Software Architecture Design

Game Repository - Relazione finale

Angelino Pasquale - M63001481
Capasso Giuseppe - M63001498
Dinetti Andrea - M63001496

Glossario

game (oppure **gioco, partita)** è l'entità principale del sistema. Un game è composto da più round.

player (oppure **giocatore)** è l'entità che può eseguire un turno nei round di un game. Un player può essere umano o un robot.

round è l'entità che contiene le informazioni di gioco principali. Ogni game è composto da un numero finito di round durante il quale ciascun giocatore effettua la propria giocata.

turn (oppure **turno)** è l'unità indivisibile di un round e rappresenta le azioni di un giocatore durante il round. Quindi contiene informazioni sull'esito dell'esecuzione dei test ed i relativi risultati nonché le classi generate per risolvere il problema.

Indice

1 Introduzione

Questo elaborato descrive le metodologie, le fasi di sviluppo e gli artefatti prodotti nel corso Software Architecture Design dell'anno 2022/23 per la realizzazione di un task specifico all'interno del progetto ENACTEST. In particolare, il progetto ha l'obiettivo principale di creare un gioco con lo scopo di promuovere le attività di *testing* e che permetta ai partecipanti di cimentarsi in sfide contro sistemi automatizzati ed, eventualmente, altri giocatori.

1.1 Caso di studio

Il caso oggetto di studio durante il corso prevede l'implementazione di un primo scenario di gioco: un giocatore può effettuare una partita contro un singolo avversario robotico testando una singola classe. In particolare, all'atto della creazione della partita, il giocatore sceglie quale *tool* sfidare. In fase preliminare, gli strumenti automatici scelti sono **EvoSuite** e **Randoop**.

1.1.1 Task assegnato

Il task assegnato prevede la progettazione e lo sviluppo dei requisiti relativi al mantenimento delle partite giocate. Di seguito è riportata la formulazione del task:

"Per ogni partita giocata dal giocatore il sistema deve mantenere lo storico di tale partita, memorizzando l'Id del giocatore, il tipo di partita (primo scenario, secondo scenario...) la data e l'ora di inizio e di termine della partita, la classe testata, l'insieme dei casi di test creati e i relativi risultati, nonché il Robot con cui si è giocato ed i casi di test creati dal Robot con i relativi risultati."

1.2 Metodologie adottate

In questa sezione, sono brevemente descritti gli approcci utilizzati durante lo svolgimento del task.

Il lavoro dei team è stato suddiviso in 3 iterazioni, della durata di circa due settimane ciascuna, al termine delle quali ogni gruppo ha presentato , in sessione plenaria, le attività svolte ottenendo dei *feedback*.

Per adeguarsi al ritmo iterativo di sviluppo sono state svolte le seguenti attività:

- Allineamento all'inizio dell'iterazione, per la definizione degli obiettivi e dei task;
- Allineamento alla fine dell'iterazione, per valutare i feedback ricevuti;
- Aggiornamento dei task e della loro assegnazione sul progetto Github (Figura 1);
- Collaborazione durante lo sviluppo con strumenti come Visual Studio Code Share (Pair Programming), e lavagne virtuali condivise;

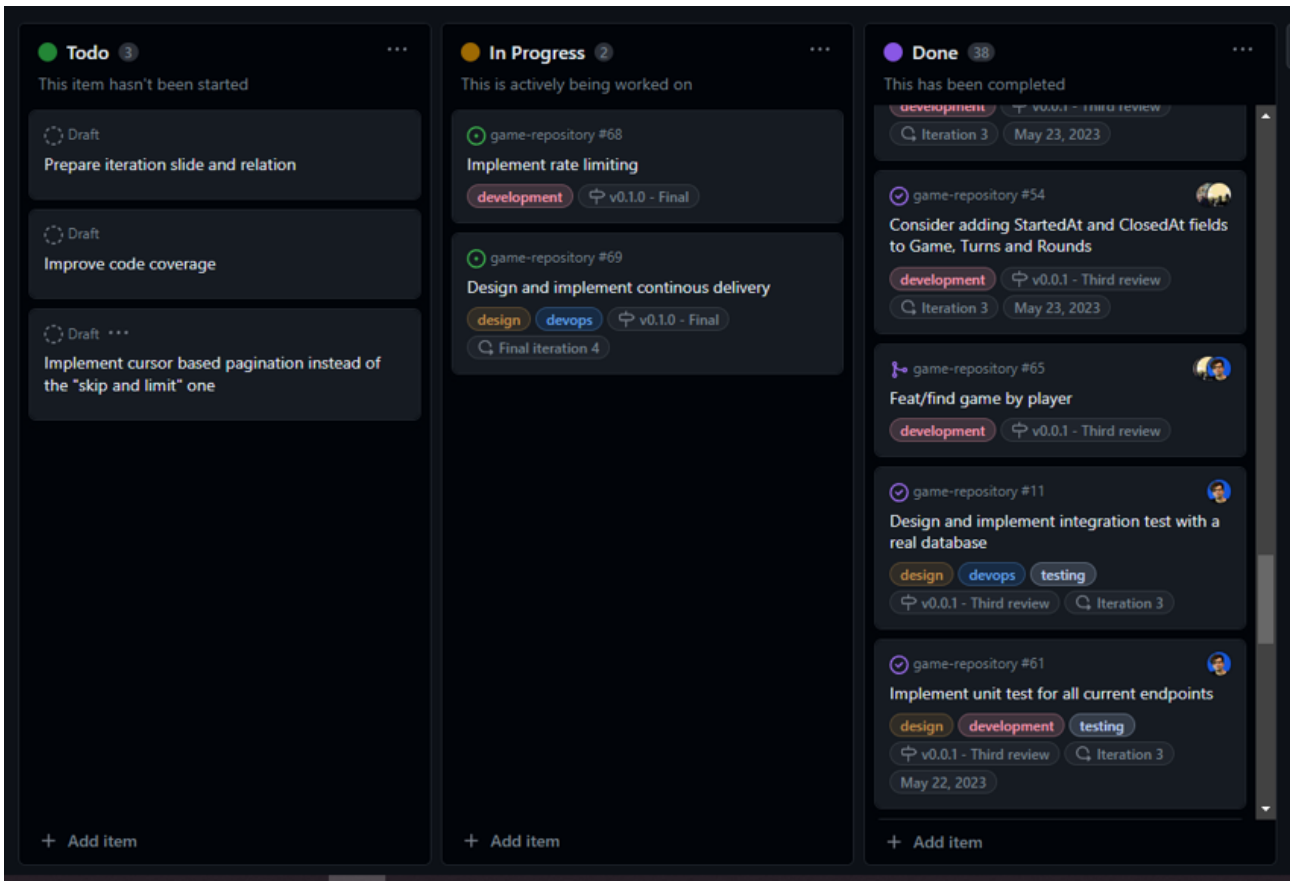

Figura 1: Task del progetto

1.2.1 Link Utili

L'applicazione è disponibile in versione demo su questo **link**.

Una guida dettagliata per l'installazione è presente su questo **link**.

Il codice sorgente prodotto è reperibile al seguente **link**.

Il team si è coordinato utilizzando una *board* condivisa consultabile al **link**.

2 Analisi dei Requisiti

L'analisi del task ha prodotto il diagramma dei casi d'uso mostrati in Figura 2.

Figura 2: Casi d'uso

Sono stati aggiunti, oltre alle operazioni CRUD richieste in partenza, dei casi d'uso che riguardano le elaborazioni statistiche dei dati di gioco.

2.1 Attori

In questa sezione, sono descritti gli attori individuati durante l'analisi dei requisiti. Gli attori presentati saranno poi indicati anche come **utenti** del sistema.

Game Engine Il *Game Engine* è l'utente utilizzatore del servizio che si occupa di implementare le logiche di gioco. Questo utente garantisce che tutti i dati salvati all'interno del *Game Repository* siano coerenti.

2.2 Requisiti Funzionali

In questa sezione, saranno analizzati i casi d'uso per ricavarne dei requisiti funzionali. In particolare, per ogni caso d'uso sarà fornita una storia utente e un tabella riassuntiva.

2.2.1 Gestione delle partite

Una partita (o *game*) è formata da *round* durante ciascuno dei quali ogni giocatore gioca un turno. In particolare, ogni turno prevede l'esecuzione dei test generati e la registrazione dei risultati nel *Game*

Repository. Una partita partita può seguire diversi scenari che indicano le modalità con cui viene svolta (regole, giocatori). Questo caso d'uso genera i seguenti requisiti:

- Creazione partita: quando viene creata la partita, il *Game Repository* registra i partecipanti e lo scenario identificato e crea i round che saranno giocati;
- Cancellazione partita: una partita può essere cancellata insieme a tutti i dati associati ad essa;
- Modifica partita: l'utente può aggiornare lo stato della partita mantenendo l'integrità dei dati;
- Ricerca partita: l'utente può recuperare le informazioni essenziali di una partita a partire dalle quali può ricostruire tutti i round giocati;
- Recupero partite: l'utente, specificando eventualmente dei filtri, può recuperare le partite;

ID	Nome	Descrizione
G-01	Creazione partita	Da utente, si vuole creare una partita specificando i giocatori partecipanti, lo scenario (ovvero, numero di turni, numero di giocatori umani e robot) e la classe da testare.
G-02	Cancellazione partita	Da utente, si vuole cancellare una partita e tutti i dati associati ad essa.
G-03	Modifica partita	Da utente, si vuole modificare una partita in corso senza compromettere l'integrità della stessa.
G-04	Recupero partita	Da utente, si vuole recuperare lo stato di una partita con la possibilità di esplorare i round e i turni di ciascun giocatore
G-05	Recupero partite	Da utente, si vogliono recuperare le partite applicando dei filtri (ad esempio prendere solo quelle terminate o quelle in corso).

Tabella 1: Gestione partite: requisiti funzionali

2.2.2 Gestione dei round

In ogni round, ogni giocatore gioca un turno in cui genera delle classi che risolvono il problema assegnato. In base all'esito del test al giocatore è assegnato un punteggio che dovrà essere salvato nel turno dello stesso. In particolare, ogni round avrà uno e un solo vincitore che sarà deciso dall'utente. Questo caso d'uso genera i seguenti requisiti:

- Creazione round: quando viene creata la partita, il *Game Repository* predispone i round da giocare eventualmente assegnando un tempo limite entro il quale devono essere conclusi o una durata rispetto alla partenza;
- Cancellazione round: un round può essere cancellato dal sistema insieme a tutti i turni associati;
- Modifica round: un round può essere aggiornato dall'utente per modificarne la durata o assegnare il vincitore;
- Ricerca round: l'utente può recuperare lo stato di un round insieme a tutti tutti effettuati dai giocatori;
- Recupero round: l'utente può recuperare tutti i round di una partita;

ID	Nome	Descrizione
G-11	Creazione round	Da utente, si vuole suddividere la partita in round durante il quale ogni giocatore eseguirà il proprio turno.
G-12	Cancellazione round	Da utente, si vuole cancellare un round e tutti i dati degli utenti associati, mantenendo l'integrità dei dati.
G-13	Modifica round	Da utente, si vuole aggiornare lo stato di round modificando impostazioni sulle regole o assegnando un vincitore.
G-14	Recupero round	Da utente, si vuole recuperare un round con le informazioni relative ai turni giocati.
G-15	Recupero round	Da utente, si vogliono recuperare tutti i round di una determinata partita.

Tabella 2: Gestione round: requisiti funzionali

2.2.3 Gestione turni

In ogni turno, un giocatore genere delle classi di test che vengono eseguite dal sistema di testing che genera un punteggio. Il *Game Repository* deve salvare i file prodotti dal giocatore e i punteggi assegnati. Inoltre, l'utente può cancellare o modificare il turno di un giocatore. I requisiti generati sono i seguenti:

- Creazione turno: quando un giocatore sottomette la sua risposta, il sistema salva i file prodotti insieme ai punteggi assegnati;
- Cancellazione turno: l'utente può cancellare il turno di un giocatore;
- Modifica turno: l'utente può modificare il turno di un giocatore;
- Ricerca turno: l'utente può effettuare la ricerca di un turno di un giocatore al fine di ottenere classi generate e punteggi ottenuti;
- Recupero turni: l'utente può recuperare il dettaglio dei turni di un particolare round;

ID	Nome	Descrizione
G-21	Creazione turno	Da utente, si vogliono salvare i file di un giocatore e i relativi punteggi ottenuti da un esecuzione dei test.
G-22	Cancellazione turno	Da utente, si vuole cancellare il turno di un giocatore.
G-23	Modifica turno	Da utente, si vuole aggiornare lo stato di un turno di un giocatore effettuando delle modifiche alle classi generate o ai punteggi ottenuti.
G-24	Recupero turno	Da utente, si vuole recuperare lo stato di un turno di un giocatore inclusi i file creati.
G-25	Recupero turni	Da utente, si vogliono recuperare tutte le informazioni dei turni di un round.

Tabella 3: Gestione turni: requisiti funzionali

2.2.4 Gestione risultati dei Robot

Per ogni classe da testare, o "livello" di una partita, è necessario confrontare i risultati dei giocatori con quelli dei robot, che consistono negli strumenti di testing automatico Randoop e EVOsuite. Piuttosto che calcolare i risultati dei robot durante ogni partita, si è scelto di memorizzarli una volta all'atto della creazione del livello, per poi recuperarli in un secondo momento. Il *Game Repository* deve quindi salvare i punteggi dei robot per ogni classe da testare, potendo distinguere tra i due tipi di tool automatici e associando a ogni risultato un livello di difficoltà. I requisiti generati sono i seguenti:

- Creazione risultati robot: l'utente crea i risultati dei robot specificando la classe testata, il tipo di robot, la difficoltà e i punteggi;
- Cancellazione risultati robot: l'utente può cancellare i risultati di un robot specificando la classe da testare;
- Ricerca risultati robot: l'utente può effettuare la ricerca di un particolare tipo di robot al fine di ottenere i punteggi relativi a una classe da testare per una specifica difficoltà;

ID	Nome	Descrizione
G-31	Creazione risultati robot	Da utente, si vogliono salvare i punteggi di diversi tipi di robot, ottenuti dall'esecuzione dei test, registrando il livello di difficoltà.
G-32	Cancellazione risultati robot	Da utente, si vuole cancellare i punteggi di un robot per una specifica classe testata.
G-33	Ricerca risultati robot	Da utente, si vuole recuperare i punteggi ottenuti da un robot relativi ad una specifica classe e difficoltà.

Tabella 4: Gestione turni: requisiti funzionali

2.3 Requisiti Non Funzionali

In Tabella 5 sono mostrati i requisiti non funzionali individuati.

ID	Nome	Descrizione
NFR-01	Gestione dei file dei giocatori	Quando viene giocato il turno, bisogna conservare i file generati dai giocatori in modo tale da rendere accettabili le *performance* della base dati e da permettere le operazioni di lettura e aggiornamento.
NFR-02	Osservabilità	Il sistema deve poter esporre contatori e metriche utili per analizzare performance ed effettuare attività di ottimizzazione.
NFR-03	Compliance	Il sistema deve esporre un'interfaccia che implementa lo standard Open API in quanto sarà utilizzato da altri servizi.
NFR-04	Paginazione	Quando vengono recuperate più partite, round o turni devono essere previsti dei meccanismi di paginazione che si occupano di limitare la quantità dei dati letta e trasmessa.
NFR-05	Sicurezza	Ogni richiesta che arriva al sistema deve essere autenticata. Nell'ambito specifico di questo progetto, l'autenticazione è gestita da un altro servizio.

Tabella 5: Requisiti non funzionali

3 Progettazione

3.1 Architettura generale del progetto

Fissati i requisiti, si è scelto di analizzare il problema con un approccio ispirato a **Domain Driven Design** in cui si è cercato di progettare le funzionalità dal punto di vista delle entità del sistema. Questa strategia mira a dividere l'architettura iniziale in componenti indipendenti.
In particolare, a valle delle iterazioni, l'architettura iniziale è stata raffinata nel corso delle lezioni fino ad arrivare a quella mostrata in Figura 3:

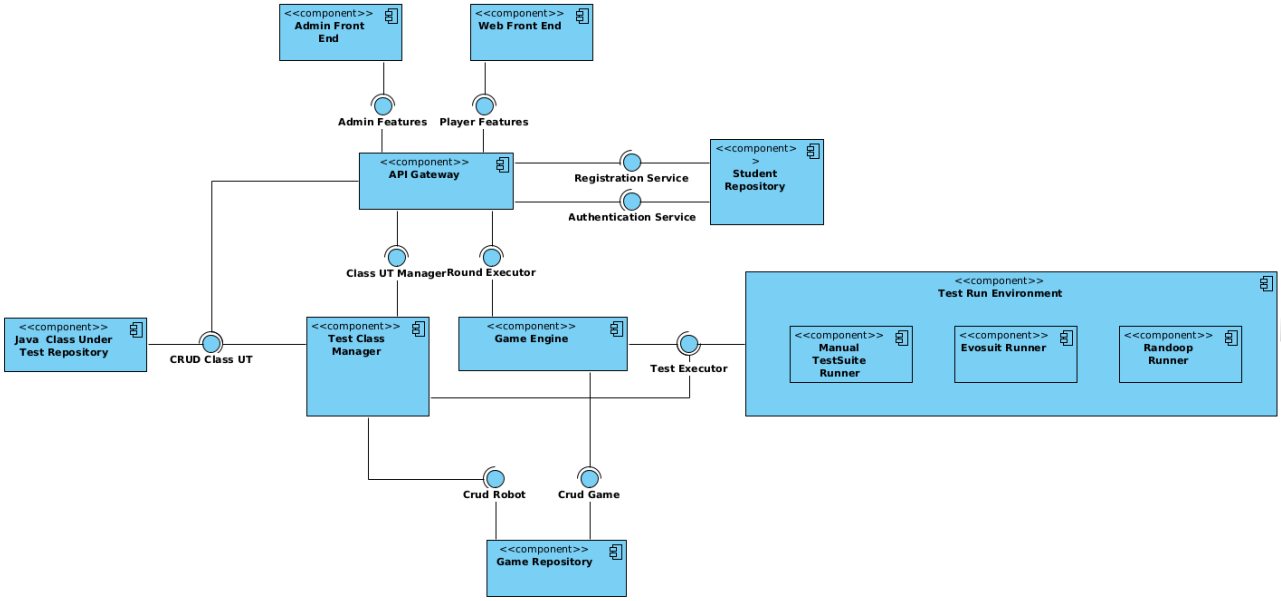

Figura 3: Context diagram

L'architettura mostrata è di tipo *frontend-backend*. In particolare, sono stati previsti due tipi di *frontend* (uno per i giocatori e un altro per l'amministratore). Entrambi, si interfacciano con il componente API gateway che si occupa di indirizzare le richieste ai servizi giusti e di effettuare le operazioni di autenticazione e autorizzazione. Questo componente rappresenta l'unico punto di contatto tra i *frontend* e il *backend*. Di seguito sono descritti gli scenari di operatività della soluzione.

3.1.1 Caricamento delle classi di test

L'amministratore, attraverso il frontend dedicato carica le classi da utilizzare per le partite degli utenti. All'atto della creazione, attraverso il *Test Class Manager* viene eseguito il testing per ogni livello di difficoltà del gioco interagendo con il componente *Test Run Environment*. Una volta generati, i risultati dei test sono salvati nel *Game Repository*; mentre le classi sono salvate nel componente *Java Class Under Test*. Il processo è riassunto nel diagramma di flusso in

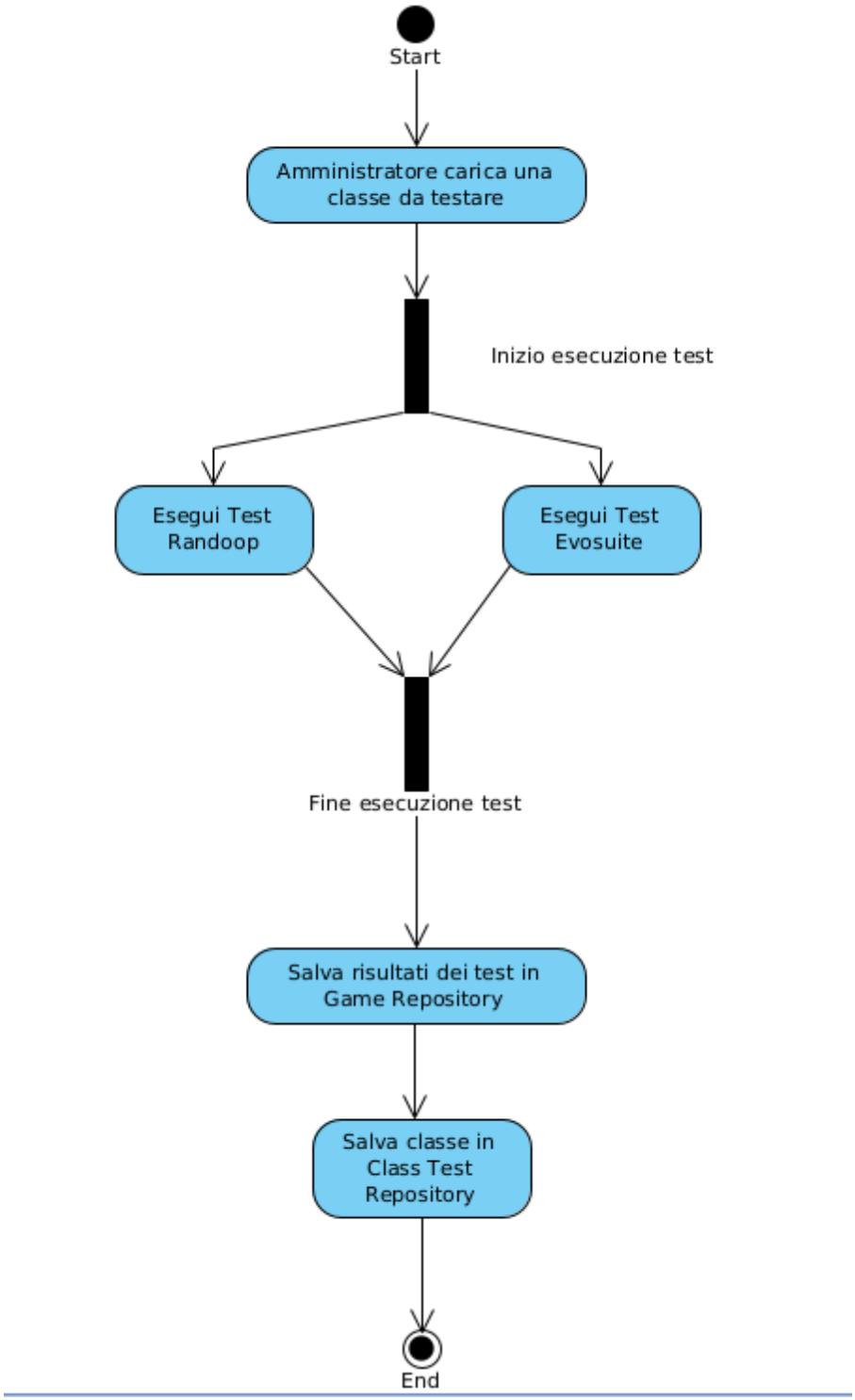

Figura 4: Flusso caricamento classe dall'amministratore

3.1.2 Scenario di gioco

Un giocatore, attraverso il proprio frontend, decide di partecipare ad una partita tra quelle esistenti o, eventualmente, ne può creare una da zero. Ogni partita è composta da Round, con robot da sfidare che svolgono le proprie mosse (turni) in ogni round fino alla fine della stessa. Infine, il *Game Engine* decreterà

un vincitore in base ai punteggi registrati ed eventuali altre metriche configurate. Per i giocatori, il componente di riferimento è il *Game Engine*. Esso si occupa di gestire la logica di gioco preoccupandosi di interagire con l'ambiente di testing per eseguire i dati di input degli utenti e con il *game repository* che si occupa di memorizzare tutte le informazioni riguardanti le partite e i file prodotti dai giocatori. Il processo è riassunto nel diagramma di flusso in Figura 13.

Diagramma ER Di seguito è mostrato il diagramma Entità Relazione del sistema, con una breve descrizione degli attributi e delle relazioni.

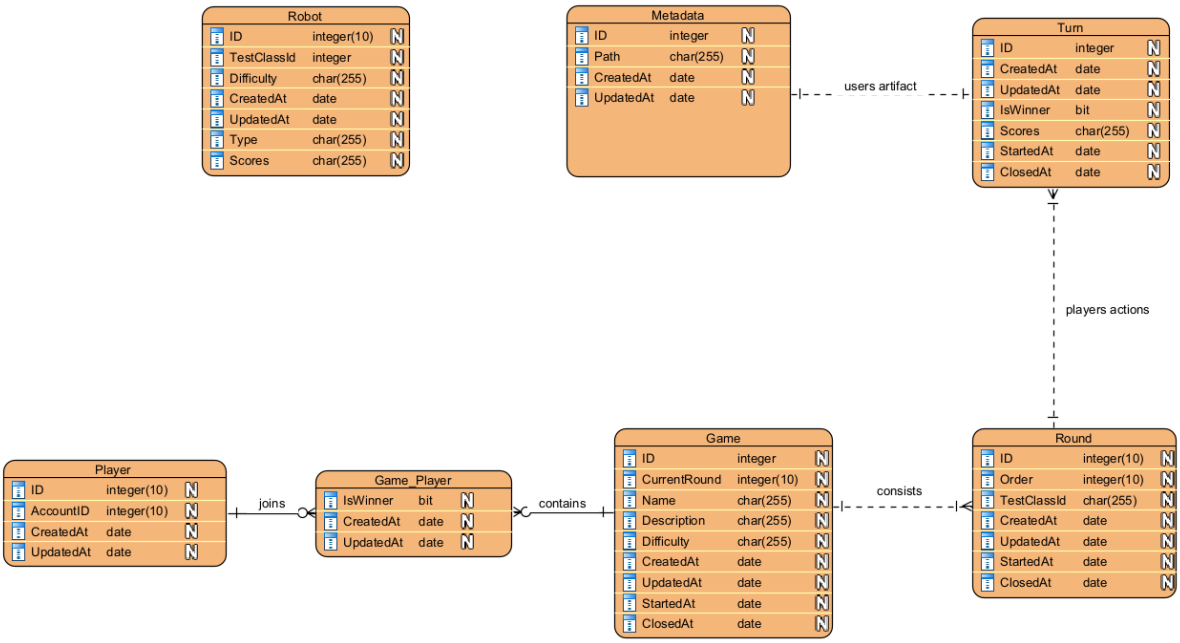

Figura 5: Diagramma ER

Prima di analizzare gli attributi specifici di ogni entità è da notare che tutte possiedono i seguenti campi, utilizzati per tracciare la data di creazione e modifica:

- **CreatedAt**
- **UpdatedAt**

Le entità di cui si deve tracciare lo stato di apertura e chiusura posseggono inoltre i seguenti attributi:

- **StartedAt**
- **ClosedAt**

Di seguito la descrizione degli attributi divisi per entità.

Game

- **ID**: identificativo univoco
- **Current Round**: indica il numero del round corrente
- **Name**: nome della partita
- **Description**: descrizione della partita
- **Difficulty**: grado di difficoltà della partita

Player

- **ID**: identificativo univoco
- **AccountID**: identificativo fornito dal componente Student Repository, che contiene i dati relativi ai giocatori

Game_Player

- **IsWinner**: valore booleano che indica il vincitore finale della partita

Round

- **ID**: identificativo univoco
- **Order**: indica l'ordine dei round all'interno della partita (primo round, secondo ,etc.)
- **TestClassId**: Identificativo della classe da testare presente presente nel Class UT Repository

Turn

- **ID**: identificativo univoco
- **IsWinner**: valore booleano che indica il vincitore del singolo round
- **Scores**: i punteggi relativi al giocatore che ha completato un round, calcolati dal componente di testing e compilazione.

Metadata

- **ID**: identificativo univoco
- **Path**: il percorso relativo nel file system in cui sono memorizzati i file delle classi testate dall'utente.

3.2 Architettura Game Repository

L'architettura del componente è ispirata al modello **Domain Driven Design** (DDD) [7] in cui ogni entità risulta indipendente dalle altre. Come mostrato in Figura 6, ogni elemento logico dell'applicazione (individuato nel diagramma in Figura 5), ha a sua disposizione un *Controller* che utilizza un *Service*. Questi due componenti (realizzati per ogni entità) interagiscono con un ORM che implementa lo schema relazionale sul database.

Si noti che le fasce orizzontali rosa sono tra loro indipendenti e che l'implementazione delle classi *Service* sono astratte da un'interfaccia. Questa scelta, non solo garantisce un basso accoppiamento tra le entità, ma ne favorisce anche la testabilità in quanto fornendo un'implementazione *mock* del service è possibile testare indipendentemente le classi *Controller* e *Service*.

Lo stile Domain Driven è un'estensione concettuale di quello esagonale[7] e può essere utilizzato per implementare microservizi secondo il principio **SOLID**.

I domini applicativi rappresentati in Figura 6 sono stati raffinati nel System Domain Model Figura 7. Si noti che ciascuna entità (in verde) implementa sia il pattern DTO §3.3.3 che Dependency Injection §3.3.2 e che inoltre, sono tra loro indipendenti; invece le relazioni della base dati sono rappresentate nel package *Model* (in rosa).

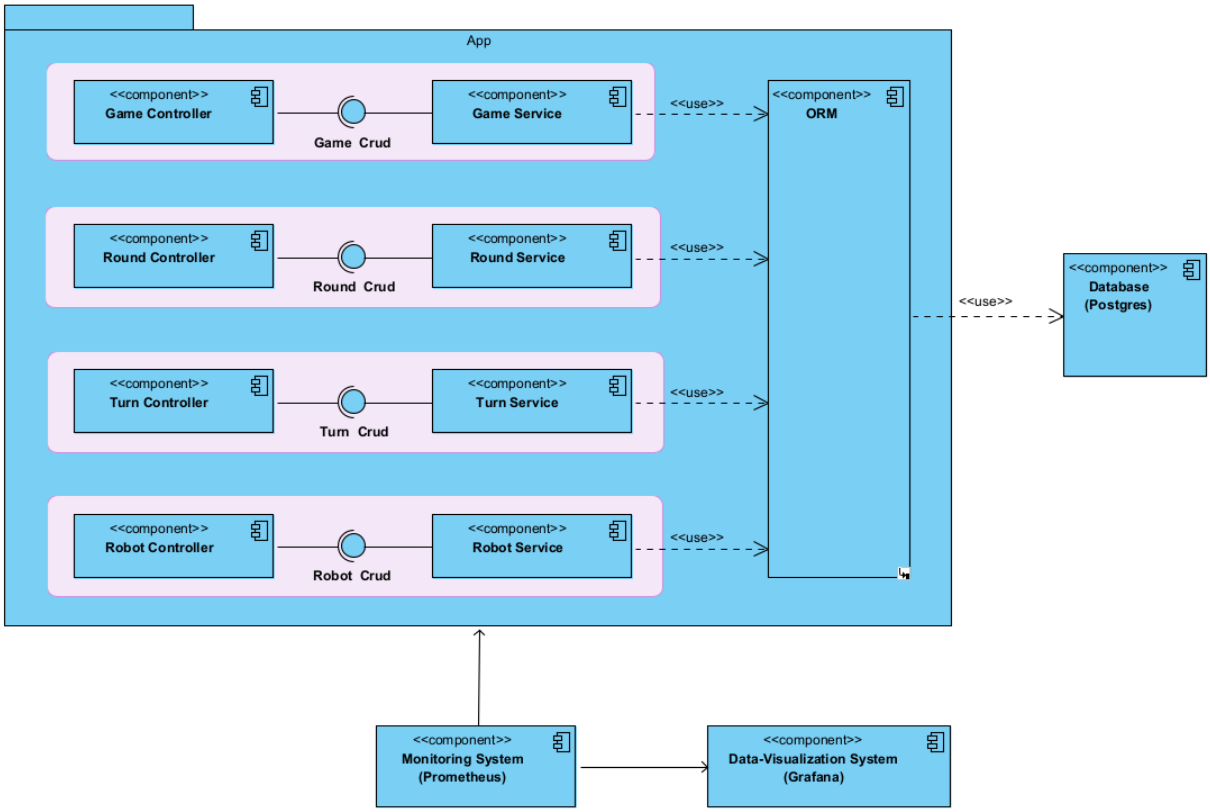

Figura 6: Architettura Game Repository

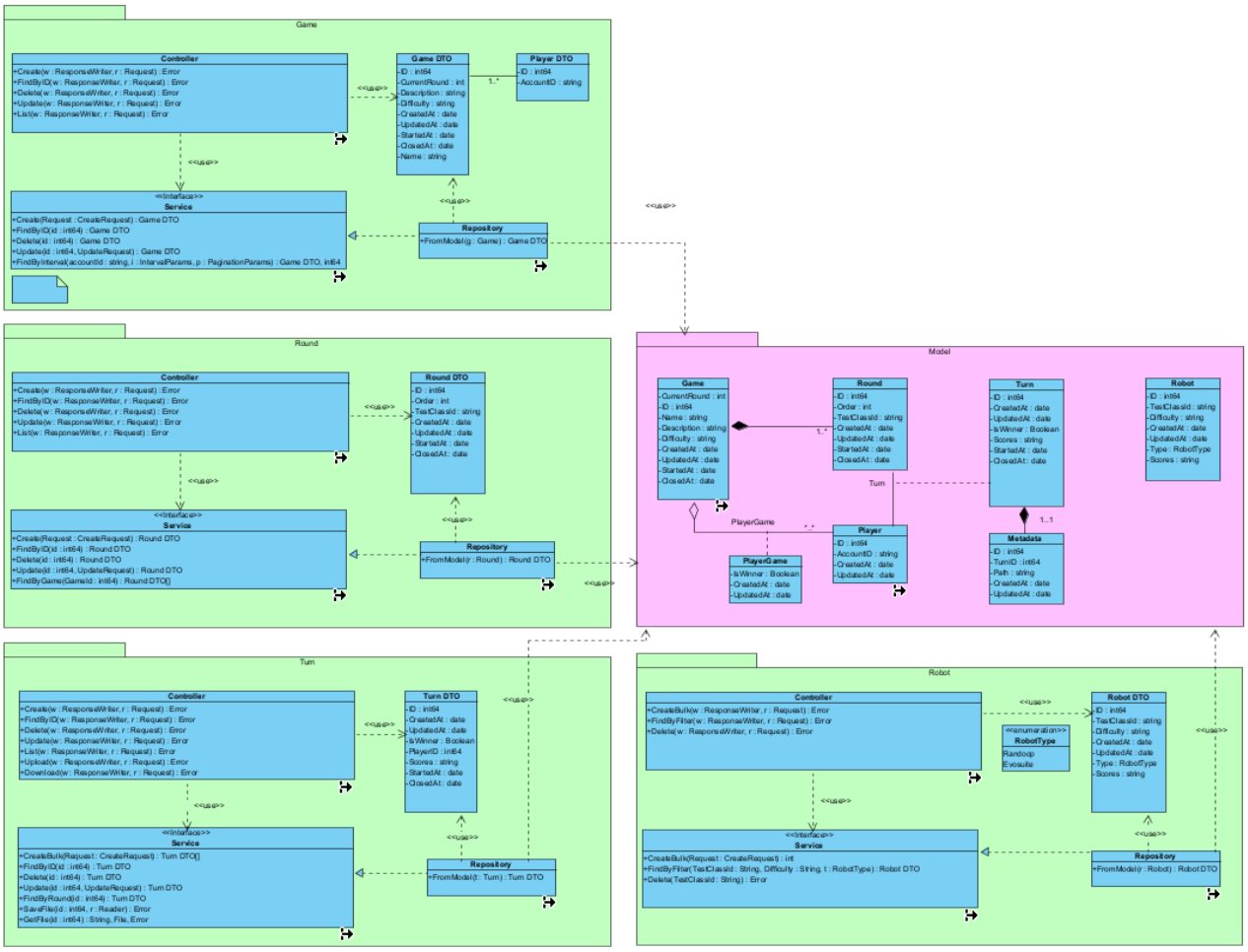

Figura 7: System Domain Model

3.3 Pattern utilizzati

3.3.1 Pattern: Facade

Il pattern Facade viene utilizzando quando una classe *Client* deve interagire con altri componenti per realizzare una funzionalità complessa (Figura 8). La soluzione a questo problema consiste nel creare una classe proxy *Facade* che nasconde la complessità dell'operazione esponendo un solo metodo chiaro, abbassando il grado di accoppiamento della classe *Client* con quelle che forniscono servizi (Figura 9).

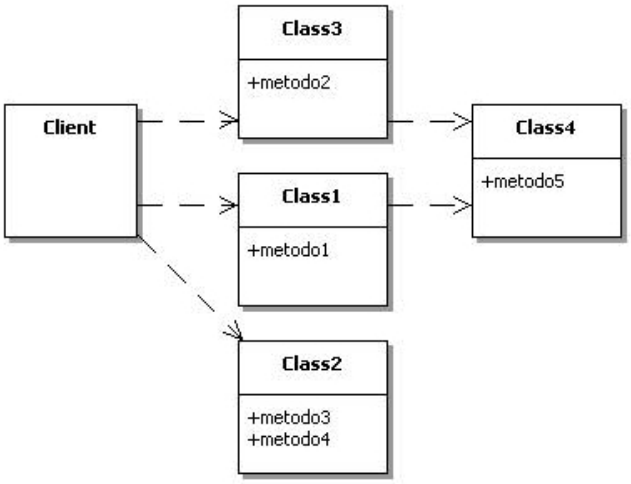

Figura 8: Situazione generica in cui è possibile utilizzare il pattern *Facade*

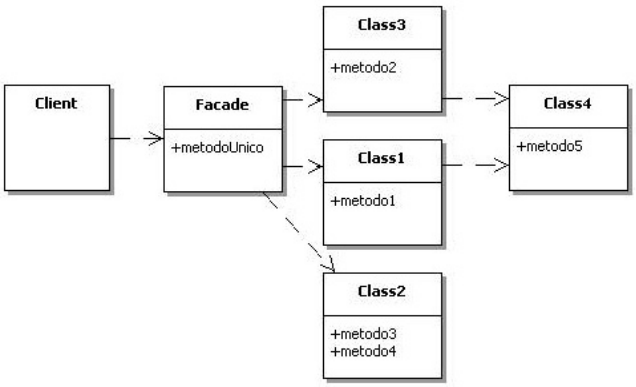

Figura 9: Applicazione del pattern *Facade* al problema in Figura 8

All'interno del componente progettato, il pattern *Facade* è stato utilizzato per semplificare la gestione degli accessi ai risultati dei robot, in quanto, dovendo gestire la moltitudine di risultati prodotti da *Randoop*, è opportuno semplificare sia l'operazione di scrittura (accettando in ingresso più valori per singolo robot), ma anche in lettura tornando un valore random nel caso in cui si decida di prelevare un risultato specifico.

3.3.2 Pattern: Dependency Injection

Il pattern **Dependency Injection** (DI) è utilizzato per ridurre l'accoppiamento tra due oggetti nel caso in cui siano legati da una relazione di tipo *use*. In particolare, l'oggetto utilizzatore accetterà un'interfaccia della dipendenza grazie alla quale sono astratte le funzionalità utilizzate rendendolo, di fatto, indipendente dall'implementazione offerta.

All'interno del componente sviluppato tutti i **controller** accettano un'interfaccia di tipo *Service* che astrae la logica implementativa che fa uso di un database per la gestione dei dati rendendo molto più semplice il testing. Infatti, in fase di test di unità, al componente controller viene fornita un'implementazione finta del servizio consentendo di testare in maniera indipendente le funzionalità di *boundary* da quelle della logica applicativa.

3.3.3 Pattern: Data Transfer Object

Il pattern Data Transfer Object (DTO)[4] consente di creare viste personalizzate sui dati da mostrare agli utenti. Inoltre, consente di assemblare in un'unica entità oggetti appartenenti a classi diverse (Figura 10).

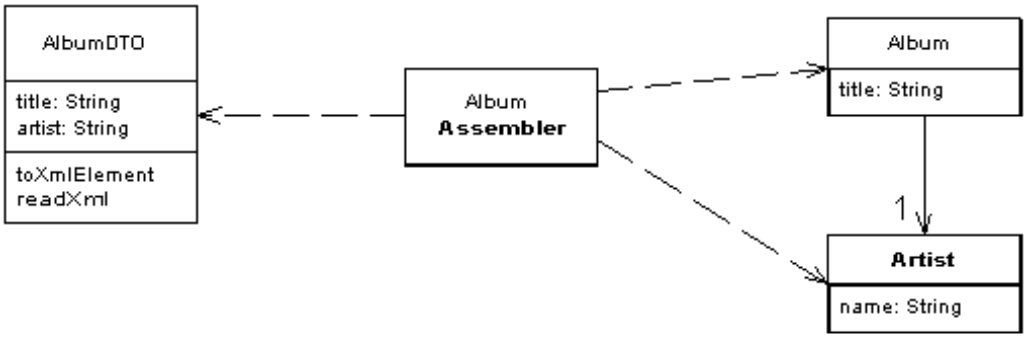

Figura 10: Esempio di applicazione del pattern DTO

All'interno del componente sviluppato, tutte le entità del modello dei dati hanno un corrispondente DTO che viene assemblato grazie alla funzione *fromModel*. L'utilizzo di questo pattern conferisce all'architettura del sistema flessibilità in quanto è possibile sempre cambiare e/o modificare i dati inviati e ricevuti dai sistemi con cui ci si interfaccia senza cambiare il modello relazionale iniziale. In questo modo, si sta, di fatto, definendo un'interfaccia di I/O tra il sistema e gli altri componenti della soluzione globale.

3.3.4 Pattern: middleware

Il pattern **Middleware** consente di implementare una catena di responsabilità secondo la quale ad ogni richiesta (o messaggio) possono essere applicate diverse funzioni ed offre la possibilità di estendere le funzionalità di un'applicazione semplicemente applicando delle configurazioni (Figura 11).

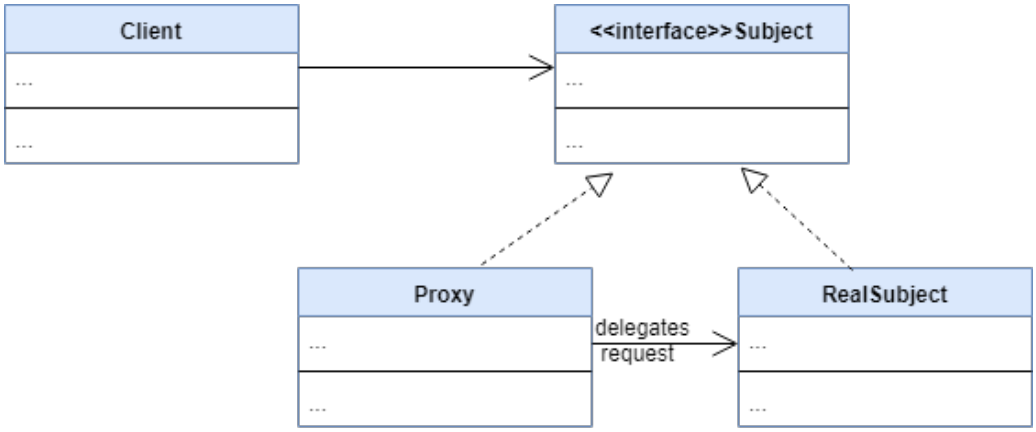

Figura 11: Esempio di realizzazione del pattern Middleware

All’interno dell’applicazione sviluppata il pattern *middleware* è utilizato per eseguire operazioni sulla validità di richieste HTTP. Ad esempio, per richieste in scrittura sono eseguiti i seguenti middleware (illustrati in Figura 12):

- **rate limiting**: si controlla che un client non effettui troppe richieste ad un server;: la richiesta, una volta terminata viene mostrata sul terminale. Per ogni richiesta, sono mostrati *path*, durata, metodo e codice HTTP della risposta;
- **controllo della validità dell’header HTTP**: na richiesta con un corpo in JSON deve necessariamente presentare l’intestazione *Content-Type* con il valore di *application/json*;
- **controllo della dimensione della richiesta**: per evitare attacchi DoS o semplicemente per proteggere il database, la dimensione delle richieste è limitata da un middleware;
- **ip reale**: se l’applicazione risiede alle spalle di un *Reverse Proxy* il valore dell’indirizzo sorgente della richiesta sarebbe sovrascritto da quello del proxy e conservato in *header standard*. Questo middleware rimette l’indirizzo IP del client all’interno di quello sorgente;
- **autenticazione**: può essere configurato un componente che per ogni richiesta verifica se è presente un token di autenticazione e contatta un servizio esterno per verificarne la validità;

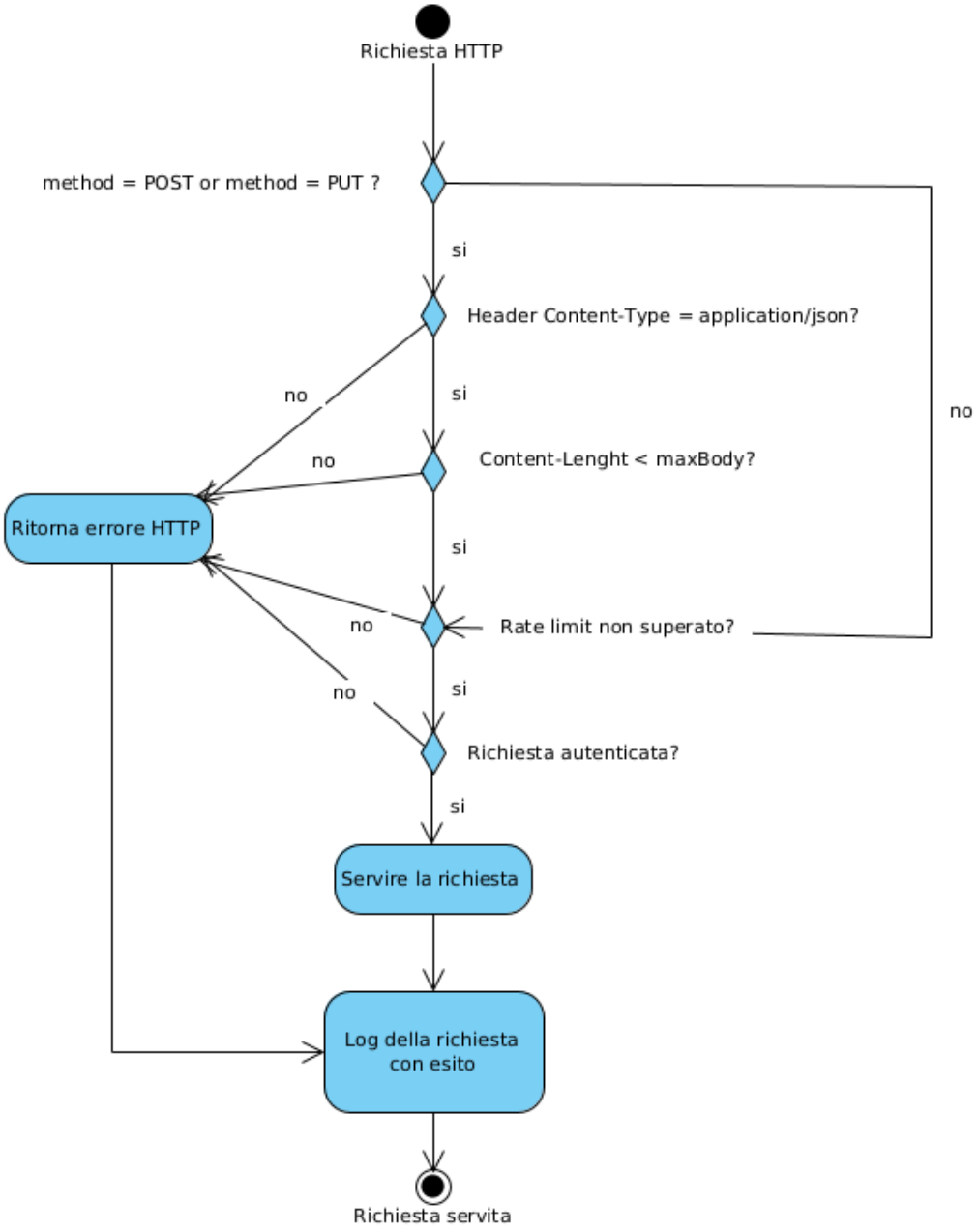

Figura 12: Catena di esecuzione per una generica richiesta HTTP all'interno del componente REST Server

Questo pattern architetturale conferisce all'applicazione la capacità di essere estesa per esempio aggiungendo un modulo di autenticazione, come mostrato in Figura 12. Inoltre, attraverso un'opportuna struttura del sistema ciascun "anello" della catena può essere configurato per essere attivato o meno conferendo al sistema un alto grado di flessibilità.

3.4 Tecnologie

3.4.1 Tecnologie analizzate

Ai fini di implementare le funzionalità richieste sono state analizzate diverse soluzioni che prevedono l'utilizzo stack tecnologici differenti, i cui vantaggi e svantaggi sono qui elencati:

Stack	Vantaggi	Svantaggi
Go	• Performance migliori in termini di latenza per le richieste e consumo di memoria per le web application rispetto alla soluzione in Java [3]; • GO è *battery included*: non è necessaria nessuna dipendenza per essere installato, produce un singolo file eseguibile con le dipendenze *linkate* staticamente. Ciò lo rende particolarmente adatto ad uno scenario CI/CD; • Possibilità di scaricare localmente le dipendenze (*vendoring*) e avere un progetto sempre compilabile anche a distanza di anni; • I/O asincrono e concorrenza nativamente implementata (Go routines e CSP); • Linguaggio molto semplice da imparare (meno di 25 keyword) e molto stabile; • Supporta nativamente strumenti per la generazione del codice automatica;	• Garbage collected; • Poca *syntactic sugar* che può rendere lo sviluppo verboso;

Tabella 6: Vantaggi e svantaggi di uno *stack* basato su Go

Stack	Vantaggi	Svantaggi
Java	• Springboot e Hibernate è uno degli *stack* tecnologici più utilizzati e maturi; • Supporto per CRUD API tramite attività di configurazione e poca scrittura di codice;	• Garbage collected; • Performance degradate rispetto alla soluzione Go e Rust [3] [8]; • *Slow startup*: richiede un grande investimento di tempo per inizializzare un progetto che lo rende poco adatto per lo sviluppo di prototipi;

Tabella 7: Vantaggi e svantaggi di uno *stack* basato su Java

Stack	Vantaggi	Svantaggi
PostgREST	• Nessuna linea di codice viene scritta; • Sfrutta appieno tutte le potenzialità di PosgreSQL; • Può essere utile per effettuare ricerche con filtri avanzati;	• Rende impossibile passare ad un altro database causando il cosiddetto *vendor locking*; • Diventa impossibile da utilizzare quando alle operazioni CRUD bisogna aggiungere della logica. Questa logica può essere molto complicata da implementare lato database; • Richiede una conoscenza avanzata del database PostgreSQL;

Tabella 8: Vantaggi e svantaggi nell'utilizzo di Postgrest

Stack	Vantaggi	Svantaggi
Rust	• *Memory safety* a tempo di compilazione garantita senza un *garbage collector* [1]; • Rust offre le migliori performance sia per memoria/CPU usati che per tempo di risposta delle richieste rispetto alle altre tecnologie presentate [10] [5]; • Compilazione di un singolo file eseguibile adatto ad uno scenario CI/CD; • Axum è basato sull'engine *hyper.rs* contraddistinto da prestazioni migliori rispetto alle altre soluzioni presentate [8]; • Semplicità nella gestione dei progetti con Cargo [2];	• Il linguaggio può risultare complicato da apprendere; • Scarso supporto per l'autogenerazione del codice data una specifica OpenAPI; • Linguaggio poco conosciuto, può rallentare lo sviluppo laddove non ci fosse supporto dalla *community*; • A causa delle ottimizzazioni e dei controlli effettuati dal compilatore le *build* possono essere lente;

Tabella 9: Vantaggi e svantaggi di uno *stack* basato su Rust

3.4.2 Tecnologie scelte

La soluzione finale prevede l'utilizzo dello stack basato su GO e di un database relazionale. In particolare è stato scelto PostgreSQL in quanto si tratta di una soluzione completamente open source, largamente utilizzata, affidabile e altamente scalabile. Mentre il database è utilizzato per memorizzare i dati relativi alle partite e ai punteggi, si è scelto di conservare i file java delle classi testate generate dai giocatori nel file system, salvando i metadati con il relativo path in una tabella apposita, così da ottimizzare le performance.

Per garantire l'osservabilità del sistema si è scelto di utilizzare Prometheus, uno strumento che traccia diverse metriche relative al sistema e permette di visualizzarle in dashboard elementari. Per permettere

una visione più avanzata delle metriche a quest’ultimo è stato collegato Grafana, che permette di creare dashboard personalizzate per poi salvarle.

4 Implementazione

4.1 Flusso partita

In questa sezione vengono mostrate le diverse azioni dell'utente (*Game Engine*) svolte durante lo svolgimento della partita, modellate come un'*activity diagram*. Ad ognuna di queste azioni corrisponde una richiesta tramite API, il cui funzionamento è modellato nella sezione successiva.

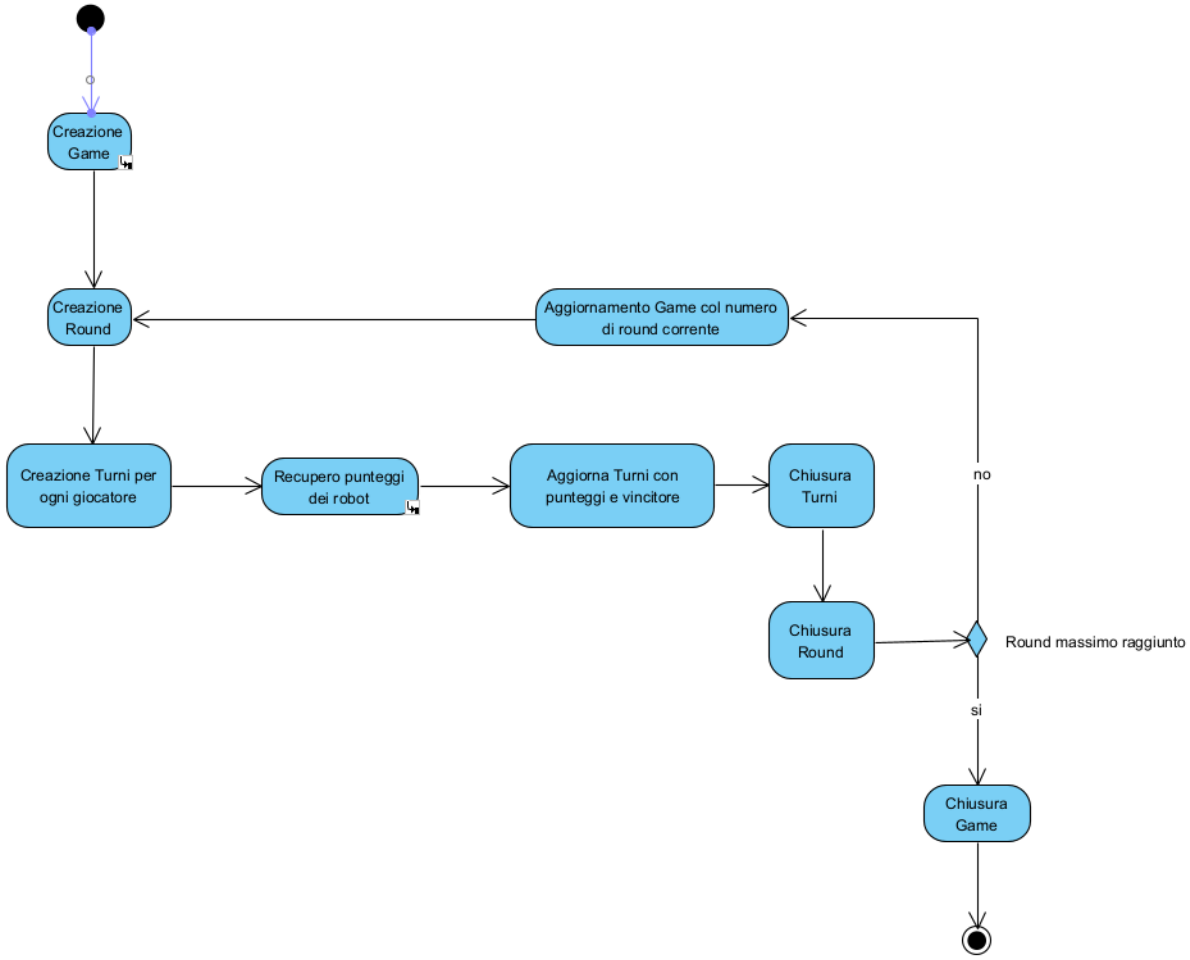

Figura 13: Flusso partita

4.2 Diagrammi di sequenza

Di seguito sono modellate alcune delle richieste che l'utente può effettuare durante lo svolgimento di una partita.

Creazione Game Il *Game Engine* invia una richiesta per la creazione della partita. Di seguito è riportato un body di esempio per la richiesta:

```
  {
  "name": "Game name",
  "players": [
    "id1",
    "id2"
  ],
  "description": "description",
  "difficulty": "easy"
}
```

Nel caso in cui la richiesta non sia valida il sistema invierà un messaggio di errore, altrimenti verrà creata la partita con i relativi giocatori salvandola nel database. In seguito verrà restituito il DTO (Data Transfer Object) relativo alla partita creato dalla funzione *FromModel*.

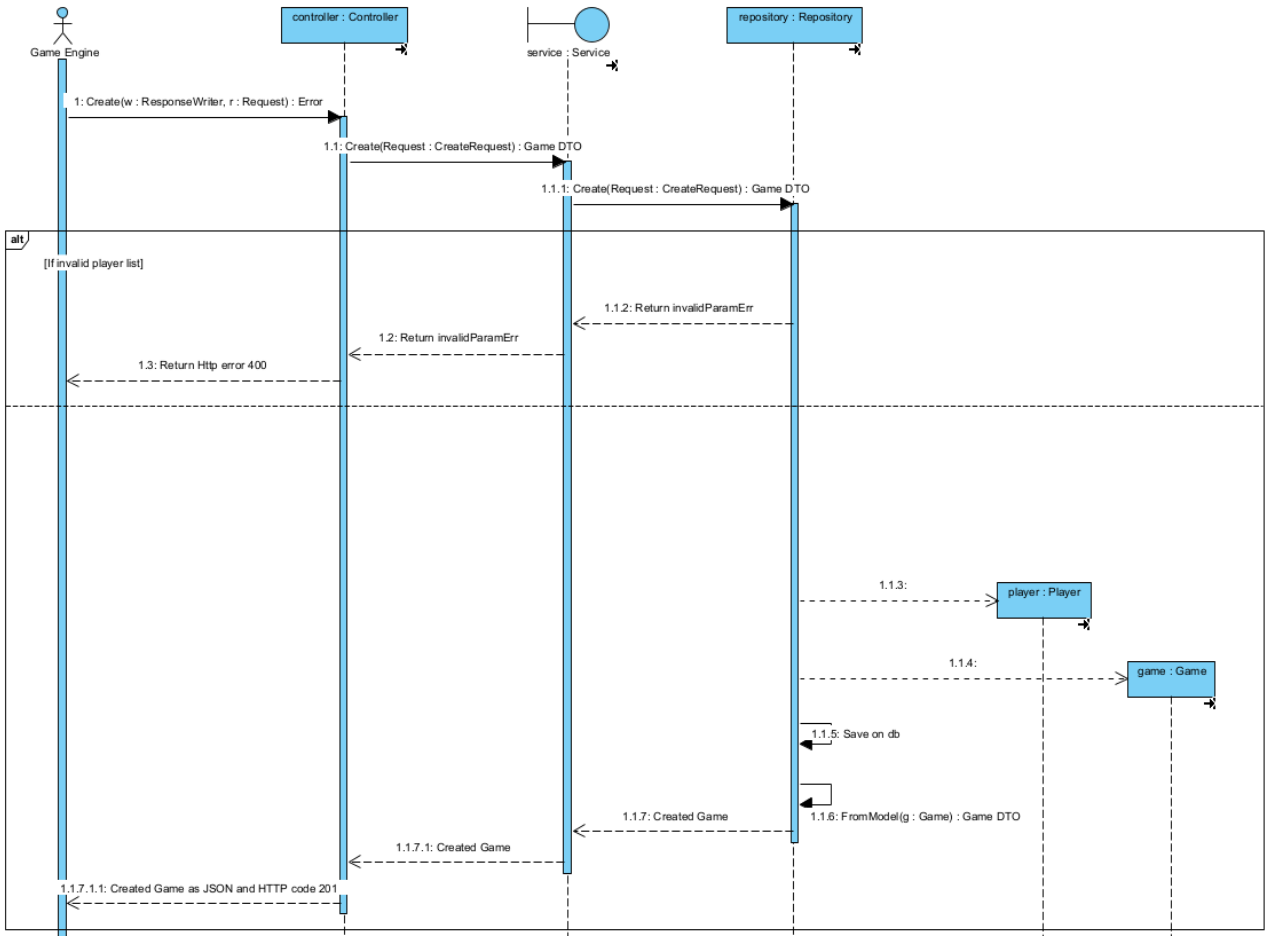

Figura 14: Creazione Partita

Aggiornamento Game L'aggiornamento della partita è necessario qualora si volesse modificare il numero del round corrente, o indicare l'inizio e fine della partita. È possibile inoltre modificare il nome e la descrizione del game. Il sistema controlla che la partita da modificare esista e che la richiesta abbia un body valido, dopo di che, in caso di successo, restituisce il DTO della partita modificata sotto forma di JSON.

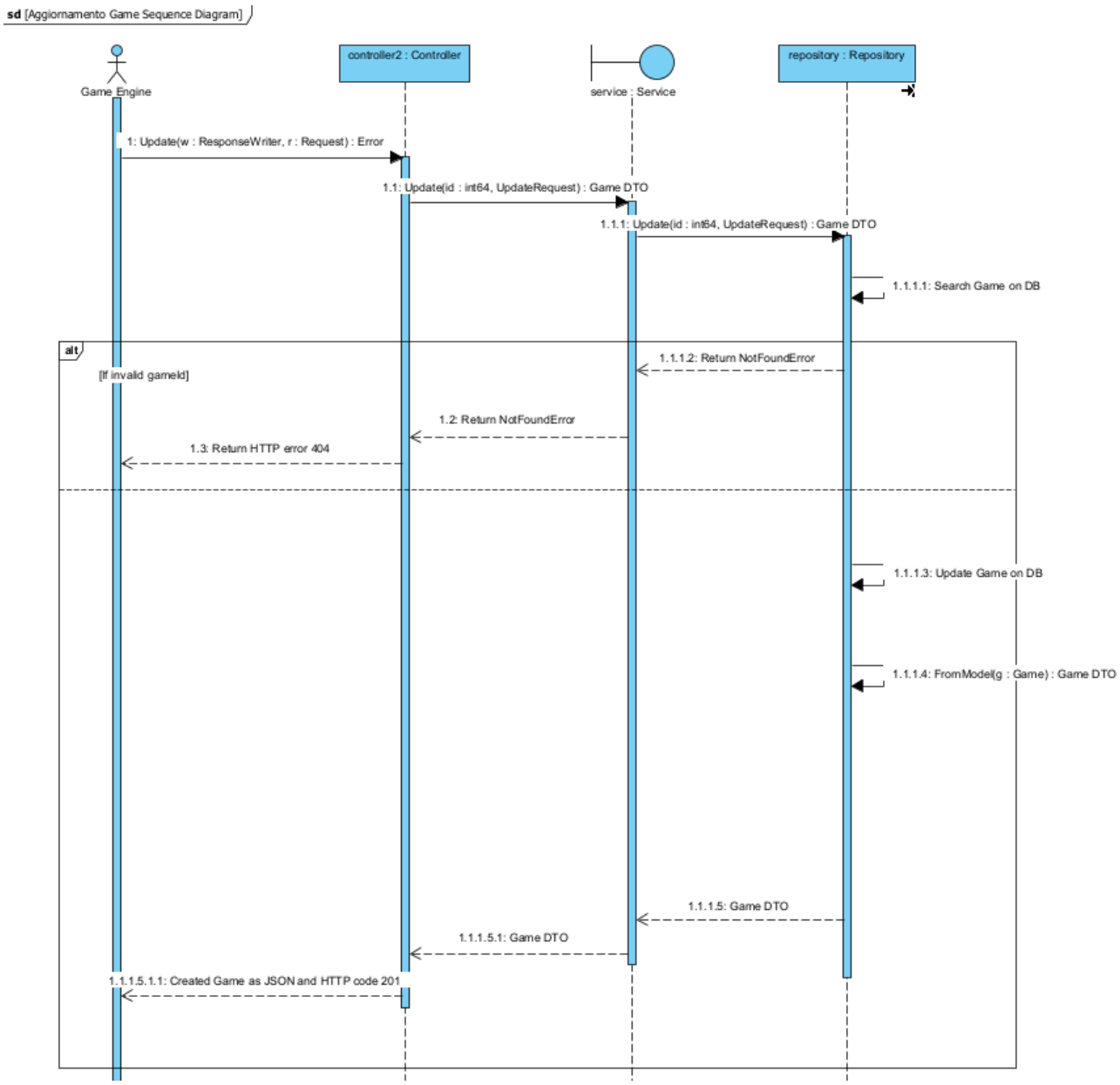

Figura 15: Aggiornamento partita

Creazione Round La creazione dei round avviene in maniera simile a quella delle partite. In particolare, l'utente dovrà specificare nel body della richiesta l'id del game di cui il round fa parte, e l'id della classe da testare. Ogni round creato, oltre a un proprio id, avrà un campo chiamato *order* che indicherà il suo ordine rispetto agli altri round nella partita.

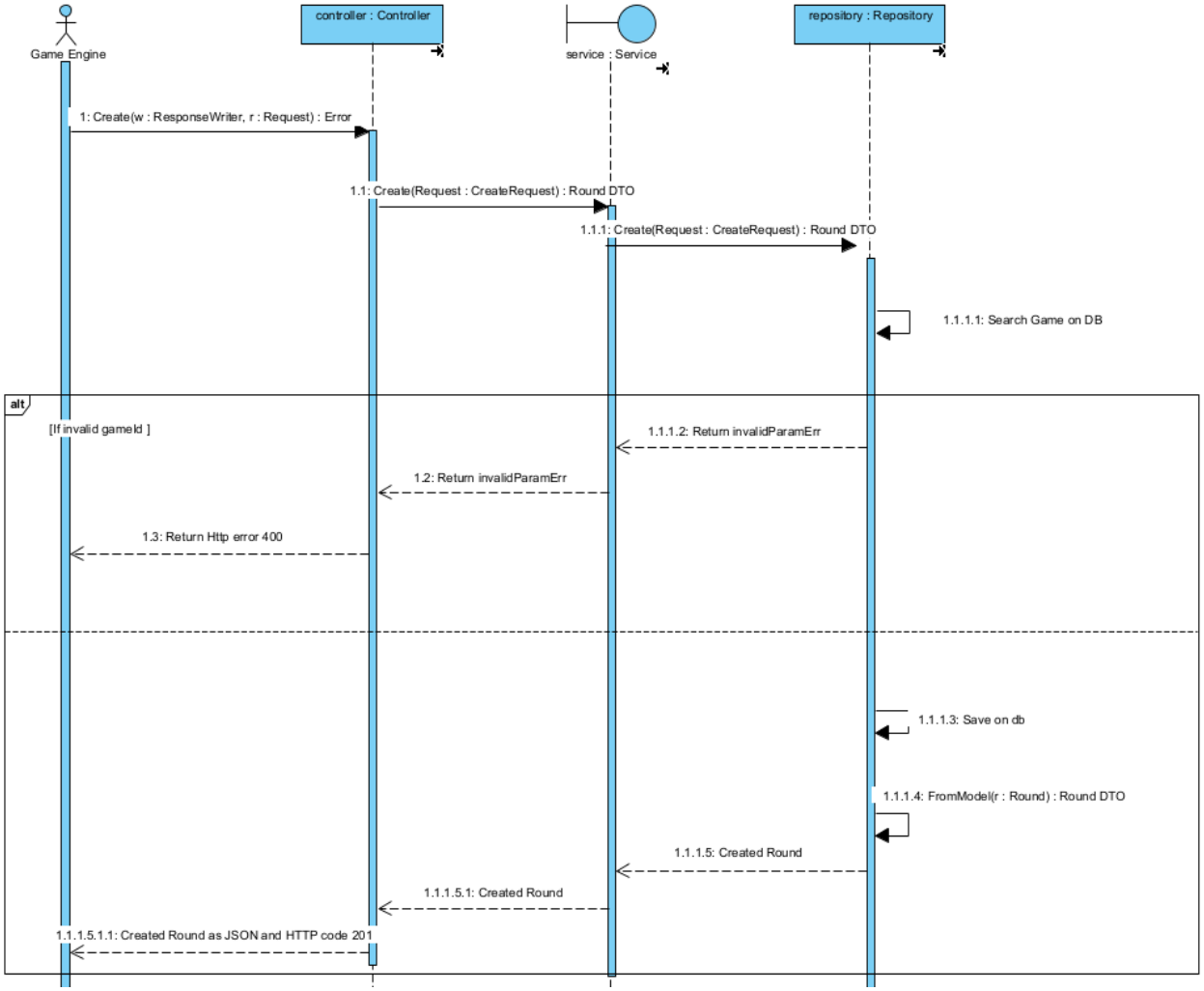

Figura 16: Creazione Round

Aggiornamento Round L'aggiornamento del round è necessario principalmente per indicarne la data di inizio e fine. Come nei casi precedenti il sistema controlla se la richiesta e valida e in caso di successo restituisce il DTO del round aggiornato.

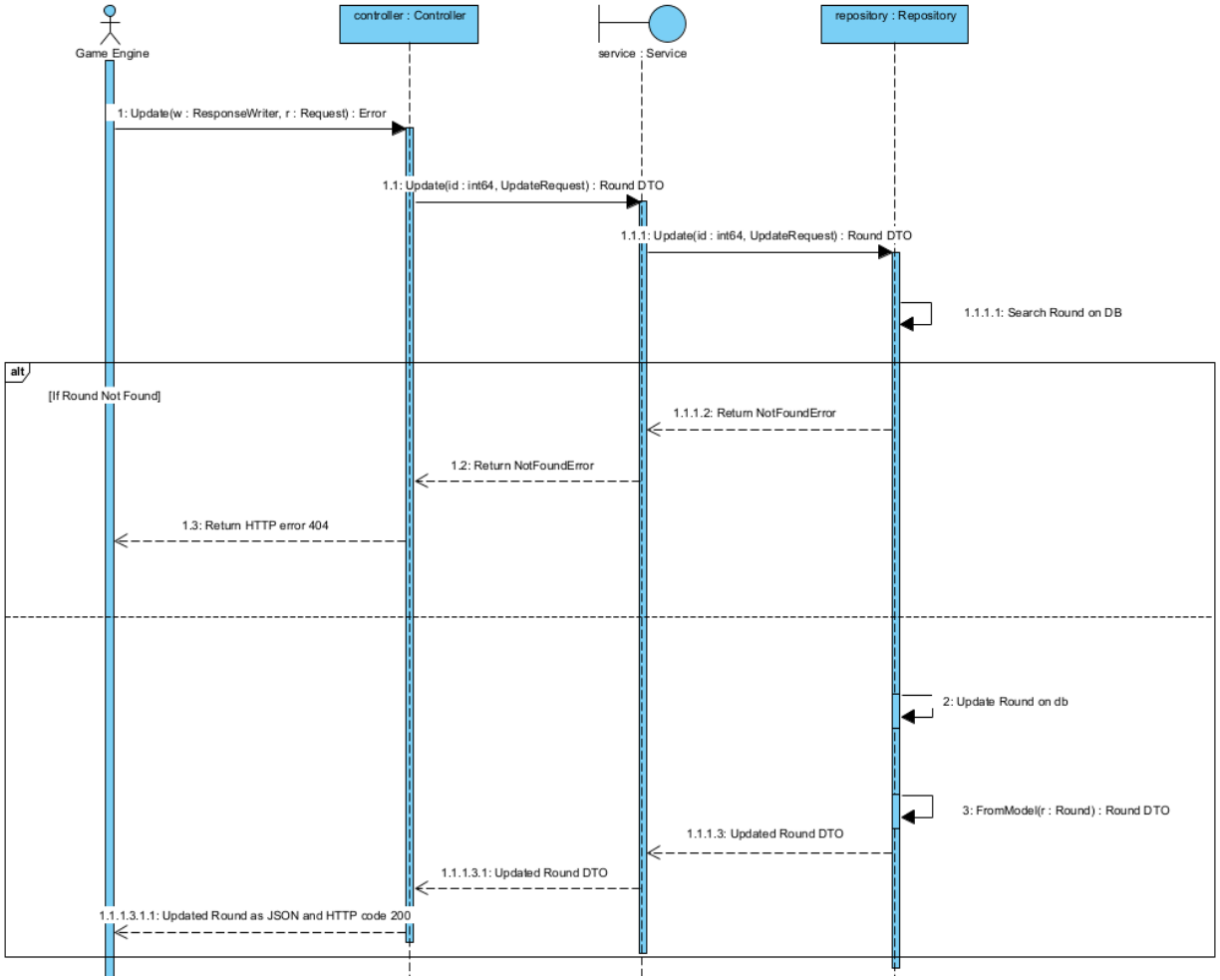

Figura 17: Aggiornamento Round

Creazione Turni Durante lo svolgimento di un round è necessario creare un turno per ogni giocatore. L'utente invia una richiesta contenente la lista degli id dei giocatori partecipanti e l'id del round. Nel caso sia valida, il sistema crea in loop una serie di turni associati a ogni giocatore, dove poi saranno contenuti i dati relativi alla sua prestazione , come i punteggi e l'eventuale vittoria. Come nei casi precedenti il sistema restituisce in seguito i DTO relativi ai turni creati.

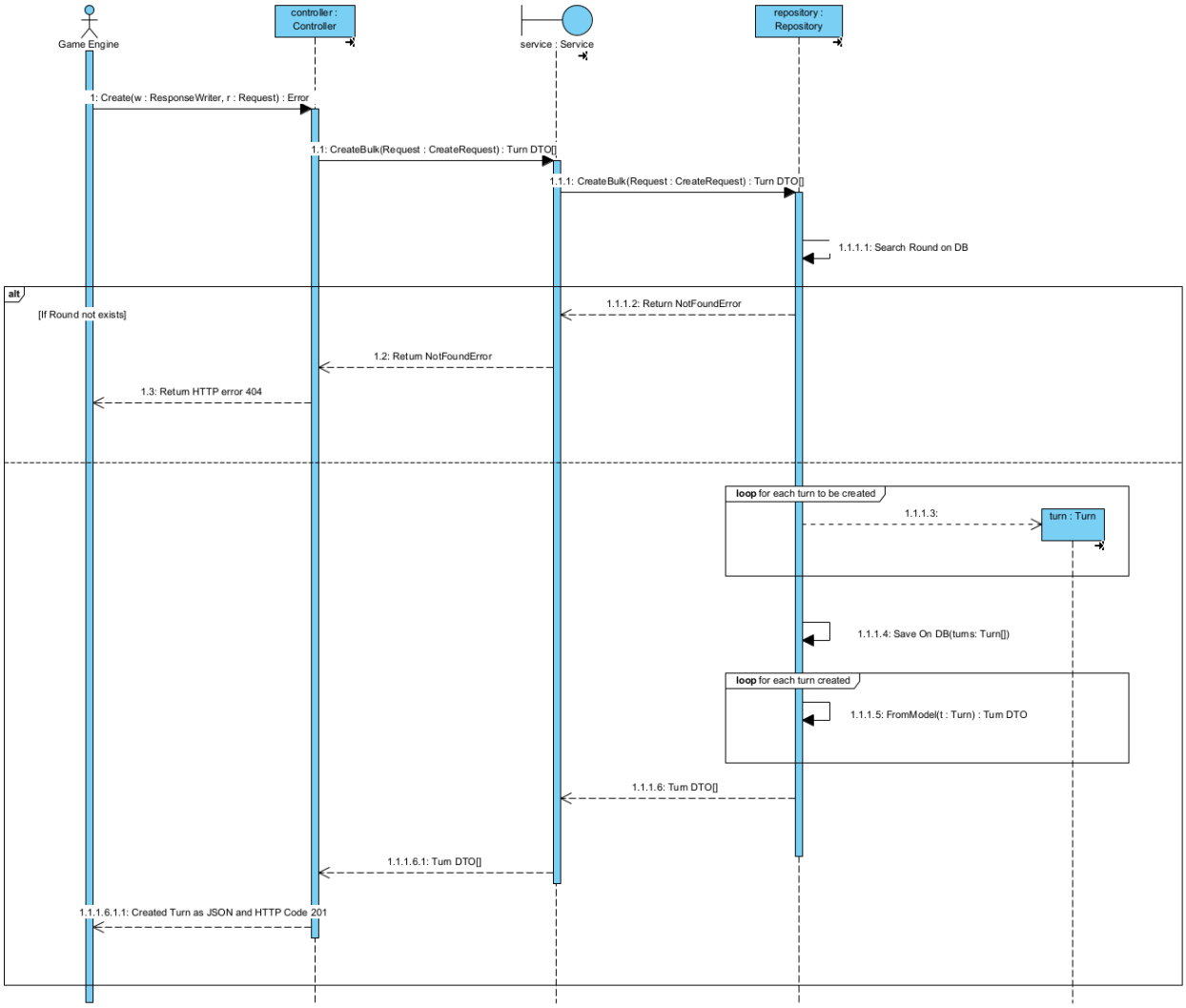

Figura 18: Creazione Turno

Aggiornamento Turni L'aggiornamento dei turni è necessario nel momento in cui l'utente deve caricare i punteggi calcolati dai componenti di testing e esecuzione delle classi. Dopo aver calcolato il punteggio migliore, è inoltre possibile indicare quale degli utenti sia il vincitore del round impostando il valore *isWinner* del suo round a *True*. Una volta ricevuta la richiesta il sistema controlla la sua validità e in caso positivo restituisce il DTO del turno aggiornato.

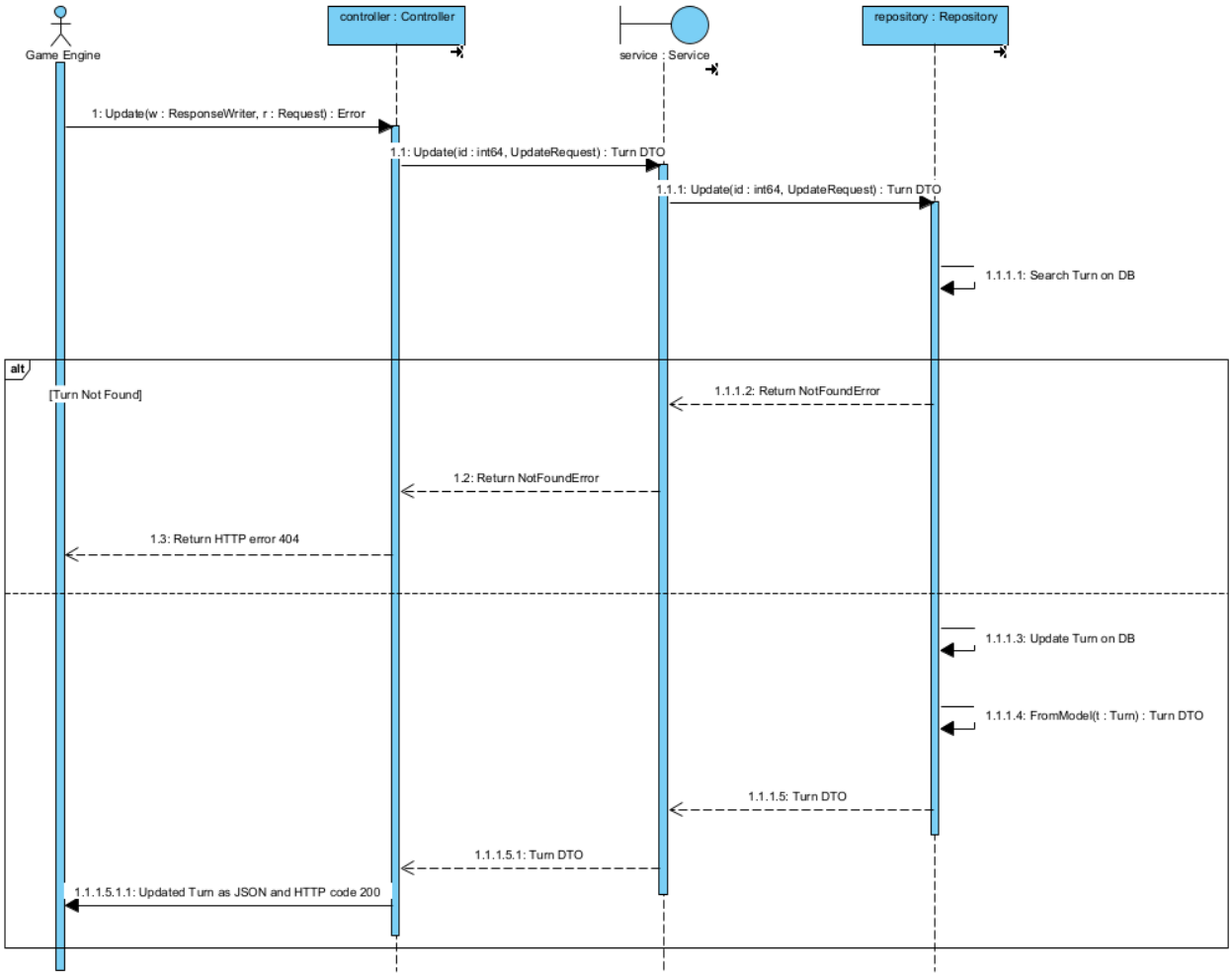

Figura 19: Aggiornamento turno

Recupero Punteggi Robot Durante ogni round l'utente dovrà recuperare i punteggi dei robot relativi alla classe di test in gioco, per paragonarli alle prestazioni dei giocatori. Nella richiesta sarà quindi necessario indicare una serie di parametri per filtrare i punteggi ottenuti:

- *TestClassId*: l'id della classe testata
- *Difficulty* : la difficoltà associata ai punteggi dei robot da recuperare. È infatti possibile memorizzare diversi punteggi di un robot per la stessa classe testata classificandoli con diversi livelli di difficoltà.
- *Type*: il tipo di robot di cui si vuole recuperare i punteggi. Fino al momento della stesura del documento sono utilizzati Randoop e EVOsuite.

Il sistema controllerà la validità della richiesta e la presenza dei punteggi cercati, e in caso di successo restituirà il DTO relativo.

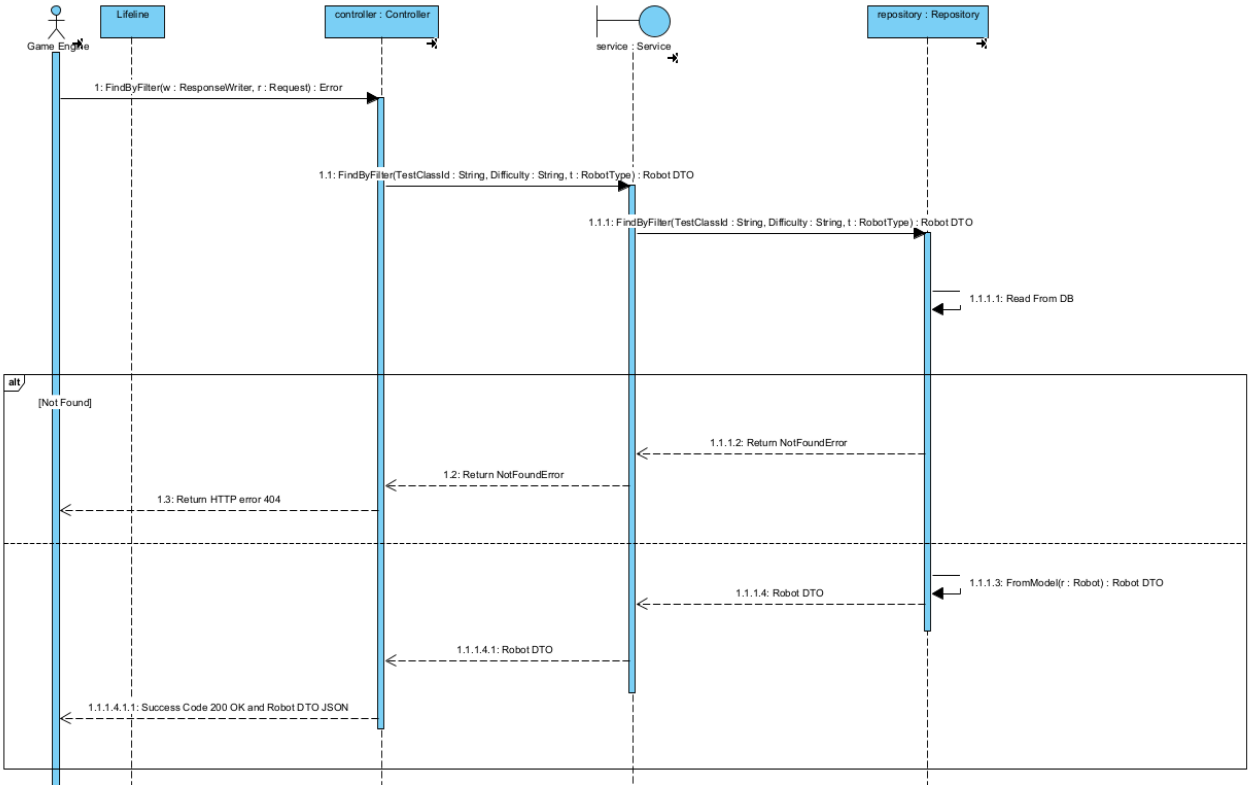

Figura 20: Recupero punteggio Robot

Salvataggio file del turno A seguito della giocata di un turno da parte di uno specifico giocatore, l'utente avrà la possibilità di salvare i file prodotti contestualmente alla giocata in un unico file zip. Nella richiesta preposta sarà necessario specificare:

- *L'id del turno* come parametro nel path della richiesta
- *Il File zip* come body di tipo application/zip della richiesta

Il sistema controllerà l'esistenza del turno indicato, restituendo, in caso di successo, un messaggio di avvenuto salvataggio qualora lo stesso andasse a buon fine.

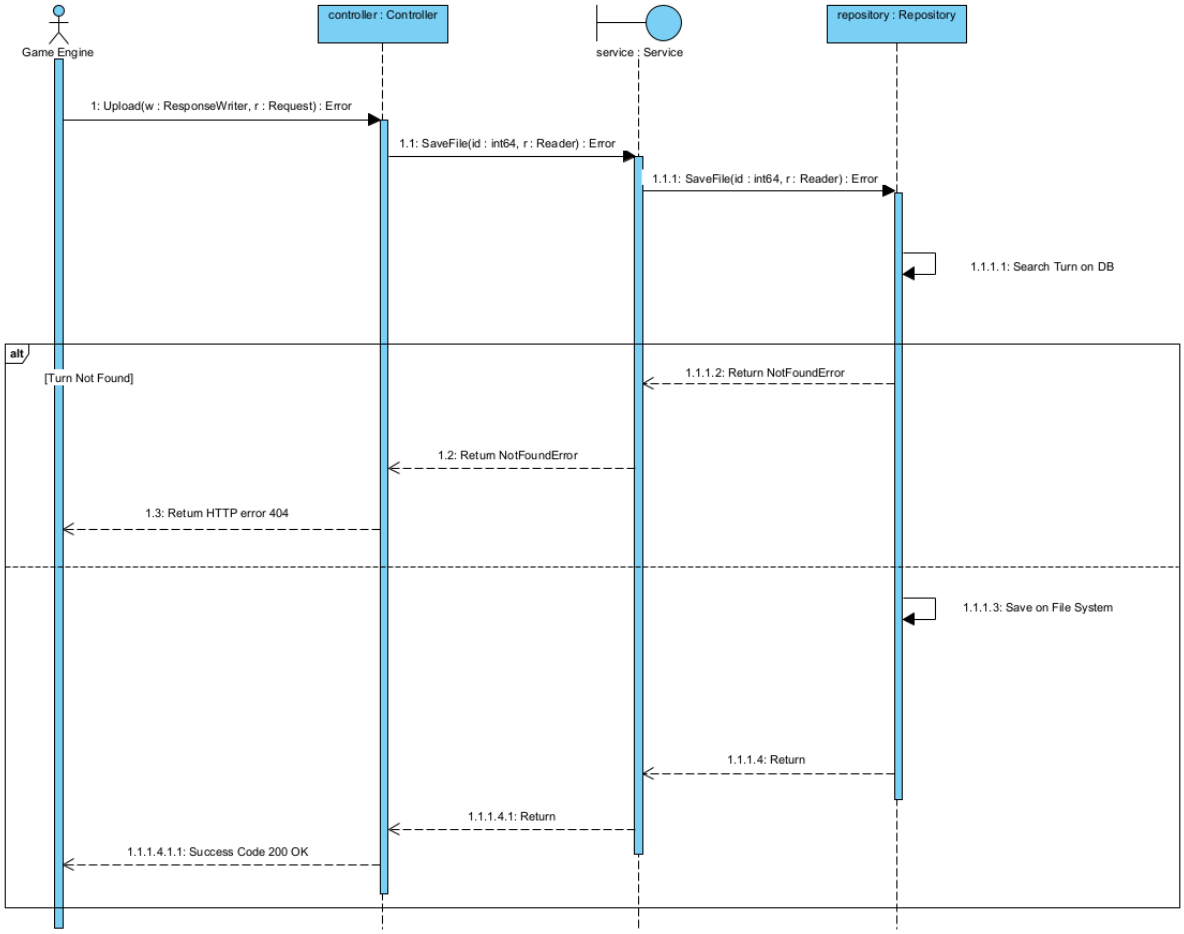

Figura 21: Salvataggio file del turno

Recupero file del turno L'utente potrà recuperare in qualunque momento l'archivio contenente i file associati ad uno specifico turno. Nella richiesta di recupero del file sarà necessario specificare *L'id del turno*. Il sistema controllerà l'esistenza di un file precedentemente salvato a sistema per il turno indicato, restituendo, in caso di successo, una response con Content-Type application/zip corrispondente al file zip precedentemente salvato.

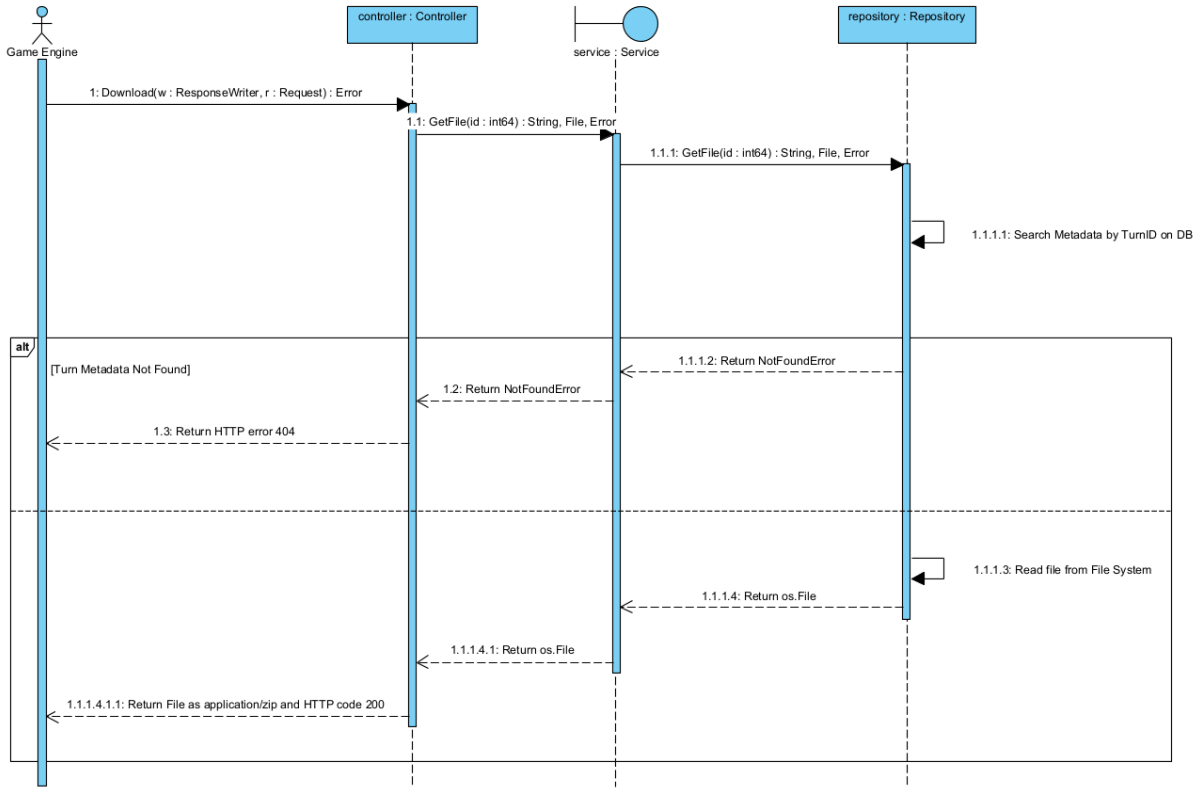

Figura 22: Recupero file del turno

4.3 Documentazione API

Swagger Al fine di garantire una corretta integrazione dell'applicativo con gli altri componenti dell'architettura oggetto di altri task, si è scelto di adottare l'utilizzo di un interfaccia *Swagger* che documentasse i servizi offerti dal componente realizzato. L'interfaccia Swagger, infatti, realizza i principi della specifica *OpenAPI* la quale fornisce uno standard per l'integrazione di componenti eterogenei che comunicano tramite API HTTP. Tale interfaccia consente di presentare il dettaglio dell'API esposta dall'applicativo facilitando il testing e l'integrazione del Game Repository e fornendo informazioni relative ai singoli servizi quali path, parametri, dettaglio della request e possibili respone del servizio. Attraverso Postman, è stata progettata la specifica OpenAPI degli endpoint ed è stata integrata attraverso github sul progetto principale. Infine, è stata effettuata un'integrazione all'interno del progetto per esporre la *Swagger UI* e servire sia la specifica OpenAPI che la collection Postman quando l'applicazione viene configurata con il *flag* ***enableSwagger*** a *True*. E possibile consultare la documentazione dell'API al link. (La stessa è riportata in Figura 23).

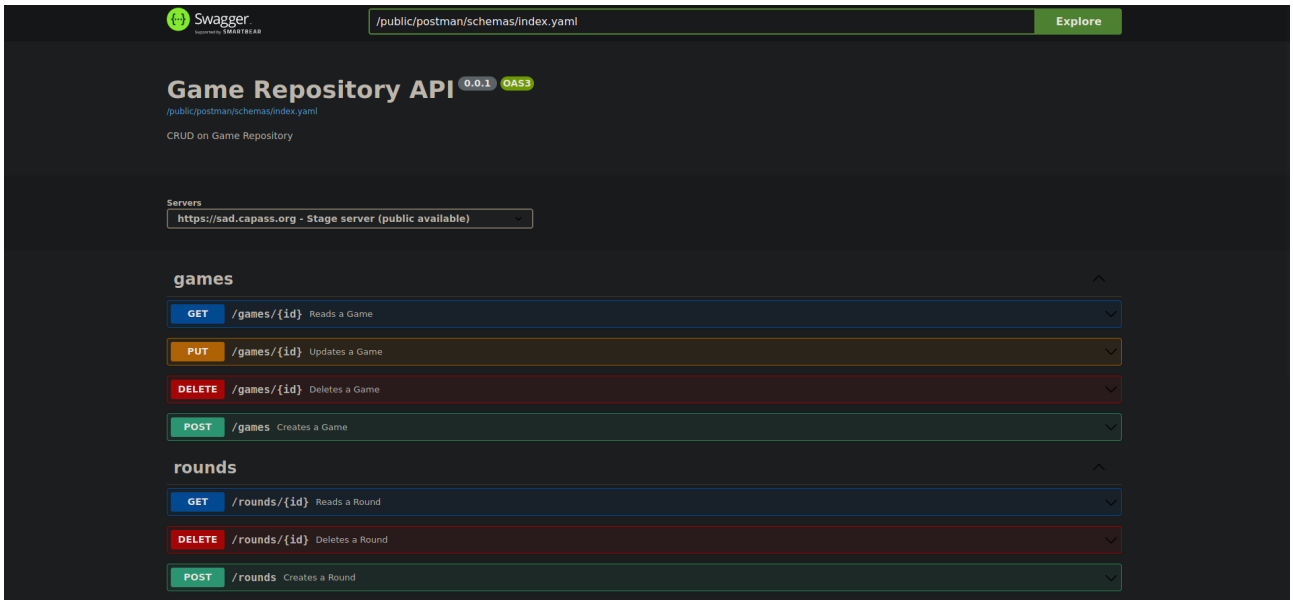

Figura 23: Swagger UI

API Diagram Nei diagrammi prodotti per la documentazione dell'applicativo è incluso un Class Diagram riportante il dettaglio dei servizi esposti dall'applicativo. Visual paradigm consente infatti di modellare le API HTTP documentandone i dettagli della richiesta, i metodi, gli end-point e i possibili output.

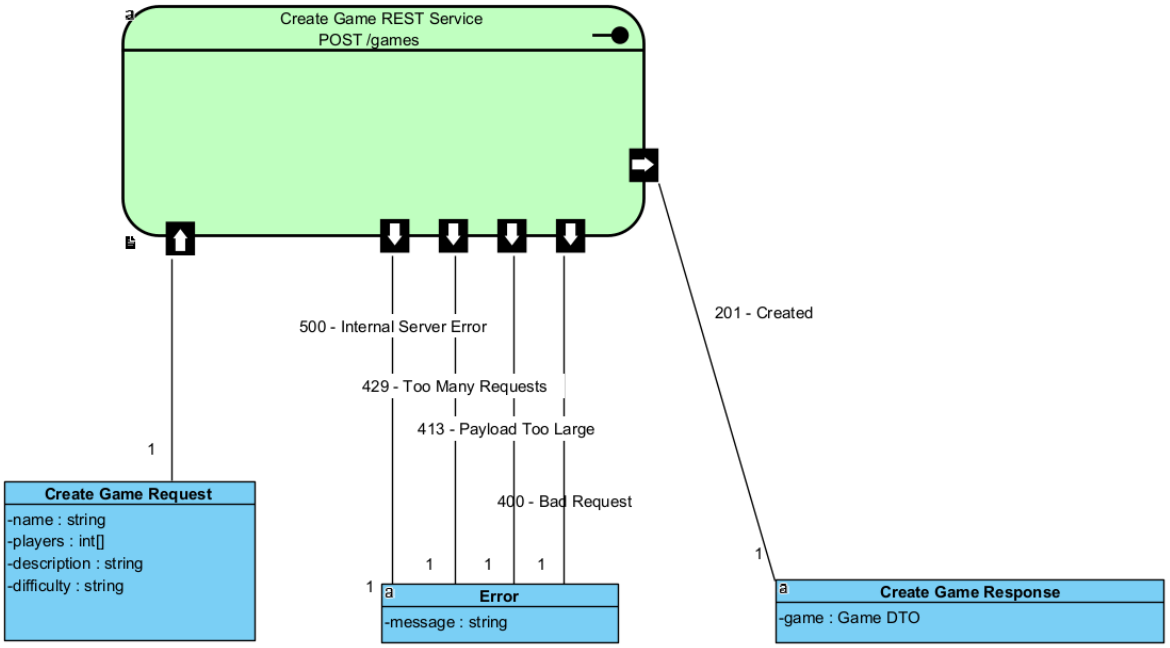

Figura 24: Dettaglio del Servizio di Create Game sul diagramma delle API

4.4 Prometheus & Grafana

Al fine di garantire l'osservabilità della soluzione si è scelto di utilizzare Prometheus e Grafana. Prometheus è un tool che offre la funzionalità di tracciare diverse metriche relative al sistema e di visualizzarle in dashboard basilari. Per estendere gli strumenti di analisi si è scelto di utilizzare Grafana, software che consente la creazione grafici e dashboard personalizzate per poi salvarle. Un esempio di dashboard è mostrato nella figura Figura 25, in cui sono tracciate le seguenti metriche:

- Request Rate: il numero di richieste al minuto;
- CPU Usage: utilizzo in percentuale della CPU. Prometheus fornisce l'utilizzo in secondi quindi è stato calcolato come il rate in 5 minuti moltiplicandolo poi per 100;
- Go Threads: il numero di thread. In blu sono mostrati quelli utilizzati sui core fisici, mentre in giallo sono mostrati quelli per gestire le richieste;
- Memory profile: memoria utilizzata;
- GC Operation: le operazioni del Garbage Collector;
- Open FDS: il numero di connessioni aperte;
- Scrape duration: la durata della lettura dei dati tra Prometheus e il servizio.

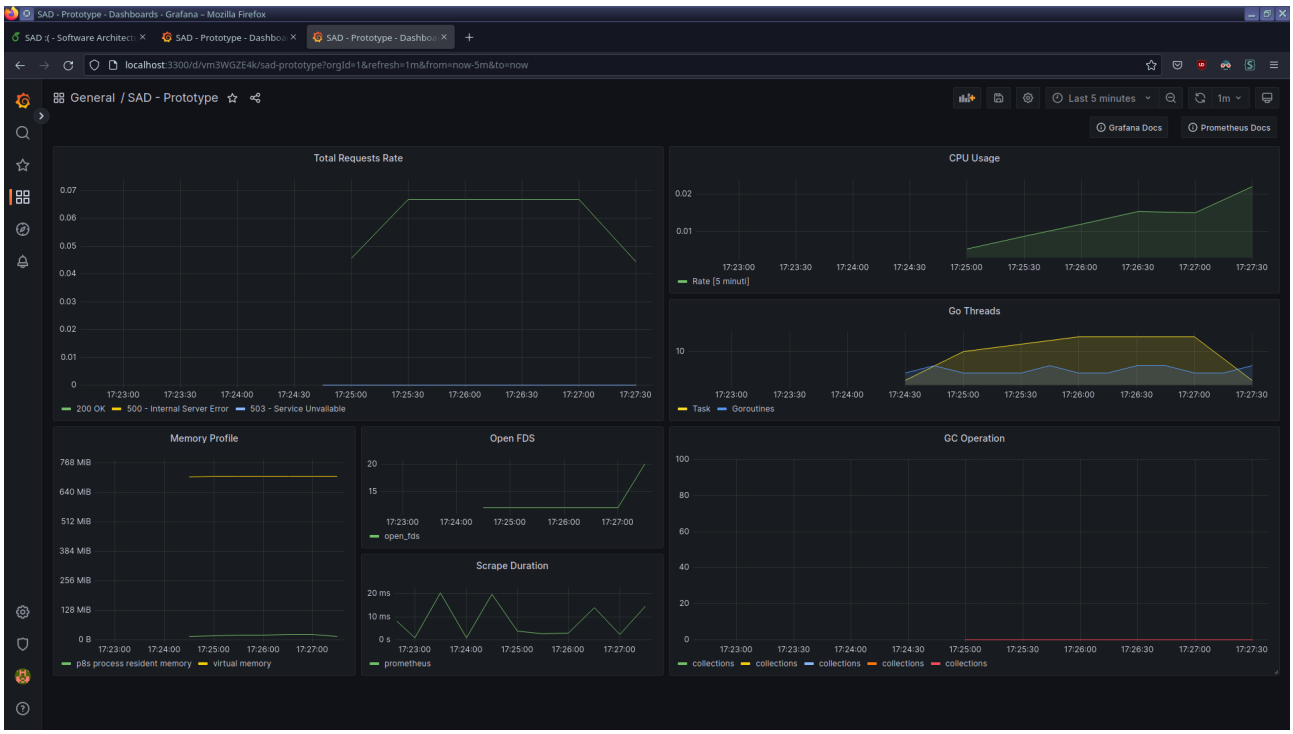

Figura 25: Esempio *dashboard* Grafana

4.5 Rate limiting

Per proteggere l'API da attacchi DDoS è stato predisposto un sistema di *rate limiting* che implementa l'algoritmo ***Token Bucket***. Uno schema dell'algoritmo è mostrato in Figura 26.

Questo componente è configurabile con 3 parametri:

- Requests per seconds: numero di richieste al secondo prima di ottenere un errore;
- Bursts: numero di token presenti nel bucket, quando sono esauriti il client viene messo in attesa;
- Enabled: quando è impostato a *False* il *rate limiting* viene disattivato;

Il rate limiting viene applicato per client: per fare questo è stato aggiunta una funzione alla catena di *middleware* dei controller che si occupa di recuperare l'ip dell'*host* originale nel caso in cui l'applicazione sia preceduta da *gateway* e/o *reverse proxy*. Infatti, un'applicazione posizionata alle spalle di un *reverse proxy*, ha come indirizzo sorgente delle richieste *HTTP* quello del *proxy* che per convenzione (per non perdere l'informazione) spostano all'interno di header standard quali *X-Forwarded-For*, *X-Real-IP* o *True-Client-IP*.

4.6 Build System

Il sistema di *build* utilizzato è interamente basato su *Make*. Per eseguire i comandi di compilazione è necessario avere un compilatore *GO*. Di seguito sono illustrati le regole e gli obiettivi del *Makefile* descritto per il progetto:

- **build**: compila l'applicazione nella cartella specificata in *OUT_DIR* (che ha come default il valore "build");
- **run**: esegue l'applicazione compilata con l'obiettivo *build*, con il file configurazione in *CONFIG* (con valore di default "config.json");

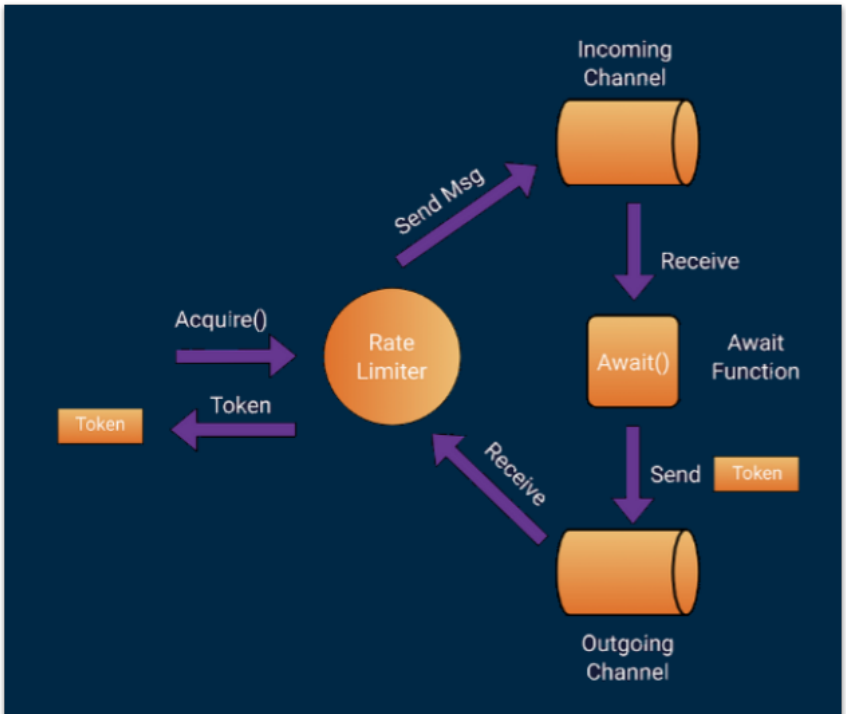

Figura 26: Algoritmo token bucket

- **dev**: avvia l'applicazione in modalità sviluppo (si riavvia ogni volta che viene modificato il codice sorgente). Prima di eseguire questa regola, eseguire il comando *make dev-dependencies* che installerà *air* (lo strumento che si occupera dell'operazione);
- **docker-build**: compila un'immagine Docker usando il tag *game- repository: <GIT_COMMIT>*;
- **docker-run**: esegue l'immagine in un container usando il parametro *CONFIG*, analogamente allo step *run*;
- **docker-push-ssh**: invia l'immagine ad un server utilizzando il protocollo SSH. I parametri per la connessione attraverso SSH devono essere passati attraverso la variabile *SSH* (vuota di default);
- **docker-push**: invia l'immagine ad un *docker registry*. Il nome del *registry* viene specificato nella variabile *REGISTRY*. In particolare, viene fatto l'invio dell'immagine sia con il tag pari al git commit sia che come *latest*;
- **test**: viene effettuato il test unità dell'applicazione;
- **test-integration**: viene effettuato il test di integrazione dell'applicazione. In particolare, se il flag *CI* viene specificato, è possibile usare la variabile *DB_URI* per specificare l'indirizzo di un *database* di test, altrimenti sarà un database usa e getta attraverso un *Docker container*. Inoltre, attraverso la variabile *COVER_DIR* la cartella in cui sarà salvato il file per le metriche di copertura;
- **clean**: rimuove gli artefatti prodotti;
- **help**: utilizza i commenti nel *Makefile* per creare una piccola guida all'utilizzo con esempi di utilizzo;

5 Testing

Il testing della soluzione avviene in maniera automatica ed è stato integrato nel ciclo di sviluppo e rilascio del software.

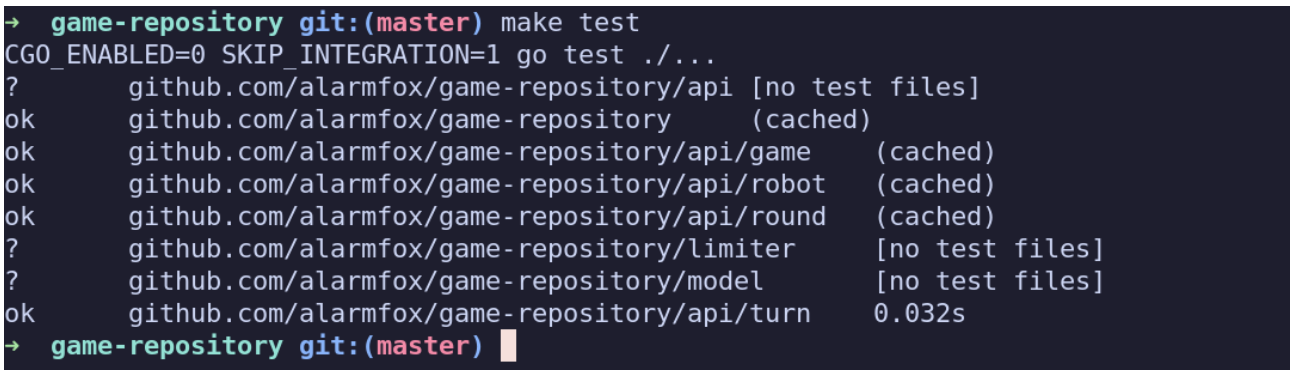

Figura 27: Unit testing attraverso il comando *make*

5.1 Unit testing

Il componente testato è il servizio REST descritto nei paragrafi precedenti. Per garantire una maggiore robustezza del sistema, l'oggetto principale del testing di unità è l'interfaccia input/output dell'applicazione. In particolare, con riferimento all'architettura mostrata in (aggiungere ref all'architettura), sono oggetto di questa attività i controller, ognuno dei quali viene testato con JSON sintatticamente validi verificando il risultato a un particolare codice dello standard HTTP. In questa fase, è stato utilizzato un approccio *table testing* secondo cui per ogni componente è stata progettata una tabella di *test case*, come mostrato in Tabella 10.

TC	Descrizione	Risultato atteso (Codice HTTP)
T01-GameNotExists	Ricerca di partita non esistente	404
T02-GameExists	Ricerca di partita esistente	200
T03-BadID	Ricerca partita con id invalido	400
T11-BadJson	Creazione partita con *JSON* invalido	400
T12-GameCreated	Creazione partita con *JSON* valido	201

Tabella 10: Unit test effettuato per il *Game Controller*

Attraverso, il sistema *Make* è stato predisposto il *target* **test** che oltre ad effettuare il testing di unità genera un file in cui vengono elaborate le metriche di copertura. In Figura 27, è mostrato un esempio di esecuzione del testing di unità.

5.2 Integration testing

Il testing di integrazione mette insieme il componente REST Server con un'istanza di un *database* reale. Con riferimento all'architettura della soluzione, sono state testate le funzioni del layer *Service* che utilizza la base dati ed è utilizzata dai controller. In questa attività, l'obiettivo principale è quello di verificare la coerenza dei dati e la correttezza dello schema descritto in (riferimento allo schema ER).

Analogamente al testing di unità, è stato utilizzato l'approccio del *table testing*. Ad esempio, per la procedura di salvataggio dei file appartenenti a un giocatore, è stata progettata la Tabella 11. Dal punto di vista implementativo, la funzione riceve in ingresso:

- **id**: identificativo del turno del giocatore rappresentato come intero a 64bit;
- **reader**: flusso di byte astratto dall'interfaccia *io.Reader*;

Inoltre, per rendere predicibile il valore dei valori assegnati dal *database* (ad esempio il valore delle sequenze per l'assegnazione degli ID quando viene effettuata una *INSERT*) dopo ogni *test case* viene eseguita l'operazione *TRUNCATE* con lo scopo di effettuare il *reset* di tutte le strutture del *database* e

TC	Descrizione	Risultato atteso
T51-NotAZip	Il corpo della richiesta non è uno zip	ErrNotAZip
T52-Success	Il file ZIP viene correttamente salvato	OK
T53-EmptyFile	Il file ZIP è vuoto	OK
T54-InvalidTurnID	L'ID del turno è negativo	ErrNotFound
T55-NullBody	Se l'ingresso ha valore NULL	ErrInvalidParam
T56-NotFound	Il turno non viene trovato all'interno del database	ErrNotFound

Tabella 11: Unit test effettuato per la funzione *SaveFile* del *TurnService*

rendere il *testing* riproducibile.

In Figura 28, è mostrato l'esecuzione del test di integrazione con il target *test-integration*. In particolare, sono previsti alcuni parametri di configurazione:

- *CI=1*: attraverso il parametro *DB_URI* viene specificato un'istanza già esistente per il database. L'operazione comporta la perdita di dati;
- CI non specificato: viene creata un'istanza usa e getta del database con Docker;

```
→  game-repository git:(master) make test-integration COVER_DIR=$(pwd)/coverage
Running integration test with a local docker container
?       github.com/alarmfox/game-repository/api [no test files]
?       github.com/alarmfox/game-repository/limiter     [no test files]
?       github.com/alarmfox/game-repository/model       [no test files]
ok      github.com/alarmfox/game-repository     0.119s  coverage: 8.3% of statements
ok      github.com/alarmfox/game-repository/api/game    0.031s  coverage: 56.2% of statements
ok      github.com/alarmfox/game-repository/api/robot   0.029s  coverage: 51.2% of statements
ok      github.com/alarmfox/game-repository/api/round   0.031s  coverage: 55.7% of statements
ok      github.com/alarmfox/game-repository/api/turn    1.020s  coverage: 66.7% of statements
d772ffd31d0a98955746db6ca029f30b1abf943ff5acabbb57d0ab68519354b5
        github.com/alarmfox/game-repository     coverage: 8.3% of statements
        github.com/alarmfox/game-repository/api/game    coverage: 56.2% of statements
        github.com/alarmfox/game-repository/api/robot   coverage: 51.2% of statements
        github.com/alarmfox/game-repository/api/round   coverage: 55.7% of statements
        github.com/alarmfox/game-repository/api/turn    coverage: 66.7% of statements
go tool cover -func /home/giuseppe/dev/uni/game-repository/coverage/profile -o=coverage.out
```

Figura 28: Esempio di esecuzione di *integration testing* con il comando *make*

6 Deployment

In questa sezione, si fa riferimento all'installazione del solo componente *Rest Server*; per gli altri componenti si rimanda alla guida ufficiale di installazione e configurazione. Tuttavia alla fine della sezione è comunque mostrata una versione completa del sistema installato e configurato con **docker-compose**. Al fine di avere una soluzione completa funzionante, è necessario che tutti gli elementi dell'architettura si trovino all'interno della stessa rete. In generale, l'architettura per installare l'applicativo è illustrata nel *deployment diagram* in Figura 29:

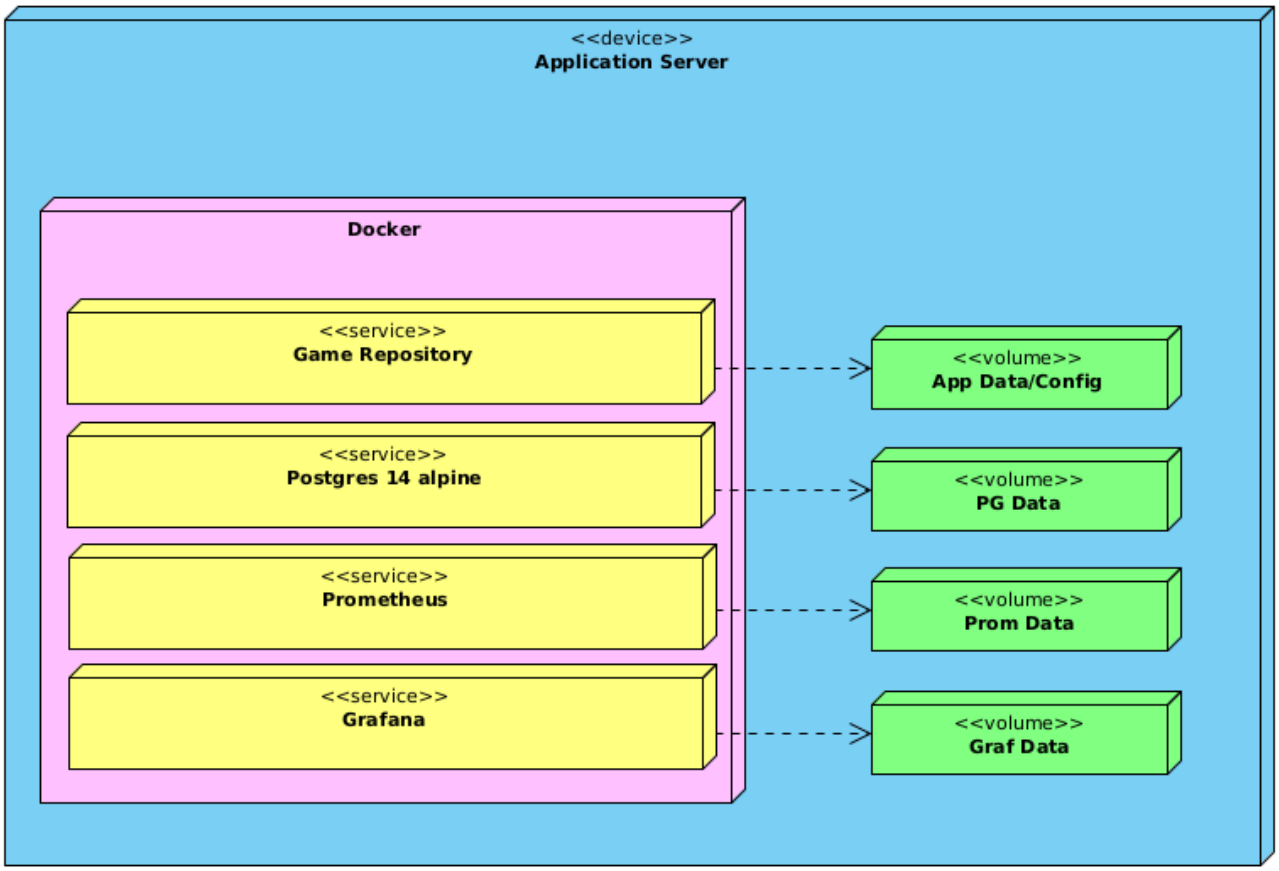

Figura 29: Deployment diagram della soluzione

6.1 Configurazione

L'applicazione si configura con un file JSON (obbligatorio anche se risulta vuoto) con la struttura riportata in Listato 1 con i valori *default* nel caso in cui non siano specificati. In particolare, l'applicazione cerca per un file chiamato *config.json* nello stesso *path* dell'eseguibile, ma è possibile specificare un percorso a scelta attraverso l'argomento CLI *config=<PATH>*.

Listing 1: Struttura file di configurazione

```
{
   "postgresUrl": "",
   "listenAddress": "localhost:3000",
   "apiPrefix": "/",
   "dataPath": "data",
   "enableSwagger": false,
   "rateLimiting": {
       "enabled": false,
       "burst": 4,
       "maxRate": 2
   },
   "authentication": {
       "enabled": true,
       "headerKey": "Authorization",
       "authEndpoint": "http://auth-service/auth",
       "method": "POST"
```

```
    }
}
```

Gli oggetti *rateLimiting* e *authentication* abilitano i moduli opzionali come indicato nel paragrafo §3.3.4.

6.2 Modalità di deploy

6.2.1 Standalone

La modalità di installazione più semplice è quella *standalone*. L'applicazione è compilata in un singolo file eseguibile che può essere compilato direttamente dal codice sorgente (usando il comando *make build*) oppure può essere scaricato dalla sezione *release* del progetto (si rimanda al paragrafo §6.3 per una spiegazione esaustiva).

6.2.2 Docker

L'applicazione può essere installata compilando un'immagine Docker per essere eseguita in un *container*. La procedura di compilazione è ottimizzata utilizzando il *multi-stage building* dell'immagine con l'adozione del sistema operativo *alpine* ottenendo un'immagine di circa 25MB. Analogamente al caso *standalone*, l'immagine può essere compilata sia dal codice sorgente (con il comando *make docker-build*), oppure è possibile scaricarne una già pronta dal *repository capas/game-repository*

Esempio di applicazione completa con docker-compose Docker compose è un'applicazione che serve ad installare *stack* applicativi (configurati in file in formato *yaml*) provvedendo a fornire per ogni applicazione servizi di utilità, come mostrato nell'elenco di seguito (non esaustivo):

- Servizi di rete: docker-compose crea una rete dedicata per ogni stack e offre (in base al nome specificato per l'applicazione) un servizio DNS;
- Dipendenze tra i servizi: i servizi possono dipendere tra loro e questo consente di decidere l'ordine di avvio;
- Sistema di healthcheck: specificato un comando, un servizio può essere marcato come "in salute" oppure "mancante";
- Politiche di riavvio: un'applicazione può essere riavviata ogni volta che c'è un crash per un numero definito di volte ecc;

In è mostrato un *docker-compose.yml* funzionanete per installare l'applicazione come mostrato in Figura 29.

Listing 2: Docker compose applicazione funzionante

```
version: '3'

volumes:
  pgdata:
  appdata:
  grafdata:
  promdata:

services:
  app:
    image: capas/game-repository:latest
    ports:
      - 3000:3000/tcp
    volumes:
      - appdata:/app/data
      - ./config.docker.json:/app/config.json
    depends_on:
      - db
```

```
db:
  image: postgres:14-alpine3.17
  environment:
    POSTGRES_PASSWORD: postgres
  volumes:
    - pgdata:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready", "-d", "postgres"]
    interval: 30s
    timeout: 60s
    retries: 5
    start_period: 80s
  ports:
    - 5432:5432

prometheus:
  image: prom/prometheus
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
    # reference https://github.com/prometheus/prometheus/blob/main/Dockerfile
    - promdata:/prometheus
  ports:
    - 3330:9090

grafana:
  image: grafana/grafana
  ports:
    - 3300:3000/tcp
  volumes:
    - grafdata:/var/lib/grafana
```

6.3 CI/CD

Con l'utilizzo di *Github Actions*, è stato automatizzato il processo di *testing* per essere eseguito prima di ogni azione sul *branch* **master** (*push*, *pull request*) per garantire uno standard minimo per la qualità del software. In particolare, ad ogni operazione viene aggiornato il file *README.md* con informazioni riguardanti la copertura delle righe di codice, lo stato dell'ultima esecuzione e il risultato dell'analisi statica del codice (ottenuta grazie all'utilizzo del servizio *Go Report Card*[6]) sotto forma di *badge*, come mostrato in Figura 30.

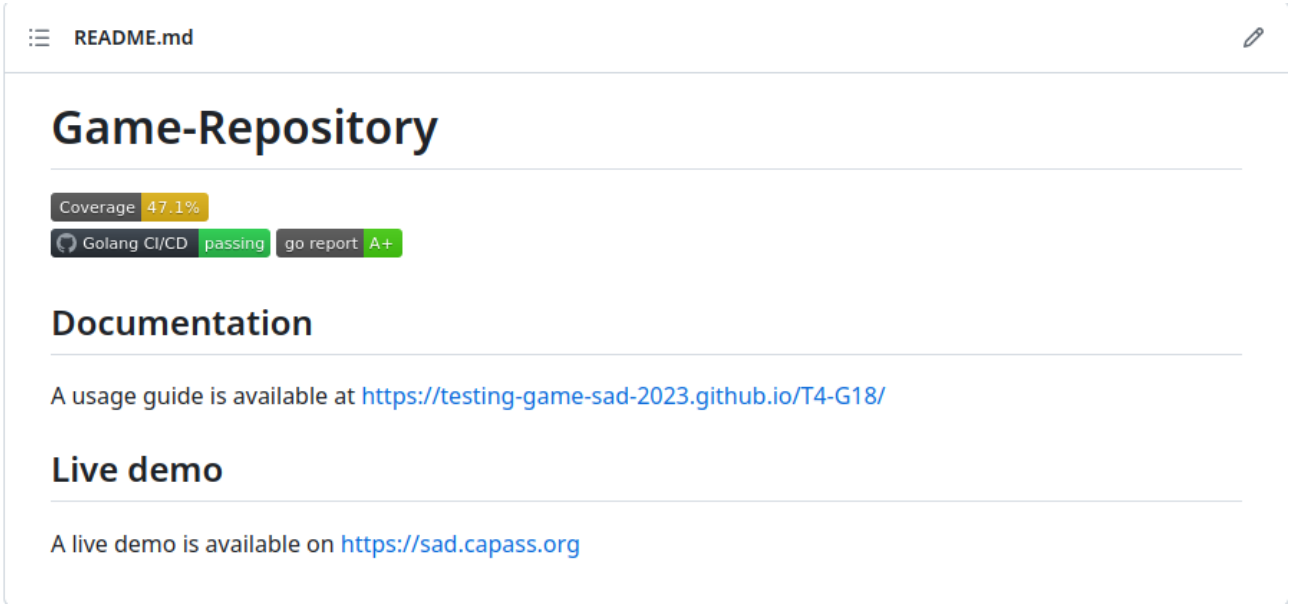

Figura 30: Badge ottenuti dopo l'esecuzione della *pipeline* CI/CD

Inoltre, è stato implementata la *Continous Delivery* attraverso *Github Actions* che viene attivata alla creazione di una *release*. All'attivazione dello script vengono eseguite le seguenti operazioni:

- Test: il *branch* scelto per la *release* viene testato (sia con test di unità e integrazione). Se questo passo fallisce, il rilascio non viene effettuato;
- Build con Docker: viene creata un'immagine Docker che viene salvata sul *repository* pubblico *capas/game-repository*;
- Deploy della versione live: con SSH, viene aggiornata l'applicazione sul server;
- Pubblicazione di *artifact*: con riferimento alla matrice architettura/sistema operativo in Tabella 12, per ciascuna cella supportata viene creato un archivio contenente: l'eseguibile, file di licenza, README.md, specifica OpenAPI e una collection Postman per gli facilitare l'integrazione con gli altri componenti;

Arch/OS	Linux	Windows	Darwin
i386	x	x	
amd64	x	x	x
arm64	x		x

Tabella 12: Matrice di compatibilità supportata

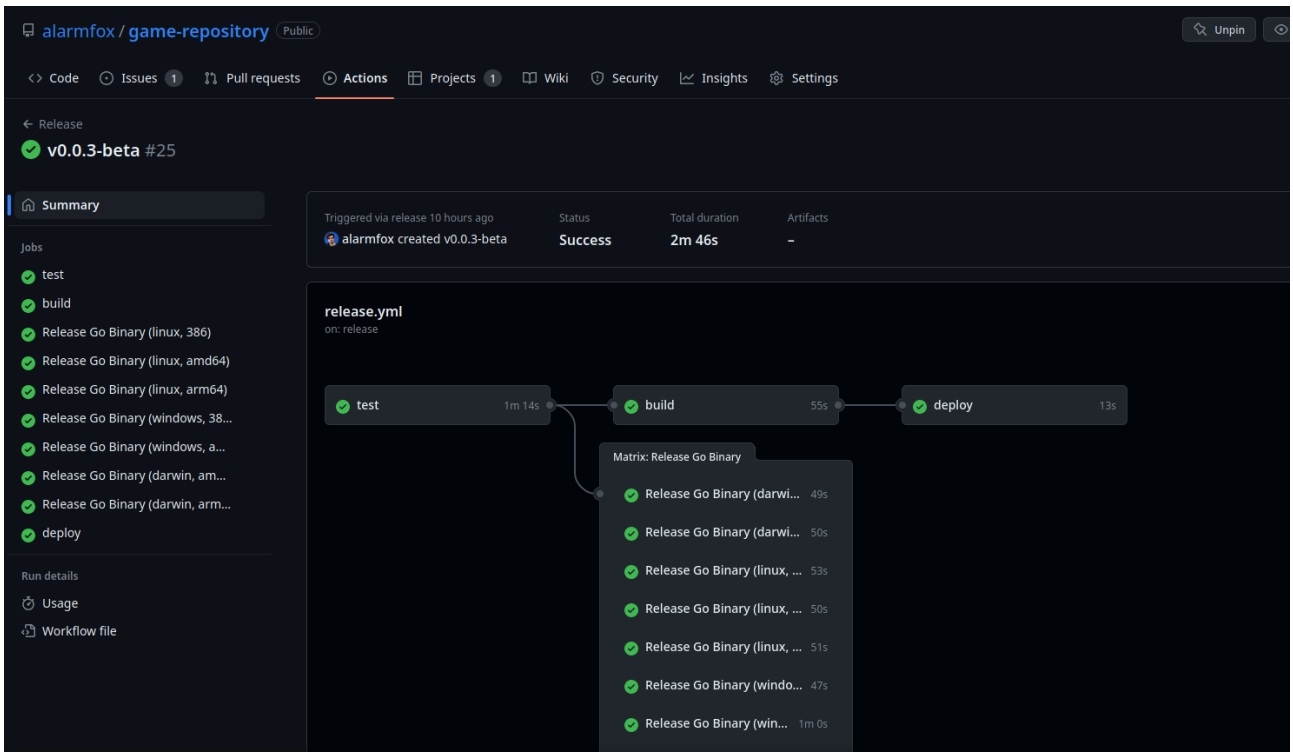

Figura 31: Esecuzione dello script CD per la release 0.0.3-beta

Documentazione online Con lo scopo di rendere la *repository self-contained*, l'applicazione offre una documentazione contenente una guida pratica all'installazione che viene compilata da una *Github Action*. In particolare, questa documentazione è generata con *mkdocs*[9] che consente di passare da semplici file di testo in *Markdown* a *HTML*. Un esempio di documentazione è mostrato in Figura 32.

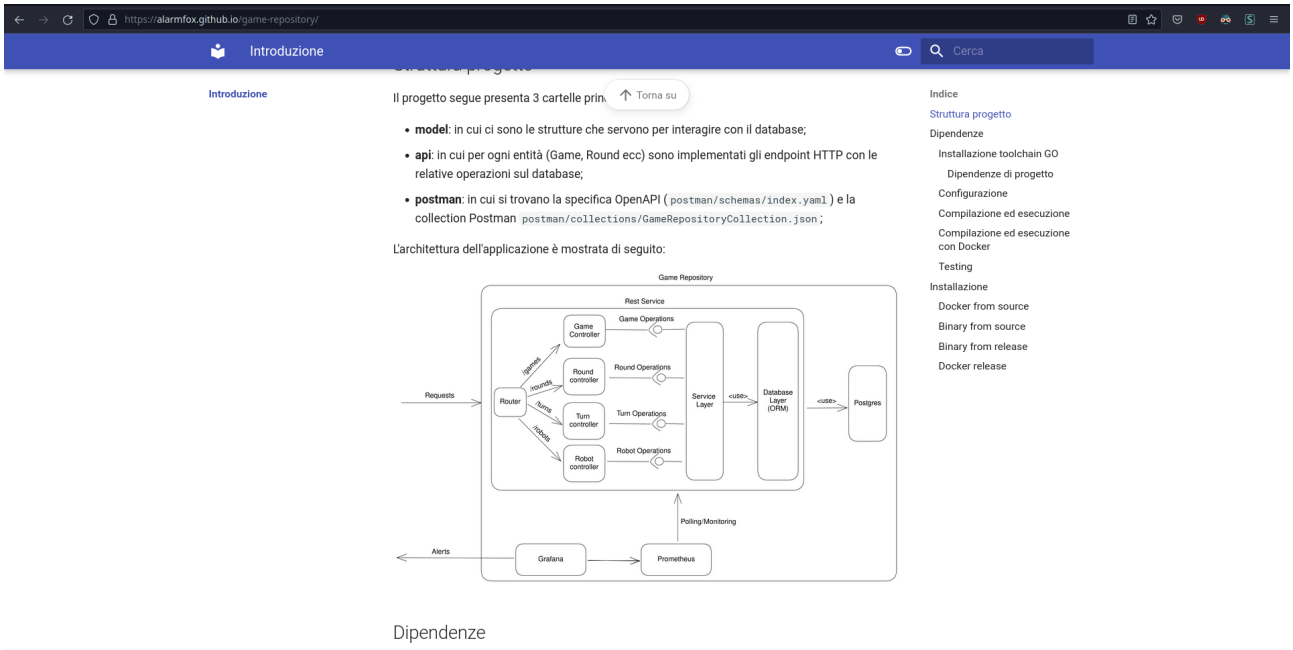

Figura 32: Screenshot della documentazione online

La documentazione è consultabile al link indicato in §1.2.1.

6.4 Integrazione con altri componenti

Essendo uno dei task con minori dipendenze e al servizio degli altri si è cercato da subito di fornire una chiara lista dei servizi messi a disposizione, interfacciandosi con i membri degli altri team. Di seguito la lista dei gruppi con il relativo task a cui sono state fornite direttamente le API e la documentazione di supporto:

- Gruppo 14 , Task 5
- Gruppo 5, Task 5
- Gruppo 8, Task 6

Coordinandosi con il gruppo G4 (task 4), nonostante le differenti tecnologie adoperate, si è scelto di esporre un'interfaccia per lo più identica, in modo che gli altri team di sviluppo potessero scegliere liberamente tra le due versioni, ma che non ci fossero discrepanze eccessive.

6.4.1 Infrastruttura di laboratorio

Il gruppo ha fornito supporto nella creazione dell'infrastruttura del laboratorio permettendo di creare diverse istanze dell'applicazione completa in cui i singoli componenti sviluppati da team indipendenti possono comunicare fra loro. In particolare, l'architettura da utilizzare è a microservizi disponendo di una macchina Windows 11 con 5 utenze dedicate all'applicazione denominate **Testing Game X** (con X da 1 a 5).

Lo scopo dell'attività è stato quello di creare, per ciascuna utenza una rete e un volume virtuale con Docker. Ogni rete è stata denominata con **testing-game-nw-x** (con x da 1 a 5); mentre ogni volume **testing-game-data-x** (con x da 1 a 5). In questo modo, ogni utente può effettuare il *deploy* del proprio stack applicativo attraverso *docker-compose* e collegare i container che devono essere in comunicazione nella rete dedicata. Per facilitare l'operazione, è stato creato un template per *docker-compose* in cui sono già dichiarati i volumi e le reti da collegare ai servizi. Inoltre, l'utilizzo di una rete comune può essere utilizzata per condividere istanze di componenti comuni. Ad esempio, se più gruppi utilizzano *MongoDB* è possibile crearne un'istanza nella rete *testing-game-nw-x* e condividerla tra i diversi stack.

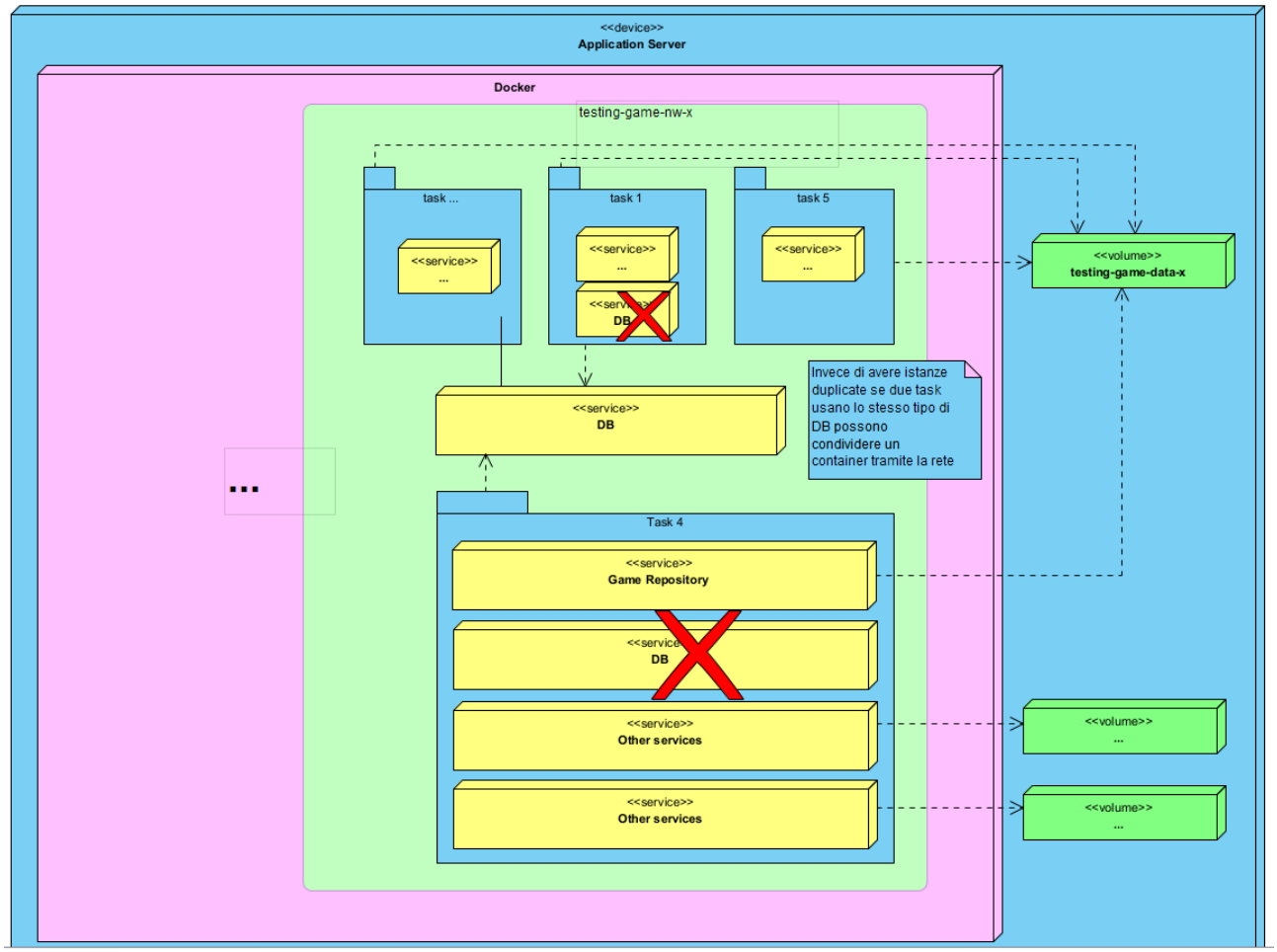

Figura 33: Deploy con network e DB condiviso

Riferimenti bibliografici

[1] *Borrow Checker*. URL: https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html (visitato il 15/04/2023).

[2] *Cargo*. URL: https://doc.rust-lang.org/cargo/ (visitato il 15/04/2023).

[3] *Comparison between Go, Java and PHP*. URL: https://web-frameworks-benchmark.netlify.app/result?asc=0&f=spring&l=rust,java&order_by=level64 (visitato il 10/04/2023).

[4] *Data Transfer Object Pattern*. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html (visitato il 15/04/2023).

[5] *Discord handles trillions of messages*. URL: https://discord.com/blog/how-discord-stores-trillions-of-messages (visitato il 10/04/2023).

[6] *Go Report Card*. URL: https://goreportcard.com/ (visitato il 15/04/2023).

[7] Benjamin Hippchen et al. "Designing microservice-based applications by using a domain-driven design approach". In: *International Journal on Advances in Software* 10.3&4 (2017), pp. 432–445.

[8] *Http benchmark*. URL: https://www.techempower.com/benchmarks/#section=data-r21&hw=ph&test=json (visitato il 10/04/2023).

[9] URL: https://www.mkdocs.org/ (visitato il 13/07/2023).

[10] *Pingora*. URL: https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/ (visitato il 10/04/2023).