Task 4 – Seconda Iterazione

Gruppo G4: Arena Letizia M63001513

Ferrara Leonardo M63001517

1. Modifiche apportate a seguito della prima review

Prima di esporre quello che verrà realizzato durante la seconda iterazione, abbiamo ritenuto opportuno sfruttare i feedback raccolti durante la prima review per apportare dei cambiamenti a quanto prodotto nella prima iterazione.

Storie utente (Rivisitate)

Durante la prima review, abbiamo compreso che le storie utente da noi scritte erano inadeguate, poiché si riferivano agli sviluppatori dei task e non agli utenti effettivi del sistema. Abbiamo pertanto deciso di apportare delle modifiche, concentrandoci sulle azioni che il sistema deve compiere.

Le riproponiamo di seguito:

- Il sistema deve poter salvare la partita creata e i relativi dati in modo da permettere ad altri task di tenerne traccia.
- Il sistema deve poter allocare i test case scritti in un database in modo da permettere agli altri task di procedere con la loro compilazione ed esecuzione.
- Il sistema deve poter allocare in un database il risultato della compilazione, ed eventualmente l'esito (in termini di fault coverage), dei test case elaborati in modo da permetterne il recupero successivamente.
- Il sistema deve poter allocare i test case scritti dal robot in un database in modo da permettere agli altri task di procedere con la loro compilazione ed esecuzione.
- Il sistema deve poter allocare i test case scritti dal robot in un database in modo da permettere agli altri task di procedere con la loro compilazione ed esecuzione.

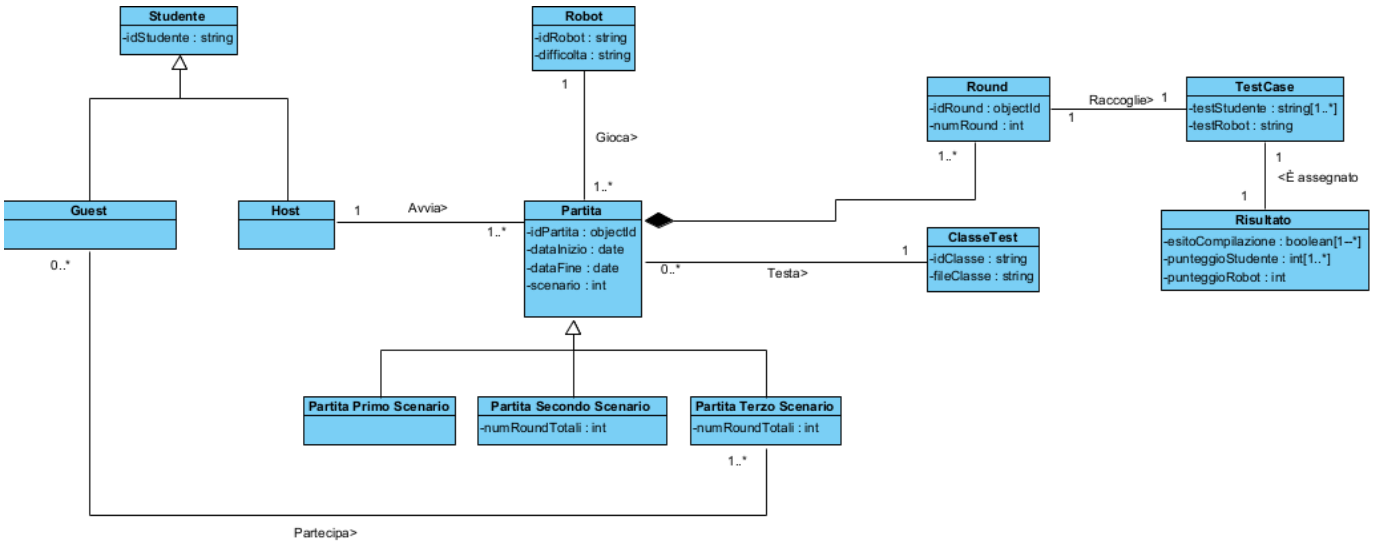
Database

Durante la prima review, ci è stata notificata l'esigenza di far funzionare il sistema su una singola macchina, e, pertanto, abbiamo deciso di avanzare la proposta di un database centralizzato in luogo di un database serverless.

In particolare, la versione scelta è MongoDB Community, alla quale si associa MongoDBCompass, un'interfaccia grafica che semplifica per lo sviluppatore la comunicazione con il database.

2. Class diagram raffinato

Di seguito è riportato il diagramma delle classi raffinato.
A seguito della prima review abbiamo approfondito la struttura del class diagram apportando delle modifiche: in particolare abbiamo specializzato la classe Studente in Host (studente che avvia la partita) e Guest (studente che partecipa alla partita in qualità di ospite) e abbiamo inoltre specializzato la classe Partita, individuando le caratteristiche dei tre possibili scenari di gioco.



3. Definizione di interfacce

Il passo successivo è definire le interfacce che consentiranno a componenti sviluppati indipendentemente di usufruire dei servizi messi a disposizione dal nostro database.

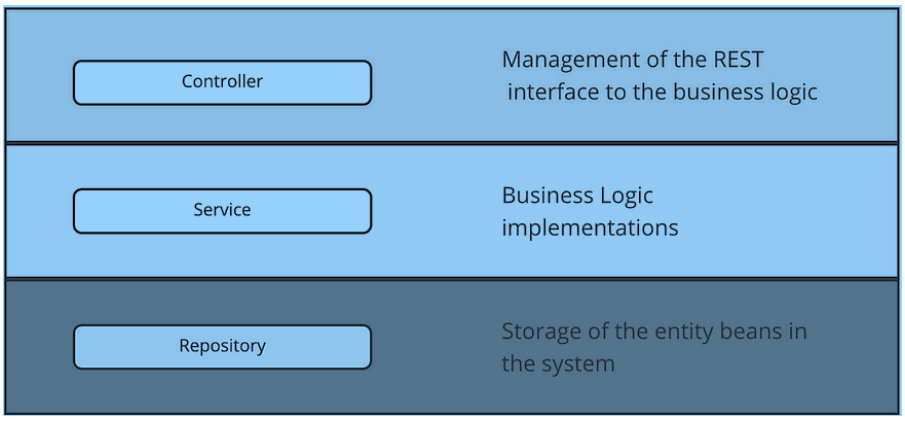
Per comprendere al meglio quali servizi debbano essere offerti si è scelto di utilizzare un'impostazione tabellare:

Operazione	Descrizione Operazione	Input	Output
createGame	Il game engine richiede il salvataggio dei dati della partita avviata dal giocatore	idGiocatore, dataInizio, scenario, idRobot, difficoltaRobot, idClasse, fileClasse	idPartita
createRound	Il game engine richiede il salvataggio dei dati del round all'interno della partita avviata dal giocatore	idPartita, numeroRound	idRound
updateTestRound	Il game engine permette di salvare i file di testo creati (sia dai giocatori che dal robot) durante la scrittura dei test case	idRound, testStudenti, testRobot	
updateRisultatoRound	Il game engine permette di salvare i risultati relativi ai test case (sia dei giocatori che del robot) relativi al round in corso	idRound, esitoCompilazione, punteggioStudente, punteggioRobot	
updateGame	Il game engine deve salvare i dati di conclusione della partita	idPartita, dataFine, risultato	

readStoricoGiocatore	Su richiesta il database deve fornire i dati relativi a tutte le partite giocate dallo specifico giocatore	idGiocatore	Per ogni partita: dataInizio, scenario, idRobot, difficoltaRobot, idClasse, fileClasse, dataFine, risultato
readGame	Su richiesta il database deve fornire lo storico della partita selezionata e dei suoi round	idPartita	Per la partita: dataInizio, scenario, idRobot, difficoltaRobot, idClasse, fileClasse, dataFine, risultato Per ogni round: numeroRound, idRound, testStudenti, testRobot, esitoCompilazione, punteggioStudente, punteggioRobot
readRound	Su richiesta il database deve fornire le informazioni relative al round selezionato nella partita indicata	idPartita, numeroRound	idRound, testStudenti, testRobot, esitoCompilazione, punteggioStudente, punteggioRobot

4. Pattern architetturale:

Nel contesto delle API REST, utilizzate in questo progetto, il pattern architetturale che ci è sembrato più appropriato è il pattern "**Controller-Service-Repository**".



L'utilizzo di questo pattern è prevalente in molte applicazioni Spring Boot, tecnologia scelta per il nostro servizio, come andremo ad approfondire in seguito.
Il vantaggio di questo pattern è la separazione degli interessi: il layer Controller ha il solo compito di esporre le funzionalità che possono essere usate da entità esterne; il layer Service si occupa della business logic e dunque dell'implementazione dei servizi richiesti; ed infine il layer Repository è responsabile dell'archiviazione e del recupero dei dati nel database.

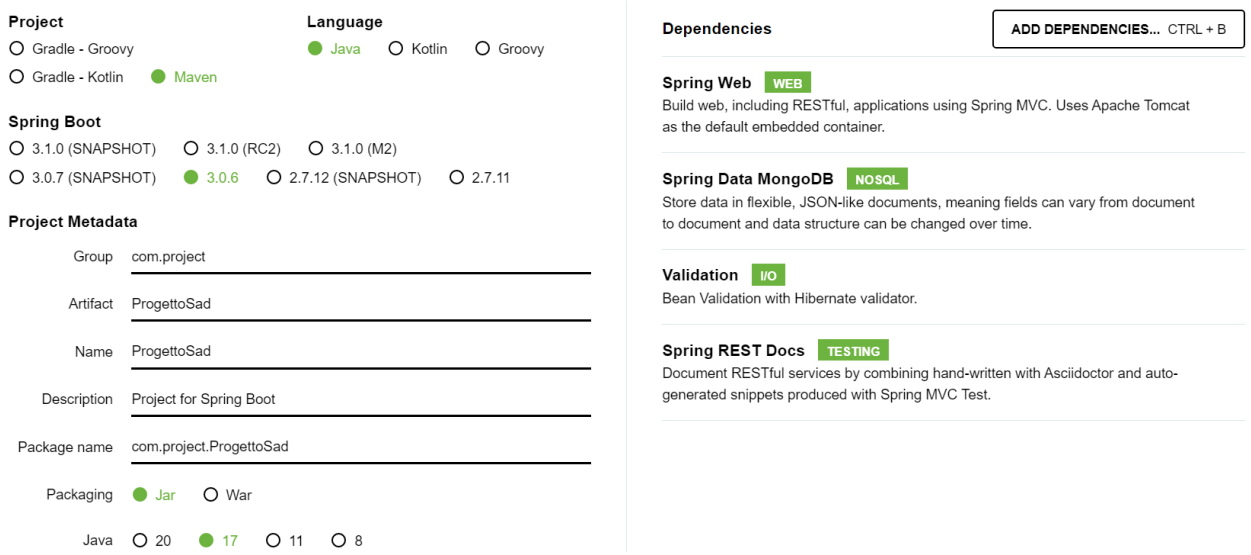
5. Studio delle tecnologie necessarie:

Creazione e condivisione del progetto:

Per permettere ai membri del gruppo di lavorare in remoto sul progetto è stato creato un repository GitHub dedicata, su cui è stato allocato il progetto. Abbiamo usufruito di GitHub Desktop per semplificare le operazioni di push e fetch.

Connessione al database e sviluppo delle API REST:

Per la connessione di Java con il database MongoDB abbiamo utilizzato il framework Spring Boot.
Grazie all'utilizzo di Spring Initializr, è stato possibile creare un progetto Maven che contenesse già alcune delle dipendenze necessarie al funzionamento dell'applicazione. Successive ricerche ci hanno poi permesso di comprendere quali altre dipendenze fossero necessarie nel file "pom.xml" generato.
In tal modo abbiamo creato la base per il nostro prototipo.



Testing delle API REST del prototipo:

Per verificare il funzionamento delle interfacce API REST effettuate abbiamo utilizzato il client HTTP Postman.
Grazie a questo software abbiamo avuto la possibilità di effettuare richieste API ed ispezionare i dati di richiesta e risposta.

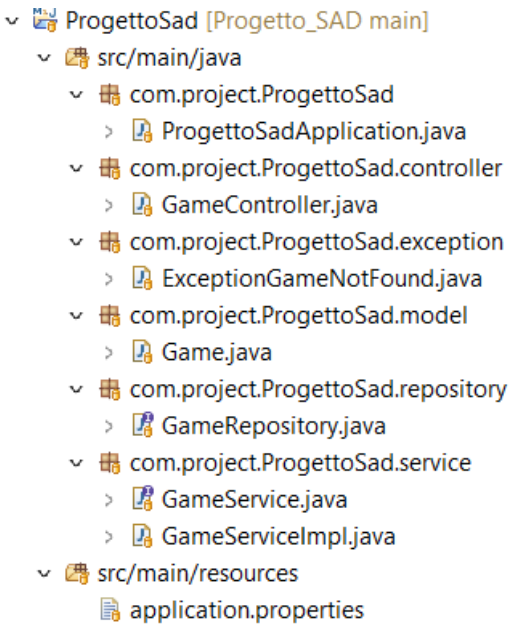
6. Prototipo:

Implementazione:

Per comprendere al meglio il funzionamento del servizio che andremo a creare, e per prendere familiarità con i tool necessari alla sua implementazione, abbiamo deciso di crearne un prototipo.

Il prototipo creato è una versione estremamente semplificata del progetto finale, le cui funzionalità non coincidono con quelle delineate in fase di analisi: il suo scopo è infatti approfondire le modalità con le quali è possibile instaurare una connessione con il database, e quali API sono necessarie a tale scopo.
Da ciò deriva la scelta di implementare una singola classe tra quelle indicate nel class diagram, la classe Game, relativa alla partita giocata, corredata da un insieme limitato di funzionalità.

La struttura del progetto è quella mostrata di lato: come si può vedere abbiamo scelto di suddividere il progetto in più package, individuando per ciascuno di essi un ruolo specifico.
Abbiamo dunque un package per: Applicazione, Controller, Eccezioni, Modello, Repository e Service.



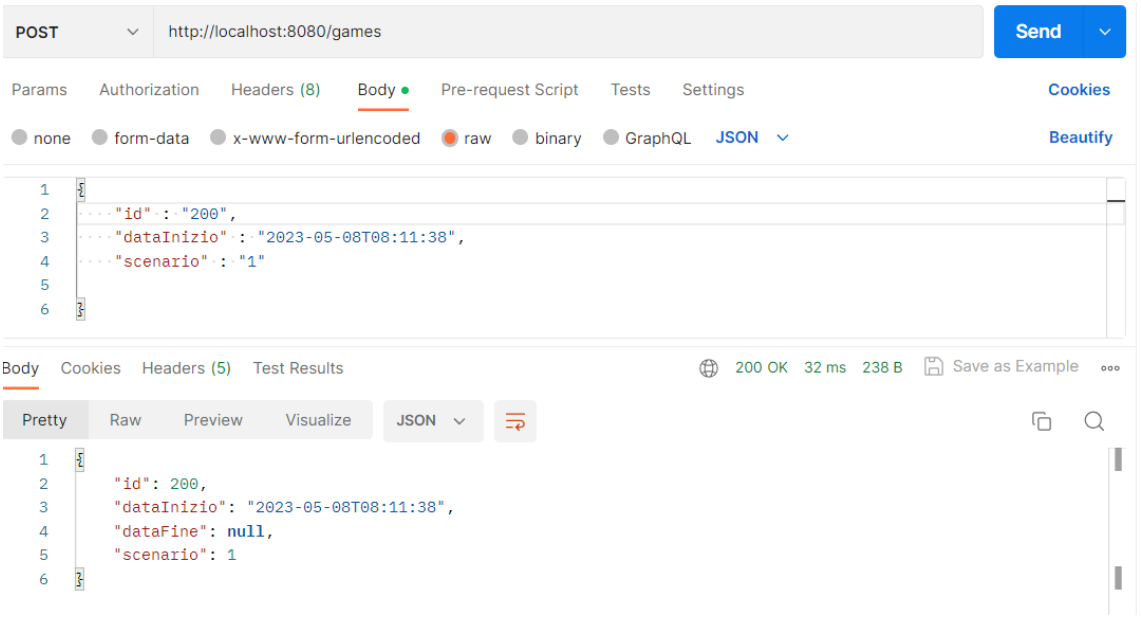
Di seguito forniamo il codice relativo al Controller creato, in modo da mostrare quali metodi sono stati offerti:

```java
package com.project.ProgettoSad.controller;

import java.util.List;

@RestController
public class GameController {

    @Autowired
    private GameService gameService;

    @GetMapping("/games")
    public ResponseEntity <List <Game>> getAllGames(){
        return ResponseEntity.ok().body(gameService.getAllGames());
    }

    @GetMapping("/games/{id}")
    public ResponseEntity <Game> getGameById(@PathVariable long id){
        return ResponseEntity.ok().body(gameService.getGameById(id));
    }

    @PostMapping("/games")
    public ResponseEntity <Game> createGame(@RequestBody Game game){
        return ResponseEntity.ok().body(this.gameService.createGame(game));
    }

    @PutMapping("/games/{id}")
    public ResponseEntity <Game> updateGame(@PathVariable long id, @RequestBody Game game){
        game.setId(id);
        return ResponseEntity.ok().body(this.gameService.updateGame(game));
    }

    @DeleteMapping("/games/{id}")
    public HttpStatus deleteGame(@PathVariable long id) {
        this.gameService.deleteGame(id);
        return HttpStatus.OK;
    }
}
```

Testing:

Una volta ultimata l'implementazione del prototipo ne abbiamo testato il funzionamento mediante Postman.

In primo luogo abbiamo fatto un'operazione di POST, in modo da inserire il primo documento nel database:



Infine abbiamo visto un esempio di GET, sia nel caso in cui il documento con l'Id indicato è effettivamente presente nella collection, sia nel caso in cui non lo è, in modo da assicurarci anche del funzionamento delle eccezioni: