dovreste poter vedere se il messaggio sia pubblicato correttamente attraverso l'Event Bus e che questo sia consumato verificando che sia presente in console il log indicato sul metodo `handleEvent()`.

Avviando l'applicazione con il comando `quarkus dev` o `mvn clean quarkus:dev` dovreste vedere i log evidenziati in figura 4 che riguardano in particolare la registrazione dei due consumer `HttpRequestConsumer` e `HttpResponseConsumer`.

```
Listening for transport dt_socket at address: 5005
__________.__                 __________                 .___          .__  __
\_   ___/|  |__   ____   ____ \_____   \ ____   __| _/_____     |__|/  |_
  |    |   |  |  \_/ __ \   |       _// __ \ / __ | /     \   /  _ \ / __ |/ __ \     |  \   __\
  |    |   |   Y  \  ___/   |    |   \  ___// /_/ | \     \___(  <_> ) /_/ \  ___/     |  ||  |
  |____|   |___|  /\___  >  |____|_  /\___  >____ |  \______  /\____/\____ |\___  > /\ |__||__|
                \/     \/         \/     \/     \/         \/            \/     \/  \/
                                                                Powered by Quarkus 3.9.2
2024-04-10 16:14:26,280 DEBUG [it.don.eve.con.htt.HttpResponseConsumer] (Quarkus Main Thread) Registering the consumers to the event bus for HT
TP response at addresses: {http-response}

2024-04-10 16:14:26,285 DEBUG [it.don.eve.con.htt.HttpRequestConsumer] (Quarkus Main Thread) Registering the consumers to the event bus for HTT
P request at addresses: {http-request}
2024-04-10 16:14:26,334 INFO  [io.quarkus] (Quarkus Main Thread) eventbus-logging-filter-jaxrs 1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.9.2)
 started in 1.120s. Listening on: http://localhost:8080
2024-04-10 16:14:26,334 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2024-04-10 16:14:26,335 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, hibernate-validator, rest, rest-jackson, smallrye-co
ntext-propagation, vertx]
```

Figura 4 - *Avvio dell'applicazione Quarkus con evidenza dei log di registrazione dei consumer*

Adesso passiamo alla seconda verifica descritta in precedenza eseguendo in console il comando `mvn test -Dtest=PublishMessageOnEventBusTest#testPublishMessageOnEventBus`. Dopo aver eseguito il comando, in console saranno mostrati i messaggi pubblicati sull'Event Bus (dallo unit test) e ricevuti dai due consumer `HttpRequestConsumer` e `HttpResponseConsumer`.

Console 9 - *Log di pubblicazione messaggio sull'Event Bus e consumazione da parte dei consumer*

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running it.dontesta.eventbus.publish.PublishMessageOnEventBusTest
2024-04-11 16:59:36,872 DEBUG [it.don.eve.con.htt.HttpRequestConsumer] (main)
Registering the consumers to the event bus for HTTP request at addresses: {http-
request}
2024-04-11 16:59:36,876 DEBUG [it.don.eve.con.htt.HttpResponseConsumer] (main)
Registering the consumers to the event bus for HTTP response at addresses: {http-
response}
2024-04-11 16:59:36,930 INFO  [io.quarkus] (main) eventbus-logging-filter-jaxrs
1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.9.2) started in 1.228s. Listening on:
http://localhost:8081
2024-04-11 16:59:36,930 INFO  [io.quarkus] (main) Profile test activated.
2024-04-11 16:59:36,930 INFO  [io.quarkus] (main) Installed features: [cdi,
hibernate-validator, rest, rest-jackson, smallrye-context-propagation, vertx]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.707 s -- in
it.dontesta.eventbus.publish.PublishMessageOnEventBusTest
2024-04-11 16:59:37,163 DEBUG [it.don.eve.con.htt.HttpRequestConsumer] (vert.x-
eventloop-thread-0) Received HTTP request message: {"message":"Message to publish on
the event bus {virtualAddress: http-request}"}
2024-04-11 16:59:37,164 DEBUG [it.don.eve.con.htt.HttpResponseConsumer] (vert.x-
eventloop-thread-0) Received HTTP response message: {"message":"Message to publish
on the event {virtualAddress: http-response}"}
2024-04-11 16:59:37,173 INFO  [io.quarkus] (main) eventbus-logging-filter-jaxrs
stopped in 0.085s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.389 s
[INFO] Finished at: 2024-04-11T16:59:37+02:00
[INFO] ------------------------------------------------------------------------
```

7.3. Adeguamento del filtro JAX-RS per inviare l'evento/messaggio sull'Event Bus

In questo sesto step, il tag di riferimento è step-6. Dopo aver accertato il corretto funzionamento dell'invio dei messaggi sull'Event Bus e la corretta ricezione di questi da parte dei consumer, è arrivato il momento di completare il filtro JAX-RS per prepare i messaggi da inviare ai due consumer `HttpRequestConsumer` e `HttpResponseConsumer`).

Il messaggio da pubblicare sarà un oggetto di tipo `io.vertx.core.json.JsonObject` e conterrà una serie d'informazioni che riguardano sia la request sia la response http.

Il motivo per cui è stato scelto di usare oggetti di tipo `io.vertx.core.json.JsonObject` risiede nel fatto che questo tipo di oggetto può essere inviato direttamente sull'Event Bus e non necessita di uno specifico codec di Vert.x.

Quanto mostrato a seguire, riporta una parte del codice del filtro JAX-RS `TraceJaxRsRequestResponseFilter` opportunamente modificato per preparare il messaggio JSON (vedi riga 27 `JsonObject prepareMessage(ContainerRequestContext requestContext)`) da pubblicare poi sull'Event Bus (vedi riga 11 `eventBus.publish(httpRequestVirtualAddress, prepareMessage(requestContext))`).

Code 5 - *Modifiche al filtro JAX-RS* `TraceJaxRsRequestResponseFilter` *per preparare il messaggio e pubblicarlo sull'Event Bus*

```
...
      if (requestUriIsFiltered(requestUri)) {
      // Aggiungi la data ora di quando la richiesta arriva al filtro
      requestContext.setProperty(LOCAL_DATE_TIME_IN, LocalDateTime.now());

      /*
       * Se l'URI richiesto è presente nell'elenco delle URI da filtrare
       * prepara e invia il messaggio della richiesta verso la destinazione
       * dell\'Event Bus.
       */
      eventBus.publish(httpRequestVirtualAddress, prepareMessage(requestContext));

      log.debug("Pubblicazione del messaggio della richiesta HTTP su Event Bus");
    }

  /**
   * Prepara il messaggio della richiesta in formato JSON per l'invio all'Event Bus.
   *
   * <p>Il messaggio restituito contiene le informazioni relative alla richiesta
HTTP
   * come URI, headers, corpo, metodo, media-type, lingua accettata, ecc.
   *
   * <p>È in formato {@link JsonObject} per essere inviato direttamente all'Event
Bus.
```

```
    *
    * @param requestContext Il contesto della richiesta
    * @return JsonObject Il messaggio della richiesta in formato JSON
    */
   private JsonObject prepareMessage(ContainerRequestContext requestContext) {
     JsonObject jsonObject;
     try {
       jsonObject = new JsonObject()
           .put(CORRELATION_ID_HEADER, requestContext.getProperty
(CORRELATION_ID_HEADER))
           .put("remote-ip-address", routingContext.request().remoteAddress().host())
           .put("headers", requestContext.getHeaders())
           .put("body", getRequestBody(requestContext))
           .put("uri-info", requestContext.getUriInfo().getRequestUri().toString())
           .put(LOCAL_DATE_TIME_IN, requestContext.getProperty(LOCAL_DATE_TIME_IN
).toString())
           .put("method", requestContext.getMethod())
           .put("media-type", "%s/%s".formatted(requestContext.getMediaType().
getType(),
               requestContext.getMediaType().getSubtype()))
           .put("acceptable-language", requestContext.getAcceptableLanguages
().toString())
           .put("acceptable-media-types", requestContext.getAcceptableMediaTypes
().toString());
     } catch (IOException ioException) {
       log.error("Errore nella generazione del JSON dal requestContext object");
       throw new RuntimeException(ioException);
     }

     return jsonObject;
   }
...
```

A questo punto abbiamo completato l'implementazione dei componenti richiesti affinché il sequence diagram mostrato in diagramma 4 sia soddisfatto. Possiamo eseguire un test per verificare che quanto sviluppato fino a questo momento funzioni utilizzando due metodi:

1. il primo consiste nell'avviare l'applicazione e fare una richiesta cURL verso l'endpoint `/api/rest/echo` verificando sui log che in messaggi pubblicati sull'Event Bus siano consumanti dai due consumer `HttpRequestConsumer` e `HttpResponseConsumer`;
2. il secondo test consiste nell'esecuzione degli unit test utilizzando il comando `quarkus test` o `mvn test`.

La figura a seguire e il JSON mostrato successivamente (console 10), rappresentano il messaggio pubblicato sull'Event Bus e consumato dai due consumer `HttpRequestConsumer` e `HttpResponseConsumer`.

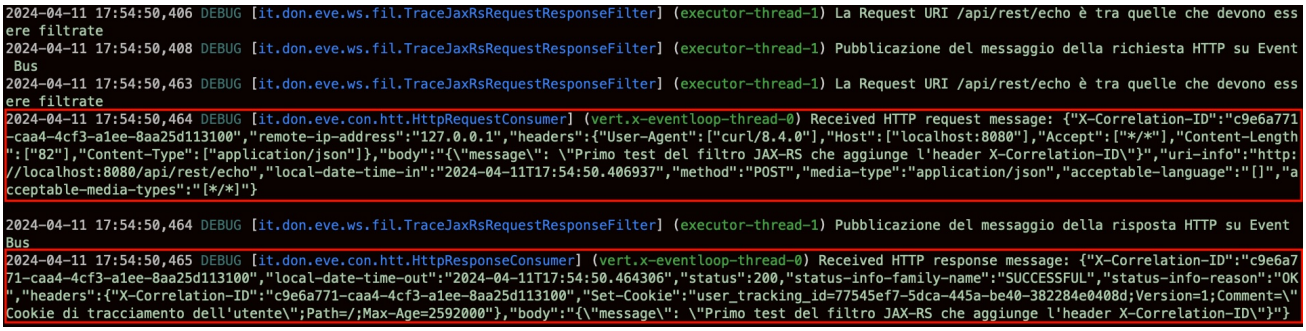


Figura 5 - *Test della generazione dei messaggi e pubblicazione*

Console 10 - *Esempio dei messaggi di request e response pubblicati sull'Event Bus e consumati da HttpRequestConsumer e HttpResponseConsumer*

```
// Messaggio in formato JSON generato dal filtro JAX-RS sulla catena di request
// e successivamente pubblicato sull'Event Bus verso il virtual address configurato
{
  "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
  "remote-ip-address": "127.0.0.1",
  "headers": {
    "User-Agent": [
      "curl/8.4.0"
    ],
    "Host": [
      "localhost:8080"
    ],
    "Accept": [
      "*/*"
    ],
    "Content-Length": [
      "82"
    ],
    "Content-Type": [
      "application/json"
    ]
  },
  "body": "{\"message\": \"Primo test del filtro JAX-RS che aggiunge l'header X-
Correlation-ID\"}",
  "uri-info": "http://localhost:8080/api/rest/echo",
  "local-date-time-in": "2024-04-11T18:01:48.467365",
  "method": "POST",
  "media-type": "application/json",
  "acceptable-language": "[]",
  "acceptable-media-types": "[*/*]"
}

// Messaggio in formato JSON generato dal filtro JAX-RS sulla catena di response
// e successivamente pubblicato sull'Event Bus verso il virtual address configurato

{
```

```
    "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
    "local-date-time-out": "2024-04-11T18:01:48.528893",
    "status": 200,
    "status-info-family-name": "SUCCESSFUL",
    "status-info-reason": "OK",
    "headers": {
        "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
        "Set-Cookie": "user_tracking_id=27765cdb-0d24-489f-b8cb-ae5b635a4f74;
Version=1;Comment=\"Cookie di tracciamento dell'utente\";Path=/;Max-Age=2592000"
    },
    "body": "{\"message\": \"Primo test del filtro JAX-RS che aggiunge l'header X-
Correlation-ID\"}"
}
```

8. Aggiornamento v1.4.0 — Nuove proprietà di configurazione del dispatcher

Quanto descritto in questa sezione si riferisce alla versione **1.4.0** del progetto software.

Con la migrazione al pattern **capture-only** e al nuovo `TraceEventDispatcher` (vedere la sezione successiva), il file `application.properties` si arricchisce di due nuove proprietà di configurazione che controllano il comportamento della coda bounded e del thread di drain.

Configurazione 2b - *Nuove proprietà per il TraceEventDispatcher (v1.4.0)*

```
# Intervallo di sleep (ms) del thread di drain quando entrambe le code sono vuote.
# Default: 20 ms
app.filter.dispatcher.interval.ms=20

# Capacità massima di ciascuna coda bounded (ArrayBlockingQueue).
# Gli eventi che arrivano a coda piena vengono scartati con log WARN.
# Default: 5000
app.filter.dispatcher.queue.capacity=5000
```

La proprietà `app.filter.dispatcher.interval` (stringa ISO-8601 usata con `@Scheduled`) è stata rinominata in `app.filter.dispatcher.interval.ms` (intero in millisecondi). Se avete configurazioni personalizzate basate sulla versione precedente, aggiornare il nome della proprietà.

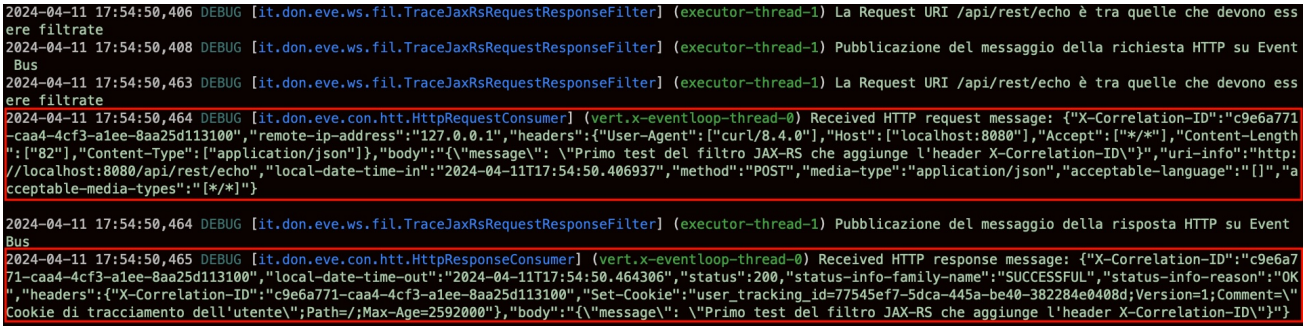


Figura 5 - *Test della generazione dei messaggi e pubblicazione*

Console 10 - *Esempio dei messaggi di request e response pubblicati sull'Event Bus e consumati da HttpRequestConsumer e HttpResponseConsumer*

```
// Messaggio in formato JSON generato dal filtro JAX-RS sulla catena di request
// e successivamente pubblicato sull'Event Bus verso il virtual address configurato
{
  "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
  "remote-ip-address": "127.0.0.1",
  "headers": {
```

```
    "User-Agent": [
      "curl/8.4.0"
    ],
    "Host": [
      "localhost:8080"
    ],
    "Accept": [
      "*/*"
    ],
    "Content-Length": [
      "82"
    ],
    "Content-Type": [
      "application/json"
    ]
  },
  "body": "{\"message\": \"Primo test del filtro JAX-RS che aggiunge l'header X-
Correlation-ID\"}",
  "uri-info": "http://localhost:8080/api/rest/echo",
  "local-date-time-in": "2024-04-11T18:01:48.467365",
  "method": "POST",
  "media-type": "application/json",
  "acceptable-language": "[]",
  "acceptable-media-types": "[*/*]"
}

// Messaggio in formato JSON generato dal filtro JAX-RS sulla catena di response
// e successivamente pubblicato sull'Event Bus verso il virtual address configurato

{
    "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
    "local-date-time-out": "2024-04-11T18:01:48.528893",
    "status": 200,
    "status-info-family-name": "SUCCESSFUL",
    "status-info-reason": "OK",
    "headers": {
        "X-Correlation-ID": "e9c2a4d3-27b9-4ceb-830e-4f293307bc5a",
        "Set-Cookie": "user_tracking_id=27765cdb-0d24-489f-b8cb-ae5b635a4f74;
Version=1;Comment=\"Cookie di tracciamento dell'utente\";Path=/;Max-Age=2592000"
    },
    "body": "{\"message\": \"Primo test del filtro JAX-RS che aggiunge l'header X-
Correlation-ID\"}"
}
```

8.1. Realizzazione del Dispatcher e Event Handler

Siamo quasi al termine! Non rimane altro che implementare gli ultimi componenti, ovvero, il Dispatcher e gli Event Handler. In questo settimo step, il tag finale di riferimento è step-7.1.3.

Iniziamo con il Dispatcher. Questo componente è sempre un consumer registrato sull'Event Bus sull'indirizzo virtuale definito in `app.eventbus.consumer.dispatcher.address` sul file di configurazione `application.properties` dell'applicazione Quarkus.

Il Dispatcher conosce gli indirizzi virtuali degli Event Handler a cui inoltrare i messaggi ricevuti leggendo gli **header** dal messaggio che ha ricevuto. A seguire l'estratto di codice del consumer `HttpRequestConsumer` revisionato per includere le informazioni per il Dispatcher. La stessa revisione è stata fatta per il consumer `HttpResponseConsumer`.

Rispetto alla precedente versione del consumer, è stata aggiunta la lettura dell'indirizzo virtuale del Dispatcher dalla configurazione `app.eventbus.consumer.dispatcher.address` e la lista degli Event Handler dalla configurazione `app.eventbus.consumer.event.handler.addresses` .

È stato creato inoltre l'oggetto `DeliveryOptions` per specificare le opzioni di consegna del messaggio e in particolare gli header `SOURCE_VIRTUAL_ADDRESS`, `SOURCE_COMPONENT` e `TARGET_VIRTUAL_ADDRESSES`.

L'header d'interesse per il Dispatcher è `TARGET_VIRTUAL_ADDRESSES` che contiene l'elenco degli indirizzi virtuali degli Event Handler a cui inoltrare il messaggio. Il Dispatcher legge questo header e inoltra il messaggio a tutti gli Event Handler specificati; le informazioni presenti sull'header del messaggio sono quindi necessarie per applicare "regole di routing" ai messaggi da inoltrare.

Code 6 - *Revisione del consumer* `HttpRequestConsumer` *per includere le informazioni per il Dispatcher*

```
...
@ApplicationScoped
public class HttpRequestConsumer {

  @ConfigProperty(name = "app.eventbus.consumer.dispatcher.address")
  String dispatcherVirtualAddress;

  @ConfigProperty(name = "app.eventbus.consumer.event.handler.addresses")
  List<String> eventHandlerVirtualAddresses;

  public static final String SOURCE_VIRTUAL_ADDRESS = "source-virtual-address";

  public static final String SOURCE_COMPONENT = "source-component";

  public static final String TARGET_VIRTUAL_ADDRESSES = "target-virtual-addresses";

  void onStart(@Observes StartupEvent ev) {
    ...
  }

  // Method to handle the event
  public void handleEvent(Message<JsonObject> message) {
    // Creare le opzioni di consegna desiderate
    DeliveryOptions options = new DeliveryOptions()
        .addHeader(TARGET_VIRTUAL_ADDRESSES, String.join(",",
eventHandlerVirtualAddresses))
        .addHeader(SOURCE_VIRTUAL_ADDRESS, httpRequestVirtualAddress)
        .addHeader(SOURCE_COMPONENT, HttpRequestConsumer.class.getName());

        eventBus.publish(dispatcherVirtualAddress, message.body(), options);
  }
}
```

Dobbiamo quindi implementare il Dispatcher tenendo conto delle informazioni presenti sull'header `TARGET_VIRTUAL_ADDRESSES` e che dobbiamo ricevere l'esito dell'elaborazione del messaggio da parte degli Event Handler. A seguire il codice completo del Dispatcher.

Il metodo `void handleEvent(Message<JsonObject> message)` è responsabile di:

1. leggere gli header dal messaggio ricevuto e in particolare l'header `TARGET_VIRTUAL_ADDRESSES` che contiene gli indirizzi virtuali degli Event Handler;
2. inviare il messaggio a tutti gli Event Handler specificati nell'header `TARGET_VIRTUAL_ADDRESSES` (tramite il metodo `eventBus.<String>request()`) e attende (in async mode) l'esito dell'elaborazione del messaggio da parte degli Event Handler. L'esito dell'elaborazione è loggato in console.

Code 7 - *Implementazione del Dispatcher*

```
package it.dontesta.eventbus.consumers.events.handlers;

import io.quarkus.runtime.StartupEvent;
import io.smallrye.mutiny.Uni;
import io.vertx.core.json.JsonObject;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;
import java.util.List;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;

@ApplicationScoped
public class Dispatcher {
  @Inject
  EventBus eventBus;

  @Inject
  Logger log;

  @ConfigProperty(name = "app.eventbus.consumer.dispatcher.address")
  String dispatcherVirtualAddress;

  public static final String SOURCE_VIRTUAL_ADDRESS = "source-virtual-address";

  public static final String SOURCE_COMPONENT = "source-component";

  public static final String TARGET_VIRTUAL_ADDRESSES = "target-virtual-addresses";

  void onStart(@Observes StartupEvent ev) {
    log.debugf(
        "Registering the dispatcher to the event bus for the event handler at
addresses: {%s}",
        dispatcherVirtualAddress);

    eventBus.consumer(dispatcherVirtualAddress, this::handleEvent);
  }

  // Method to handle the event
  public void handleEvent(Message<JsonObject> message) {
    // Leggere gli header dalle DeliveryOptions
    String sourceVirtualAddress = message.headers().get(SOURCE_VIRTUAL_ADDRESS);
    String sourceComponent = message.headers().get(SOURCE_COMPONENT);
    List<String> targetVirtualAddressesList =
        List.of(message.headers().get(TARGET_VIRTUAL_ADDRESSES).split(","));
```

```
    log.debugf(
        new StringBuilder().append(
                "Received event message from source virtual address: %s and source
component: %s ")
            .append("for the target virtual addresses: %s").toString(),
        sourceVirtualAddress, sourceComponent, message.headers().get
(TARGET_VIRTUAL_ADDRESSES));

    // Invia l'evento a tutti i target virtual addresses
    targetVirtualAddressesList.forEach(targetVirtualAddress -> {

      // Creare le opzioni di consegna desiderate
      DeliveryOptions options = new DeliveryOptions()
          .addHeader(SOURCE_VIRTUAL_ADDRESS, sourceVirtualAddress)
          .addHeader(SOURCE_COMPONENT, sourceComponent);

      log.debugf("Sending event message to target virtual address: %s",
targetVirtualAddress);

      Uni<String> response = eventBus.<String>request(targetVirtualAddress, message
.body())
          .onItem().transform(Message::body);

      response.subscribe().with(
          result -> {
            log.debugf("Received response from target virtual address: %s with
result: %s",
                targetVirtualAddress, result);
          },
          failure -> {
            log.errorf(
                "Failed to receive response from target virtual address: %s with
failure: %s",
                targetVirtualAddress, failure);
          }
      );
    });
  }
}
```

Passiamo adesso all'implementazione degli Event Handler. Questi componenti sono consumer registrati sull'Event Bus con gli indirizzi virtuali definiti in `app.eventbus.consumer.event.handler.addresses` sul file di configurazione `application.properties` dell'applicazione Quarkus.

Gli Event Handler sono responsabili di ricevere i messaggi dal Dispatcher, elaborarli e restituire l'esito dell'elaborazione. Implementeremo due Event Handler, uno per il tracciamento su un

database NoSQL come MongoDB e l'altro per il tracciamento su un broker di messaggi come Apache ActiveMQ Artemis.

Lascerò a voi il compito della realizzazione dell'Event Handler per il tracciamento su un database SQL come MySQL o PostgreSQL. Vi auguro quindi buon divertimento!

9. Aggiornamento v1.4.0 — TraceEventDispatcher ad alte prestazioni

Quanto descritto in questa sezione si riferisce alla versione **1.4.0** del progetto software.

A partire dalla versione **1.4.0**, il componente `Dispatcher` è stato sostituito da un nuovo bean `@ApplicationScoped` denominato `TraceEventDispatcher`, progettato per eliminare i colli di bottiglia riscontrati sotto carico elevato.

Problemi risolti

L'approccio precedente usava `@Scheduled` per il drain della coda e `ConcurrentLinkedQueue` non bounded. Sotto carico elevato ciò causava:

- **starvation dello scheduler**: il thread `@Scheduled` competeva con i thread HTTP per il pool di worker Quarkus, non ricevendo CPU sufficiente;
- **rischio OOM** e errori `SRMSG00034` ("Insufficient downstream requests to emit item") causati dalla coda non limitata che si accumulava senza controllo.

Architettura del TraceEventDispatcher

Code 7b - *Struttura semplificata del TraceEventDispatcher (v1.4.0)*

```
@ApplicationScoped
public class TraceEventDispatcher {

  // Code bounded per request e response (capacità configurabile, default 5000)
  private ArrayBlockingQueue<RequestTrace> requestQueue;
  private ArrayBlockingQueue<ResponseTrace> responseQueue;

  // Contatori atomici degli eventi scartati per backpressure
  private final AtomicLong droppedRequests = new AtomicLong();
  private final AtomicLong droppedResponses = new AtomicLong();

  void onStart(@Observes StartupEvent ev) {
    requestQueue = new ArrayBlockingQueue<>(queueCapacity);
    responseQueue = new ArrayBlockingQueue<>(queueCapacity);

    // Thread daemon OS-scheduled dedicato — non condivide il worker pool Quarkus
    Thread.ofPlatform().daemon(true).name("trace-event-dispatcher").start(this
::drainLoop);
  }

  // Accoda O(1) — chiamato dal filtro JAX-RS
  public void enqueueRequest(RequestTrace trace) {
```

```
    if (!requestQueue.offer(trace)) {
      droppedRequests.incrementAndGet();
      log.warn("Request trace dropped — queue full");
    }
  }

  // Drain in burst: costruisce JSON, serializza e pubblica sull'Event Bus
  private void drainLoop() {
    while (!Thread.currentThread().isInterrupted()) {
      RequestTrace req = requestQueue.poll();
      ResponseTrace res = responseQueue.poll();
      if (req == null && res == null) {
        Thread.sleep(intervalMs);    // idle sleep configurabile
        continue;
      }
      if (req != null) eventBus.publish(httpRequestVirtualAddress, toJson(req));
      if (res != null) eventBus.publish(httpResponseVirtualAddress, toJson(res));
    }
  }

  @PreDestroy
  void onStop() { /* flush finale della coda prima dello shutdown */ }
}
```

I punti architetturali salienti sono riassunti nella tabella seguente.

Componente	Descrizione
Thread daemon	`Thread.ofPlatform().daemon(true).name("trace-event-dispatcher")` — thread OS-scheduled dedicato che non condivide il pool di worker Quarkus con i thread HTTP.
`ArrayBlockingQueue` (bounded)	Capacità configurabile via `app.filter.dispatcher.queue.capacity` (default 5 000). Gli eventi in eccesso vengono scartati con `WARN` e conteggiati da `AtomicLong`.
Drain in burst mode	Il thread drena in modo continuo quando le code contengono elementi; va in sleep di `app.filter.dispatcher.interval.ms` (default 20 ms) quando entrambe le code sono vuote.
Flush su `@PreDestroy`	Prima dello shutdown, il dispatcher svuota le code residue per evitare la perdita di eventi già accodati.
Backpressure	Drop-on-full con contatori `droppedRequests` / `droppedResponses` (`AtomicLong`) ispezionabili a runtime.

Risultati di benchmark (v1.3.0 → v1.4.0)

I benchmark eseguiti su Red Hat Developer Sandbox (OpenShift 4.21.7 / k8s v1.34.5, 1 Pod, 15 000 richieste totali con JMeter/Taurus) mostrano i seguenti miglioramenti:

Metrica	v1.3.0	v1.4.0	Variazione
Throughput	baseline	+10–15%	↑
Latenza media	baseline	−11–14%	↓
Latenza p95	baseline	−17–22%	↓
Errori	0	0	—

9.1. Implementazione Event Handler MongoDB

Prima di procedere con l'implementazione dell'Event Handler per il tracciamento su MongoDB, è necessario aggiungere la dipendenza per il driver MongoDB al progetto Quarkus. Questo può essere fatto tramite la CLI di Quarkus o tramite Maven. A seguire il comando per aggiungere la dipendenza tramite la CLI di Quarkus o tramite Maven.

Console 11 - *Installazione estensione MongoDB di Quarkus tramite CLI o Maven*

```
# Aggiungere la dipendenza per il driver MongoDB
# tramite la CLI di Quarkus
quarkus extension add mongodb-client

# Aggiungere la dipendenza per il driver MongoDB
# tramite il comando Maven
mvn quarkus:add-extension -Dextensions='mongodb-client'
```

La documentazione di Quarkus MongoDB fornisce tutte le informazioni necessarie per configurare il client MongoDB e per implementare le operazioni CRUD. Nel nostro caso utilizzeremo la versione reattiva del client MongoDB.

L'Event Handler `MongoDbEventHandler` è il componente per il tracciamento su MongoDB; è un consumer registrato sull'Event Bus con l'indirizzo virtuale definito in `app.eventbus.consumer.event.handler.addresses[1]` sul file di configurazione `application.properties` dell'applicazione Quarkus.

Il metodo `void handleEvent(Message<JsonObject> message)` è responsabile di:

1. leggere gli header dal messaggio ricevuto e in particolare l'header `SOURCE_COMPONENT` che contiene il nome del componente sorgente;
2. creare un documento MongoDB a partire dal messaggio ricevuto tenendo conto del componente sorgente (vedi il metodo `Document getMongoDbDocument(String sourceComponent, JsonObject jsonObject)`);
3. inserire il documento MongoDB nel database specificato dalla configurazione `app.eventbus.consumer.event.handler.nosql.mongodb.database.name` e nella collezione indicata dalla configurazione `app.eventbus.consumer.event.handler.nosql.mongodb.database.collection`. Utilizzando la parte reattiva del client MongoDB, l'inserimento del documento nel database è asincrono (vedi

corpo del metodo `subscribe().with(...)`) e il risultato dell'operazione è inviato come risposta al Dispatcher (vedi il metodo `message.reply()` in caso di successo e il metodo `message.fail()` in caso di fallimento).

Code 8 - *Implementazione dell'Event Handler per il tracciamento su MongoDB*

```
package it.dontesta.eventbus.consumers.events.handlers.nosql;

import io.quarkus.mongodb.reactive.ReactiveMongoClient;
import io.quarkus.mongodb.reactive.ReactiveMongoCollection;
import io.quarkus.runtime.StartupEvent;
import io.vertx.core.json.JsonObject;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import it.dontesta.eventbus.consumers.http.HttpRequestConsumer;
import it.dontesta.eventbus.consumers.http.HttpResponseConsumer;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;
import org.bson.Document;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;

@ApplicationScoped
public class MongoDbEventHandler {

  @Inject
  EventBus eventBus;

  @Inject
  Logger log;

  @Inject
  ReactiveMongoClient mongoClient;

  @ConfigProperty(name = "app.eventbus.consumer.event.handler.addresses[1]")
  String mongoDbEventHandlerVirtualAddress;

  @ConfigProperty(name =
"app.eventbus.consumer.event.handler.nosql.mongodb.database.name",
      defaultValue = "audit")
  String databaseName;

  @ConfigProperty(name =
"app.eventbus.consumer.event.handler.nosql.mongodb.database.collection",
      defaultValue = "jax-rs-requests")
  String databaseCollectionName;

  public static final String SOURCE_COMPONENT = "source-component";

  void onStart(@Observes StartupEvent ev) {
    log.debugf(
        "Registering the MongoDB event handler at addresses: {%s}",
```

```
        mongoDbEventHandlerVirtualAddress);

    eventBus.consumer(mongoDbEventHandlerVirtualAddress, this::handleEvent);
  }

  // Method to handle the event
  public void handleEvent(Message<JsonObject> message) {
    // Recupera il componente sorgente dagli header del messaggio
    // e il corpo del messaggio stesso in formato JsonObject
    String sourceComponent = message.headers().get(SOURCE_COMPONENT);
    JsonObject body = message.body();

    // Crea un documento MongoDB a partire dal messaggio dell'evento
    Document mongoDbDocument = getMongoDbDocument(sourceComponent, body);

    if (mongoDbDocument == null) {
      message.fail(1, "Could not create a MongoDB document from the event
message.");
      return;
    }

    // Inserisci il documento MongoDB nel database
    getCollection().insertOne(mongoDbDocument).subscribe().with(
        result -> message.reply(
            "Documents inserted successfully with Id %s".formatted(result
.getInsertedId())),
        failure -> message.fail(-1, "Errors occurred while inserting the document.")
    );
  }

  /**
   * Metodo per ottenere la collezione MongoDB specificata nel parametro di
configurazione
   * {@code app.eventbus.consumer.event.handler.nosql.mongodb.database.collection}.
   *
   * @return la collezione MongoDB specificata
   */
  private ReactiveMongoCollection<Document> getCollection() {
    return mongoClient.getDatabase(databaseName).getCollection
(databaseCollectionName);
  }

  /**
   * Metodo per creare un documento MongoDB a partire dal messaggio dell'evento.
   *
   * <p>In questo caso si considerano solo due componenti sorgente:
HttpRequestConsumer e
   * HttpResponseConsumer tramite il parametro sourceComponent
   * restituendo un documento MongoDB creato con la stessa struttura.
```

```
    *
    * @param sourceComponent il componente sorgente dell'evento
    * @param jsonObject      il messaggio dell'evento
    * @return il documento MongoDB creato
    */
  private Document getMongoDbDocument(String sourceComponent, JsonObject jsonObject)
{
    if (sourceComponent.equals(HttpRequestConsumer.class.getName())) {
      return Document.parse(jsonObject.encode());
    }

    if (sourceComponent.equals(HttpResponseConsumer.class.getName())) {
      return Document.parse(jsonObject.encode());
    }

    return null;
  }

}
```

A questo punto non resta che provare il funzionamento dell'Event Handler per il tracciamento su MongoDB. Per fare ciò abbiamo due possibilità:

1. avviare l'applicazione tramite il comando `quarkus dev` e fare una richiesta cURL verso l'endpoint `/api/rest/echo` verificando che il messaggio sia correttamente tracciato su MongoDB;
2. eseguire gli unit test utilizzando il comando `quarkus test` o `mvn test` verificando che il messaggio sia correttamente tracciato su MongoDB.

Console 12 - *Esecuzione del test di tracciamento su MongoDB chiamando l'endpoint* `/api/rest/echo`

```
# Chiamata cURL verso l'endpoint /api/rest/echo per testare il tracciamento su
MongoDB
curl -v -H "Content-Type: application/json" \
  -d '{"message": "Test di tracking richiesta JAX-RS su MongoDB tramite Event
Handler MongoDbEventHandler"}' \
  http://localhost:8080/api/rest/echo

# Risultato atteso

* Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080
> POST /api/rest/echo HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: _/_
> Content-Type: application/json
> Content-Length: 101
```

```
>
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
< content-length: 101
< Set-Cookie: user_tracking_id=a17dbd96-fda1-4cec-92b1-a0c72bee645b;Version=1
;Comment="Cookie di tracciamento dell'utente";Path=/;Max-Age=2592000
< X-Correlation-ID: a3fb63ac-7c69-46d0-90df-704ddad49664
<
* Connection #0 to host localhost left intact
{"message": "Test di tracking richiesta JAX-RS su MongoDB tramite Event Handler
MongoDbEventHandler"}
```

Verificando il log dell'applicazione dovreste vedere i messaggi di tracciamento circa lo store su MongoDB come mostrato a seguire, e in particolare quelli contenenti gli identificativi dei documenti BSON, che dovrebbero essere due, quello di richiesta e di risposta.

Console 13 - *Log di tracciamento su MongoDB*

```
2024-04-12 12:02:26,389 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: nosql-trace with result:
Documents inserted successfully with Id BsonObjectId{value=661906b2a06f98122c332bf3}
2024-04-12 12:02:26,392 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: nosql-trace with result:
Documents inserted successfully with Id BsonObjectId{value=661906b2a06f98122c332bf4}
```

Una volta ottenuti gli identificativi dei documenti BSON, potete verificare il contenuto degli stessi eseguendo il comando `podman exec -it <container_name or container_id> mongo audit --eval "db.getCollection('jax-rs-requests').find('<bsonObjectId>')"`. Prima di eseguire il comando precedente, dovreste sostituire `<container_name or container_id>` con il nome o l'identificativo del container MongoDB e `<bsonObjectId>` con l'identificativo del documento BSON.

Per ottenere l'identificativo del container MongoDB, eseguite il comando `podman ps` e cercate il container creato dall'immagine di MongoDB. A seguire un esempio di output del comando `podman ps`.

Nota: per coloro che non abbiano installato Podman, possono utilizzare i comandi Docker al posto di Podman, per cui, il comando precedente per verificare il documento su MongoDB diventa: `docker exec -it <container_name or container_id> mongo audit --eval "db.getCollection('jax-rs-requests').find('<bsonObjectId>')"`.

Console 14 - *Esempio di output del comando* `podman ps`

```
CONTAINER ID  IMAGE                                COMMAND               CREATED
STATUS            PORTS                     NAMES
81bf31e3e393  docker.io/testcontainers/ryuk:0.6.0  /bin/ryuk             About an
hour ago  Up About an hour  0.0.0.0:42179->8080/tcp   testcontainers-ryuk-c0624f11-
3e13-4c74-9b9b-398295c423ff
ef48a8b51727  docker.io/library/mongo:4.4          --replSet docker-...  About an
hour ago  Up About an hour  0.0.0.0:45925->27017/tcp  awesome_leavitt
```

Una volta ottenuto l'identificativo del container di MongoDB, è possibile procedere con la verifica del contenuto dei documenti BSON. A seguire l'esempio di ciò che dovreste ottenere.

Console 15 - *Contenuto del documento Bson con identificativo 661906b2a06f98122c332bf3 che contiene alcune informazioni della request JAX-RS*

```
# Interrogazione di MongoDB per ottenere il contenuto del documento Bson con
identificativo 661906b2a06f98122c332bf3
podman exec -it ef48a8b51727 mongo audit --eval "db.getCollection('jax-rs-
requests').find(ObjectId('661906b2a06f98122c332bf3')).pretty()"

# Output atteso

{
    "_id" : ObjectId("661906b2a06f98122c332bf3"),
    "X-Correlation-ID" : "a3fb63ac-7c69-46d0-90df-704ddad49664",
    "remote-ip-address" : "127.0.0.1",
    "headers" : {
        "User-Agent" : [
            "curl/8.4.0"
        ],
        "Host" : [
            "localhost:8080"
        ],
        "Accept" : [
            "*/*"
        ],
        "Content-Length" : [
            "101"
        ],
        "Content-Type" : [
            "application/json"
        ]
    },
    "body" : "{"message": "Test di tracking richiesta JAX-RS su MongoDB tramite
Event Handler MongoDbEventHandler"}",
    "uri-info" : "http://localhost:8080/api/rest/echo",
    "local-date-time-in" : "2024-04-12T12:02:26.372554",
    "method" : "POST",
    "media-type" : "application/json",
    "acceptable-language" : "[]",
    "acceptable-media-types" : "[*/*]"
}
```

I due documenti possono essere collegati tramite l'identificativo `X-Correlation-ID` che è presente sia nel documento della request che in quello della response.

Console 16 - *Contenuto del documento Bson con identificativo 661906b2a06f98122c332bf4 che contiene alcune informazioni della response JAX-RS*

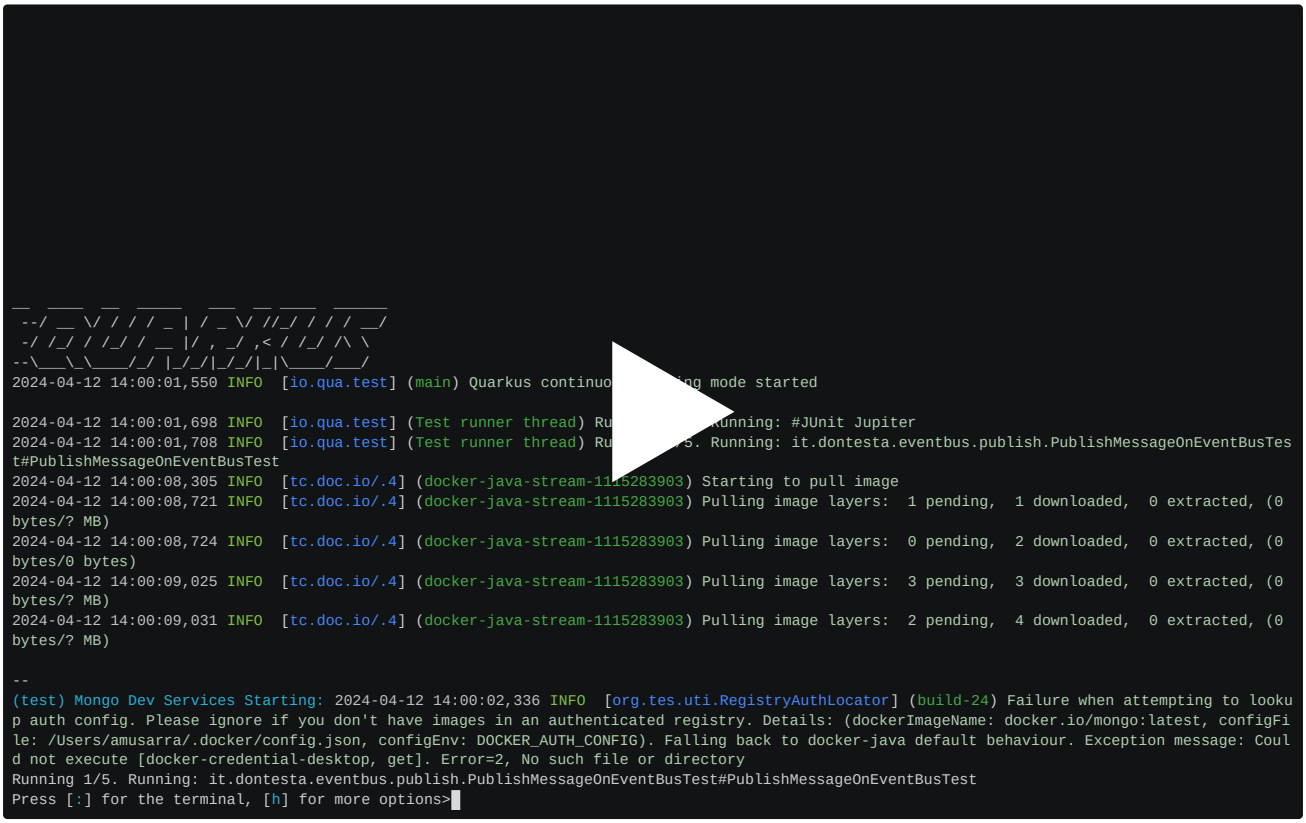
```
# Interrogazione di MongoDB per ottenere il contenuto del documento Bson con
identificativo 661906b2a06f98122c332bf4
podman exec -it ef48a8b51727 mongo audit --eval "db.getCollection('jax-rs-
requests').find('661906b2a06f98122c332bf4').pretty()"

# Output atteso

{
    "_id" : ObjectId("661906b2a06f98122c332bf4"),
    "X-Correlation-ID" : "a3fb63ac-7c69-46d0-90df-704ddad49664",
    "local-date-time-out" : "2024-04-12T12:02:26.373859",
    "status" : 200,
    "status-info-family-name" : "SUCCESSFUL",
    "status-info-reason" : "OK",
    "headers" : {
        "X-Correlation-ID" : "a3fb63ac-7c69-46d0-90df-704ddad49664",
        "Set-Cookie" : "user_tracking_id=a17dbd96-fda1-4cec-92b1-
a0c72bee645b;Version=1;Comment="Cookie di tracciamento dell'utente";Path=/;Max-
Age=2592000"
    },
    "body" : "{"message": "Test di tracking richiesta JAX-RS su MongoDB tramite
Event Handler MongoDbEventHandler"}"
}
```

Se ricordate, in configurazione abbiamo definito tutti e quattro gli indirizzi virtuali per gli Event Handler ma abbiamo implementato solo due di questi; di conseguenza, quando il Dispatcher cercherà di inviare il messaggio alle due destinazioni mancanti, riceverà dall'Event Bus un errore di mancata ricezione di risposta dagli Event Handler che difatto non sono disponibili (`(NO_HANDLERS,-1) No handlers for address sql-trace`). Questo è un comportamento normale e non preoccupatevi, in produzione non dovreste mai avere questo tipo di problema, in quanto tutti gli Event Handler dovrebbero essere implementati e funzionanti o qualora non fossero disponibili, basterà non metterli in configurazione.

Console 17 - *Log di tracciamento su MongoDB e errori di mancata ricezione di risposta dagli Event Handler*

```
2024-04-12 12:02:26,376 ERROR [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Failed to receive response from target virtual address: queue-trace with
failure: (NO_HANDLERS,-1) No handlers for address queue-trace
2024-04-12 12:02:26,381 ERROR [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Failed to receive response from target virtual address: sql-trace with
failure: (NO_HANDLERS,-1) No handlers for address sql-trace
```

Lascio a voi la verifica attraverso gli unit test dell'Event Handler per il tracciamento su MongoDB eseguendo il comando `quarkus test` o `mvn test`. In ogni caso dovreste ottenere un risultato simile a quello mostrato a seguire e che potete vedere anche qui Test Quarkus Event Bus: Event Handler MongoDB.



Asciinema 1 - *Esecuzione degli unit test che mostra l'Event Handler per il tracciamento su MongoDB in azione*

A questo punto abbiamo completato l'implementazione dell'Event Handler per il tracciamento su MongoDB. Possiamo procedere con l'implementazione dell'Event Handler per il tracciamento sulla coda AMQP.

9.2. Implementazione Event Handler AMQP

Prima di procedere con l'implementazione dell'Event Handler per il tracciamento sul broker AMQP, è necessario aggiungere le dipendenze per il supporto AMQP al progetto Quarkus. Questo può essere fatto tramite la CLI di Quarkus o tramite Maven. A seguire il comando per aggiungere la dipendenza tramite la CLI di Quarkus o tramite Maven.

Console 18 - *Installazione estensione AMQP Messaging di Quarkus tramite CLI o Maven*

```
# Aggiungere la dipendenza per il Reactive Messaging RabbitMQ connector
# tramite la CLI di Quarkus
quarkus extension add quarkus-messaging-amqp

# Aggiungere la dipendenza per il Reactive Messaging RabbitMQ connector

# tramite il comando Maven
mvn quarkus:add-extension -Dextensions='messaging-rabbitmq'
```

La documentazione di Quarkus AMQP Messaging 1.0 fornisce tutte le informazioni necessarie per configurare il client AMQP e per implementare le operazioni di invio e ricezione di messaggi. Il broker AMQP che utilizzeremo è Apache ActiveMQ Artemis che implementa il protocollo AMQP 1.0 ed è già integrato in Quarkus come Dev Services.

La prima attività da fare è configurare il client AMQP per la connessione al broker AMQP. Questo può essere fatto tramite il file di configurazione `application.properties` dell'applicazione Quarkus. A seguire la configurazione per la connessione al broker AMQP.

Configurazione 4 - *Configurazione del client AMQP per la connessione al broker AMQP*

```
# Configure the outgoing/incoming AMQP connector and address
# The outgoing connector is used to send the HTTP request and
# response events to the AMQP broker
# The incoming connector is used to receive the HTTP request and
# response events from the AMQP broker
mp.messaging.outgoing.http-request-out.connector=smallrye-amqp
mp.messaging.outgoing.http-response-out.connector=smallrye-amqp
mp.messaging.incoming.http-request-in.connector=smallrye-amqp
mp.messaging.incoming.http-response-in.connector=smallrye-amqp
mp.messaging.outgoing.http-request-out.address=http-request
mp.messaging.outgoing.http-response-out.address=http-response
mp.messaging.incoming.http-request-in.address=http-request
mp.messaging.incoming.http-response-in.address=http-response
```

La configurazione comporta la definizione di quattro connettori AMQP (la cui implementazione è quella di Smallrye), due per l'invio degli eventi HTTP request e response al broker AMQP e due per la ricezione degli eventi HTTP request e response dal broker AMQP. Ogni connettore è associato a un indirizzo AMQP che rappresenta la coda AMQP a cui inviare o da cui ricevere i messaggi.

A questo punto possiamo procedere con l'implementazione dell'Event Handler per il tracciamento sul broker. A seguire il codice completo dell'Event Handler `AmqpEventHandler` per il tracciamento su Apache ActiveMQ Artemis. Questo componente è un consumer registrato sull'Event Bus con l'indirizzo virtuale definito in `app.eventbus.consumer.event.handler.addresses[2]` sul file di configurazione `application.properties` dell'applicazione Quarkus.

In questa implementazione facciamo uso di due componenti fondamentali che fanno parte del **MicroProfile Reactive Messaging**, ovvero, `Emitter` e `Channel`. L'`Emitter` è utilizzato per inviare effettivamente i messaggi al canale specificato, mentre il `Channel` è il canale di messaggistica asincrona attraverso il quale i messaggi vengono inviati e ricevuti (fare riferimento alla configurazione dell'applicazione per i canali `http-request-out` e `http-response-out`).

Il metodo `void handleEvent(Message<JsonObject> message)` è responsabile di:

1. leggere gli header dal messaggio ricevuto e in particolare l'header `SOURCE_COMPONENT` che contiene il nome del componente sorgente;
2. inviare il messaggio alla coda AMQP (vedi metodo `sendToQueue(Message<JsonObject> message, Emitter<JsonObject> emitter)`) per i messaggi di richiesta HTTP se il componente sorgente è `HttpRequestConsumer` e alla coda AMQP per i messaggi di risposta HTTP se il componente sorgente è `HttpResponseConsumer`. L'invio del messaggio alla coda AMQP è asincrono e il risultato dell'operazione è inviato come risposta al Dispatcher (vedi il metodo `message.reply()` in caso di successo e il metodo `message.fail()` in caso di fallimento).

Code 9 - *Implementazione dell'Event Handler per il tracciamento sul broker AMQP che in questo caso è Apache ActiveMQ Artemis*

```
package it.dontesta.eventbus.consumers.events.handlers.queue;

import io.quarkus.runtime.StartupEvent;
import io.vertx.core.json.JsonObject;
import io.vertx.mutiny.core.eventbus.EventBus;
import io.vertx.mutiny.core.eventbus.Message;
import it.dontesta.eventbus.consumers.http.HttpRequestConsumer;
import it.dontesta.eventbus.consumers.http.HttpResponseConsumer;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.eclipse.microprofile.reactive.messaging.Emitter;
import org.jboss.logging.Logger;
```

```
@ApplicationScoped
public class AmqpEventHandler {

  @Channel("http-request-out")
  Emitter<JsonObject> requestEmitter;

  @Channel("http-response-out")
  Emitter<JsonObject> responseEmitter;

  @Inject
  EventBus eventBus;

  @Inject
  Logger log;

  @ConfigProperty(name = "app.eventbus.consumer.event.handler.addresses[2]")
  String amqpEventHandlerVirtualAddress;

  public static final String SOURCE_COMPONENT = "source-component";

  void onStart(@Observes StartupEvent ev) {
    log.debugf("Registering the AMQP event handler at addresses: {%s}",
        amqpEventHandlerVirtualAddress);

    eventBus.consumer(amqpEventHandlerVirtualAddress, this::handleEvent);
  }

  // Method to handle the event
  public void handleEvent(Message<JsonObject> message) {
    // Recupera il componente sorgente dagli header del messaggio
    String sourceComponent = message.headers().get(SOURCE_COMPONENT);

    // Invia il messaggio alla coda AMQP per i messaggi di richiesta HTTP
    if (sourceComponent.equals(HttpRequestConsumer.class.getName())) {
      sendToQueue(message, requestEmitter);
    }

    // Invia il messaggio alla coda AMQP per i messaggi di risposta HTTP
    if (sourceComponent.equals(HttpResponseConsumer.class.getName())) {
      sendToQueue(message, responseEmitter);
    }
  }

  /**
   * Invio del messaggio alla coda AMQP e rimanere in attesa di una risposta
   * per confermare l'invio del messaggio e notificarlo al Dispatcher.
   *
   * @param message Il messaggio da inviare alla coda AMQP
```

```
    * @param emitter L'oggetto Emitter per inviare il messaggio alla coda AMQP
    */
   private void sendToQueue(Message<JsonObject> message,
                            Emitter<JsonObject> emitter) {
     emitter.send(message.body()).whenComplete((result, error) -> {
       if (error != null) {
         message.fail(1, error.getMessage());
       } else {
         message.reply("Message sent to AMQP queue successfully!");
       }
     });
   }
}
```

A questo punto non resta che provare il funzionamento dell'Event Handler per il tracciamento sul broker AMQP. Per fare ciò abbiamo due possibilità:

1. avviare l'applicazione tramite il comando `quarkus dev` e fare una richiesta cURL verso l'endpoint `/api/rest/echo` verificando che il messaggio sia correttamente tracciato sul broker AMQP;
2. eseguire gli unit test utilizzando il comando `quarkus test` o `mvn test` verificando che il messaggio sia correttamente tracciato sul broker AMQP.

Per verificare che i messaggi siano stati effettivamente consegnati al message broker AMQP, è possibile usare il comando `artemis consumer` per attaccarsi alla coda AMQP e consumare i messaggi pubblicati dall'Event Handler. A seguire un esempio di come fare.

Per reperire il `container-id` del container di Apache ActiveMQ Artemis, eseguire il comando `podman ps` e cercare il container creato dall'immagine di Apache ActiveMQ Artemis.

Per reperire l'indirizzo IP del container di Apache ActiveMQ Artemis, eseguire il comando `podman container inspect --format '{{.NetworkSettings.IPAddress}}' <container-id>` e prendere l'indirizzo IP del container.

Console 19 - *Consumazione dei messaggi dalla coda AMQP http-request e http-response*

```
# Comando per registrare un consumer sulla coda AMQP http-request
podman exec -it <container-id> ./broker/bin/artemis consumer --user guest --password
guest --verbose --url tcp://<indirizzo-ip-container>:61616 --protocol AMQP
--destination http-request

# Comando per registrare un consumer sulla coda AMQP http-response
podman exec -it <container-id> ./broker/bin/artemis consumer --user guest --password
guest --verbose --url tcp://<indirizzo-ip-container>:61616 --protocol AMQP
--destination http-response

# Esempio di output atteso
Consumer http-request, thread=0 Received {"X-Correlation-ID":"f47c01b2-3ba5-4539-
```

```
9ab4-a3cd78f01f2c","remote-ip-address":"127.0.0.1","headers":{"User-Agent"
:["curl/8.4.0"],"Host":["localhost:8080"],"Accept":["*/*"],"Content-Length":["102"]
,"Content-Type":["application/json"]},"body":"{\"message\": \"Test di tracking
richiesta JAX-RS su AMQP Broker tramite Event Handler AmqpEventHandler\"}","uri-
info":"http://localhost:8080/api/rest/echo","local-date-time-in":"2024-04-
13T00:31:07.682214","method":"POST","media-type":"application/json","acceptable-
language":"[]","acceptable-media-types":"[*/*]"}
JMS Message ID:null
Received text sized at 558

# Esempio di output atteso
Consumer http-response, thread=0 Received {"X-Correlation-ID":"f47c01b2-3ba5-4539-
9ab4-a3cd78f01f2c","local-date-time-out":"2024-04-13T00:31:07.752305","status"
:200,"status-info-family-name":"SUCCESSFUL","status-info-reason":"OK","headers":{"X-
Correlation-ID":"f47c01b2-3ba5-4539-9ab4-a3cd78f01f2c","Set-Cookie"
:"user_tracking_id=496e1a23-98ed-4b8e-b18b-f512b6225dde;Version=1;Comment=\"Cookie
di tracciamento dell'utente\";Path=/;Max-Age=2592000"},"body":"{\"message\": \"Test
di tracking richiesta JAX-RS su AMQP Broker tramite Event Handler AmqpEventHandler
\"}"}
JMS Message ID:null
Received text sized at 523
```

All'interno del progetto è disponibile un componente che funge da consumer per i messaggi inviati alla coda AMQP. Questo componente è `AmqpConsumer` e le annotazioni `@Incoming("http-request-in")` e `@Incoming("http-response-in")` sono utilizzate per definire i canali di messaggistica asincrona attraverso i quali i messaggi vengono ricevuti.

I metodi `CompletionStage<Void> consumeHttpRequest(Message<JsonObject> requestMessage)` e `CompletionStage<Void> consumeHttpResponse(Message<JsonObject> responseMessage)`:

1. ricevono i messaggi inviati alla coda AMQP `http-request` e `http-response`;
2. loggano il contenuto dei messaggi ricevuti;
3. impostano il messaggio come completato tramite il metodo `message.ack()`.

Code 10 - *Implementazione del consumer per i messaggi inviati alla coda AMQP http-request e http-response*

```java
package it.dontesta.eventbus.consumers.events.handlers.queue.incoming;

import io.vertx.core.json.JsonObject;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import java.util.concurrent.CompletionStage;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Message;
import org.jboss.logging.Logger;

@ApplicationScoped
```

```
public class AmqpConsumer {

  @Inject
  Logger log;

  @Incoming("http-request-in")
  public CompletionStage<Void> consumeHttpRequest(Message<JsonObject>
requestMessage) {
    // Implementa la logica per consumare il messaggio della richiesta HTTP
    log.debug("Received HTTP request message: " + requestMessage.getPayload());
    return requestMessage.ack();
  }

  @Incoming("http-response-in")
  public CompletionStage<Void> consumeHttpResponse(Message<JsonObject>
requestMessage) {
    // Implementa la logica per consumare il messaggio della richiesta HTTP
    log.debug("Received HTTP response message: " + requestMessage.getPayload());
    return requestMessage.ack();
  }
}
```

Dalla configurazione dell'applicazione Quarkus, questo consumer è stato disabilitato per evitare che di default i messaggi inviati alla coda AMQP siano consumati all'interno della stessa applicazione. Per abilitare il consumer è necessario commentare la configurazione indicata a seguire.

Configurazione 5 - *Disabilitazione del consumer per i messaggi inviati alla coda AMQP http-request e http-response*

```
# The list of types that should be excluded from discovery.
# The types should be specified using their fully qualified name.
# The types are separated by a comma.
# In this case, the AmqpConsumer class is excluded
quarkus.arc.exclude-types
=it.dontesta.eventbus.consumers.events.handlers.queue.incoming.AmqpConsumer
```

10. Bonus

Il progetto Quarkus è nato come strumento di sviluppo di applicazioni Java native per gli ambienti cloud. **Quale migliore occasione per testare la nostra applicazione Quarkus su OpenShift?**

Ho pensato quindi a questo Bonus per chi volesse installare l'applicazione Quarkus su OpenShift e in particolare sulla Developer Sandbox (o DevSandbox) di Red Hat.

Per apprezzare a pieno e seguire i contenuti di questo Bonus, è necessario avere una conoscenza di base di OpenShift e della Developer Sandbox di Red Hat e in particolare del mondo Kubernetes. Per questo motivo, non entrerò nei dettagli di come installare e configurare OpenShift e la Developer Sandbox di Red Hat, ma mi limiterò a fornire le istruzioni per installare l'applicazione Quarkus su OpenShift. A seguire una serie di risorse utili per apprendere i concetti di base di OpenShift e Kubernetes.

1. Kubernetes. Guida per gestire e orchestrare i container (Libro di Serena Sensini)
2. OpenShift for Developers (eBook - Gratuito)
3. Deploying to OpenShift (eBook - Gratuito)
4. Kubernetes Patterns (eBook - Gratuito)
5. Kubernetes Native Microservices with Quarkus and MicroProfile (eBook - Gratuito)

Il tag di riferimento del progetto su GitHub per il bonus è `bonus-openshift-1` e per installare l'applicazione Quarkus su OpenShift seguite i passaggi indicati nel resto dei capitoli di questo bonus.

10.1. Cos'è la Developer Sandbox di Red Hat?

La Developer Sandbox di Red Hat è un ambiente cloud preconfigurato e pronto all'uso che fornisce agli sviluppatori un'opportunità per esplorare, sperimentare e testare le tecnologie Red Hat senza dover configurare un'infrastruttura complessa, di conseguenza senza la necessità di installare o configurare nulla localmente sulle proprie macchine.

Per iniziare a utilizzare la Developer Sandbox di Red Hat, è sufficiente registrarsi gratuitamente al seguente link e installare la CLI di OpenShift necessaria per eseguire il login sul cluster OpenShift fornito dalla Developer Sandbox di Red Hat e opzionalmente per eseguire operazioni sul cluster.

Quelle poche operazioni necessarie sul cluster OpenShift le faremo usando esclusivamente la CLI di OpenShift e in particolare il comando `oc`. Ovviamente, per chi volesse utilizzare l'interfaccia grafica di OpenShift può farlo accedendo alla Developer Sandbox di Red Hat utilizzando il browser.

10.2. Adeguare il progetto Quarkus per OpenShift

Quarkus offre la possibilità di generare automaticamente risorse OpenShift in base a valori predefiniti e alla configurazione fornita dall'utente; per ottenere ciò, è necessario apportare alcune modifiche alla configurazione del progetto e in particolare:

1. aggiungere la dipendenza `quarkus-openshift` al progetto;
2. aggiungere la dipendenza per le specifiche MicroProfile Health;
3. configurare il plugin `quarkus-openshift` per generare le risorse OpenShift in modo adeguato.

L'estensione OpenShift è in realtà un'estensione wrapper che configura l'estensione Kubernetes con impostazioni predefinite in modo che sia più semplice per l'utente iniziare con Quarkus su OpenShift. Le impostazioni predefinite includono la generazione di risorse OpenShift come `Deployment`, `Service`, `Route`, `ConfigMap`, `Secret`, `PersistentVolumeClaim` e `BuildConfig`.

Per aggiungere la dipendenza `quarkus-openshift` al progetto, è possibile utilizzare la CLI di Quarkus o Maven. A seguire il comando per aggiungere la dipendenza tramite la CLI di Quarkus o tramite Maven.

Console 20 - *Installazione estensione OpenShift di Quarkus tramite CLI o Maven*

```
# Aggiungere la dipendenza per il deploy su OpenShift
# tramite la CLI di Quarkus
quarkus extension add quarkus-openshift

# Aggiungere la dipendenza per il deploy su OpenShift

# tramite il comando Maven
mvn quarkus:add-extension -Dextensions='quarkus-openshift'
```

Per quanto riguarda la configurazione del plugin `quarkus-openshift` tramite il file di configurazione `application.properties` dell'applicazione Quarkus, al momento inseriremo solo la configurazione che ci permetterà di esporre l'applicazione all'esterno del cluster OpenShift. A seguire la configurazione da inserire nel file `application.properties`.

Configurazione 6 - *Configurazione per esporre l'applicazione all'esterno del cluster OpenShift*

```
# If true, the service will be exposed outside of the cluster
# and will be assigned a route.
# If false, the service will only be accessible within the cluster (default)
# Environment variable: QUARKUS_OPENSHIFT_ROUTE_EXPOSE
quarkus.openshift.route.expose=true
```

Per impostazione prendefinita l'esposizione all'esterno è configurata utilizzando il protocollo HTTP. Per abilitare il HTTPS, sarà necessario configurare il plugin `quarkus-openshift` con le informazioni relative al certificato e alla chiave privata.

È opportuno avere una visione d'insieme delle risorse OpenShift/Kubernetes che saranno conivolte nel processo di deploy e di cui il plugin `quarkus-openshift` genererà la descrizione in formato yaml (ma solo di quelle riguardanti l'applicazione). Quello mostrato a seguire è il diagramma delle risorse coinvolte nel deploy dell'applicazione Quarkus su OpenShift.

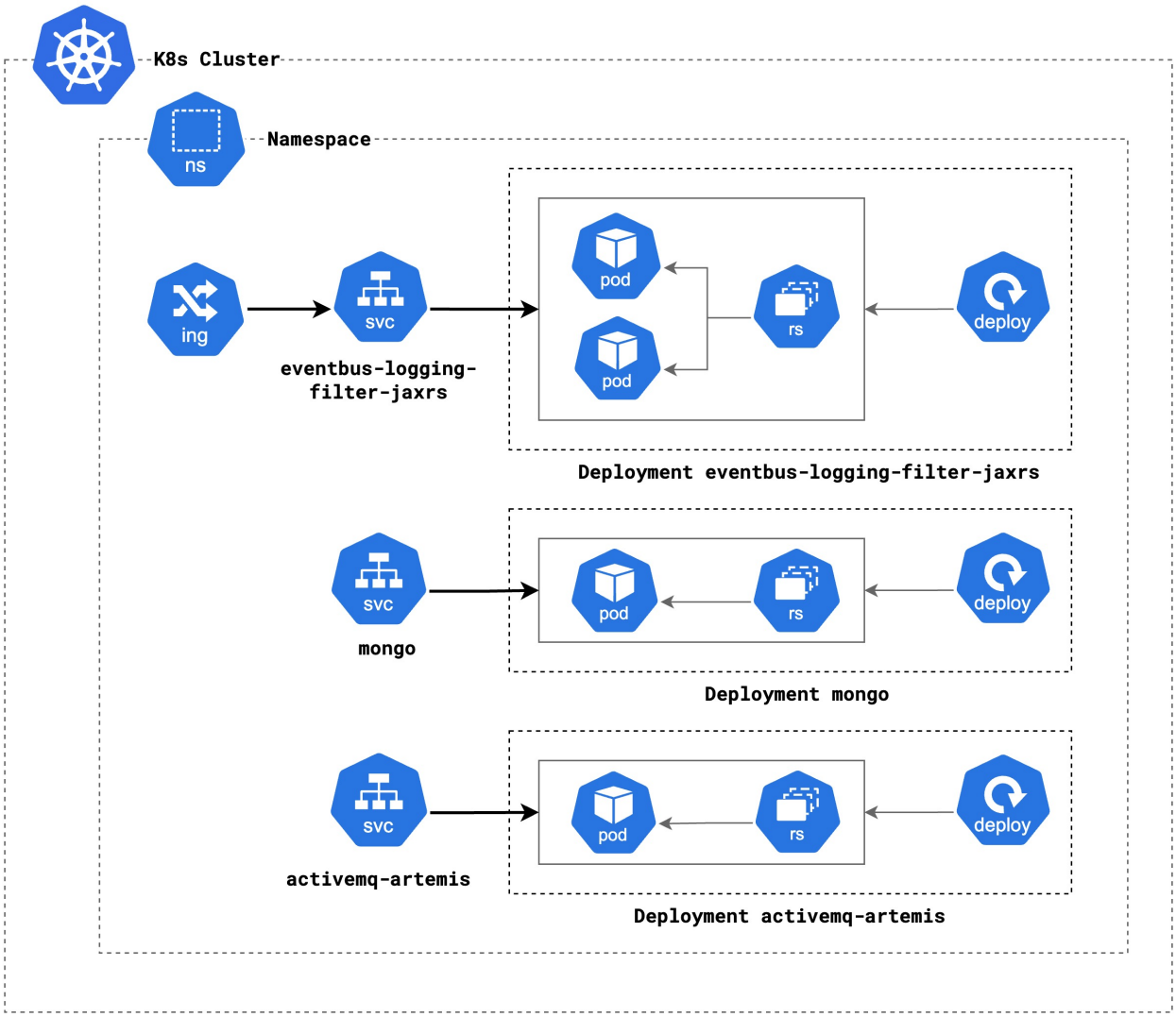


Figura 6 - *Architettura delle risorse coinvolte nel deploy dell'applicazione Quarkus su OpenShift*

Il diagramma è rappresentato da un Deployment che contiene il **Pod** (o i Pod) con il container dell'applicazione Quarkus, un **Service** per esporre il Pod (o i Pod) all'interno del cluster OpenShift e un **Route** (implementazione di OpenShift dell'Ingress) per esporre il Service all'esterno del cluster OpenShift. Inoltre, il diagramma mostra anche i servizi esterni come il database MongoDB e il broker AMQP che sono utilizzati dall'applicazione Quarkus.

Ricordiamo che all'interno di un cluster Kubernetes, i pod di ogni deployment comunicano tra loro attraverso il Service, e questo è importante ricordarlo al momento della configurazione dell'applicazione Quarkus e in particolare delle connessioni ai servizi esterni come il database MongoDB e il broker AMQP.

Nota: il plugin `quarkus-openshift` genererà automaticamente le risorse OpenShift/Kubernetes in base alla configurazione fornita dall'utente e alle impostazioni predefinite.

Perché dobbiamo aggiungere il supporto per le specifiche MicroProfile Health?

Il motivo è che l'estensione OpenShift di Quarkus utilizza le specifiche MicroProfile Health per verificare lo stato dell'applicazione e per determinare se l'applicazione è pronta per ricevere il traffico. Questa estensione la cui implementazione è fornita da SmallRye Health è un componente fondamentale per ottenere gratuitamente e sviluppare le probe di Liveness, Readiness e Startup Probe, concetti essenziali per le applicazioni cloud native, in particolare quando occorre orchestrare e gestire i container all'interno di un'infrastruttura come Kubernetes. Ecco un'analisi più dettagliata dell'importanza di ciascun tipo di probe.

1. **Liveness Probe**: è un meccanismo per determinare se un'applicazione è in esecuzione e funzionante. Se l'applicazione non è in esecuzione, Kubernetes la riavvierà. Questo è utile per evitare che un'applicazione non funzionante riceva traffico.
2. **Readiness Probe**: è un meccanismo per determinare se un'applicazione è pronta per ricevere il traffico. Se l'applicazione non è pronta, Kubernetes non invierà traffico all'applicazione. Questo è utile per evitare che un'applicazione non pronta riceva traffico.
3. **Startup Probe**: è un meccanismo per determinare se un'applicazione è stata avviata correttamente. Se l'applicazione non è stata avviata correttamente, Kubernetes la riavvierà. Questo è utile per evitare che un'applicazione non avviata correttamente riceva traffico.

Per aggiungere la dipendenza per le specifiche MicroProfile Health al progetto, è possibile utilizzare la CLI di Quarkus o Maven. A seguire il comando per aggiungere la dipendenza tramite la CLI di Quarkus o tramite Maven.

Console 21 - *Installazione estensione MicroProfile Health di Quarkus tramite CLI o Maven*

```
# Aggiungere la dipendenza per le specifiche MicroProfile Health
# tramite la CLI di Quarkus
quarkus extension add smallrye-health

# Aggiungere la dipendenza per le specifiche MicroProfile Health

# tramite il comando Maven
mvn quarkus:add-extension -Dextensions='smallrye-health'
```

Importando l'estensione smallrye-health, Quarkus genererà automaticamente le probe di Liveness, Readiness e Started per l'applicazione. Queste probe saranno utilizzate dall'estensione OpenShift per determinare se l'applicazione è pronta per ricevere il traffico. Avviando quindi l'applicazione, avremo a disposizione le seguenti probe.

1. **Liveness**: `/q/health/live` - questa probe è utilizzata per determinare se l'applicazione è in esecuzione e funzionante.
2. **Readiness**: `/q/health/ready` - questa probe è utilizzata per determinare se l'applicazione è pronta per ricevere il traffico e di conseguenza servire le richieste.
3. **Started**: `/q/health/started` - questa probe è utilizzata per determinare se l'applicazione è stata avviata correttamente.
4. **Health**: `/q/health` - questa probe è utilizzata per determinare lo stato generale dell'applicazione.

Interrogando le probe di Liveness e Readiness (utilizzando il cURL), otterremo rispettivamente i seguenti output. Tutti gli endpoint delle probe restituiscono un semplice JSON con lo stato (`status`) complessivo dell'applicazione e delle eventuali verifiche effettuate (`checks`). Nel caso della nostra applicazione, le verifiche effettuate sono relative alla connessione al database MongoDB e alla connessione al broker AMQP (che in questo caso riporta lo stato dei due channel per la pubblicazioni dei messaggi di richiesta e risposta JAX-RS).

Console 22 - *Interrogazione delle probe di Liveness e Readiness*

```
# Risposta della probe Liveness /q/health/live
{
    "status": "UP",
    "checks": [
        {
            "name": "SmallRye Reactive Messaging - liveness check",
            "status": "UP",
            "data": {
                "http-response-out": "[OK]",
                "http-request-out": "[OK]"
            }
        }
    ]
}

# Risposta della probe Liveness /q/health/ready
{
    "status": "UP",
    "checks": [
        {
            "name": "MongoDB connection health check",
            "status": "UP",
            "data": {
                "<default-reactive>": "OK"
```

```
            }
        },
        {
            "name": "SmallRye Reactive Messaging - readiness check",
            "status": "UP",
            "data": {
                "http-response-out": "[OK]",
                "http-request-out": "[OK]"
            }
        }
    ]
}
```

Console 22 - Risposta della probe di Liveness e Readiness

10.3. Primo deploy su OpenShift

Per effettuare il deploy dell'applicazione Quarkus su OpenShift, è necessario eseguire i seguenti passaggi:

1. effettuare il login sul cluster OpenShift fornito dalla Developer Sandbox di Red Hat;
2. build e deply dell'applicazione Quarkus su OpenShift;

Per eseguire il login sul cluster OpenShift fornito dalla Developer Sandbox di Red Hat, è necessario eseguire il comando `oc login` in due possibili modalità: tramite username e password o tramite token. A seguire i comandi per effettuare il login sul cluster OpenShift.

Console 23 - *Login sul cluster OpenShift fornito dalla Developer Sandbox di Red Hat*

```
# Login tramite username e password (richiesta dopo l'esecuzione del comando)
oc login -u myusername

# Login tramite token (via preferenziale)
oc login --token=<tuo-token> --server=<tuo-cluster>
```

Per ottenere il token di accesso al cluster OpenShift fornito dalla Developer Sandbox di Red Hat, è necessario accedere alla Developer Sandbox di Red Hat e cliccare sul pulsante `Copy Login Command` per copiare il comando di login che contiene il token di accesso al cluster OpenShift. A seguire un esempio di comando di login con token.

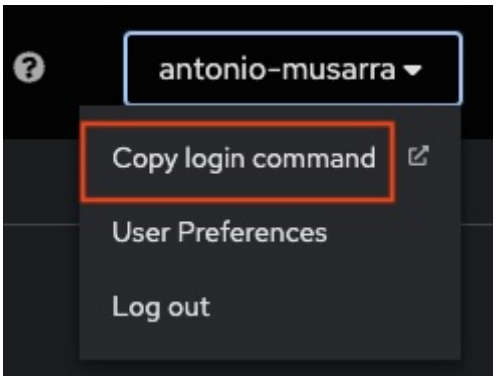


Figura 7 - *Come ottenere il comando di login con token*

Figura 8 - *Pagina che mostra il Token API e il comando di login tramite OC e cURL*

Se il login è andato a buon fine dovreste vedere un messaggio simile a quello mostrato a seguire, dov'è indicato il nome del cluster OpenShift, l'utente con cui avete effettuato il login e il progetto di default in cui siete loggati.

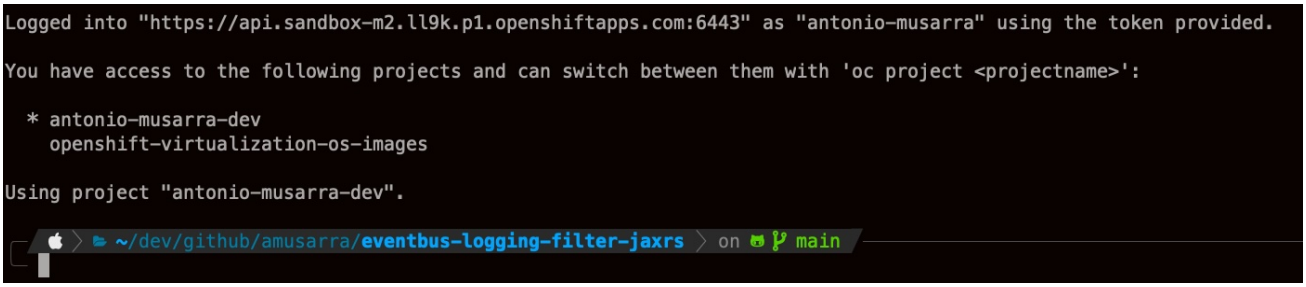


Figura 8 - *Messaggio di login completato con successo*

Dopo aver effettuato il login sul cluster OpenShift, è possibile procedere con il processo di build e deploy dell'applicazione Quarkus su OpenShift. Per fare ciò, è necessario eseguire il comando `mvn clean package -Dquarkus.openshift.deploy=true` che effettuerà la build dell'applicazione Quarkus e il deploy su OpenShift. A seguire un esempio di output (parziale) del comando di build e deploy che mostra il buon esito dell'operazione.

Ricordo che, è sempre possibile monitorare l'operazione di build e deploy dell'applicazione Quarkus su OpenShift tramite la console Web di OpenShift o tramite la CLI di OpenShift. Per chi volesse vedere l'intera operazione di build e deploy dell'applicazione Quarkus su OpenShift, è possibile visualizzare l'asciinema Primo deploy su OpenShift dell'applicazione Quarkus Event Bus Logging Filter JAX-RS.

Console 24 - *Output parziale del comando di build e deploy dell'applicazione Quarkus su OpenShift*

```
[INFO] Scanning for projects...
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Pushing
image image-registry.openshift-image-registry.svc:5000/antonio-musarra-dev/eventbus-
logging-filter-jaxrs:1.0.0-SNAPSHOT ...
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Getting
image source signatures
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Copying
blob sha256:dc4586ee36f78ddcdcf0f695ddfab9f607315ef196793fc7c00d96c196864290
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Copying
blob sha256:eca9236fb686825c1ec7ba1f1b339f6300ed2d4fffdf50611dde66cb8f6eeaa9
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Copying
blob sha256:dc35b837139a95d1b9f7f7b0435a024a74ab972416bdc248f3f608c9f917a753
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Copying
config sha256:9df58fd4ebf70122955dbc07d06435e22ab1b3425b06538927f0c6cc38f2dc62
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Writing
manifest to image destination
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor]
Successfully pushed image-registry.openshift-image-registry.svc:5000/antonio-
musarra-dev/eventbus-logging-filter-
jaxrs@sha256:eaaa9125c441238b6940326ebd218c1873a0a270ffd5a218926390e079916c2c
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Push
successful
```

```
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to openshift
server: https://api.sandbox-m2.ll9k.p1.openshiftapps.com:6443/ in namespace:
antonio-musarra-dev.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Service
eventbus-logging-filter-jaxrs.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream
openjdk-21.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream
eventbus-logging-filter-jaxrs.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: BuildConfig
eventbus-logging-filter-jaxrs.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Deployment
eventbus-logging-filter-jaxrs.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Route
eventbus-logging-filter-jaxrs.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] The deployed
application can be accessed at: http://eventbus-logging-filter-jaxrs-antonio-
musarra-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
235171ms
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  04:08 min
[INFO] Finished at: 2024-04-14T15:10:46+02:00
[INFO] ------------------------------------------------------------------------
```

L'operazione di build e in particolare quella di deploy potrebbe durare diversi minuti perché il plugin `quarkus-openshift` deve eseguire diverse operazioni sul cluster, tra cui la creazione di un certo numero di risorse quali: `Deployment`, `Pods`, `Service`, `Route`, `ConfigMap`,`Secret` oltre a l'immagine dell'applicazione Quarkus che viene caricata sul registry del cluster OpenShift. Da tenere anche in considerazione che la Developer Sandbox di Red Hat è un ambiente condiviso e quindi la velocità di risposta potrebbe variare in base al carico di lavoro del cluster. Nel mio caso l'operazione di build e deploy è durata circa 4 minuti.

Al termine dell'operazione di build e deploy, il comando di build e deploy restituirà l'URL dell'applicazione Quarkus su OpenShift. Per accedere all'applicazione Quarkus su OpenShift è sufficiente copiare l'URL e incollarlo nel browser.

Anche se build e deploy è stato completato con successo, **siete del tutto sicuri che l'applicazione sia disponibile?**

La risposta è negativa. Puntando alla URL dell'applicazione Quarkus, otterreste il messaggio di errore HTTP/503. Questo accade perché l'applicazione potrebbe non essere ancora pronta per ricevere il traffico. A seguire l'output mostrato dal comando `curl -I http://eventbus-logging-filter-jaxrs-antonio-musarra-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com` e dal browser che evidenziano proprio l'indisponibilità dell'applicazione.

Console 25 - *Output del comando cURL puntando all'applicazione dopo il primo deploy*

```
HTTP/1.1 503 Service Unavailable
pragma: no-cache
cache-control: private, max-age=0, no-cache, no-store
content-type: text/html
Date: Sun, 14 Apr 2024 13:46:49 GMT
Connection: close
```

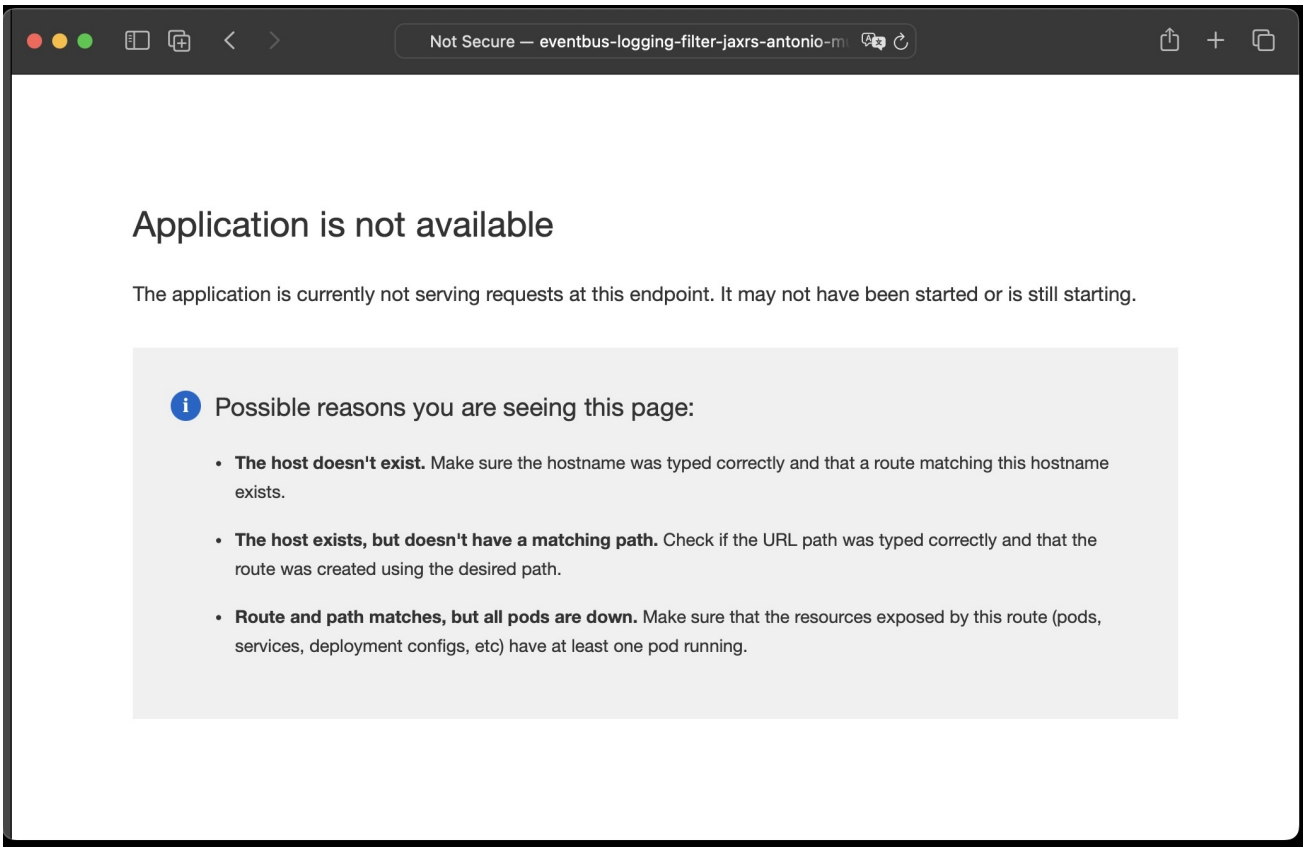


Figura 9 - *Messaggio di errore HTTP/503 mostrato dal browser per l'applicazione Quarkus non pronta*

È probabile che molti di voi abbiano già intuito il motivo dell'indisponibilità dell'applicazione ma cerchiamo in ogni caso di capire il motivo di questa indisponibilità utilizzando la CLI di OpenShift e il modo di risolvere il problema. In questi casi la prima azione da compiere che potrebbe senza dubbio dare l'indicazione di ciò che sia andato storto, è la verifica degli eventi che sono accaduti durante il deploy dell'applicazione. Per fare ciò, è necessario eseguire il comando `oc get events` che restituirà l'elenco degli eventi accaduti durante il deploy. A seguire l'output del comando `oc get events`.

Le informazioni riportate dal comando `oc get events` mostrano che l'applicazione non è pronta per ricevere il traffico (`Startup probe failed: HTTP probe failed with statuscode: 503`). Questo accade perché l'applicazione non ha superato le probe di Startup; tra le altre cose il pod è stato ucciso e riavviato più volte (`Back-off restarting failed container`).

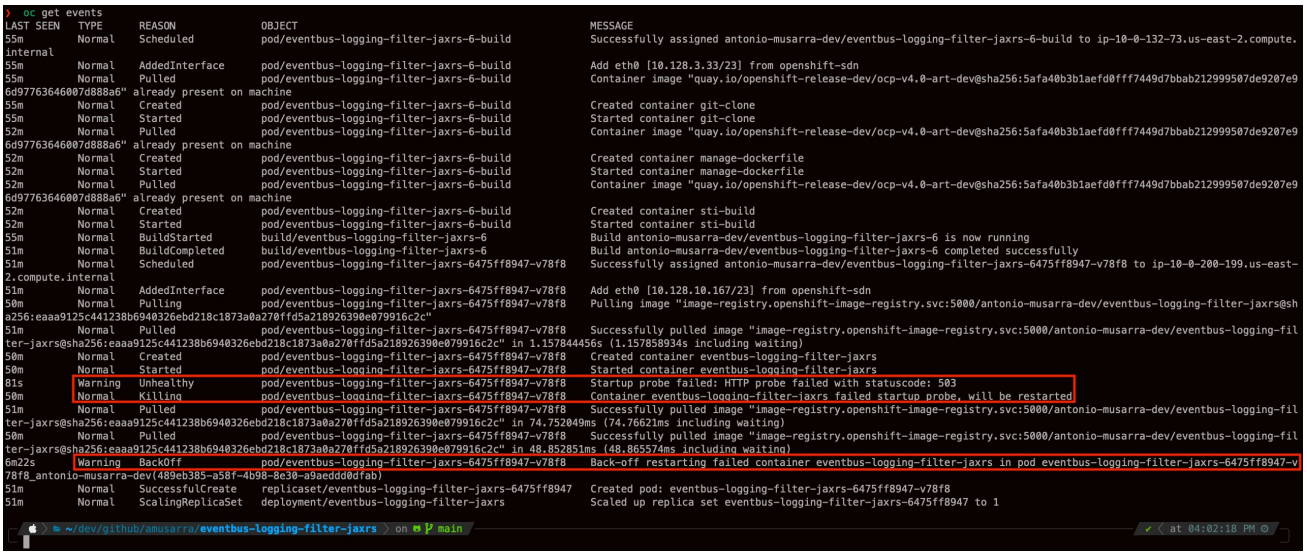


Figura 10 - *Output del comando* `oc get events` *per verificare il motivo per l'applicazione sia non pronta*

Visto che il problema è sul pod dell'applicazione, è necessario verificare il motivo per cui l'applicazione non è pronta per ricevere il traffico. In questo caso possiamo andare a vedere direttamente il log del pod dell'applicazione per capire il motivo per cui l'applicazione non è pronta per ricevere il traffico. Per fare ciò, è necessario eseguire il comando `oc logs <pod-name>` che restituirà il log del pod dell'applicazione.

Dai log è evidente il motivo della non disponibilità dell'applicazione, ovvero la mancata connessione al broker AMQP. Questo accade perché l'applicazione Quarkus non è riuscita a connettersi al broker AMQP e quindi non è pronta per ricevere il traffico. Questa evidenza è anche confermata dal log della probe di Startup che mostra il fallimento della connessione al broker AMQP.

Console 26 - *Log del pod dell'applicazione Quarkus per verificare il motivo per cui l'applicazione non è pronta*

```
2024-04-14 14:19:33,709 ERROR [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0)
SRMSG16215: Unable to connect to the broker, retry will be attempted:
io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused:
localhost/127.0.0.1:5672
Caused by: java.net.ConnectException: Connection refused
    at java.base/sun.nio.ch.Net.pollConnect(Native Method)
    at java.base/sun.nio.ch.Net.pollConnectNow(Net.java:682)
    at java.base/sun.nio.ch.SocketChannelImpl.finishConnect
(SocketChannelImpl.java:973)
    at io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect
(NioSocketChannel.java:337)
    at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnect
(AbstractNioChannel.java:339)
    at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:776)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized
(NioEventLoop.java:724)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:650)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:562)
```

```
    at io.netty.util.concurrent.SingleThreadEventExecutor$4.run
(SingleThreadEventExecutor.java:997)
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run
(FastThreadLocalRunnable.java:30)
    at java.base/java.lang.Thread.run(Thread.java:1583)

2024-04-14 14:19:36,100 INFO  [io.sma.rea.mes.amqp] (executor-thread-1) SRMSG16212:
Establishing connection with AMQP broker
2024-04-14 14:19:36,543 INFO  [io.sma.health] (vert.x-eventloop-thread-1)
SRHCK01001: Reporting health down status: {"status":"DOWN","checks":[{"name"
:"SmallRye Reactive Messaging - startup check","status":"DOWN","data":{"http-
response-in":"[KO]","http-response-out":"[KO]","http-request-in":"[KO]","http-
request-out":"[KO]"}}]}
2024-04-14 14:19:36,583 INFO  [io.quarkus] (Shutdown thread) eventbus-logging-
filter-jaxrs stopped in 0.024s
```

Dopo questa breve analisi le idee dovrebbero essere più chiare. Se ricordate bene, l'applicazione ha delle dipendenze esterne, verso il database NoSQL MongoDB e verso il broker AMQP Apache ActiveMQ Artemis. Fino a quando siamo stati nella fase di sviluppo dell'applicazione, queste dipendenze non ci hanno dato problemi perché eravamo in un ambiente controllato e locale e per di più abbiamo fatto uso dei Dev Services di Quarkus che hanno reso trasparente per noi l'uso di queste dipendenze.

Passando all'ambiente di "produzione" o comunque diverso dal nostro ambiente di sviluppo locale, le cose sono cambiate, queste dipendenze non sono più disponibili e il plugin `quarkus-openshift` non crea i descrittori necessari per creare tutte le risorse indispensabili per tirare su le dipendenze sull'ambiente di deploy OpenShift.

Se andassimo a vedere il contenuto del file `target/kubernetes/openshift.yml` , all'interno non troveremmo nessuna risorsa per il broker AMQP Apache ActiveMQ Artemis e per il database NoSQL MongoDB. Questo è il motivo per cui l'applicazione non è pronta per ricevere il traffico, poiché non riesce a connettersi al broker AMQP Apache ActiveMQ Artemis e quindi non è pronta per ricevere il traffico.

Per risolvere il problema occorre quindi, creare le risorse per il broker AMQP Apache ActiveMQ Artemis e per il database NoSQL MongoDB sul cluster OpenShift e configurare opportunamente l'applicazione Quarkus per connettersi a queste risorse.

10.4. Creare le risorse per il broker AMQP e MongoDB sul OpenShift

Iniziamo creando il broker AMQP partendo dall'immagine Docker di Apache ActiveMQ Artemis. Per fare ciò è sufficiente utilizzare il comando `oc new-app` specificando l'immagine Docker di Apache ActiveMQ Artemis.

Console 27 - *Creazione del broker AMQP Apache ActiveMQ Artemis su OpenShift partendo dall'immagine Docker*

```
# Creazione del broker AMQP Apache ActiveMQ Artemis
oc new-app apache/activemq-artemis:2.33.0

# Output del comando oc new-app
--> Found container image 1714b7a (3 weeks old) from Docker Hub for
"apache/activemq-artemis:2.33.0"

    * An image stream tag will be created as "activemq-artemis:2.33.0" that will
track this image

--> Creating resources ...
    imagestream.image.openshift.io "activemq-artemis" created
    deployment.apps "activemq-artemis" created
    service "activemq-artemis" created
--> Success
    Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
     'oc expose service/activemq-artemis'
    Run 'oc status' to view your app.
```

Se tutto è andato per il verso giusto, dovreste ottenere un output simile a quello a seguire che indica il pod dell'Apache ActiveMQ Artemis in stato `Running` e il servizio `activemq-artemis` attivo con in evidenza le porte in binding. A seguire l'output del comando `oc get pods` e `oc get services`.

Console 28 - *Output del comando* `oc get pods` *e* `oc get services` *per verificare la creazione delle risorse per AMQP*

```
NAME                                    READY   STATUS       RESTARTS   AGE
pod/activemq-artemis-f9584d88c-pgf68    1/1     Running      0          45m

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)       AGE
service/activemq-artemis            ClusterIP   172.30.35.104   <none>
1883/TCP,..   45m
```

Adesso passiamo alla creazione delle risorse per il database NoSQL MongoDB. Per fare ciò è sufficiente utilizzare il comando `oc new-app` specificando l'immagine Docker di MongoDB.

Console 29 - *Creazione del database NoSQL MongoDB su OpenShift partendo dall'immagine Docker*

```
# Creazione di MongoDB
oc new-app mongo:7.0.8

# Output del comando oc new-app
--> Found image fb4debd (11 days old) in image stream "antonio-musarra-dev/mongo"
under tag "7.0.8" for "mongo:7.0.8"


--> Creating resources ...
    deployment.apps "mongo" created
    service "mongo" created
--> Success
    Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
     'oc expose service/mongo'
    Run 'oc status' to view your app.
```

Per verificare che le risorse per il database NoSQL MongoDB siano state create correttamente, è possibile utilizzare il comando `oc get pods` e `oc get services` che restituiranno l'output mostrato a seguire.

Console 30 - *Output del comando* `oc get pods` *e* `oc get services` *per verificare la creazione delle risorse per MongoDB*

```
NAME                                          READY   STATUS    RESTARTS
AGE
pod/mongo-dc76d7f8d-p7dv7                     1/1     Running   0
9m40s

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
service/mongo                       ClusterIP   172.30.248.66   <none>
27017/TCP  9m40s
```

Abbiamo creato con successo le risorse per il broker AMQP Apache ActiveMQ Artemis e per il database NoSQL MongoDB sul cluster OpenShift. Ora è necessario configurare l'applicazione Quarkus per connettersi a queste risorse e quindi essere pronta per ricevere il traffico.

Nota: per la creazione delle risorse abbiamo usato la via più semplice e immediata ma non quella consigliata per un ambiente di collaudo, validazione e produzione dove è necessario configurare le risorse in modo più dettagliato e sicuro, come per esempio attraverso l'uso di Helm Charts od Operator e all'interno di un progetto di GitOps.

10.5. Configurare l'applicazione Quarkus per connettersi alle risorse

La configurazione dell'applicazione Quarkus per connettersi alle risorse create sul cluster OpenShift è abbastanza semplice e richiede solo di aggiungere le informazioni di connessione alle risorse nel file di configurazione `application.properties` dell'applicazione Quarkus.

Per questa configurazione adotteremo la strategia di usare ConfigMap e Secret per la configurazione delle risorse esterne come il broker AMQP Apache ActiveMQ Artemis e il database NoSQL MongoDB. Questa strategia è molto utile perché permette di separare le informazioni di configurazione dall'applicazione e di gestirle in modo centralizzato, e in particolare di gestire le informazioni sensibili come username e password in modo sicuro.

Per fare ciò, è necessario creare due risorse, una di tipo ConfigMap e due di tipo Secret, per contenere le informazioni di configurazione del broker AMQP Apache ActiveMQ Artemis e del database NoSQL MongoDB. I descrittore YAML delle risorse ConfigMap e Secret li dobbiamo scrivere noi perché non se occuperà il plugin `quarkus-openshift`; quest'ultimo si occuperà però di aggiungere queste risorse allo yaml finale di deploy dell'applicazione Quarkus (che ricordiamo essere posizionato in `target/kubernetes/openshift.yml`).

Detto ciò, andremo a posizionare il file `common.yml` all'interno del folder `src/main/kubernets`. A seguire il descrittore YAML per la ConfigMap e per il Secret.

Configurazione 8 - *Descrittore YAML per la ConfigMap e il Secret per il broker AMQP Apache ActiveMQ Artemis e per il database NoSQL MongoDB*

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: amqp-hostname-port
data:
  amqp-host: activemq-artemis
  amqp-port: 5672
binaryData: {}
immutable: false
---
kind: Secret
apiVersion: v1
metadata:
  name: amqp-username-password
data:
  amqp-password: YXJ0ZW1pcw==
  amqp-username: YXJ0ZW1pcw==
type: Opaque
---
kind: Secret
apiVersion: v1
metadata:
  name: mongodb
data:
  mongodb-connection-url: bW9uZ29kYjovL21vbmdvOjI3MDE3L2F1ZGl0Cg==
type: Opaque
```

La configurazione del file `application.properties` dell'applicazione Quarkus per connettersi al broker AMQP e a MongoDB, deve essere quindi modificata come segue. In particolare:

1. la proprietà `quarkus.openshift.env.configmaps` deve contenere il nome della ConfigMap che preserva l'hostname e la porta del broker AMQP;
2. la proprietà `quarkus.openshift.env.secrets` deve contenere il nome del Secret dedicato al broker AMQP e al database NoSQL MongoDB;
3. le proprietà `quarkus.amqp-host`, `quarkus.amqp-port`, `quarkus.amqp-username`, `quarkus.amqp-password` sono utilizzate per la connessione al broker AMQP i cui valori sono forniti dalla ConfigMap e dal Secret;
4. la proprietà `quarkus.mongodb.connection-string` è utilizzata per la connessione al database NoSQL MongoDB i cui valori sono forniti dal Secret.

Configurazione 9 - *Configurazione per connettersi al broker AMQP e al database NoSQL MongoDB*

```
# The name of the ConfigMap that contains the AMQP host, port
quarkus.openshift.env.configmaps=amqp-hostname-port

# The name of the Secret that contains the AMQP username and password
# and the MongoDB connection URL (that will be contained the username and password)
quarkus.openshift.env.secrets=amqp-username-password,mongodb

# AMQP configuration for production profile
# The AMQP host, port, username, and password
# are provided by the ConfigMap and Secret.
# See the src/kubernetes/common.yaml file for more details.
%prod.amqp-host=${AMQP_HOSTNAME}
%prod.amqp-port=${AMQP_PORT}
%prod.amqp-username=${AMQP_USERNAME}
%prod.amqp-password=${AMQP_PASSWORD}

# Configure the mongoDB client for a single instance on localhost
# are privided by the Secret.
# See the src/kubernetes/common.yaml file for more details.
%prod.quarkus.mongodb.connection-string=${MONGODB_CONNECTION_URL}
```

I valori delle proprietà `quarkus.amqp-host`, `quarkus.amqp-port`, `quarkus.amqp-username`, `quarkus.amqp-password` e `quarkus.mongodb.connection-string` sono forniti dalla ConfigMap e dal Secret come variabili d'ambiente, e sono accessibili nella forma `${NOME_VARIABILE}` all'interno del file `application.properties`. La magia di questo meccanismo parte da qui, esattamente dalla riga 4 alla riga 10 dell'estratto del file `openshift.yml` generato dal plugin `quarkus-openshift`.

Configurazione 10 - *Estratto del file* `openshift.yml` *generato dal plugin* `quarkus-openshift`

```
    - env:
        - name: JAVA_APP_JAR
          value: /deployments/quarkus-run.jar
      envFrom:
        - secretRef:
            name: amqp-username-password
        - secretRef:
            name: mongodb
        - configMapRef:
            name: amqp-hostname-port
      image: eventbus-logging-filter-jaxrs:1.0.0-SNAPSHOT
```

Le informazioni autenticazione, come username e password sono quelle di default per l'immagine Docker di Apache ActiveMQ Artemis e di MongoDB. Ovviamente questa non è una buona pratica e dovreste cambiare queste informazioni con quelle che ritenete più sicure per il vostro ambiente di produzione.

10.6. Deploy finale dell'applicazione su OpenShift

Adesso che abbiamo configurato le due risorse mancanti, il broker AMQP e MongoDB, è possibile procedere con il deploy finale dell'applicazione Quarkus su OpenShift. Per fare ciò, è necessario eseguire il comando `mvn clean package -Dquarkus.openshift.deploy=true` o `quarkus build -Dquarkus.openshift.deploy=true` che effettuerà la build dell'applicazione Quarkus e il deploy su OpenShift.

A meno di "catastrofi", l'operazione di build e deploy dovrebbe andare a buon fine e l'applicazione Quarkus dovrebbe essere pronta per ricevere il traffico. Potete fare un veloce controllo utilizzando il comando `oc get pods` che adesso dovrebbe restituire l'output mostrato a seguire.

Da notare che l'applicazione è pronta per ricevere il traffico, come indicato dallo stato `Running` del pod dell'applicazione `eventbus-logging-filter-jaxrs-7f8b5cd8b7-4z4ms`.

Console 31 - *Output del comando* `oc get pods` *per verificare che l'applicazione sia pronta*

```
NAME                                          READY   STATUS    RESTARTS   AGE
activemq-artemis-f9584d88c-pgf68              1/1     Running   0          144m
eventbus-logging-filter-jaxrs-7f8b5cd8b7-4z4ms  1/1     Running   0          7m26s
mongo-dc76d7f8d-p7dv7                         1/1     Running   0          58m
```

Adesso, è quindi possibile chiamare l'API del servizio di echo e aspettarci una risposta positiva.

Console 32 - *Output del comando cURL per chiamare l'API del servizio di echo*

```
# Chiamata all'API del servizio di echo tramite cURL
curl -v -H "Content-Type: application/json" \
    -d '{"message": "Test di tracking richiesta JAX-RS su Dev Sandbox OpenShift"}' \
    http://eventbus-logging-filter-jaxrs-antonio-musarra-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com/api/rest/echo

= Output del comando cURL

* Trying 18.220.238.101:80...
* Connected to eventbus-logging-filter-jaxrs-antonio-musarra-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com (18.220.238.101) port 80
> POST /api/rest/echo HTTP/1.1
> Host: eventbus-logging-filter-jaxrs-antonio-musarra-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com
> User-Agent: curl/8.4.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 73
>
< HTTP/1.1 200 OK
< content-type: application/json;charset=UTF-8
```

```
< content-length: 73
< set-cookie: user_tracking_id=9319d019-fc84-4077-b9c4-005c896e8fcb;Version=1
;Comment="Cookie di tracciamento dell'utente";Path=/;Max-Age=2592000
< x-correlation-id: cefda5e4-346f-46f6-8494-a621e634b1bc
< set-cookie: 7eb8860bc15109552408e7020c87e397=a31130c8ef37120c72b3115bf77a7334;
path=/; HttpOnly
< Date: Sun, 14 Apr 2024 18:32:09 GMT
< Connection: keep-alive
<
* Connection #0 to host eventbus-logging-filter-jaxrs-antonio-musarra-
dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com left intact
{"message": "Test di tracking richiesta JAX-RS su Dev Sandbox OpenShift"}
```

Ottimo! Il servizio di echo ha risposto correttamente alla richiesta HTTP e come aspettato. Questo significa che l'applicazione Quarkus è pronta per ricevere il traffico; tramite il comando `oc logs <nome-del-pod>` possiamo verificare che il sistema di tracciamento delle richieste e delle risposte HTTP funzioni correttamente.

Dall'estratto dai log del pod dell'applicazione a seguire, possiamo vedere che il sistema di tracciamento delle richieste verso il broker AMQP e il database NoSQL MongoDB sta funzionando correttamete.

Console 33 - *Estratto dai log del pod dell'applicazione Quarkus per verificare il funzionamento del sistema di tracciamento*

```
2024-04-14 18:32:09,370 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: nosql-trace with result:
Documents inserted successfully with Id BsonObjectId{value=661c21291140b32ae7b43ea9}
2024-04-14 18:32:09,371 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: nosql-trace with result:
Documents inserted successfully with Id BsonObjectId{value=661c21291140b32ae7b43eaa}
2024-04-14 18:32:09,377 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: queue-trace with result:
Message sent to AMQP queue successfully!
2024-04-14 18:32:09,378 DEBUG [it.don.eve.con.eve.han.Dispatcher] (vert.x-eventloop-
thread-0) Received response from target virtual address: queue-trace with result:
Message sent to AMQP queue successfully!
```

Lascio a voi il divertimento di fare le verifiche su MongoDB e AMQP per vedere se i dati sono stati effettivamente inseriti e se i messaggi sono correttamente in coda.

Lavoro finito; l'applicazione che abbiamo sviluppato è cloud native!

10.7. Diamo una spinta all'applicazione Quarkus

Per concludere, possiamo dare una spinta all'applicazione Quarkus per renderla più performante e scalabile. È probabile che qualcuno di voi pensi già alla possibilità di scalare l'applicazione in modo orizzontale per gestire un carico di lavoro più elevato, e quindi agire attraverso il comando `oc scale --replicas=3 deployment/eventbus-logging-filter-jaxrs` che scalerà l'applicazione Quarkus a 3 repliche e trovarsi in una situazione simile a quella mostrata a seguire; ma non è questa l'idea della spinta o del boost. **Allora, qual è l'idea?**

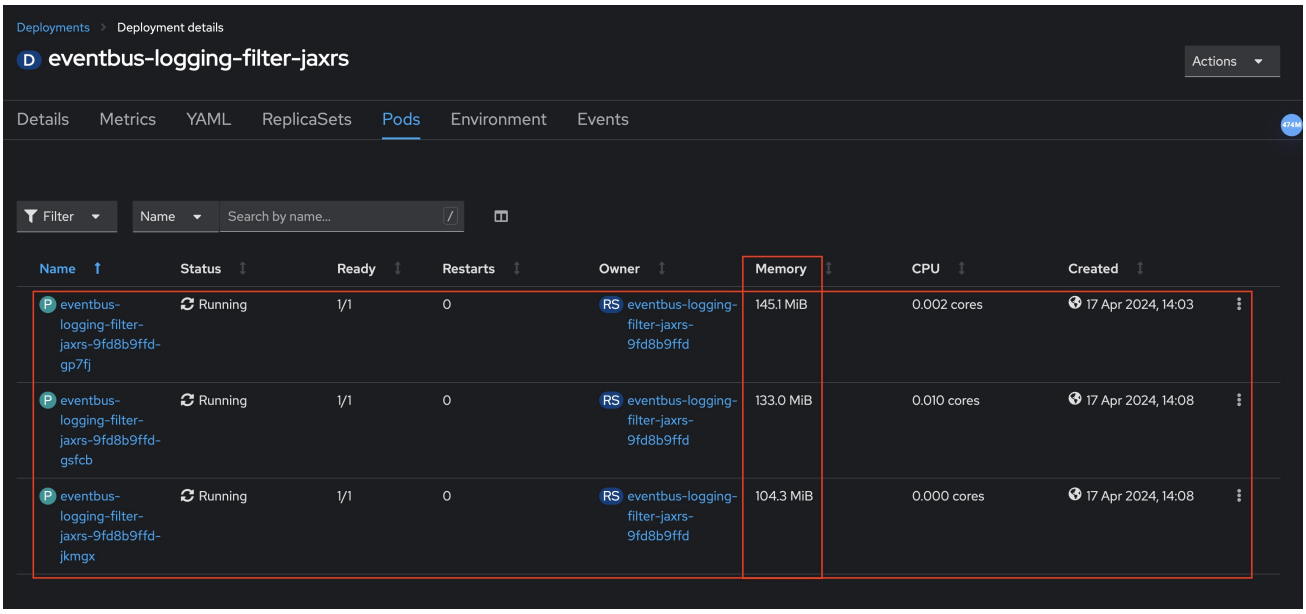


Figura 11 - *Status del deployment dell'applicazione Quarkus con 3 repliche (vista dalla console di OCP*

La spinta o il boost che vogliamo dare all'applicazione Quarkus, la possiamo ottenere attivando GraalVM per la compilazione nativa dell'applicazione Quarkus. La compilazione nativa dell'applicazione Quarkus con GraalVM è una delle caratteristiche più interessanti di Quarkus perché consente di creare un eseguibile nativo dell'applicazione che è più veloce e leggero rispetto all'eseguibile Java.

L'eseguibile nativo per la nostra applicazione conterrà il codice dell'applicazione, le librerie richieste, le API Java e una versione ridotta di una VM che migliora il tempo di avvio dell'applicazione e produce un ingombro minimo sul disco. Il diagramma mostrato in Figura 13 illustra il processo di creazione di un eseguibile nativo con GraalVM.

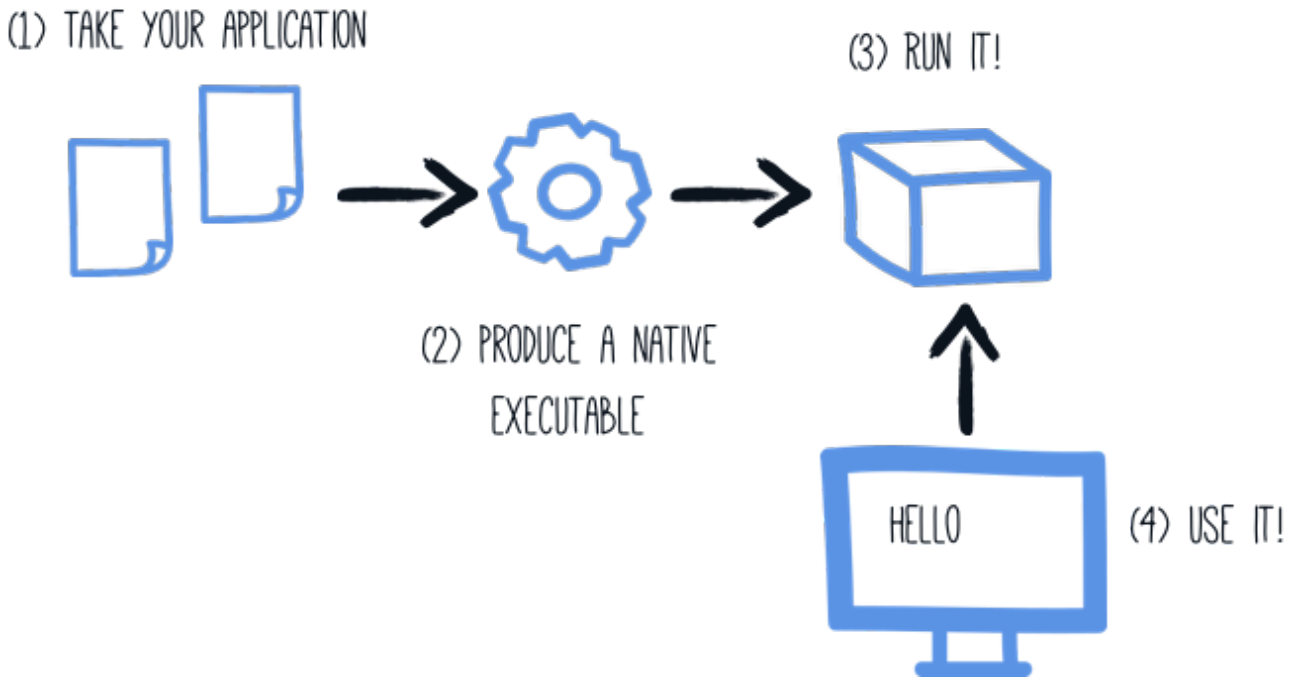


Figura 12 - *Processo di creazione di un eseguibile nativo con GraalVM (immagine tratta da Quarkus.io)*

Andando sul pratico, vediamo prima lo start-up time dell'applicazione Quarkus in modalità JVM che possiamo vedere direttamente dai log dell'applicazione usando il comando `oc logs <nome-del-pod>`. In particolare, l'output del comando `oc logs <nome-del-pod>` mostra che l'applicazione Quarkus in modalità JVM ha un tempo di avvio di 2,646 secondi, come mostrato a seguire.

```
❯ oc logs eventbus-logging-filter-jaxrs-9fd8b9ffd-gp7fj
Starting the Java application using /opt/jboss/container/java/run/run-java.sh ...
INFO exec -a "java" java -XX:MaxRAMPercentage=80.0 -XX:+UseParallelGC -XX:MinHeapFreeRatio=10 -XX:MaxHeapFreeRatio=20 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyW
eight=90 -XX:+ExitOnOutOfMemoryError -cp "." -jar /deployments/quarkus-run.jar
INFO running in /deployments
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
                                                    Powered by Quarkus 3.9.2
2024-04-17 12:03:31,246 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-response-in
2024-04-17 12:03:31,357 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-request-in
2024-04-17 12:03:31,359 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-response-out
2024-04-17 12:03:31,367 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-request-out
2024-04-17 12:03:31,464 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 12:03:31,474 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 12:03:31,481 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 12:03:31,482 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 12:03:31,484 DEBUG [it.don.eve.con.eve.han.que.AmqpEventHandler] (main) Registering the AMQP event handler at addresses: {queue-trace}
2024-04-17 12:03:31,491 DEBUG [it.don.eve.con.eve.han.Dispatcher] (main) Registering the Dispatcher to the event bus for the event handler at addresses: {dispa
tcher-event}
2024-04-17 12:03:31,492 DEBUG [it.don.eve.con.eve.han.nos.MongoDbEventHandler] (main) Registering the MongoDB event handler at addresses: {nosql-trace}
2024-04-17 12:03:31,493 DEBUG [it.don.eve.con.htt.HttpResponseConsumer] (main) Registering the consumers to the event bus for HTTP response at addresses: {http
-response}
2024-04-17 12:03:31,541 DEBUG [it.don.eve.con.htt.HttpRequestConsumer] (main) Registering the consumers to the event bus for HTTP request at addresses: {http-r
equest}
2024-04-17 12:03:31,760 INFO  [io.quarkus] (main) eventbus-logging-filter-jaxrs 1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.9.2) started in 2.646s. Listening o
n: http://0.0.0.0:8080
2024-04-17 12:03:31,761 INFO  [io.quarkus] (main) Profile prod activated.
2024-04-17 12:03:31,761 INFO  [io.quarkus] (main) Installed features: [cdi, hibernate-validator, kubernetes, messaging, messaging-amqp, mongodb-client, rest, r
est-jackson, smallrye-context-propagation, smallrye-health, vertx]
```

Figura 13 - *Start-up time dell'applicazione Quarkus in modalità JVM*

Adesso procediamo con la compilazione dell'applicazione Quarkus in modalità nativa con GraalVM e deploy su OpenShift. Per fare ciò, è possibile proseguire in diversi modi e in ogni caso assicurati di avere un ambiente di container runtime (come Docker o Podman) funzionante.

Console 34 - *Compilazione dell'applicazione Quarkus in modalità nativa con GraalVM e deploy su OpenShift*

```
# Compilazione dell'applicazione Quarkus in modalità nativa con GraalVM
# e deploy su OpenShift. In questo caso è obbligatorio avere GraalVM installato.
# 1. Tramite Maven
mvn clean package -Pnative -Dquarkus.openshift.deploy=true

# 2. Tramite Quarkus CLI
quarkus build -Pnative -Dquarkus.openshift.deploy=true

# Compilazione dell'applicazione Quarkus in modalità nativa con l'immagine GraalVM
# e deploy su OpenShift. In questo caso non è obbligatorio avere GraalVM installato.
# 3. Tramite Maven
mvn clean package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.openshift.deploy=true

# 4. Tramite Quarkus CLI
quarkus build -Pnative -Dquarkus.native.container-build=true -
Dquarkus.openshift.deploy=true
```

Molto spesso è necessario solo creare un eseguibile Linux nativo per la propria applicazione Quarkus (ad esempio per eseguirla in un ambiente containerizzato come quello di OpenShift) e si vorrebbe evitare il problema di installare la versione GraalVM corretta per eseguire questa attività (ad esempio, negli ambienti CI è pratica comune installare il minor numero di software possibile). I comandi 3 e 4 mostrati sopra, consentono di creare un eseguibile Linux nativo per la propria applicazione Quarkus senza dover installare GraalVM.

Nel mio caso specifico, dove l'ambiente di sviluppo è macOS su Apple Silicon M1 Max con 32GByte di RAM, ho dovuto usare una strategia leggermente diversa per creare l'eseguibile Linux. In particolare:

1. ho creato una VM utilizzando Colima impostando un template basato macOS Virtualization Framework per avere il supporto per l'architettura amd64. La macchina è stata creata con 60GByte di spazio disco, 4GByte di RAM e 4 core CPU;
2. ho eseguito il pull dell'immagine Docker di Mandrel Builder specificando la piattaforma linux/amd64 utilizzando il comando `docker pull quay.io/quarkus/ubi-quarkus-mandrel-builder-image:jdk-21 --platform linux/amd64`;
3. ho eseguito la build dell'applicazione Quarkus in modalità nativa con GraalVM e deploy su OpenShift utilizzando il comando `quarkus build --native --no-tests -Dquarkus.native.container-build=true -Dquarkus.native.builder-image.pull=never -Dquarkus.openshift.deploy=true`.

Nota: Colima è un'applicazione open-source che consente di creare e gestire macchine virtuali su macOS utilizzando il framework di virtualizzazione di macOS. Il vantaggio di Colima è che consente di creare macchine virtuali con architetture diverse da quella dell'host e questo grazie al framework di virtualizzazione di macOS che sfrutta Rosetta 2. L'articolo Come avviare un'istanza SQL Server su macOS Apple Silicon spiega come installare e configurare Colima su macOS Apple Silicon M1.

Console 35 - *Comando per creare l'eseguibile Linux nativo per l'applicazione Quarkus*

```
# Comando per creare l'eseguibile Linux nativo per l'applicazione Quarkus
# tramite Mandrel Builder eseguito su una VM con architettura ibrida aarch64/amd64
# grazie a Colima.
quarkus build --native --no-tests \
    -Dquarkus.native.container-build=true \
    -Dquarkus.native.builder-image.pull=never \
    -Dquarkus.openshift.deploy=true
```

Il flag usati nel comando `quarkus build` sono i seguenti:

- il parametro `--no-tests` è stato utilizzato per evitare l'esecuzione dei test durante la build dell'applicazione Quarkus in modalità nativa con GraalVM;
- il parametro `--native` è stato utilizzato per specificare che la build dell'applicazione Quarkus deve essere eseguita in modalità nativa con GraalVM;
- il parametro `-Dquarkus.native.container-build=true` è stato utilizzato per specificare che la build dell'applicazione Quarkus deve essere eseguita all'interno di un container;
- il parametro `-Dquarkus.native.builder-image.pull=never` è stato utilizzato per specificare che l'immagine di Mandrel Builder non deve essere scaricata perché scaricata in precedenza con il comando `podman pull`; Non specificando questo parametro, il comando `quarkus build` cercherà di scaricare l'immagine di Mandrel Builder e prenderà quella per l'architettura corrente, che nel mio caso è aarch64 e non amd64, di conseguenza l'eseguibile prodotto sarebbe stato per l'architettura sbagliata;
- Il parametro `-Dquarkus.openshift.deploy=true` è stato utilizzato per specificare che l'applicazione Quarkus deve essere deployata su OpenShift.

Nel caso in cui volessimo passare dei parametri a GraalVM, potremmo farlo utilizzando `-Dquarkus.native.additional-build-args`, specificando i parametri di nostro interesse.

Il processo di build crea un eseguibile Linux nativo dell'applicazione all'interno della cartella `target` con il nome `eventbus-logging-filter-jaxrs-1.0.0-SNAPSHOT-runner`. Questo eseguibile è pronto per essere deployato su OpenShift e per essere eseguito in un container. I più curiosi possono verificare che l'eseguibile sia effettivamente stato creato per Linux/x86-64 utilizzando il comando `file`. A seguire l'output del comando `file`.

Console 36 - *Output del comando* `file` *per verificare che l'eseguibile sia un eseguibile Linux nativo*

```
eventbus-logging-filter-jaxrs-1.0.0-SNAPSHOT-runner: ELF 64-bit LSB executable, x86-
64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 3.2.0, BuildID[sha1]=11bbeea7b79b4df5b3b075cc3ff7c4cac330e5f1, not
stripped
```

Su asciiema è disponibile Build Event Bus Logging Filter JAX-RS as Native (linux/amd64), l'intero processo di build e deploy dell'applicazione Quarkus in modalità nativa con GraalVM su OpenShift la cui durata è di circa 5 minuti e invito a guardarlo per avere un'idea più chiara del processo.

Asciinema 2 - *Processo di build e deploy dell'applicazione Quarkus in modalità nativa con GraalVM su OpenShift*

Al termine della build dell'applicazione Quarkus in modalità nativa con GraalVM, è possibile verificare che l'applicazione Quarkus sia stata deployata correttamente su OpenShift utilizzando il comando `oc get pods` e successivamente tramite il comando `oc logs <nome-del-pod>` per verificare i tempi di start-up dell'applicazione Quarkus in modalità nativa che ci aspettiamo siano molto più bassi rispetto alla modalità JVM.

```
❯ oc logs eventbus-logging-filter-jaxrs-5656c8d89d-cz7zb
QUARKUS_OPTS environment variable was not set, using default values of -Xmx24M -Xms16M -Xmn24M
                                                                                Powered by Quarkus 3.9.2
2024-04-17 13:33:50,133 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-response-in
2024-04-17 13:33:50,134 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-request-in
2024-04-17 13:33:50,135 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-response-out
2024-04-17 13:33:50,136 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16201: AMQP broker configured to activemq-artemis:5672 for channel http-request-out
2024-04-17 13:33:50,137 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 13:33:50,138 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 13:33:50,138 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 13:33:50,138 INFO  [io.sma.rea.mes.amqp] (main) SRMSG16212: Establishing connection with AMQP broker
2024-04-17 13:33:50,139 DEBUG [it.don.eve.con.eve.han.nos.MongoDbEventHandler] (main) Registering the MongoDB event handler at addresses: {nosql-trace}
2024-04-17 13:33:50,139 DEBUG [it.don.eve.con.htt.HttpRequestConsumer] (main) Registering the consumers to the event bus for HTTP request at addresses: {http-r
equest}
2024-04-17 13:33:50,139 DEBUG [it.don.eve.con.eve.han.Dispatcher] (main) Registering the Dispatcher to the event bus for the event handler at addresses: {dispa
tcher-event}
2024-04-17 13:33:50,139 DEBUG [it.don.eve.con.eve.han.que.AmqpEventHandler] (main) Registering the AMQP event handler at addresses: {queue-trace}
2024-04-17 13:33:50,139 DEBUG [it.don.eve.con.htt.HttpResponseConsumer] (main) Registering the consumers to the event bus for HTTP response at addresses: {http
-response}
2024-04-17 13:33:50,142 INFO  [io.quarkus] (main) eventbus-logging-filter-jaxrs 1.0.0-SNAPSHOT native (powered by Quarkus 3.9.2) started in 0.035s. Listening o
n: http://0.0.0.0:8080
2024-04-17 13:33:50,142 INFO  [io.quarkus] (main) Profile prod activated.
2024-04-17 13:33:50,142 INFO  [io.quarkus] (main) Installed features: [cdi, hibernate-validator, kubernetes, messaging, messaging-amqp, mongodb-client, rest, r
est-jackson, smallrye-context-propagation, smallrye-health, vertx]
2024-04-17 13:33:50,151 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16213: Connection with AMQP broker established
2024-04-17 13:33:50,151 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16213: Connection with AMQP broker established
2024-04-17 13:33:50,151 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16213: Connection with AMQP broker established
2024-04-17 13:33:50,152 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16213: Connection with AMQP broker established
2024-04-17 13:33:50,154 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16203: AMQP Receiver listening address http-request-in
2024-04-17 13:33:50,154 INFO  [io.sma.rea.mes.amqp] (vert.x-eventloop-thread-0) SRMSG16203: AMQP Receiver listening address http-response-in
```

Figura 14 - *Start-up time dell'applicazione Quarkus in modalità nativa*

Woa! Come previsto, il tempo di start-up dell'applicazione Quarkus in modalità nativa è di **0,035 secondi**, un tempo di start-up molto più basso rispetto alla modalità JVM (siamo vicini al 100%). Questo significa che l'applicazione in modalità nativa è più veloce in start-up e leggera rispetto alla modalità JVM, e quindi più adatta per gestire un carico di lavoro più elevato. Dalla figura a seguire fate attenzione alla memoria usata dall'applicazione Quarkus compilata in nativo, decisamente un bel risparmio rispetto alla versione JVM.

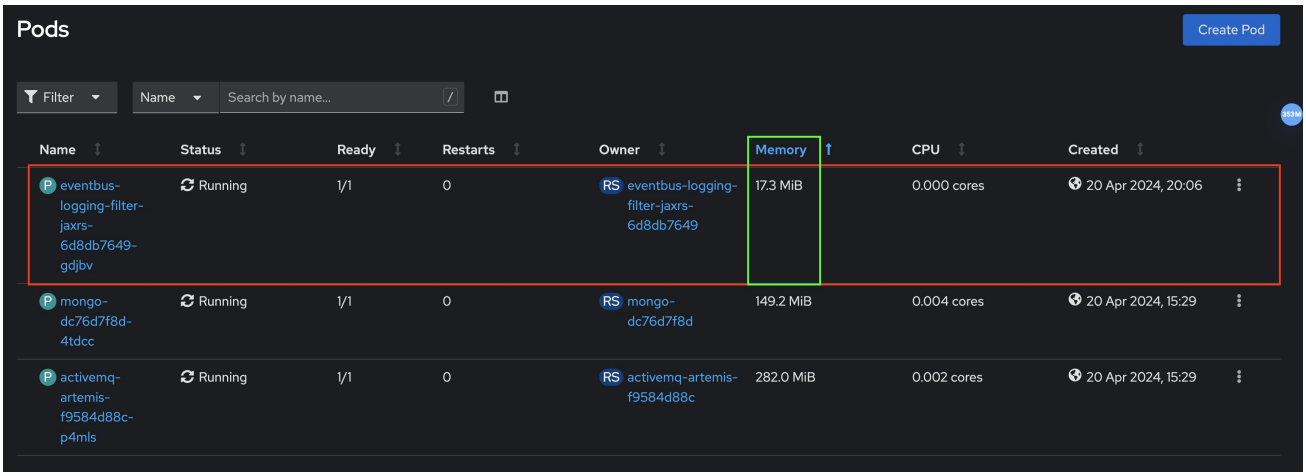


Figura 15 - *Status del deployment dell'applicazione Quarkus in modalità nativa (vista dalla console di OCP*

Togliamoci la soddisfazione di chiamare l'API del servizio di echo e verificare che l'applicazione Quarkus in modalità nativa funzioni correttamente e risponda in un tempo minore rispetto alla modalità JVM. Facciamo un veloce test con il comando ab (Apache HTTP server benchmarking tool) per verificare le prestazioni dell'applicazione.

Console 37 - *Esecuzione del comando ab per testare le prestazioni dell'applicazione Quarkus in modalità nativa*

```
# Esecuzione del comando ab per testare le prestazioni dell'applicazione Quarkus in
modalità nativa.
# In questo caso il comando ab esegue 100 richieste totali con 5 concorrenti.
ab -n 100 -c 5 -T 'application/json' -k \
  -p src/test/resources/payload-1.json \
  http://eventbus-logging-filter-jaxrs-antonio-musarra-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com/api/rest/echo
```

Anche qui i risultati sono stati molto positivi, l'applicazione Quarkus in modalità nativa ha risposto correttamente alle richieste HTTP e ha dimostrato di essere più veloce rispetto alla modalità JVM.

La versione nativa la cui build è stata effettuata con GraalVM (non utilizzando nessun parametro di ottimizzazione ulteriore) è risultata essere più veloce anche sotto stress rispetto alla versione JVM e tutto questo senza apportare alcuna modifica al codice sorgente dell'applicazione Quarkus. Sono sicuro che con un pò di attività di ottimizzazione e tuning, si possano ottenere risultati ancora migliori, ma questo è un argomento per un altro articolo.

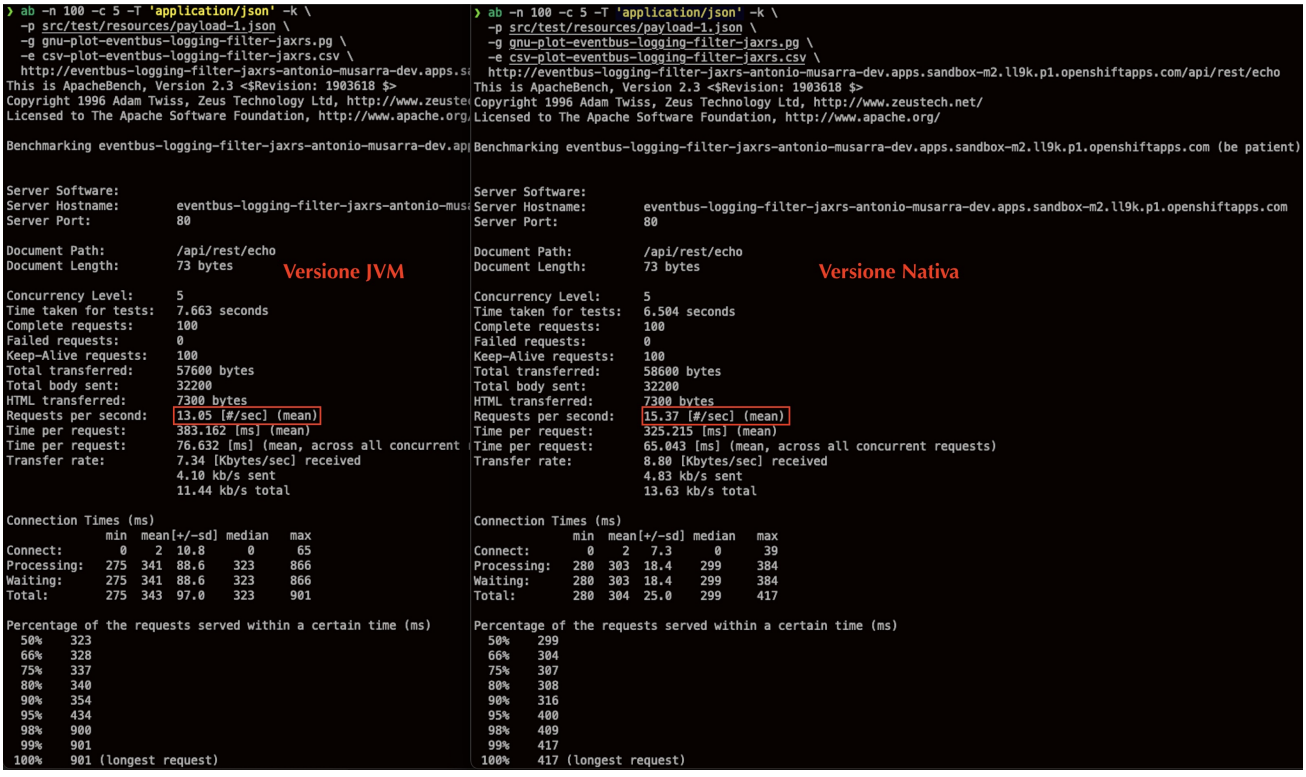


Figura 16 - *Esecuzione del comando ab per testare le prestazioni dell'applicazione Quarkus in modalità nativa*

11. Risorse

Vi lascio con alcune risorse (Web e Libri) che potrebbero esservi utili per approfondire l'argomento trattato in questo articolo.

11.1. Web

- Quarkus
- Quarkus - Event Bus
- Quarkus - OpenShift
- Getting started with MongoDB and Quarkus: Beyond the basics
- Red Hat build of Quarkus: Kubernetes-native Java
- Reactive VS Imperative - Primo Episodio

11.2. Libri

- Quarkus In Action: Building Kubernetes-Native Java Applications by Alex Soto Bueno, Jason Porter, and John Clingan (Manning Publications, 2023)
- Kubernetes Native Microservices with Quarkus and MicroProfile
- Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design

12. Conclusioni

L'Event Bus di Quarkus si presenta come una potente e flessibile risorsa per la gestione degli eventi all'interno delle moderne applicazioni Java. Grazie alla sua integrazione nativa con il framework Quarkus e l'utilizzo del motore di eventi di Vert.x, l'Event Bus offre una serie di utilizzi e vantaggi significativi per gli sviluppatori.

Nel corso dell'articolo abbiamo descritto i campi di applicazione dell'Event Bus e abbiamo affrontato in modo concreto la progettazione e implementazione di un sistema per il tracciamento delle richieste e delle risposte HTTP in un'applicazione Quarkus utilizzando l'Event Bus, riscontrando come sia stato difatti abbastanza semplice creare questo sistema dove ogni singolo componente è responsabile di un'azione specifica e comunica con gli altri componenti attraverso la pubblicazione e sottoscrizione di messaggi.

A dimostrazione del fatto che Quarkus sia nato per essere cloud native, abbiamo visto come sia stato semplice effettuare il deploy dell'applicazione Quarkus su OpenShift, un ambiente cloud native, e come sia stato possibile configurare l'applicazione Quarkus per connettersi alle risorse esterne come il broker AMQP Apache ActiveMQ Artemis e il database NoSQL MongoDB.

13. Evoluzioni nella versione 1.4.x

Le versioni **1.4.0** e **1.4.1** hanno introdotto importanti miglioramenti architetturali e qualitativi che rendono il sistema più robusto e adatto a scenari ad alta concorrenza:

- **Pattern capture-only**: il filtro JAX-RS delega tutta la logica di serializzazione e pubblicazione al `TraceEventDispatcher`, riducendo al minimo l'impatto sul ciclo di vita HTTP.
- **Thread daemon dedicato**: elimina la starvation dello scheduler `@Scheduled` sotto carico elevato.
- **Code bounded con backpressure**: `ArrayBlockingQueue` con capacità 5 000 previene il rischio OOM e gli errori `SRMSG00034` di SmallRye Reactive Messaging.
- **Annotazioni RESTEasy Reactive**: `@ServerRequestFilter` / `@ServerResponseFilter` semplificano la registrazione del filtro e abilitano l'iniezione diretta di `RoutingContext`.
- **Endpoint reattivo**: `POST /api/rest/echo/reactive` restituisce `Uni<Response>` per il percorso non-bloccante.
- **Copertura test ≥ 95 %**: `TraceEventDispatcher` al 97,6% e `TraceJaxRsRequestResponseFilter` al 95,0% (JaCoCo line coverage).

Questi interventi si sono tradotti in miglioramenti misurabili: +10–15% di throughput, −11–14% di latenza media e −17–22% di latenza al percentile 95, con zero errori, rispetto alla versione 1.3.0.

L'Event Bus di Quarkus: Comunicazione asincrona per Applicazioni Cloud Native

L'Event Bus di Quarkus offre una soluzione potente e scalabile per la comunicazione asincrona all'interno delle moderne applicazioni cloud native. Basato su Eclipse Vert.x, questo sistema di messaggistica consente una comunicazione efficiente e decentralizzata tra i diversi componenti dell'applicazione.

Vantaggi e Utilizzi:
- Integrazione di servizi in un'architettura a microservizi
- Notifiche in tempo reale e aggiornamenti all'interno dell'applicazione
- Elaborazione di flussi di dati in tempo reale
- Implementazione di modelli reattivi per migliorare la scalabilità

Sfide e Considerazioni:
- Complessità aggiuntiva nell'introduzione dell'Event Bus
- Overhead di gestione e possibili problemi di prestazioni
- Necessità di gestire attentamente la coerenza dei dati
- Complessità di debug in ambienti distribuiti
- Possibili ritardi nella consegna degli eventi e dipendenza dall'infrastruttura sottostante

Come funziona:
- Componenti dell'Event Bus: Produttori, Consumatori e Bus degli Eventi
- Funzionamento dell'Event Bus attraverso la pubblicazione, la sottoscrizione, la distribuzione e la gestione degli eventi
- Affidabilità e scalabilità nell'invio e nella gestione degli eventi

Approfondimenti e Implementazioni:
- Strategie per mitigare le sfide associate all'utilizzo dell'Event Bus
- Esempi pratici di integrazione dell'Event Bus nelle architetture cloud native
- Considerazioni per la progettazione e l'implementazione di sistemi basati sull'Event Bus

Scopri come l'Event Bus di Quarkus può migliorare la tua architettura applicativa e ottimizzare le prestazioni delle tue applicazioni cloud native.

www.linkedin.com/in/amusarra

github.com/amusarra