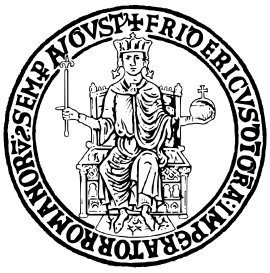
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Studio di nuove Java Virtual Machines per il miglioramento delle performance di applicazioni web

Relatore
Prof. Walter Balzano

Tutor aziendale
Ph.D. Fabio Scippacercola

Candidato
Vincenzo Tramo
N86002592

Anno Accademico 2022–2023

Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Studio di nuove Java Virtual Machines per il miglioramento delle performance di applicazioni web

Relatore
Prof. Walter Balzano

Tutor aziendale
Ph.D. Fabio Scippacercola

Candidato
Vincenzo Tramo
N86002592

Anno Accademico 2022–2023

Sommario

Le specifiche della Java Virtual Machine (JVM) si sono evolute nel corso di circa 20 anni per fornire una piattaforma di prim'ordine per le applicazioni aziendali che eseguono su server che dispongono di tanta memoria e potenza di calcolo. A differenza di oggi, in passato il modello preferito per realizzare applicazioni ad alta disponibilità era quello monolitico. Questo modello spinge la progettazione a massimizzare le prestazioni al costo di un'avvio veloce e del consumo di memoria. Se l'applicazione è destinata a rimanere in esecuzione a lungo termine, l'aumento del tempo di avvio di quest'ultima al costo di una migliore efficienza del runtime è un compromesso accettabile. Tuttavia, con l'avvento delle soluzioni basate sul cloud, sono sorte esigenze di prestazioni e di memoria diverse e tale compromesso non risulta essere più adeguato. L'esecuzione di applicazioni web nel cloud scritte in linguaggi di programmazione basati sulla JVM, in particolare in orchestratori di container come Kubernetes, comporta una serie di complicazioni. I servizi devono essere avviati rapidamente per offrire una scalabilità elevata e, dato che i costi del cloud spesso si basano sul consumo di risorse, si vuole che i servizi abbiano un ingombro di memoria ridotto. In questa tesi verranno discusse delle JVM alternative a quelle tradizionali come Eclipse OpenJ9 e, in particolare, GraalVM che cercano di adattare le applicazioni Java al mondo cloud. Verranno valutati i costi, sia in termini di tempo che di risorse, e le difficoltà incontrate nell'applicare queste tecnologie ad un'applicazione web in miniatura simile a quelle reali enterprise large-scale. Inoltre, in questa tesi verrà fatta un'analisi comparativa delle singole soluzioni per valutarne le prestazioni e la maturità produttiva eseguendo l'applicativo su differenti ambienti di esecuzione messi a disposizione da OpenJ9 e GraalVM.

Indice

Introduzione

L'avvento del cloud computing e le sue nuove esigenze

L'avvento del cloud computing ha completamente rivoluzionato il modo con cui le applicazioni web vengono oggigiorno costruite. Molte aziende IT utilizzano il cloud, che offre tecnologie sempre più innovative, affidabili e sicure, permettendo loro di rimanere al passo con un mercato competitivo e dinamico. Tuttavia, in tutto il settore, le aziende stanno cercando di contenere i costi del cloud, riducendo la capacità di carico delle istanze eseguite in esso. In particolare nel mondo Java, gli sviluppatori cercano di ottimizzare l'utilizzo delle risorse del server distribuendo i carichi di lavoro su istanze sempre più piccole. L'adozione dello scaling orizzontale (sarebbe a dire aumentare e diminuire le istanze di un servizio in base al traffico in entrata) richiede che i servizi Java devono essere in grado di avviarsi rapidamente, poiché devono essere in grado di supportare il rapido aumento del numero di istanze richiesto dal picco di traffico. Ma alcune caratteristiche antiquate della JVM rendono difficile l'utilizzo efficace delle risorse sulle istanze cloud e non sono adatte a questo tipo di situazioni. Le JVM tradizionali sono state originariamente costruite sulla base di alcuni presupposti di progettazione che le rendono incompatibili con il cloud computing, i microservizi e gli orchestratori di container come Kubernetes. Le JVM tradizionali sono state progettate per eseguire applicazioni aziendali su server on-premise, dotati di molta memoria e tanta potenza di calcolo, e sono ottimizzate per massimizzare le prestazioni a lungo termine a scapito dell'avvio veloce e del consumo di memoria. Nel modello di servizio di cloud computing Function-as-a-Service (FaaS) [1], piccole funzioni stateless ed event-driven vengono invocate numerose volte in parallelo. Per ciascuna di queste invocazioni di funzioni, è necessario lanciare un nuovo container e un nuovo runtime, con conseguente latenza significativa. In questo tipo di modello si preferisce uno scaling orizzontale e il consumo di risorse on-demand. Le applicazioni Java sono tipicamente troppo pesanti e lente per una distribuzione serverless in ambiente cloud. Per questo motivo sono state progettate e realizzate negl'ultimi anni nuovi Java Development Kits (JDK) come Eclipse OpenJ9 e GraalVM che cercano di adattare le applicazioni Java al mondo cloud e hanno suscitato un forte interesse da parte delle aziende che adottano tecnologie basate sul cloud verso queste JVM moderne. L'attività di tirocinio è stata fatta presso Fervento srl, una start-up innovativa che utilizza le ultime tecnologie dell'Ingegneria del Software e dell'Informatica per creare sistemi ICT business-critical. Anticipando l'evoluzione del mercato, Fervento ha voluto indagare nell'attività di tirocinio

sulla maturità di questi strumenti e sulle attività che una impresa deve fare per convertire i suoi software asset alle nuove JVM. In questa tesi verranno valutati gli impatti e i benefici di queste nuove tecnologie emergenti, in particolare verrà valutata una caratteristica di GraalVM per compilare un'intera applicazione Java *ahead-of-time* costruendo una *native image* dell'applicazione. Le conoscenze acquisite dallo studio di questi strumenti innovativi durante l'attività di tirocinio verranno applicate su un'applicazione web [2] realizzata attuando il processo di sviluppo Fervento, un processo aziendale reale per lo sviluppo di applicazioni web in Java con Spring Boot e, infine, verranno valutati oggettivamente i costi e i benefici di tale attività.

Struttura dell'elaborato

Nel Capitolo 1 verranno introdotti i termini e gli argomenti cruciali per comprendere i riferimenti trattati nei capitoli successivi. Verrà spiegato cosa sia una Web API e il concetto di REST, verranno presentati Spring Boot, Maven e OpenAPI come strumenti e framework utilizzati. Inoltre verranno introdotte anche tecnologie di virtualizzazione come Docker e Docker Compose.

Nel Capitolo 2 verrà presentato il percorso formativo intrapreso durante il tirocinio, focalizzato sull'utilizzo delle tecnologie illustrate nel capitolo precedente per la creazione di un Toy System: un'applicazione web realizzata seguendo una procedura aziendale specifica, il processo di sviluppo Fervento.

Nel Capitolo 3 verranno presentate le JVM moderne pensate per il cloud come Eclipse OpenJ9 e GraalVM. Verranno mostrate le difficoltà e i costi industriali risultati necessari nell'applicare queste tecnologie al Toy System. Infine, verranno descritti i test di performance progettati ed eseguiti sulle varie versioni del Toy System in esecuzione su diversi ambienti di runtime e verranno valutati i risultati ottenuti con l'obiettivo di capire i benefici che possono essere raggiunti utilizzando tali tecnologie.

–1–
Background

Contenuti: **1.1 Che cos'è una Web API.** 1.1.1 Lo stile architetturale REST – 1.1.2 REST API. **1.2 Spring Boot.** 1.2.1 Maven. **1.3 OpenAPI.** 1.3.1 Struttura di un documento OpenAPI. **1.4 La virtualizzazione e i container.** 1.4.1 Docker – 1.4.2 Docker Compose.

In questo capitolo verranno indicati i termini chiavi e i concetti necessari per capire i riferimenti che si trovano nei capitoli successivi. Nella sezione 1.1 verrà spiegato che cos'è una Web API e lo stile architetturale REST. Nella sezione 1.2 verrà introdotto il framework Spring Boot e lo strumento di build Maven. Nella sezione 1.3 verrà introdotto un formato di descrizione delle API sempre più utilizzato: OpenAPI. Infine, nella sezione 1.4 verranno introdotte le moderne tecnologie di virtualizzazione come Docker e Docker Compose.

1.1 Che cos'è una Web API

Il termine API, acronimo di *Application Programming Interface* (in italiano interfaccia di programmazione delle applicazioni), indica un insieme di definizioni e protocolli per la creazione e l'integrazione di applicazioni software. Una web API è un'API sul web a cui si può accedere tramite il protocollo HTTP.

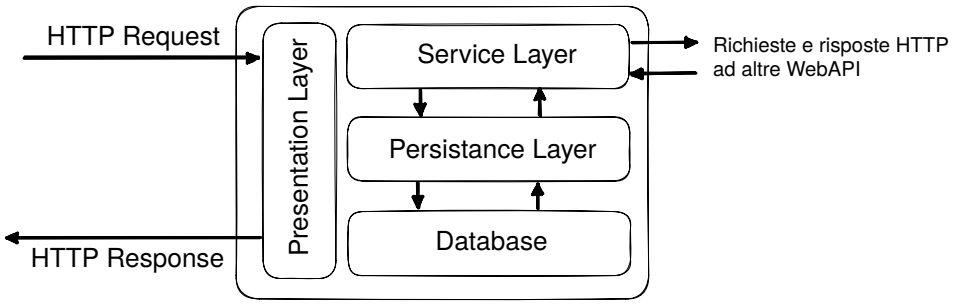


Figura 1.1: Organizzazione di una web API secondo un'architettura a livelli

La Figura 1.1 ritrae come potrebbe essere organizzata ad alto livello una web API. Avendo questa rappresentazione in mente, possiamo vedere che una web API funziona ricevendo richieste HTTP da uno o più client HTTP che attivano l'esecuzione di diversi livelli all'interno dell'API. Il *presentation layer* (in italiano livello di presentazione) riceve le richieste HTTP e le traduce in una forma che possa essere letta da altri livelli. Successivamente, il *service layer* (in italiano livello di servizio) applica la logica di business ed interagisce con il *persistance layer* (in italiano livello di persistenza) che a sua volta interagisce con il database. Ogni livello deve rispondere alle richieste ricevute dal livello superiore fino a raggiungere il presentation layer che invierà la risposta sotto forma di risposta HTTP al client HTTP. Ognuno di questi livelli può essere costruito in molti modi. Le web API possono essere progettate utilizzando una serie di approcci diversi come lo stile architetturale REST, GraphQL oppure SOAP. Lo stile architetturale REST è quello più largamente diffuso ed è stato l'approccio utilizzato per progettare e sviluppare la web API durante l'attività di tirocinio.

1.1.1 Lo stile architetturale REST

Lo stile architetturale REST (*REpresentational State Transfer*) fu introdotto da Roy Fielding nella sua tesi di dottorato [3]. REST è un insieme di vincoli che, se rispettati, inducono ad un insieme di proprietà, la maggior parte delle quali risultano essere vantaggiose per applicazioni decentralizzate e basate sulla rete. Tale stile architetturale era rivolto per una particolare classe di applicazioni software (sistemi ipermediali distribuiti) per il consumo automatico di applicazioni. Nella pratica, i principi REST si sono evoluti in tante derivazioni [4] e alla base sopravvivono le nozioni di protocollo di trasporto, verbi e codici di errore HTTP mappati su concetti applicativi e rappresentazioni tipicamente in JSON. Lo scopo dello stile architetturale REST è facilitare la creazione di sistemi distribuiti efficienti, scalabili e affidabili. Per ottenere tutto questo un'architettura software deve essere conforme ai sei seguenti vincoli per essere tecnicamente considerata RESTful:

- *Uniform Interface*
- *Stateless*
- *Cacheable*
- *Client-Server*
- *Layered System*
- *Code on Demand* (opzionale)

Uniform Interface

Il vincolo *uniforme interface* definisce l'interfaccia tra i client e i server, semplificando e disaccoppiando l'architettura permettendo ad ogni parte di evolvere indipendentemente. Tale vincolo si basa su quattro principi:

- *Resource-Based* (basato sulle risorse): le risorse individuali possono essere identificate nelle richieste utilizzando gli URI (*Uniform Resource Identifier*) come identificatori di risorsa. La rappresentazione della risorsa che viene restituita al client può essere, ad esempio, in formato XML oppure JSON ed è concettualmente separata dalla risorsa in sé;
- *Manipulation of Resources Through Representations* (manipolazione delle risorse attraverso le rappresentazioni): le rappresentazioni delle risorse restituite al client devono contenere abbastanza informazioni per modificare o eliminare la risorsa sul server;
- *Self-descriptive Messages* (messaggi autodescrittivi): ogni messaggio deve contenere abbastanza informazioni per capire come processarlo;
- *Hypermedia as the Engine of Application State* (HATEOAS - ipermedia come motore dello stato dell'applicazione): i client inviano il proprio stato attraverso il contenuto del body, i parametri di query, gli headers di richiesta e il nome della risorsa. I servizi inviano al client lo stato attraverso il contenuto del body, codici di risposta e headers di risposta.

Stateless

Il vincolo *stateless* indica che ogni richiesta che viene inviata da un client al server deve contenere tutte le informazioni necessarie per comprendere tale richiesta senza l'aiuto di nessuna informazione salvata lato server. Le informazioni relative allo stato della sessione devono essere interamente mantenute lato client. Tale vincolo consente una maggiore scalabilità, poiché il server non deve mantenere, aggiornare o comunicare lo stato della sessione.

Cacheable

Il vincolo *cacheable* indica che i dati all'interno di una risposta ad una richiesta debbano essere implicitamente o esplicitamente etichettati come cacheable o non-cacheable, cioè se il client può memorizzare i dati contenuti nella risposta nella propria cache. Se i dati in una risposta possono essere memorizzati nella cache, il client ha il diritto di memorizzare e riutilizzare i dati della risposta per richieste successive equivalenti. Questo vincolo permette di ridurre il numero di interazioni client-server migliorando ulteriormente la scalabilità e le performance.

Client-Server

Il principio che sta alla base del vincolo *client-server* è la separazione delle responsabilità. In un'architettura basata sullo stile REST, i client e i server sono separati in modo tale che ognuno di essi si occupi di compiti specifici. I client si concentrano sull'interfaccia utente e lo stato dell'utente, mentre i server si occupano della memorizzazione dei dati.

Questa separazione rende il codice dei client più portabile, i server più semplici e facilmente scalabili, e permette ai client e ai server di essere sviluppati e sostituiti in modo indipendente.

Layered system

Il vincolo *layered system* impone che il sistema sia organizzato in layers. Ciascun layer fornisce servizi al layer immediatamente sopra e utilizza i servizi offerti dal layer immediatamente sotto. Questo vincolo limita il comportamento di ogni layer garantendo che ciascuno di essi non possa vedere oltre il livello immediato con cui interagisce.

Code on demand

Il vincolo *code on demand* consente ad un server di estendere le funzionalità di un client trasferendo della logica che il client può eseguire. Tale vincolo è l'unico opzionale.

1.1.2 REST API

Un paradigma di progettazione di un'API è un insieme di principi e linee guida che vengono seguiti per progettare e sviluppare un'API. Ogni paradigma ha le sue specifiche e caratteristiche che influiscono su come l'API gestisce le richieste e le risposte, nonché sulla semplicità e sulla flessibilità dell'utilizzo dell'API stessa. Una REST API è un paradigma di progettazione dell'API basato sullo stile architetturale REST presentato nella sezione precedente. Nonostante un sistema RESTful non limiti la comunicazione ad un particolare tipo di protocollo, l'uso di HTTP [5] è una scelta comune perché presenta molte caratteristiche di un sistema RESTful, ma non è un requisito. Un sistema RESTful può essere implementato utilizzando qualsiasi protocollo che supporti le caratteristiche RESTful come l'uso di metodi standard, URI per identificare le risorse e codici di stato per indicare lo stato della risposta. Una progettazione di un'API secondo il paradigma REST è centrata sulle risorse. Le risorse sono il fulcro intorno al quale ruota tutto il sistema ed esse sono identificate univocamente utilizzando gli URL (*Uniform Resource Locator*). Un URL è una sequenza compatta di caratteri che identifica una risorsa su una rete di computer [6]. Una risorsa è rappresentata utilizzando formati standard come JSON e XML. Su ogni risorsa è possibile eseguire operazioni di lettura, modifica, creazione o eliminazione utilizzando i metodi HTTP standard come GET, PUT, POST o DELETE. In generale, un metodo HTTP indica l'azione che un consumatore di una REST API vuole fare su una determinata risorsa. Una REST API utilizza i codici di stato HTTP per indicare se una particolare richiesta HTTP è andata a buon fine o meno, come ad esempio 200 OK per richiesta riuscita, 404 Not Found per risorsa non trovata. In sintesi, le REST API sono una scelta efficace per la gestione delle risorse in quanto utilizzano metodi standard del protocollo HTTP e codici di stato per comunicare e garantire l'integrità delle richieste e delle risposte. Utilizzando metadati e meccanismi di autenticazione, queste API forniscono un alto livello di sicurezza e controllo sull'accesso alle risorse. Inoltre, sono facili da implementare e utilizzare per gli sviluppatori, rendendo queste API una scelta scalabile e affidabile per molteplici progetti.

1.2 Spring Boot

Spring Boot è il framework Java più popolare e usato per lo sviluppo di applicazioni web ed è un'estensione del framework Spring progettato per semplificare lo sviluppo di applicazioni costruite con quest'ultimo. Infatti, la sua popolarità è attribuita alla sua capacità di creare applicazioni Spring pronte all'uso senza preoccuparsi troppo delle configurazioni necessarie per rendere l'applicazione eseguibile. Spring Boot permette agli sviluppatori di concentrarsi sulla logica di business piuttosto che sulle configurazioni.

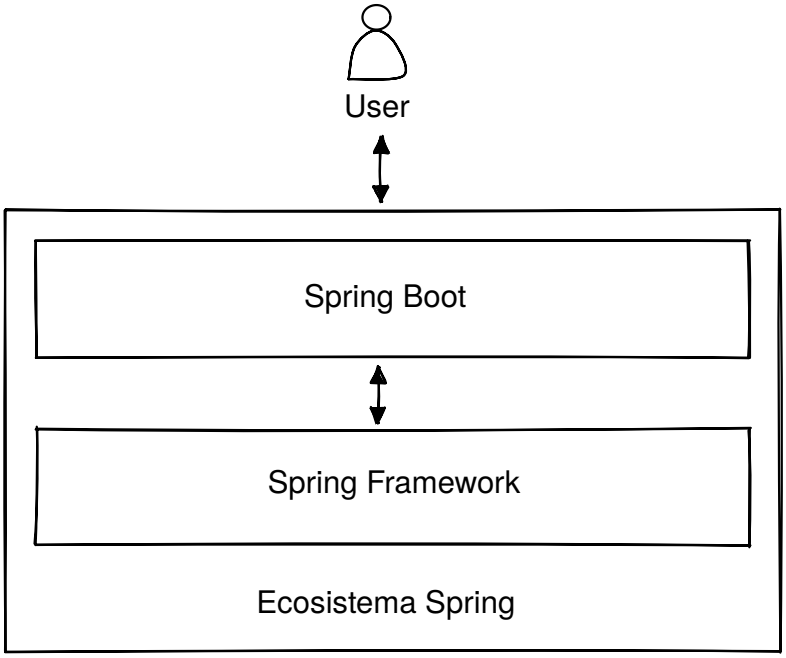


Figura 1.2: Come uno sviluppatore vede Spring Boot

La Figura 1.2 ritrae Spring Boot come un layer che si interpone tra lo sviluppatore e il framework Spring. Il layer Spring Boot ha il compito di configurare automaticamente diversi componenti del framework Spring favorendo così il paradigma *convention over configuration* [7], il che significa che molte configurazioni vengono impostate automaticamente se non vengono specificate esplicitamente dallo sviluppatore. Spring Boot offre anche una serie di strumenti e funzionalità di facile utilizzo come una console di amministrazione, una dashboard di monitoraggio e una gestione automatica delle dipendenze, che aiutano gli sviluppatori a creare e gestire le applicazioni in modo efficiente. Spring Boot è compatibile con una vasta gamma di ambienti, dalle grandi applicazioni monolitiche a quelle distribuite su più server, e supporta un'ampia varietà di tecnologie come REST, WebSockets, data access e sicurezza. In sintesi, Spring Boot è un framework di sviluppo popolare che offre una configurazione semplice, una vasta gamma di strumenti e funzionalità e la compatibilità con una varietà di ambienti e tecnologie, che lo rendono una scelta ideale per gli sviluppatori per creare applicazioni web in modo efficiente. Un

progetto Spring Boot può essere creato facilmente con strumenti di build come Maven e Gradle. Gli strumenti di build permettono di automatizzare operazioni ripetitive, gestire facilmente le dipendenze dell'applicazione e garantiscono consistenza tra gli sviluppatori. Uno degli strumenti di build più utilizzati per avviare un progetto Spring Boot è Maven. Spring Boot combinato con Maven fornisce un potente framework per lo sviluppo di applicazioni Java.

1.2.1 Maven

Uno degli strumenti di build più usati nel mondo Java è sicuramente Maven. Maven è un progetto della Apache Software Foundation ed è open source. Maven ha un ciclo di vita di build predefinito che comprende i passaggi più comuni, noti come *phases* (fasi), che ci si aspetta in una build. L'insieme delle fasi possono essere raggruppate nei seguenti passaggi ognuno con un compito ben preciso:

- *Validate*: controlla se le configurazioni del progetto sono corrette;
- *Compile*: compila il codice sorgente;
- *Test*: esegue test di unità;
- *Package*: genera artefatti come ad esempio file JAR;
- *Verify*: esegue test di integrazione;
- *Install*: installa l'artefatto nella repository locale;
- *Deploy*: rende l'artefatto disponibile ad altri, tipicamente per eseguirlo in un'ambiente di Continuous Integration (CI) [8].

Qualsiasi progetto Maven condividerà questo ciclo di vita di build. Nel modello Maven, vari *plugins* assegnano dei *goals* a queste fasi. Un goal è un task concreto, un vero e proprio programma con uno scopo ben preciso. Un plugin Maven è un insieme di goals che possono essere associati a una o più fasi. Maven include anche i cicli di vita *clean* e *site*. Il ciclo di vita *clean* è destinato alla pulizia (ad esempio rimuovere i risultati prodotti da una compilazione), mentre il ciclo di vita *site* è inteso per la generazione di documentazione. Maven permette anche allo sviluppatore di definire il proprio ciclo di vita. Oltre a fornire un ciclo di vita di build standardizzato, Maven utilizza un layout convenzionale (rappresentato in Figura 1.3) come struttura del progetto predefinita.

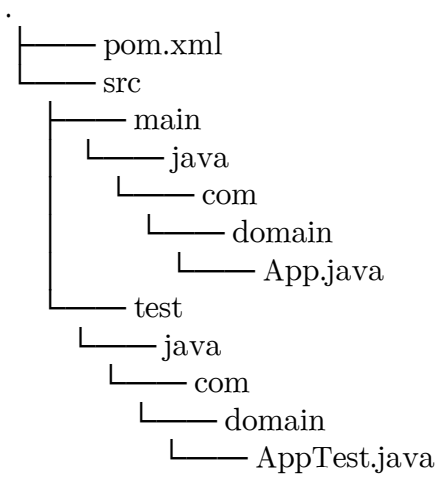


Figura 1.3: Struttura convezionale del progetto seguita da Maven

Come è possibile notare in Figura 1.3, alla radice del progetto troviamo un file `pom.xml`. POM sta per *Project Object Model* ed è un file fondamentale che fornisce informazioni a Maven riguardo al progetto in questione e dettagli di configurazione per compilare il progetto. Quando si esegue un task o un goal, Maven cerca il POM nella directory corrente. Legge il POM, ottiene le informazioni di configurazione necessarie e poi esegue il goal. Il POM contiene valori di default come la directory del codice sorgente, la directory che contiene i test e la directory che contiene gli artefatti prodotti (ad esempio i risultati prodotti dalla compilazione del progetto). Alcune delle configurazioni che possono essere specificate sono le dipendenze, i plugins o i goals da eseguire. Il file `pom.xml` in Figura 1.3 contiene il minimo indispensabile per dare informazioni sul progetto a Maven:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.domain</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>
</project>
```

Tale file dichiara due campi particolarmente importanti: `groupId` e `artifactId`. Questi due campi combinati con una versione formano le cosidette *GAV coordinates* (group, artifact, version) e permettono di identificare univocamente e globalmente uno specifico rilascio dell'artefatto. Le coordinate GAV sono fondamentali perché gli strumenti di build le traducono in veri e propri URL che corrispondono ad una posizione in un repository

Maven. Le dipendenze di un'applicazione possono essere installate manualmente da riga di comando nella propria repository locale, oppure possono essere specificate nel file `pom.xml`. Ad esempio, per aggiungere la libreria JUnit come dipendenza al proprio progetto basta inserire nel file `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Maven cercherà nella repository remota di default *Maven Central* [9] (che è anche la repository di default di altri strumenti di build basati sulla JVM) la libreria JUnit versione `4.13.2`. In sintesi, Maven è uno strumento di gestione delle dipendenze e di build per progetti Java e non solo. Esso rende semplice la gestione delle dipendenze, la gestione dei plugin e la generazione di artefatti per la distribuzione delle applicazioni ed è stato creato per semplificare il processo di sviluppo rendendo più efficiente e organizzato il lavoro degli sviluppatori.

1.3 OpenAPI

Quando si descrive un'interfaccia di programmazione, è importante fornire informazioni dettagliate sui dati e sulle funzionalità che essa offre. Utilizzare un formato di descrizione API strutturato permette di descrivere in modo preciso e dettagliato un'API. Essendo una descrizione standardizzata, essa offre diversi vantaggi. Uno dei principali vantaggi è che un formato di descrizione API strutturato consente una maggiore comprensione dell'API da parte degli sviluppatori. Inoltre, essendo un formato standardizzato, può essere utilizzato da diversi strumenti, come ad esempio quelli di generazione automatica del codice, per generare codice sorgente per l'API in modo automatico. Le specifiche OpenAPI (*OpenAPI Specification* OAS - precedentemente *Swagger Specification*[1]) sono delle specifiche che definiscono un'interfaccia standard indipendente dal linguaggio per le REST API permettendo sia ai computer che agli umani di comprendere le capacità del servizio senza accedere al codice sorgente. Le OAS sono delle specifiche aperte; chiunque può contribuire attraverso il suo repository GitHub [11]. L'insieme degli strumenti forniti da Swagger sfruttano questo standard per generare documentazione, codice sorgente (sia lato client che lato server) e test automatici in molti linguaggi di programmazione, incluso

[1] Questo formato è stato donato all'OpenAPI Initiative [10] nel novembre 2015 e rinominato OpenAPI Specification nel gennaio 2016. Le specifiche sono state originariamente create da Tony Tam e si chiamavano Swagger Specification. Queste specifiche facevano parte di un unico framework chiamato Swagger. Il marchio Swagger esiste ancora e fornisce strumenti API utilizzando le OAS.

Java [12]. L'utilizzo di un formato di descrizione dell'API presenta vantaggi per tutto il ciclo di vita dell'API e in particolare durante la fase di progettazione.

1.3.1 Struttura di un documento OpenAPI

Un documento OpenAPI è una risorsa autonoma o composita che definisce o descrive un'API o elementi di un'API. Un documento che aderisce alle specifiche di OpenAPI può essere rappresentato sia in JSON che in YAML.

Codice 1.1 Un semplice documento OpenAPI

```yaml
openapi: 3.0.1
info:
    title: Shopping API
    version: 1.0
paths:
    /products:
        description: Il catalogo dei prodotti
            get:
                summary: Cerca prodotti
                parameters:
                    - name: free
                        description: free query
                        in: query
                        schema:
                            type: string
                responses:
                    '200':
                        description: products matching free query
                        content:
                            application/json:
                                schema:
                                    type: array
                                    items:
                                        properties:
                                            name:
                                                type: string
                                            price:
                                                type: number
```

Il Codice 1.1 mostra un semplice documento OpenAPI scritto in YAML [13]. Esso fornisce informazioni generali sull'API, come il nome e la versione. Descrive le risorse disponibili (identificate dai loro percorsi) e le operazioni di ogni risorsa identificate dai metodi HTTP, inclusi i loro parametri e risposte.

OpenAPI è uno standard aperto per la descrizione di interfacce di programmazione, in particolare delle REST API. Esso fornisce un modo strutturato per descrivere i dati e le funzionalità di un'API, rendendola facilmente comprensibile per gli sviluppatori. Inoltre, OpenAPI può essere utilizzato con diversi strumenti per generare automaticamente il codice per l'API, il che rende più semplice e veloce lo sviluppo delle applicazioni. OpenAPI

è uno standard in continua evoluzione e supportato da molte comunità di sviluppatori, che lo rendono uno strumento sempre più utilizzato e accettato nel mondo dello sviluppo di API. La sua adozione aiuta a migliorare la qualità, la trasparenza e la facilità d'uso delle API, rendendole più accessibili e comprensibili per tutti gli sviluppatori.

1.4 La virtualizzazione e i container

Nel corso della storia dell'informatica, si è sempre cercato di creare livelli di astrazione tra il software e l'hardware su cui viene eseguito. Il processo che permette di creare un'astrazione dell'hardware è la virtualizzazione. La virtualizzazione è un processo in cui un vero e proprio software viene utilizzato per creare un livello di astrazione sull'hardware del computer e consente di dividere le risorse hardware di un singolo computer in più computer virtuali. Il software usato come livello di astrazione è conosciuto come *hypervisor*. Un hypervisor è un software leggero che permette di creare, gestire ed eseguire più sistemi operativi l'uno accanto all'altro condividendo le stesse risorse hardware. Questi sistemi operativi sono disponibili come macchine virtuali (VMs), ovvero file che riproducono un intero ambiente hardware all'interno del hypervisor. Un hypervisor può essere di tipo 1 o di tipo 2. Un hypervisor di tipo 1 è un hypervisor installato direttamente sull'hardware e funziona come un sistema operativo leggero dedicando la maggior parte delle risorse della macchina alle macchine virtuali che esegue. Un hypervisor di tipo 2 è un hypervisor installato sopra ad un sistema operativo. In ogni caso, gli hypervisors hanno la responsabilità di gestire le risorse dell'hardware sottostante per le macchine virtuali. I container rappresentano un ulteriore passo verso una virtualizzazione sempre più efficiente. Un container è un ambiente isolato con il proprio filesystem indipendente su cui è possibile eseguire un'applicazione. I container sono diventati il principale modo con cui un'applicazione Java viene distribuita. La differenza principale tra un container e una macchina virtuale è che i container condividono tutti lo stesso sistema operativo della macchina host mentre ogni macchina virtuale ha il proprio sistema operativo. Questa differenza è rappresentata schematicamente in Figura 1.4.

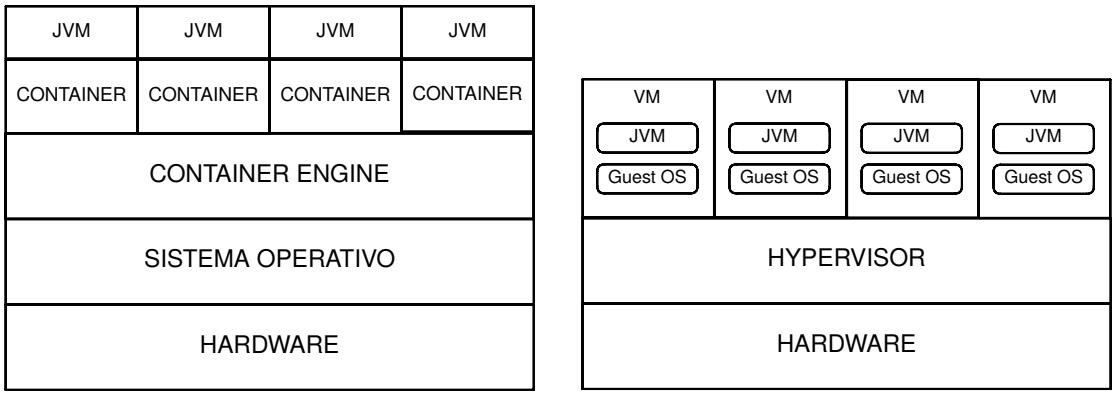


Figura 1.4: Container vs VM

I container sono più leggeri delle macchine virtuali. Inoltre, i container si avviano rapidamente e consumano il necessario, permettendo di avviare molti più container che macchine virtuali sulla stessa macchina hardware. Il crescente utilizzo dei container su diverse piattaforme e provider ha creato la necessità di uno standard comune per garantire la compatibilità tra di essi. Questo ha portato alla creazione dell'OCI (*Open Container Initiative*) [14] che fornisce un insieme di standard aperti e condivisi per i container, permettendo loro di essere facilmente distribuiti e utilizzati su diverse piattaforme. I container consentono di impacchettare l'applicazione con tutte le sue dipendenze e configurazioni in un unico ambiente isolato. Anche se esistono altre tecnologie di containerizzazione e di orchestrazione dei container, *Docker* e *Kubernetes* dominano il mercato dei container e dell'orchestrazione, rispettivamente.

1.4.1 Docker

Docker fornisce strumenti ed una piattaforma per gestire il ciclo di vita dei container [15]. Esso è organizzato secondo un'architettura client-server: un *Docker Client* parla a un *Docker Daemon* (server) il quale si occupa di costruire, avviare e distribuire i container. Un container è creato a partire da una *Docker Image* (immagine Docker). Le immagini sono template read-only che contengono istruzioni per creare un container. Per costruire un'immagine è necessario definire un *Dockerfile*. Un Dockerfile è un file di testo senza estensioni che contiene uno script di istruzioni che Docker utilizza per assemblare un'immagine [16]. L'utilizzo di un Dockerfile permette di creare immagini con tutti gli strumenti e librerie necessarie per compilare ed eseguire l'applicazione. Con solo Docker installato sulla propria macchina, ogni sviluppatore può costruire l'immagine di un'applicazione e avviare un container che esegue l'applicazione in isolamento senza preoccuparsi di scaricare e installare gli strumenti e le librerie necessarie per compilarla. Docker rende più semplice il processo di sviluppo, test, distribuzione ed esecuzione delle applicazioni, poiché separa l'applicazione dall'infrastruttura sottostante, rendendola più portabile e facile da distribuire. L'utilizzo dei Docker container ha un impatto positivo sulla sicurezza, poiché essi forniscono un livello di isolamento tra le diverse parti dell'applicazione, rendendola più sicura e protetta. Tuttavia, per semplificare la configurazione e l'esecuzione di applicazioni multi-container, le funzionalità di Docker sono estese da Docker Compose, un insieme di strumenti che offrono un modo più semplice e ordinato per gestire più container.

1.4.2 Docker Compose

Docker Compose è uno strumento per facilitare la definizione e la condivisione di applicazioni costituite da più container su una singola macchina host [17]. Esso permette di definire lo stack dell'applicazione in un file che viene mantenuto alla radice del progetto. Con Compose, possiamo creare un file YAML per definire i servizi e con un solo comando possiamo eseguire l'intera applicazione oppure interromperla. Essenzialmente, un file Docker Compose descrive lo stato desiderato che l'applicazione deve avere e possiamo

vederlo come un file in cui si piazzano tutte le opzioni da utilizzare quando si eseguono i comandi Docker per eseguire i container. Un esempio di un file Docker Compose è il seguente:

Codice 1.2 Definizione di un file `docker-compose.yml`

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

In sintesi, Docker Compose è uno strumento che fa parte dell'ecosistema Docker e che consente di gestire più container Docker contemporaneamente. Esso fornisce un modo semplice per configurare e gestire più container utilizzando un file di configurazione, invece di utilizzare i comandi Docker individuali. Con Docker Compose gli sviluppatori possono definire le dipendenze tra i container, avviare, fermare e gestire più container contemporaneamente e creare ambienti di sviluppo simili a quelli di produzione.

–2–
Sviluppo Toy System con Processo Fervento

Contenuti: **2.1 Processo di sviluppo Fervento.** 2.1.1 Use Case Polisportiva. **2.2 Design con OpenAPI.** 2.2.1 Documentare una web API con Swagger/OpenAPI 3 – 2.2.2 Generare codice a partire da un documento OpenAPI. **2.3 Implementazione con Spring Boot.** 2.3.1 Design del backend – 2.3.2 Implementazione di un REST endpoint.

In questo capitolo verrà presentato il lavoro fatto durante il tirocinio il quale ha riguardato l'apprendimento delle tecnologie introdotte nel capitolo precedente per acquisire la sensibilità necessaria per la ricerca compiuta nel capitolo finale di questa tesi. Il problema in oggetto descritto in questo capitolo è stato sufficiente a stimolare un sottoinsieme di base di funzionalità presenti nelle web app industriali. L'applicazione web [2] è stata costruita seguendo il processo Fervento. Tale processo aziendale è stato applicato per sistemi dallo scope più ampio ed è stato applicato sotto la visione del tutor aziendale.

2.1 Processo di sviluppo Fervento

Il processo di sviluppo Fervento è un processo aziendale per lo sviluppo di applicazioni web in Java con Spring Boot. Tale processo di sviluppo è fortemente gestito da generatori di codice e utilizza Maven, introdotto nel Capitolo 1 in Sezione 1.2.1, come strumento di assistenza al processo di sviluppo stesso. In particolare, Maven viene utilizzato per avviare nuovi progetti, gestire le dipendenze del progetto e creare immagini Docker dell'applicazione. Il processo di sviluppo Fervento parte dalla definizione delle API utilizzando il formato di descrizione delle interfacce di programmazione OpenAPI, introdotto nel Capitolo 1 in Sezione 1.3. Il processo di sviluppo segue quindi un approccio *Design-First* o *Specification-First* (prima la progettazione) in cui si progetta prima il documento API in maniera dettagliata e poi si procede all'implementazione utilizzando come codice di

base quello generato dagli strumenti di generazione di codice. In generale, il processo di sviluppo Fervento segue questo schema:

1. Definizione dell'API;
2. Autogenerazione del progetto;
3. Implementazione della logica di business.

2.1.1 Use Case Polisportiva

L'applicazione utilizzata come base per lo studio delle differenti JVM e strumenti correlati è un'applicazione web sviluppata in Java con Spring Boot e offre REST API. Una REST API è un'API conforme ai vincoli dello stile architetturale REST descritti nel Capitolo 1 in Sezione 1.1.1, che consente l'interazione con servizi web RESTful. L'applicazione è stata sviluppata seguendo il processo di sviluppo Fervento. L'API in questione fornisce servizi per gestire una o più polisportive esponendo diversi endpoint per manipolare e ottenere risorse. In particolare, il diagramma dei casi d'uso in Figura 2.1 descrive le funzioni che l'applicazione espone tramite API:

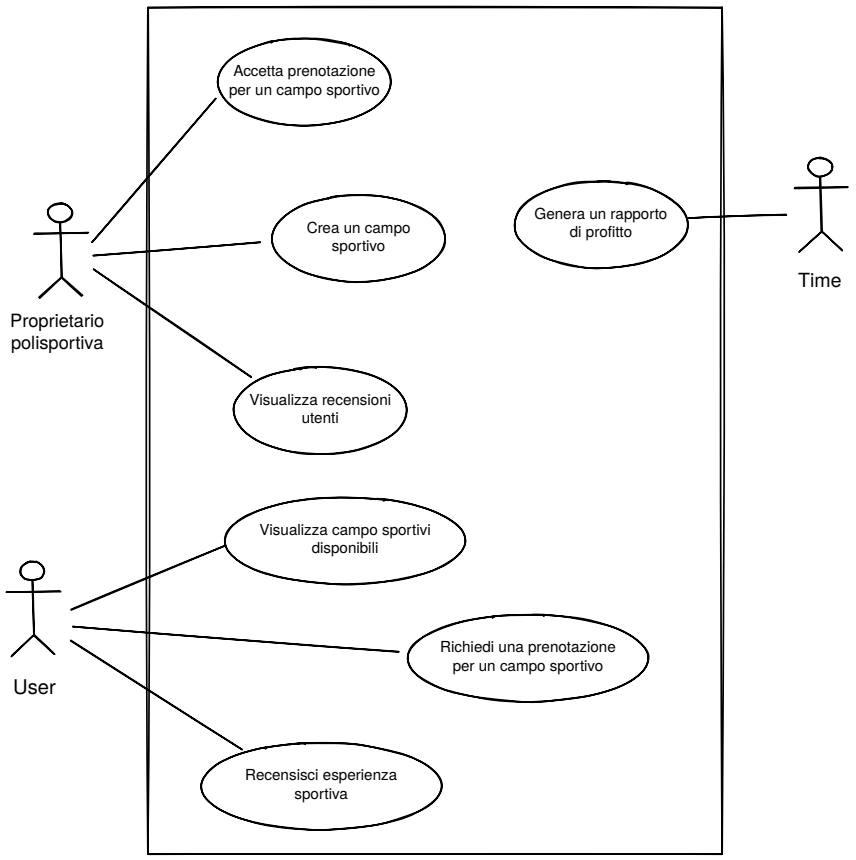


Figura 2.1: Use case diagram del Toy System

2.2 Design con OpenAPI

In questa sezione verrà presentato come definire un'API utilizzando OpenAPI e come generare codice a partire da un documento OpenAPI.

2.2.1 Documentare una web API con Swagger/OpenAPI 3

La definizione di un documento OpenAPI che descrive dettagliatamente la REST API in grado di soddisfare le funzioni presentate nel diagramma dei casi d'uso in Figura 2.1 è stato uno dei primi passi fatti per la realizzazione dell'applicativo. In questa sezione verrà presentato un solo endpoint sotto forma di documento OpenAPI relativo alla risorsa `SportsField` che rappresenta un campo sportivo. Un campo sportivo appartiene ad una polisportiva e può essere un campo da calcio, da tennis, da pallavolo oppure da basket. Il class diagram informale rappresentato in Figura 2.2 rappresenta la gerarchia di campi sportivi e le sue dirette relazioni più importanti con le altre classi.

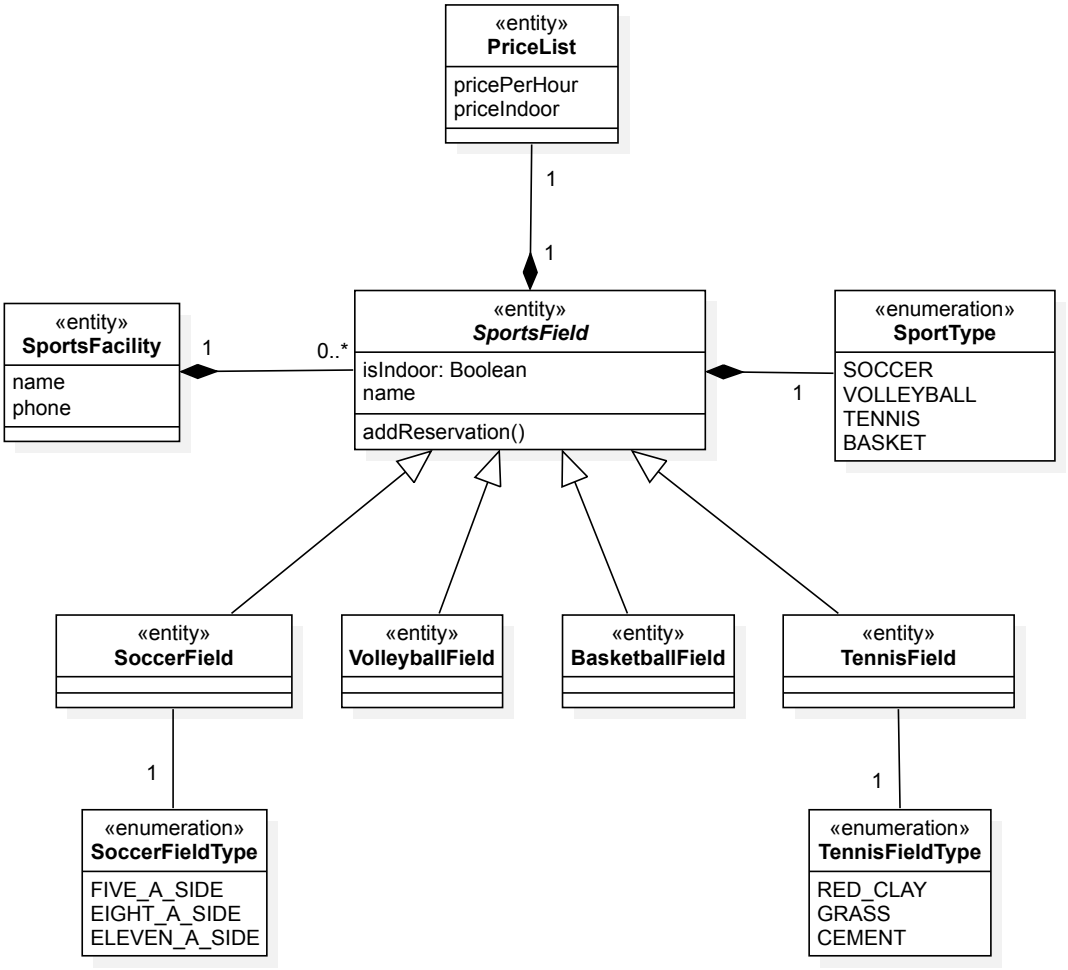


Figura 2.2: Class diagram informale gerarchia campi sportivi

L'endpoint che vogliamo documentare con OpenAPI è il seguente:

Metodo HTTP	Path	Descrizione
GET	/sports-fields	Restituisce la lista di tutti i campi sportivi. I campi sportivi possono essere filtrati per sport, per proprietario e si può richiedere di ordinare la lista per nome (crescente o decrescente).

Tabella 2.1: Descrizione non dettagliata dell'endpoint `GET /sports-fields`

La Tabella 2.1 descrive concettualmente il compito di tale endpoint ma non fornisce tuttavia una descrizione esaustiva. OpenAPI permette di esprimere in maniera semplice, coincisa e non ambigua un REST endpoint come quello in Tabella 2.1 in tutti i suoi aspetti. Di seguito viene riportato come scrivere un documento OpenAPI che descriva in maniera più dettagliata l'endpoint in Tabella 2.1.

Codice 2.1 Definizione dell'endpoint GET /sports-fields con OpenAPI

```yaml
openapi: 3.0.1
info:
  title: Polysport API
  version: '1.0'
tags:
  - name: Sports Fields
    description: API for managing sports fields
paths:
  /sports-fields:
    get:
      summary: >
        Gets the list of sports fields
      operationId: getSportsFields
      tags:
        - Sports Fields
      parameters:
        - name: filter_by_sport
          in: query
          required: false
          schema:
            $ref: '#/components/schemas/SportEnum'
        - name: sort_by
          in: query
          required: false
          schema:
            type: string
            enum: ['name.desc', 'name.asc']
            default: 'name.asc'
        - name: filter_by_owner_id
```

```
      in: query
      required: false
      schema:
        $ref: '#/components/schemas/ID'
  responses:
    '200':
      description: >
        The list of sports fields
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/SportsField'
    '400':
      $ref: '#/components/responses/400BadRequest'
```

Un documento OpenAPI deve iniziare con il campo `openapi` e specificare in esso la versione delle specifiche OpenAPI. Tale versione viene utilizzata dagli strumenti di Swagger per interpretare il documento. Il campo `info` è obbligatorio e ha lo scopo di associare metadati riguardo l'API come la sua versione e il titolo. Il campo `tags` fornisce ulteriori metadati che possono essere utilizzati nel resto del documento. Dopo aver settato questi campi, è possibile cominciare a definire i percorsi e le operazioni disponibili che la REST API dovrà offrire. In particolare, sotto e all'interno del campo `paths` viene specificato il percorso `/sports-fields`. Sotto ad un percorso bisogna specificare le operazioni che sono possibili a nome di quella risorsa (come ad esempio `get`, `put` o `post`). Quindi, viene specificata l'operazione `get` che a sua volta contiene i seguenti campi:

- `summary`: un breve riassunto dell'operazione;
- `operationId`: una stringa che identifica univocamente l'operazione;
- `tags`: i tag ci consentono di assegnare un nome alla nostra operazione per facilitarne la ricerca nella documentazione. I tag possono essere utilizzati per il raggruppamento logico delle operazioni per risorse o qualsiasi altro qualificatore (come quello definito all'inizio del documento);
- `parameters`: in questo campo vanno specificati tutti i parametri obbligatori e facoltativi che vanno inseriti all'interno della richiesta. In questo caso abbiamo tre *query parameters* (facoltativi):
  - `filter_by_sport`: può assumere un insieme limitato di valori e ha l'obiettivo di filtrare i campi sportivi per sport;
  - `sort_by`: impone un ordinamento alla lista di campi sportivi; Può assumere soltanto due valori `name.asc` (ordina per nome del campo in modo crescente) oppure `name.desc` (ordina per nome del campo in modo decrescente);

- `filter_by_owner_id`: può assumere solo valori interi non negativi. Se specificato, vengono restituiti soltanto i campi sportivi appartenenti alla persona identificata dall'id indicato.

Il campo `responses` contiene tutte le possibili risposte che possono essere restituite. In particolare l'operazione può andare a buon fine (Status Code 200) oppure la richiesta inviata potrebbe non essere corretta (Status Code 400). Il campo `schema` rappresenta un tipo di dato. Le specifiche OpenAPI definiscono tre tipi di dato primitivi: `integer` (formato `int32` o `int64` cioè long), `number` (formato `float` o `double`) e `string` (formato `password`). Il tipo di dato va specificato nel campo `type` mentre il formato nel campo `format` (opzionale). Entrambi i campi sono sottocampi di `schema`. I tipi di dato più complessi come gli oggetti (o modelli) hanno il campo `type` uguale ad `object`. I tipi di dato possono essere definiti in un unico posto nello stesso documento o in un altro documento e si può far riferimento ad essi con il campo `$ref`. Al termine del documento, nel campo `components`, vengono definiti i tipi di dato utilizzati più spesso nel documento nel sottocampo `schemas`.

Codice 2.2 Definizione globale degli schemas (tipi di dato)

```yaml
components:
    schemas:
        ID:
          type: integer
          format: int64
          readOnly: true
          minimum: 0
        SportEnum:
          type: string
          enum: ['soccer', 'basket', 'volleyball', 'tennis']
        BaseSportsField:
          type: object
          required:
            - id
            - sportsFacilityId
            - name
            - sport
            - isIndoor
            - priceList
          properties:
            id:
              $ref: '#/components/schemas/ID'
            sportsFacilityId:
              $ref: '#/components/schemas/ID'
            name:
              type: string
            sport:
              type: string
            isIndoor:
              type: boolean
```

```
    priceList:
      $ref: '#/components/schemas/SportsFieldPriceList'
SoccerField:
  allOf:
    - $ref: '#/components/schemas/BaseSportsField'
    - type: object
      required:
        - soccerFieldType
      properties:
        soccerFieldType:
          type: string
          enum: ['5 a side', '8 a side', '11 a side']
VolleyballField:
  allOf:
    - $ref: '#/components/schemas/BaseSportsField'
    - type: object
BasketballField:
  allOf:
    - $ref: '#/components/schemas/BaseSportsField'
    - type: object
TennisField:
  allOf:
    - $ref: '#/components/schemas/BaseSportsField'
    - type: object
      properties:
        tennisFieldType:
          type: string
          enum: ['red clay', 'grass', 'cement']
SportsField:
  oneOf:
    - $ref: '#/components/schemas/BasketballField'
    - $ref: '#/components/schemas/VolleyballField'
    - $ref: '#/components/schemas/SoccerField'
    - $ref: '#/components/schemas/TennisField'
  discriminator:
    propertyName: sport
    mapping:
      soccer:     '#/components/schemas/SoccerField'
      basket:     '#/components/schemas/BasketballField'
      volleyball: '#/components/schemas/VolleyballField'
      tennis:     '#/components/schemas/TennisField'
SportsFieldPriceList:
  type: object
  properties:
    pricePerHour:
      type: number
      format: float
      minimum: 0
```

Quando la richiesta `GET /sports-fields` va a buon fine (HTTP Status Code 200), il body della risposta sarà in formato `application/json` e, in particolare, sarà un array di oggetti

di tipo `SportsField` proprio come indicato in Codice 2.1. In base al tipo di campo sportivo, il contenuto della risposta può cambiare. Per questo motivo, nella definizione dello schema `SportsField`, è stato inserito un sottocampo `oneOf` il quale indica che il body contenuto nella risposta sarà uno dei quattro indicati sotto di esso. Per far sì che le cose funzionino effettivamente, è necessario indicare una proprietà che assumerà il ruolo di `discriminator` (*proprietà discriminante*) [18]. La proprietà discriminante deve essere presente in ogni tipo di dato indicato sotto `oneOf`. Il discriminatore dichiara in modo esplicito quale proprietà è possibile esaminare per determinare il tipo di oggetto (in questo caso il valore della proprietà `sport`). I possibili valori sono `soccer`, `basket`, `volleyball` e `tennis`, ognuno dei quali ha un particolare schema associato. Quindi, se un oggetto nella lista ha il valore `soccer` nella proprietà `sport`, verrà scelto lo schema `SoccerField` come rappresentazione della risorsa. Una possibile risposta potrebbe essere dunque la seguente:


```
[
  {
    "id": 1032,
    "sportsFacilityId": 1017,
    "name": "Eden Soccer Field",
    "sport": "soccer",
    "isIndoor": true,
    "priceList": {
      "pricePerHour": 10
    },
    "soccerFieldType": "5 a side"
  },
  {
    "id": 1033,
    "sportsFacilityId": 1018,
    "name": "Tennis Club Napoli",
    "sport": "tennis",
    "isIndoor": false,
    "priceList": {
      "pricePerHour": 8
    },
    "tennisFieldType": "red clay"
  }
]
```


Questo supporto al polimorfismo offerto da OpenAPI ha reso possibile rappresentare la gerarchia di campi sportivi così come è stata presentata in precedenza nel class diagram in Figura 2.2.

2.2.2 Generare codice a partire da un documento OpenAPI

Uno dei vantaggi dell'utilizzo di OpenAPI per documentare una web API sono le altre opportunità che ci offre dopo che la documentazione è stata messa a punto e completata. Progettando un'API in OpenAPI, possiamo utilizzare gli strumenti offerti da Swagger per accelerare lo sviluppo. *Swagger Codegen* è uno strumento molto utile per la generazione automatica di codice per le API in una varietà di linguaggi, come Java, Go, Ruby e molti altri [19]. Ciò consente di risparmiare tempo nella creazione della struttura di base dell'API e consente agli sviluppatori di concentrarsi sulla logica di business specifica da implementare. Swagger Codegen può anche aiutare a garantire che l'API sia implementata in modo coerente con la progettazione concordata, poiché il codice generato dovrebbe seguire fedelmente la struttura definita nel documento OpenAPI.

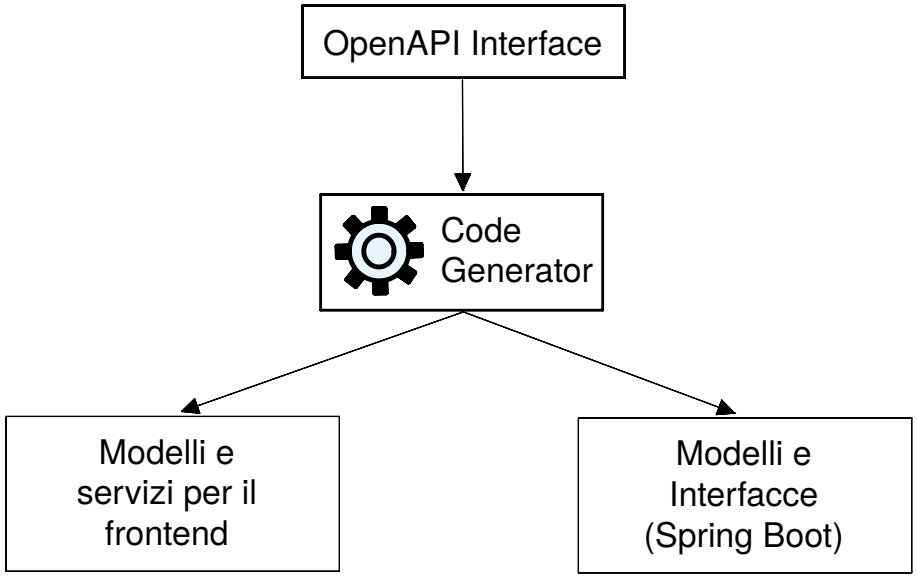


Figura 2.3: Processo di generazione del codice a partire da un documento OpenAPI

Per accelerare il processo di inizializzazione di un progetto Spring Boot, il processo di sviluppo Fervento sfrutta un toolkit di Maven per modelli di progetti Maven chiamato *Archetype*. Gli archetipi Maven sono dei template che vengono utilizzati per generare nuovi progetti rapidamente secondo una determinata struttura [20]. Qualsiasi progetto Maven può essere convertito in un archetipo. Un progetto può essere generato da un archetipo che si trova in una repository remota (come Maven Central) oppure nella repository locale di Maven. Il processo di sviluppo Fervento utilizza un archetipo Maven predisposto alla generazione di codice a partire da un documento OpenAPI. Dopo aver installato l'archetipo correttamente nella repository locale, per generare un progetto da esso basta eseguire il comando `mvn archetype:generate` con l'opzione `-DarchetypeCatalog=local` (per usare il catalogo locale) e scegliere l'archetipo:

```
$ mvn archetype:generate -DarchetypeCatalog=local

[INFO] Scanning for projects...
[INFO] ------------------< org.apache.maven:standalone-pom >---------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------[ pom ]---------------------------------
[INFO] >>> maven-archetype-plugin:3.2.1:generate (default-cli) > ...
1: local -> systems.fervento:springboot-archetype (Parent pom providing \
    dependency and plugin management for applications built with Maven)
Choose a number or apply filter \
    (format: [groupId:]artifactId, case sensitive contains): : 1
Define value for property 'groupId': systems.fervento
Define value for property 'artifactId': polisportiva
Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' systems.fervento: : \
    systems.fervento.polisportiva
...
[INFO] BUILD SUCCESS
```

Il progetto così generato incorpora un plugin fondamentale nel `pom.xml` per la generazione del codice a partire da un documento OpenAPI [21]. Il plugin in questione si chiama `org.openapitools:openapi-generator-maven-plugin:<version>` ed esegue il goal `generate` nella fase `generate-sources`. Questo vuol dire che ogni volta che eseguiamo un comando `mvn` per eseguire la fase `generate-sources` o qualsiasi altra fase che si trova dopo `generate-sources` (come `compile`), il generatore di codice effettuerà il parsing del documento OpenAPI e genererà i modelli e le interfacce API corrispondenti in una directory specifica. Tale plugin è stato configurato per cercare il documento OpenAPI nella cartella `${project.basedir}/src/main/resources/api.openapi.yml` dove `project.basedir` è il percorso assoluto della directory in cui il progetto corrente si trova.

2.3 Implementazione con Spring Boot

In questa sezione verrà mostrata più in dettaglio l'architettura della REST API e come i vari componenti interagiscono tra di loro nel processo di sviluppo Fervento. Inoltre, verrà mostrato l'implementazione dell'endpoint presentato in Sezione 2.2.1 e come sfruttare il codice generato da Swagger Codegen per accelerare il processo di sviluppo.

2.3.1 Design del backend

L'architettura del backend è molto simile a quella presentata in Figura 1.1 nel Capitolo 1. La Figura 2.4 ritrae l'architettura dell'applicazione più in dettaglio.

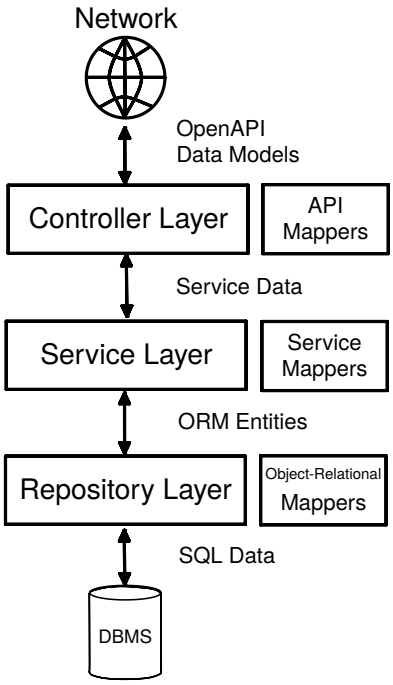


Figura 2.4: Backend design

Il *Controller Layer* riveste il ruolo di livello di presentazione e si occupa di interpretare le richieste dell'utente, di coordinare l'elaborazione delle richieste con gli altri livelli dell'architettura e di restituire la risposta all'utente. Implementare questo livello diventa molto semplice quando si adotta un'approccio *Specification-First* siccome il documento OpenAPI è già stato scritto e i modelli che il livello di controllo deve restituire al chiamante sono già pronti. Un documento OpenAPI è un vero e proprio contratto da rispettare e il livello di controllo deve restituire al client esattamente gli stessi modelli definiti nel contratto. Infatti, ogni controller che appartiene al livello di controllo, implementa un'interfaccia autogenerata. Ogni livello ha i propri dati su cui può lavorare per eseguire i propri compiti e fornire servizi al livello superiore. Un livello non conosce nulla dei

livelli superiori e, quindi, non conosce i modelli che il livello superiore utilizza. Per questo motivo, come mostrato in Figura 2.4, ogni livello ha i propri *mappers*. Un mapper è un componente di software che si occupa di effettuare una mappatura, ovvero di trasformare o convertire un oggetto o un insieme di dati in un altro formato impostando così delle comunicazioni tra sottosistemi che devono comunque rimanere all'oscuro l'uno dell'altro [22]. Ogni livello, per poter comunicare con il livello immediatamente sotto di esso, deve convertire (quando necessario) le informazioni in un formato comprensibile al livello che riceve tali dati. I mapper possono essere scritti manualmente o generati automaticamente utilizzando apposite librerie che facilitano questo compito. Lo strumento utilizzato per velocizzare questo processo di traduzione delle informazioni tra livelli è *MapStruct* [23]. Il *Service Layer* si occupa di gestire le regole di business dell'applicazione (calcoli e validazione). Un componente che appartiene a questo livello può essere implementato ignorando completamente il livello di controllo e trattando il livello di accesso ai dati (*Repository Layer*) come un insieme astratto di funzioni che permettono di ottenere e manipolare i dati. Il livello di servizio, quando vuole comunicare con il livello di accesso ai dati, se necessario converte la propria rappresentazione dei dati in entità e invoca una o più funzioni del livello di accesso ai dati. Alla fine, il livello di servizio mappa le entità nella propria rappresentazione dei dati e restituisce il risultato al livello di controllo. Il *Repository Layer* si occupa di gestire i dati persistenti in un database o in servizi remoti. L'astrazione dalla sorgente dei dati permette di far fronte con grazia a un cambiamento nella tecnologia del database o di supportare servizi di persistenza che possono cambiare con poco preavviso. Il framework Spring offre un componente chiamato *Spring Data JPA* [24] che permette allo sviluppatore di implementare il livello di accesso ai dati con poche righe di codice. Lo sviluppatore scrive le interfacce del repository, compresi i metodi di ricerca personalizzati, e Spring ne fornisce automaticamente l'implementazione. Spring Data JPA utilizza le *Java Persistance API* (JPA) [25] le quali sono delle specifiche per gestire la persistenza di oggetti e del mapping oggetto-relazione (*object/relational mapping*). In poche parole, l'object/relational mapping (ORM) è la persistenza automatica (e trasparente) degli oggetti di un'applicazione Java nelle tabelle di un RDBMS (*Relational Database Management System*). In sostanza, l'ORM funziona trasformando (reversibilmente) i dati da una rappresentazione ad un'altra. Spring Data JPA è un livello aggiuntivo in cima alle implementazioni JPA. L'ORM utilizzato che implementa le specifiche JPA è *Hibernate* [26].

Lombok

Uno strumento utilizzato frequentemente durante lo sviluppo dell'applicazione è *Lombok*. Lombok è una libreria Java che permette allo sviluppatore di scrivere codice Java più velocemente generando automaticamente costruttori, getter, setter, toString e altro utilizzando delle semplici annotazioni da intestare sulle classi Java . Ad esempio, per generare un metodo getter per ogni variabile d'istanza della classe, basta specificare l'annotazione `@Getter` sulla classe [27].

2.3.2 Implementazione di un REST endpoint

In questa sezione verrà mostrata l'implementazione dell'endpoint `GET /sports-fields` definito nel documento OpenAPI in Codice 2.1 nella Sezione 2.2.1. Il codice è organizzato in quattro package principali: `controller`, `service`, `repository` ed `entity`.

Livello di controllo

Il livello di controllo sfrutta i modelli (convenzionalmente chiamati *Data Models*) e le interfacce generate da Swagger Codegen. I controller implementano la definizione degli endpoint così come sono stati definiti nel documento OpenAPI: l'aderenza al contratto definito dal documento OpenAPI viene imposta tramite l'implementazione di interfacce autogenerate da Swagger Codegen. L'endpoint `GET /sports-fields` fa parte del controller `SportsFieldApiController` che implementa l'interfaccia `SportsFieldsApi` autogenerata da Swagger Codegen.

Codice 2.3 Implementazione endpoint `GET /sports-fields` livello di controllo

```
package systems.fervento.sportsclub.controller;
...

@RestController
public class SportsFieldApiController implements SportsFieldsApi {

    private final SportsFieldService sportsFieldService;
    ...

    @Override
    public ResponseEntity<List<SportsField>> getSportsFields(
        final SportEnum filterBySport,
        final String sortBy,
        final Long filterByOwnerId
    ) {
        final String sport = (filterBySport == null) ?
            null :
            filterBySport.toString();
        return ResponseEntity.ok(
            sportsFieldService
                .getSportsFields(sortBy, sport, filterByOwnerId)
                .stream()
                .map(sportsFieldApiMapper::mapToSportsFieldApi)
                .collect(toList())
        );
    }

    ...
}
```

Da come è possibile notare in Codice 2.3, il metodo `getSportsFields` effettua l'override del metodo ereditato dall'interfaccia autogenerata `SportsFieldsApi` ed utilizza i modelli

autogenerati come `SportsField` e `SportsEnum`. Il metodo invoca il livello di servizio e il risultato ottenuto da tale invocazione viene mappato nei modelli previsti dal documento OpenAPI e restituito al mittente della richiesta HTTP.

Livello di servizio

Il livello di servizio è composto da dati convenzionalmente chiamati *Service Data* che permettono di realizzare la logica di business e ignorano completamente la parte di comunicazione su rete e design della REST API. Il metodo invocato dal livello di controllo per soddisfare la richiesta `GET /sports-fields` è il metodo di servizio `getSportsFields` della classe `SportsFieldService`.

Codice 2.4 Implementazione endpoint `GET /sports-fields` livello di servizio

```
package systems.fervento.sportsclub.service;
...

@Service
public class SportsFieldService {
    private final SportsFieldRepository sportsFieldRepository;
    private final SportsFieldDataMapper sportsFieldDataMapper;
    ...

    public List<SportsFieldData> getSportsFields(
        String sortBy,
        final String sport,
        final Long ownerId
    ) {
        Objects.requireNonNull(sortBy);
        final String[] sortingInfo  = sortBy.split("\\.");
        final String sortOrder    = sortingInfo[1];
        final String sortProperty = sortingInfo[0];

        final Sort.Direction sortDirection = "desc".equals(sortOrder)
            ? Sort.Direction.DESC
            : Sort.Direction.ASC;

        final Sort sort = Sort.by(sortDirection, sortProperty);

        final List<SportsFieldData> sportsFieldsData = sportsFieldRepository
            .getSportsFields(sort, sport, ownerId)
            .stream()
            .map(sportsFieldDataMapper::mapToSportsFieldData)
            .toList();

        return sportsFieldsData;
    }

    ...
}
```

Il metodo `getSportsField` in Codice 2.4 ha l'obiettivo di restituire una lista di oggetti di tipo `SportsFieldData` interagendo con il livello di accesso ai dati. Il contenuto della lista può essere filtrato per sport, per id di un proprietario di uno o più campi sportivi e ordinato per il nome del campo sportivo (crescente o decrescente). Il risultato ottenuto dal livello di accesso ai dati deve essere poi mappato in dati appartenenti al livello di servizio prima di essere restituiti al chiamante.

Livello di accesso ai dati

Il livello di accesso ai dati gestisce i dati persistenti chiamati *Entity Data* e maschera le difficoltà di interazione con una base di dati. Le *entity* sono classi che rappresentano una tabella di un database. Il Codice 2.5 mostra l'interfaccia `SportsFieldRepository` che estende `JpaRepository` la quale è un'interfaccia fornita da Spring Data JPA e fornisce metodi di base per l'accesso ai dati.

Codice 2.5 Implementazione endpoint `GET /sports-fields` livello di accesso ai dati

```
package systems.fervento.sportsclub.repository;
...

@Repository
public interface SportsFieldRepository
    extends JpaRepository<SportsFieldEntity, Long> {

    @Query(
        "select s " +
        "from SportsFieldEntity s " +
        "where (:sport is null   or :sport = s.sport)
           and (:ownerId is null or :ownerId = s.sportsFacility.id)"
    )
    List<SportsFieldEntity> getSportsFields(
        Sort sort,
        String sport,
        Long ownerId
    );

    ...
}
```

Per utilizzare `JpaRepository`, è sufficiente creare un'interfaccia che estenda `JpaRepository` e specificare l'entità e il tipo della chiave primaria dell'entità come tipi di parametro dell'interfaccia `JpaRepository` (in questo caso `SportsFieldEntity` e `Long`). Spring creerà automaticamente un'implementazione di questa interfaccia durante l'avvio dell'applicazione fornendo i metodi di base per eseguire le operazioni CRUD. Tuttavia, il metodo `getSportsFields` in Codice 2.5 definisce una query personalizzata per effettuare interrogazioni sulla base di dati più complesse come il filtraggio e l'ordinamento. Questo è possibile farlo grazie all'annotazione `@Query` di Spring Data JPA la quale permette di definire query in SQL oppure in JPQL (*Java Persistance Query Language*) [28].

–3–
Valutazione dei costi-benefici industriali dell'adozione delle nuove JVM

Contenuti: **3.1 Eclipse OpenJ9.** 3.1.1 Politiche di Garbage Collection. **3.2 GraalVM.** 3.2.1 GraalVM Native Image. **3.3 Valutazione dei benefici di Performance.** 3.3.1 Descrizione dei test – 3.3.2 Risultati e Grafici.

In questo capitolo, verranno brevemente presentate le Java Virtual Machines ad alte prestazioni come Eclipse OpenJ9 e, in particolare, GraalVM e il suo compilatore Graal che può essere usato sia come compilatore just-in-time sia come compilatore ahead-of-time per le applicazioni Java. Verranno descritte le caratteristiche e i dettagli di ogni tecnologia come anche le difficoltà incontrate nell'applicarle al Toy System [2] e i costi, sia in termini di tempo che di risorse, che sono stati necessari per l'integrazione delle stesse, come la necessità di risolvere problemi, effettuare test, apportare modifiche al codice e acquisire nuove competenze tecnologiche. Queste tecnologie sono state applicate per eseguire e valutare le performance del Toy System in un ambiente isolato (docker container) e verrà fatta una valutazione dei risultati ottenuti per ogni tecnologia evidenziando vantaggi e svantaggi. Nonostante siano stati applicati su un sistema modello, i risultati ottenuti dai test delle performance sono comunque significativi in quanto il Toy System utilizzato in questa tesi rappresenta un sistema in miniatura di come sono quelli enterprise large-scale, rendendo i risultati ottenuti trasferibili a sistemi di grandi dimensioni.

3.1 Eclipse OpenJ9

Eclipse OpenJ9 è un'implementazione autonoma della Java Virtual Machine (JVM) sviluppata dalla Eclipse Foundation [29]. La JVM OpenJ9 è integrata con le librerie di classi Java di OpenJDK per creare un Java Development Kit (JDK) completo che ottimizza le

prestazioni, le dimensioni e l'utilizzo delle risorse dell'applicazione e risulta adatta per le distribuzioni cloud. OpenJ9 è configurato con un insieme di opzioni predefinite che forniscono un ambiente di runtime ottimale per le applicazioni Java con carichi di lavoro comuni. Tuttavia, le ampie opzioni di configurazione assicurano che la JVM possa essere regolata per soddisfare i requisiti di un'ampia gamma di applicazioni Java, da quelle aziendali complesse eseguite su hardware mainframe a quelle di breve durata eseguite su servizi cloud basati su container.

3.1.1 Politiche di Garbage Collection

OpenJ9 mette a disposizione diverse politiche di garbage collection (GC) ognuna delle quali struttura la memoria heap in maniera differente. La politica di garbage collection utilizzata per il Toy System è la *Generational Concurrent GC policy* [30], che è quella di default. Tale politica divide lo Java heap in due aree principali, la *nursery area*, in cui nuovi oggetti vengono allocati e la *tenure area*, in cui risiedono gli oggetti che raggiungono una determinata età (detta anche *tenure age*). La survivor area è a sua volta suddivisa in due aree, *allocate area*, in cui vengono effettivamente allocati i nuovi oggetti e *survivor area*, in cui vengono piazzati gli oggetti che sopravvivono al processo di garbage collection. Quando la nursery area diventa piena, viene avviato un particolare ciclo di garbage collection che coinvolge un'operazione denominata *GC scavenge operation* [31] la quale copia gli oggetti raggiungibili (presenti nell'allocate area) o nella survivor area o nella tenure area se hanno raggiunto la tenure age. A questo punto i ruoli dell'allocate area e della survivor area si scambiano, e la survivor area diventa il nuovo spazio su cui allocare nuovi oggetti pronto per la prossima GC scavenge operation. Questa politica di garbage collection risulta adatta per le applicazioni transazionali come quella del Toy System cioè quelle applicazioni che eseguono operazioni di richiesta/risposta, dato che sono costituite da molti oggetti allocati nella memoria heap che hanno una vita piuttosto breve. La JVM OpenJ9 di default alloca il 25% della dimensione massima della memoria heap all'allocate area e il restante 75% alla tenure area. Tuttavia, questi valori possono cambiare a runtime, in quanto OpenJ9 può contrarre o espandere la memoria heap (nei limiti della dimensione massima specificata) e tali mutazioni avvengono come parte di un ciclo di GC. Il meccanismo di garbage collection di OpenJ9 permette di avere un consumo di memoria molto ridotto mantenendo comunque un throughput alto cercando di contenere i cicli di GC durante l'esecuzione. Il costo di applicazione di OpenJ9 al Toy System riscontrato è pari a zero dato che è un *drop-in replacement*, cioè può sostituire un altro JDK senza che siano necessarie modifiche al codice o configurazioni particolari.

3.2 GraalVM

GraalVM è una virtual machine universale che mira a fornire prestazioni elevate sia per le applicazioni on-premise e sia per i microservizi nel cloud [32]. Costruita sulla Java HotSpot virtual machine, GraalVM migliora le prestazioni di quest'ultima attraverso l'utilizzo di

ottimizzazioni del compilatore avanzate. In particolare, fornisce un compilatore just-in-time (JIT) chiamato Graal che accelera le applicazioni Java migliorando l'intero ambiente di esecuzione offrendo un Java Development Kit (JDK) compatibile e con prestazioni migliori. Il compilatore Graal è un'implementazione della Java Virtual Machine Compiler Interface (JVMCI) [33], il quale è completamente scritto in Java e utilizza le tecniche di ottimizzazione del compilatore Java JIT (compilatore C2 di HotSpot) come baseline. Nonostante GraalVM sia basata su Java, essa non supporta solo Java, ma consente uno sviluppo poliglotta con JavaScript, Python, R, Ruby, C, C++ e Rust. I linguaggi come C, C++ e Rust vengono eseguiti su Sulong il quale è un ambiente di esecuzione per i linguaggi basati su LLVM (Low Level Virtual Machine). Inoltre, GraalVM fornisce un framework estensibile chiamato Truffle che permette di costruire ed eseguire qualsiasi linguaggio sulla piattaforma. La Figura 3.1 ritrae l'architettura di GraalVM ad alto livello.

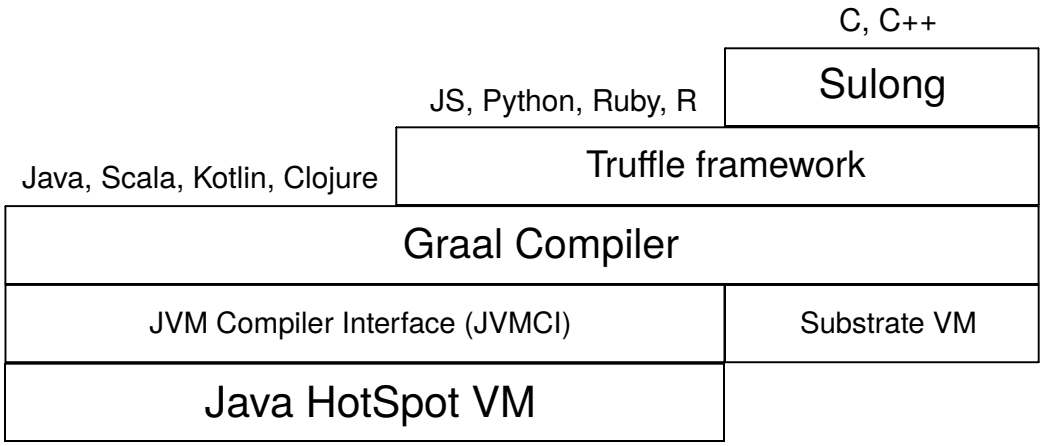


Figura 3.1: GraalVM Architecture

Come Eclipse OpenJ9, GraalVM è un drop-in replacement e quindi può sostituire un qualsiasi JDK senza apportare modifiche al codice o fornire particolari configurazioni.

3.2.1 GraalVM Native Image

Il compilatore Graal può essere utilizzato come un compilatore ahead-of-time per compilare l'applicazione Java in un eseguibile nativo (detta anche *native image*) [34]. Gli eseguibili nativi hanno dimensioni molto ridotte, si avviano quasi istantaneamente e consumano una frazione delle risorse di calcolo necessarie per eseguire la stessa applicazione Java su una JVM. Una native image non ha un compilatore JIT, quindi non ha bisogno di una JVM completa per eseguire. Invece, viene integrata nell'eseguibile nativo una virtual machine di dimensione ridotte chiamata SubstrateVM, fornendo il sottoinsieme necessario di funzionalità JVM come la gestione della memoria e lo scheduling dei thread. La costruzione di una native image richiede tempo dato che viene fatta principalmente analisi del codice statica per ottimizzare quanto più possibile l'immagine. Nel processo di

costruzione della native image, il compilatore Graal AOT raggruppa tutto il codice dell'applicativo, le sue dipendenze insieme alle classi del JDK e alle classi della SubstrateVM in un singolo package. Successivamente, viene fatta un'analisi statica del codice (detta anche *points-to analysis*) cioè quel processo che determina quali elementi dell'applicazione sono effettivamente utilizzati (classi, metodi e campi). Questo processo prosegue iterativamente (partendo dal `main`) fino a quando non è possibile scoprire nient'altro. Solo gli elementi raggiungibili sono inclusi nell'immagine finale e, una volta creata, non è possibile aggiungere nuovi elementi in fase di esecuzione, ad esempio tramite il class loading (questo vincolo viene anche chiamato *closed-world assumption*). Inoltre, le classi possono anche essere inizializzate durante la costruzione dell'immagine per eliminare l'overhead a runtime dovuto alle inizializzazioni delle classi stesse. Dopo che l'analisi ha raggiunto la convergenza, tutte le classi raggiungibili all'interno dell'heap (gli oggetti `Class`), gli oggetti creati durante la costruzione dell'immagine (ad esempio gli oggetti creati durante l'inizializzazione delle classi) e il runtime SubstrateVM vengono salvati in un *Native Image Heap*, che fa parte dell'eseguibile dell'immagine nativa e sarà direttamente accessibile dopo l'avvio dell'immagine stessa. L'immagine finale avrà quindi due sezioni, una che contiene il codice compilato binario ottimizzato e l'altra che contiene il native image heap. Siccome viene fatta un'analisi statica e dunque l'applicativo non viene eseguito, il native image builder non è consapevole degli elementi dinamici dell'applicazione e deve essere informato riguardo aspetti come la riflessione, le risorse esterne utilizzate, la serializzazione e i proxy dinamici. Anche se l'analisi statica cerca comunque di raccogliere le informazioni dinamiche del programma, essa non riesce tuttavia a prevedere in modo esaustivo tutti gli usi. Esistono diversi modi per fornire i metadati al native image builder (anche detti *reachability metadata*). Nel costruire l'immagine nativa del Toy System è risultato necessario fornire alcuni metadati riguardo alcune classi dell'applicativo. Il polimorfismo presente nella gerarchia delle classi in figura 2.2 mostrata nel capitolo 2 non viene correttamente individuato dal native image builder perché le informazioni riguardo le classi concrete della classe astratta `SportsField` non vengono registrate. Queste informazioni possono essere fornite al native image builder sotto forma di file JSON da posizionare alla radice del progetto [35] ma, fortunatamente, il framework Spring Boot 3 supporta la costruzione dell'immagine nativa e fornisce delle API di facile utilizzo per fornire i metadati necessari [36]. Il codice 3.1 rappresenta le configurazioni risultate necessarie per fornire i metadati al native image builder per il corretto funzionamento dell'applicativo.

Codice 3.1 Fornire metadati al native image builder con Spring Boot 3

```
package systems.fervento.sportsclub.config;

import org.springframework.aot.hint.MemberCategory;
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.RuntimeHintsRegistrar;
...

@Configuration
@ImportRuntimeHints(AOTConfig.RunTimeHintsRegistrarImpl.class)
```

```
public class AOTConfig {
    static class RunTimeHintsRegistrarImpl implements RuntimeHintsRegistrar {

        @Override
        public void registerHints(
            RuntimeHints hints,
            ClassLoader classLoader
        ) {
            List.of(
                RFC3339DateFormat.class,
                TennisField.class,
                TennisFieldAllOf.class,
                SoccerField.class,
                SoccerFieldAllOf.class,
                BasketballField.class,
                VolleyballField.class,
                SportsFieldPriceList.class,
            ).forEach(type ->
                hints.reflection().registerType(
                    type,
                    MemberCategory.values()
                )
            );
        }
    }
}
```

Con l'API `RuntimeHints` di Spring Boot 3, è possibile fornire i metadati necessari alla native image. Il codice 3.1 verrà invocato dall'AOT Engine di Spring Boot 3 [37] a tempo di compilazione, in particolare verrà invocato il metodo `registerHints`. Tale metodo crea una lista di oggetti di tipo `Class` che rappresenta una particolare classe e, per ogni oggetto `Class` nella lista, vengono registrate tutte le informazioni di riflessione (come i metodi e i campi) della classe le quali verranno poi inserite nella native image. Non fornire tutti i metadati necessari per l'esecuzione della native image potrebbe far fallire il processo di costruzione della stessa o, nel caso peggiore, il native image builder potrebbe comunque riuscire a costruire l'immagine ma l'applicazione potrebbe non funzionare come si ci aspetta durante l'esecuzione. Ad esempio, il codice 3.1 non è strettamente necessario per la costruzione della native image ma la sua omissione non consente all'applicazione a runtime di serializzare correttamente alcune classi. Una richiesta HTTP all'endpoint `GET /sports-fields` ritornerebbe il seguente body nella risposta HTTP:

```
[
  {
    "sport": "soccer"
  },
  {
    "sport": "tennis"
```

```
  }
]
```

Tuttavia, tale body contenuto nella risposta HTTP non è quello definito nel documento OpenAPI presentato nel capitolo 2. Questo perché l'applicazione conosce l'esistenza delle sottoclassi di `SportsField` (le quali tutte hanno in comune il campo `sport`) ma non conosce come sono fatte internamente, in termini di campi e metodi.

La compilazione AOT di Graal permette di costruire immagini native con uno startup time molto ridotto e con un footprint di memoria estremamente più basso rispetto alle tradizionali applicazioni Java che eseguono su una JVM. Queste caratteristiche rendono un'immagine nativa ideale per la distribuzione sul cloud e per la realizzazione di microservizi in cui queste peculiarità sono molto importanti. Tuttavia, come visto in questa sezione, l'immagine nativa richiede delle configurazioni particolari ed è inoltre cruciale verificare il corretto funzionamento del software per rilevare l'assenza di metadati essenziali. In aggiunta, le immagine native non hanno il vantaggio delle ottimizzazioni a runtime che i compilatori just-in-time eseguono all'interno di una JVM. Questo comporta un consumo di risorse basso e un throughput di punta immediato eliminando il tempo di warmup che di solito le JVM hanno. L'immagine nativa non è pensata per esecuzioni di lunga durata e quindi non raggiunge throughput alti mentre le JVM sono ideate proprio per stare in esecuzione per molto tempo e riescono a raggiungere alte prestazioni a lungo termine dovute alle ottimizzazioni del compilatore JIT. Tale differenza di throughput è dovuta al fatto che il runtime SubstrateVM utilizza di default un garbage collector seriale che utilizza un solo thread ideale per dimensioni dell'heap ridotte. Tuttavia, è possibile cambiare il garbage collector seriale con il garbage collector G1 basato su quello di HotSpot e ottimizzato per avere un throughput alto [38].

3.3 Valutazione dei benefici di Performance

In questa sezione verranno descritti i test di performance fatti sul Toy System sui vari ambienti di esecuzione. Verrà valutato il tempo di avvio dell'applicazione, il throughput e il consumo di CPU e di memoria dei differenti runtime e verranno fatte delle considerazioni sui benefici di performance che si possono ottenere scegliendo un determinato ambiente di esecuzione piuttosto che un altro.

3.3.1 Descrizione dei test

I test delle performance sono stati effettuti mentre l'applicazione eseguiva in un docker container. Per ogni ambiente di esecuzione, è stato creato un file docker il quale crea un'immagine docker dell'applicazione con l'obiettivo di eseguire il Toy System su un determinato runtime. In particolare, sono stati realizzati cinque file docker:

- `Dockerfile`: per eseguire su OpenJDK;

- `Dockerfile.graalvm`: per eseguire su GraalVM;
- `Dockerfile.openj9`: per eseguire su OpenJ9;
- `Dockerfile.native`: per costruire ed eseguire la native image con un garbage collector seriale single-threaded (Serial GC);
- `Dockerfile.native-g1`: per costruire ed eseguire la native image con un garbage collector multi-threaded (G1 GC).

Durante le valutazioni empiriche, i container sono stati eseguiti su un server dotato di un processore Intel(R) Core(TM) i9-9940X da 14 core a 3.30GHz (2 thread per core), 64 GB di RAM e un SSD da 1TB mentre i test delle performance su una macchina diversa con un processore AMD Ryzen 5 3500U con Radeon Vega Mobile Gfx a 4 core (2 thread per core), 8 GB di RAM e un SSD da 256GB. Per condurre i test, è stato utilizzato il tool JMeter [39], che consente di progettare e sottoporre carichi di lavoro al software da testare. Sono stati creati due carichi di lavoro, denominati Test 1 e Test 2, basati sui casi d'uso descritti nello use case diagram del Toy System in Figura 2.1 nel Capitolo 2. Entrambi i test inviano le stesse richieste, ma differiscono nella durata e nella quantità di utenti concorrenti attivi.

Test	Comportamento	Numero target di utenti concorrenti attivi	Tempo di Ramp Up (sec)	Hold Load (sec)
1	Crea un campo sportivo	100	1200	300
	Accetta prenotazione per un campo sportivo	100	1200	300
	Visualizza recensioni utenti	100	1200	300
	Visualizza campi sportivi disponibili	100	1200	300
	Richiedi una prenotazione per un campo sportivo	100	1200	300
	Recensisci esperienza sportiva	100	1200	300
2	Crea un campo sportivo	10	2	60
	Accetta prenotazione per un campo sportivo	10	2	60
	Visualizza recensioni utenti	10	2	60
	Visualizza campi sportivi disponibili	10	2	60
	Richiedi una prenotazione per un campo sportivo	10	2	60
	Recensisci esperienza sportiva	10	2	60

Tabella 3.1: Formalizzazione dei carichi di lavoro

I test sono dettagliati nella Tabella 3.1. Ogni comportamento viene eseguito ripetutamente (parallelamente a tutti gli altri comportamenti) da un gruppo di utenti concorrenti per tutta la durata del test. Il Test 1 è uno stress test e ha l'obiettivo di portare il software ai suoi limiti per capire quali sono effettivamente questi limiti e come si comporta l'applicazione sui diversi runtime quando è soggetta a un numero di richieste sproporzionato. Il Test 1 aumenterà lentamente il numero di utenti virtuali che accedono al Toy System fino a quando il volume del carico inizia a mettere in difficoltà l'applicazione, magari causando errori o tempi di risposta inaccettabili. In particolare, il Test 1 aumenterà il numero di utenti concorrenti fino a un massimo di 600 utenti nell'arco di 20 minuti (tempo di ramp up) e, successivamente, rimane costante con tale numero di utenti per ulteriori 5 minuti (tempo di hold load). Inoltre, il Test 1 è stato progettato anche con lo scopo di individuare i punti in cui le performance cominciano a degradare permettendo così di

progettare un secondo test con lo scopo di sottoporre il Toy System a carichi sostenibili. Infatti, il Test 2 è stato progettato proprio per capire come i vari ambienti di esecuzione si comportano quando sono soggetti a carichi previsti e quindi a un numero di richieste nel tempo sostenibile. A differenza del Test 1, il Test 2 ha una durata breve e raggiunge un numero massimo di utenti pari a 60 con un tempo di ramp up di 2 secondi (quindi un carico quasi istantaneo) e un tempo di hold load di 60 secondi.

3.3.2 Risultati e Grafici

Tempo di startup

Dalla Figura 3.2, si osserva che GraalVM non offre nessun vantaggio in termini di tempo di avvio dell'applicazione rispetto a OpenJDK. Inoltre, OpenJ9 sembrerebbe avere un tempo di avvio superiore a tutti gli altri runtime. La versione native image di GraalVM dell'applicazione offre un tempo di avvio quasi istantaneo predisponendo l'applicazione a essere eseguita in container rendendo lo scaling orizzontale più efficiente. Un tempo di avvio ridotto è un fattore chiave per garantire la scalabilità, l'efficienza e la flessibilità delle applicazioni cloud e l'immagine nativa di GraalVM soddisfa questa necessità.

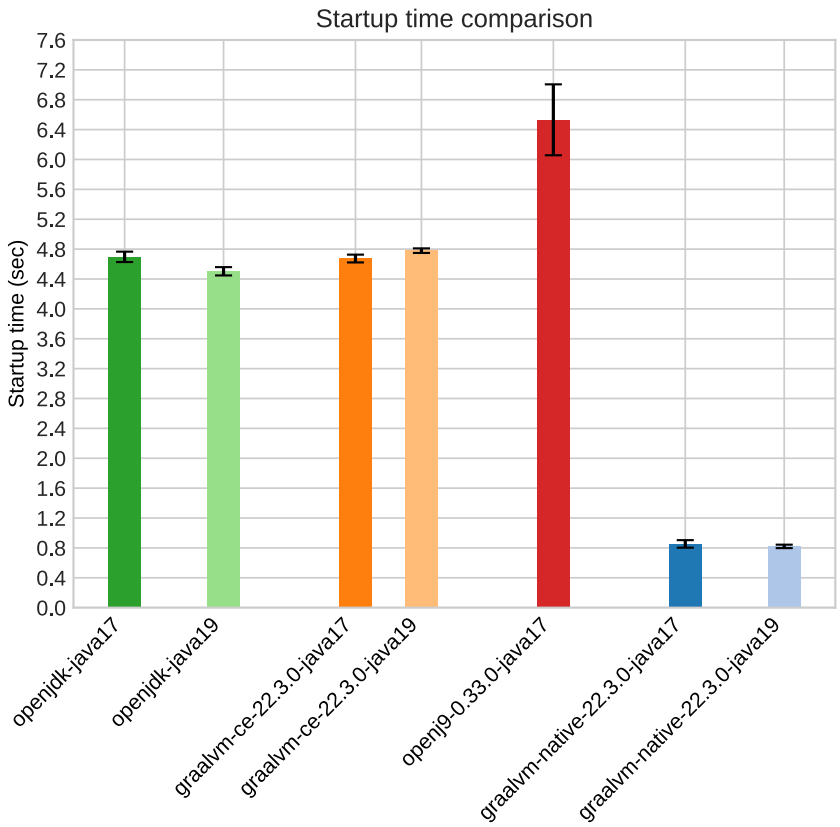


Figura 3.2: Confronto tempo di avvio dell'applicazione

Test 1 (Stress Test) - Throughput

La Figura 3.3 mostra l'andamento del throughput del Toy System in esecuzione sulle diverse JVM all'aumentare del numero di utenti concorrenti. Dal grafico, si osserva che tutte le JVM hanno un comportamento molto simile anche per quanto riguarda il tempo di warm up (o tempo di riscaldamento), cioè quel tempo necessario affinché l'applicazione Java raggiunga le prestazioni ottimali del codice compilato grazie al compilatore JIT. Da come si evince dal grafico, GraalVM riesce a raggiunge un throughput di punta più alto rispetto alle altre JVM.

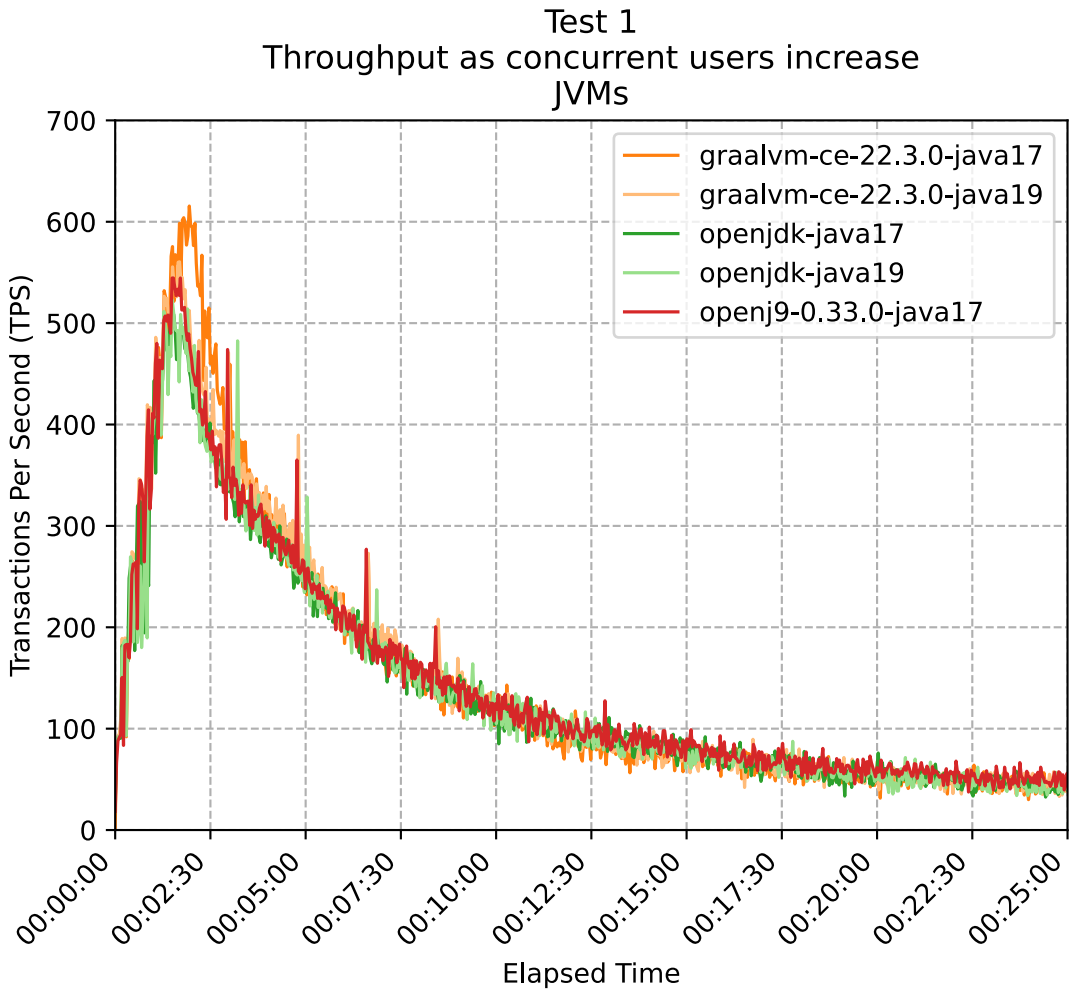


Figura 3.3: Throughput nel tempo (JVMs)

Il grafico in Figura 3.4 mostra invece l'andamento del throughput delle versioni native del Toy System all'aumentare del numero di utenti concorrenti. La versione dell'immagine nativa usando il GC seriale (single-threaded) non offre un throughput elevato e ha un comportamento instabile quando il sistema è sotto stress dovuto alle attività del GC seriale. Si osserva invece che l'immagine nativa usando il GC G1 (multi-threaded) offre un throughput molto simile a HotSpot/OpenJ9.

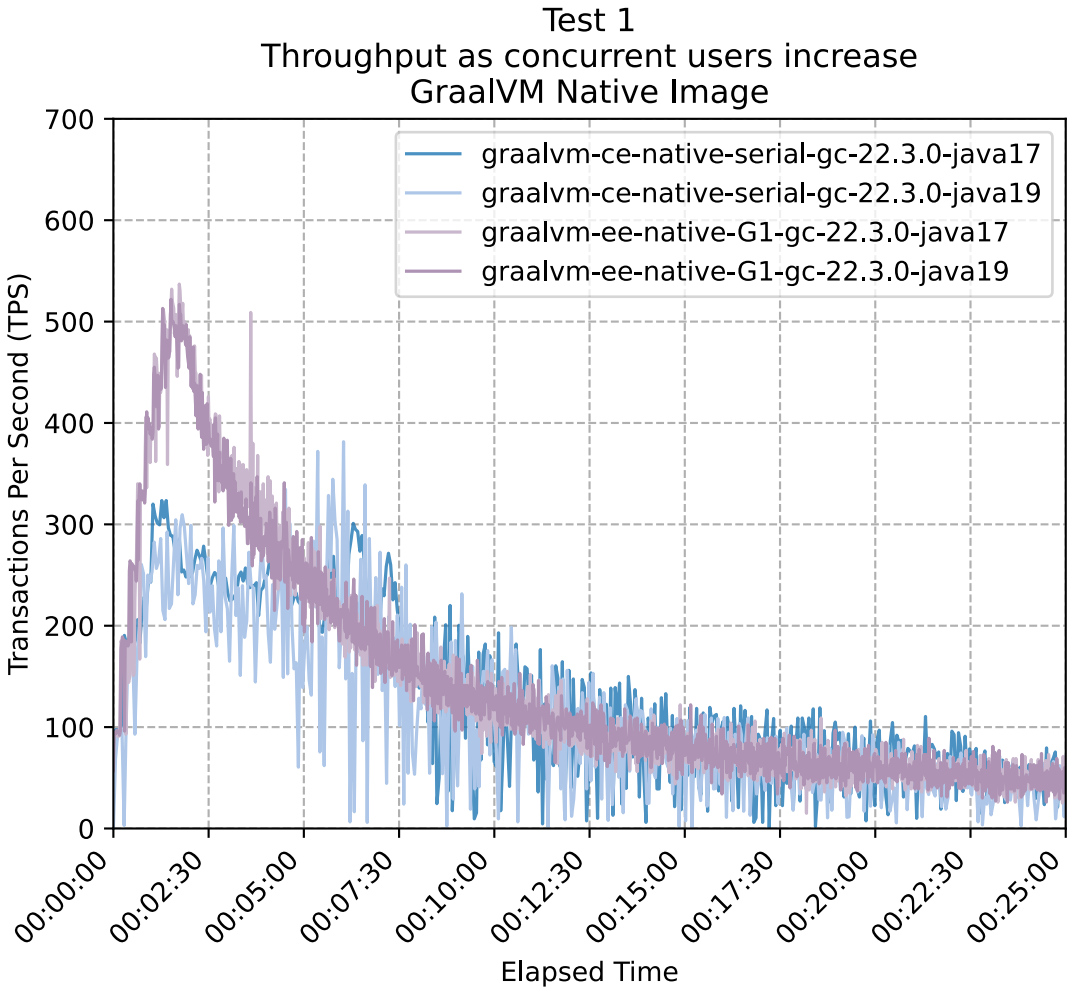


Figura 3.4: Throughput nel tempo (GraalVM Native Image)

Test 1 (Stress Test) - Uso della CPU

La Figura 3.5 mostra l'uso della CPU all'aumentare del numero di utenti concorrenti. Dal grafico si osserva che tutte le JVM sfruttano al massimo le capacità computazionali del server quando il sistema è sotto stress e non si notano particolari differenze tra i vari ambienti di esecuzione. Il grafico è in linea con quelli visti nelle sezioni precedenti: esso evidenza che il punto in cui le performance iniziano a degradare è lo stesso punto in cui l'uso della CPU è al massimo.

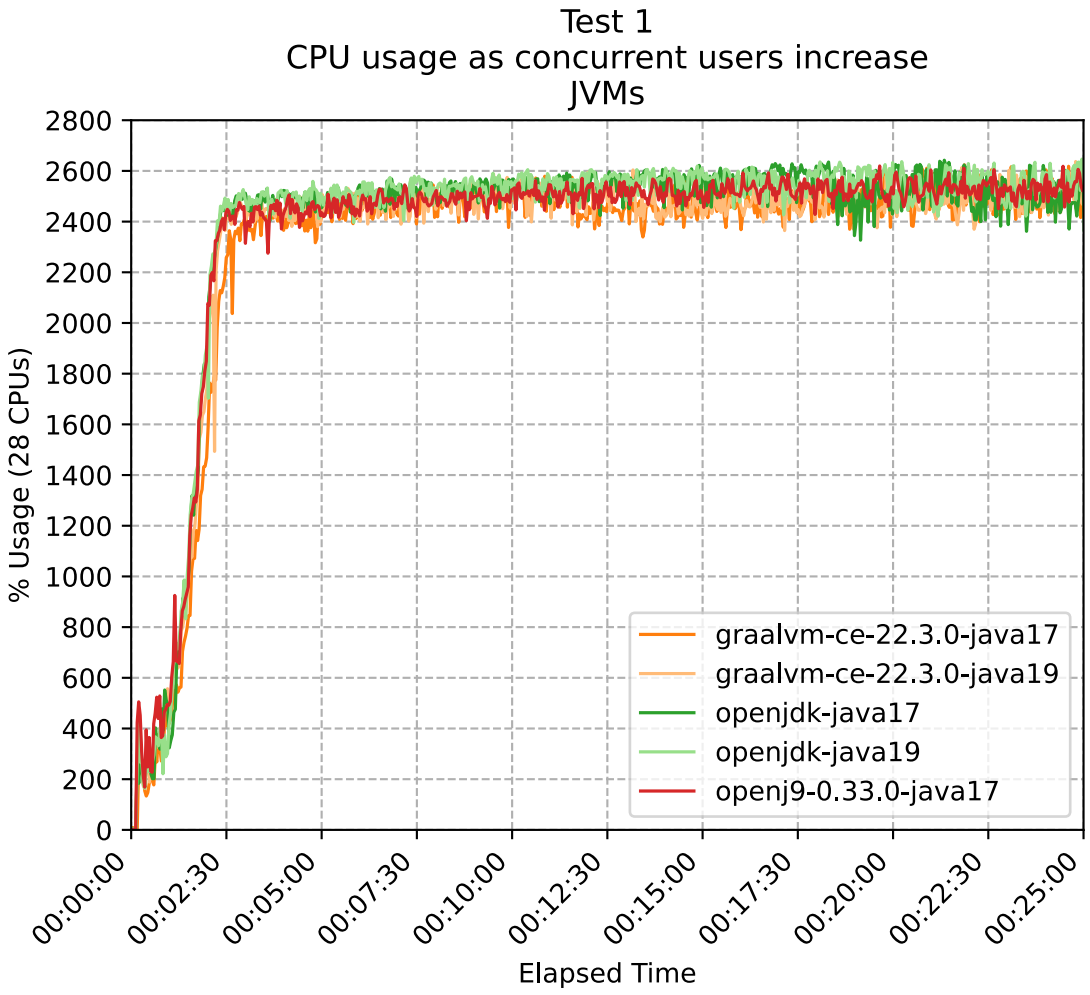


Figura 3.5: Utilizzo della CPU nel tempo (JVMs)

Il grafico in Figura 3.6 evidenza che la versione nativa del Toy System con un GC seriale ha un comportamento molto oscillante. Ciò è dovuto al fatto che il GC seriale utilizza un solo thread per eseguire le sue operazioni. Tali operazioni avvengono frequentemente dato che la zona dell'heap su cui è possibile allocare nuovi oggetti (utilizzando un garbage collector seriale) è molto piccola e la sua saturazione avviene molto spesso essendo il sistema sottoposto a carichi elevati. Mentre il garbage collector seriale non è pensato per heap di grandi dimensioni e utilizza un solo thread per le sue operazioni, il GC G1 utilizza la maggior parte dei thread hardware per eseguire le proprie azioni. Infatti, da come si evince dal grafico, l'uso della CPU dell'immagine nativa con il GC G1 è molto simile a quello di una JVM.

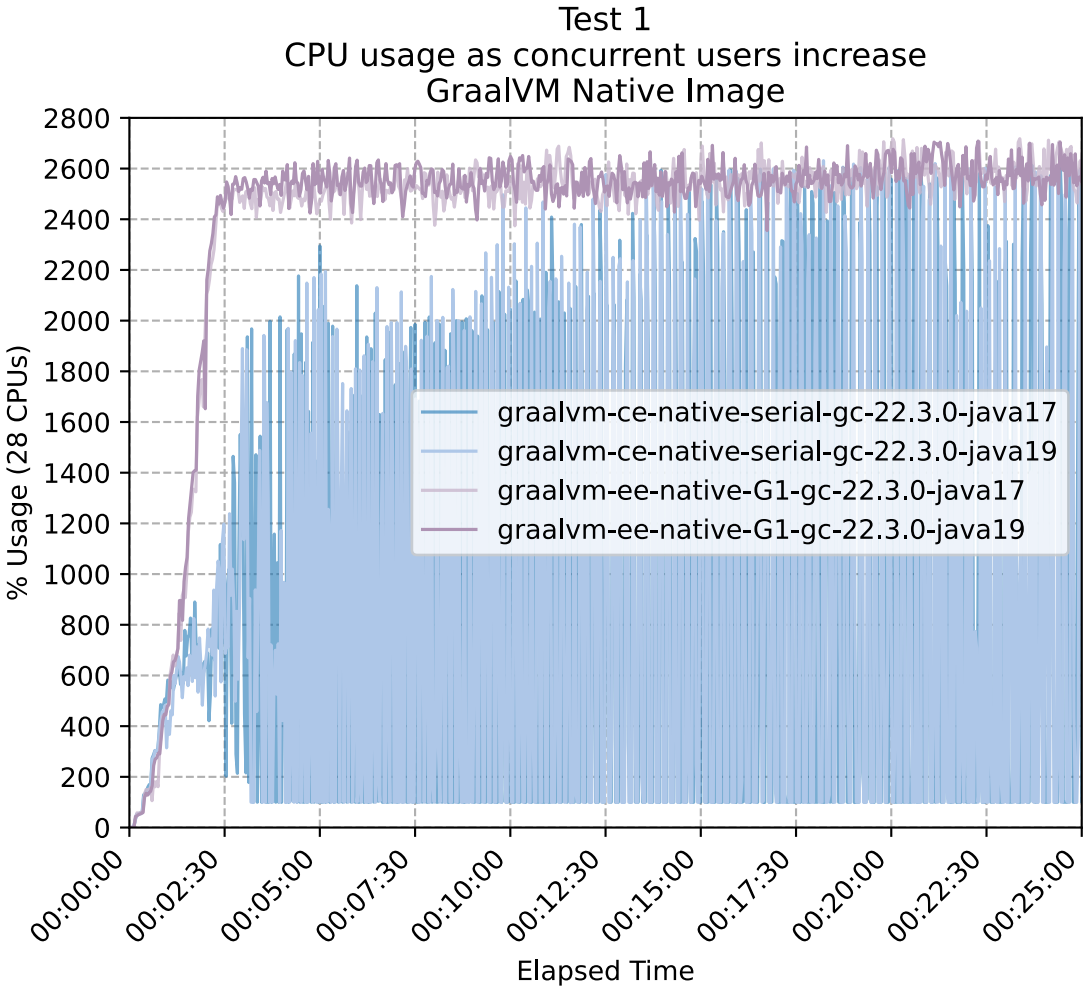


Figura 3.6: Utilizzo della CPU nel tempo (GraalVM Native Image)

Test 1 (Stress Test) - Utilizzo della memoria

Dal grafico in Figura 3.7, si osserva che GraalVM arriva ad utilizzare l'intero heap prima di tutte le altre JVM. Si osserva inoltre che OpenJ9 ha un footprint di memoria molto più basso rispetto a OpenJDK e GraalVM. Ciò a dovuto al garbage collector, in particolare alla Generational Concurrent GC policy [30] di OpenJ9.

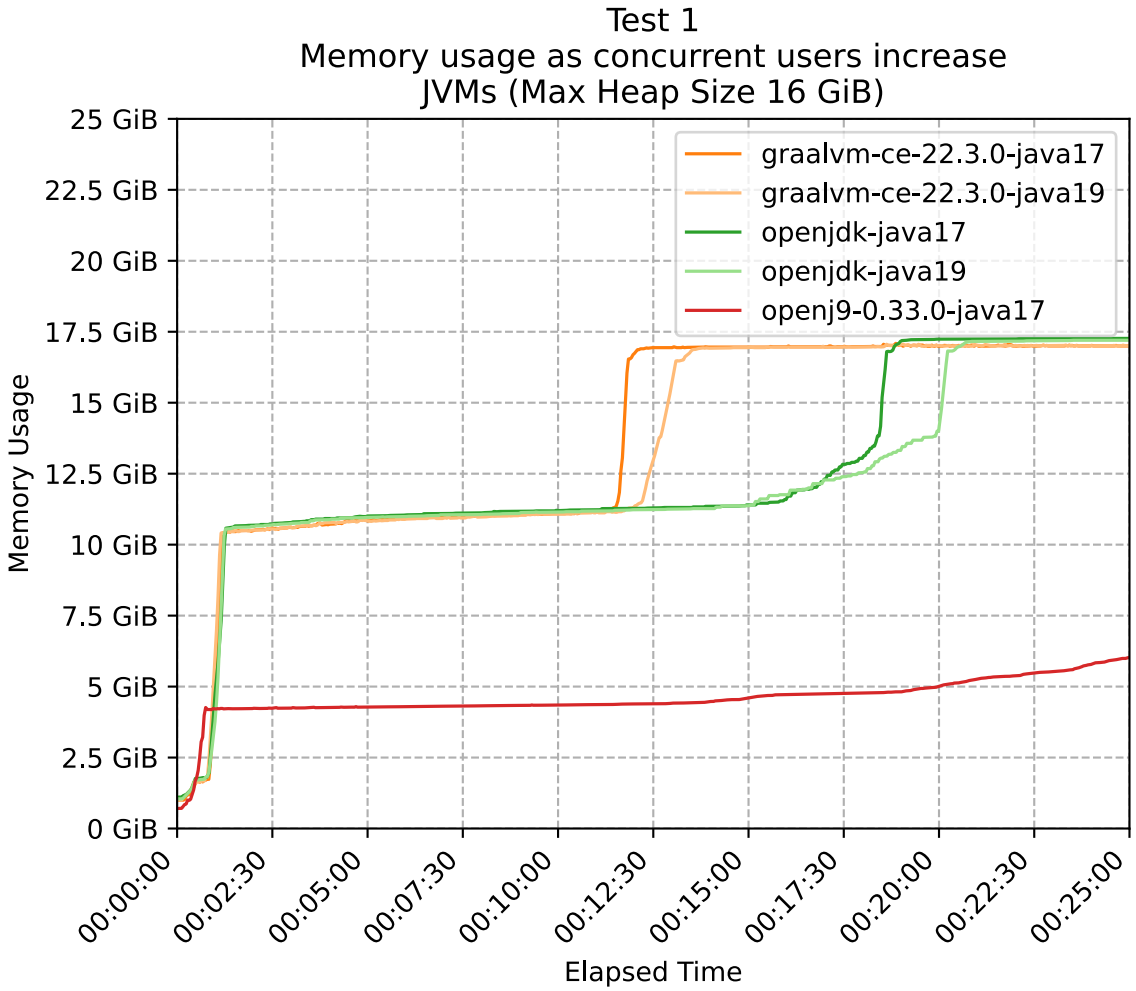


Figura 3.7: Utilizzo della memoria nel tempo (JVMs)

Il grafico in Figura 3.8 mostra l'uso della memoria delle versioni native dell'applicazione. Dal grafico si evince che, rispetto alle JVM, le immagini native hanno un consumo di memoria ridotto e questo è vero in particolare per la versione nativa con il garbage collector seriale la quale, anche qui, presenta un comportamento oscillante dovuto al garbage collector stesso.

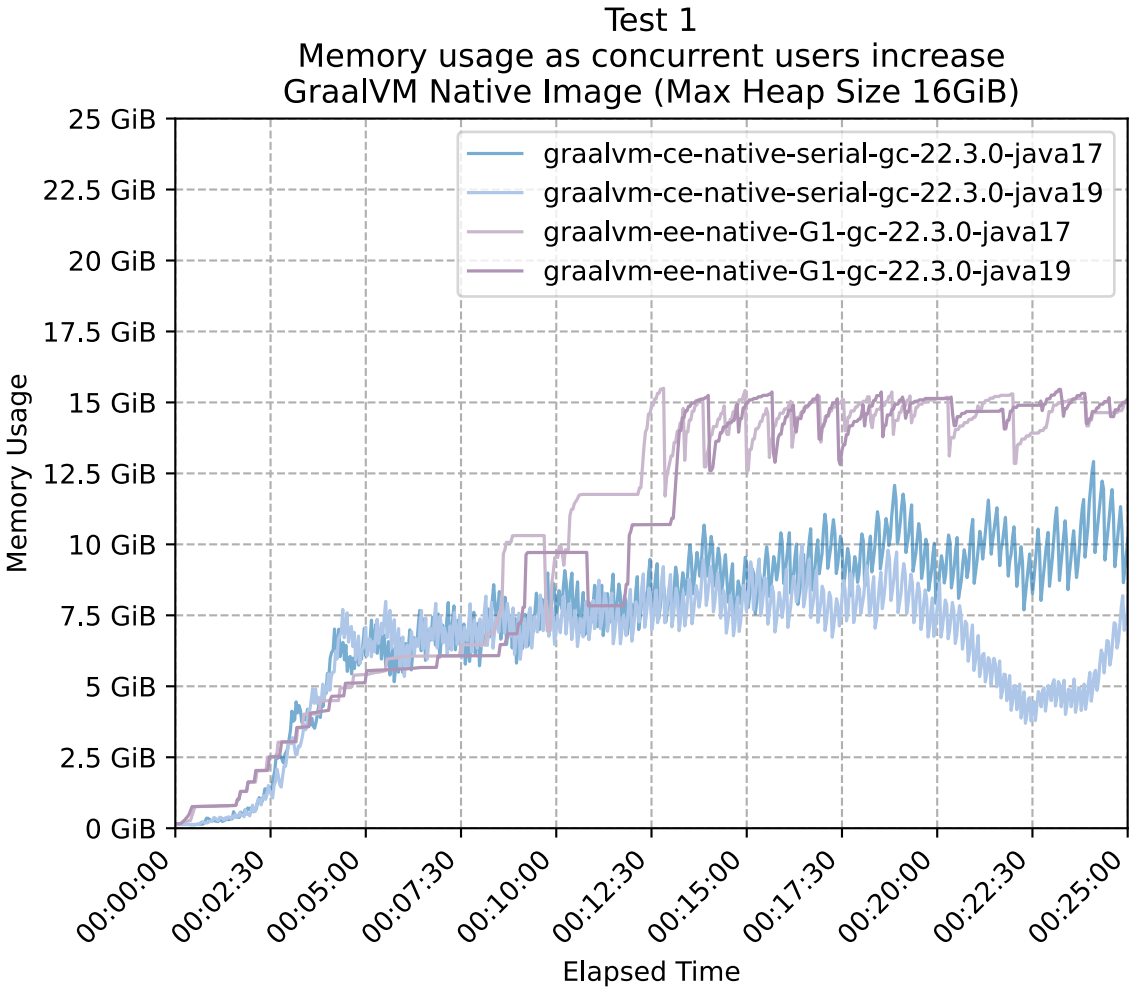


Figura 3.8: Utilizzo della memoria nel tempo (GraalVM Native Image)

Test 2 (Carico sostenibile) - Throughput

Il grafico in Figura 3.9 mostra l'andamento del throughput quando il sistema è sottoposto a un carico sostenibile e costante nel tempo. Si osserva che GraalVM offre in media un throughput più alto mentre OpenJ9 offre un throughput nel tempo non proprio stabile dovuto alle attività del suo garbage collector.

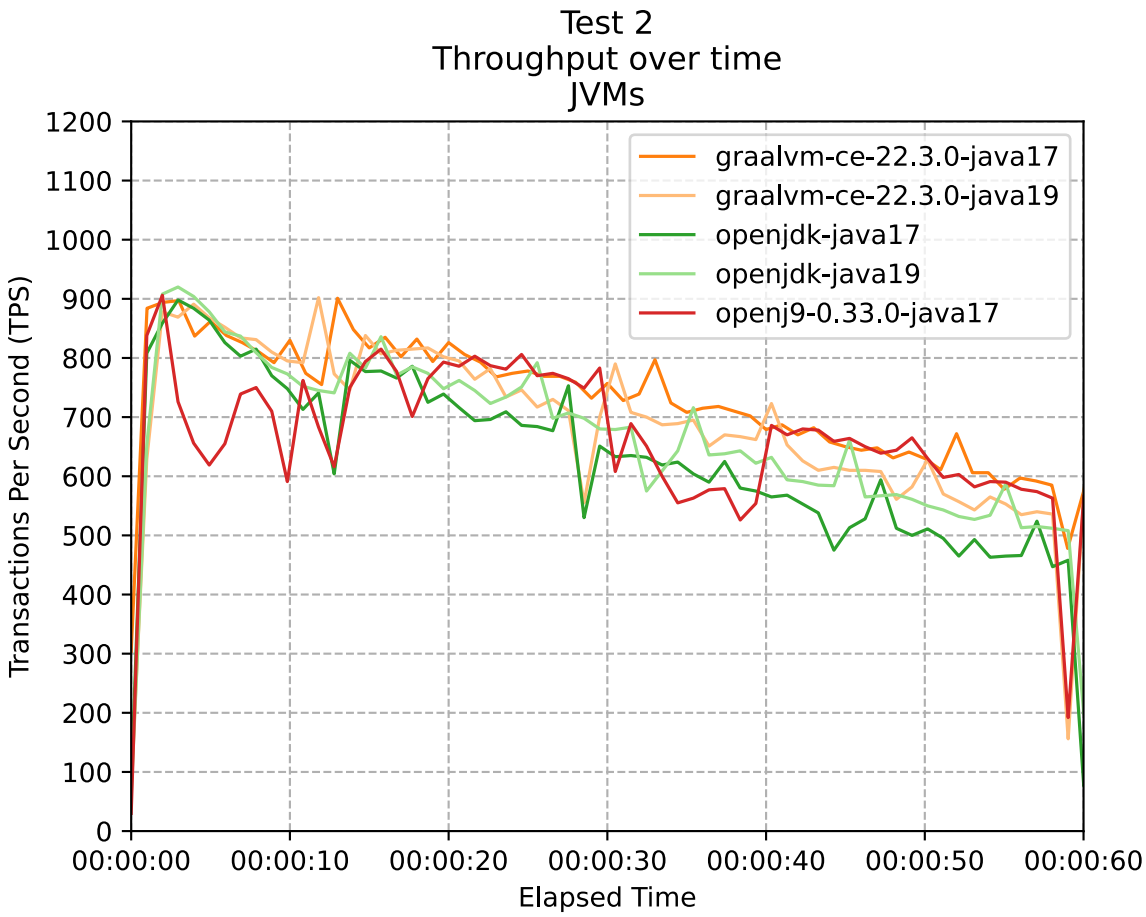


Figura 3.9: Throughput nel tempo (JVMs)

Da come si può notare dal grafico in Figura 3.10, l'immagine nativa con un garbage collector seriale offre un throughput elevato immediatamente: a differenza delle JVM, il tempo di warm up è nullo. D'altro canto, l'immagina nativa con un GC G1 offre un throughput elevato nel tempo, ma richiede, nella fase iniziale, qualche secondo prima di poter raggiungere il picco.

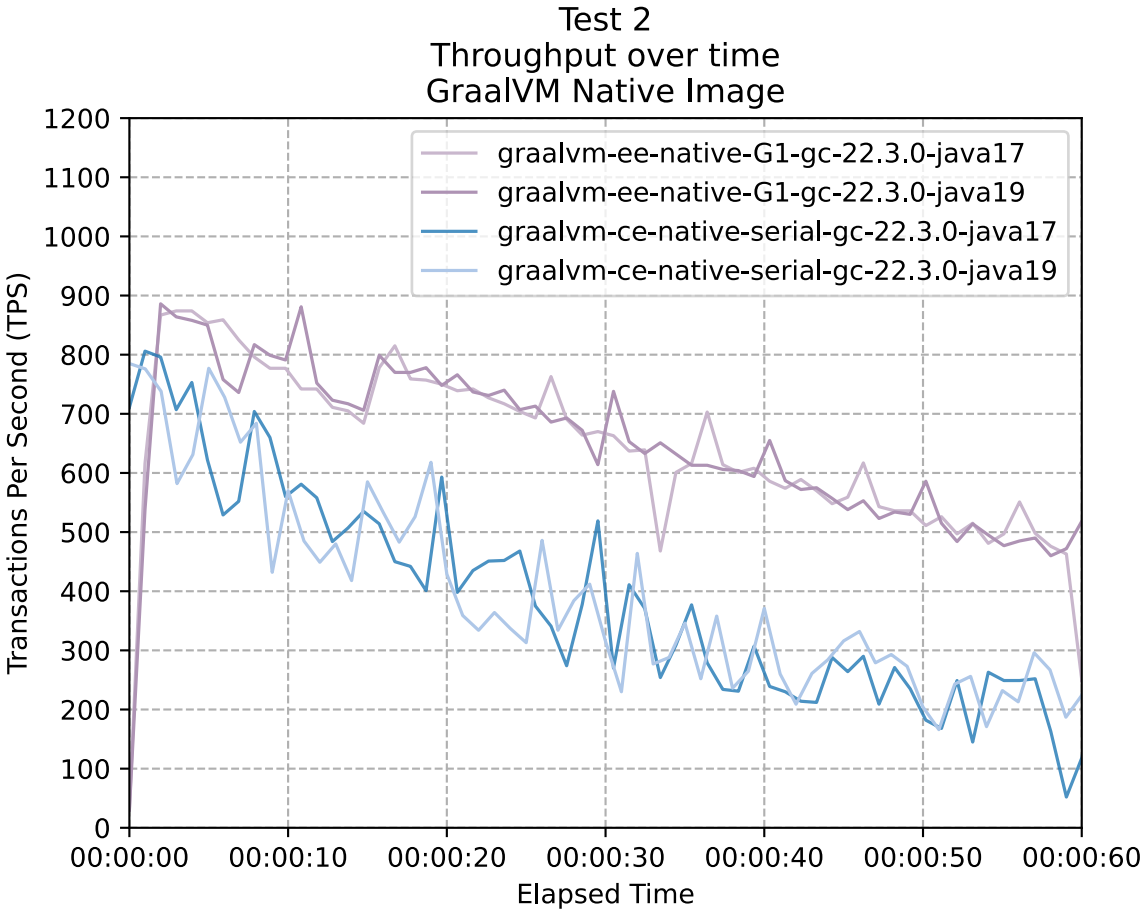


Figura 3.10: Throughput nel tempo (GraalVM Native Image)

Test 2 (Carico sostenibile) - Utilizzo della CPU

Dal grafico in Figura 3.11, si osserva che OpenJDK utilizza più potenza di calcolo per raggiungere, rispetto alle altre JVM, un throughput avvolte anche più basso e quindi risulta essere meno efficiente. OpenJ9 presenta un comportamento non sempre stabile dovuto alle attività del suo garbage collector, mentre GraalVM presenta un consumo di CPU in generale più basso di OpenJDK e di OpenJ9 nelle prime fasi del test e ha un comportamento stabile nel tempo.

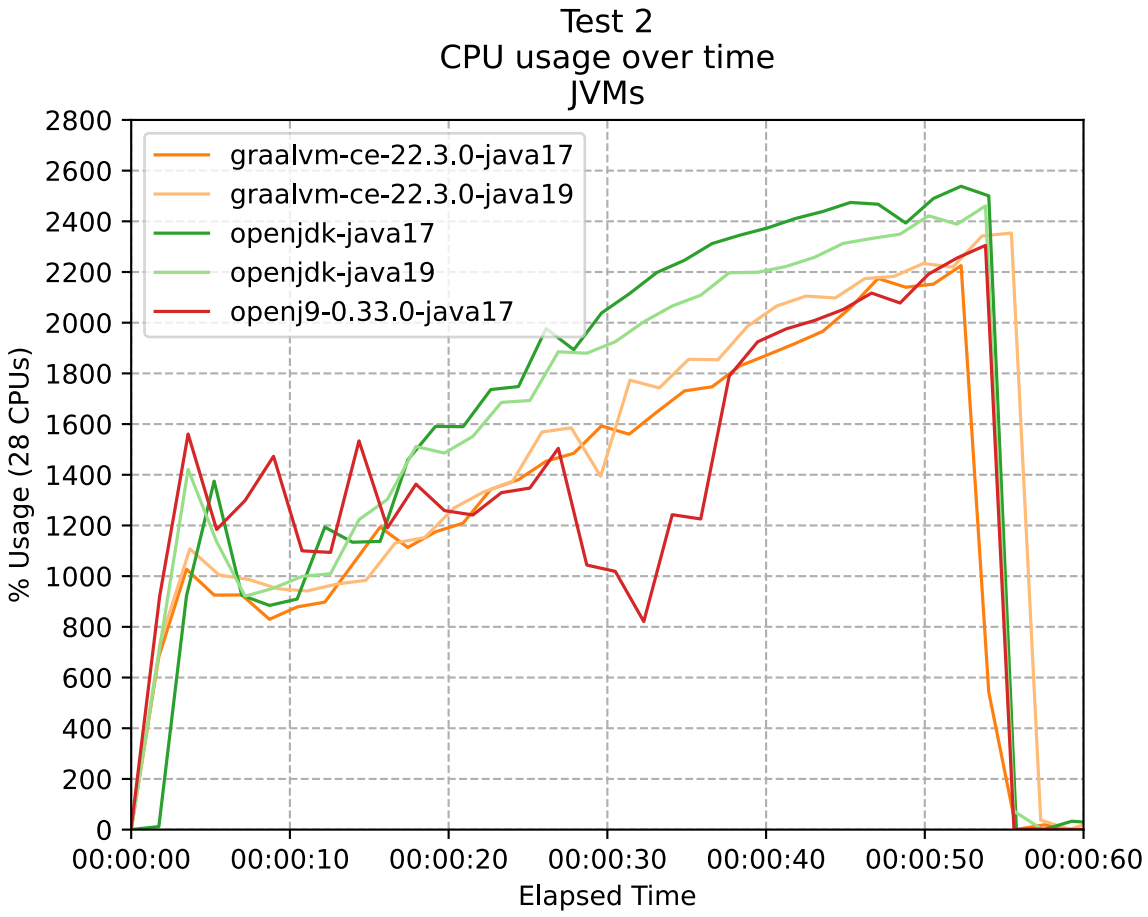


Figura 3.11: Utilizzo della CPU nel tempo (JVMs)

Quando il sistema è sottoposto a un carico sostenibile, l'immagine nativa con un garbage collector seriale non richiede molte risorse computazionali. Da come si può osservare nel grafico in Figura 3.12, l'immagine native con un GC G1, al contrario, ha bisogno di più risorse computazionali per via delle attività del garbage collector multi-threaded per garantire un throughput elevato.

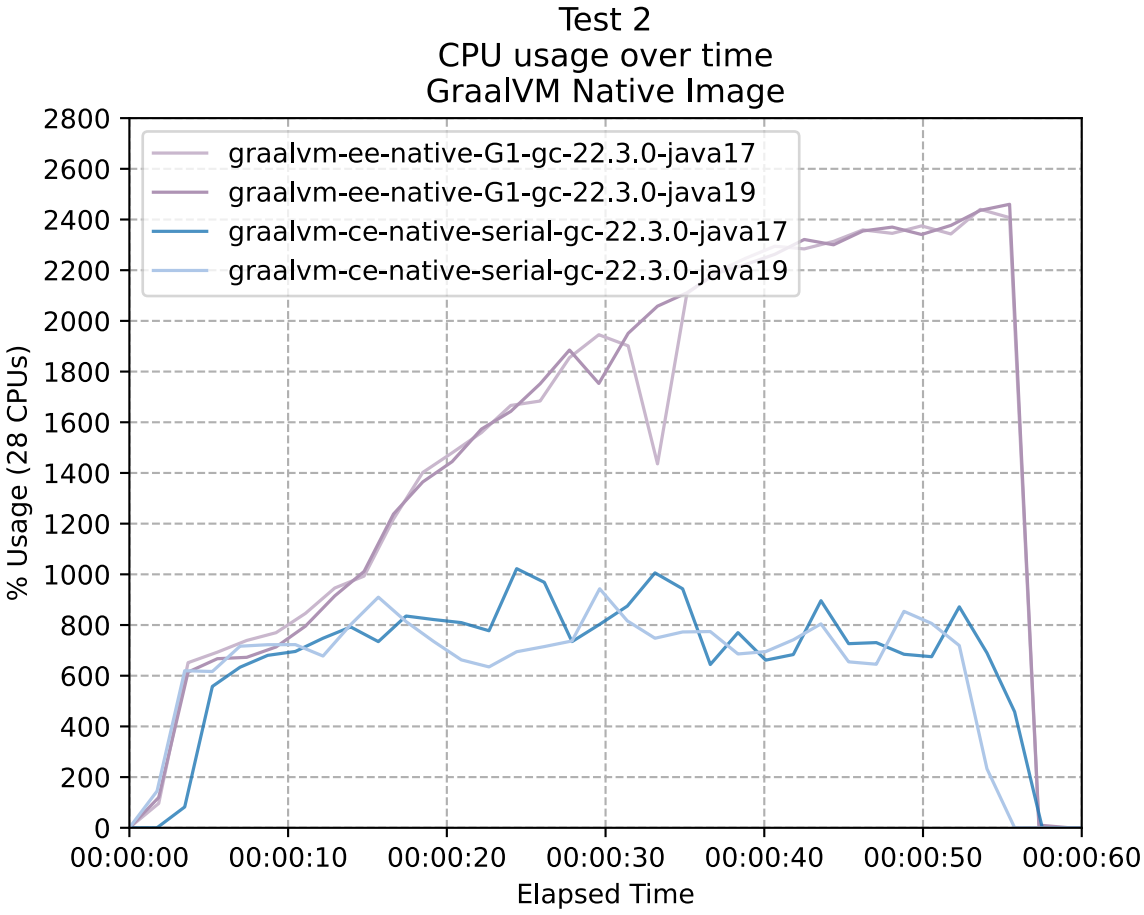


Figura 3.12: Utilizzo della CPU nel tempo (GraalVM Native Image)

Test 2 (Carico sostenibile) - Utilizzo della memoria

Ancora una volta, quando il sistema è sottoposto ad un carico sostenibile, OpenJ9 ha un footprint di memoria decisamente più basso (grafico in Figura 3.13), mentre il punto in cui le altre JVM raggiungono il massimo utilizzo della memoria heap è pressoché uguale. Anche se OpenJ9 offre un consumo di memoria molto basso dovuto al suo garbage collector, d'altro canto, come visto nei grafici precedenti, esso presenta un comportamento poco stabile per quanto riguarda il throughput e l'utilizzo delle risorse di calcolo.

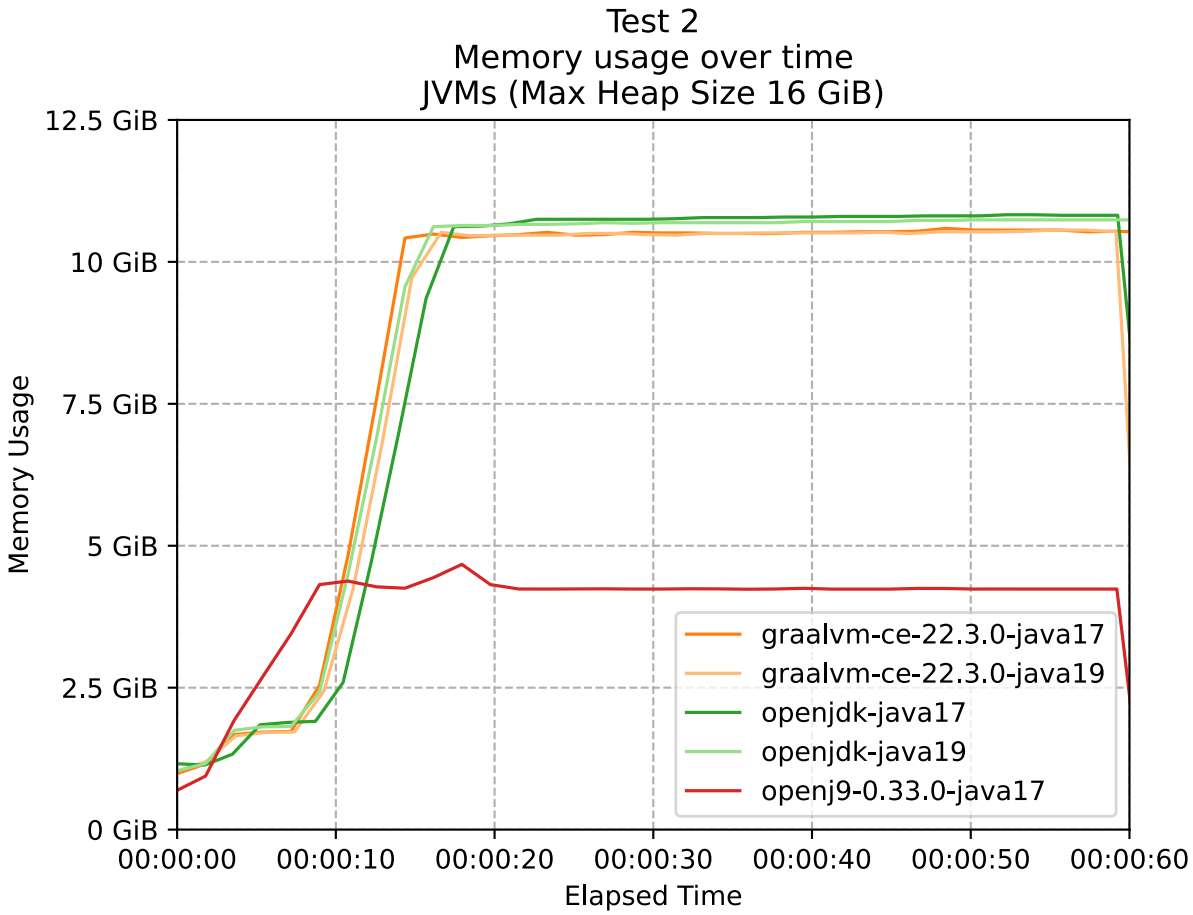


Figura 3.13: Utilizzo della memoria nel tempo (JVMs)

Infine, a fronte di un carico sostenibile, la native image del Toy System, da come si può osservare dal grafico in Figura 3.14, ha un consumo di memoria nel tempo estremamente più basso rispetto alle JVM indipendentemente dal tipo di garbage collector utilizzato (seriale oppure G1). Questo è dovuto principalmente dall'assenza di un intero processo JVM che esegue a tempo di esecuzione e di conseguenza dall'assenza di un compilatore JIT e di un garbage collector sofisticato che competono entrambi insieme al processo dell'applicazione per le risorse.

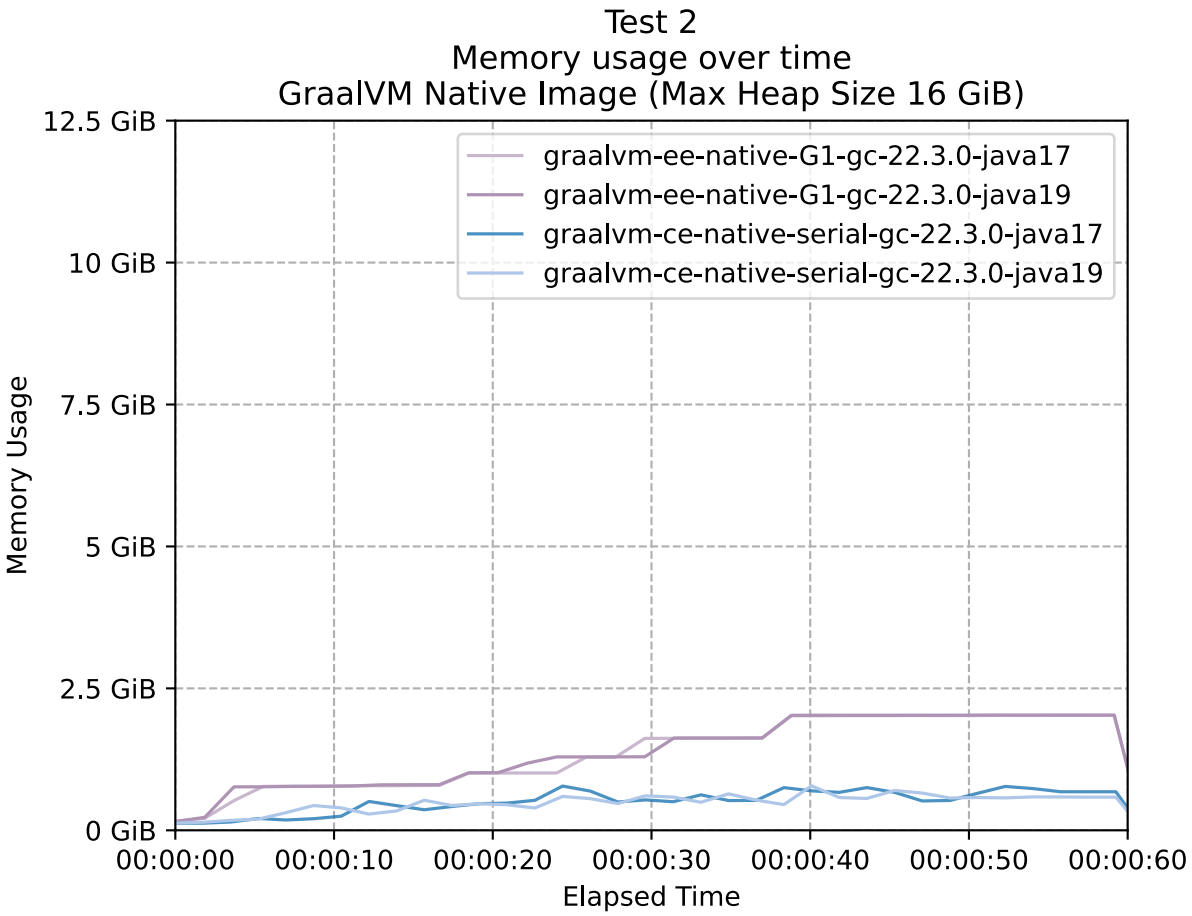


Figura 3.14: Utilizzo della memoria nel tempo (GraalVM Native Image)

–4–
Conclusioni

Il linguaggio di programmazione Java è stato progettato in modo che le applicazioni potessero funzionare ovunque e per lungo tempo, massimizzando il throughput a scapito del consumo di memoria. Negli ambienti cloud, il concetto di application server è messo in difficoltà dalla natura intrinsecamente distribuita del cloud computing. Le applicazioni sono ospitate sulle piattaforme cloud e distribuite su più macchine. Affinché tutto ciò sia possibile, le applicazioni devono essere progettate per essere leggere, modulari e portabili, ideali per gli ambienti cloud. In questo scenario, non c'è bisogno di hardware massiccio on-premise. Eliminare l'esigenza di un costoso hardware on-premise è uno dei maggiori benefici degli ambienti cloud. Java non è stato originariamente pensato per il cloud computing e deve evolvere per adattarsi a questo nuovo ambiente distribuito. In questa tesi sono stati discussi nuovi ambienti di esecuzione per le applicazioni Java. Sono stati valutati i costi e i benefici di questi runtime applicandoli su un sistema web in miniatura di come sono quelli enterprise large-scale, rendendo i risultati facilmente trasferibili a sistemi reali. I risultati presentati in questa tesi mostrano che le prestazioni di GraalVM hanno superato quelle dei concorrenti in molti aspetti, mentre OpenJ9 grazie al suo garbage collector permette di avere un consumo di memoria basso senza perdere i vantaggi di una JVM. Le immagini native di GraalVM offrono una grande opportunità di eseguire applicazioni Java in container eliminando il consumo di risorse del runtime Java. Inoltre, i risultati mostrano che le immagini native offrono un avvio quasi istantaneo e un overhead di memoria a runtime molto basso. Ciò può essere molto importante per le distribuzioni cloud in cui si desidera autoscalare i servizi o si hanno vincoli di calcolo e memoria, come ad esempio in un ambiente FaaS (Function as a Service). I sistemi software con un'architettura serverless e basati sui microservizi stanno diventando uno standard e probabilmente prenderanno il sopravvento sulla maggior parte delle applicazioni distribuite. Questo stile di progettazione richiede agli sviluppatori di costruire applicazioni in maniera modulare, composta da piccoli componenti che possono comunicare e scalare indipendentemente tra differenti cloud. Nuovi framework emergenti basati sulle JVM e che sfruttano le tecnologie di GraalVM come Quarkus [40] e Micronaut [41] per la costruzione di applicazioni con un'architettura a microservizi e serverless stanno diventano sempre più popolari, mentre il framework Spring Boot 3 con la sua nuova versione [42] ha introdotto un supporto built-in

per convertire le app Spring in immagini native usando GraalVM. Le immagini native di GraalVM stanno diventando e diventeranno il nuovo modo di costruire e distribuire applicazioni basate sulle JVM in ambienti cloud.

Bibliografia/Sitografia

[1] Martin Fowler Mike Roberts. *Serverless Architecture*. Ultimo accesso in data 14 Gennaio 2023. URL: https://martinfowler.com/articles/serverless.html.

[2] GitHub. *Toy System GitHub Repository*. Ultimo accesso in data 9 Febbraio 2023. URL: https://github.com/vtramo/sportsclub-restapi-toysystem.

[3] Roy Thomas Fielding. *Representational State Transfer (REST)*. Ultimo accesso in data 10 Gennaio 2023. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[4] Roy T. Fielding. *Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture"*. Ultimo accesso in data 06 Gennaio 2023. URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/28279b2a527287a42501f4c4122ef7c09b85a36f.pdf.

[5] Fastly J. Reschke R. Fielding M. Nottingham. *RFC 9110 HTTP Semantics*. Ultimo accesso in data 10 Gennaio 2023. URL: https://www.rfc-editor.org/rfc/rfc9110.html.

[6] Roy T. Fielding. *Uniform Resource Identifier (URI)*. Ultimo accesso in data 06 Gennaio 2023. URL: https://www.rfc-editor.org/rfc/rfc3986.

[7] Wikipedia. *Convention Over Configuration Design Paradigm*. Ultimo accesso in data 10 Gennaio 2023. URL: https://en.wikipedia.org/wiki/Convention_over_configuration.

[8] Martin Fowler. *Continuous Integration*. Ultimo accesso in data 14 Gennaio 2023. URL: https://martinfowler.com/article/continuousIntegration.html.

[9] Maven Repository. *Maven Central Repository*. Ultimo accesso in data 10 Gennaio 2023. URL: https://mvnrepository.com/repos/central.

[10] OpenAPI Initiative. Ultimo accesso in data 10 Gennaio 2023. URL: https://www.openapis.org/.

[11] OpenAPI Initiative. *OpenAPI Specification*. Ultimo accesso in data 10 Gennaio 2023. URL: https://github.com/OAI/OpenAPI-Specification.

[12] Swagger. *Swagger Tools*. Ultimo accesso in data 10 Gennaio 2023. URL: https://swagger.io/tools/.

[13] The Official YAML Web Site. *YAML*. Ultimo accesso in data 14 Gennaio 2023. URL: https://yaml.org/.

[14] Open Container Initiative. *Open Container Initiative (OCI)*. Ultimo accesso in data 14 Gennaio 2023. URL: https://opencontainers.org/.

[15] Docker Documentation. *Docker Overview*. Ultimo accesso in data 10 Gennaio 2023. URL: https://docs.docker.com/get-started/.

[16] Docker Documentation. *Dockerfile*. Ultimo accesso in data 10 Gennaio 2023. URL: https://docs.docker.com/engine/reference/builder/.

[17] Docker Compose Documentation. *Docker Compose Overview*. Ultimo accesso in data 10 Gennaio 2023. URL: https://docs.docker.com/compose/.

[18] Swagger/OpenAPI. *Inheritance and Polymorphism*. Ultimo accesso in data 14 Gennaio 2023. URL: https://swagger.io/docs/specification/data-models/inheritance-and-polymorphism/.

[19] Swagger. *Swagger Codegen*. Ultimo accesso in data 14 Gennaio 2023. URL: https://swagger.io/tools/swagger-codegen/.

[20] Maven. *Maven Archetype*. Ultimo accesso in data 14 Gennaio 2023. URL: https://maven.apache.org/guides/introduction/introduction-to-archetypes.html.

[21] OpenAPI/Swagger. *OpenAPI Generator Plugin*. Ultimo accesso in data 06 Gennaio 2023. URL: https://openapi-generator.tech/docs/plugins/.

[22] Martin Fowler. *Mapper*. Ultimo accesso in data 06 Gennaio 2023. URL: https://martinfowler.com/eaaCatalog/mapper.html.

[23] MapStruct. *MapStruct - Java bean mappings*. Ultimo accesso in data 06 Gennaio 2023. URL: https://mapstruct.org/.

[24] Spring Framework. *Spring Data JPA*. Ultimo accesso in data 14 Gennaio 2023. URL: https://spring.io/projects/spring-data-jpa.

[25] Oracle. *Java Persistance API*. Ultimo accesso in data 14 Gennaio 2023. URL: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

[26] Hibernate - JPA Provider. *Hibernate*. Ultimo accesso in data 06 Gennaio 2023. URL: https://hibernate.org/orm/.

[27] Lombok Java Library. *Lombok features*. Ultimo accesso in data 06 Gennaio 2023. URL: https://projectlombok.org/features/.

[28] Oracle. *Java Persistance Query Language*. Ultimo accesso in data 14 Gennaio 2023. URL: https://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html.

[29] Eclipse Foundation. *Eclipse OpenJ9 Java Virtual Machine*. Ultimo accesso in data 30 Gennaio 2023. URL: https://www.eclipse.org/openj9/docs/introduction/.

[30] Eclipse Foundation. *OpenJ9 Generational Concurrent GC policy*. Ultimo accesso in data 30 Gennaio 2023. URL: https://www.eclipse.org/openj9/docs/gc/#gencon-policy-default.

[31] Eclipse Foundation. *OpenJ9 GC scavenge operation*. Ultimo accesso in data 30 Gennaio 2023. URL: https://www.eclipse.org/openj9/docs/gc_overview/#gc-scavenge-operation.

[32] GraalVM. *Architecture Overview*. Ultimo accesso in data 1 Febbraio 2023. URL: https://www.graalvm.org/latest/docs/introduction/.

[33] OpenJDK. *JEP 243: Java-Level JVM Compiler Interface*. Ultimo accesso in data 1 Febbraio 2023. URL: https://openjdk.org/jeps/243.

[34] Oracle. *GraalVM Native Image*. Ultimo accesso in data 2 Febbario 2023. URL: https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/enterprise-native-image/.

[35] Oracle. *Graal Reachability Metadata*. Ultimo accesso in data 2 Febbraio 2023. URL: https://github.com/oracle/graal/blob/master/docs/reference-manual/native-image/ReachabilityMetadata.md.

[36] Spring. *Spring Boot GraalVM Native Image Support - Custom Hints*. Ultimo accesso in data 2 Febbraio 2023. URL: https://docs.spring.io/spring-boot/docs/3.0.0/reference/htmlsingle/#native-image.advanced.custom-hints.

[37] Spring. *Spring Boot AOT Engine*. Ultimo accesso in data 2 Febbraio 2023. URL: https://docs.spring.io/spring-framework/docs/6.0.2/reference/html/core.html#core.aot.

[38] GraalVM Native Image. *Memory Management at Image Run Time*. Ultimo accesso in data 4 Febbraio 2023. URL: https://www.graalvm.org/22.0/reference-manual/native-image/MemoryManagement/.

[39] Apache Software Foundation. *Apache JMeter*. Ultimo accesso in data 3 Febbraio 2023. URL: https://jmeter.apache.org/.

[40] Quarkus Framework. *Quarkus*. Ultimo accesso in data 9 Febbraio 2023. URL: https://quarkus.io/.

[41] Micronaut Framework. *Micronaut*. Ultimo accesso in data 9 Febbraio 2023. URL: https://micronaut.io/.

[42] Spring Boot 3.0. *Spring Boot 3.0 Notes*. Ultimo accesso in data 9 Febbraio 2023. URL: https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Release-Notes.

Lista delle Figure

Lista delle Tabelle