POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria informatica

Tesi di Laurea Magistrale

Sviluppo di Microservizi Java su AWS EKS: Confronto tra Spring Boot e Quarkus

Relatore

Prof. Fulvio RISSO

Candidato

Samuele GIANGRECO

Tutor Aziendale

Massimo PACILEO

Aprile 2024

Sommario

Nell'odierno panorama di sviluppo software, la progettazione di applicazioni distribuite come microservizi rappresenta un paradigma sempre più diffuso. In tale contesto, la scelta del framework di sviluppo assume un ruolo di primaria importanza, in quanto influenza direttamente le prestazioni, l'utilizzo delle risorse e la facilità di manutenzione del software. La presente tesi si propone di analizzare e confrontare due framework di sviluppo Java utilizzati per la creazione di microservizi: Spring Boot e Quarkus. L'obiettivo principale è quello di fornire una valutazione approfondita delle loro caratteristiche e prestazioni, al fine di supportare gli sviluppatori nella scelta del framework più adatto alle loro esigenze specifiche. Per raggiungere l'obiettivo prefissato, sono stati sviluppati tre microservizi equivalenti, implementandoli separatamente con Spring Boot e Quarkus. I microservizi sono stati progettati per sfruttare le funzionalità specifiche di ciascun framework e sono stati successivamente deployati su un cluster AWS EKS. Per valutare le loro prestazioni in un contesto realistico, sono stati condotti test di carico rigorosi, simulando differenti scenari di utilizzo. L'analisi dei dati raccolti ha evidenziato interessanti differenze tra i due framework. In termini di consumo di memoria, Quarkus si è dimostrato più efficiente, con un risparmio di memoria fino al 36% rispetto a Spring Boot. Mentre, per quanto concerne le prestazioni in termini di tempi di risposta e numero di richieste al secondo, in alcuni test queste sono molto vicine mentre in altri Spring Boot ha nettamente surclassato Quarkus, con un vantaggio che ha raggiunto il 290% nell'architettura specifica utilizzata. Tuttavia, è importante sottolineare che questi risultati dipendono anche dalle tecniche utilizzate, incluse librerie, configurazioni scelte e di default. Alla luce dei risultati ottenuti, la scelta del framework più adatto dipende dalle priorità del progetto in esame. Se l'obiettivo primario è l'ottimizzazione del consumo di memoria, Quarkus rappresenta una valida alternativa. Tuttavia, se le performance in termini di tempi di risposta e throughput sono ritenute prioritarie, Spring Boot si configura come la scelta migliore. Oltre alle prestazioni e al consumo di risorse, è importante considerare anche la facilità d'uso. Entrambi i framework offrono un buon livello di usabilità, con Spring Boot che vanta una curva di apprendimento più graduale grazie alla sua ampia diffusione e alla ricca documentazione disponibile.

Ringraziamenti

Indice

Elenco delle tabelle VIII

Elenco delle figure IX

1 Introduzione 1

2 Stato dell'arte 3
2.1 Evoluzione delle Architetture Software . . . 3
2.1.1 Architetture Monolitiche . . . 4
2.1.2 Microservizi . . . 4
2.1.3 Cloud . . . 6
2.2 Ambiente di sviluppo . . . 8
2.2.1 Java . . . 8
2.2.2 Java Virtual Machine (JVM) . . . 9
2.2.3 GraalVM . . . 10
2.3 Framework . . . 11
2.3.1 Spring Boot . . . 13
2.3.2 Quarkus . . . 14
2.4 Docker . . . 14
2.5 Kubernetes . . . 16
2.6 AWS . . . 19

3 Design e implementazione 27
3.1 Obiettivo della tesi . . . 27
3.2 Sviluppo . . . 28
3.2.1 Architettura . . . 28
3.2.2 Implementazione con Spring Boot . . . 33
3.2.3 Implementazione con Quarkus . . . 41
3.3 Deployment . . . 47
3.3.1 Immagine Docker . . . 47
3.3.2 Cluster EKS . . . 50

3.3.3 Networking 54

4 Test e risultati 60

4.1 test di carico 60

4.2 Monitoraggio 64

4.3 Confronto Sviluppo 72

5 Sviluppi Futuri 75

6 Conclusione 77

Bibliografia 79

Elenco delle tabelle

4.1 Misurazioni dei tempi di avvio su un campione di 10 avvii 69
4.2 Dimensione dell'immagine 69

Elenco delle figure

2.1 Componenti Kubernetes 19
2.2 Architettura di Amazon Ec2 22

3.1 Architettura 29
3.2 Impostazioni di auto-scaling 52

4.1 Setup test di carico 62
4.2 Test di carico su Spring Boot 63
4.3 Test di carico su Quarkus 64
4.4 Architettura test di carico con locust 67
4.5 Consumo di cpu su spring boot con 50 utenti 70
4.6 Consumo di cpu su quarkus con 50 utenti 71
4.7 Consumo di memoria su spring boot con 50 utenti 71
4.8 Consumo di memoria su quarkus con 50 utenti 72

Capitolo 1

Introduzione

La mia tesi, condotta presso l'azienda Blue Reply con la supervisione del tutor aziendale Massimo Pacileo, si colloca in un momento cruciale per il settore dell'informatica aziendale, caratterizzato da una crescente spinta verso l'adozione e l'ottimizzazione delle soluzioni basate sul cloud computing. Blue Reply, in linea con questa tendenza, ha manifestato l'interesse e la volontà di esplorare le opportunità offerte dal cloud per migliorare l'efficienza operativa, la flessibilità e la scalabilità delle proprie applicazioni.

L'idea di approfondire la valutazione dei framework di sviluppo Java nasce dall'obiettivo strategico di Blue Reply di rimanere all'avanguardia nel panorama tecnologico, anticipando le esigenze del mercato e adottando soluzioni innovative. In particolare, l'azienda si è posta la seguente domanda: data l'importanza crescente del cloud, sia per lo sviluppo di nuove applicazioni che per la migrazione di quelle esistenti, è ancora opportuno affidarsi esclusivamente a Spring Boot, il framework di sviluppo Java ampiamente consolidato e preferito in azienda, oppure esistono alternative più adatte al contesto del cloud, come Quarkus?

Per rispondere a questa domanda e guidare le future scelte di sviluppo e migrazione delle applicazioni, la mia ricerca si è concentrata sull'analisi comparativa delle prestazioni di Spring Boot e Quarkus. L'obiettivo è stato quello di valutare non solo le differenze in termini di performance e scalabilità tra i due framework, ma anche di comprendere come ciascuno di essi si integri con le specifiche esigenze e le dinamiche proprie dell'ambiente cloud.

Al fine di ottenere risultati significativi e applicabili, è stata sviluppata un'applicazione di esempio rappresentativa delle tipiche funzionalità richieste in un contesto aziendale, focalizzandosi su una richiesta di approvvigionamento. Questa applicazione è stata utilizzata come banco di prova per eseguire una serie di test

accurati e confrontare le performance di Spring Boot e Quarkus in condizioni simili. In aggiunta alla valutazione delle performance, la mia ricerca si propone di esplorare l'impegno richiesto per lo sviluppo e la gestione delle applicazioni nei due diversi framework. Comprendere l'effort necessario per sviluppare e mantenere un'applicazione è cruciale per la pianificazione e la gestione delle risorse aziendali.

Attraverso questa ricerca, non solo si mira a fornire una panoramica approfondita delle capacità e delle limitazioni dei due framework nel contesto del cloud computing, ma anche a fornire raccomandazioni pratiche per ottimizzare lo sviluppo e la migrazione delle applicazioni all'interno di Blue Reply e, più in generale, per supportare le decisioni strategiche nel campo dell'IT aziendale.

Capitolo 2

Stato dell'arte

In questo Capitolo, esploreremo i concetti fondamentali di programmazione in Java, fornendo una solida base per comprendere lo sviluppo di applicazioni enterprise. Successivamente, ci addentreremo nel framework Spring Boot, che offre un approccio conveniente e modulare per la creazione di servizi web e microservizi. Quarkus, invece, verrà esaminato come una nuova generazione di framework Java, ottimizzato per il cloud e le architetture serverless, con un'attenzione particolare alle prestazioni e alla riduzione del consumo di risorse.

Parallelamente, approfondiremo i concetti di containerizzazione mediante Docker, che rivoluziona il modo in cui le applicazioni vengono distribuite e gestite, fornendo un ambiente isolato e portatile per eseguire software in modo consistente su qualsiasi infrastruttura. Successivamente, esploreremo Kubernetes, un orchestratore di container che automatizza il deployment, la scalabilità e la gestione delle applicazioni containerizzate in ambienti distribuiti.

Infine, esamineremo l'integrazione di queste tecnologie nell'ambiente di cloud computing offerto da Amazon Web Services (AWS), uno dei principali fornitori di servizi cloud al mondo. Attraverso questa panoramica teorica, i lettori acquisiranno una solida comprensione dei concetti e delle tecnologie chiave che costituiscono il tessuto connettivo della moderna architettura delle applicazioni, preparandoli così per un'analisi più approfondita delle implementazioni pratiche all'interno della tesi.

2.1 Evoluzione delle Architetture Software

Nell'era digitale in continua evoluzione, la progettazione e lo sviluppo di software sono soggetti a un costante rinnovamento. Tra i vari fattori che guidano questa evoluzione, la struttura architetturale dei sistemi software riveste un ruolo cruciale.

In questa sezione, esploreremo l'evoluzione delle architetture software, dall'approccio tradizionale monolitico alle moderne infrastrutture basate su microservizi, con un'attenzione particolare all'integrazione nel paradigma del cloud computing.

2.1.1 Architetture Monolitiche

Le architetture monolitiche rappresentano un modello tradizionale di progettazione software in cui un'applicazione è sviluppata come un'unica unità monolitica. In questo approccio, tutti i componenti dell'applicazione, inclusi l'interfaccia utente, la logica di business e l'accesso ai dati, sono solitamente implementati e distribuiti come un'unica entità. Questo tipo di architettura è caratterizzato dalla sua semplicità e dalla sua facilità di sviluppo iniziale, in quanto tutte le funzionalità sono consolidate in un unico codice base.

Tuttavia, le architetture monolitiche possono diventare complesse e difficili da mantenere con l'aumentare delle dimensioni e della complessità dell'applicazione. Inoltre, la scalabilità può essere limitata, poiché è necessario scalare l'intera applicazione anziché singoli componenti. Nonostante questi svantaggi, le architetture monolitiche continuano ad essere utilizzate in vari contesti, specialmente in applicazioni di piccole e medie dimensioni o quando i requisiti di scalabilità e manutenibilità sono meno critici.

2.1.2 Microservizi

I microservizi si configurano come un paradigma architetturale software modulare, in cui un'applicazione viene scomposta in una serie di servizi granulari, autonomi e autosufficienti. Questi servizi collaborano tra loro attraverso API ben definite per fornire un'unica funzionalità completa, sono caratterizzati da:

- **Autonomia e indipendenza**: Ogni microservizio è un'entità autonoma, con un ciclo di vita indipendente, che può essere sviluppata, testata, distribuita e scalata in modo indipendente dagli altri servizi. Questa autonomia favorisce la modularità e l'isolamento dei componenti, rendendo lo sviluppo e la manutenzione del software più flessibili e agili.
- **Coesione e responsabilità singola**: Inoltre, ciascun microservizio è focalizzato su una singola funzionalità ben definita e ha la responsabilità di gestirla completamente. Questa coesione interna facilita la comprensione del codice, la sua manutenzione e la sua evoluzione, riducendo l'accoppiamento tra i servizi e rendendo l'architettura più robusta e flessibile.
- **Scalabilità e granularità**: I microservizi possono essere scalati indipendentemente in base alle loro esigenze specifiche, consentendo una granularità più

precisa rispetto a un'architettura monolitica. È possibile scalare orizzontalmente aggiungendo nuove istanze di un servizio o verticalmente aumentando le risorse assegnate a un'istanza. Questa granularità permette di ottimizzare l'utilizzo delle risorse e di scalare l'applicazione in modo efficiente in base al carico di lavoro.

- **Comunicazione tramite API**: I microservizi comunicano tra loro attraverso API leggere e ben definite, promuovendo un'architettura "loosely coupled". Le API espongono le funzionalità di un servizio in modo che altri servizi possano utilizzarle, rendendo l'architettura più flessibile e modulare, agevolando così l'evoluzione e la manutenzione dei servizi.

Inoltre dal punto di vista della scalabilità e della resilienza, i microservizi offrono vantaggi significativi. La possibilità di scalare individualmente i servizi in base al loro carico di lavoro ottimizza l'utilizzo delle risorse e garantisce prestazioni efficienti. In caso di guasto, l'isolamento dell'errore al singolo servizio interessato impedisce il malfunzionamento dell'intera applicazione, garantendo una maggiore resilienza. La scalabilità orizzontale dei microservizi facilita inoltre la gestione di picchi di carico improvvisi, assicurando la continuità del servizio e la soddisfazione degli utenti.

Per quanto riguarda manutenzione e il deployment, i microservizi offrono una maggiore facilità d'azione. Essendo indipendenti, possono essere manutenuti e aggiornati singolarmente senza la necessità di ridistribuire l'intera applicazione. La modularità agevola l'individuazione e la correzione dei bug, riducendo il tempo di risoluzione dei problemi. Inoltre, il processo di deployment risulta semplificato e rapido, in quanto si limita al microservizio interessato anziché coinvolgere l'intera applicazione.

Altri benefici includono una migliore osservabilità e monitoraggio dell'applicazione, che si traduce in una visibilità più dettagliata e precisa del comportamento dei singoli servizi. Inoltre, l'adozione di un'architettura a microservizi favorisce una maggiore autonomia e controllo da parte dei team di sviluppo, contribuendo a un senso di responsabilità.

L'adozione di un'architettura a microservizi, pur vantando notevoli vantaggi, presenta anche alcuni svantaggi significativi da considerare con attenzione. La gestione di un ecosistema di microservizi può risultare più complessa rispetto a un'architettura monolitica, richiedendo strumenti e tecnologie specifiche per il ciclo di vita dei servizi, come distribuzione, monitoraggio e configurazione. La comunicazione efficiente e affidabile tra i microservizi è fondamentale per il corretto funzionamento dell'applicazione, e la sua complessità aumenta con il numero di

servizi

La distribuzione dei dati su diversi servizi nell'architettura a microservizi rende la loro gestione più complessa. La coerenza e l'integrità dei dati devono essere garantite attraverso meccanismi di sincronizzazione e transazioni distribuite, con un'attenzione maggiore alla sicurezza data la distribuzione su più sistemi.

L'architettura a microservizi può introdurre nuovi errori e bug difficili da individuare e risolvere. La gestione delle eccezioni e dei guasti diventa più complessa in un'architettura distribuita, rendendo il monitoraggio e l'osservabilità cruciali per identificare e risolvere rapidamente i problemi.

La comunicazione tra i microservizi può generare un sovraccarico di rete e di risorse computazionali. La gestione di un elevato numero di microservizi può richiedere una maggiore infrastruttura e risorse, rendendo l'ottimizzazione delle prestazioni un aspetto critico per l'efficienza dell'architettura.

Lo sviluppo e la manutenzione di un'architettura a microservizi possono richiedere costi iniziali superiori rispetto a un'architettura monolitica. La complessità di gestione e la necessità di competenze specifiche possono aumentare i costi di sviluppo e manutenzione nel lungo termine, con un investimento significativo in strumenti e tecnologie per il monitoraggio e l'osservabilità.

2.1.3 Cloud

Il cloud computing rappresenta una delle tecnologie più rivoluzionarie degli ultimi anni, con un impatto significativo su come le organizzazioni concepiscono, gestiscono e distribuiscono le risorse informatiche. A differenza del modello tradizionale di possedere e gestire fisicamente server e infrastrutture, il cloud computing offre un approccio basato su risorse virtuali accessibili attraverso Internet.

In termini semplici, il cloud computing consente agli utenti di accedere a risorse informatiche come server, archiviazione, database, software e altre risorse attraverso provider di servizi cloud, anziché ospitarle e gestirle internamente. Questo modello offre numerosi vantaggi, tra cui la scalabilità, l'accessibilità da qualsiasi luogo e dispositivo, la flessibilità e la riduzione dei costi operativi.

Per comprendere appieno il cloud computing, è essenziale comprendere i tre principali modelli di servizio offerti dai fornitori di servizi cloud:

1. **Infrastructure as a Service (IaaS)**: In questo modello, i fornitori di servizi cloud mettono a disposizione infrastrutture IT virtuali, come server, storage e

reti, consentendo agli utenti di utilizzare queste risorse per ospitare le proprie applicazioni e carichi di lavoro senza dover gestire fisicamente l'hardware sottostante.

2. **Platform as a Service (PaaS)**: Nel modello PaaS, i fornitori offrono piattaforme di sviluppo e di esecuzione per gli sviluppatori, che possono quindi creare, testare e distribuire le proprie applicazioni senza dover gestire l'infrastruttura sottostante. Questo approccio accelera il ciclo di sviluppo delle applicazioni e semplifica la gestione delle risorse.

3. **Software as a Service (SaaS)**: Con il modello SaaS, i fornitori di servizi cloud offrono applicazioni software completamente gestite e accessibili tramite Internet. Gli utenti possono accedere a queste applicazioni su abbonamento e utilizzarle senza dover installare o gestire alcun software localmente.

L'adozione del cloud computing offre una serie di benefici significativi per le organizzazioni di ogni dimensione e settore:

- **Scalabilità**: Il cloud computing consente alle organizzazioni di scalare rapidamente le risorse in base alle esigenze, consentendo una maggiore flessibilità e reattività alle variazioni della domanda.

- **Riduzione dei costi**: Eliminando la necessità di investire in infrastrutture fisiche e di gestire complessi data center, il cloud computing può ridurre i costi operativi e di capitale.

- **Accessibilità e mobilità**: Grazie alla disponibilità da qualsiasi luogo e dispositivo connesso a Internet, il cloud computing consente ai dipendenti di collaborare in modo più efficace e di accedere alle risorse aziendali in modo più flessibile.

- **Agilità aziendale**: Con la capacità di distribuire rapidamente nuove risorse e applicazioni, il cloud computing consente alle organizzazioni di adattarsi più rapidamente ai cambiamenti del mercato e di innovare più velocemente.

- **Affidabilità e resilienza**: I fornitori di servizi cloud offrono spesso servizi altamente affidabili e resilienti, con garantie di uptime e misure di sicurezza avanzate.

In sintesi, il cloud computing rappresenta una componente fondamentale della trasformazione digitale delle organizzazioni, consentendo loro di rimanere competitive in un ambiente aziendale sempre più digitale e dinamico.

2.2 Ambiente di sviluppo

Lo sviluppo software odierno richiede la creazione di applicazioni affidabili, efficienti e scalabili. In questo contesto, la scelta del linguaggio di programmazione e della piattaforma di sviluppo assume un ruolo fondamentale.

Java si è affermato come uno dei linguaggi di programmazione più utilizzati al mondo, grazie alla sua portabilità, sicurezza e flessibilità. La Java Virtual Machine (JVM) fornisce un ambiente di runtime indipendente dalla piattaforma per l'esecuzione di bytecode Java.

Tuttavia, le esigenze di prestazioni e la complessità delle applicazioni moderne richiedono spesso soluzioni più avanzate. GraalVM è una piattaforma di sviluppo software basata sulla JVM che offre diverse funzionalità avanzate, come la compilazione AOT, il supporto per linguaggi dinamici e una JVM a basso profilo (SubstrateVM).

2.2.1 Java

Java è stato creato da un team di ingegneri Sun Microsystems, guidato da James Gosling, tra il 1991 e il 1995. Il progetto, inizialmente denominato "Oak", era volto alla creazione di un linguaggio di programmazione per dispositivi elettronici di consumo, come televisori e decoder digitali. Tuttavia, le sue caratteristiche di portabilità, sicurezza e flessibilità lo hanno reso adatto a un'ampia gamma di applicazioni, determinandone il rapido successo. Ha come caratteristiche principali:

- **Programmazione a oggetti**: Java promuove un approccio di sviluppo modulare e riutilizzabile basato su classi e oggetti. Questa caratteristica facilita la progettazione di software complesso e la manutenzione del codice, favorendo la coesione e la leggibilità.
- **Portabilità**: Il bytecode generato dal compilatore Java è eseguibile su qualsiasi piattaforma che abbia installato la Java Virtual Machine (JVM), rendendolo indipendente dal sistema operativo e dall'architettura hardware. Questa caratteristica ha contribuito alla diffusione di Java in diversi contesti, favorendo lo sviluppo di applicazioni cross-platform.
- **Sicurezza**: Java è stato progettato con la sicurezza in mente, includendo funzionalità come il garbage collection e la verifica del bytecode per ridurre i rischi di vulnerabilità e crash. La JVM fornisce un ambiente sandbox che limita l'accesso alle risorse di sistema e isola le applicazioni l'una dall'altra, aumentando la sicurezza e la stabilità.

- **Ampia libreria standard**: Java offre una vasta libreria standard di API per la gestione di comuni attività di sviluppo, come la gestione di file, la rete e l'interfaccia utente. La libreria standard include diverse funzionalità già pronte, evitando agli sviluppatori di doverle implementare autonomamente e velocizzando il processo di sviluppo.
- **Comunity attiva**: Java vanta una grande e attiva comunity di sviluppatori che fornisce supporto, risorse e librerie open-source per diverse esigenze di sviluppo. La comunity contribuisce alla crescita e all'evoluzione del linguaggio, offrendo soluzioni innovative e un ecosistema ricco di strumenti e librerie.

2.2.2 Java Virtual Machine (JVM)

La Java Virtual Machine (JVM) è un componente fondamentale dell'ecosistema Java. È una macchina virtuale che funge da ambiente di runtime per l'esecuzione di bytecode Java. La JVM è responsabile di diverse funzioni, tra cui:

- **Caricamento e conversione del bytecode**: La JVM carica il bytecode compilato dai file .class in memoria e lo converte in codice macchina specifico per la CPU del sistema in uso, ottimizzando l'esecuzione del codice per la piattaforma target. La JVM può inoltre applicare tecniche di ottimizzazione avanzate per migliorare ulteriormente le prestazioni.
- **Gestione della memoria**: La JVM gestisce la memoria allocata per le applicazioni Java in modo efficiente e sicuro. Il garbage collector automatico libera la memoria non più utilizzata, evitando perdite di memoria e semplificando la gestione del ciclo di vita degli oggetti. La JVM offre diverse configurazioni per la gestione della memoria, permettendo di ottimizzare l'utilizzo delle risorse in base alle esigenze specifiche.
- **Interazione con il sistema operativo**: La JVM mette a disposizione un set di API standard per l'interazione con il sistema operativo sottostante, includendo l'accesso a file, rete, input/output e altre funzionalità native del sistema. Le API della JVM garantiscono un'interfaccia uniforme per l'accesso alle risorse del sistema, indipendentemente dalla piattaforma in uso.
- **Sicurezza**: La JVM è progettata per fornire un ambiente di runtime sicuro per l'esecuzione di applicazioni Java. La verifica del bytecode e il sandboxing sono due meccanismi chiave che garantiscono la sicurezza e l'integrità del codice. La JVM offre inoltre diverse funzionalità di sicurezza avanzate per proteggere le applicazioni da vulnerabilità e intrusioni.
- **Prestazioni**: La JVM è stata ottimizzata per fornire buone prestazioni per l'esecuzione di applicazioni Java. La compilazione JIT, il garbage collection

efficiente e l'ottimizzazione del codice contribuiscono a migliorare le prestazioni complessive. La JVM offre diverse opzioni per la configurazione delle prestazioni, permettendo di ottimizzare l'esecuzione in base alle esigenze specifiche.

La JVM è disponibile per diverse piattaforme, tra cui Windows, Linux e macOS. Questa portabilità garantisce che le applicazioni Java (.jar) possano essere eseguite su qualsiasi piattaforma con una JVM installata, senza la necessità di ricompilare il codice.

2.2.3 GraalVM

[1]GraalVM è una piattaforma di runtime avanzata che offre un ambiente di sviluppo e di esecuzione versatile per diverse applicazioni. Si basa su HotSpot/OpenJDK, offrendo un JDK performante con funzionalità avanzate come la compilazione ahead-of-time (AOT) con Native Image.[2]

Ha come caratteristiche principali:

- **GraalVM JIT Compiler**: Il suo compilatore just-in-time (JIT) si distingue per la sua capacità di ottimizzare il bytecode Java, generando codice macchina nativo specifico per l'architettura del processore. Rispetto al tradizionale JIT di HotSpot, questo approccio offre un'incredibile accelerazione per le applicazioni Java. Utilizzando tecniche di analisi avanzate e il profiling continuo, il compilatore identifica dinamicamente le aree di codice che richiedono ottimizzazione, adattandosi al comportamento dell'applicazione. Inoltre, integra funzionalità di compilazione selettiva, consentendo di ottimizzare solo le parti specifiche del codice che influenzano maggiormente le prestazioni complessive.
- **Immagine nativa**: È una delle innovazioni più significative di GraalVM, consentendo la compilazione preventiva delle applicazioni Java in binari autonomi eseguibili. Questo elimina la dipendenza da una macchina virtuale Java (JVM) durante l'esecuzione, garantendo un avvio istantaneo e un'esecuzione senza overhead. Le prestazioni delle applicazioni compilate con immagine nativa sono elevate e allo stesso tempo, riducono significativamente l'impronta di memoria, rendendo queste applicazioni ideali per ambienti con risorse limitate. Inoltre, questo approccio offre un elevato livello di sicurezza, grazie all'isolamento del codice all'interno dell'eseguibile e alla mitigazione delle vulnerabilità legate alla JVM.
- **Supporto Multi-Linguaggio**: Oltre alle prestazioni elevate e alla sicurezza, GraalVM si distingue per supportare numerosi linguaggi di programmazione, tra cui JavaScript, Python, R, Ruby e altri ancora. Questo supporto consente

la creazione di applicazioni polyglot, che integrano senza soluzione di continuità diversi linguaggi all'interno dello stesso progetto. Grazie ai runtime specifici per ogni linguaggio ottimizzati per sfruttare al meglio le caratteristiche native, GraalVM facilita la collaborazione tra team di sviluppo che utilizzano linguaggi differenti, promuovendo la creazione di software completo e flessibile.

- **Truffle**: Una piattaforma di implementazione di linguaggi integrata in GraalVM, semplifica e accelera la creazione di runtime efficienti per nuovi linguaggi. Basata su principi di modularità e componibilità, Truffle consente la costruzione di runtime su misura per le specifiche esigenze di ogni linguaggio, fornendo un set di librerie e strumenti comuni che riducono il tempo e lo sforzo necessario per lo sviluppo. Questo approccio promuove l'innovazione nella creazione di linguaggi performanti, aprendo nuove possibilità per lo sviluppo software.
- **SDK GraalVM**: offre un set completo di strumenti per la creazione, la gestione e l'analisi delle applicazioni GraalVM. Tra questi strumenti, un debugger avanzato consente di ispezionare il codice in esecuzione e individuare eventuali errori, mentre un profiler completo analizza le prestazioni dell'applicazione e identifica i colli di bottiglia. Inoltre, un'interfaccia a riga di comando automatizza varie operazioni e facilita la gestione delle applicazioni GraalVM, contribuendo a migliorare l'efficienza e la produttività dello sviluppo.

2.3 Framework

Nell'ambito dello sviluppo software, i framework rappresentano un insieme strutturato di librerie, strumenti e linee guida che facilitano lo sviluppo di applicazioni. In particolare, nel contesto della programmazione in Java, i framework giocano un ruolo fondamentale nell'accelerare il processo di sviluppo, migliorare la manutenibilità del codice e favorire l'adozione di pratiche di programmazione standardizzate.

L'adozione di un framework offre numerosi vantaggi, che spesso si basano su un insieme comune di tecniche:

- **Inversion of Control (IoC) / Dependency Injection (DI)**: La IoC e la DI sono tecniche fondamentali utilizzate in molti framework Java. Consentono di ridurre le dipendenze tra componenti di un'applicazione, migliorando la modularità e la testabilità del codice. Inoltre, semplificano la gestione delle risorse e promuovono una separazione chiara tra la configurazione dell'applicazione e la logica di business.
- **Aspect-Oriented Programming (AOP)**: L'AOP è una tecnica utilizzata per separare le preoccupazioni trasversali, come il logging, la sicurezza e la

gestione delle transazioni, dalla logica di business principale dell'applicazione. I framework che supportano l'AOP consentono agli sviluppatori di definire e applicare facilmente aspetti trasversali in tutta l'applicazione, migliorando la modularità e la manutenibilità del codice.

- **Convention over Configuration**: Molti framework Java adottano il principio della "convenzione anziché configurazione", che prevede che le decisioni predefinite siano assunte dal framework piuttosto che dagli sviluppatori, riducendo la quantità di codice boilerplate necessario per configurare e utilizzare il framework. Questa pratica accelera lo sviluppo e favorisce la coerenza all'interno dell'applicazione.

- **Testing e Automatizzazione**: I framework spesso forniscono strumenti integrati per la scrittura e l'esecuzione di test automatizzati, semplificando il processo di verifica del corretto funzionamento dell'applicazione. Questi strumenti contribuiscono a garantire la qualità del software e consentono agli sviluppatori di individuare e correggere errori più rapidamente.

- **Sicurezza**: I framework Java spesso includono funzionalità di sicurezza integrate, come gestione delle sessioni, autenticazione degli utenti e prevenzione di attacchi comuni come SQL injection e cross-site scripting (XSS). Queste funzionalità aiutano gli sviluppatori a sviluppare applicazioni robuste e sicure.

- **Internazionalizzazione e Localizzazione**: Molti framework offrono supporto per l'internazionalizzazione e la localizzazione, consentendo agli sviluppatori di creare applicazioni che possono essere facilmente adattate a diverse lingue e culture senza modifiche significative al codice sorgente.

- **Documentazione e Supporto della Community**: I framework Java ben consolidati di solito offrono una vasta documentazione e un supporto attivo dalla community. Questo include guide, tutorial, forum di discussione e altro ancora, che aiutano gli sviluppatori a comprendere meglio il framework e a risolvere eventuali problemi che possono incontrare durante lo sviluppo dell'applicazione.

- **Interoperabilità**: Java è noto per la sua interoperabilità, che consente agli sviluppatori di integrare facilmente le loro applicazioni con altri sistemi e tecnologie. I framework Java spesso facilitano questa interoperabilità fornendo API standard e supporto per protocolli di comunicazione comuni.

- **Aggiornamenti e Manutenzione**: I framework Java ben progettati sono di solito soggetti a frequenti aggiornamenti e miglioramenti. Questi aggiornamenti possono includere nuove funzionalità, correzioni di bug, ottimizzazioni delle prestazioni e aggiornamenti di sicurezza, che consentono agli sviluppatori di

mantenere le proprie applicazioni al passo con gli standard attuali e di adottare le migliori pratiche di sviluppo.

2.3.1 Spring Boot

[3]Spring Boot è un framework open source che facilita e velocizza lo sviluppo di applicazioni Java, in particolare quelle basate su microservizi. Si basa sulla popolare libreria Spring Framework, offrendo un'ampia gamma di funzionalità preconfigurate che riducono il tempo e lo sforzo necessari per la creazione di applicazioni Java robuste e scalabili.

Le caratteristiche principali di Spring Boot includono:

- **Configurazione automatica**: Spring Boot elimina la necessità di configurare manualmente molti aspetti comuni delle applicazioni Java, come il server web, il contesto e la gestione delle proprietà. Spring Boot è in grado di auto-configurare l'applicazione in base alle dipendenze presenti nel classpath e alle proprietà disponibili nell'ambiente.
- **Dipendenze automatiche**: Spring Boot gestisce automaticamente le dipendenze del progetto, semplificando la configurazione e la build dell'applicazione. Spring Boot utilizza un sistema di "starter" per includere automaticamente le dipendenze necessarie per specifiche funzionalità, come la gestione del database, la sicurezza o la messaggistica.
- **Avvio rapido**: Le applicazioni Spring Boot possono essere avviate rapidamente e facilmente, senza la necessità di un complesso processo di configurazione. Non è necessario configurare un server web o un contesto manualmente.
- **Indicatori di integrità e metriche**: Spring Boot fornisce indicatori di integrità e metriche predefiniti che consentono di monitorare lo stato dell'applicazione in tempo reale.
- **Supporto per i microservizi**: Spring Boot è ideale per lo sviluppo di applicazioni a microservizi, grazie al suo supporto integrato per Spring Cloud. Spring Cloud fornisce una serie di strumenti per la gestione e la scoperta dei microservizi, la configurazione distribuita, la sicurezza e il monitoraggio.
- **Sicurezza**: Spring Boot offre funzionalità di sicurezza integrate, come l'autenticazione e l'autorizzazione, che possono essere facilmente configurate e utilizzate.
- **Gestione delle eccezioni**: Spring Boot fornisce un sistema di gestione delle eccezioni robusto e flessibile che semplifica la gestione degli errori in modo coerente.

- **Testing**: Spring Boot facilita lo sviluppo di test unitari e di integrazione per le applicazioni. Spring Boot fornisce un set di annotazioni e di strumenti per semplificare la scrittura e l'esecuzione dei test.

- **Produzione**: Spring Boot fornisce funzionalità pronte per la produzione, come la generazione di pacchetti JAR autosufficienti e la configurazione dei server di produzione.

2.3.2 Quarkus

Nel panorama delle applicazioni Java, il movimento verso l'architettura cloud-native è diventato sempre più predominante. Con l'aumento della domanda di applicazioni altamente scalabili, resilienti e ottimizzate per i container, le organizzazioni cercano soluzioni che semplifichino lo sviluppo e la gestione di tali applicazioni. In questo contesto, Quarkus si è rapidamente affermato come un framework innovativo per lo sviluppo di applicazioni Java native cloud.

I principali vantaggi di Quarkus rispetto a Spring Boot sono[4]:

- **Minore consumo di risorse**: Quarkus è progettato per essere leggero e performante. Le applicazioni Quarkus in genere consumano meno memoria rispetto alle applicazioni Spring Boot. Questo è un vantaggio significativo per le applicazioni in esecuzione su microservizi o in ambienti cloud.

- **Avvio più rapido**: Le applicazioni Quarkus si avviano molto più velocemente rispetto alle applicazioni Spring Boot. Questo è dovuto al fatto che Quarkus utilizza un'architettura completamente nativa e non richiede la JVM (Java Virtual Machine) per l'avvio.

- **Eseguibile più leggero**: Grazie alla compilazione nativa con GraalVM. Questo consente di creare immagini di container estremamente leggere e di ridurre il tempo di avvio dell'applicazione.

- **Supporto nativo per Kubernetes**: Quarkus offre un supporto nativo per Kubernetes, la piattaforma di orchestrazione container leader. Questo significa che è possibile sviluppare, distribuire e gestire applicazioni Quarkus su Kubernetes con grande facilità.[5]

2.4 Docker

Negli ultimi anni, la virtualizzazione ha rivoluzionato il modo in cui vengono sviluppate, distribuite e gestite le applicazioni software. Le tecnologie di virtualizzazione

tradizionali, come le macchine virtuali (VMs), hanno reso possibile l'esecuzione di sistemi operativi e applicazioni all'interno di un ambiente isolato e autonomo, consentendo la consolidazione delle risorse hardware e la scalabilità delle infrastrutture IT.

Tuttavia, l'approccio basato su VMs presenta alcune limitazioni significative in termini di efficienza delle risorse, overhead di gestione e tempi di avvio. La necessità di fornire soluzioni più leggere, agili ed efficienti ha portato allo sviluppo di nuove tecnologie di virtualizzazione, tra cui Docker.

Docker è una piattaforma open-source per la creazione, la distribuzione e l'esecuzione di applicazioni in container. Il concetto chiave di Docker è il container, un'unità standardizzata di software che incorpora il codice dell'applicazione, le sue dipendenze e le configurazioni necessarie per l'esecuzione. Rispetto alle VMs, i container sono più leggeri, più rapidi da avviare e condividono il kernel del sistema operativo host, riducendo così l'overhead e ottimizzando l'utilizzo delle risorse.

I principali obiettivi di Docker includono:

- **Portabilità**: I container Docker possono essere eseguiti su qualsiasi piattaforma che supporti Docker, eliminando le differenze di ambiente tra i vari ambienti di sviluppo, test e produzione.

- **Isolamento**: Ogni container fornisce un ambiente isolato per l'esecuzione dell'applicazione, garantendo che le dipendenze e le configurazioni non interferiscano con altre applicazioni o con il sistema operativo host.

- **Agilità**: Docker semplifica il processo di sviluppo, distribuzione e aggiornamento delle applicazioni, consentendo agli sviluppatori di creare, testare e distribuire applicazioni in modo rapido e coerente.

- **Scalabilità**: I container Docker possono essere facilmente scalati orizzontalmente per gestire carichi di lavoro dinamici e crescenti, consentendo alle applicazioni di adattarsi alle esigenze del business in modo flessibile ed efficiente.

L'architettura di Docker è composta da diversi componenti chiave, tra cui:

- **Docker Engine**: Il motore Docker è il componente principale responsabile della creazione, dell'esecuzione e della gestione dei container. È composto da un daemon di sistema (dockerd) che gestisce i container e un'interfaccia a riga di comando (CLI) che consente agli utenti di interagire con il motore Docker.

- **Docker Image**: Un'immagine Docker è un pacchetto autonomo e leggero che contiene il codice dell'applicazione, le dipendenze e le configurazioni necessarie per eseguire il container. Le immagini Docker seguono un modello di layering, consentendo la creazione efficiente di immagini riducendo al minimo la duplicazione dei dati.

- **Docker Container**: Un container Docker è un'istanza in esecuzione di un'immagine Docker. I container offrono un ambiente isolato e portatile per l'esecuzione dell'applicazione, consentendo agli sviluppatori di distribuire facilmente le proprie applicazioni senza preoccuparsi delle differenze di ambiente.

- **Docker Registry**: Il registro Docker è un repository centralizzato per la memorizzazione e la distribuzione delle immagini Docker. Docker Hub è il registro pubblico ufficiale di Docker, ma è possibile configurare registri privati per gestire e condividere immagini personalizzate all'interno di un'organizzazione.

2.5 Kubernetes

Kubernetes è un sistema open-source per l'automazione della distribuzione, della scalabilità e della gestione di applicazioni containerizzate. Originariamente sviluppato da Google e successivamente donato alla Cloud Native Computing Foundation (CNCF), Kubernetes è diventato lo standard de facto per l'orchestrazione di container, offrendo un'infrastruttura robusta e flessibile per la distribuzione di applicazioni moderne basate su microservizi.

Kubernetes offre una vasta gamma di vantaggi:

- **Efficienza**: Kubernetes si distingue per la sua capacità di automatizzare la distribuzione, il ridimensionamento e la gestione dei container. Ciò libera le risorse umane da compiti ripetitivi e onerosi, consentendo un impiego più efficace del tempo. Inoltre, semplifica la gestione del ciclo di vita delle applicazioni, dalla loro creazione alla distribuzione e alla dismissione, contribuendo così a ottimizzare i processi operativi e a ridurre i costi complessivi.

- **Scalabilità**: È un'altra caratteristica di rilievo, grazie alla sua capacità di adattare automaticamente le risorse in base al carico di lavoro. Questo include sia la scalabilità orizzontale che verticale, permettendo al sistema di aumentare o diminuire il numero di nodi nel cluster in modo trasparente. Inoltre, la piattaforma è in grado di gestire cluster di notevoli dimensioni, garantendo un'elevata resilienza grazie a funzionalità come la replica automatica dei container e la gestione dei guasti.

- **Flessibilità**: La piattaforma supporta una vasta gamma di casi d'uso, dallo sviluppo e distribuzione di nuove applicazioni alla modernizzazione di quelle legacy. Inoltre, è compatibile con diverse tecnologie di containerizzazione, linguaggi di programmazione e framework di sviluppo, offrendo una grande versatilità. La facilità di integrazione con altri strumenti e tecnologie consente di creare un ambiente di sviluppo e deployment completo e personalizzato.

- **Sicurezza**: Kubernetes offre un controllo granulare sulle risorse assegnate ai container e sui permessi di accesso al cluster. Inoltre, fornisce funzionalità di sicurezza avanzate, come il controllo dell'ammissione, l'autenticazione e l'autorizzazione, garantendo la conformità agli standard di sicurezza come il CIS Kubernetes Benchmark.

- **Ampia Community e Supporto**: La community attiva e vivace di Kubernetes è un ulteriore punto di forza, con una vasta rete di sviluppatori e utenti pronti a fornire supporto e contribuire al progetto. Inoltre, sono disponibili numerose risorse per imparare e utilizzare Kubernetes, tra cui documentazione, tutorial, corsi online e libri. Infine, diversi fornitori offrono supporto commerciale per facilitare l'adozione e la gestione della piattaforma.

Oltre a questi vantaggi, Kubernetes offre anche un miglioramento dell'osservabilità delle applicazioni grazie alla fornitura di metriche e log dettagliati. Inoltre, facilita l'adozione di metodologie DevOps e Agile, accelerando il processo di sviluppo e deployment delle applicazioni, e promuove una cultura di collaborazione tra team di sviluppo e operation, favorendo la comunicazione e la condivisione delle conoscenze.

Componenti principali

[6]Un cluster Kubernetes è composto da diversi componenti principali:

- **Nodo**: Un nodo rappresenta l'unità di elaborazione fondamentale in un cluster Kubernetes. Può essere una macchina fisica o virtuale che esegue il software Kubernetes e ospita i container. I nodi possono essere di diverse tipologie, con differenti capacità e caratteristiche, per soddisfare le esigenze specifiche delle applicazioni in esecuzione.

- **Pod**: Un è una raccolta di uno o più container che vengono eseguiti insieme su un nodo, nonché la più piccola unità di un'applicazione Kubernetes. I container all'interno di un pod condividono lo stesso spazio di rete e IPC, consentendo una stretta collaborazione e comunicazione. I pod sono la componente fondamentale per l'esecuzione di applicazioni in Kubernetes e possono essere configurati in vari modi per soddisfare diverse esigenze.

- **Servizio**: Un servizio rappresenta un'astrazione che espone un gruppo di pod come un unico servizio. I servizi possono essere utilizzati per bilanciare il carico tra i pod e per esporre le applicazioni all'esterno del cluster.
- **Deployment**: Un deployment specifica il numero desiderato di repliche di un pod e la loro configurazione. Il deployment è un oggetto che controlla la creazione e la gestione dei pod, garantendo che il numero desiderato di repliche sia sempre in esecuzione.
- **ReplicaSet**: Un replica set è un controller che implementa il concetto di replica desiderata. Il replica set gestisce la creazione e la rimozione dei pod in base al numero desiderato di repliche specificato in un deployment.
- **Controller**: Un controller è un componente software che implementa un ciclo di controllo per gestire lo stato desiderato del cluster e agisce per garantirne la conformità. Esistono diversi tipi di controller in Kubernetes, ognuno con un ruolo specifico.
- **Piano di Controllo**: Il piano di controllo è il componente centrale di Kubernetes che gestisce i cluster e le applicazioni. Il piano di controllo è composto da diversi componenti, tra cui l'API server e lo scheduler.
- **API Server**: L'API server è l'interfaccia principale per la gestione di Kubernetes. L'API server espone un'API RESTful che consente di creare, modificare ed eliminare risorse Kubernetes.
- **Scheduler**: Lo scheduler è responsabile dell'assegnazione dei pod ai nodi del cluster. Lo scheduler considera diverse metriche, come le risorse disponibili sui nodi e le affinità tra i pod, per assegnare i pod ai nodi in modo efficiente.
- **Kubelet**: È un agente che opera su ciascun nodo del cluster Kubernetes. Si assicura che i container siano in esecuzione correttamente all'interno dei pod, seguendo le direttive ricevute dal control plane di Kubernetes riguardanti quali pod devono essere eseguiti su quel nodo e come farlo. Monitora costantemente lo stato dei pod, avviandoli, terminandoli o aggiornandoli se necessario, e gestisce i volumi, montandoli e garantendo l'accesso ai container.
- **Kube-proxy**: È un componente di rete che funziona su ciascun nodo del cluster Kubernetes. Ha il compito di gestire le regole di rete per consentire la comunicazione dei servizi all'interno del cluster. Implementa il concetto di servizi in Kubernetes, consentendo ai servizi di essere raggiungibili attraverso IP e porta stabili, indipendentemente dai pod sottostanti. Kube-proxy utilizza diverse modalità di bilanciamento del carico per distribuire il traffico tra i pod di un servizio e gestisce il routing del traffico in base alle regole definite nelle risorse Service di Kubernetes.

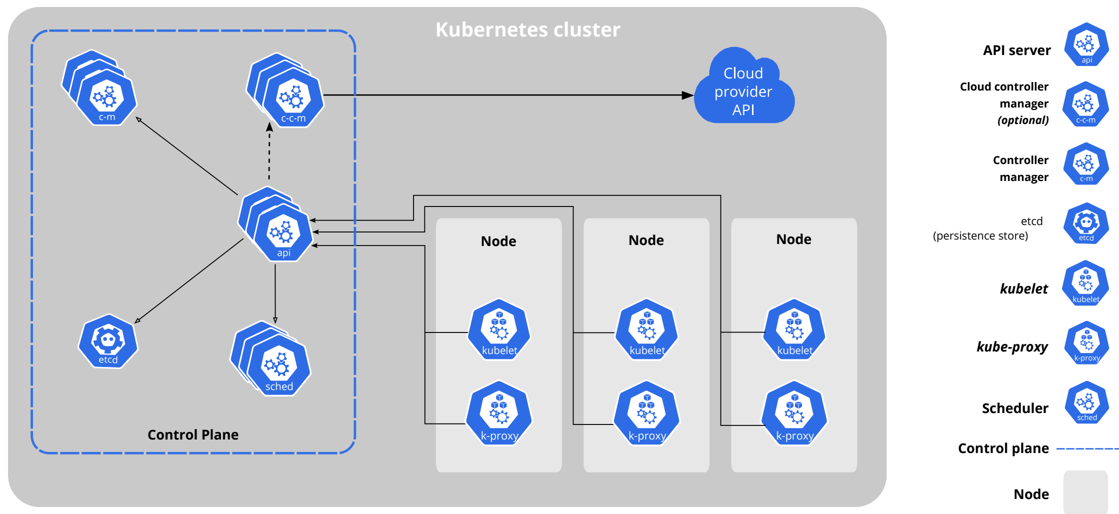

Figura 2.1: Componenti Kubernetes

2.6 AWS

Amazon Web Services (AWS) è la piattaforma di cloud computing più diffusa al mondo, offerta da Amazon. Fornisce una vasta gamma di servizi infrastrutturali e applicativi, consentendo alle organizzazioni di sfruttare risorse informatiche su richiesta, quali server, storage, database e molto altro, senza dover investire nell'acquisto e nella gestione di infrastrutture fisiche.

AWS offre una vasta gamma di servizi, suddivisi in diverse categorie:

- **Calcolo**: Amazon Elastic Compute Cloud (EC2) permette di avviare istanze virtuali con diversi tipi di risorse, adattabili alle esigenze specifiche dell'applicazione. EC2 offre scalabilità, flessibilità e controllo completo sull'ambiente di esecuzione.

- **Archiviazione**: Amazon Simple Storage Service (S3) è un servizio di storage oggetti altamente scalabile e affidabile, ideale per archiviare dati, file multimediali e backup. S3 offre una disponibilità elevata e una durabilità dei dati garantita.

- **Database**: AWS offre una vasta gamma di servizi di database, tra cui Amazon Relational Database Service (RDS) per database relazionali, Amazon DynamoDB per database NoSQL completamente gestiti e Amazon Redshift per l'analisi di grandi dataset.

- **Networking**: Amazon Virtual Private Cloud (VPC) consente di creare una rete virtuale isolata all'interno dell'infrastruttura di AWS, fornendo controllo completo sulla configurazione della rete, inclusa la creazione di subnet, routing e accesso sicuro tramite VPN o connessioni Direct Connect.
- **Machine Learning e Intelligenza Artificiale**: AWS offre una serie di servizi per lo sviluppo e l'implementazione di modelli di machine learning, tra cui Amazon SageMaker per la creazione, l'addestramento e il rilascio di modelli ML, e Amazon Rekognition per l'analisi delle immagini e il riconoscimento dei volti.
- **Containers**: Amazon Elastic Kubernetes Service (EKS) consente di eseguire, gestire e scalare applicazioni basate su container utilizzando Kubernetes. EKS semplifica la gestione dell'infrastruttura sottostante, consentendo agli sviluppatori di concentrarsi sullo sviluppo delle applicazioni.

Amazon EC2: Elastic Compute Cloud

[7]Amazon Elastic Compute Cloud (EC2) è uno dei servizi di cloud computing più fondamentali offerti da Amazon Web Services (AWS). EC2 consente agli utenti di eseguire macchine virtuali (note come istanze) su richiesta, consentendo loro di scalare orizzontalmente e verticalmente in base alle esigenze di carico di lavoro. Questo servizio fornisce una vasta gamma di opzioni di configurazione, tra cui diverse istanze con diverse capacità di calcolo, memoria e storage.

L'architettura di base di Amazon EC2 si compone di diverse componenti chiave:

- **Istanze EC2**: Queste sono le macchine virtuali che possono essere istanziate e gestite dagli utenti. Le istanze EC2 sono disponibili in una varietà di dimensioni e tipologie, che vanno dalle istanze ottimizzate per la computazione a quelle ottimizzate per la memoria, la rete o lo storage.
- **Amazon Machine Images (AMI)**: Le AMI sono modelli virtuali preconfigurati che contengono una configurazione del sistema operativo, dell'applicazione e dei dati di avvio. Gli utenti possono utilizzare AMI predefinite fornite da AWS o creare le proprie AMI personalizzate per avviare le loro istanze EC2.
- **Regole di sicurezza**: EC2 utilizza i gruppi di sicurezza per controllare il traffico di rete verso le istanze EC2. I gruppi di sicurezza consentono agli utenti di definire regole di firewall per specificare quali tipi di traffico sono ammessi o vietati verso le istanze.
- **Elastic Block Store (EBS)**: Amazon EBS fornisce storage persistente per le istanze EC2. Gli utenti possono creare volumi EBS e attaccarli alle loro

istanze per memorizzare dati che devono persistere oltre la vita dell'istanza stessa.

Una delle caratteristiche principali di Amazon EC2 è la sua capacità di scalabilità e flessibilità. Gli utenti possono facilmente ridimensionare le risorse computazionali in base alle esigenze di carico di lavoro. Questo può essere fatto aggiungendo o rimuovendo istanze, modificando le dimensioni delle istanze esistenti o utilizzando servizi aggiuntivi come Elastic Load Balancing per distribuire il traffico tra più istanze.

Amazon EC2 offre prestazioni affidabili attraverso la distribuzione delle istanze su più zone di disponibilità all'interno di una regione. Le zone di disponibilità sono data center separate e fisicamente isolate, progettate per garantire la massima resilienza e disponibilità del servizio.

Inoltre, EC2 offre un'ampia gamma di opzioni di monitoraggio e gestione delle prestazioni attraverso servizi come Amazon CloudWatch, che consente agli utenti di monitorare le risorse EC2 e ricevere avvisi in tempo reale sulle prestazioni e sulla salute delle istanze.

In sintesi, Amazon EC2 fornisce un'infrastruttura di calcolo altamente scalabile, flessibile e affidabile che consente agli utenti di eseguire e gestire facilmente le loro applicazioni e carichi di lavoro in cloud.

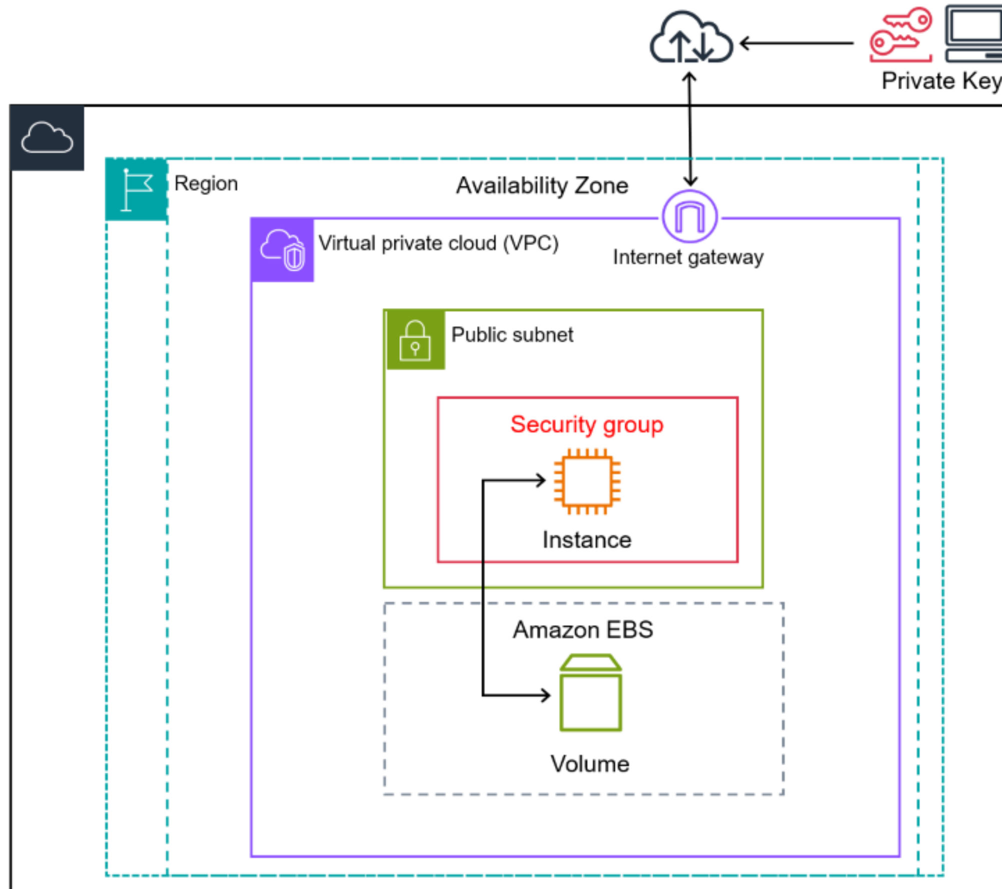

Figura 2.2: Architettura di Amazon Ec2

Amazon Elastic Kubernetes Service (Amazon EKS)

[8]Amazon Elastic Kubernetes Service (Amazon EKS) è un servizio gestito che semplifica la distribuzione, la gestione e la scalabilità delle applicazioni containerizzate utilizzando Kubernetes su Amazon Web Services (AWS). Amazon EKS semplifica l'esecuzione di Kubernetes su AWS senza dover gestire il controllo del piano di lavoro e senza dover installare, operare e gestire server Kubernetes. Questo servizio consente alle organizzazioni di focalizzarsi sullo sviluppo delle applicazioni senza dover preoccuparsi della complessità dell'infrastruttura sottostante.

- **Caratteristiche principali**: Gestione semplificata del cluster Amazon EKS semplifica la gestione del cluster Kubernetes, consentendo agli sviluppatori di concentrarsi sullo sviluppo delle applicazioni senza dover gestire l'infrastruttura sottostante. Il servizio gestisce automaticamente i nodi, fornisce aggiornamenti per il software Kubernetes e gestisce la disponibilità e la scalabilità del cluster.

- **Sicurezza**: Amazon EKS offre un alto livello di sicurezza per le applicazioni containerizzate. Utilizza le funzionalità di sicurezza di AWS per proteggere i dati e le risorse del cluster. Inoltre, supporta le politiche IAM (Identity and Access Management) per controllare l'accesso alle risorse del cluster e alle API Kubernetes.

- **Integrazione con altri servizi AWS**: Amazon EKS si integra con altri servizi AWS, consentendo agli utenti di utilizzare servizi come Amazon EC2, AWS Fargate, Amazon EBS e Amazon VPC per le loro applicazioni containerizzate. Questa integrazione semplifica lo sviluppo, la distribuzione e la gestione delle applicazioni su AWS.

- **Scalabilità**: Amazon EKS offre la possibilità di scalare rapidamente le applicazioni in base alle esigenze di carico di lavoro. Supporta l'auto-scaling dei nodi del cluster, consentendo di aggiungere o rimuovere risorse di calcolo in modo dinamico in base alla domanda.

- **Monitoraggio e logging**: Amazon EKS fornisce strumenti di monitoraggio e logging per tenere traccia delle prestazioni e del comportamento delle applicazioni containerizzate. Supporta l'integrazione con servizi come Amazon CloudWatch e AWS CloudTrail per la registrazione e l'analisi dei log.

Cluster Auto-scaling

Il cluster auto-scaling su Amazon Elastic Kubernetes Service (EKS) è un'opzione che consente di gestire dinamicamente le risorse del cluster Kubernetes in base alla domanda effettiva delle applicazioni in esecuzione. Funziona in modo simile all'auto-scaling dei gruppi di istanze EC2.

Ecco come funziona il cluster auto-scaling su EKS:

- **Pianificazione delle risorse**: Per abilitare l'auto-scaling, è necessario definire una politica di auto-scaling per il cluster, che determina come e quando aggiungere o rimuovere nodi dal cluster in base a metriche specifiche, come l'utilizzo della CPU o della memoria.

- **Integrazione con EC2 Auto Scaling**: Amazon EKS utilizza EC2 Auto Scaling per gestire le istanze EC2 che costituiscono i nodi del cluster. Puoi definire le tue politiche di auto-scaling utilizzando EC2 Auto Scaling, che poi verranno applicate al tuo cluster EKS.

- **Configurazione delle metriche**: Puoi configurare le metriche da monitorare per determinare quando scalare il cluster. Queste metriche possono includere

l'utilizzo della CPU, la memoria disponibile, il numero di pod in esecuzione e altre metriche rilevanti per le tue applicazioni.

- **Configurazione dei gruppi di nodi**: È possibile definire i gruppi di nodi che costituiranno il cluster e specificare le dimensioni minime e massime per ciascun gruppo. In base alle metriche di auto-scaling configurate, EC2 Auto Scaling aggiungerà o rimuoverà istanze da questi gruppi di nodi.

- **Gestione dinamica delle risorse**: Una volta configurato, il cluster auto-scaling monitorerà continuamente le metriche definite e apporterà automaticamente aggiustamenti alla dimensione del cluster per adattarsi alle esigenze delle applicazioni in esecuzione. Ciò garantisce che il cluster utilizzi le risorse in modo efficiente, evitando sia sprechi sia situazioni di sottoutilizzo.

In sintesi, il cluster auto-scaling su Amazon EKS consente di ottimizzare l'utilizzo delle risorse del cluster Kubernetes in modo dinamico e automatizzato, garantendo che il tuo ambiente di container sia scalabile e reattivo alle variazioni del carico di lavoro.

Amazon Virtual Private Cloud (VPC)

Amazon Virtual Private Cloud (VPC) è un servizio fondamentale offerto da AWS che consente agli utenti di creare e gestire un'area di rete isolata all'interno dell'infrastruttura cloud di AWS. Le VPC consentono agli utenti di avere un controllo granulare sulla configurazione della rete, inclusi gli indirizzi IP, le tabelle di route e le impostazioni di sicurezza.

Le VPC sono composte da un insieme di componenti chiave:

- **CIDR Blocks**: Una VPC è definita da un blocco CIDR (Classless Inter-Domain Routing) che determina l'intervallo di indirizzi IP disponibili all'interno della VPC stessa.

- **Subnet**: All'interno di una VPC, è possibile creare subnet, ciascuna delle quali è associata a un intervallo di indirizzi IP all'interno del blocco CIDR della VPC. Le subnet possono essere pubbliche o private, a seconda della loro accessibilità dalla rete pubblica di Internet.

- **Tabelle di route**: Ogni subnet in una VPC è associata a una tabella di route che definisce come instradare il traffico in ingresso e in uscita. Le tabelle di route possono essere personalizzate per instradare il traffico verso Internet, verso altre subnet nella stessa VPC o verso endpoint specifici.

- **Security Groups**: I security group sono una forma di firewall virtuale che controlla il traffico in ingresso e in uscita per le istanze all'interno della VPC. Possono essere configurati per consentire o bloccare specifici tipi di traffico in base alle regole definite dall'utente.

Amazon RDS

Amazon Relational Database Service (Amazon RDS) è un servizio di database relazionale basato su cloud offerto da Amazon Web Services (AWS). Lanciato nel 2009, Amazon RDS ha rivoluzionato il modo in cui le organizzazioni gestiscono i loro database, offrendo una soluzione scalabile, gestita e altamente disponibile per una vasta gamma di motori di database relazionali.

Le caratteristiche principali di Amazon RDS sono:

- **Gestione Semplificata**: Amazon RDS semplifica notevolmente la gestione dei database, eliminando la necessità di installare, configurare e mantenere l'infrastruttura sottostante. Con pochi clic attraverso la console di gestione AWS o tramite API, gli utenti possono facilmente creare istanze di database, eseguire il backup dei dati, scalare le risorse e monitorare le prestazioni.

- **Multi-Motore**: Amazon RDS supporta diversi motori di database relazionali, tra cui MySQL, PostgreSQL, MariaDB, Oracle e Microsoft SQL Server. Questo offre flessibilità agli utenti per scegliere il motore di database più adatto alle loro esigenze senza dover gestire diverse piattaforme di database.

- **Backup e Ripristino Automatico**: Amazon RDS automatizza il processo di backup dei dati e offre funzionalità di ripristino point-in-time, consentendo agli utenti di ripristinare rapidamente i propri database a uno stato precedente specifico in caso di errori o perdita di dati.

- **Scalabilità Orizzontale e Verticale**: Con Amazon RDS, gli utenti possono facilmente scalare le risorse del database sia orizzontalmente (aggiungendo più istanze di database) che verticalmente (aumentando le dimensioni delle risorse delle istanze esistenti) per gestire carichi di lavoro in crescita o per migliorare le prestazioni.

- **Alta Disponibilità**: Amazon RDS offre opzioni per configurare le istanze di database in configurazioni multi-AZ (disponibilità zone) per garantire la massima disponibilità e ridondanza. In caso di guasto di un'area di disponibilità, Amazon RDS commuta automaticamente il traffico su un'istanza di backup in un'altra zona, garantendo la continuità del servizio.

- **Sicurezza**: Amazon RDS implementa misure di sicurezza avanzate per proteggere i dati sensibili degli utenti. Queste includono il supporto per l'accesso basato su ruoli (IAM), crittografia dei dati in transito e a riposo, e la possibilità di isolare i database in reti private virtuali (VPC).
- **Monitoraggio e Avvisi**: Amazon RDS fornisce strumenti integrati per monitorare le prestazioni del database e ricevere avvisi in tempo reale su eventi critici o anomalie, consentendo agli utenti di reagire prontamente a potenziali problemi di prestazioni o disponibilità.

Capitolo 3

Design e implementazione

In questo capitolo parleremo dell'obiettivo della tesi e delle caratteristiche dell'applicazione che è stata sviluppata nei due framework in esame.

3.1 Obiettivo della tesi

L'obiettivo di questa tesi è sviluppare una piccola applicazione, suddivisa in microservizi, che sia il più realistica possibile. In questo modo, si potranno trarre considerazioni significative sul lavoro necessario per sviluppare applicazioni utilizzando due diversi framework: Spring Boot e Quarkus.

Per raggiungere questo obiettivo, si è deciso di ricreare una parte di un'applicazione già sviluppata in azienda con tecnologie completamente differenti (low-code) utilizzando il linguaggio Java. La parte in questione simula il processo di un utente già registrato che vuole effettuare il login e successivamente recuperare informazioni relative all'azienda per formulare una richiesta di approvvigionamento aziendale.

Inizialmente, è stata realizzata l'intera architettura utilizzando Spring Boot. Successivamente, si è cercato di adattare il codice precedentemente scritto al framework Quarkus. Si sono quindi ottenute due versioni diverse dell'applicazione che, pur avendo lo stesso funzionamento e identiche API, permettono di testare e confrontare le loro performance.

Da studi precedentemente condotti[9] sembrerebbe che Quarkus performi meglio di Spring Boot quando containerizzato in kubernetes, si vuole vedere se si ottengono gli stessi risultati una volta che verrà fatto il deployment dell'applicazione su AWS EKS o se ci sono altre complicazioni relative all'esecuzione in cloud.

3.2 Sviluppo

Nella presente sezione, esploreremo l'implementazione dell'applicazione nei due framework: Spring Boot e Quarkus.

Inizieremo analizzando l'architettura generale del sistema, evidenziando i principali componenti e le interazioni tra di essi. Successivamente, approfondiremo l'implementazione utilizzando il framework Spring Boot, mettendo in luce le caratteristiche, le funzionalità e le best practice che guidano il processo di sviluppo.

Continueremo quindi esaminando l'implementazione alternativa del sistema con il framework Quarkus, esplorando le sue peculiarità, le prestazioni e le differenze rispetto all'approccio basato su Spring Boot.

3.2.1 Architettura

L'architettura dei microservizi sviluppata è composta da tre microservizi:

1. **Login**:
   - Gestisce l'autenticazione degli utenti mediante username e password.
   - Memorizza le informazioni di autenticazione in un database RDS (Relational Database Service).
2. **Gateway**:
   - Agisce come punto di ingresso per le richieste esterne.
   - Verifica l'autenticazione dell'utente tramite token JWT (JSON Web Token).
   - Inoltra il traffico al microservizio appropriato in base all'URL della richiesta.
3. **Request**:
   - Si interfaccia con API esterne per recuperare dati.
   - Esegue principalmente chiamate GET per i benchmark.
   - Gestisce i dati ricevuti e li restituisce al client.

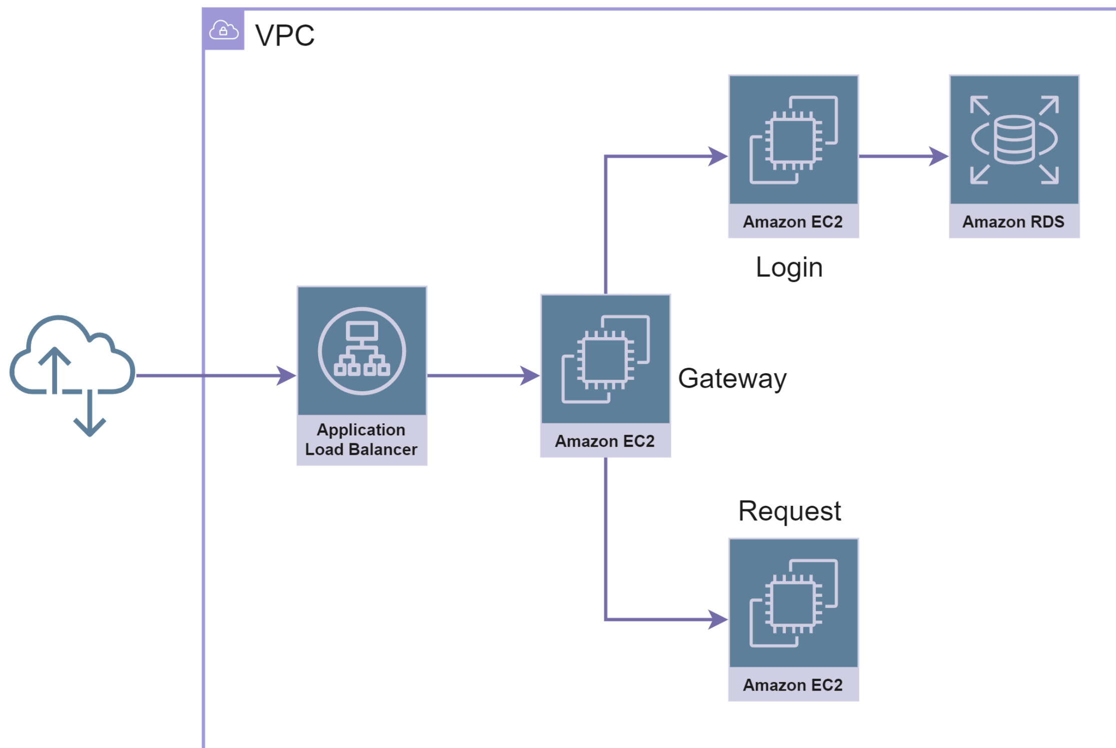

Figura 3.1: Architettura

JWT

I token JSON Web Token (JWT) sono uno standard aperto (RFC 7519) per la trasmissione sicura di informazioni tra due parti. Sono composti da tre parti: un'intestazione, un payload e una firma. L'intestazione contiene informazioni sul tipo di token e sull'algoritmo di firma utilizzato. Il payload contiene i dati veri e propri, come l'identificativo dell'utente, i ruoli e le autorizzazioni. La firma viene utilizzata per garantire l'integrità e l'autenticità del token.[10]

I token JWT sono utilizzati in una varietà di applicazioni, tra cui:

- **Autenticazione e autorizzazione**: I token JWT possono essere utilizzati per autenticare gli utenti e per autorizzare l'accesso alle risorse.
- **Single sign-on (SSO)**: I token JWT possono essere utilizzati per implementare il single sign-on, che consente agli utenti di accedere a più applicazioni con un'unica autenticazione.
- **Scambio di dati**: I token JWT possono essere utilizzati per scambiare dati in modo sicuro tra due parti.

La generazione della firma di un token JWT avviene attraverso l'utilizzo di un algoritmo di firma e una chiave segreta. L'algoritmo di firma calcola un hash dell'intestazione e del payload del token, che viene poi firmato utilizzando la chiave segreta. La firma viene codificata in Base64 e aggiunta al token.

Per validare un token JWT, il server destinatario deve procedere nel seguente modo: decodificare la firma del token, calcolare un hash dell'intestazione e del payload del token e confrontare l'hash calcolato con quello firmato. Solo se gli hash coincidono, il token è considerato valido; in caso contrario, deve essere rifiutato.

Esistono diversi algoritmi di firma utilizzabili per firmare i token JWT. Tra i più comuni troviamo HS256, un algoritmo simmetrico che sfrutta una chiave segreta condivisa tra server e client, e RS256, un algoritmo asimmetrico che richiede l'utilizzo di una coppia di chiavi: una privata, utilizzata per firmare il token, e una pubblica, impiegata per la convalida della firma.

Nel nostro caso, adottiamo un approccio algoritmico asimmetrico, in cui la chiave privata risiede nel microservizio di Login. Questa chiave viene impiegata per generare i token JWT e restituirli al client. Al contempo, la chiave pubblica è memorizzata nel microservizio di gateway e viene utilizzata per convalidare i token JWT inclusi nelle richieste provenienti dal client.

Microservizio di Login

Il microservizio di login gestisce l'autenticazione degli utenti. L'utente fornisce le sue credenziali (username e password) al microservizio tramite una richiesta HTTP POST. Il microservizio esegue quindi le seguenti operazioni:

1. **Validazione della richiesta**: verifica che la richiesta sia formata correttamente e che i dati necessari siano presenti.

2. **Ricerca dell'utente**: Recupera l'utente dal database relazionale in base al nome utente fornito.

3. **Verifica della password**: Confronta la password fornita con la password memorizzata nel database per l'utente identificato.

4. **Generazione del token JWT**: Se la password è corretta, il microservizio genera un token JWT che codifica:

    - L'identificativo univoco dell'utente.
    - Il suo ruolo.

    - Altre informazioni utili.

5. **Restituzione della risposta**: Il microservizio invia una risposta HTTP al client che include:
    - Il token JWT.
    - Informazioni aggiuntive sull'utente come nome completo e ruolo.

Microservizio di Gateway

Il secondo microservizio sviluppato è il Gateway, concepito per agire come punto di ingresso unificato per le richieste esterne. Implementando un design API Gateway, questo componente svolge un ruolo fondamentale nell'architettura dei microservizi, apportando numerosi vantaggi:

- **Autenticazione e Autorizzazione**: Il Gateway intercetta le richieste in ingresso e verifica l'autenticazione degli utenti tramite l'utilizzo di token JWT. In base al ruolo associato al token, il Gateway autorizza o nega l'accesso alle risorse interne, garantendo così la sicurezza del sistema.
- **Routing Intelligente**: Il Gateway agisce come un instradatore intelligente, analizzando l'URL di ogni richiesta e inoltrandola al microservizio appropriato. Questo meccanismo di routing flessibile consente di gestire un elevato numero di microservizi in modo efficiente e scalabile.

In particolare, tutte le richieste di autenticazione vengono indirizzate al servizio di login, mentre tutte le altre sono inviate all'altro microservizio.

Microservizio di Request

Il terzo microservizio, accessibile solo dopo l'autenticazione, si interfaccia con la piattaforma low-code Pega, utilizzata da Blue Reply, per recuperare dati utili ai benchmark. In particolare, il microservizio esegue le seguenti operazioni:

1. **Richiesta API**:
    - Invia una chiamata GET verso un'API specifica di Pega.
    - L'API fornisce informazioni su delle Business Unit (BU) che simulano la suddivisione reale dell'azienda, e successive informazioni per formulare una richiesta d'acquisto.
2. **Elaborazione dati**:
    - Gestisce i dati ricevuti dall'API Pega, in formato JSON di circa 1.3KB.

- Elimina tutte le informazioni superflue fornite da Pega riducendone la dimensione a circa 0.7KB

3. **Restituzione dati**: Restituisce i dati elaborati al client.

Maven

Maven rappresenta uno strumento fondamentale per la gestione e l'automatizzazione delle build all'interno di progetti software di varie dimensioni e complessità. Nato dall'esigenza di superare le limitazioni dei precedenti strumenti di build come Ant, Maven si è affermato come uno standard nel panorama dello sviluppo software Java. In questa sezione, esploreremo in dettaglio le caratteristiche principali di Maven, il suo ruolo nel ciclo di sviluppo del software e i benefici che porta alla gestione del progetto.

Maven è costruito su un'architettura modulare e basata sui plugin. Il cuore di Maven è rappresentato dal Maven Core, che fornisce le funzionalità di base per la gestione dei progetti e la loro compilazione. Il vero potenziale di Maven emerge attraverso l'uso dei plugin, che estendono le funzionalità di base per supportare attività specifiche come la compilazione, i test, il packaging e la distribuzione del software.

La struttura di un progetto Maven è definita da un file di configurazione denominato "pom.xml" (Project Object Model). Questo file contiene le informazioni essenziali sul progetto, come le dipendenze, i plugin utilizzati, i repository remoti e le configurazioni di build. Grazie a questa struttura standardizzata, i progetti Maven sono altamente portabili e facilmente condivisibili tra i membri del team e attraverso le community open source.

Uno dei concetti fondamentali di Maven è il ciclo di vita delle build. Maven definisce una serie predefinita di fasi attraverso le quali passa ogni progetto durante il processo di build. Queste fasi includono "validate", "compile", "test", "package", "verify", "install" e "deploy". Ogni fase corrisponde a una serie di obiettivi predefiniti e può essere estesa o personalizzata utilizzando plugin specifici.

Il ciclo di vita delle build di Maven promuove una metodologia di sviluppo basata su best practice e automatizzata. Ad esempio, durante la fase di "test", Maven esegue automaticamente tutti i test definiti nel progetto, garantendo un approccio rigoroso alla verifica della qualità del software. Grazie a questa automatizzazione, i team di sviluppo possono concentrarsi maggiormente sullo sviluppo del codice senza dover gestire manualmente le diverse fasi della build.

Un'altra caratteristica chiave di Maven è la gestione delle dipendenze. Utilizzando il concetto di repository, Maven consente ai progetti di specificare le proprie dipendenze da librerie esterne e di risolverle automaticamente durante il processo di build. Maven supporta sia i repository locali che remoti, consentendo ai team di accedere a un vasto ecosistema di librerie e componenti software open source.

La gestione delle dipendenze di Maven offre numerosi vantaggi, tra cui la ridondanza delle librerie, la facilità di aggiornamento e la gestione centralizzata delle versioni. Inoltre, Maven garantisce la coerenza e la riproducibilità delle build, fornendo un meccanismo affidabile per la gestione delle dipendenze del progetto.

3.2.2 Implementazione con Spring Boot

Nell'ambito di questa tesi focalizzata sull'analisi dei framework, la fase di implementazione si è concentrata sull'impiego dei pattern e delle dipendenze cardine offerti da Spring Boot. L'obiettivo primario è stato quello di sfruttare appieno le potenzialità di questo framework, garantendo al contempo una struttura solida e scalabile all'applicazione sviluppata. Pertanto, di seguito, esploreremo in dettaglio le tecniche e le librerie utilizzate durante l'implementazione, approfondendo il loro ruolo e l'impatto nell'architettura complessiva dell'applicazione.

Bean

L'annotazione @Bean in Spring Boot rappresenta un elemento chiave per la definizione di componenti gestiti dal container di IoC (Inversion of Control). Essa permette di creare e registrare oggetti all'interno del contesto Spring, rendendoli disponibili per l'iniezione di dipendenze in altre parti dell'applicazione. Ecco il loro utilizzo:

1. **Definizione dei bean**: I bean possono essere definiti in diversi modi. Uno dei modi più comuni è utilizzando annotazioni come @Component, @Service, @Repository o @Controller sopra una classe Java. Ad esempio, una classe annotata con @Component sarà gestita come un bean dal container IoC di Spring. Alternativamente, è possibile definire i bean tramite configurazione XML o utilizzando JavaConfig, fornendo una maggiore flessibilità nella configurazione.

2. **Scansione del package**: Quando l'applicazione Spring Boot viene avviata, il container IoC esegue una scansione del package dell'applicazione alla ricerca di classi contrassegnate con le annotazioni specifiche (@ComponentScan). Questa scansione identifica automaticamente i bean e li registra nel contesto dell'applicazione.

3. **Creazione dei bean**: Una volta individuati, i bean vengono istanziati dal container IoC di Spring. Il container si occupa di creare le istanze dei bean e risolvere le loro dipendenze, garantendo che ogni bean abbia accesso alle risorse necessarie per funzionare correttamente.

4. **Iniezione delle dipendenze**: Uno dei vantaggi principali dei bean è la possibilità di iniettare dipendenze tra di loro. Questo significa che un bean può fare riferimento ad altri bean di cui dipende direttamente o indirettamente. Spring Boot gestisce automaticamente l'iniezione delle dipendenze utilizzando annotazioni come @Autowired, @Inject o @Resource.

5. **Scopo dei bean**: Ogni bean può avere uno specifico scopo o ciclo di vita definito tramite annotazioni come @Scope. Lo scope determina quanto a lungo un bean rimane vivo nel contesto dell'applicazione e come viene gestita la sua creazione e distruzione.

6. **Gestione dei bean**: Spring Boot si occupa della gestione dei bean durante l'intero ciclo di vita dell'applicazione. Ciò include la creazione, l'iniezione delle dipendenze, la distruzione dei bean non più necessari e la gestione delle transazioni, se necessario.

Spring Boot Starter Web

La dipendenza `spring-boot-starter-web` è un componente essenziale nell'ecosistema di sviluppo di Spring Boot per la creazione di applicazioni web robuste e scalabili. Quando si utilizza questa dipendenza, Spring Boot configura automaticamente molti aspetti dell'applicazione web, riducendo notevolmente il lavoro di configurazione manuale richiesto agli sviluppatori.

Una delle principali funzionalità fornite da questa dipendenza è il framework Spring MVC (Model-View-Controller), che semplifica lo sviluppo delle applicazioni web seguendo il pattern architetturale MVC. Spring MVC gestisce le richieste HTTP, mappandole ai metodi dei controller e consentendo una gestione pulita e modulare delle varie operazioni dell'applicazione, come l'elaborazione dei dati e la generazione delle risposte.

Inoltre, la dipendenza include un container servlet embedded, come Tomcat o Jetty, che permette di eseguire l'applicazione senza la necessità di configurare un server esterno. Questo semplifica notevolmente il processo di sviluppo e distribuzione, consentendo agli sviluppatori di concentrarsi esclusivamente sulla logica dell'applicazione senza preoccuparsi della configurazione del server.

Altre funzionalità offerte da Spring Boot Starter Web includono la gestione delle risorse statiche, come file CSS, JavaScript e immagini, che vengono automaticamente servite senza necessità di configurazione aggiuntiva. Inoltre, fornisce supporto per la serializzazione JSON tramite librerie come Jackson, semplificando la trasformazione degli oggetti Java in formato JSON per l'invio delle risposte alle richieste HTTP.

Inoltre, la dipendenza gestisce in modo trasparente la gestione delle eccezioni, consentendo agli sviluppatori di definire in modo centralizzato come gestire errori comuni, migliorando l'esperienza dell'utente e semplificando il debugging dell'applicazione.

In conclusione, la dipendenza Spring Boot Starter Web semplifica notevolmente lo sviluppo di applicazioni web basate su Spring Boot, fornendo una serie di funzionalità preconfigurate che consentono agli sviluppatori di concentrarsi sulla creazione della logica senza dover gestire dettagli infrastrutturali complessi.

Ad esempio per creare un web server che espone delle semplici API basta scrivere:

```
@RestController
@RequestMapping("/api")
public class LoginController {
    @GetMapping("test")
    public String test(){
        return "hello world";
    }
}
```

In particolare l'annotazione @RestController in Spring Boot viene utilizzata per dichiarare una classe come un controller REST. Questo significa che ogni metodo all'interno di questa classe viene considerato un endpoint REST e il framework si aspetta che i risultati di tali metodi vengano convertiti direttamente in formato JSON e inclusi nella risposta HTTP. L'annotazione @RestController combina l'annotazione @Controller di Spring con l'annotazione @ResponseBody, semplificando il processo di sviluppo di API RESTful, in quanto elimina la necessità di annotare esplicitamente ogni metodo con @ResponseBody.

L'annotazione @RequestMapping("/api") viene utilizzata per mappare ogni richiesta HTTP che inizia con /api e indirizzarla a questa classe.

Similmente @GetMapping("test") permette di indirizzare le operazioni di HTTP GET per /test alla funzione test().

Spring Data JDBC

Spring Data JDBC è un framework di persistenza dati leggero e efficiente che offre un'alternativa agli approcci ORM tradizionali come Hibernate. Basato sul principio del mapping diretto degli oggetti al database relazionale, Spring Data JDBC mira a semplificare lo sviluppo di applicazioni che richiedono interazioni con il database, riducendo al minimo la complessità e l'overhead associati agli strati di astrazione aggiuntivi.

Un aspetto distintivo di Spring Data JDBC è la sua aderenza ai principi del design semplice e della trasparenza, fornendo agli sviluppatori un controllo più diretto sulle query SQL generate e sulle operazioni di persistenza. Utilizzando semplici annotazioni Java e convenzioni di denominazione, è possibile mappare facilmente le classi di dominio alle tabelle del database e definire le relazioni tra di esse. Nello specifico:

1. Spring Data JDBC utilizza annotazioni per mappare le entità Java alle tabelle del database. Le annotazioni più comuni sono:

    - @Table: Specifica il nome della tabella del database associata all'entità.
    - @Id: Identifica la proprietà della classe che rappresenta la chiave primaria della tabella.

    Quindi ho definito

    ```
    @Table("USER_INFO")
    public record UserInfo(
        @Id
        Integer id,
        String username,
        String fullName,
        Integer role,
        String password,
        Integer newPass ){
        public UserInfo {}
    }
    ```

    Per mappare il record UserInfo con la tabella USER_INFO all'interno del database

2. Spring Data JDBC fornisce l'interfaccia CrudRepository che implementa le operazioni CRUD (Create, Read, Update, Delete) di base per le entità. È

possibile creare interfacce repository personalizzate che estendono CrudRepository e definiscono metodi per eseguire query più complesse. Infatti la seguente interfaccia:

```
public interface UserRepository extends
CrudRepository<UserInfo, Integer> {
    UserInfo findFirstByUsername(String username);
}
```

Mi permette di effettuare in modo semplice e veloce una query SQL come:

```
SELECT * FROM UserInfo WHERE username = ?
```

3. Spring Data JDBC si occupa di gestire la connessione al database, l'esecuzione di query e la mappatura dei risultati alle entità Java. Per configurare la connessione al database bisogna farlo nel file `application.properties`, ad esempio per connettermi al database Oracle ho usato:

```
spring.datasource.url=jdbc:oracle:thin:@DB_URL:ORCL
spring.datasource.username=****
spring.datasource.password=****
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
```

Spring Security

La dipendenza Spring Security è un framework potente e altamente configurabile progettato per gestire la sicurezza nelle applicazioni Java basate su Spring. Al suo nucleo, Spring Security si occupa principalmente di due aspetti fondamentali della sicurezza delle applicazioni web: autenticazione e autorizzazione.

L'autenticazione è il processo di verifica delle credenziali dell'utente, come nome utente e password, per confermare la sua identità. Spring Security offre un'ampia gamma di opzioni per l'autenticazione, tra cui autenticazione basata su form, autenticazione basata su token (come JSON Web Token), autenticazione basata su certificati e altro ancora. Inoltre, supporta l'autenticazione tramite database utenti, LDAP (Lightweight Directory Access Protocol), OAuth, e integrazioni con servizi di autenticazione esterni.

Una volta che l'utente è stato autenticato con successo, viene gestita l'autorizzazione, che determina quali azioni può compiere all'interno dell'applicazione. Spring

Security consente di definire regole di autorizzazione basate su ruoli utente, consentendo di limitare l'accesso alle risorse dell'applicazione in base al ruolo dell'utente. Ad esempio, è possibile configurare regole che consentono solo agli utenti con il ruolo "amministratore" di accedere a determinate pagine o eseguire determinate operazioni.

Microservizio di Login

Il microservizio di autenticazione utilizza la dipendenza `spring-security` per gestire il processo di login. Le richieste all'endpoint di login `/auth/token` passano attraverso una `SecurityFilterChain` definito come:

```
@Bean
SecurityFilterChain tokenSecurityFilterChain(HttpSecurity http) {
    return http
        .securityMatcher(new AntPathRequestMatcher("/auth/token"))
        .authorizeHttpRequests(auth ->
            auth.anyRequest().authenticated())
        .addFilterBefore(customAuthenticationFilter,
            UsernamePasswordAuthenticationFilter.class)
        .httpBasic(withDefaults())
        .build();
}
```

che applica un filtro `cuscustomAuthenticationFilter` per:

1. **Verificare l'header della richiesta**: il filtro controlla che le informazioni di login (username e password) codificate tramite Basic Authentication header corrispondano a un utente presente nel database. L'estrazione delle informazioni dall'header non è gestito da Spring Security ma ho dovuto implementarlo esplicitamente.

2. **Generare il token JWT**: in caso di successo, il servizio genera un token JWT che codifica informazioni sull'utente (username, ruolo) e lo restituisce insieme alle informazioni dell'utente.

   ```
   public String generateToken(String name, String role) {
       Instant now = Instant.now();
       JwtClaimsSet claims = JwtClaimsSet.builder()
               .issuer("self")
               .issuedAt(now)
               .expiresAt(now.plus(1, ChronoUnit.HOURS))
               .subject(name)
   ```

```
                .claim("scope", role)
                .build();
        return this.encoder.encode(
          JwtEncoderParameters.from(claims)).getTokenValue();
    }
```

3. **Scartare la richiesta**: se le informazioni di login non sono corrette, la richiesta viene scartata.

Microservizio di Gateway

Il microservizio Gateway è il più semplice da implementare, grazie alla libreria `spring-cloud-starter-gateway`. Questa potente dipendenza permette di generare un gateway completo con poche righe di codice nel file `application.yaml`.

```
spring:
    cloud:
        gateway:
            routes:
            -   id: login
                uri: http://login-service-clusterip-service:8081
                predicates:
                    - Path= /auth/**
            -   id: requests
                uri: http://request-service-clusterip-service:8082
                predicates:
                    - Path= /api/requests/**
                filters:
                -   name: Auth
```

In effetti, sono sufficienti due definizioni di route: la prima indirizzata al servizio di login, per la quale il gateway non applica alcun controllo, mentre per il traffico destinato al micorservizio di request, è necessario definire un filtro che verifichi la validità del token JWT presente nella richiesta. Anche qui la gestione dell'header e la verifica del token sono state gestite esplicitamente.

Microservizio di Request

La scelta di utilizzare una libreria esterna per le chiamate API esterne nel microservizio è stata ponderata e significativa. L'obiettivo era valutare se l'utilizzo di una libreria esterna potesse causare problemi di compatibilità in caso di cambio di framework.

OkHttp

OkHttp è una libreria open source progettata per essere un client HTTP efficiente. Offre diverse funzionalità avanzate che la rendono una buona scelta per lo sviluppo di applicazioni Android e Java.

Ha come caratteristiche di default:

- **Supporto HTTP/2**: OkHttp sfrutta il protocollo HTTP/2 per migliorare le prestazioni e l'efficienza delle richieste HTTP.
- **Pool di connessioni**: La libreria gestisce un pool di connessioni riutilizzabili, riducendo la latenza e migliorando la scalabilità.
- **Compressione GZIP trasparente**: OkHttp può comprimere automaticamente le richieste e le risposte HTTP utilizzando la compressione GZIP, riducendo il consumo di dati.
- **Caching delle risposte**: La libreria consente di memorizzare nella cache le risposte HTTP per evitare di doverle recuperare nuovamente dal server, migliorando le prestazioni e riducendo il consumo di dati.

OkHttp è affidabile anche in situazioni di rete problematiche: Recupera automaticamente dai comuni problemi di connessione. Se un indirizzo IP non è raggiungibile, tenta automaticamente con indirizzi alternativi. Ciò è fondamentale per scenari IPv4+IPv6 o servizi ospitati in data center ridondanti. Supporta i protocolli di sicurezza TLS 1.3, ALPN, e anche il certificate pinning. Può essere configurato con opzioni di fallback per una compatibilità più ampia.

OkHttp è semplice da usare: la sua API per richieste e risposte è costruita su pattern di progettazione eleganti, offrendo immutabilità. Supporta sia chiamate bloccanti (sincrone) che chiamate asincrone gestite da callback. [11]

Ad esempio la seguente chiamata api al servizio esterno permette di ottenere un oggetto di tipo JsonNode, che funge da contenitore versatile per ottenere la risposta dal server, senza dover necessariamente definire una classe java specifica:

```
public JsonNode getJsonNode(String apiUrl) throws Exception{
    String credentials = "*" + ":" + "*";
    String base64Credentials = Base64.getEncoder()
        .encodeToString(credentials.getBytes());

    Request request = new Request.Builder()
    .url(apiUrl)
```

```
    .post(RequestBody.create("{}", MediaType.get("application/json")))
    .addHeader("Content-Type", "application/json")
    .addHeader("Authorization", "Basic "+ base64Credentials)
    .build();

    Call call = client.newCall(request);
    Response response = call.execute();

    String responseString=response.body().string();
    JsonNode rootNode = objectMapper.readTree(responseString);
    return rootNode.get("data");
}
```

3.2.3 Implementazione con Quarkus

Nella fase di riscrittura dei microservizi mediante l'impiego di Quarkus, si è mirato a conservare quanto più possibile la struttura preesistente dei servizi stessi. A tale scopo, per esempio, nella realizzazione del server si è optato per l'utilizzo di RESTEasy Reactive, una scelta che si presenta molto simile all'utilizzo di Spring Boot Starter Web. In questa sezione, esamineremo dettagliatamente il processo di riscrittura dei microservizi, focalizzandoci sulle decisioni prese, sulle tecniche impiegate e sull'integrazione delle funzionalità offerte da Quarkus nell'ambito dell'applicazione.

RESTEasy Reactive

`RESTEasy Reactive` in Quarkus è un'implementazione di JAX-RS moderna e performante per la creazione di API RESTful in Java. Si basa su Vert.x, una piattaforma di sviluppo reattiva ad alte prestazioni, e offre una serie di vantaggi rispetto alle soluzioni tradizionali:

- **Prestazioni elevate**: Quarkus-RESTEasy-Reactive sfrutta le capacità di Vert.x per raggiungere prestazioni elevate. Vert.x è una piattaforma di sviluppo reattiva basata su eventi che offre un'elevata scalabilità e un basso consumo di risorse.
- **Modelli di programmazione reattivi**: Permette di sviluppare API RESTful reattive, che possono gestire in modo efficiente richieste asincrone e flussi di dati. Questo è in contrasto con le API RESTful tradizionali, che sono basate su un modello di programmazione bloccante.
- **Integrazione con Quarkus**: Si integra perfettamente con l'ecosistema Quarkus, offrendo un'esperienza di sviluppo fluida e coerente. Quarkus è

una piattaforma di sviluppo Java che fornisce un ambiente ottimizzato per la creazione di applicazioni native e containerizzate.

- **Facilità d'uso**: Offre un'API intuitiva e facile da usare, che rende lo sviluppo di API RESTful un processo rapido e semplice. L'API è basata su annotazioni JAX-RS standard, che semplifica la scrittura di codice.
- **Ampia gamma di funzionalità**: Supporta una vasta gamma di funzionalità JAX-RS, tra cui annotazioni, intercettori, filtri e convalida dei dati. Questo permette di sviluppare API RESTful complete e robuste.

In questo caso per creare un web server che espone delle semplici api basta scrivere:

```
@Path("/auth")
public class LoginController {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello from RESTEasy Reactive";
}
```

In particolare quì basta utilizzare l'annotazione `@Path` per identificare il controller REST e ogni metodo all'interno di questa classe viene considerato un endpoint REST e il valore di ritorno sarà quello specificato dall'annotazione `@Produces`.

CDI (Contexts and Dependency Injection)

Quarkus supporta CDI, uno standard per l'iniezione di dipendenza e la gestione del ciclo di vita dei bean. È possibile utilizzare CDI per iniettare dipendenze nelle proprie classi e per creare bean che vengono gestiti automaticamente dal container Quarkus.

I CDI beans in Quarkus sono annotati con @ApplicationScoped, @RequestScoped, o altre annotazioni di scope definite da CDI. Queste annotazioni definiscono il ciclo di vita dei bean e il loro contesto di esecuzione.

È possibile utilizzare l'annotazione @Inject per iniettare dipendenze nelle proprie classi come il corrispettico @Autowire di Spring Boot. L'annotazione @Singleton indica che una classe deve avere un'unica istanza, che verrà creata e gestita dal container Quarkus.

Non sono disponibili invece numerose annotazioni usate in Spring Boot come @Service e @Repository che però possono essere sostituiti mantenendo quansi tutte le funzionalità da @Singleton, che indica una classe che deve avere un'unica istanza, che verrà creata e gestita dal container Quarkus.

Quarkus JDBC

Quarkus fornisce un modello di configurazione unificato per definire le sorgenti dati sia per JDBC che per driver reattivi. Questo approccio semplifica la gestione delle diverse tipologie di database e rende il codice più portabile.

Grazie all'integrazione con Agroal e Vert.x, Quarkus offre un pooling di connessioni ad alte prestazioni e scalabile per i driver JDBC e reattivi. Questo permette di ottimizzare l'utilizzo delle risorse e di migliorare la scalabilità dell'applicazione.

Inoltre offre supporto per un'ampia gamma di driver JDBC, tra cui PostgreSQL, MySQL, Oracle e Microsoft SQL Server. Questo permette di utilizzare Quarkus con la maggior parte dei database relazionali disponibili.

Per configurare il datasource è bastato specificare le seguenti impostazioni nel file application.properties:

```
quarkus.datasource.db-kind=oracle
quarkus.datasource.username=*
quarkus.datasource.password=*
quarkus.datasource.jdbc.url=jdbc:oracle:thin:@DB_URL:ORCL
```

Quarkus Security JPA

Quarkus Security offre un'integrazione con JPA che permette di utilizzare database relazionali per la gestione di utenti, ruoli e privilegi. JPA è uno standard consolidato per l'accesso a database relazionali, offrendo flessibilità e scalabilità per la gestione di dati di autenticazione e autorizzazione. L'integrazione con Quarkus Security rende l'utilizzo di JPA per la sicurezza trasparente e semplice. Non è necessario scrivere codice aggiuntivo per la mappatura tra le entità JPA e i concetti di sicurezza di Quarkus.

Microservizio di Login

Il microservizio di login sfrutta la libreria `quarkus-security-jpa` per:

- **Recuperare le informazioni di login dal database**: La libreria si interfaccia direttamente con il database per recuperare l'utente associato alle credenziali inviate.

- **Autenticazione**: Verifica la corrispondenza tra le credenziali inviate e quelle memorizzate nel database.

La classe dell'utente è state definita come:

```
@Entity
@UserDefinition
@Table(name = "USER_INFO")
public class User_Info extends PanacheEntity {
    @Username String username;
    String full_Name;
    @Roles String role;
    @Password String password;
    Integer new_Pass;
}
```

L'annotazione @Entity è fondamentale in JPA (Java Persistence API). Marca la classe User_Info come un'entità, ovvero un oggetto che verrà mappato in una tabella di un database relazionale.

@UserDefinition annota la classe come classe che rappresenta un utente nel sistema e serve a gestire l'autenticazione degli utenti.

Come in Spring Boot, @Table(name = "User_Info") specifica il nome esplicito della tabella di database a cui l'entità User_info dovrà essere mappata. In questo caso, la tabella si chiama "User_Info.

L'estensione della classe PanacheEntity determina l'utilizzo di Quarkus Panache. Panache offre un approccio semplificato per lavorare con le entità JPA, fornendo metodi utili per query, update e altre operazioni comuni con il database. In questo caso Panache si occupa di gestire automaticamente l'ID degli utenti.

Inoltre aggiungendo `quarkus.http.auth.basic=true` nel file application.yaml, quarkus sarà in grado da estrarre automaticamente le credenziali di autenticazione dall'header delle chiamate API che riceve.

@Username e @Password vengono usati per gestire l'autenticazione con username e password, in particolare la password utilizza l'hashing di bcrypt di default.

Mentre @Role viene recuperato per associare un ruolo all'utente specifico.

Quindi grazie alla libreria scelta, abbiamo potuto gestire l'autenticazione definendo la connessione al database e la classe degli utenti. In particolare, quando una chiamata API contiene credenziali nell'header, queste vengono estratte e confrontate con l'utente presente nel database. Se coincidono, la richiesta viene servita, altrimenti viene scartata.

La libreria offre inoltre il vantaggio di gestire la crittografia delle password in modo sicuro, utilizzando l'algoritmo bcrypt di default. L'alternativa `quarkus-security-jdbc` presentava dei limiti di personalizzazione per il formato delle password, non supportando il formato bcrypt. Per questo motivo, `quarkus-security-jpa` è risultata la scelta più adatta.

Infine, la libreria `quarkus-smallrye-jwt`, impiegata per generare il token JWT, presenta una sintassi più agevole e intuitiva rispetto all'alternativa adoperata in Spring Boot, facilitando così il processo di sviluppo. Una delle sue caratteristiche distintive è la semplificazione della gestione della chiave pubblica. È possibile specificare la posizione del file direttamente nel file application.properties (`smallrye.jwt.sign.key.location=certs/private.pem`), e la libreria lo impiegherà automaticamente.

```
public String generateToken(User_Info user) {
    Instant now = Instant.now();
    String jwt = Jwt.subject(user.getUsername())
    .claim("user",user)
    .groups(user.getRole())
    .expiresAt(now.plus(1, ChronoUnit.HOURS))
    .sign();
    return jwt;
}
```

Microservizio Gateway

Nell'ecosistema Quarkus non esiste una dipendenza specifica per la creazione di un gateway simile a `spring-cloud-starter-gateway` presente in Spring Boot. Per ovviare a questa mancanza, la logica del gateway è stata implementata manualmente, mentre il client `quarkus-rest-client-reactive-jackson` è stato utilizzato per inviare richieste HTTP agli altri microservizi e per ricevere e gestire le risposte:

```
Client client = ClientBuilder.newBuilder().build();
Builder requestBuilder = client.target(targetUrl).request();
```

```
headers.getRequestHeaders().forEach((key, values) ->
    values.forEach(value -> requestBuilder.header(key, value)));

Response response = requestBuilder.post(Entity.json(requestBody));
InputStream responseStream = response.readEntity(InputStream.class);
String responseBody = convertInputStreamToString(responseStream);
return Response.fromResponse(response).entity(responseBody).build();
```

Anche qui è impiegata la libreria `quarkus-smallrye-jwt` per la validazione automatica del token JWT nell'header delle richieste. Questa procedura consente l'estrazione automatica delle informazioni contenute nel token. La specifica posizione del file, che contiene la chiave pubblica necessaria per la validazione dei token JWT, può essere indicata direttamente nel file application.properties (`mp.jwt.verify.publickey.location=certs/public.pem`).

```
@Path("/")
public class GatewayResource {
    @Context
    private HttpHeaders headers;
    @Inject
    JsonWebToken jwt;

    @Path("/api/requests{path: .*}")
    @GET
    @RolesAllowed({ "0", "1" })
    @Produces(MediaType.APPLICATION_JSON)
    public Response routeRequests(@PathParam("path") String path) {
        ...
    }
}
```

L'utilizzo delle suddette librerie consente di recuperare automaticamente gli header e il contenuto del token JWT per ciascuna richiesta. Tale approccio agevola l'applicazione di ulteriori controlli sui dati contenuti nel token. Inoltre, è possibile sfruttare il ruolo dell'utente codificato all'interno del token per regolare l'accesso alle diverse route una volta effettuata l'autenticazione. Nell'esempio fornito, la route è configurata per accettare utenti con diversi livelli di autorizzazione (0=user, 1=admin).

Microservizio di Request

Il microservizio di Request ha subito pochissime modifiche durante il passaggio a Quarkus. La sua logica è rimasta pressoché identica.

In fase di compilazione dell'eseguibile, tuttavia, si sono verificati dei problemi di compatibilità con la libreria esterna `okhttp`. Questi problemi erano dovuti a una versione non aggiornata della libreria.

Fortunatamente, aggiornando okhttp all'ultima versione disponibile (5.0.0-alpha.12), i problemi di compatibilità sono stati risolti.

Tuttavia l'esperienza con la libreria `okhttp` evidenzia un punto chiave: non tutte le librerie esterne utilizzate in un progetto reale potrebbero essere compatibili con GraalVM. Questo può rallentare il passaggio a Quarkus, richiedendo l'aggiornamento o la sostituzione di librerie non compatibili.

3.3 Deployment

Nell'ambito dello sviluppo e della distribuzione di software, il processo di deployment riveste un ruolo fondamentale nell'efficace implementazione di sistemi complessi e scalabili. In questa sezione, esamineremo tre elementi chiave del deployment che hanno giocato un ruolo cruciale nel contesto della mia ricerca: l'uso di immagini Docker, la configurazione di cluster utilizzando Amazon Elastic Kubernetes Service (EKS) e le considerazioni di networking associate.

3.3.1 Immagine Docker

Per ogni versione di ogni microservizio sono state create due immagini Docker distinte:

- **Spring Boot**: Basata sulla JVM, offre un ambiente di runtime compatibile con le applicazioni Spring Boot tradizionali.
- **Quarkus**: Sfrutta GraalVM, promettendo prestazioni e scalabilità superiori.

Spring Boot

Per creare l'immagine Docker dei microservizi Spring Boot ho dovuto scrivere un semplice Dockerfile come questo:

```
FROM amazoncorretto:20

WORKDIR /app

COPY target/login-0.0.1-SNAPSHOT.jar /app/
```

```
EXPOSE 8081

CMD ["java", "-jar", "login-0.0.1-SNAPSHOT.jar"]
```

- **amazoncorretto:20** è un'immagine che contiene un Java Runtime Environment (JRE) ottimizzato per l'esecuzione di applicazioni Java su Amazon Web Services (o in generale sul cloud).
- **WORKDIR /app** Imposta la directory di lavoro all'interno del container come /app. Comandi successivi nel Dockerfile saranno eseguiti nel contesto di questa directory.
- **COPY target/login-0.0.1-SNAPSHOT.jar /app/** Copia il file JAR dell'applicazione Java, login-0.0.1-SNAPSHOT.jar, dalla directory target/ della build host (dove presumibilmente viene compilato il tuo progetto) alla directory /app nel container Docker.
- **EXPOSE 8081** Informa Docker che il container sarà in ascolto sulla porta 8081 per connessioni in entrata. Questo è più per documentazione e aiuta con il networking, ma non apre automaticamente la porta all'esterno.
- **CMD ["java", "-jar", "login-0.0.1-SNAPSHOT.jar"]** Questa è l'istruzione che viene eseguita quando si avvia un container basato su quest'immagine Docker. Significa eseguire il comando Java (java -jar) per avviare l'applicazione contenuta nel JAR chiamato login-0.0.1-SNAPSHOT.jar.

Quarkus

Invece per creare l'immagine Docker dei microservizi Quarkus ho semplicemente dovuto ritoccare uno dei dockerfile che vengono generati automaticamente alla creazione del progetto:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.9
WORKDIR /work/
RUN chown 1001 /work \
    && chmod "g+rwX" /work \
    && chown 1001:root /work
COPY --chown=1001:root target/*-runner /work/application

EXPOSE 8091
USER 1001

ENTRYPOINT ["./application", "-Dquarkus.http.host=0.0.0.0"]
```

- **FROM registry.access.redhat.com/ubi8/ubi-minimal:8.9** L'immagine di base utilizzata è "ubi8/ubi-minimal:8.9" fornita da Red Hat. Si tratta di una versione minimale di Red Hat Universal Base Image (UBI) 8.9, ottimizzata per le dimensioni ridotte.

- **WORKDIR /work/** Imposta /work/ come directory di lavoro all'interno del container. Le istruzioni successive opereranno all'interno di questa directory.

- **RUN chown 1001 /work && chmod "g+rwX" /work && chown 1001:root /work** Comando composto da più parti:

  1. **chown 1001 /work**: Imposta l'utente con ID 1001 come proprietario della directory /work/.
  2. **chmod "g+rwX" /work**: Assegna permessi di lettura, scrittura ed esecuzione al gruppo proprietario della directory /work/.
  3. **chown 1001:root /work**: Imposta nuovamente l'utente 1001 come proprietario e "root" come gruppo associato alla directory /work/.

- **COPY –chown=1001:root target/*-runner /work/application** Copia tutti i file che terminano con "-runner" dalla directory "target" del progetto nella directory /work/application del container. Durante la copia viene impostato l'utente 1001 come proprietario e il gruppo "root".

- **EXPOSE 8091** Indica al Docker runtime che il container in esecuzione ascolterà sulla porta 8091. Ciò è necessario per consentire il traffico in entrata dall'esterno del container.

- **USER 1001** Specifica che tutti i comandi successivi e l'esecuzione dell'applicazione stessa avverranno con l'utente avente ID 1001. Questo viene spesso fatto per motivi di sicurezza, per evitare di eseguire l'applicazione interna come utente root.

- **ENTRYPOINT ["./application", "-Dquarkus.http.host=0.0.0.0"]** Imposta il comando da eseguire all'avvio del container. L'applicazione (presumibilmente chiamata "application") verrà eseguita con l'opzione "-Dquarkus.http.host=0.0.0.0" che istruisce l'applicazione ad ascoltare le connessioni in arrivo su tutte le interfacce di rete (permettendo l'accesso esterno).

Nello specifico, mi sono limitato a impostare la porta corretta (8091), il resto andava bene. Il Dockerfile è più completo, ma fa fondamentalmente la stessa cosa del precedente. La cosa positiva è che rende il processo di deploy più veloce perché non c'è bisogno di scriverlo da zero, il che è utile soprattutto per chi non ha molta esperienza con Docker.

3.3.2 Cluster EKS

Entrambe le versioni dell'applicazione sono state distribuite su due cluster separati di Amazon Elastic Kubernetes Service (EKS). I cluster condividono le seguenti caratteristiche:

- **Gruppo di auto-scaling**: Regola automaticamente il numero di istanze in base al carico di lavoro.
- **Numero di istanze di default**: 1 per cluster.
- **Policy di auto-scaling**: Aumento del carico CPU oltre l'80%.
- **Tipo di istanza**: t3.small.
- **Caratteristiche istanza**: 2 vCPU, 2 GiB di memoria, supporto fino a 11 pod.

Le istanze t3.small sono state selezionate per due motivi principali: economicità e potenza. Dal punto di vista economico, rappresentano una delle opzioni più convenienti attualmente disponibili. Inoltre, offrono una potenza computazionale sufficiente per ospitare i tre pod dei microservizi e gli agenti di monitoraggio necessari.

Il cluster è stato creato usando il seguente template:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: spring-boot-cluster
  region: eu-south-1
  version: "1.28"

vpc:
  subnets:
    public:
      eu-south-1a: { id: subnet-0ae0110925819d888 }
      eu-south-1b: { id: subnet-03fabf5b0957b7c90 }
      eu-south-1c: { id: subnet-00e2079b429a24fd9 }

nodeGroups:
  - name: spring-boot-ng
    instanceType: t3.small
    desiredCapacity: 1
```

Nel contesto della definizione dei parametri per la configurazione del cluster, sono stati determinati vari aspetti cruciali. Inizialmente, è stata delimitata la regione geografica in cui il cluster verrà istanziato, insieme alla specifica denominazione e versione di Kubernetes da utilizzare. Successivamente, sono stati dettagliati il tipo di istanza richiesto e il numero desiderato di istanze, quest'ultimo fissato a 1. Tale scelta è stata motivata dalla volontà di consentire l'incremento delle istanze solamente in risposta a un aumento significativo del traffico. Infine, sono state individuate le subnet corrispondenti a una specifica VPC, già destinata all'hosting del database. Pertanto, al fine di semplificare l'interconnessione con il database esistente, è stata impostata la creazione del cluster all'interno della medesima VPC.

Auto-scaling Groups

Gli Auto-scaling Groups operano all'interno dell'ambiente di AWS, consentendo agli utenti di definire le politiche di scalabilità basate su determinati parametri, come l'utilizzo delle risorse, il carico di lavoro o il tempo di risposta delle applicazioni. Questi gruppi sono composti da un insieme di istanze EC2 (Elastic Compute Cloud) configurate secondo le specifiche dell'utente e gestite in modo centralizzato attraverso il servizio Auto-scaling.

Per creare un Auto-scaling Group, è necessario definire diversi elementi fondamentali, tra cui:

1. **Configurazione delle Istanze**: Specificare le caratteristiche delle istanze EC2 che saranno utilizzate nel gruppo, come tipo di istanza, dimensioni, AMI (Amazon Machine Image), e altre configurazioni di rete e sicurezza.

2. **Scaling policy**: Definire le politiche di scalabilità che determinano quando e come aggiungere o rimuovere istanze nel gruppo in base a metriche specifiche, come il carico CPU, le richieste HTTP, o metriche personalizzate.

3. **Grandezza Iniziale e Massima del Gruppo**: Specificare il numero iniziale di istanze nel gruppo e il numero massimo di istanze consentite per garantire la scalabilità appropriata.

4. **Grandezza Minima del Gruppo**: Definire il numero minimo di istanze che devono essere sempre attive per garantire la disponibilità dell'applicazione.

Nel nostro caso, è stato automaticamente generato un Auto-scaling Group all'atto della creazione di ciascun cluster. È stato necessario apportare delle modifiche alle configurazioni delle istanze affinché, al momento della creazione di una nuova istanza, essa possa accedere ai security groups necessari per stabilire la connessione al database. Inoltre, è stata implementata una policy di scaling che prevede la

creazione di una nuova istanza nel caso in cui la percentuale di utilizzo della CPU dell'istanza attuale superi l'80%. Infine, è stata regolata la dimensione massima del gruppo a 2, così da consentire il funzionamento dell'auto-scaling.

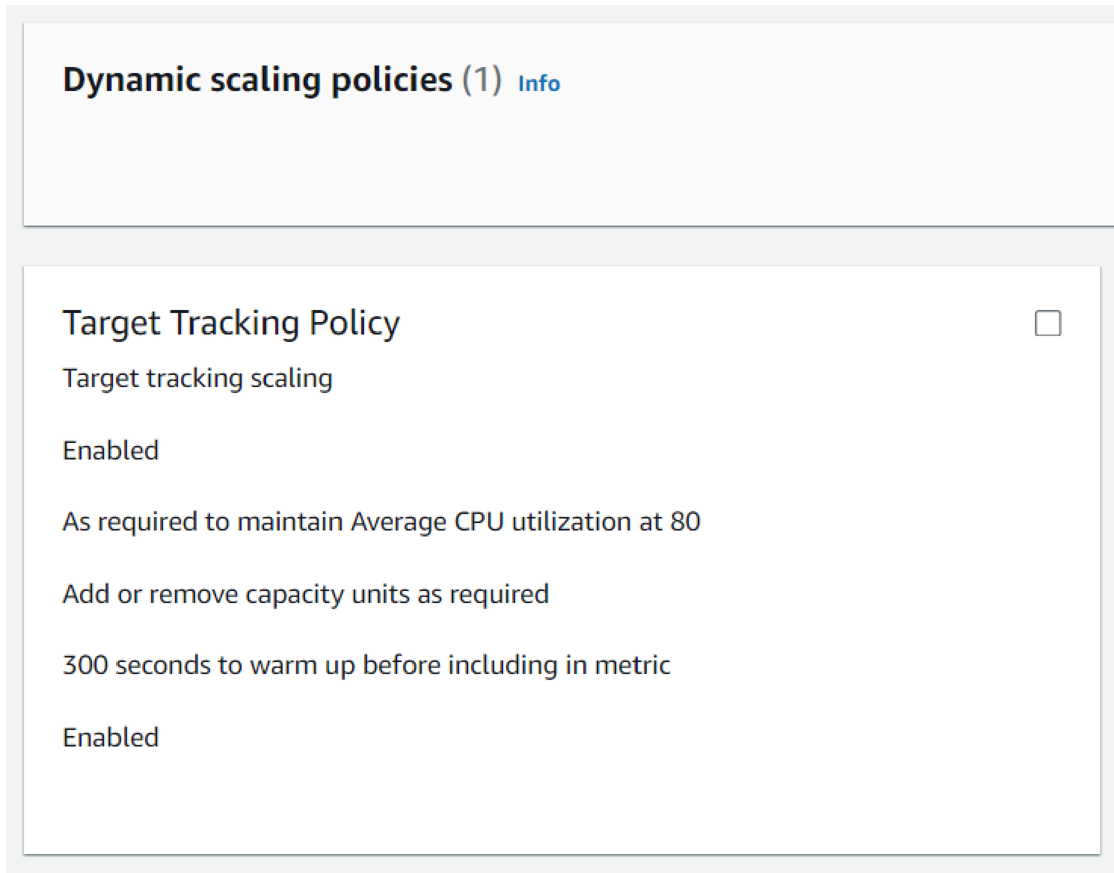

Figura 3.2: Impostazioni di auto-scaling

DaemonSet

[12] I DaemonSet sono un'importante risorsa di gestione dei pod che garantisce che ogni nodo all'interno del cluster esegua una copia specifica di un pod. Questo concetto è particolarmente utile per applicazioni che richiedono di essere eseguite su ogni nodo del cluster per motivi come il monitoraggio, la raccolta di metriche, la gestione dei log o altre attività di sistema.

Quando si crea un DaemonSet in EKS, Kubernetes pianifica automaticamente l'esecuzione di una copia del pod sui nodi esistenti e si assicura che nuovi nodi aggiunti al cluster eseguano anche il pod corrispondente. In caso di ridimensionamento del

cluster o l'aggiunta di nuovi nodi, Kubernetes si occupa automaticamente di avviare e distribuire le istanze dei pod in base alla configurazione definita nel DaemonSet.

Per ogni microservizio è stato implementato un DaemonSet, come con molte altre componenti di aws, questi possono essere creati utilizzando un template di deployment:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: login-service-linux-daemonset
  labels:
    app: login-service-linux-app
spec:
  selector:
    matchLabels:
      app: login-service-linux-app
  template:
    metadata:
      labels:
        app: login-service-linux-app
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                - amd64
                - arm64
              - key: alpha.eksctl.io/nodegroup-name
                operator: In
                values:
                - mng-workload
      containers:
      - name: login-service
        image: s295413/login-service:latest
        ports:
        - name: http
```

```
        containerPort: 8081
      imagePullPolicy: IfNotPresent
    nodeSelector:
      kubernetes.io/os: linux
```

Si tratta di un layout di deployment standard per deamonset, i due campi da attenzionare sono:

- **image**: L'immagine viene recuperata direttamente da dockerhub, un registro di container pubblico nel quale viene prima caricata usando il comando `docker push image_url`
- **containerPort**: Questo campo specifica il numero di porta del container che verrà esposto. Il servizio all'interno del container sarà in ascolto su questa porta per le richieste in ingresso.

3.3.3 Networking

L'accesso ai microservizi è gestito in modo da garantire la sicurezza. In particolare, i microservizi Autenticazione e Chiamate API esterne sono raggiungibili solo attraverso il gateway, come misura di sicurezza. Di conseguenza, tutti i microservizi sono stati collocati all'interno di una Virtual Private Cloud (VPC) privata.

Servizi

[13]Se si utilizza un Deployment per eseguire l'applicazione, esso può creare e distruggere dinamicamente i Pods. Di conseguenza, non è sempre possibile sapere quanti Pods attivi e sani ci siano in un dato momento, né è chiaro quali siano i nomi di tali Pods. I Pods di Kubernetes vengono creati e distrutti per mantenere lo stato desiderato del cluster. Si tratta di risorse effimere e non è ragionevole aspettarsi che un singolo Pod sia affidabile e persistente nel tempo.

I Servizi in Kubernetes sono un'astrazione che ti aiuta ad esporre gruppi di Pods su una rete. Ogni oggetto Servizio definisce un insieme logico di endpoint (generalmente questi endpoint sono Pods) insieme a una politica su come rendere quei Pods accessibili.

Ad esempio, considera un backend di elaborazione di immagini senza stato che viene eseguito con 3 repliche. Queste repliche sono interscambiabili: i frontends non hanno preferenze su quale backend utilizzare. Anche se i Pods effettivi che compongono il backend possono cambiare, i client frontend non dovrebbero essere a conoscenza di ciò, né dovrebbero gestire direttamente l'insieme dei backends.

Kubernetes offre diversi tipi di servizi per soddisfare le esigenze specifiche delle applicazioni:

- **ClusterIP**: Il servizio ClusterIP espone un set di pod all'interno del cluster Kubernetes. È accessibile solo all'interno del cluster tramite un indirizzo IP virtuale.
- **NodePort**: Questo tipo di servizio espone un set di pod su un determinato portale su ciascun nodo del cluster. È utile per consentire l'accesso dall'esterno del cluster.
- **LoadBalancer**: Il servizio LoadBalancer consente di esporre un set di pod tramite un bilanciatore di carico cloud esterno. Kubernetes integra automaticamente il bilanciamento del carico con i servizi supportati dal provider di cloud.
- **ExternalName**: Questo tipo di servizio consente di associare un nome esterno a un servizio di rete esterno, senza esporre un servizio all'interno del cluster. È utile per l'integrazione con servizi esterni al cluster Kubernetes.

ClusterIP

Il servizio ClusterIP associa un indirizzo IP virtuale interno al cluster, noto come ClusterIP, alle risorse del pod di destinazione. Questo indirizzo IP è accessibile solo all'interno del cluster Kubernetes stesso e non è esposto alla rete esterna.

Tra i vantaggi di questo approccio, si evidenziano una maggiore sicurezza, dato che l'accesso è limitato ai pod all'interno del cluster, la facilità di utilizzo senza necessità di configurare manualmente gli indirizzi IP, e la stabilità garantita dai nomi dei servizi che rimangono costanti anche in caso di cambiamenti degli indirizzi IP dei pod.

Nel nostro caso, il servizio clusterIP viene utilizzato per la comunizione tra il gateway e gli altri due pod, di conseguenza i servizi clusterIP vengono creati aggangiandosi a questi ultimi. Ad esempio:

```
apiVersion: v1
kind: Service
metadata:
  name: login-service-clusterip
spec:
  selector:
    app: login-service-linux-app
```

```
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
```

Non avendo specificato il tipo, viene scelto quello di default, ovvero ClusterIP. Viene quindi assegnato un nome e definito il meccanismo con cui il servizio individuerà i pod a cui collegarsi. I pod che hanno un'etichetta (label) con chiave "app" e valore "login-service-linux-app" verranno associati al servizio. Infine, vengono impostate le porte in modo che le richieste che arrivano sulla porta 8081 del servizio vengano instradate alla porta 8081 di uno dei pod selezionati.

Nel nostro caso viene assegnato un servizio clusterIP ai pod di login e request in modo che questi possano essere individuati all'interno del proprio cluster e in particolare il gateway può usare come url il nome del servizio insieme alla porta ad esempio `http://login-service-clusterip:8081` per chiamare il servizio di login.

NodePort

Un servizio NodePort in Kubernetes è un tipo di servizio che espone un'applicazione eseguita su un cluster Kubernetes su un port specifico su ciascun nodo del cluster. Quando un servizio è configurato come NodePort, Kubernetes assegna dinamicamente un numero di porta nell'intervallo 30000-32767 su tutti i nodi del cluster. Questa porta rimane aperta su ogni nodo, consentendo alle applicazioni esterne al cluster di accedere alle risorse ospitate all'interno del cluster attraverso il nodo stesso. Ad esempio:

```
apiVersion: v1
kind: Service
metadata:
  name: gateway-service-nodeport
  labels:
    app: gateway-service-linux-app
spec:
  type: NodePort
  selector:
    app: gateway-service-linux-app
  ports:
    - name: web
      port: 8080
      nodePort: 30010
```

Qui viene specificato esplicitamente il tipo NodePort e i pod che hanno un'etichetta (label) con chiave "app" e valore "gateway-service-linux-app" verranno associati al servizio. Infine, vengono impostate le porte in modo che le richieste che arrivano sulla porta 30010 del servizio vengano instradate alla porta 8080 di uno dei pod selezionati.

Ciò consente di accedere al microservizio di Gateway dall'esterno del cluster usando l'indirizzo IP del nodo + la porta 30010. Tuttavia, questo approccio è ancora poco scalabile. Se si aggiunge un altro nodo al cluster, questo avrà un diverso indirizzo IP e quindi non sarà raggiungibile. Per risolvere questo problema utilizziamo un load balancer.

Bilanciamento del Carico Elastico (ELB)

Il servizio di Bilanciamento del Carico Elastico (ELB) di Amazon Web Services (AWS) distribuisce automaticamente il traffico in ingresso su più destinazioni, come istanze EC2, contenitori e indirizzi IP, in una o più Zone di Disponibilità.

ELB monitora lo stato di salute delle destinazioni registrate e instrada il traffico solo verso quelle in buona salute.

Il servizio è in grado di adattare le dimensioni del bilanciatore in base alle variazioni del traffico in ingresso nel tempo, garantendo la scalabilità per la maggior parte dei carichi di lavoro.

Tipi di bilanciatori di carico:

- **Bilanciatori di Carico Applicativi (ALB)**: instradano il traffico in base al contenuto delle richieste, come il percorso URL o l'host header.
- **Bilanciatori di Carico di Rete (NLB)**: instradano il traffico in base all'indirizzo IP e alla porta di destinazione.
- **Bilanciatori di Carico Gateway (GWLB)**: instradano il traffico verso i gateway virtuali AWS.
- **Bilanciatori di Carico Classici (CLB)**: offrono una funzionalità simile ai ALB, ma con una minore flessibilità.

Application Load Balancer (ALB)

Un Application Load Balancer (ALB) è un componente fondamentale nell'architettura delle applicazioni distribuite e scalabili. Esso funge da punto di ingresso

per il traffico delle richieste degli utenti e distribuisce equamente tali richieste tra diversi servizi di backend. Questo meccanismo migliora la scalabilità, l'affidabilità e la sicurezza delle applicazioni web. Esso può svolgere le seguenti funzioni:

1. **Routing delle richieste**: Il ALB esamina ogni richiesta in arrivo e determina quale servizio di backend dovrebbe gestirla. Questa decisione può essere basata su una varietà di fattori, tra cui il percorso URL, i valori degli header HTTP, il tipo di richiesta e così via. Ad esempio, se un utente richiede la pagina principale del sito web, la richiesta potrebbe essere indirizzata a un servizio di frontend, mentre se richiede una risorsa API, la richiesta potrebbe essere instradata a un servizio di backend specifico.

2. **Bilanciamento del carico**: Una volta determinato il servizio di backend appropriato, il ALB distribuisce le richieste tra le istanze o i container che forniscono quel servizio. Questo processo è noto come bilanciamento del carico e può essere effettuato in base a diversi algoritmi, come round-robin, least connections, least response time, e così via. L'obiettivo è di distribuire uniformemente il carico di lavoro tra le risorse disponibili, evitando sovraccarichi e garantendo prestazioni ottimali.

3. **Gestione dello stato delle sessioni**: In molte applicazioni web, è necessario mantenere lo stato della sessione dell'utente per garantire un'esperienza coerente durante la navigazione. Il ALB supporta questa funzionalità attraverso la gestione delle sessioni. Può instradare le richieste dello stesso utente alla stessa istanza o gruppo di istanze per garantire la coerenza dello stato della sessione. Questo è particolarmente utile quando si gestiscono applicazioni complesse o transazionali.

4. **Monitoraggio della salute delle risorse**: Il ALB monitora costantemente lo stato di salute delle risorse di backend, come istanze EC2 o container, per garantire che siano in grado di gestire le richieste in arrivo. Se una risorsa diventa inaccessibile o non risponde correttamente, il ALB interrompe il routing del traffico verso di essa e lo ridirige a risorse sane. Questo meccanismo di rilevamento delle anomalie aiuta a garantire l'affidabilità e la disponibilità dell'applicazione.

5. **Gestione delle certificazioni SSL/TLS**: Per garantire la sicurezza delle comunicazioni tra gli utenti e l'applicazione, il ALB supporta la gestione delle certificazioni SSL/TLS. È in grado di terminare le connessioni SSL/TLS in modo sicuro e trasferire il traffico in chiaro al backend, semplificando la gestione delle certificazioni e migliorando le prestazioni complessive dell'applicazione.

Nel nostro caso, il target group coincide con l'auto-scaling group, così che quando nuove instanze vengono aggiunte, queste diventano subito visibili dal load balancer,

il routing è configurato in modo che tutte le richieste vengono indirizzate verso il microservizio di gateway(attraverso il servizio NodePort), il bilanciamento del traffico è round robin, non viengno gestite sessioni o certificazioni. Infine il monitoring delle risorse è configurato in modo da chiamare ripetutamente l'api di test fino a quando questa non ritorni un valore positivo, il che significa che tutti i pod sono attivi e quindi la nuova instanza può essere aggiunta a quelle effettivamente chiamabili dal load balancer.

Una volta configurato, il comportamento del sistema sarà il seguente:

1. Il client manda una chiamata API al load balancer, accessibile a un indirizzo del tipo `spring-boot-load-balancer-clusterID.eu-south-1.elb.amazonaws.com`

2. queste vengono mandate al servizio NodePort del gateway che quindi le indirizza al pod di Gateway

3. il gateway gestisce la richiesta e la inoltra al servizio appropriato utilizzando il suo indirizzo ClusterIP

4. la richiesta viene eseguita dal microservizio (login o request) e il valore di ritorno passa risale lo stack di chiamate per arrivare al client

5. qualora le chimate dovessero essere così tante da sovraccaricare il singolo nodo in esecuzione, il sistema di auto-scaling si attiva facendo partire una nuova istanza

6. la nuova istanza farà partire i pod relativi ai 3 microservizi, quando saranno pronti il load balancer potrà mandare parte del traffico anche a questo nodo, che si comporterà in maniera analoga al primo, rendendo il processo di auto-scaling completamente trasparente all'utente finale.

7. quando le chiamate diminuiscono e i due nodi si ritrovano più scarichi, l'auto-scaling procede a terminare una delle due istanze.

Capitolo 4

Test e risultati

Questo capitolo si propone di condurre un'analisi dettagliata delle prestazioni e del monitoraggio dell'applicazione implementata sia su Spring Boot che su Quarkus. Attraverso una serie di test mirati, esamineremo attentamente le capacità di entrambi i framework in condizioni di carico elevato e l'impatto che tali test hanno sulle risorse di sistema.

4.1 test di carico

La sezione seguente si concentra sull'analisi dei test di carico condotti sull'applicazione, sviluppata utilizzando sia Spring Boot che Quarkus. Lo scopo principale di questi test è valutare le prestazioni, la scalabilità e la stabilità di entrambi i framework in condizioni di utilizzo reali.

Attraverso una serie di test mirati, esamineremo come le applicazioni implementate su Spring Boot e Quarkus gestiscano il carico di lavoro simulato, valutando parametri chiave come il tempo di risposta, la capacità di gestire un numero crescente di richieste concorrenti e la stabilità del sistema durante picchi di traffico.

Questo studio mira a fornire una panoramica delle capacità di prestazione dei due framework, consentendo di prendere decisioni informate sulla scelta del framework più adatto alle esigenze specifiche del progetto. Attraverso l'analisi dei risultati ottenuti dai test di carico, sarà possibile comprendere in che misura Spring Boot e Quarkus possono soddisfare le esigenze di scalabilità, efficienza e affidabilità richieste dalle moderne applicazioni distribuite.

Postman

Postman è un potente strumento di sviluppo API che consente di testare, documentare e monitorare API RESTful e GraphQL. Offre un'interfaccia intuitiva e ricca di funzionalità che lo rende ideale per sviluppatori di tutti i livelli di esperienza.

Le funzionalità principali sono:

- **Creazione e invio di richieste API**: Postman consente di creare facilmente richieste HTTP con diversi metodi, header e body. È possibile inviare richieste singole o utilizzare la funzione di collezione per automatizzare l'invio di sequenze di richieste.

- **Analisi delle risposte API**: Postman fornisce una visualizzazione completa della risposta API, inclusi header, body e codice di stato. È possibile utilizzare strumenti integrati per analizzare il contenuto della risposta e formattare i dati JSON e XML.

- **Test e debug API**: Postman offre diverse funzionalità per testare e debuggare API, come la possibilità di impostare breakpoint, visualizzare le variabili di ambiente e convalidare i dati JSON e XML.

- **Documentazione API**: Postman consente di generare automaticamente documentazione API in formato HTML o Markdown. La documentazione include informazioni sulle richieste, le risposte e i modelli di dati dell'API.

- **Monitoraggio API**: Postman offre funzionalità di monitoraggio che consentono di tenere traccia delle prestazioni e dell'affidabilità dell'API. È possibile impostare alert per ricevere notifiche in caso di problemi con l'API.

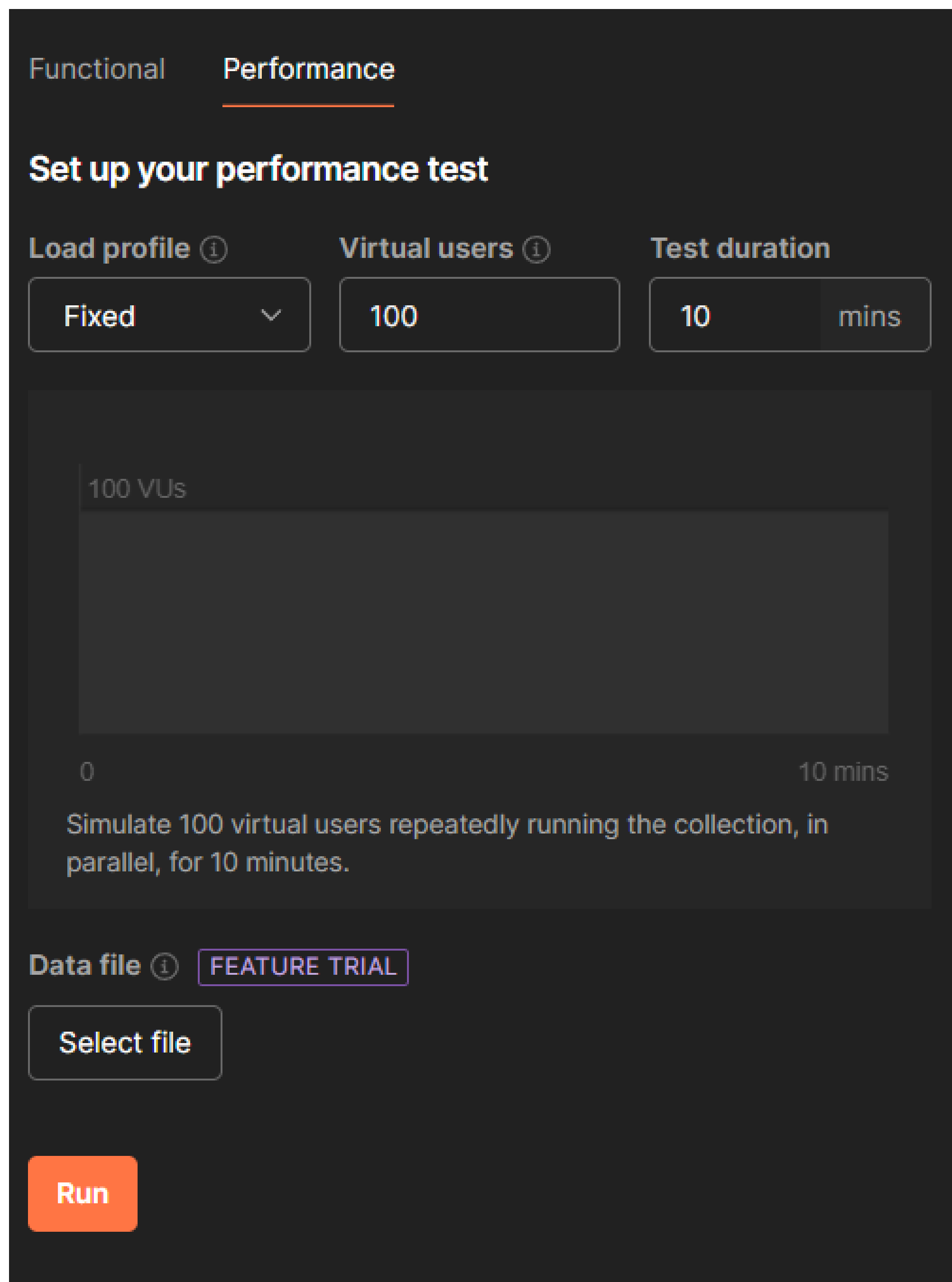

Figura 4.1: Setup test di carico

Esecuzione test

Per testare le performance delle api ho utilizzato postman in modo da simulare un carico massimo di 100 utenti contemporanei che utilizzano le API.

I risulatati per spring boot 4.2 e quarkus 4.3 evidenziano:

- **Migliori prestazioni di Quarkus per la chiamata al servizio esterno /bu**: Quarkus gestisce le richieste al servizio esterno con un leggero vantaggio rispetto a Spring Boot.

- **Prestazioni inferiori di Quarkus per la chiamata al database /token**: Quarkus risulta significativamente più lento rispetto a Spring Boot per le chiamate al database.

Le variazioni nelle performance potrebbero derivare da diverse ragioni, tra cui le impostazioni predefinite per la gestione delle connessioni al database, che potrebbero differire tra Quarkus e Spring Boot, e le caratteristiche di performance delle specifiche librerie utilizzate per l'accesso al database.

Un'analisi approfondita delle specifiche di quarkus-jpa potrebbe permettere un'ottimizzazione delle impostazioni predefinite per la gestione delle connessioni al database, con conseguente miglioramento delle prestazioni di Quarkus.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST token http://lb-workload-1198537406.eu-south-1.elb.amazonaws.com:30010/auth/token	7,327	12.08	122	7,022	9,690	18,867	0
GET bu http://lb-workload-1198537406.eu-south-1.elb.amazonaws.com:30010/api/requests/bu	11,029	18.18	462	4,307	5,203	13,949	0

Figura 4.2: Test di carico su Spring Boot

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST token http://q-lb-workload-1577877039.eu-south-1.elb.amazonaws.com:30020/auth/token	2,521	4.15	661	22,199	27,902	31,639	0
GET bu http://q-lb-workload-1577877039.eu-south-1.elb.amazonaws.com:30020/api/requests/bu	11,106	18.29	494	4,279	5,171	6,517	0

Figura 4.3: Test di carico su Quarkus

4.2 Monitoraggio

Nel contesto dello sviluppo di applicazioni moderne, il monitoraggio delle risorse di sistema e delle prestazioni dell'applicazione riveste un ruolo fondamentale per garantire un funzionamento ottimale e affidabile..

Questa sezione si concentra sull'analisi approfondita dei dati di monitoraggio raccolti durante i test di carico eseguiti sull'applicazione in esame. L'obiettivo primario è valutare come l'aumento del carico influenzi l'utilizzo delle risorse di sistema, tra cui CPU e memoria, sia nel contesto di un'applicazione sviluppata con Spring Boot che con Quarkus.

Attraverso l'analisi dei dati di monitoraggio, saranno identificate le eventuali inefficienze o problematiche di gestione delle risorse da parte dei due framework in scenari di carico elevato. Inoltre, verranno esplorate le differenze nel comportamento delle risorse tra le due piattaforme, consentendo una valutazione comparativa approfondita delle prestazioni e dell'efficienza.

Comprendere come Spring Boot e Quarkus gestiscano le risorse di sistema durante periodi di carico intenso è cruciale per garantire che le applicazioni sviluppate su questi framework possano scalare in modo affidabile e mantenere alte prestazioni anche in condizioni di utilizzo intenso.

Attraverso l'analisi dettagliata dei dati di monitoraggio, questa sezione contribuirà a fornire una visione chiara e approfondita dell'impatto dei test di carico sulle risorse di sistema e delle prestazioni complessive delle applicazioni implementate su Spring Boot e Quarkus.

Amazon Cloudwatch

Amazon CloudWatch è un servizio di monitoraggio e osservabilità offerto da Amazon Web Services (AWS) progettato per fornire visibilità completa sulle risorse e sulle applicazioni in esecuzione nell'ambiente cloud. È uno strumento fondamentale per la gestione delle risorse cloud e per garantire prestazioni affidabili e scalabilità delle applicazioni.

Amazon CloudWatch raccoglie dati di monitoraggio da una varietà di fonti all'interno dell'ambiente AWS, tra cui istanze EC2, servizi di database come Amazon RDS e Amazon DynamoDB, nonché servizi di contenitore come Amazon ECS. Questi dati includono metriche di utilizzo delle risorse, registri di applicazioni e altri dati di monitoraggio generati dalle risorse stesse. Una volta raccolti, i dati vengono elaborati e archiviati in modo sicuro per consentire l'analisi e la generazione di avvisi in tempo reale.

La struttura di Amazon CloudWatch si basa su tre componenti principali:

1. **Metriche**: Le metriche rappresentano dati di monitoraggio di varie risorse AWS. Queste metriche possono includere informazioni come utilizzo della CPU, utilizzo dello storage, numero di richieste HTTP e molto altro ancora. Le metriche possono essere inviate a CloudWatch tramite le API AWS o tramite agenti di monitoraggio installati sui server.

2. **Dashboard**: I dashboard di Amazon CloudWatch consentono agli utenti di visualizzare e monitorare le metriche in modo chiaro e personalizzabile. I dashboard possono essere creati per visualizzare metriche specifiche, consentendo agli utenti di monitorare le prestazioni e l'utilizzo delle risorse in tempo reale.

3. **Allarmi**: Amazon CloudWatch permette agli utenti di impostare allarmi basati su metriche specifiche. Gli allarmi possono essere configurati per avvisare gli utenti tramite notifiche email o tramite integrazioni con altri servizi AWS, come Amazon SNS (Simple Notification Service), quando una metrica supera una soglia specificata.

Amazon CloudWatch viene utilizzato per una vasta gamma di casi d'uso all'interno dell'ambiente AWS. Alcuni esempi includono:

1. **Monitoraggio delle prestazioni delle applicazioni**: CloudWatch può essere utilizzato per monitorare le prestazioni delle applicazioni, identificare eventuali problemi di prestazioni e ottimizzare le risorse per garantire un funzionamento ottimale.

2. **Automazione delle operazioni**: CloudWatch può essere integrato con AWS Lambda per automatizzare azioni basate su eventi di monitoraggio. Ad esempio, è possibile configurare Lambda per scalare automaticamente le risorse in risposta a un aumento del carico di lavoro rilevato da CloudWatch.

3. **Gestione del costo**: Amazon CloudWatch fornisce dati dettagliati sull'utilizzo delle risorse, consentendo agli utenti di monitorare e ottimizzare i costi operativi nell'ambiente cloud. Gli utenti possono identificare risorse sottoutilizzate o sovradimensionate e apportare modifiche di conseguenza per ridurre i costi.

Locust

Locust è un framework open-source per il testing di carico e il benchmarking delle API e dei servizi web. È scritto in Python e offre un'ampia gamma di funzionalità per simulare migliaia di utenti simultanei che interagiscono con un'applicazione o un servizio web. È ampiamente utilizzato nell'industria per valutare le prestazioni di un'applicazione sotto carico simulato e per identificare eventuali punti deboli nell'architettura o nell'implementazione dell'applicazione.

Locust si distingue per diverse caratteristiche chiave che lo rendono una scelta popolare per il benchmarking delle API:

- **Scalabilità Orizzontale**: Locust è progettato per essere altamente scalabile orizzontalmente, consentendo agli utenti di distribuire il carico su più macchine per simulare un numero elevato di utenti concorrenti. Ciò consente di testare le applicazioni in scenari realistici di utilizzo.

- **Scrittura dei Test in Python**: Una delle caratteristiche più apprezzate di Locust è la possibilità di scrivere i test utilizzando Python, un linguaggio di programmazione popolare e flessibile. Questo consente ai team di sviluppo di integrare facilmente i test di carico nell'ambiente di sviluppo esistente e di sfruttare le librerie Python per personalizzare e estendere i test secondo le proprie esigenze.

- **Architettura Basata su Eventi**: Locust utilizza un modello basato su eventi per simulare il comportamento degli utenti. Gli utenti virtuali, sono definiti come classi Python che eseguono compiti specifici durante il test. Gli eventi come la richiesta di una risorsa o il completamento di un task vengono gestiti asincronamente, consentendo di simulare scenari complessi e dinamici.

- **Monitoraggio in Tempo Reale**: Locust fornisce un'interfaccia web in tempo reale che consente di monitorare le prestazioni del sistema durante

l'esecuzione dei test. Questa interfaccia visualizza metriche chiave come il numero di richieste al secondo, il tempo di risposta medio e le statistiche di errore, consentendo di identificare eventuali problemi di prestazioni in modo rapido ed efficace.

Nel contesto della nostra indagine, si è proceduto al caricamento della dashboard Locust all'interno di un apposito cluster EKS. Tale decisione è stata presa al fine di sfruttare appieno le capacità di auto-scaling, consentendo così una gestione più efficiente e dinamica delle richieste simultanee.[14]

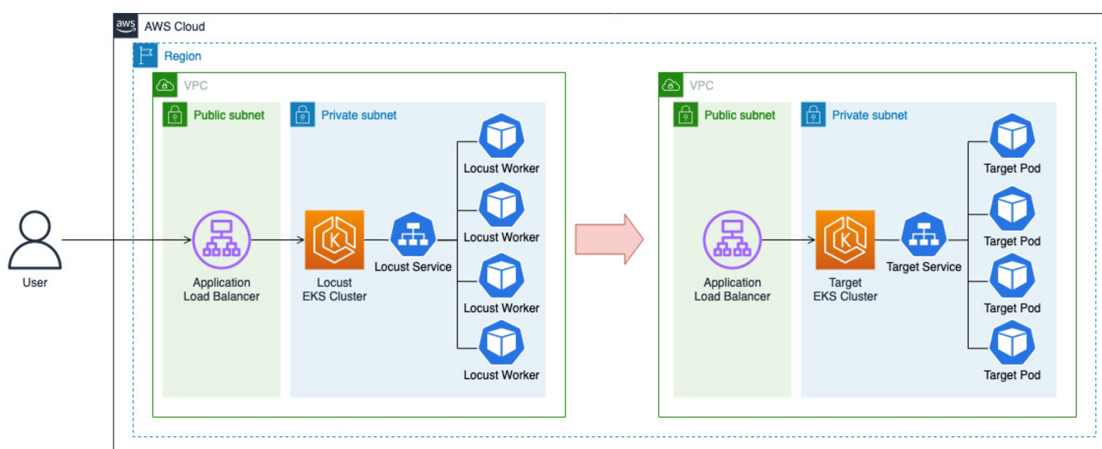

Figura 4.4: Architettura test di carico con locust

Esecuzione test

La maggior parte delle metriche viene recuperato da cloudwatch distribuito all'interno dei due cluster. Per valutare le performance delle API, abbiamo impiegato Locust, uno strumento che ci permette di realizzare una simulazione realistica del ciclo completo di utilizzo dell'applicazione con un elevato numero di utenti contemporaneamente. In particolare, Locust simula il comportamento di un utente reale, inviando richieste a /token e successivamente utilizzando il token ottenuto per accedere a /bu.

La logica di funzionamento delle chiamate viene scritta in python:

```
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(1, 5)
```

```
@task(1)
def get_index(self):
    response1 = self.client.post("/auth/token",
        json={}, auth=("name", "password"))
    token = response1.json().get("token", "")

    headers = {"Authorization": f"Bearer {token}"}
    self.client.get("/api/requests/bu", headers=headers)
```

1. HttpUser è la classe base in Locust che definisce un singolo utente virtuale che metterà sotto carico il sistema.

2. wait_time è una variabile di classe che definisce un tempo di attesa casuale tra 1 e 5 secondi prima che un utente virtuale esegua un'altra attività (task).

3. Il decoratore @task(1) indica che il metodo rappresenta una singola azione dell'utente. Il numero 1 assegnato come argomento serve a definire la probabilità di esecuzione: le azioni con valori più alti hanno maggiore priorità rispetto a quelle con valori meno elevati.

4. La prima azione effettuata dall'utente è mandare una richiesta POST a "/auth/token", passando un payload JSON vuoto e utilizzando l'autenticazione di base con il nome utente "name" e la password "password".

5. La risposta ottenuta al punto precedente contiene un JSON con un token di autenticazione. Questo token è estratto utilizzando response1.json(). Se, per qualche motivo, il token non è presente la variabile token conserva una stringa vuota ("").

6. Infine effettua una richiesta GET all'endpoint "/api/requests/bu". L'header di autorizzazione viene incluso per garantire che la richiesta sia considerata autenticata.

Le metriche più significative per valutare le performance dei due framework sono:

1. **Tempo di avvio** che rappresenta il periodo richiesto per l'avvio di un pod e può essere ottenuto analizzando i log relativi al pod stesso. Questo parametro fornisce importanti indicazioni sulla velocità con cui il servizio viene avviato. Importante in applicazioni scalabili, in cui il pod deve essere disponibile nel più breve tempo possibile per poter servire le richieste velocemente e non creare indisponibilità agli utenti finali

2. **Dimensione dell'immagine** che indica lo spazio occupato dall'immagine nel nodo. Questo dato può risultare significativo per valutare la scalabilità complessiva del sistema e i relativi costi di archiviazione.
3. **Consumo di cpu** offre informazioni sulla quantità di processore utilizzata dal servizio. Questa misurazione può essere particolarmente utile per individuare eventuali congestioni o limiti prestazionali.
4. **Consumo di memoria** del servizio può aiutare a individuare eventuali problemi di gestione della memoria che potrebbero influenzare le prestazioni complessive.
5. **Tempo di auto-scaling** rappresenta il periodo trascorso tra il superamento della soglia di utilizzo della CPU e la disponibilità dei nuovi pod. Questo dato è cruciale per valutare l'efficacia del processo di auto-scaling e la sua capacità di rispondere prontamente alle variazioni di carico.

Tempo di avvio

Tabella 4.1: Misurazioni dei tempi di avvio su un campione di 10 avvii

Microservizio	Spring Boot			Quarkus		
	Media (s)	Min (s)	Max (s)	Media (s)	Min (s)	Max (s)
Login	20.6036	10.060	25.058	0.228	0.294	0.166
Gateway	16.967	8.066	20.829	0.071	0.022	0.186
Request	12.412	10.147	14.301	0.066	0.023	0.158

Non ci stupisce molto, l'eseguibile nativo ha proprio questo come principale vantaggio .

Dimensione dell'immagine

Tabella 4.2: Dimensione dell'immagine

Microservizio	Spring Boot (MB)	Quarkus (MB)
Login	255.04	74.43
Gateway	265.01	58.00
Request	243.42	60.16

Anche questo è uno dei principali vantaggi noti nell'utilizzo di quarkus.

Consumo cpu

Nell'ambito dell'analisi dettagliata del consumo di CPU in un contesto di utilizzo con 50 utenti operanti simultaneamente, è emerso un quadro significativo. In particolare, si è constatato che l'applicazione sviluppata con Spring Boot presenta un tasso di utilizzo della CPU che si mantiene in maniera costante all'87%. Al contrario, l'applicazione basata su Quarkus manifesta un picco di utilizzo della CPU che raggiunge il 93%.

Questo marcato incremento nel consumo di CPU quando si utilizza Quarkus implica che l'applicazione tende a saturare la capacità di elaborazione del nodo in modo più accelerato rispetto a quella sviluppata con Spring Boot. In termini pratici, ciò significa che, con un numero di utenti inferiore rispetto a Spring Boot, Quarkus raggiungerà la saturazione del sistema.

Di conseguenza, per garantire prestazioni ottimali e prevenire possibili sovraccarichi o degrado delle performance, diventa essenziale implementare un sistema di auto-scaling. Una soluzione efficace prevede l'allocazione di un nuovo nodo con una capacità di elaborazione adatta specificamente all'applicazione Quarkus.

In questo scenario, mentre un singolo nodo potrebbe essere sufficiente per gestire l'applicazione sviluppata con Spring Boot, l'applicazione Quarkus richiederebbe un nuovo nodo anche per un numero di utenti inferiore. Questa strategia di allocazione consentirebbe di bilanciare il carico di lavoro, mantenendo una capacità di elaborazione ottimale per entrambe le applicazioni e assicurando così un funzionamento efficiente e stabile del sistema nel suo complesso.

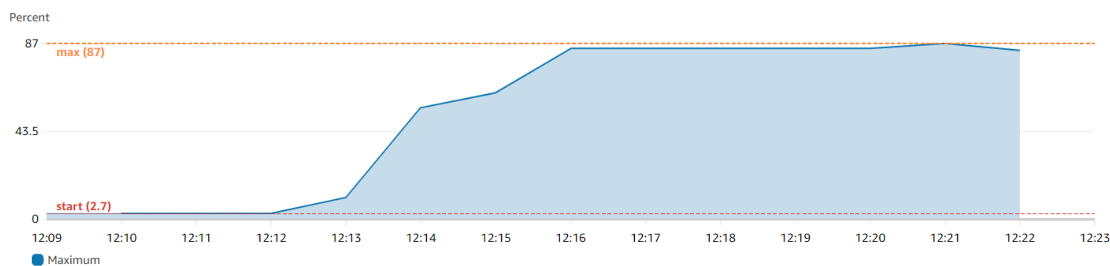

Figura 4.5: Consumo di cpu su spring boot con 50 utenti

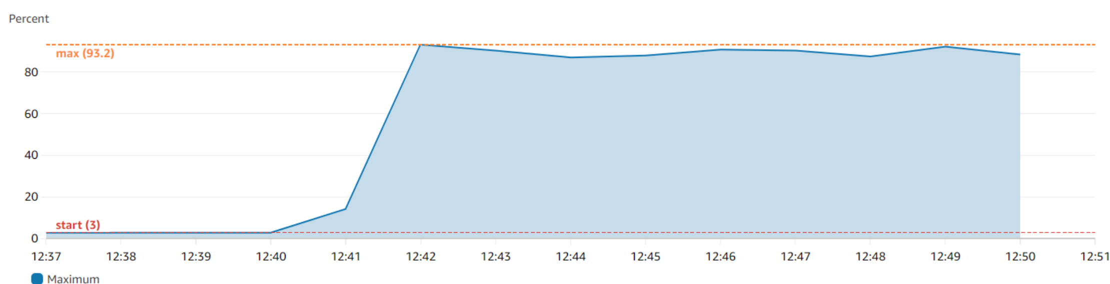

Figura 4.6: Consumo di cpu su quarkus con 50 utenti

Consumo di Memoria

Nell'ambito dell'analisi comparativa del consumo di memoria tra le versioni Spring Boot 4.7 e Quarkus 4.8, è emersa una situazione di particolare rilevanza: Spring Boot ha manifestato un consumo di memoria del 67.2%. Al contrario, Quarkus ha mantenuto un livello di utilizzo della memoria pressoché invariato, presentando variazioni minime tra il 49.5% e il 49.6%. Questo aumento sostanziale del consumo di memoria riscontrato in Spring Boot solleva legittimamente l'ipotesi di una possibile anomalia o di problematiche connesse alla gestione della memoria.

È fondamentale sottolineare che, nonostante l'assenza di specifici compiti o processi che potrebbero sovraccaricare la memoria nell'applicazione nel suo stato corrente, il comportamento di Spring Boot è risultato notevolmente meno efficiente. Questa osservazione diventa ancor più significativa se confrontata con le prestazioni di Quarkus, che, pur svolgendo funzioni analoghe, ha mostrato un utilizzo della memoria sostanzialmente stabile e contenuto.

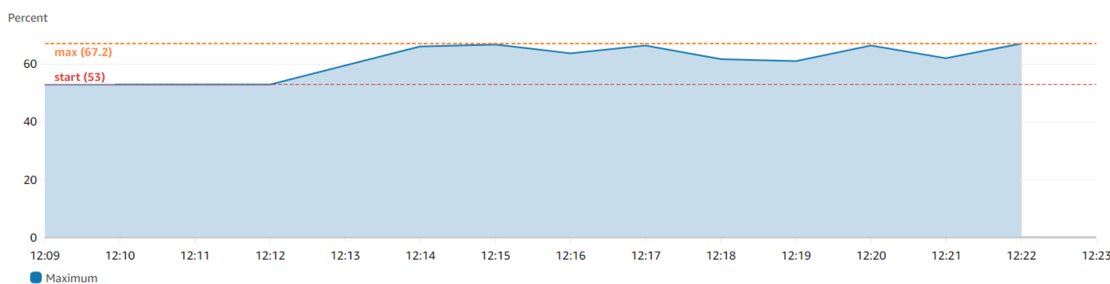

Figura 4.7: Consumo di memoria su spring boot con 50 utenti

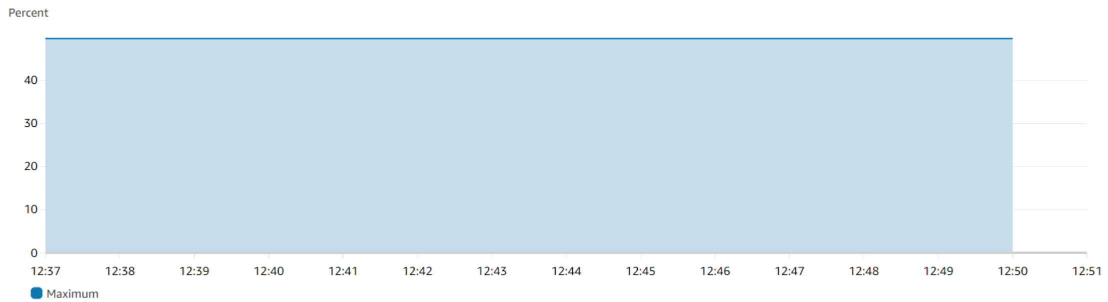

Figura 4.8: Consumo di memoria su quarkus con 50 utenti

Tempo di auto-scaling

Il tempo di auto-scaling è definito come il lasso di tempo che intercorre tra il momento in cui viene superata la soglia di utilizzo della CPU e l'effettiva disponibilità dei nuovi pod nel sistema. Questo parametro riveste un'importanza fondamentale per valutare l'efficacia del processo di auto-scaling e la sua capacità di adattarsi rapidamente alle variazioni del carico di lavoro.

Nel contesto di Spring Boot, è stato osservato che sono necessari 4.18 secondi affinché il secondo pod diventi operativo. D'altra parte, con Quarkus, il tempo impiegato è di 3.59 secondi. Questi risultati sono notevoli e riflettono quanto già osservato in merito ai tempi di avvio in ambiente locale. Tuttavia, è opportuno considerare che questi tempi potrebbero apparire meno significativi se confrontati con i tempi di avvio del nodo AWS, che non scendono mai al di sotto dei 3.50 secondi.

4.3 Confronto Sviluppo

Desidero infine esporre alcune riflessioni dettagliate riguardo al processo di sviluppo all'interno dei due framework presi in esame, basandomi sulla mia esperienza come sviluppatore con una solida competenza in Java, ma con una conoscenza quasi nulla di entrambi i framework.

Entrambi i framework, Spring e Quarkus, offrono agli sviluppatori un utile strumento online noto come Spring Initializer[15] e Quarkus Platform[16], rispettivamente. Questi strumenti sono progettati per facilitare e accelerare il processo di inizio sviluppo per i progetti basati sui suddetti framework.

Spring Initializer consente di generare un'infrastruttura di progetto completa, fornendo un'implementazione di base per un progetto Maven o Gradle. Questo include non solo la struttura di base del progetto, ma anche una serie di dipendenze e configurazioni predefinite specifiche per i progetti basati su Spring Boot. Analogamente, Quarkus Platform offre una funzionalità simile per i progetti basati su Quarkus. Attraverso questo strumento, è possibile creare rapidamente uno scheletro di progetto Quarkus, completo di dipendenze e configurazioni iniziali, pronte per essere implementate e personalizzate secondo le esigenze specifiche del progetto.

Sotto il profilo delle dipendenze, è innegabile che Spring Boot presenti un numero decisamente superiore di dipendenze rispetto a Quarkus. Questa caratteristica implica che, per numerosi scenari d'uso comuni, è probabile trovare una dipendenza già esistente che gestisca la funzionalità desiderata, semplificando notevolmente il processo di sviluppo. Al contrario, Quarkus potrebbe non disporre di una vasta gamma di dipendenze predefinite, costringendo lo sviluppatore a implementare la funzionalità da zero. Questo, a sua volta, richiede una profonda comprensione dell'argomento trattato per garantire risultati ottimali. Personalmente, ho sperimentato questa situazione con la dipendenza Spring Cloud Gateway, che non ha un equivalente diretto in Quarkus, obbligandomi quindi a sviluppare la funzionalità partendo da zero.

Spring Boot registra un utilizzo notevolmente maggiore rispetto a Quarkus, di conseguenza è notevolmente più semplice reperire risorse didattiche e guide che esplorano l'ampio spettro delle dipendenze offerte. Questa abbondanza di materiale consente agli sviluppatori di approfondire esempi di codice e scenari più complessi rispetto a quanto presentato nella documentazione ufficiale. Inoltre, facilita lo studio di casi d'uso che rispecchiano più fedelmente le situazioni reali, offrendo una maggiore comprensione su come combinare e utilizzare congiuntamente diverse dipendenze.

Quarkus, d'altro canto, risulta notevolmente carente in termini di risorse didattiche disponibili. È possibile rinvenire soltanto alcune guide focalizzate sulle funzionalità più elementari e fondamentali. Di conseguenza, diventa essenziale fare riferimento alla documentazione ufficiale per ottenere maggiori informazioni sull'utilizzo delle diverse dipendenze offerte da Quarkus.

La documentazione relativa a Spring Boot si distingue per la sua qualità ed esaustività. Essa fornisce una descrizione dettagliata dei vari parametri, garantendo una comprensione soddisfacente e agevolando gli sviluppatori nella comprensione e nell'utilizzo delle funzionalità offerte dal framework. Inoltre, la documentazione è arricchita da esempi pratici che contribuiscono a chiarire e consolidare la conoscenza

acquisita.

D'altra parte, la documentazione di Quarkus, pur presentando un approccio didattico attraverso guide introduttive, risulta meno completa e approfondita rispetto a quella di Spring Boot. Sebbene le guide introduttive siano efficaci nel fornire una panoramica generale sull'argomento e presentino esempi pertinenti, presentano una criticità significativa. Questa consiste nel non approfondire sufficientemente la descrizione dei parametri e delle funzionalità più avanzate del framework.

Questo limite comporta la necessità per gli sviluppatori di procedere spesso per tentativi o di consultare direttamente il codice sorgente al fine di ottenere una comprensione dettagliata e completa delle potenzialità e delle modalità di utilizzo avanzate di Quarkus. Pertanto, mentre entrambe le documentazioni hanno i propri punti di forza, è evidente che quella di Spring Boot offre una guida più dettagliata e completa, facilitando il processo di apprendimento e di utilizzo del framework.

La principale caratteristica distintiva di Quarkus, dal punto di vista dell'esperienza utente, risiede nella sua capacità di semplificare i casi d'uso più basilari. In tali circostanze, è possibile individuare dipendenze predefinite che eseguono le funzionalità desiderate con una minima necessità di configurazione, richiedendo solamente l'aggiunta di alcuni parametri di setup. Al contrario, Spring Boot spesso impone di redigere manualmente almeno alcune classi al fine di implementare queste medesime funzionalità.

Complessivamente, l'esperienza di sviluppo offerta da Quarkus si presenta come più semplificata e scorrevole. Tuttavia, nonostante questa apparente semplicità, presenta una serie di criticità intrinseche che ultimamente mi hanno portato a preferire lo sviluppo in Spring Boot, basandomi sulla mia personale esperienza.

Capitolo 5

Sviluppi Futuri

Nonostante Quarkus presenti numerosi vantaggi e sia considerato un framework di rilievo nel panorama dello sviluppo software, è imprescindibile riconoscere la presenza di alcune debolezze intrinseche che meritano ulteriore analisi e approfondimento.

In particolare, un aspetto che necessita di un'indagine più dettagliata riguarda il funzionamento delle dipendenze correlate all'interfaccia con il database. È importante identificare e comprendere le cause delle prestazioni inferiori di Quarkus rispetto a Spring Boot in questo specifico contesto. Un'analisi approfondita potrebbe rivelare le potenziali aree di ottimizzazione e miglioramento per il framework.

Inoltre, l'ambito di questa ricerca rivela il comportamento dei vari framework quando sottoposti a stress dal punto di vista della CPU. Tuttavia, sarebbe interessante estendere l'indagine per valutare il comportamento di Quarkus e Spring Boot in condizioni di stress legate alla memoria. Un'analisi comparativa in questo contesto potrebbe fornire informazioni preziose sulle prestazioni e sulle capacità dei due framework in scenari più complessi e sfidanti.

Una delle premesse fondamentali di questa tesi è stata l'analisi del comportamento di un'applicazione progettata per funzionare anche in un'infrastruttura locale, una volta trasferita in un ambiente cloud attraverso l'impiego di due differenti framework. Un ulteriore passo di approfondimento consiste nell'esaminare come l'utilizzo di vari strumenti e servizi offerti dal cloud influenzi le prestazioni di tale applicazione.

Nell'ambito delle tecnologie che potrebbero essere esaminate e implementate per questo specifico caso d'uso, vi è l'opportunità di gestire l'autenticazione su AWS utilizzando Amazon Cognito. Amazon Cognito è un servizio di AWS che

fornisce funzionalità per la gestione dell'identità e dell'autenticazione, permettendo di aggiungere facilmente registrazione e accesso agli utenti nelle applicazioni. Di conseguenza, risulta di interesse esaminare se il microservizio dedicato all'autenticazione possa essere ottimizzato o addirittura rimpiazzato.

Analogamente, sarebbe utile valutare se il microservizio di gateway possa essere sostituito con il servizio AWS API Gateway. AWS API Gateway è un servizio completamente gestito che semplifica la creazione, la pubblicazione, la manutenzione, il monitoraggio e la protezione delle API RESTful e WebSocket.

Inoltre, la logica centrale dell'applicazione, ovvero il microservizio di gestione delle richieste in Java, potrebbe essere ristrutturata e implementata come una funzione Lambda. AWS Lambda è un servizio di elaborazione serverless che consente di eseguire codice senza dover gestire l'infrastruttura sottostante.

È opportuno sottolineare che queste potenziali modifiche comportano l'integrazione profonda della logica dell'applicazione con i servizi offerti da AWS, rendendo quasi impraticabile la transizione verso un altro provider cloud senza la necessità di riscrivere l'intero codice dell'applicazione. Tuttavia, risulta estremamente interessante approfondire i benefici derivanti da questa scelta in termini di semplificazione dello sviluppo, miglioramento delle prestazioni e ottimizzazione dei costi.

Capitolo 6

Conclusione

L'analisi condotta in questa tesi ha fornito una panoramica dettagliata e comparativa tra due tra i più diffusi framework di sviluppo Java per microservizi: Spring Boot e Quarkus. L'obiettivo era di valutare le caratteristiche, le prestazioni e il consumo di risorse di entrambi i framework, al fine di assistere gli sviluppatori nella scelta del più adatto alle specifiche esigenze dei loro progetti.

I risultati ottenuti hanno mostrato chiare differenze tra i due framework in vari aspetti. Quarkus ha dimostrato di essere notevolmente più efficiente in termini di consumo di memoria, presentando un risparmio fino al 36% rispetto a Spring Boot. Questo dato rende Quarkus una scelta interessante per progetti in cui l'ottimizzazione del consumo di risorse è una priorità fondamentale.

Tuttavia, quando si tratta di prestazioni in termini di tempi di risposta e numero di richieste al secondo, la situazione è più sfumata. In alcuni test, le prestazioni di Quarkus e Spring Boot sono risultate molto vicine, mentre in altri Spring Boot ha mostrato un vantaggio significativo, raggiungendo un aumento del 290% nell'architettura specifica testata. Questo suggerisce che la scelta del framework più adatto non dipende solo dalle caratteristiche intrinseche dei framework stessi, ma anche dalle specifiche tecniche, librerie e configurazioni utilizzate nel progetto.

Oltre alle prestazioni e al consumo di risorse, un altro fattore cruciale nella scelta di un framework è la facilità d'uso. Entrambi i framework, Spring Boot e Quarkus, offrono un livello di usabilità elevato. Tuttavia, grazie alla sua ampia diffusione e alla ricca documentazione disponibile, Spring Boot potrebbe presentare una curva di apprendimento più graduale, rendendolo una scelta più accessibile per gli sviluppatori meno esperti.

In conclusione, la decisione sulla scelta del framework più adatto dovrebbe essere basata sulle priorità specifiche del progetto. Se l'ottimizzazione del consumo di memoria è fondamentale, Quarkus emerge come una soluzione valida. Al contrario, se le prestazioni in termini di tempi di risposta e throughput sono prioritari, Spring Boot si presenta come la scelta più appropriata. In ogni caso, è essenziale considerare attentamente le esigenze del progetto, le specifiche tecniche e le preferenze personali per prendere una decisione informata e ben ponderata.

Bibliografia

[1] *What is Graalvm*. URL: https://www.oracle.com/it/java/graalvm/what-is-graalvm/ (cit. a p. 10).

[2] *GraalVM*. URL: https://it.wikipedia.org/wiki/GraalVM (cit. a p. 10).

[3] *What is spring boot*. URL: https://developer.oracle.com/it/learn/technical-articles/what-is-spring-boot-apps-service (cit. a p. 13).

[4] *Vantaggi di Quarkus*. URL: https://www.ionos.it/digitalguide/server/configurazione/cose-quarkus/ (cit. a p. 14).

[5] *Deploying to kubernetes*. URL: https://quarkus.io/guides/deploying-to-kubernetes (cit. a p. 14).

[6] *Kubernetes components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (cit. a p. 17).

[7] *ec2*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (cit. a p. 20).

[8] *EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html (cit. a p. 22).

[9] M. Šipek, D. Muharemagić, B. Mihaljević e A. Radovan. «Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus». In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. 2020, pp. 1746–1751. DOI: 10.23919/MIPRO48935.2020.9245290 (cit. a p. 27).

[10] *JWT*. URL: https://jwt.io/introduction (cit. a p. 29).

[11] *OkHttp*. URL: https://square.github.io/okhttp/ (cit. a p. 40).

[12] *DaemonSet*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset (cit. a p. 52).

[13] *Servizi*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (cit. a p. 54).

[14] *Locust load testing*. URL: https://github.com/aws-samples/Load-testing-your-workload-running-on-Amazon-EKS-with-Locust/tree/main (cit. a p. 67).

[15] *Spring Initializr*. URL: https://start.spring.io/ (cit. a p. 72).

[16] *Quarkus Platform*. URL: https://code.quarkus.io/ (cit. a p. 72).