

Documentazione T10-G40
Software Architecture Design 2022/23

Lorenzo Pannone - Matr. M63/1492 Pasquale Riello - Matr. M63/1516
Prof.ssa Anna Rita Fasolino

Indice

1 Introduzione	3
1.1 Caso di studio	3
1.1.1 Ipotesi importante	4
1.1.2 Task assegnato	4
1.1.3 Metodologie adottate	4
1.2 Workflow diagram	5
2 Analisi dei requisiti	7
2.1 User Stories	7
2.2 Modello dei casi d'uso	7
2.2.1 Attori	8
2.3 Scenari	9
2.3.1 Create game	10
2.3.2 Write Code	10
2.3.3 Inspect ClassUnderTest	11
2.3.4 Run with JaCoCo	11
2.3.5 Compile	12
2.3.6 Run	12
2.3.7 Add Class	13
2.4 Dipendenze tra i Task	13
3 Progettazione	15
3.1 L' architettura a microservizi scelta	15
3.1.1 Gateway Pattern	15
3.1.2 UI Gateway	15
3.1.3 API Gateway	15
3.2 Component Diagram	16
3.2.1 UI Gateway	17
3.2.2 API Gateway	17
3.2.3 Student Front End	17
3.2.4 Student Repository	17
3.2.5 Admin Front End	17
3.2.6 Admin Repository	18
3.2.7 Game Engine	18
3.2.8 Student Test Runner	18
3.2.9 Test Class Manager	18
3.2.10 Java Class Under Test Repository	18
3.2.11 Randoop Runner	18
3.2.12 Evosuite Runner	18
3.2.13 Game Repository	18
3.3 Composite Structure Diagram	19
3.3.1 Modifiche apportate	19

3.4	Gateway Sequence Diagrams	22
3.4.1	UI Gateway	22
3.4.2	API Gateway	23
3.5	Nuovi Sequence Diagrams	24
3.5.1	Inspect ClassUnderTest	24
3.5.2	Create Game	25
3.5.3	Run	26
3.5.4	Run with JaCoCo	27
3.5.5	Compile	28
3.5.6	Add class	29
3.6	Activity Diagrams	30
3.6.1	Add Class	30
4	Deployment	31
5	Testing	32
5.1	Tool utilizzati	32
5.2	Casi di Test	32
5.2.1	Login Test	32
5.2.2	Editor Test	35
5.3	Risultati	41
6	Installazione	42
6.1	Passo 1	42
6.2	Passo 2	42
6.3	Passo 3	43
7	Guida alle future integrazioni	44
7.1	Integrazione Container	44
7.2	Integrazione con UI Gateway	44
7.3	Integrazione con API Gateway	45
7.4	Integrazione Installer	45

Capitolo 1

Introduzione

Il testo qui presentato documenta l'attività di integrazione dei microservizi che compongono il gioco educativo "Man vs Automated Testing Tools challenges" ideato nel contesto del progetto ENACTEST.

1.1 Caso di studio

Il caso di studio 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.

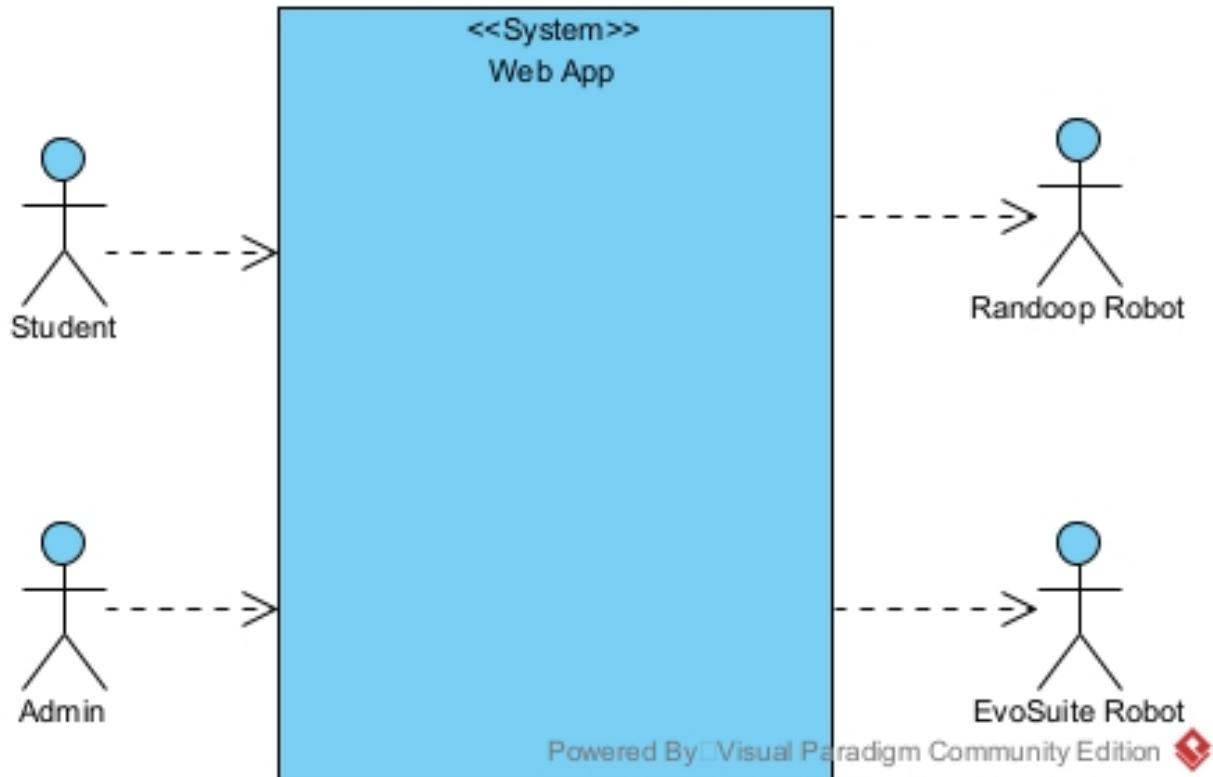


Figure 1.1: Diagramma di contesto ad alto livello

1.1.1 Ipotesi importante

A causa del non completo funzionamento del Task 8, si procederà all'esclusione dall'integrazione del robot EvoSuite, che potrà essere integrato successivamente in modo agevole nello stesso modo del robot Randoop.

Nota sulla documentazione

Per quanto possibile, si è cercato di inglobare all'interno di questa documentazione parti importanti di documentazione prodotta da altri team, in maniera tale da fornire una visione quanto più completa dell'intero sistema software. Tuttavia, per una più completa comprensione, si raccomanda di consultare prima le documentazioni prodotte dai singoli Task per poi comprendere le modifiche da noi apportate ai singoli task per renderli funzionanti, e quindi delle variazioni del loro funzionamento in questo contesto.

1.1.2 Task assegnato

L'architettura della Web Application è stata divisa in 8 componenti da T1, T2-3, T4...etc. fino a T9. Il task assegnato T10 si occupa dell'integrazione di questi componenti al fine di ottenere la web application finale funzionante. I task sono stati prodotti più volte da gruppi diversi. I task specifici da integrare sono riportati in tabella (fig 1.2).

Cluster	T1	T2-3	T4	T5	T6	T7	T8	T9	Task di Integrazione
1	G11	G1	G18	G2	G12	G31	G21	G19	T10

Figure 1.2: Tabella task da integrare

1.1.3 Metodologie adottate

Per lo sviluppo del nostro software abbiamo adottato un **approccio agile** basato sul paradigma **SCRUM** ovvero un Framework agile per la gestione iterativa ed incrementale del ciclo di sviluppo del software. La nostra metodologia di lavoro si è basata su *sprint* settimanali, durante i quali i componenti del gruppo si sono concentrati su un insieme di funzionalità specifiche. Ogni sprint è stato preceduto da una pianificazione in cui sono stati definiti gli obiettivi dell'iterazione e le attività necessarie per raggiungerli.

Si è fatto intensamente uso della pratica del **pair programming**, estremamente idonea al team dato il limitato numero di persone componenti. Questa pratica ci ha consentito di migliorare la qualità del codice e di ridurre il numero di errori, in quanto ogni riga di codice è stata esaminata da almeno due persone. Ciò è stato possibile grazie a strumenti quali **git**, fondamentale per la gestione e l'integrazione delle modifiche individuali, la funzione **code share** di Visual Studio Code che ci ha permesso di scrivere codice a quattro mani e di poter revisionare istantaneamente l'un l'altro. Per il testing è stato utilizzato **Selenium**, ecosistema per automatizzare il browser, mentre **Nginx** è stato utilizzato come reverse proxy per rendere più fluida la navigazione web, facendo in modo da rendere tutta l'applicazione web disponibile sulla porta 80.

Tramite tre iterazioni, di lunghezza che oscilla tra 1-2 settimane, siamo giunti al prodotto finale completo di documentazione. Di seguito è visualizzato il prodotto della pianificazione delle iterazioni (fig 1.3).

Iteration Planning



Figure 1.3: Iteration Planning

1.2 Workflow diagram

Lo scenario di gioco che è qui sviluppato è quello descritto in figura 1.4, con singolo giocatore, singolo turno, singolo round. Per descriverlo è stato usato un workflow diagram, sequenze logiche di attività che, insieme, modellano processi aziendali. Da questo schema è facile individuare i microservizi sviluppati.

CAPITOLO 1. INTRODUZIONE

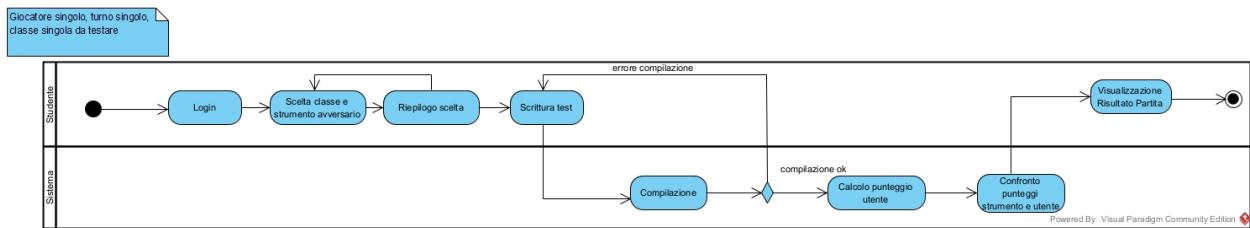


Figure 1.4: Primo scenario di gioco

Capitolo 2

Analisi dei requisiti

Per i singoli requisiti si rimanda alla documentazione degli altri task dove vi è un'analisi molto approfondita.

2.1 User Stories

Ulteriori requisiti sono stati individuati e il più importante è quello che riguarda la generazione dei test del robot, contestualmente all'inserimento della classe all'interno della repository da parte dell'admin. Il motivo sta nel fatto che questa operazione richiede tempo e, dunque, va disaccoppiata dalla partita: i risultati devono essere già a disposizione per non rallentare l'esperienza di gioco dello studente e vanno precalcolati. Questi nuovi requisiti sono espressi nelle seguenti user stories.

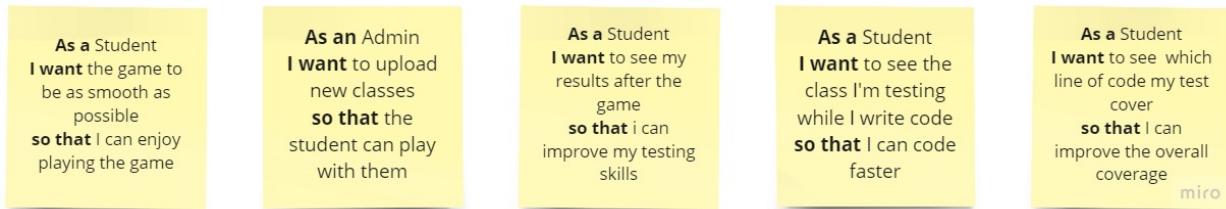


Figure 2.1: User Stories importanti individuate

2.2 Modello dei casi d'uso

Si è deciso di realizzare un diagramma dei casi d'uso generale di tutte le funzionalità disponibili in questa integrazione. Sono evidenziati in verde i casi d'uso qui prodotti e/o resi funzionanti.

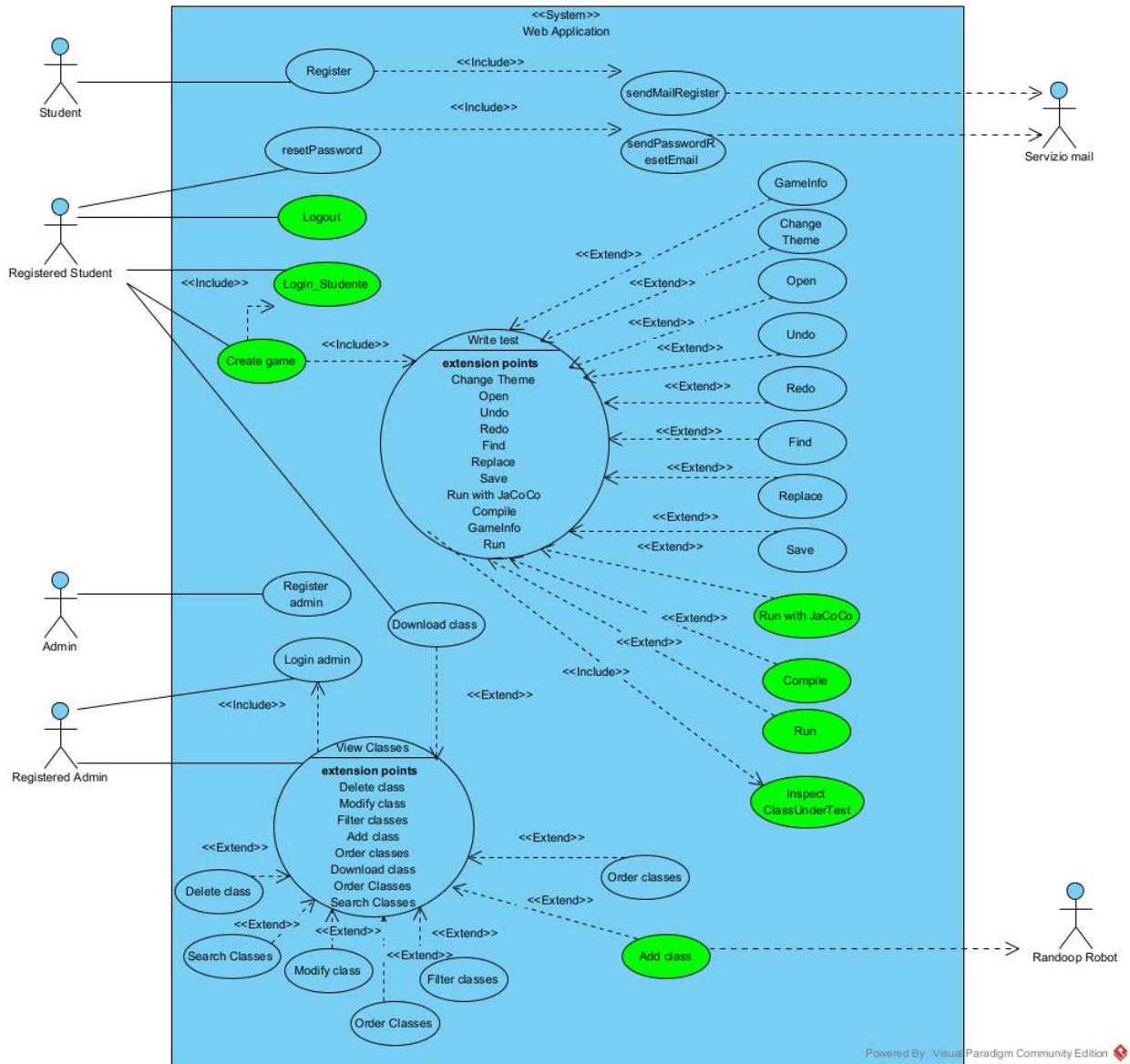


Figure 2.2: Diagramma dei casi d'uso

Si è cercato per quanto possibile di mantenere coerenti i nomi dei casi d'uso con quelli dei singoli task, anche se è stato necessario per i casi d'uso relativi all'amministratore, tradurre i nomi in inglese per mantenere coerenza linguistica. "View classes" e "Create Game" includono il login che è necessario fare preliminarmente. Sono poi disponibili una gamma di casi d'uso d'estensione, opzionali per i due tipi di utente.

Si noti che "Run" in realtà consiste nella sottomissione del tentativo da parte dello studente, ma per mantere coerenza con il task 6 è stato lasciato così e non è stato rinominato in "Submit".

2.2.1 Attori

Il **"Registered Student"** è colui che può giocare la partita contro il robot scrivendo il suo test. L' **"Registered Admin"** è colui che gestisce le classi con cui è possibile giocare all'interno del sistema. Se questi due attori non sono registrati sono rinominati Student e Admin.

"Randoop Robot" è un attore secondario che interagisce con il caso d'uso di aggiunta di una nuova classe da parte dell' Admin: contenutualmente il robot produrrà i suoi test.

”Servizio Mail” è ancora un attore secondario che interagisce nella fase di registrazione e recupero password con lo studente.

2.3 Scenari

Verranno qui descritti gli scenari dei casi d'uso evidenziati in verde, in cui sono presenti modifiche. Per i restanti, si rimanda alle documentazioni del Task 1 per i casi d'uso dell'Admin; al Task 2-3 per i casi d'uso di autenticazione; al task 6 per le funzioni di estensione semplici del caso d'uso ”Write Code”.

Le modifiche più importanti sono in ”Add class” dove si è deciso di eseguire, localmente e contestualmente all'inserimento della classe da parte dell'admin nella repository, il robot Randoop di generazione automatica dei test. Dunque lo scenario è diverso da quello originale.

Caso d'uso <i>Login</i>	
Attore primario	Studente registrato
Attore secondario	-
Descrizione	Permette ad uno studente di effettuare il login e iniziare a giocare.
Pre-Condizioni	Lo studente deve essersi registrato.
Sequenza di eventi principale	1) Lo studente registrato, nella schermata di login, inserisce e-mail e password nell'apposito spazio; 2) Clicca su "Accedi".
Post-Condizioni	Lo studente visualizza la schermata di inizio della partita
Casi d'uso correlati	-
Sequenza di eventi alternativi	1) Il login fallisce se e-mail e/o password sono errati, mostrando un messaggio di errore; 2) Se l'utente ha dimenticato la password può reimpostarla.

Figure 2.3: Login

Caso d'uso <i>Logout</i>	
Attore primario	Studente registrato
Attore secondario	-
Descrizione	Permette ad uno studente registrato di effettuare il logout.
Pre-Condizioni	Lo studente deve essersi loggato.
Sequenza di eventi principale	1) Clicca su "Logout".
Post-Condizioni	Lo studente torna alla pagina di login.
Casi d'uso correlati	-
Sequenza di eventi alternativi	-

Figure 2.4: Logout

2.3.1 Create game

Lo studente può iniziare una partita dopo aver effettuato il login. Si vuole sia possibile per l'utente scegliere quale robot sfidare e con quale livello, su quale classe scrivere i test e poi salvare queste informazioni opportunamente. Dopo la creazione della partita, si avvia il caso d'uso "Write Code".

Caso d'uso:	Create Game
Attore primario	Student
Attore secondario	-
Descrizione	Studente sceglie la classe da testare e il robot con il livello da affrontare
Pre-Condizioni	Lo studente ha effettuato il login
Sequenza di eventi principale	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando lo studente effettua il login. 2. Il sistema mostra una lista di classi. 3. Lo studente sceglie la classe 4. Il sistema mostra i robot e i lievelli disponibili. 5. Lo studente sceglie il robot e il livello e clicca Submit 6. Il sistema mostra un riepilogo allo studente 7. Lo studente conferma le sue scelte. 8. Il sistema salva le informazioni e mostra allo studente l'editor
Post-Condizioni	Lo studente può scrivere il suo test
Casi d'uso correlati	nessuno
Sequenza di eventi alternativi	<p><i>Lo studente non clicca Submit ma Back</i> <i>Il sistema lo riporta alla schermata di selezione classi</i></p>

Figure 2.5: Create Game

2.3.2 Write Code

Dopo la creazione della partita da parte dello studente, sarà possibile scrivere effettivamente il test e attivare opzionalmente alcune funzionalità sull'editor.

Caso D'uso:	Write Code
Attori:	Player
Precondizioni:	Partita avviata
Sequenza degli Eventi:	Il giocatore scrive all'interno dell'editor di testo un codice Java
Postcondizioni:	
Sequenza Alternativa:	Il player possiede già una classe di test e vuole caricarla
Postcondizioni Alternative:	Caricamento da locale della classe di test

Figure 2.6: Write Code

2.3.3 Inspect ClassUnderTest

All'avvio dell'editor ("Write Code") si vuole si carichi in un apposito riquadro la classe che lo studente ha scelto di testare.

Caso D'uso:	<i>Inspect ClassUnderTest</i>
Attori:	Player
Precondizioni:	Scelta della ClassUnderTest dalla Repository
Sequenza degli Eventi:	La Class Under Test viene caricata all'interno dell'editor di testo
Postcondizioni:	La Class Under Test viene visualizzata nella apposita finestra di gioco
Sequenza Alternativa:	
Postcondizioni Alternative:	

Figure 2.7: Inspect ClassUnderTest

2.3.4 Run with JaCoCo

Si vuole che, alla pressione del tasto "Run with JaCoCo", l'utente possa visualizzare la copertura del suo test. In particolare si vuole che il codice della classe under test sia evidenziato in maniera tale da indicare le linee coperte dal test.

Caso D'uso:	<i>Run with Jacoco</i>
Attori:	Player
Precondizioni:	Classe di test compilata
Sequenza degli Eventi:	Click sul bottone
Postcondizioni:	Colorazione delle linee di codice coperte dalla classe di test attraverso Jacoco
Sequenza Alternativa:	Il player non fa click sul pulsante
Postcondizioni Alternative:	Assenza di Test di Copertura con Jacoco

Figure 2.8: Run with JaCoCo

2.3.5 Compile

Si vuole che, alla pressione del tasto "Compile", sia mostrato l'output della compilazione a video, in una apposita sezione.

Caso D'uso:	Compile
Attori:	Player
Precondizioni:	Classe di test implementata
Sequenza degli Eventi:	<i>Click sul pulsante</i> relativo all'operazione di compilazione
Postcondizioni:	Visualizzazione dell'esito della compilazione
Sequenza Alternativa:	Il player non fa click sul pulsante
Postcondizioni Alternative:	Il codice non è compilato

Figure 2.9: Compile

2.3.6 Run

Si vuole che, alla pressione del tasto di "Run", l'utente sottometta il suo tentativo contro il robot e vengano visualizzati i risultati della partita.

Caso D'uso:	Run
Attori:	Player
Precondizioni:	Classe di test correttamente compilata
Sequenza degli Eventi:	<i>Click sul pulsante</i> relativo all'operazione di esecuzione
Postcondizioni:	Visualizzazione dei risultati all'interno della Console
Sequenza Alternativa:	Il player non fa click sul pulsante
Postcondizioni Alternative:	Non vengono mostrati in console i risultati della partita

Figure 2.10: Run

2.3.7 Add Class

Si vuole che l'admin possa inserire una classe nella repository e che contestualmente a ciò, vengano prodotti i test e calcolate le coperture di Randoop.

Caso d'uso:	Add Class
Attore primario	Admin
Attore secondario	Robot Randoop
Descrizione	L'admin aggiunge una classe alla repository delle class under test
Pre-Condizioni	L'admin ha effettuato il login
Sequenza di eventi principale	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando l'amministratore clicca sul pulsante Add class. 2. Il sistema mostra un form da compilare 3. L'admin inserisce il nome, la data, la difficoltà della classe, optionalmente una descrizione e tre categorie, e fa l'upload del file della classe 4. Il sistema salva la classe 5. Il Robot Randoop genera i test relativi a quella classe divisi in livelli 6. Il sistema calcola copertura raggiunta si salvano i risultati opportunamente 7. Il sistema mostra le classi inserite.
Post-Condizioni	<p>La classe inserita è disponibile allo studente durante la creazione della partita. Il Robot Randoop e il livello sono selezionabili dallo studente durante la creazione della partita. Il punteggio è disponibile per l'elaborazione del vincitore alla conclusione della partita</p>
Casi d'uso correlati	<i>nessuno</i>
Sequenza di eventi alternativi	<i>nessuno</i>

Figure 2.11: Add Class

2.4 Dipendenze tra i Task

Per effettuare l'integrazione dei vari Task è stato necessario tracciare e mettere in evidenza quelle che sono le dipendenze tra i vari Task, ponendo quindi l'attenzione sulle operazioni (casi d'uso) che richiedono l'utilizzo di servizi offerti da altri.

Per fornire, quindi, una panoramica abbastanza completa, si è deciso di rappresentare, in forma tabellare, le dipendenze tra le varie operazioni mettendo in evidenza, per ogni operazione (Operations):

- i task da cui dipende (Collaborators)
- le operazioni dei task da cui dipende che utilizza
- le REST API endpoint che permettono di effettuare quelle operazioni

CAPITOLO 2. ANALISI DEI REQUISITI

Service	Operations	Collaborators
Task 1	uploadClasse()	Task 4: creazioneRisultatiRobot(): /robots POST
Task 5	visualizzaClassi() visualizzaRobot() salvaPartita()	Task 1: elencaClassi(): /home GET Task 4: ricercaRisultatiRobot(): /robots GET Task 4: creazionePartita(): /games POST creazioneRound() /rounds POST creazioneTurno() /turns POST
Task 6	inspectClassUnderTest() compile() getCoverageReport() run()	Task 1: downloadClasse(): /downloadFile GET Task 7: compileExecuteCoverage(): /compile-and-codecoverage POST Task 7: compileExecuteCoverage(): /compile-and-codecoverage POST Task 7: compileExecuteCoverage(): /compile-and-codecoverage POST Task 4: modificaPartita(): /games PUT modificaRound(): /rounds PUT modificaTurno(): /turns PUT

Figure 2.12: Tabella delle dipendenze

Si rimanda alla documentazione dei singoli task per la documentazione delle API presenti nella tabella.

Capitolo 3

Progettazione

3.1 L' architettura a microservizi scelta

L'architettura a microservizi della web application realizzata si sviluppa come rappresentato in figura 3.1. Il client può accedere all'applicazione esclusivamente interfacciandosi con i due gateway tramite internet. I microservizi non sono quindi esposti all'esterno della loro rete locale: ciò aggiunge un ulteriore livello di sicurezza.

3.1.1 Gateway Pattern

Alla base c'è il tentativo di realizzare un'applicazione sicura e scalabile: ciò è reso possibile grazie all'utilizzo del **Gateway Pattern**.

Esso consiste nell'introdurre dei componenti che costituiscono gli unici punti di accesso ai microservizi: ciò permette di implementare dei meccanismi di autenticazione, autorizzazione, routing, load-balancing e API composition (una sorta di facade) delle richieste molto più versatili ed efficienti.

3.1.2 UI Gateway

Tale componente è stato realizzato utilizzando *Nginx*, realizzando di fatto un **reverse proxy**: la sola responsabilità di questo gateway è quella di effettuare il routing delle richieste relative al frontend (UI). Ciò ha permesso di rendere disponibile l'intera applicazione alla porta 80 (porta predefinita per le connessioni HTTP), ottenendo così una certa uniformità.

3.1.3 API Gateway

Tale componente è stato realizzato utilizzando Spring Boot insieme a Netflix Zuul, realizzando, anche in questo caso, un reverse proxy ma con più responsabilità: si occupa di effettuare il routing, di gestire autenticazione e autorizzazione delle richieste relative alle REST API.

Ciò ha permesso di rendere disponibili tutte le API al di sotto del percorso URL ”/api/”, ottenendo così anche qui una certa uniformità.

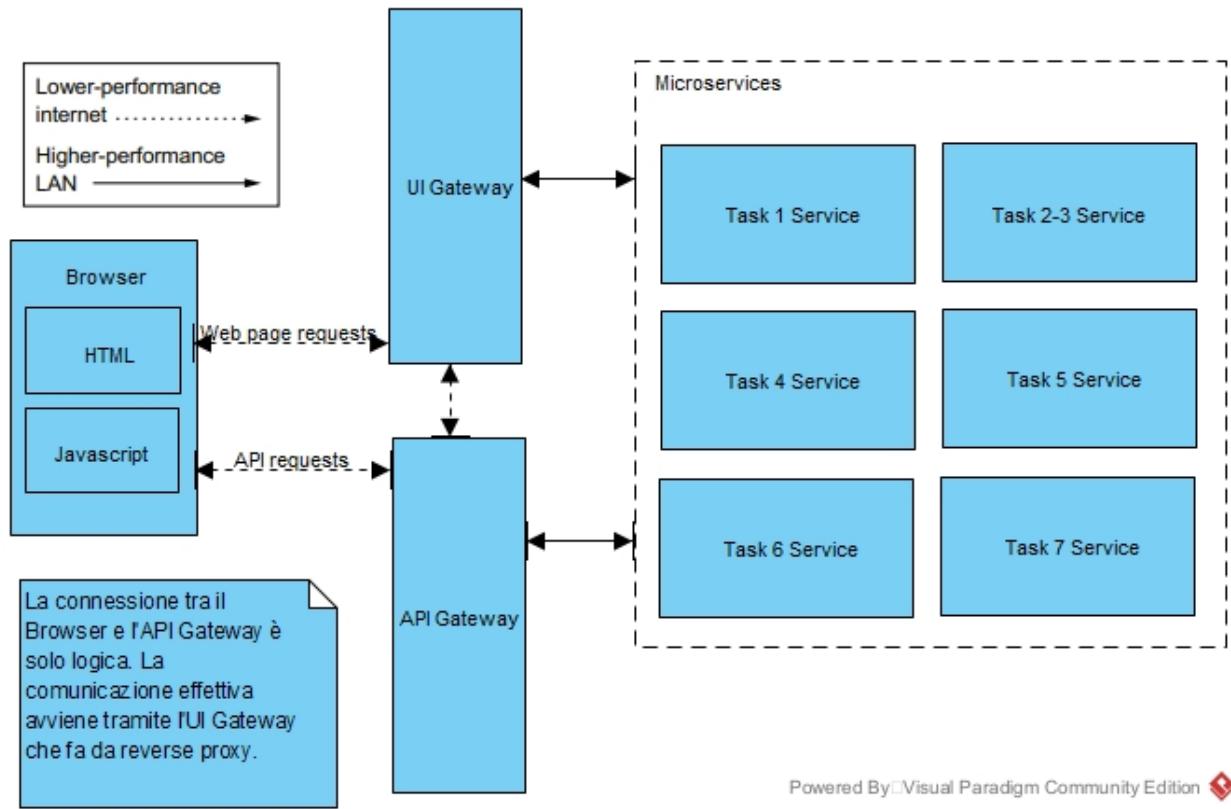


Figure 3.1: Architettura generale sistema

3.2 Component Diagram

Il component diagram ci aiuta a capire a tempo di esecuzione quali entità ci sono nel sistema e come interagiscono tra di loro per adempiere alle funzioni richieste.

Si noti che i componenti che svolgono funzioni relative allo studente sono disaccoppiate da quelli dell'amministratore: il motivo sta nel differente approccio all'autenticazione.

Per quanto riguarda l'accesso dello Studente, grazie al servizio di autenticazione offerto dal componente Student Repository tramite token JWT, è stato possibile implementare un controllo di sicurezza ulteriore grazie all'uso dell'API Gateway, che però non protegge le API lato Admin.

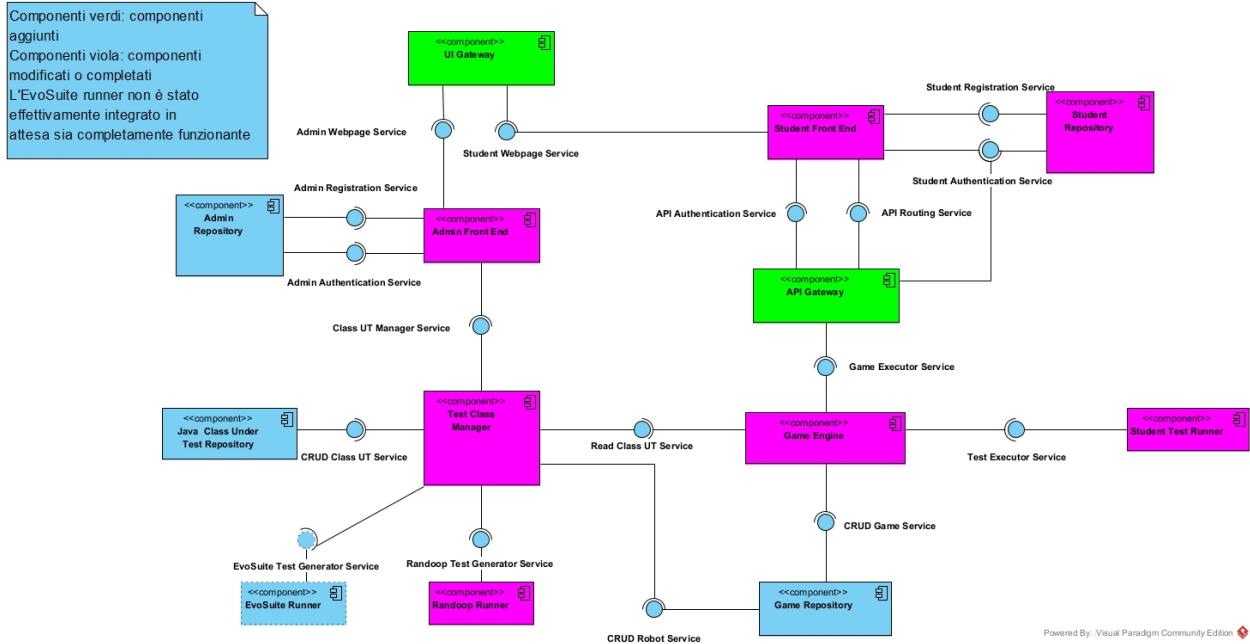


Figure 3.2: Component Diagram

3.2.1 UI Gateway

Gestisce le richieste alla User Interface implementando il meccanismo di routing.
Usa i servizi dello Student e dell'Admin Front End per fornire le corrette pagine web al client che le richiede.
Realizzato con Nginx.

3.2.2 API Gateway

Gestisce le richieste alle API implementando meccanismi di routing, autenticazione e autorizzazione. Introduce quindi un livello di sicurezza che si basa sul controllo di validità del token JWT memorizzato in uno specifico cookie.

Usa i servizi offerti dal Game Engine per adempiere alle richieste di cui se ne è verificata la sicurezza.
Realizzato con Spring Boot e Netflix Zuul.

3.2.3 Student Front End

Fornisce le User Interfaces navigabili dall'utente, dal login alle schermate di gioco.
Realizzato con Spring Boot e Code Mirror.

3.2.4 Student Repository

Si occupa della gestione dell'autenticazione dell'utente Studente: mantiene in un database i dati relativi agli studenti registrati e gestisce l'autenticazione tramite un token JWT memorizzato in un cookie.
Realizzato con Spring Boot e MySQL.

3.2.5 Admin Front End

Fornisce le User Interfaces di gestione delle classi e di accesso per l'utente Admin.
Realizzato con Spring Boot.

3.2.6 Admin Repository

Si occupa della gestione dell'autenticazione dell'utente Admin: mantiene in un database gli amministratori registrati.

Realizzato con Spring Boot e MongoDB.

3.2.7 Game Engine

Si occupa della gestione della partita.

Usa i servizi offerti dallo Student Test Runner e dal Game Repository per un corretto svolgimento della partita: ottiene i risultati della compilazione e della coverage del test dello studente, i risultati del robot e salva le informazioni della partita in corso. Usa, inoltre, i servizi del Test Class Manager al fine di ottenere la classe che l'utente ha scelto di testare.

Realizzato con Spring Boot.

3.2.8 Student Test Runner

Si occupa della compilazione e del calcolo della coverage del codice di test sviluppato dall'utente studente che riceve dal Game Engine.

Realizzato con Maven, JaCoCo, Spring Boot.

3.2.9 Test Class Manager

Si occupa della gestione delle classi giocabili inserite dall'Admin.

Usa i servizi del componente Java Class Under Test Repository per mantenere i dati delle classi.

Usa il servizio di compilazione e calcolo della coverage dei due robot, ogni volta che viene inserita una classe nel sistema.

Usa il servizio del Game Repository per salvare e rendere disponibili al Game Engine i risultati prodotti dai robot.

Realizzato con Spring Boot.

3.2.10 Java Class Under Test Repository

Si occupa del salvataggio delle classi inserite dall'Admin: salva i file della classi sul filesystem e le informazioni su di essa in un database.

Realizzato con MongoDB e Spring Boot.

3.2.11 Randoop Runner

Genera test a partire da una classe con il robot Randoop e li salva localmente, per poi calcolarne la copertura con il tool Emma. La generazione dei test avviene secondo più livelli in modo dinamico, se si generano fissate delle impostazioni, più test con diversa copertura, li si raggruppa in livelli.

Realizzato con Randoop e Emma.

3.2.12 Evosuite Runner

Dovrebbe generare test data una classe con il robot EvoSuite e calcolarne la copertura.

Realizzato con EvoSuite.

3.2.13 Game Repository

Mantiene lo storico delle partite, le classi giocabili, i risultati dei robot per livello e tutte le informazioni necessarie al corretto svolgimento della partita.

Realizzato con Chi (GoLang) e PostgreSQL.

3.3 Composite Structure Diagram

Nel seguente diagramma, si è riportato la mappatura di chi, tra i vari task, ha sviluppato quale componente visibile a tempo di esecuzione. La notazione scelta, per mantenere coerenza con Component Diagram è di inserire un package per ogni componente che specifichi quali task ne sono responsabili e ne contribuiscono al funzionamento.

Mentre alcuni componenti sono completamente realizzati da singoli task, ce ne sono altri come lo Student Front End e il Game Engine che sono distribuiti su due task.

Lo Student Front End è realizzato dal task 2-3 per quanto riguarda la parte dell'accesso al gioco, dunque le schermate di login, registrazione, recupero password. Per la parte che riguarda il gioco vero e proprio, il front end è stato realizzato nel task 5.

Il Game Engine, invece, è realizzato nella parte di salvataggio iniziale della partita, dal task 5, e per il resto delle funzionalità dal task 6 (run, compile etc.).

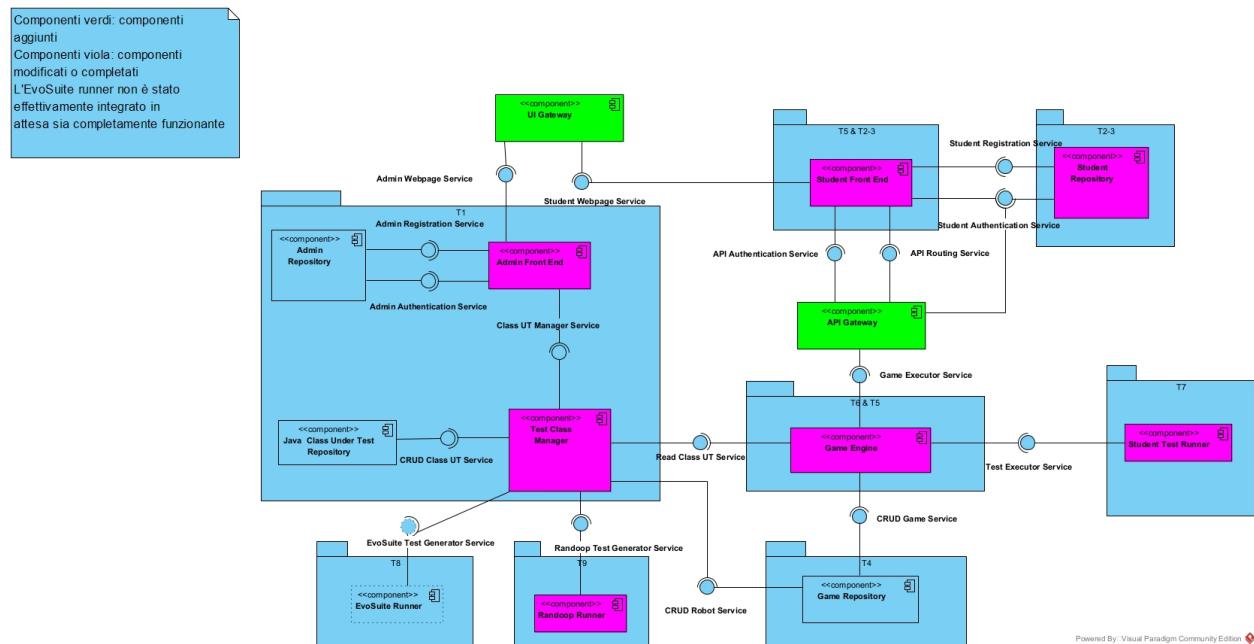


Figure 3.3: Composite Structure Diagram

3.3.1 Modifiche apportate

Nei seguenti paragrafi saranno riportate tutte le modifiche effettuate ai singoli Task per rendere l'intero sistema funzionante e permetterne così l'integrazione.

Task 1

- Inizialmente le pagine HTML che costituiscono l'Admin Front End erano disponibili solamente in locale (sul File System): si è fatto in modo che queste venissero servite direttamente dal WebServer (Spring), altrimenti sarebbero risultate irraggiungibili da qualsiasi utente.
- È stata aggiunta la generazione dei test e il calcolo della copertura da parte dei robot (attualmente solo Randoop) contestualmente all'inserimento della classe da parte dell'amministratore. Per fare ciò si è reso necessario utilizzare il volume condiviso "VolumeT9" con il Task 9.
- È stata effettuata la sostituzione della versione di Java da 17 a 8 per poter utilizzare l'eseguibile .jar del Task 9 che richiede Java 8.

Task 2-3

- È stata aggiunta la creazione del Cookie contenente il token JWT quando viene effettuato l'accesso da parte dello studente.
- È stata aggiunta la rimozione del Cookie contenente il token JWT quando viene effettuato il logout da parte dello studente.
- È stata aggiunta la validazione del token JWT contenuto nel Cookie apposito ad ogni richiesta di Front End.
- È stata aggiunto un nuovo endpoint ("validateToken") alla REST API che permette di effettuare la validazione dei token JWT da parte di altri servizi: viene utilizzato dall'API Gateway per verificare l'autenticazione delle richieste.

Task 5

- Il salvataggio della partita utilizzando i file .csv, disponibili in locale, è stato sostituito dal salvataggio che viene effettuato utilizzando il Game Repository (realizzato dal Task 4).
- È stata aggiunta la validazione del token JWT contenuto nel Cookie apposito ad ogni richiesta di Front End.
- È stata aggiunto il recupero del codice delle classi giocabili, dal Task 1.
- È stata aggiunto il recupero dei robot e dei livelli disponibili per ogni classe dal Task 4.
- È stata realizzata una gestione "stateless" delle richieste, andando a rimuovere l'utilizzo di variabili globali che, di fatto, rendevano impossibile l'uso concorrente del sistema da parte di più studenti.
- Sono stati rimossi tutti gli endpoint della REST API che non erano previsti: "/download", "/send-Variable" e "/login-variables" (il primo perché quella funzionalità è prevista dal Task 1 e gli altri due perché venivano utilizzati per la gestione delle variabili globali).

Task 6

- Sono stati rimossi tutti gli endpoint della REST API che non erano previsti: "/getResultsXML" e "/getResultsRobot" che venivano utilizzati lato client nel momento in cui avveniva il "Submit" della partita. Venivano utilizzati per calcolare l'esito della partita, quando in realtà ciò dovrebbe avvenire lato backend e si è fatto in modo che ora sia quest'ultimo a fornire i risultati.
- È stata modificata in una GET l'API "/receiveClassUnderTest", che era implementata come una POST, dato che era una richiesta di recupero e non di caricamento di una risorsa.
- È stata aggiunta una API "/run" che si occupa della sottomissione del tentativo da parte dell'utente, quindi si occupa del salvataggio della partita, compilazione e calcolo copertura e calcolo vincitore.

Task	Endpoint	Metodo	Descrizione	Parametri	Risposta
6	/run	POST	Salva la partita e ritorna i risultati	<ul style="list-style-type: none"> • Id del Game, • Id del Turn • Id del Round • Id della classe sotto test • Robot scelto • Livello scelto • Codice classe sotto test • Codice classe utente • Nome classe sotto test 	<ul style="list-style-type: none"> • Coverage • Output di compilazione utente • punteggio del robot • punteggio robot • vincitore

Figure 3.4: API Run

- È stata aggiunto il recupero del codice della classe da testare dal Task 1.
- È stata aggiunto il recupero dell'output della compilazione e del calcolo della copertura dal Task 7.
- È stata aggiunto il salvataggio del tentativo di gioco e dei risultati della partita dello studente utilizzando il Game Repository (realizzato dal Task 4).

Task 7

- È stata risolto un problema che capitava durante la compilazione on-demand offerta da questo componente: si verificava un deadlock a causa di una mancata lettura dallo stream buffer.

Task 9

- È stato aggiunto il salvataggio dei file di copertura (in formato xml) per ogni livello generato da Randoop.
- Sono state effettuate delle modifiche che hanno permesso di eseguire il componente anche su Docker (in origine eseguibile solo su Windows), tra cui:
 - Sostituzione di percorsi che funzionavano solo su Windows e non su Linux (\ anzichè /)
 - Sostituzione del file Batch (Windows) in un file Bash (Linux), utilizzato per la generazione dei test e il calcolo della copertura
- È stata effettuata la sostituzione della versione di Java da 11 a 8 (in quanto Emma, il tool di copertura, richiede tassativamente Java 8): sono state anche rimpiazzate tutte quelle funzioni che venivano utilizzate all'interno del componente che sono disponibili da Java 11.

- È stata effettuata una modifica al file "pom.xml" perchè la compilazione del progetto generava un .jar privo della dipendenza JSoup e quindi inutilizzabile.

3.4 Gateway Sequence Diagrams

Ogni richiesta da parte del client viene filtrata dai componenti UI Gateway e API Gateway: si ometterà negli altri sequence diagram questa interazione ai fini di una maggiore leggibilità e sarà apposta una apposita nota per notificarlo.

3.4.1 UI Gateway

Ogni volta che un client richiede l'accesso a un pagina web, si interfaccia con l'UI Gateway che si occupa di effettuarne il routing. L'interazione è descritta nel seguente diagramma di sequenza.

Il client richiede una pagina web e l'UI gateway, dopo aver interrogato la propria tabella di routing, effettua la richiesta al microservizio richiesto per conto del client e gli inoltra la risposta.

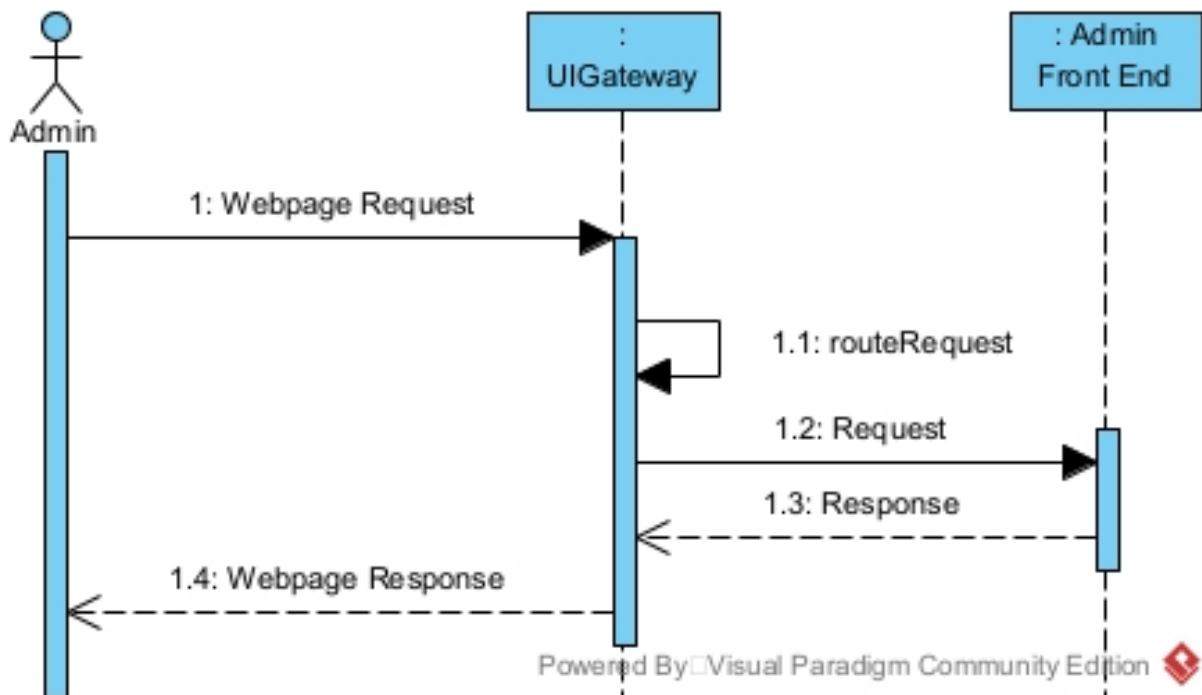


Figure 3.5: UI Gateway Sequence Diagram (Admin)

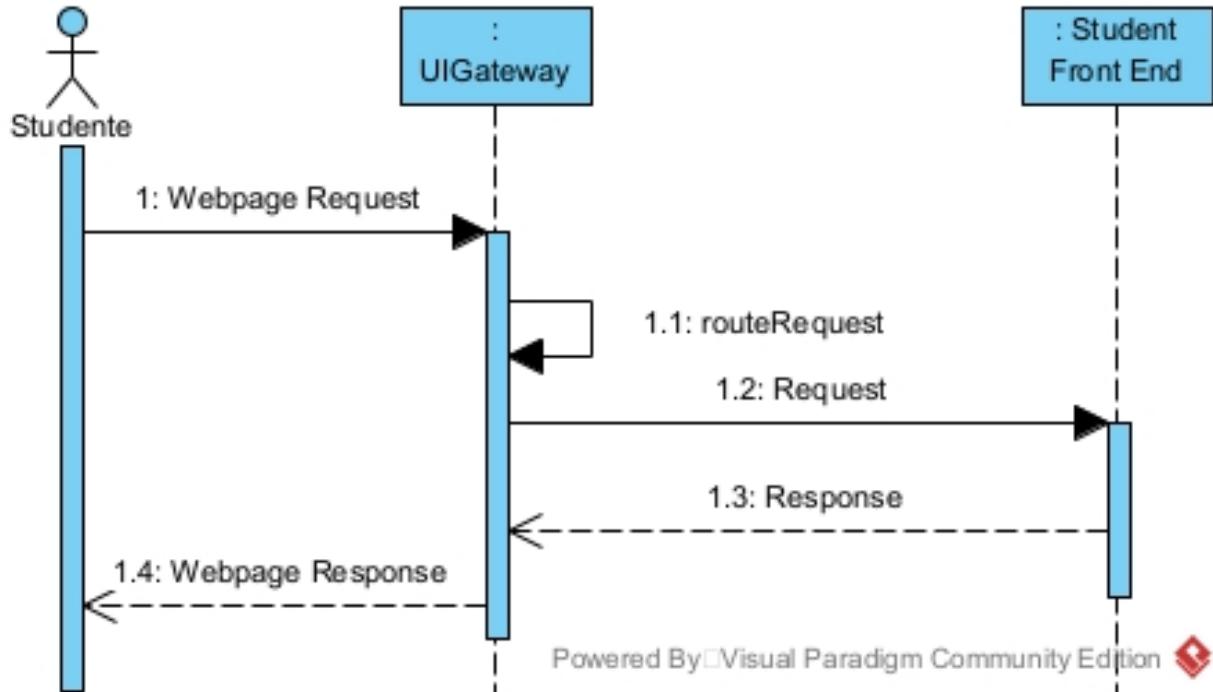


Figure 3.6: UI Gateway Sequence Diagram (Student)

3.4.2 API Gateway

Ogni volta che un client richiede il servizio di un API, si interfaccia con l'API Gateway che si occupa di effettuarne il routing, di verificarne l'autenticazione e di prepararne l'autorizzazione. L'interazione è descritta nel seguente diagramma di sequenza.

Quando il client richiede l'utilizzo di un API, l'API Gateway effettua le seguenti operazioni:

1. estrae il token JWT contenuto in un cookie della richiesta
2. verifica che il token JWT sia valido, utilizzando il servizio offerto dallo Student Repository (nel caso in cui non lo sia, restituisce un errore "Not Authorized")
3. estrae i claim contenuti all'interno del token JWT (ovvero l'ID dello studente) e li aggiunge come Header di autorizzazione della richiesta
4. interroga la propria tabella di routing
5. effettua la richiesta al microservizio richiesto per conto del client e gli inoltra la risposta

Come si può notare, in realtà questo componente non implementa completamente un meccanismo di autorizzazione, ma la avvia: tale responsabilità, infatti, è comunque lasciata al singolo microservizio, permettendo così di dare un maggiore controllo e versatilità ad ognuno di essi.

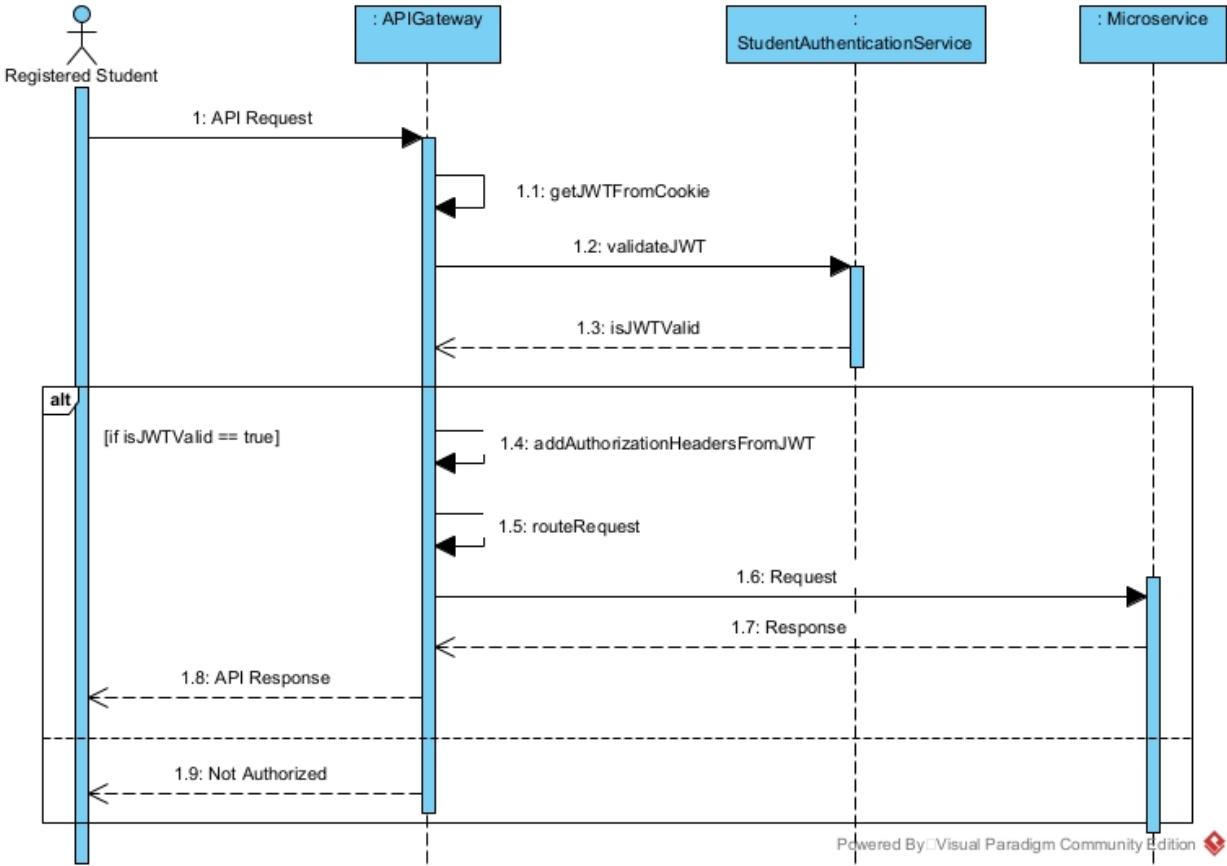


Figure 3.7: API Gateway Sequence Diagram

3.5 Nuovi Sequence Diagrams

Al fine di ottenere l'applicazione web desiderata funzionante, sono state apportate alcune modifiche alle funzioni implementate dai task e aggiunte delle altre.

3.5.1 Inspect ClassUnderTest

Questo caso d'uso si attiva dopo la scelta della classe, del robot e del livello da sfidare da parte dello studente, all'atto della pressione del tasto submit. Permette il caricamento in una apposita schermata del codice della classe che lo studente ha scelto di testare. Vengono effettuate le seguenti operazioni:

1. Viene effettuata una richiesta GET all'API "/receiveClassUnderTest" (realizzata dal Task 6) presente nel Game Engine Controller e indicando nei parametri il nome della classe. Si omette il passaggio per l'API Gateway descritto precedentemente.
2. Il Game Engine Controller, a sua volta, effettua una richiesta GET all'API "/downloadFile" (realizzata dal Task 1) presente nella Class Repository Controller e indicando il nome della classe all'interno dell'URL, per riceverne il codice.
3. Se tutto va a buon fine il codice viene ritornato come stringa e poi visualizzato nell'apposito riquadro, altrimenti si ritorna all'attore studente l'errore HTTP che ha impedito il successo dell'operazione.

I nomi usati per le entità sono stati modificati da quelli reali per rendere più comprensibile l'operazione.

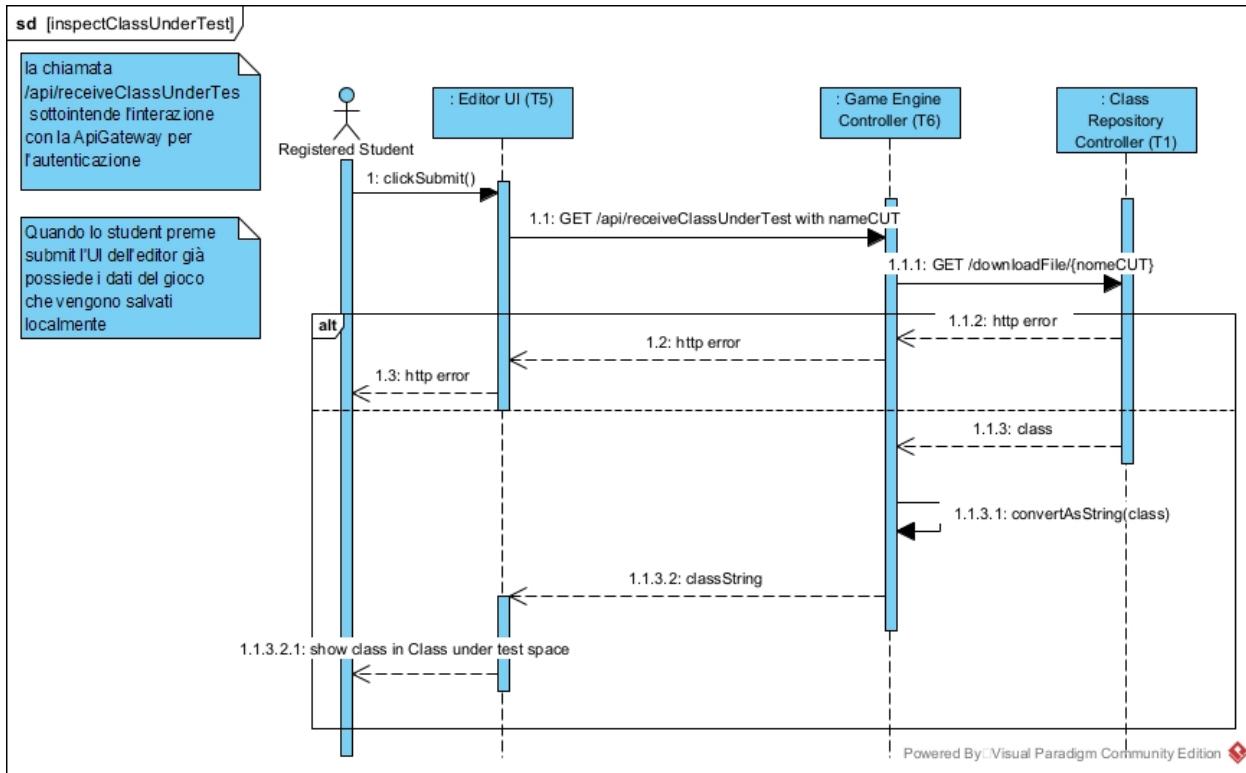


Figure 3.8: Inspect ClassUnderTest Sequence Diagram

3.5.2 Create Game

Questo caso d'uso permette la creazione della partita, a partire della selezione della classe da testare e dell'avversario da affrontare. Per UI dell'editor si considera tutto il front end sviluppato all'interno del task 5. Vengono effettuate le seguenti operazioni:

1. L'Editor UI mostra la schermata di selezione della classe all'utente studente, egli la sceglie e gli verrà poi mostrata, in base alla scelta, una lista di robot disponibili classificati per livello.
2. L'Editor UI chiede conferma della selezione complessiva. Lo studente clicca poi Submit e viene svolto il caso d'uso incluso InspectClassUnderTest che restituisce il codice della classe scelta da mostrare poi successivamente a video.
3. Viene poi effettuata una richiesta POST all'API "/save-data" (realizzata dal Task 5) e presente nel GUI Controller. Qui avviene anche una fase di controllo sull'autorizzazione all'uso dell'API, ovvero se l'utente autenticato che ha passato i controlli dell'API gateway, sta facendo una richiesta legittima di salvataggio della propria partita (funzione checkApiAuthorization). Se ciò fallisce si restituisce all'utente l'errore.
4. Viene dunque creato un oggetto Game con le informazioni sulla partita che viene poi utilizzato dalla funzione presente in Game Data Writer, che si occupa del salvataggio vero e proprio.
5. Game Data Writer legge tutti i campi necessari dall'oggetto Game ricevuto
6. Game Data Writer effettua poi una serie di richieste POST all'API "/games", "/rounds" e "/turns" verso il Game Repository (realizzato dal Task 4) per creare una partita, un round ed un turno.

7. Infine, le informazioni relative alla partita appena creata vengono restituite all'Editor UI per essere memorizzate e viene visualizzato l'Editor che permette allo studente di giocare la partita.

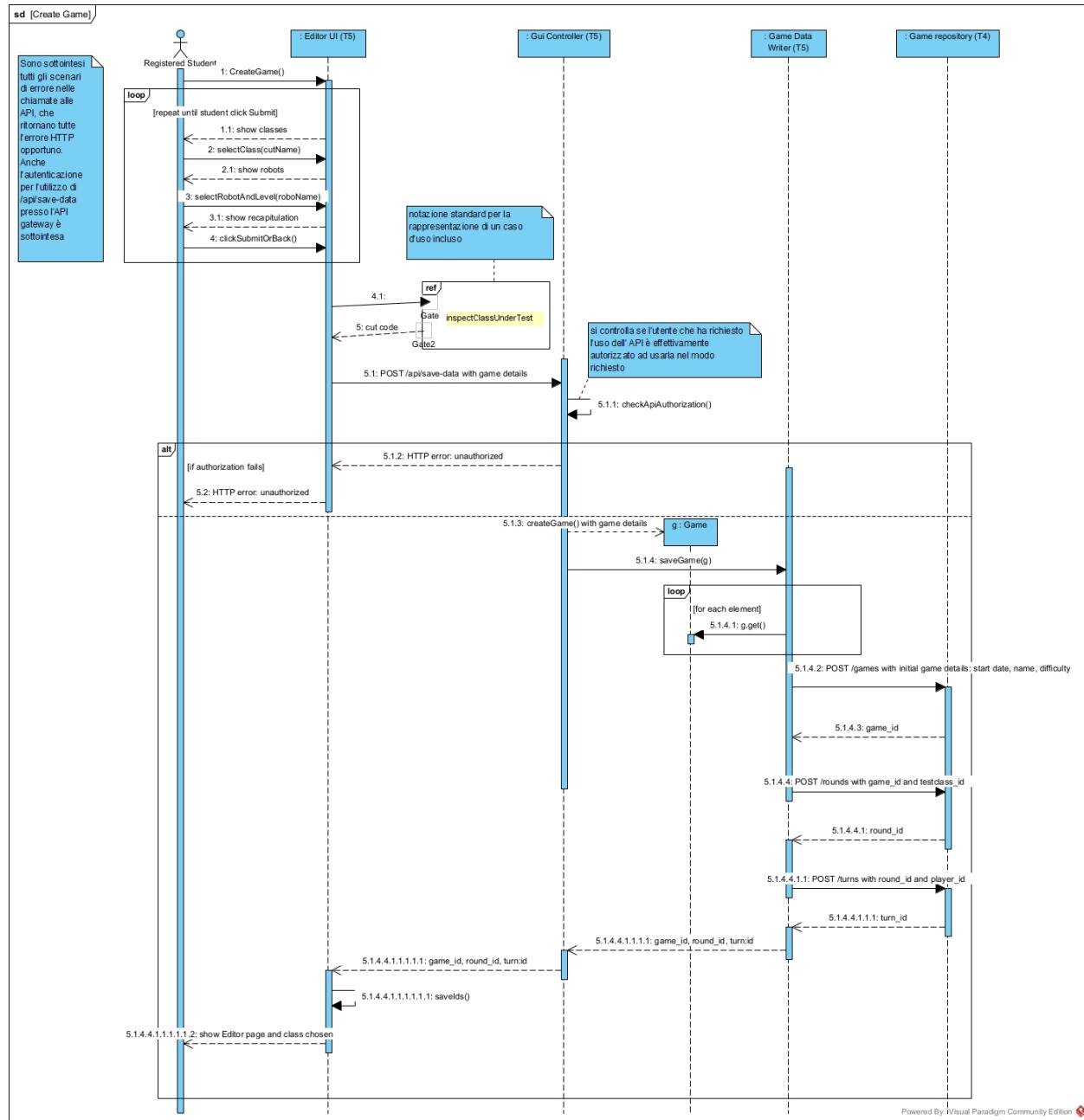


Figure 3.9: Create Game Sequence Diagram

3.5.3 Run

Questo caso d'uso permette la conclusione e il salvataggio della partita, nonché il calcolo del vincitore. Si attiva all'atto di pressione del tasto Run da parte dello studente mentre si trova nell'editor. Vengono effettuate le seguenti operazioni:

1. L'Editor UI effettua una richiesta POST all'API ”/run” (realizzata dal Task 6) presente nel Game Engine Controller, che si occupa di svolgere tutte le azioni necessarie, inserendo all'interno del body le informazioni sulla partita effettuata e il codice sottomesso.
2. Il Game Engine Controller effettua a sua volta una richiesta POST all'API ”/compile-and-codecoverage” (realizzata dal Task 7) che ha il compito di compilare il codice dell'utente e calcolarne la coverage, restituendo quest'ultima assieme all'output di compilazione.
3. Il Game Engine Controller effettua una richiesta GET all'API ”/robots” verso il Game Repository (realizzato dal Task 4), ottenendo così il punteggio del robot scelto dall'utente.
4. Il Game Engine Controller effettua poi una serie di richieste PUT alle API ”/turns”, ”/rounds” e ”/games” verso il Game Repository (realizzato dal Task 4) per aggiornare le informazioni sulla partita, sul turno e sul round (indicando nel caso del turno anche chi è il vincitore).
5. I risultati della partita saranno poi restituiti all'Editor UI.

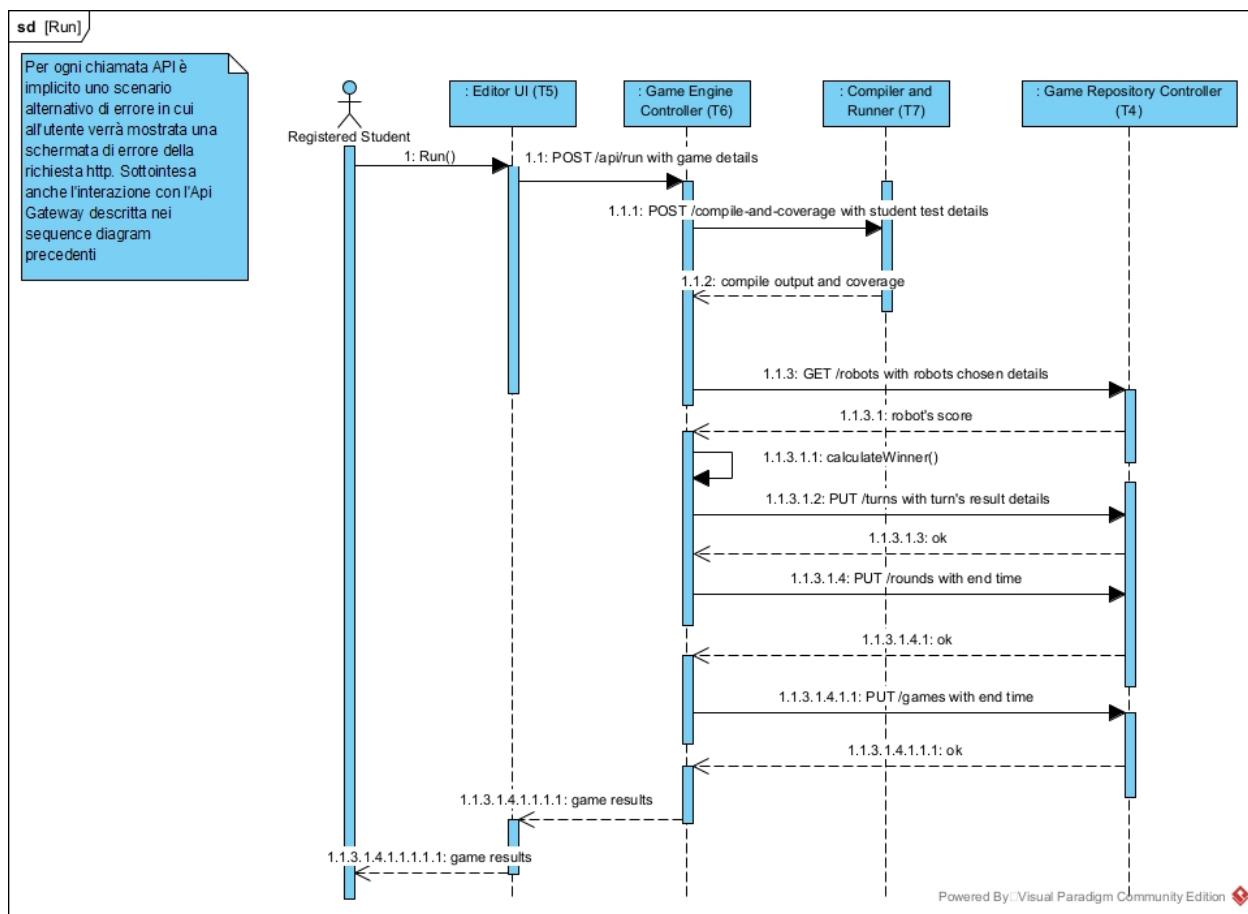


Figure 3.10: Run Sequence Diagram

3.5.4 Run with JaCoCo

Questo caso d'uso si occupa del calcolo della copertura del codice scritto dall'utente studente. Si attiva alla pressione del tasto JaCoCo presente nell'Editor UI. Vengono effettuate le seguenti operazioni:

1. L'Editor UI si rivolge tramite una richiesta POST all'API ”/getJaCoCoReport” (realizzata dal Task 6), presente nel Game Engine Controller, fornendo nel body il codice dell'utente.
2. A sua volta il Game Engine Controller si rivolge, tramite una richiesta POST all'API ”/compile-and-get-coverage” (realizzata dal Task 7) che ha il compito di compilare il codice dell'utente e calcolarne la coverage.
3. In caso di errore, questo viene restituito fino all'utente. Altrimenti, al Game Engine Controller vengono ritornate le coverage e l'output di compilazione: solo la copertura viene trasmessa all'Editor UI che evidenzierà le linee di codice della classe sotto test nel seguente modo: se la linea è coperta, parzialmente coperta e non coperta, si usano rispettivamente i colori verde, giallo e rosso.
In questo modo l'utente può ricevere un feedback sull'andamento della partita e può migliorarsi senza sottomettere il tentativo.

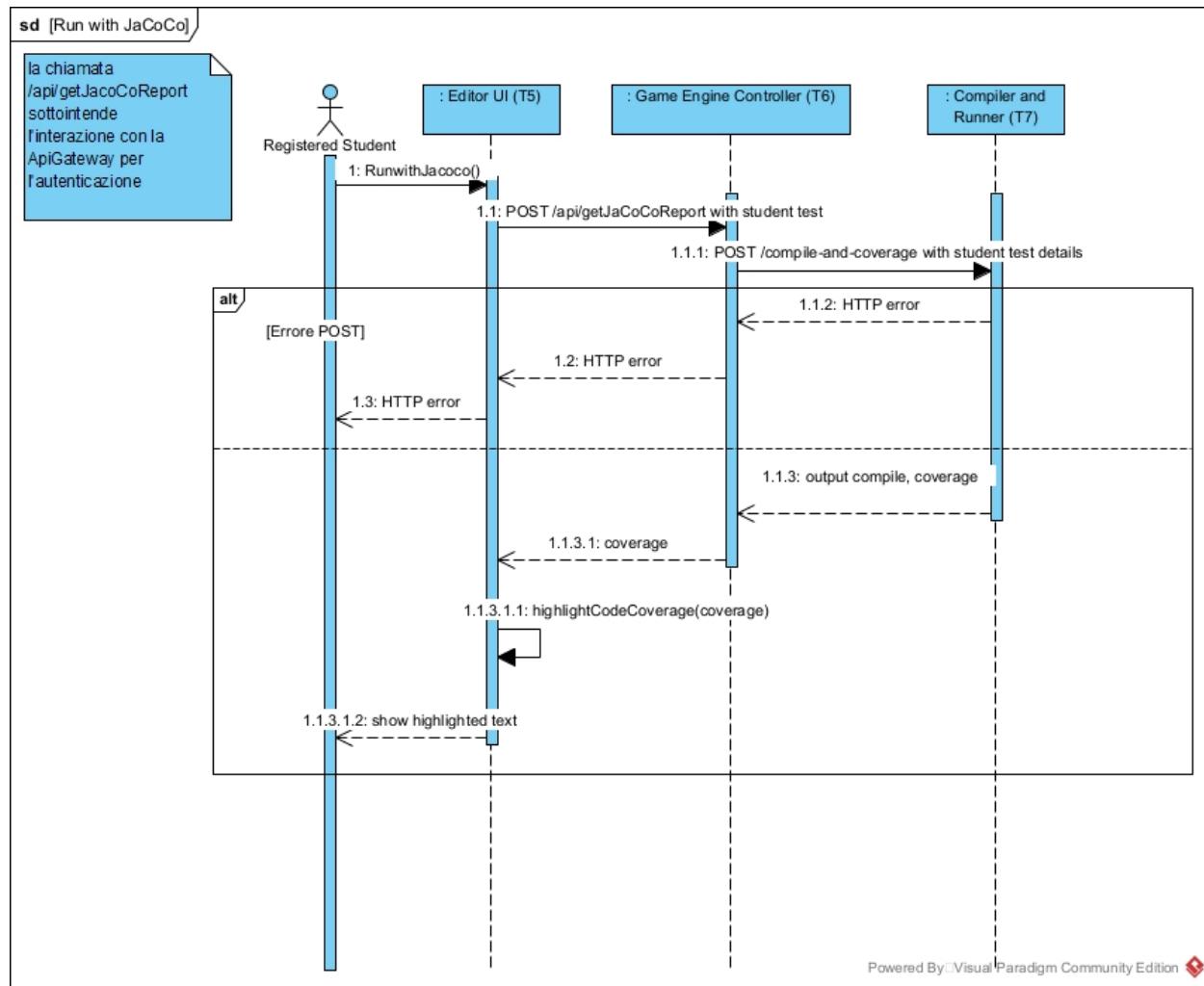


Figure 3.11: Run with JaCoCo Sequence Diagram

3.5.5 Compile

Questo caso d'uso permette di compilare il tentativo dello studente e di visualizzare il risultato della compilazione in un apposito riquadro. Si attiva alla pressione del tasto Compile presente nell'Editor UI. Vengono effettuate le seguenti operazioni:

1. L'Editor UI si rivolge tramite una richiesta POST all'API "/sendInfo" (realizzata dal Task 6), presente nel Game Engine Controller, fornendo nel body il codice dell'utente.
 2. A sua volta il Game Engine Controller si rivolge, tramite una richiesta POST all'API "/compile-and-get-coverage" (realizzata dal Task 7) che ha il compito di compilare il codice dell'utente e calcolarne la coverage.
 3. In caso di errore, questo viene restituito fino all'utente. Altrimenti, al Game Engine Controller vengono ritornate le coverage e l'output di compilazione: solo l'output di compilazione viene trasmesso all'Editor UI che lo mostrerà a video.
- In questo modo l'utente può ricevere un feedback sull'andamento della partita e vedere se effettivamente il codice prodotto compila e, se non lo fa, quali sono gli errori mostrati dal compilatore.

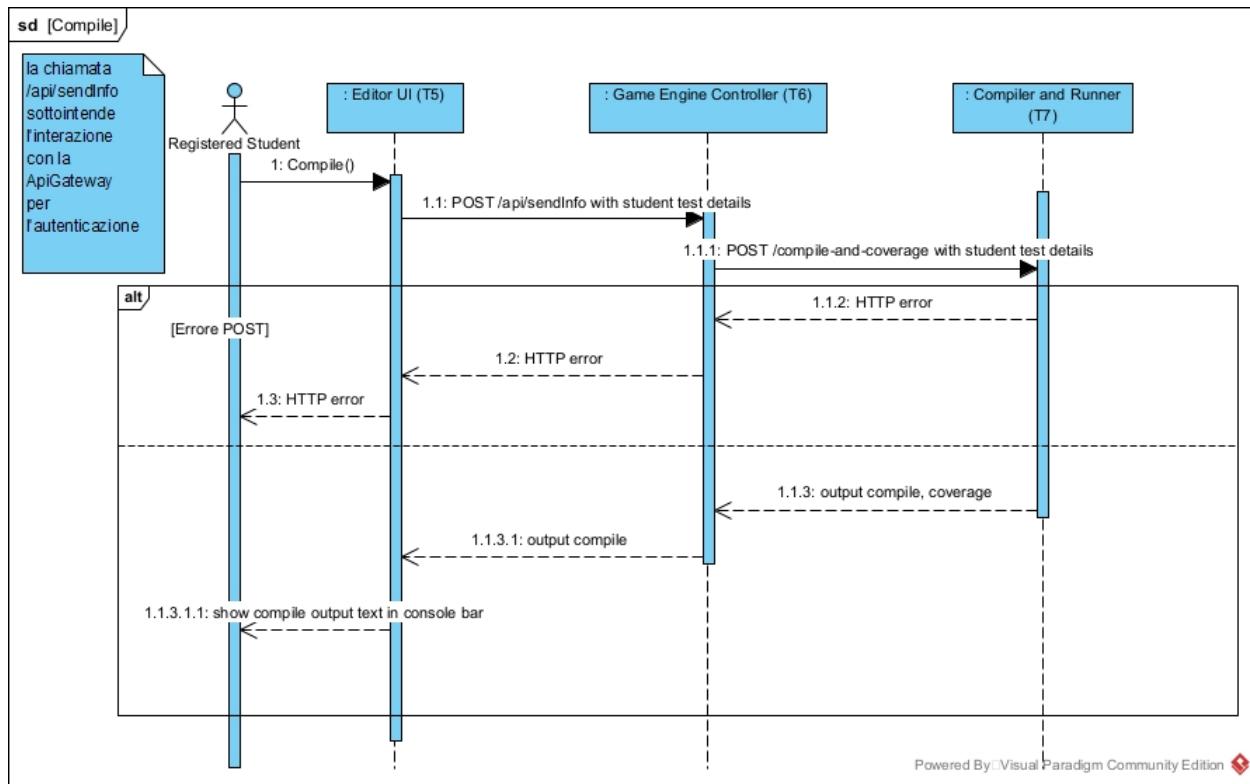


Figure 3.12: Compile Sequence Diagram

3.5.6 Add class

Questo caso d'uso permette all'amministratore di inserire una nuova classe per il gioco. Si attiva alla presione del testo Add Class presente sull'Admin UI. Vengono effettuate le seguenti operazioni:

Si è ripreso il sequence diagram sviluppato dal Task 1, aggiungendo un'unica chiamata per mostrare l'integrazione con il componente che si occupa della generazione dei test con Randoop.

La funzione nuova è la "generateAndSaveRobots()" della classe RobotUtil: ha come parametro il file della classe e permette al robot Randoop di generare i test. In questa funzione avviene anche il salvataggio del punteggio, per ogni livello generato, delle coperture del robot calcolato con il tool Emma. Per far ciò, viene effettuata una richiesta di tipo POST all'API "/robots" verso il Game Repository, che ne permette la memorizzazione. Si noti che, per come è stato implementato il caso d'uso dal task 1, il file sorgente della classe si troverà sul File System (nella cartella Files-Upload), mentre i metadati relativi (nome, percorso in cui è memorizzato, data, categoria etc.) saranno sul database.

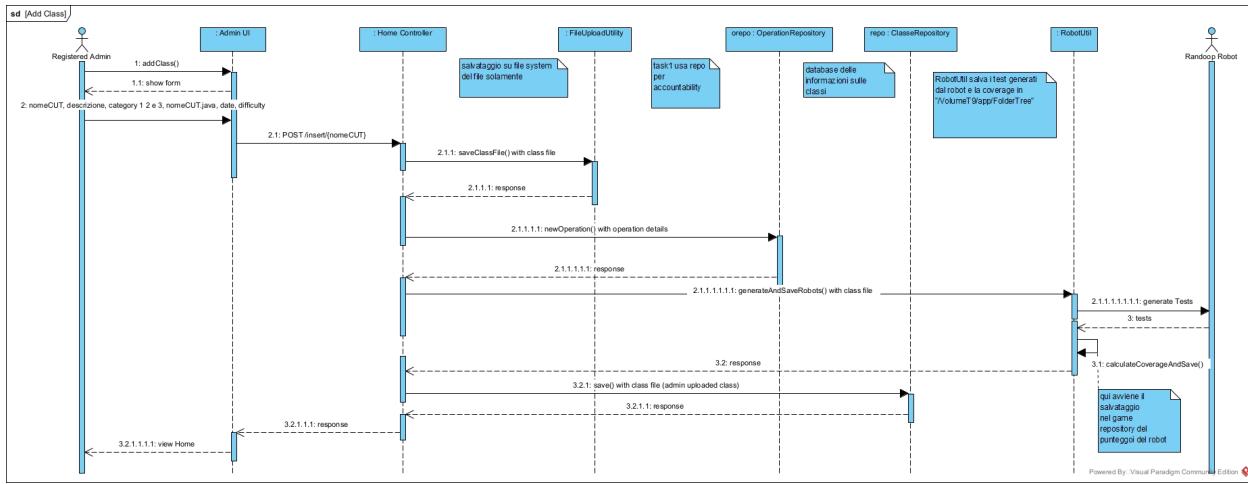


Figure 3.13: Add Class Sequence Diagram

3.6 Activity Diagrams

Si riporta di seguito l'unico activity diagram modificato rispetto alle precedenti documentazioni.

3.6.1 Add Class

Con questo diagramma si rende più evidente la scelta di integrazione fatta per la generazione dei test con il robot Randoop.

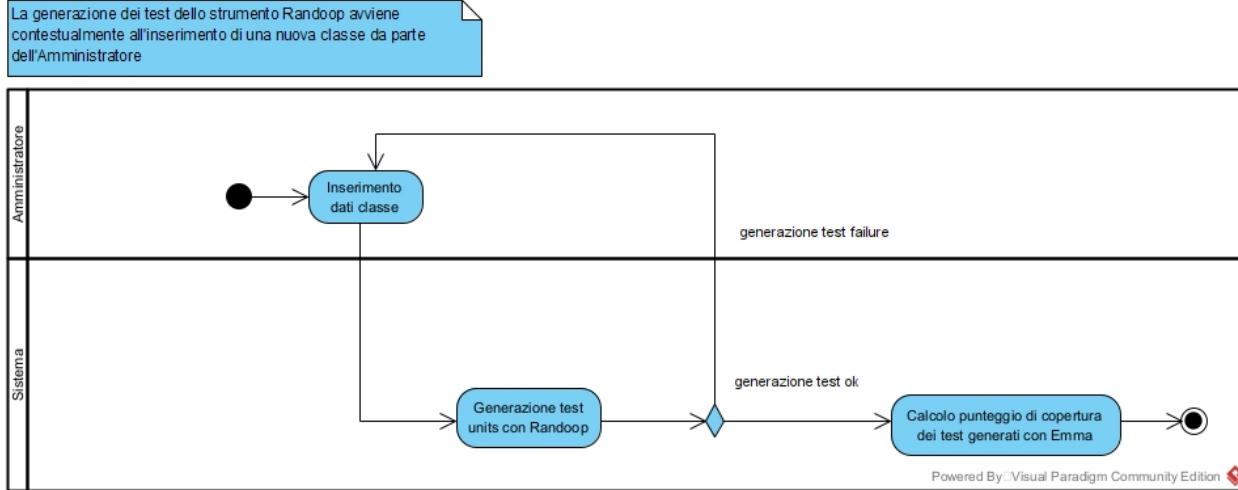


Figure 3.14: Add Class Activity Diagram

Capitolo 4

Deployment

Nel seguente diagramma si mette in evidenza la disposizione fisica dei componenti di un sistema software e la loro interazione all'interno di un ambiente di esecuzione.

Il server che ospita l'applicativo utilizza come sistema operativo Windows 11 e, più precisamente, il software è in esecuzione nell'ambiente Docker, distribuito sui vari container (quelli evidenziati in viola sono stati sviluppati e aggiunti da noi).

Si può notare la presenza di altri due elementi specifici: il volume "VolumeT9" e la rete "Global Network":

- il volume è condiviso dai container dei Task 1 e 9, ciò risulta necessario in quanto entrambi devono poter accedere ad una stessa porzione del File System: quella relativa alle classi e alla copertura generate da Randoop ed Emma.
- la rete è condivisa dai container di tutti i Task, ciò permette di isolare i container dalle reti dell'host e al tempo stesso permettere la comunicazione tra di essi (si ottiene così una sorta di rete virtuale): l'unico container che è esposto verso l'esterno è il Gateway che si occupa di gestire le richieste.

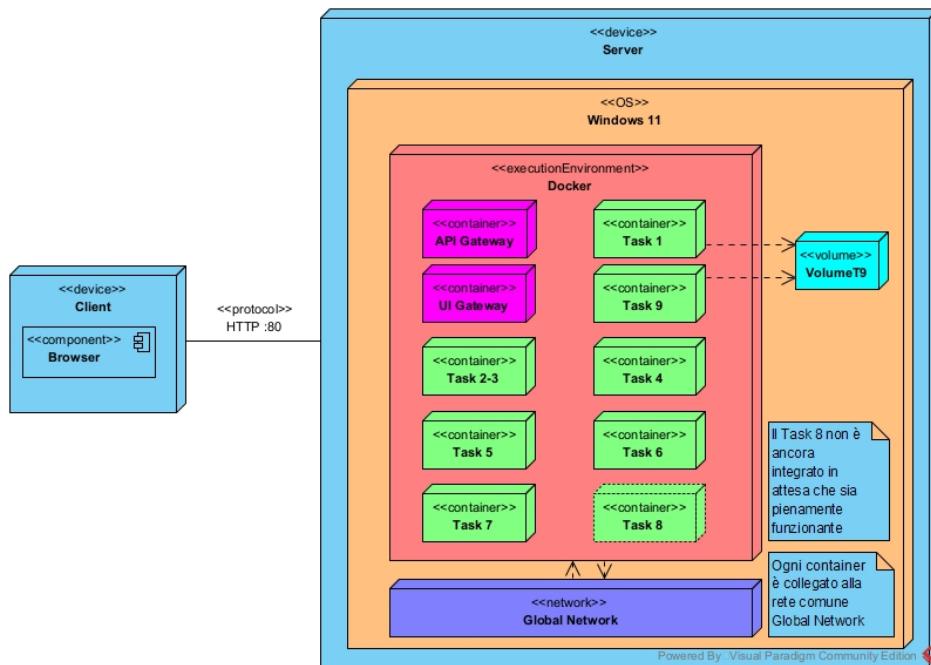


Figure 4.1: Diagramma di Deployment

Capitolo 5

Testing

Il nostro obiettivo è l'integrazione di tutti i componenti per ottenere un'applicazione funzionante. A tale scopo, è stato condotto il "Test End-to-End (E2E)", noto anche come "Test di Sistema".

Si tratta di un metodo di testing che valuta l'intero flusso dell'applicazione, garantendo che tutti i componenti, combinati tra loro, operino come previsto in scenari reali. Infatti, il software viene testato dal punto di vista dell'utente finale, simulando uno scenario realistico, inclusa l'interfaccia utente, i servizi di backend, i database e la comunicazione di rete. Tra gli scopi c'è quello di convalidare il comportamento complessivo dell'applicazione, includendo funzionalità, affidabilità, prestazioni e sicurezza.

L'obiettivo finale del test E2E è individuare difetti o problemi che potrebbero sorgere quando le diverse parti dell'applicazione interagiscono tra loro. Solitamente, il test E2E viene eseguito dopo il test di integrazione, che verifica i singoli moduli, e prima del test di accettazione dell'utente, che assicura che l'applicazione soddisfi i requisiti dell'utente.

5.1 Tool utilizzati

Per effettuare questo tipo di test è stato utilizzato Selenium, uno strumento per il testing automatizzato per applicazioni web, insieme a JUnit, il noto framework di testing, e Google Chrome, il browser utilizzato per collegarsi all'applicazione.

5.2 Casi di Test

Di seguito sono riportati i vari test effettuati. I flussi dell'applicazione testati sono quelli relativi all'accesso (Login Test) e all'utilizzo dell'editor (Editor Test).

In entrambi si possono individuare due fasi importanti: una relativa alla configurazione dell'ambiente e un'altra relativa ai test veri e propri.

5.2.1 Login Test

Setup

La fase di configurazione consiste nel definire quali operazioni devono essere effettuare in via preliminare, prima che i test comincino.

Entrando più nello specifico, tali operazioni riguardano:

- l'impostazione del driver di Selenium che gli permette di utilizzare il browser
- l'apertura del browser all'avvio di un test
- la chiusura del browser al termine di un test

```

○ ○ ○

...
private static ChromeDriver driver;
private static int timeout = 10;

@BeforeClass
public static void setDriver() {
    System.setProperty("webdriver.chrome.driver", "/path/to/driver");
}

@Before
public void openBrowser(){
    driver = new ChromeDriver();
    driver.manage().timeouts().implicitlyWait(timeout, TimeUnit.SECONDS);
}

@After
public void closeBrowser(){
    driver.close();
}
...

```

Figure 5.1: Login Test - Setup

Credenziali valide

Tale test verifica che l'accesso avvenga correttamente utilizzando credenziali valide (ovvero di un studente già registrato).

Per fare ciò, il robot effettua le seguenti operazioni:

1. si collega alla pagina di accesso (/login)
2. inserisce le credenziali (email e password)
3. clicca il pulsante di accesso
4. verifica che sia stato effettuato il redirect alla prima pagina dell'editor (/main)

```

○ ○ ○

...
    @Test
    public void validCredentials(){
        driver.get("http://localhost/login");
        driver.findElement(By.id("email")).sendKeys("mario.rossi@mail.com");
        driver.findElement(By.id("password")).sendKeys("MarioRossi0");
        driver.findElement(By.cssSelector("input[type=submit]")).click();

        WebDriverWait wait = new WebDriverWait(driver, timeout);

        String urlPaginaDiRedirezione = "http://localhost/main";
        try {
            wait.until(ExpectedConditions.urlToBe(urlPaginaDiRedirezione));
        } catch(TimeoutException e) {
            Assert.fail();
        }

        Assert.assertEquals("Test fallito! Il login non è avvenuto correttamente.",
        driver.getCurrentUrl(), urlPaginaDiRedirezione);
    }
...

```

Figure 5.2: Login Test - Credenziali valide

Credenziali non valide

Tale test verifica che l'accesso venga negato utilizzando credenziali non valide.
Per fare ciò, il robot effettua le seguenti operazioni:

1. si collega alla pagina di accesso (/login)
2. inserisce le credenziali (email e password)
3. clicca il pulsante di accesso
4. verifica che sia stato effettuato il redirect alla prima pagina dell'editor (/main)

```

○ ○ ○

...
    @Test
    public void invalidCredentials(){
        driver.get("http://localhost/login");
        driver.findElement(By.id("email")).sendKeys("pippobaudo@gmail.com");
        driver.findElement(By.id("password")).sendKeys("password");
        driver.findElement(By.cssSelector("input[type=submit]")).click();

        WebDriverWait wait = new WebDriverWait(driver, timeout);

        try {
            wait.until(ExpectedConditions.textToBe(By.tagName("body"), "Incorrect
password"));
        } catch(TimeoutException e) {
            Assert.fail();
        }

        Assert.assertEquals("Test fallito! Il login è avvenuto correttamente.",
driver.findElement(By.tagName("body")).getText(), "Incorrect password");
    }
...

```

Figure 5.3: Login Test - Credenziali non valide

5.2.2 Editor Test

Setup

La fase di configurazione consiste nel definire quali operazioni devono essere effettuare in via preliminare, prima che i test comincino.

Entrando più nello specifico, tali operazioni riguardano:

- l'impostazione del driver di Selenium che gli permette di utilizzare il browser
- l'impostazione del percorso dei download
- l'apertura del browser e l'autenticazione dell'utente all'avvio di un test
- la chiusura del browser al termine di un test

```

○ ○ ○

...
private static ChromeDriver driver;
private static int timeout = 60;

@BeforeClass
public static void setDriver() {
    System.setProperty("webdriver.chrome.driver", "/path/to/driver");
}

@Before
public void openBrowser(){
    ChromeOptions options = new ChromeOptions();
    options.setCapability(CapabilityType.UNEXPECTED_ALERT_BEHAVIOUR,
UnexpectedAlertBehaviour.ACCEPT);
    HashMap<String, Object> chromePrefs = new HashMap<String, Object>();
    chromePrefs.put("profile.default_content_settings.popups", 0);
    chromePrefs.put("download.default_directory", "/path/to/download");
    options.setExperimentalOption("prefs", chromePrefs);

    driver = new ChromeDriver(options);
    driver.manage().timeouts().implicitlyWait(timeout, TimeUnit.SECONDS);

    driver.get("http://localhost/login");
    driver.findElement(By.id("email")).sendKeys("mariorossi@mail.com");
    driver.findElement(By.id("password")).sendKeys("Mariorossi0");
    driver.findElement(By.cssSelector("input[type=submit]")).click();

    WebDriverWait wait = new WebDriverWait(driver, timeout);

    String urlPaginaDiRedirezione = "http://localhost/main";
    try {
        wait.until(ExpectedConditions.urlToBe(urlPaginaDiRedirezione));
    } catch(TimeoutException e) {
        Assert.fail();
    }
}

@After
public void closeBrowser(){
    driver.close();
}
...

```

Figure 5.4: Editor Test - Setup

Download classe

Tale test verifica che il download di una classe avvenga correttamente.
Per fare ciò, il robot effettua le seguenti operazioni:

1. clicca la prima classe disponibile

2. clicca il pulsante di download
3. verifica che il file scaricato sia presente all'interno dei download

```
○ ○ ○

...
@Test
public void download() throws InterruptedException {
    driver.findElement(By.id("0")).click();

    driver.findElement(By.id("downloadButton")).click();

    File f = new File("/path/to/download/class.java");

    Thread.sleep(5000);

    Assert.assertTrue(f.exists());
}
...
```

Figure 5.5: Editor Test - Download classe

Scelta classi e robot

Tale test verifica che la scelta di una classe e di un robot avvenga correttamente.
Per fare ciò, il robot effettua le seguenti operazioni:

1. seleziona la classe e il robot
2. clicca il pulsante di conferma
3. verifica che sia stato effettuato il redirect alla pagina di conferma (/report)

```
○ ○ ○

...
@Test
public void selection() {
    String urlPaginaDiRedirezione = "http://localhost/report";

    moveToReport(urlPaginaDiRedirezione);

    Assert.assertEquals("Test fallito! La selezione non è avvenuta correttamente.",
        driver.getCurrentUrl(), urlPaginaDiRedirezione);
}
...
```

Figure 5.6: Editor Test - Riepilogo scelta

Avvio partita

Tale test verifica che l'avvio di una partita avvenga correttamente.
Per fare ciò, il robot effettua le seguenti operazioni:

1. clicca il pulsante di conferma delle scelte effettuate
2. verifica che sia stato effettuato il redirect all'editor (/editor)

```
○ ○ ○  
...  
@Test  
public void startGame() {  
    String urlPaginaDiRedirezione = "http://localhost/editor";  
    moveToEditor(urlPaginaDiRedirezione);  
  
    Assert.assertEquals("Test fallito! L'avvio della partita non è avvenuto  
correttamente.", driver.getCurrentUrl(), urlPaginaDiRedirezione);  
}  
...
```

Figure 5.7: Editor Test - Avvio partita

Compilazione classe utente

Tale test verifica che la compilazione della classe di test scritta dall'utente avvenga correttamente.
Per fare ciò, il robot effettua le seguenti operazioni:

1. attende che la classe da testare sia caricata
2. clicca il pulsante di compilazione
3. verifica che sia visibile il risultato della compilazione

```

○ ○ ○

...
    @Test
    public void compile() {
        String urlPaginaDiRedirezione = "http://localhost/editor";
        moveToEditor(urlPaginaDiRedirezione);

        WebDriverWait wait = new WebDriverWait(driver, timeout);

        try {
            wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#sidebar-
textarea + div > * div.CodeMirror-code > *"), 1));
        } catch(TimeoutException e) {
            Assert.fail();
        }

        driver.findElement(By.id("compileButton")).click();

        try {
            wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#console-
textarea + div > * div.CodeMirror-code > *"), 1));
        } catch(TimeoutException e) {
            Assert.fail();
        }
    }
...

```

Figure 5.8: Editor Test - Compilazione classe utente

Copertura classe utente

Tale test verifica che la copertura della classe di test scritta dall'utente avvenga correttamente. Per fare ciò, il robot effettua le seguenti operazioni:

1. attende che la classe da testare sia caricata
2. clicca il pulsante di copertura
3. verifica che sia visibile il risultato della copertura

```

○ ○ ○

...
@Test
public void coverage() {
    String urlPaginaDiRedirezione = "http://localhost/editor";
    moveToEditor(urlPaginaDiRedirezione);

    WebDriverWait wait = new WebDriverWait(driver, timeout);

    try {
        wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#sidebar-
textarea + div > * div.CodeMirror-code > *"), 1));
    } catch(TimeoutException e) {
        Assert.fail();
    }

    driver.findElement(By.id("coverageButton")).click();

    try {
        wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#sidebar-
textarea + div > * div.CodeMirror-code > * .uncovered-line"), 0));
    } catch(TimeoutException e) {
        Assert.fail();
    }
}
...

```

Figure 5.9: Editor Test - Copertura classe utente

Submit della partita

Tale test verifica che il tentativo dell'utente venga consegnato e che la partita venga processata correttamente. Per fare ciò, il robot effettua le seguenti operazioni:

1. attende che la classe da testare sia caricata
2. clicca il pulsante di submit
3. verifica che sia visibile l'esito della partita

```

○ ○ ○

...
    @Test
    public void run() {
        String urlPaginaDiRedirezione = "http://localhost/editor";
        moveToEditor(urlPaginaDiRedirezione);

        WebDriverWait wait = new WebDriverWait(driver, timeout);

        try {
            wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#sidebar-
textarea + div > * div.CodeMirror-code > *"), 1));
        } catch(TimeoutException e) {
            Assert.fail();
        }

        driver.findElement(By.id("runButton")).click();

        try {
            wait.until(ExpectedConditions.numberOfElementsToBeMoreThan(By.cssSelector("#console-
textarea2 + div > * div.CodeMirror-code > *"), 1));
        } catch(TimeoutException e) {
            Assert.fail();
        }
    }
...

```

Figure 5.10: Editor Test - Submit della partita

5.3 Risultati

Di seguito si riportano i risultati dei test con l'output di Maven.

```

Results :

Tests run: 8, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  38.919 s
[INFO] Finished at: 2023-09-10T15:29:38+02:00
[INFO] -----

```

Figure 5.11: Risultati "mvn test"

Capitolo 6

Installazione

In questo capitolo è riportata la procedura di installazione dell'intera applicazione. Essa avviene in maniera completamente automatica tramite l'avvio del file batch "installer.bat" che funge da installer.

Tale procedura è stata semplificata e resa possibile grazie all'utilizzo di Docker: la logica su cui si basa prevede di accedere alle cartelle dei singoli Task in cui sono presenti i file di configurazione "docker-compose.yml" ed effettuare così l'installazione di un Task alla volta.

Inizialmente si è cercato di realizzare un unico file di configurazione "docker-compose.yml" che racchiudesse tutti i container attraverso il tool a linea di comando offerto dallo stesso Docker ("merge"), tuttavia si verificavano dei problemi causati dall'utilizzo di percorsi relativi all'interno dei singoli file di configurazione.

Nota: si sono riscontrati dei problemi con Docker Desktop quando non è spuntata l'opzione "Use WSL based engine" all'interno delle impostazioni. Dunque, si prega di modificare questa impostazione per un corretto funzionamento.

6.1 Passo 1

Si deve avviare lo script "installer.bat". Saranno effettuate le seguenti operazioni:

1. creazione della rete "global-network" comune a tutti i container
2. creazione del volume "VolumeT9" comune ai Task 1 e 9
3. installazione di ogni singolo container

6.2 Passo 2

Si deve configurare il container "manvsclass-mongo_db-1" così come descritto anche nella documentazione del Task 1. Per fare ciò bisogna fare le seguenti operazioni:

1. posizionarsi all'interno del terminale del container
2. digitare il comando "mongosh"
3. digitare i seguenti comandi:

```
1 use manvsclass
2 db.createCollection("ClassUT");
3 db.createCollection("interaction");
4 db.createCollection("Admin");
5 db.createCollection("Operation");
6 db.ClassUT.createIndex({ difficulty: 1 })
7 db.Interaction.createIndex({ name: "text", type: 1 })
8 db.interaction.createIndex({ name: "text" })
9 db.Admin.createIndex({username: 1})
```

6.3 Passo 3

L'intera applicazione è adesso pienamente configurata e raggiungibile sulla porta :80, raggiungibile cioè all'indirizzo "http://localhost".

Capitolo 7

Guida alle future integrazioni

Di seguito sono indicate delle procedure per effettuare l'integrazione di ulteriori futuri Task, mantenendo l'architettura generale da noi proposta.

7.1 Integrazione Container

Per procedere con l'integrazione del container all'interno della stessa rete condivisa, si deve utilizzare il seguente file di configurazione "docker-compose.yml":

```
1 version: '2'
2 services:
3     # sostituire "app" con il nome del container
4     app:
5         build: ./
6         expose:
7             # sostituire "8000" con la porta utilizzata
8             - 8000
9         networks:
10            - global-network
11
12 networks:
13     global-network:
14         external: true
```

7.2 Integrazione con UI Gateway

Per procedere con l'integrazione di nuovo Front End, si deve modificare il file di configurazione "/ui-gateway/default.conf" utilizzato da Nginx andando ad aggiungere un nuovo blocco "location":

```
1 ...
2 # sostituire "webpage" con l'endpoint utilizzato che serve il frontend
3 location ~ ^/(webpage) {
4     include /etc/nginx/includes/proxy.conf;
5     # sostituire "app" con il nome del container e "8000" con la porta utilizzata
6     proxy_pass http://app:8000;
7 }
8 ...
```

7.3 Integrazione con API Gateway

Per procedere con l'integrazione di nuovo endpoint REST API, si deve modificare il file di configurazione ”/api-gateway/src/resources/application.yml” utilizzato da Netflix Zuul andando ad aggiungere un nuovo blocco ”route”:

```

1 ...
2 routes:
3   // sostituire "app" con il nome del servizio che si vuole dare a questo endpoint
4   app-service:
5     sensitiveHeaders:
6       // sostituire "endpoint" con il nome dell'endpoint
7     path: /api/endpoint/**
8     url: http://app:8000/endpoint
9 ...

```

7.4 Integrazione Installer

Per procedere con l'integrazione all'interno dell'installer, si deve modificare lo script batch di installazione ”installer.bat” andando a modificare la lista delle cartelle in cui sono presenti i file di configurazione ”docker-compose.yml”:

```

1 ...
2 set list=".\\T1-G11\\applicazione\\manvsclass" ".\\T23-G1" ".\\T4-G18" ".\\T5-G2\\t5" ".\\T6-
3   G12\\T6" ".\\T7-G31\\RemoteCCC" ".\\T9-G19\\Progetto-SAD-G19-master" ".\\api_gateway" ".\\
ui_gateway"
4 ...

```