

EduQuest

Progetto di Ingegneria del Software
a.a. 2025-2026

Luca Rainieri, Andrea Ranica, Mattia Turconi, Chenhao Wang

Indice

Introduzione.....	4
Organizzazione del lavoro.....	6
Sviluppo iterativo e task.....	6
Divisione del lavoro.....	7
Tecnologie.....	7
Ambiente di lavoro.....	8
Analisi e progettazione.....	10
Diagramma dei casi d'uso.....	10
Prima iterazione.....	10
Seconda iterazione.....	11
Casi d'uso in formato dettagliato.....	14
Prima iterazione.....	14
Seconda iterazione.....	18
Modello di dominio.....	22
Prima iterazione.....	22
Seconda iterazione.....	23
Diagrammi di sequenza di sistema (SSD).....	25
TakeQuiz Prima iterazione.....	25
Seconda iterazione.....	26
CreateChallenge Prima iterazione.....	27
Seconda iterazione.....	27
EditQuiz Prima iterazione.....	28
Seconda iterazione.....	28
Register Prima iterazione.....	29
Seconda iterazione.....	29
Contratti.....	30
Diagramma delle classi software.....	31
Prima iterazione.....	31
Seconda iterazione.....	32
UI package.....	33
Controller package.....	34
Security package.....	35
Service package.....	36
DTO package.....	39
Repository package.....	40
Domain package.....	41
Diagrammi di sequenza (SD).....	42
Prima iterazione.....	42
Seconda iterazione.....	44
Diagrammi delle macchine a stati (STM).....	46
Diagrammi delle attività (AD):.....	47
Analisi statica e qualità del codice.....	48

Sonarcloud.....	48
Understand.....	49
Testing e collaudo del sistema.....	54
Collaudo delle logiche temporali tramite Postman.....	54
Conclusione.....	55

Introduzione

EduQuest è un'applicazione web che trasforma i questionari in esperienze ludiche e interattive, rendendo la valutazione formativa più coinvolgente attraverso dinamiche di gioco e feedback in tempo reale. L'obiettivo è rendere l'apprendimento più coinvolgente e motivante, migliorando partecipazione e memorabilità dei contenuti grazie a meccaniche di gamification e monitoraggio puntuale delle performance.

Le funzionalità principali richieste in questa applicazione erano le seguenti:

- **Popolamento della banca dati di domande:** ogni utente registrato (che sia un insegnante o uno studente) ha la possibilità di aggiungere domande alla base di dati. Le domande possono essere aperte o chiuse, e ad ogni domanda può essere allegato un contenuto multimediale (come un'immagine, un video o un video da YouTube), riproducibile durante l'esecuzione di quest'ultime
- **Creazione di quiz interattivi con domande di diverse difficoltà:** i docenti hanno la facoltà di creare quiz per gli studenti, tramite l'aggiunta di domande dalla banca dati, la cui difficoltà determina la difficoltà complessiva del quiz, e di decidere se mostrarli o tenerli privati. Gli studenti possono compilare i quiz e, al termine, ricevere un feedback immediato sotto forma di valutazione percentuale, definita sulla base della correttezza degli esiti
- **Esecuzione di test a tempo con tentativi limitati:** a partire dai quiz, i docenti possono creare (ed eliminare dal database) test con limiti di tempo e numero massimo di tentativi. Gli studenti hanno la possibilità di compilare questi test solo per il numero di tentativi impostato ed entro il tempo scelto dall'autore del test
- **Statistiche quiz e domande:** i docenti possono vedere le statistiche generali di ogni domanda del database (numero di risposte date e tasso di successo) e le statistiche specifiche che concernono i quiz da loro creati (con punteggio medio e analisi delle risposte fornite); gli studenti, invece, hanno la possibilità di vedere le statistiche generali delle domande da loro create e dei quiz pubblici
- **Ranking studenti:** ogni utente registrato può vedere, tramite la sezione del profilo, le statistiche di tutti gli studenti, che riguardano il numero di quiz completati, le risposte fornite e la percentuale di successi. Tramite tali dati gli studenti vengono posizionati all'interno di classifiche, aumentando la competitività all'interno dell'applicazione
- **Missioni settimanali con badge per incentivare la partecipazione:** all'inizio di ogni settimana, vengono assegnate quattro missioni a ciascuno studente, sorteggiate casualmente dalla banca dati. Se viene raggiunto l'obiettivo, lo studente riceve un badge sul suo profilo, che riporta la data di completamento della missione e una breve descrizione di quest'ultima
- **Sfide tra studenti:** ogni studente ha la possibilità di sfidare altri studenti in una sfida asincrona 1 contro 1, su un quiz a sua scelta. Lo studente impone la durata di quella sfida: se scaduto il tempo un solo studente o nessuno dei due ha compilato il rispettivo quiz, la sfida viene annullata; se, invece, entrambi compilano il quiz, vince lo studente che ha totalizzato più punti. In caso di punteggio uguale, la sfida terminerà con un pareggio.

Inoltre è stata data importanza anche ai seguenti requisiti non funzionali:

- **Interfaccia semplice per studenti e insegnanti:** gli utenti possono usufruire di un'interfaccia semplice, ma curata, che permetta loro di servirsi in modo agevole di tutte le funzioni dell'applicazione

- **Performance nel caricamento delle pagine e nella latenza dei feedback:** le pagine vengono caricate rapidamente, con feedback immediato al termine della compilazione dei quiz
- **Compatibilità con dispositivi mobili e desktop tramite il browser:** l'applicazione è stata testata in ambiente desktop e verificata in modalità mobile emulation tramite i developer tools del browser
- **Affidabilità e meccanismi di salvataggio progressivo delle risposte:** il servizio è garantito sempre, tramite una logica di salvataggio parziale delle risposte, che per qualunque evenienza permette di riprendere da dove si è stati interrotti.

Organizzazione del lavoro

Sviluppo iterativo e task

Abbiamo deciso di organizzare il lavoro in modo iterativo, suddividendo lo sviluppo in due iterazioni. Durante la prima iterazione, lo scopo è stato quello di realizzare l'architettura del software e le funzionalità principali. Il lavoro svolto durante la prima iterazione è stato il seguente:

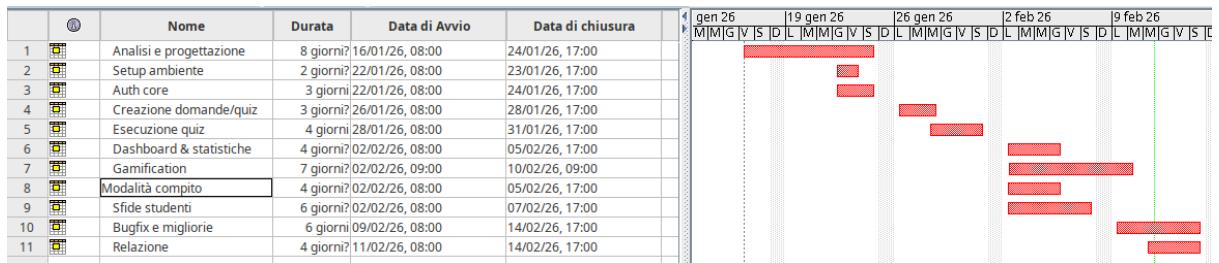
- **Analisi e progettazione:** abbiamo deciso di dedicare una porzione abbondante del tempo a nostra disposizione per effettuare la parte di analisi e progettazione del software nel modo più approfondito possibile. Nonostante poi inevitabilmente l'architettura iniziale abbia subito continui cambiamenti e aggiornamenti durante lo sviluppo (come verrà specificato successivamente in questa relazione), questo ci ha consentito di mantenere una visione d'insieme coerente di tutto il progetto, evitando modifiche importanti all'architettura durante lo sviluppo
- **Setup ambiente:** abbiamo dedicato qualche ora del nostro tempo per predisporre l'ambiente di lavoro; questo include la preparazione del repository Git (con le varie integrazioni tramite GitHub Actions come SonarQube), l'ambiente di esecuzione (con Eclipse e Maven) e una serie di tool utili durante lo sviluppo (Visual Studio Code, Postman, Google Chrome...)
- **Sviluppo delle funzionalità core:** una volta completata tutta la parte di analisi e progettazione, e avendo l'ambiente pronto per lo sviluppo, abbiamo iniziato a implementare l'architettura del software, soffermandoci sulle funzionalità principali, quali:
 - Autenticazione: login, registrazione
 - Creazione di domande e quiz
 - Una versione basilare di esecuzione dei quiz
- **Scrittura di test:** per garantire un corretto funzionamento dell'applicazione, abbiamo portato avanti l'attività di testing sin dalle prime funzionalità, in parallelo all'attività di sviluppo

Al termine della prima iterazione, abbiamo ottenuto un software che permetteva agli insegnanti di creare questionari (formati dalle domande create dagli utenti) e agli studenti di eseguire questi quiz.

Sulla base di quanto sviluppato nelle prime due settimane, la seconda iterazione ha riguardato:

- **Sviluppo di nuove funzionalità:**
 - Gamification: missioni, ranking, badge e sfide tra studenti
 - Modalità test: miglioria dell'esecuzione creata nella prima iterazione per integrare le funzionalità dei test
- **Miglioramento delle funzionalità già esistenti:** bug fixing, integrazione di mancanze emerse durante gli sviluppi, alcune attività di refactoring, estensione dei test per coprire le nuove funzionalità
- **Documentazione e relazione**

Quanto spiegato prima è visibile in questo diagramma di Gantt, che racchiude l'organizzazione del lavoro lungo entrambe le iterazioni.



La pianificazione iniziale del progetto si è rivelata abbastanza realistica: per completezza, riportiamo la prima versione del diagramma di Gantt fatta il primo giorno del progetto, che non ha subito modifiche drastiche rispetto a quella finale qui sopra riportata.



Alcune attività si sono rivelate più brevi di quanto preventivato (ad esempio la creazione dei quiz e delle domande o l'autenticazione), mentre altre sono state più impegnative del previsto (come le sfide tra studenti). Tutto sommato, siamo riusciti a rispettare le scadenze che ci eravamo prefissati per le varie attività.

Divisione del lavoro

La parte iniziale di analisi e progettazione è stata svolta principalmente di persona, in modo da collaborare al meglio per trovare una soluzione che fosse condivisa da tutti i membri del gruppo. Lo sviluppo è invece stato svolto in modo più autonomo da ogni membro del gruppo; nonostante questo, il team si è costantemente tenuto aggiornato tramite opportuni canali di comunicazione e riunioni a cadenza giornaliera per rimanere allineato sulle attività da svolgere.

Tecnologie

Data la natura web dell'applicazione, l'architettura del software è stata divisa in modo netto tra frontend e backend.

Per lo sviluppo lato server la scelta è ricaduta sul linguaggio di programmazione Java, in quanto è stato il linguaggio prevalentemente utilizzato da parte di tutti i membri del gruppo durante gli scorsi anni di università. Abbiamo adottato il framework Spring per la realizzazione del web service: si tratta di un ecosistema solido che ci ha semplificato l'implementazione dell'architettura REST.

Nello specifico, abbiamo scelto di utilizzare Spring Boot assieme a Java 21. I moduli principali che abbiamo integrato nella nostra applicazione sono stati:

- Spring Data JPA e H2 Database: gestione della persistenza dei dati. Abbiamo utilizzato un database in-memory che ci ha aiutato soprattutto nella fase di testing e di prototipazione
- Spring Security e JSON Web Token: gestione della sicurezza. L'autenticazione del nostro sistema è stata implementata con l'utilizzo dei token JWT, che sono uno standard moderno ed ampiamente utilizzato in ambito web service

- Cloudinary SDK: piattaforma usata per la gestione dei contenuti multimediali

La comunicazione tra backend e frontend avviene tramite l'esposizione di API RESTful. Il backend espone una serie di endpoint che ricevono e restituiscono dati in formato JSON, invocabili tramite i verbi standard del protocollo HTTP (*GET, POST, PUT, DELETE*). Questo approccio garantisce una separazione delle responsabilità tra la parte di frontend e backend, permettendo al frontend di occuparsi esclusivamente della presentazione, mentre tutta la logica di business viene implementata dal backend.

Esempio: se il frontend volesse ottenere la lista di tutti i quiz da visualizzare nella dashboard dello studente, dovrebbe effettuare una richiesta GET all'endpoint /api/quizzes.

Abbiamo scelto di utilizzare il build system Maven, che è ideale per lavorare con progetti Java e che permette di gestire in modo centralizzato tutte le dipendenze necessarie.

Il frontend è stato realizzato con il linguaggio di markup HTML5 e il linguaggio di scripting JavaScript “Vanilla”; infatti, abbiamo preferito non utilizzare dei framework come *React* o *Angular*. Questa scelta è stata dettata dalla volontà di mantenere il progetto snello e di approfondire la nostra conoscenza di JavaScript per la manipolazione del DOM. La parte estetica è stata gestita con l'utilizzo della libreria Bootstrap, che mette a disposizione una serie di componenti già pronti e con un comportamento responsivo su diversi tipi di dispositivi (desktop, mobile...)

Nonostante l'assenza di framework strutturati, abbiamo dato importanza alla modularità dell'interfaccia sfruttando la tecnologia dei Web Components: la maggior parte delle funzionalità è stata incapsulata all'interno di componenti riutilizzabili e spesso condivisi tra diverse pagine. Grazie a questo approccio, le varie componenti (come la barre di navigazione, la visualizzazione dei quiz oppure le informazioni di uno studente) sono state scritte una sola volta e istanziate in momenti e luoghi diversi dell'applicazione. Ogni componente interagisce con il DOM tramite attributi e proprietà, e comunica con altri componenti principalmente tramite l'emissione di eventi.

Ambiente di lavoro

L'intero codice sorgente è stato gestito tramite Git sulla repository GitHub che ci è stata assegnata all'inizio del progetto. Lo sviluppo è iniziato su un branch dedicato, chiamato *develop*, che è stato successivamente unito al branch *main* una volta implementate la maggior parte delle funzionalità.

Per la segnalazione di bug o migliorie varie sono state sfruttate le *issue* di GitHub: ogni issue è stata assegnata a un membro del gruppo, generalmente colui che si è occupato dell'implementazione relativa al problema segnalato.

Per l'analisi statica del codice abbiamo utilizzato SonarCloud tramite i GitHub Workflow: ad ogni commit effettuato sul branch *main* o *develop*, il codice è stato analizzato per permetterci di trovare problemi e migliorie al codice scritto fino al momento del commit.

Inoltre, abbiamo usufruito del tool Understand per rilevare al termine del progetto l'eventuale utilizzo di antipattern strutturali o code smells. Per i dettagli, si rimanda alla sezione dedicata [Analisi statica e qualità del codice.](#)

I rilasci e le build sono state gestite anch'esse con i GitHub Workflow tramite la pubblicazione di tag. Ad ogni nuovo tag (che identifica una versione, ad esempio v0.0.1) viene attivato automaticamente un workflow di CI/CD che gestisce la compilazione e la creazione di una nuova versione rilasciata.

Lo sviluppo del backend dell'applicazione è stato effettuato utilizzando l'IDE Eclipse, che offre un'ottima integrazione con l'ecosistema di Maven e con la libreria di testing JUnit.

Lo sviluppo del frontend invece è stato supportato dall'editor di testo Visual Studio Code; grazie alle numerose estensioni a disposizione ci ha permesso di avere un ambiente di sviluppo semplice, ma al tempo stesso completo, per le nostre necessità.

Infine, l'implementazione dei test è stata effettuata di pari passo con lo sviluppo delle funzionalità del software. Le tecnologie che abbiamo utilizzato per questo sono Junit, Mockito e Spring MockMvc.

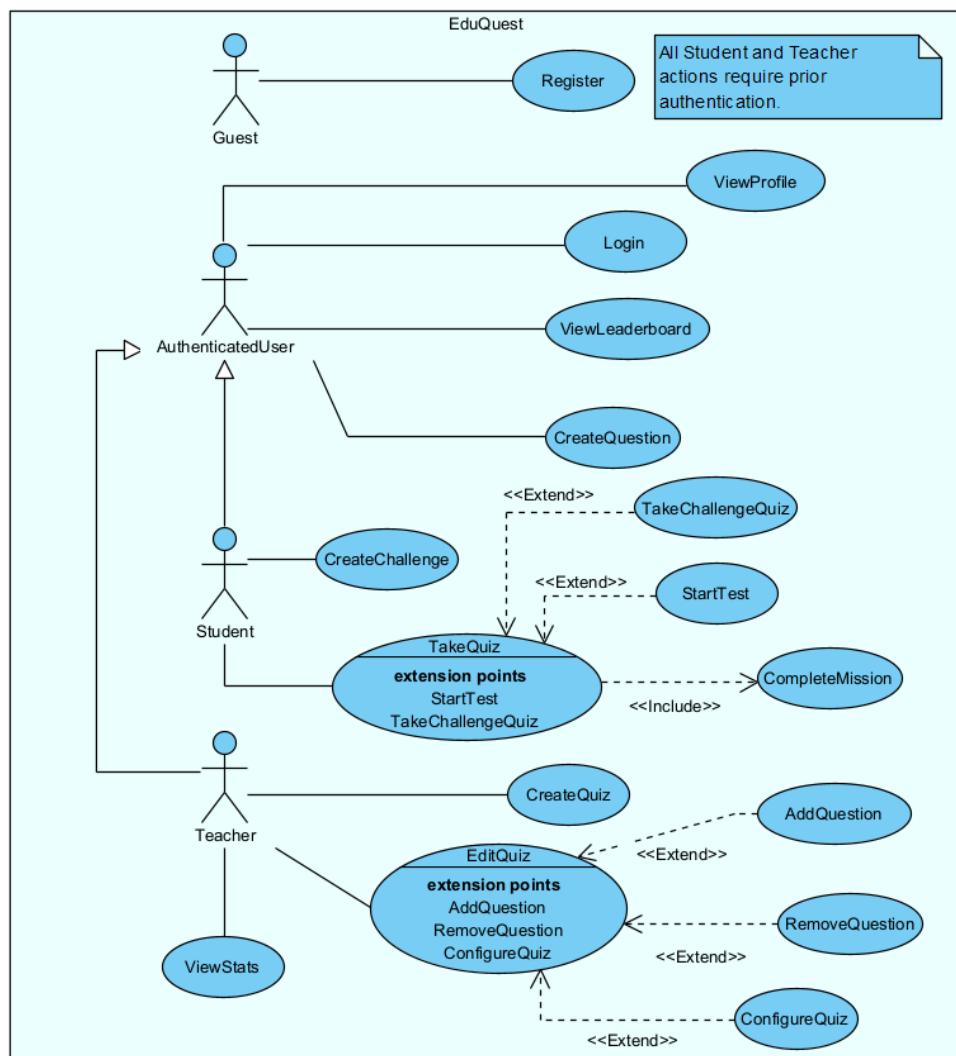
Analisi e progettazione

La parte più importante e decisiva dell'intero progetto è stata quella di analisi e progettazione del sistema: grazie a questa fase, abbiamo costruito le basi dell'architettura del nostro sistema, che sono state fondamentali per la fase di sviluppo. Per ogni sezione, inseriremo prima i diagrammi ideati nella prima iterazione (all'inizio del progetto), con una breve descrizione (se necessaria, in caso di cambiamenti significativi), poi sotto riporteremo la versione finale che abbiamo adottato nella seconda e ultima iterazione di questo progetto.

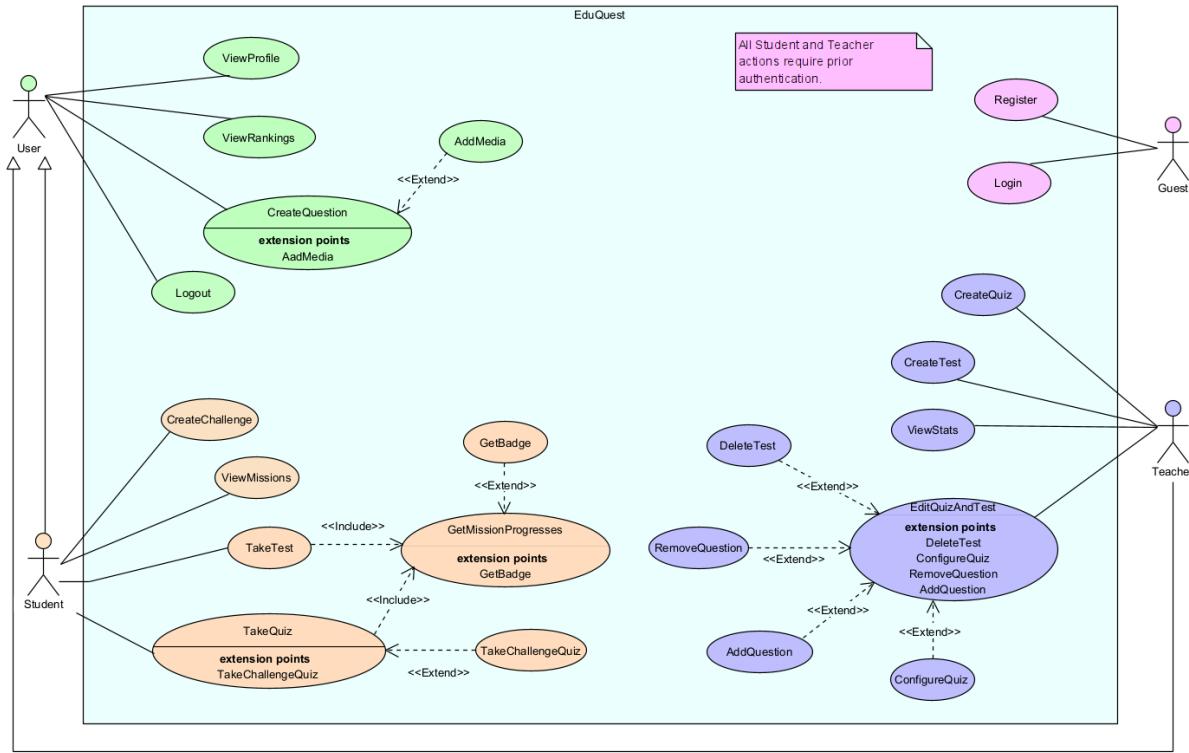
Diagramma dei casi d'uso

Il diagramma dei casi d'uso modella le interazioni funzionali tra gli utenti e il sistema, in questo caso Eduquest, definendo i confini del sistema e mostrando quali sono le funzionalità accessibili ai diversi attori che operano al suo interno.

Prima iterazione



Seconda iterazione



Sebbene i diagrammi, a prima vista, per via dei colori e della disposizione degli elementi, possano sembrare molto diversi, in realtà la logica di base è la stessa, non sussistono grandi differenze. Inizialmente, non avevamo considerato, avendo stilato un obiettivo minimo, di aggiungere i contenuti multimediali per le domande, ma durante l'implementazione ci siamo resi conto di avere le risorse per farlo, quindi abbiamo deciso di aggiungere questa funzionalità all'interno dell'applicazione.

Passando all'analisi dettagliata, nel sistema Eduquest osserviamo quattro attori:

- *Guest*: si tratta dell'utente non autenticato all'interno del sistema. Quando si apre l'applicazione, chiunque è un Guest. Le uniche azioni che può compiere, tramite rispettiva interfaccia, sono *Register* (per registrarsi) e *Login* (in caso di utente già registrato, per accedere al proprio profilo)
- *User*: utente autenticato, che ha passato la fase di login. Si tratta di una generalizzazione, in quanto all'interno del sistema vengono catalogati due tipi di users: *Student* e *Teacher*. Che si tratti di uno o dell'altro, entrambi possono svolgere alcune funzionalità basilari dell'applicazione, quali:
 - *ViewProfile*: ogni utente può visualizzare il proprio profilo, all'interno di una sezione dedicata. All'interno del profilo sono riportate le informazioni generali dell'utente, quali ID, nome, cognome, email e ruolo. Proprio quest'ultimo determina la visualizzazione, in questa interfaccia, delle statistiche (numero di quiz completati, media del punteggio ottenuto nei quiz, numero di risposte date e percentuale di risposte corrette) e dei badge ottenuti dallo studente (informazioni non visibili, ovviamente, all'interno del profilo di un docente). È presente, inoltre, una barra di ricerca, accessibile a tutti, per visualizzare il profilo di tutti gli utenti, consentendo ai docenti la possibilità di vedere le statistiche di ogni studente

- *ViewRankings*: abbiamo implementato un sistema di classifiche, accessibile da tutti, che ordina gli studenti in base a tre categorie: numero di quiz completati, media del punteggio ottenuto, numero di risposte corrette
- *CreateQuestion*: studenti e docenti possono creare domande, con la possibilità di aggiungere contenuti multimediali (*AddMedia*), nel database, sfruttabili poi dai docenti per popolare i quiz
- *Logout*: infine, ognuno può effettuare il logout dal proprio profilo
- *Student*: gli studenti sono gli utenti centrali del sistema, in quanto possono effettuare più operazioni dei docenti e sono i protagonisti della gamification, obiettivo dell'applicazione. Eccone in dettaglio i casi d'uso:
 - *TakeQuiz*: ogni studente può vedere la lista dei quiz creati e resi pubblici dai docenti e iniziarne la compilazione in apposita interfaccia. Può rispondere alle domande, scorrendo avanti e indietro, e può anche decidere di lasciare in sospeso la compilazione tornando alla home, in quanto le risposte parziali vengono salvate, riprendendo il quiz dalla home quando meglio preferisce. Al termine, può visualizzare il suo punteggio e rivedere le risposte date, per comprendere eventuali errori
 - *TakeTest*: lo studente può decidere di svolgere un test creato dal docente. Un test è un quiz, ma con numero massimo di tentativi e con tempo limite di compilazione. Come per i quiz, lo studente può lasciare in sospeso il test e riprenderlo successivamente, ma il timer continuerà a scorrere, prevenendo così la possibilità di *cheating* da parte sua. Una volta scaduto il timer o consegnato il test, verranno salvate le risposte date e lo studente verrà riportato alla home, dove potrà vedere i suoi risultati. Una volta terminati i tentativi, non potrà più tentare quel test
 - *CreateChallenge*: ciascun studente può decidere di dilettarsi sfidando altri suoi colleghi nella compilazione di uno specifico quiz (*TakeChallengeQuiz*), scelto in apposito menù, assieme allo studente da sfidare e alla durata massima (in ore) della sfida. Se entrambi compilano il quiz entro il tempo limite, la sfida verrà considerata conclusa e sarà determinato un vincitore oppure, in caso di punteggio uguale, verrà decretato il pareggio. Se anche solo uno dei due non completa la sfida entro il tempo limite, essa verrà considerata scaduta e non verrà stabilito il vincitore
 - *ViewMissions*: gli studenti, nella propria schermata home, possono visualizzare le missioni settimanali a loro fornite. Esse riguardano la compilazione di quiz, la risposta a domande...
 - *GetMissionProgresses*: gli studenti, compilando i quiz e i test e giocando alle sfide, compiono i progressi nelle missioni che, una volta completate, assegnano un badge specifico all'interno dell'apposita sezione del profilo (*GetBadge*)
- *Teacher*: i docenti sono l'altra faccia della medaglia dell'applicazione, coloro che gestiscono la parte relativa alla creazione dei quiz e dei test di cui possono usufruire gli studenti. Eccone le funzioni:
 - *CreateQuiz*: il docente, nella propria home, ha un bottone per creare un quiz, che se premuto apre un'interfaccia in cui deve inserire semplicemente nome e descrizione. Di default, il quiz creato è impostato privato, in quanto privo di domande (vedasi il caso d'uso *EditQuizAndTest*)
 - *CreateTest*: oltre alla possibilità di creazione del quiz, il docente ha anche un pulsante per la creazione dei test, che mostra un apposito menù in cui deve selezionare un quiz, il tempo limite (in minuti) e il numero massimo concesso di tentativi

- *EditQuizAndTest*: questo caso d'uso racchiude le funzionalità di gestione dei quiz e dei test. Un test può essere cancellato dalla banca dati (*DeleteTest*), tramite un bottone rosso posto nella home del docente. Per quanto riguarda i quiz, invece, ognuno di essi ha una pagina dedicata, dove il docente può modificare nome e descrizione del quiz e cambiare la sua visibilità tra pubblica e privata (*ConfigureQuiz*) e aggiungere (*AddQuestion*) o togliere (*RemoveQuestion*) domande al quiz
- *ViewStats*: infine, il docente può vedere le statistiche di ogni suo quiz, di tutte le domande o delle statistiche delle domande relative ad uno specifico quiz

Ogni azione di *Student* e di *Teacher* presuppone, naturalmente, un precedente login.

Casi d'uso in formato dettagliato

Prima iterazione

Nome del Caso d'Uso	TakeQuiz
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	Studente
Parti interessate e interessi	Lo studente vuole compilare un questionario preparato da un docente
Pre-condizioni	I dati dello studente sono già salvati nel sistema
Garanzia di successo	Il questionario è stato compilato correttamente
Scenario principale di successo	<ol style="list-style-type: none"> 1. Lo studente visualizza la lista di tutti i questionari presenti nel magazzino <ol style="list-style-type: none"> a. I questionari possono essere ordinati e filtrati 2. Lo studente sceglie il questionario da compilare 3. Lo studente avvia la sessione di compilazione 4. Per ogni domanda del questionario: <ol style="list-style-type: none"> a. Lo studente visualizza la domanda e le possibili risposte b. Lo studente scrive/sceglie la risposta che vuole dare c. Il sistema mostra la domanda successiva 5. Il sistema notifica il completamento della compilazione 6. Il sistema aggiorna l'avanzamento dell'utente nelle missioni
Estensioni	<ul style="list-style-type: none"> - 1a. Lo studente può decidere di filtrare i questionari in base ad alcuni filtri di ricerca (argomento, difficoltà...) - 4c1. Se lo studente si disconnette durante la compilazione, il sistema deve salvare la risposta di ogni domanda e permettere all'utente di riprendere da dove si era interrotto
Extension points	<p>StartTest:</p> <ul style="list-style-type: none"> - Alla fine della compilazione il docente deve confermare le risposte dell'utente, che verrà poi notificato con il punteggio ottenuto - Il compito presenta un limite di tempo entro cui deve essere completato, altrimenti verrà inviato con le risposte date fino al punto in cui il tempo è scaduto - Ogni compito ha un limite di tentativi a disposizione, quindi uno studente può compilarlo di nuovo per altre volte e verrà selezionato il punteggio maggiore <p>TakeChallengeQuiz:</p> <ul style="list-style-type: none"> - Dopo aver completato il questionario, la sfida viene aggiornata con il punteggio dello studente
Requisiti speciali	Il sistema deve garantire salvataggi parziali delle risposte date al questionario, in modo da salvare parte del lavoro in caso di disconnessioni.
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Ininterrotta; per compilare un questionario successivo è necessario finire il precedente

Nome del Caso d'Uso	CreateChallenge
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	Studente
Parti interessate e interessi	Lo studente (sfidante) vuole sfidare un altro studente (sfidato) a completare un questionario. Entrambi gli studenti hanno interesse nel vincere (compiere meno errori possibili) e nel completare il questionario entro la data di scadenza della sfida
Pre-condizioni	I dati dello studente devono essere già salvati nel sistema
Garanzia di successo	La sfida è stata inviata correttamente dallo studente sfidante e completata da entrambi
Scenario principale di successo	<ol style="list-style-type: none"> 1. Lo studente sfidante sceglie l'opzione per sfidare un altro studente 2. Lo studente sfidante sceglie il questionario 3. Lo studente sfidante sceglie la data di termine della sfida 4. Lo studente sfidante sceglie lo studente da sfidare 5. Il sistema notifica la creazione della sfida
Estensioni	Nessuna
Extension points	Nessuno
Requisiti speciali	Il sistema deve memorizzare la sfida e tenere le risposte di entrambi gli studenti in memoria fino a quando non è terminata
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Potenzialmente illimitata

Nome del caso d'uso	EditQuiz
Portata	Sistema Eduquest
Livello	Obiettivo utente
Attore primario	<i>Teacher</i>
Parti interessate e interessi	Il docente vuole correggere gli errori e aggiornare il contenuto di un quiz esistente
Pre-condizioni	L'attore è autenticato come docente. Il quiz deve esistere nel sistema e deve essere accessibile dal docente (cioè, creato da lui)
Garanzia di successo	I cambiamenti fatti vengono salvati nel sistema. La versione aggiornata è quella che verrà presentata agli studenti in futuro
Scenario principale di successo	<ol style="list-style-type: none"> 1. L'insegnante accede e visualizza l'elenco dei quiz creati 2. L'insegnante seleziona lo specifico quiz che intende modificare 3. L'insegnante apporta le modifiche necessarie 4. L'insegnante clicca sul pulsante per salvare le modifiche 5. Il sistema valida i nuovi dati (es. campi obbligatori vuoti) 6. Il sistema salva l'aggiornamento e notifica all'insegnante l'esito positivo dell'operazione
Estensioni	<ul style="list-style-type: none"> - 5.1. Annullamento delle modifiche: l'insegnante decide di non salvare e clicca su "annulla". Il sistema ripristina lo stato precedente senza salvare i dati - 6.1. Dati non validi: il sistema avvisa che i dati inseriti non sono validi. Il sistema propone di modificare i dati o di annullare l'operazione
Extension points	<ul style="list-style-type: none"> - AddQuestion - RemoveQuestion - ConfigureQuiz
Requisiti speciali	Nessuno
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Potenzialmente ininterrotta

Nome del Caso d'Uso	Register
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	Utente non registrato
Parti interessate e interessi	L'utente non registrato vuole registrarsi
Pre-condizioni	-
Garanzia di successo	L'utente si è registrato correttamente
Scenario principale di successo	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per registrarsi 2. Il sistema chiede di inserire i dati anagrafici per la registrazione e la scelta del ruolo (Studente o Docente) 3. L'utente inserisce i dati richiesti e conferma 4. Il sistema verifica che i dati siano validi, crea un nuovo profilo nel database e invia una conferma del successo
Estensioni	<ul style="list-style-type: none"> - 4.1. I dati inseriti dall'utente non sono validi. Il sistema da errore e ritorna al punto 3
Extension points	-
Requisiti speciali	Il sistema deve garantire la privacy dell'utente e l'unicità dell'account
Elenco delle variabili tecnologiche e dei dati	-
Frequenza di ripetizione	Una volta sola per utente non registrato

Seconda iterazione

Nome del Caso d'Uso	TakeQuiz
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	<i>Student</i>
Parti interessate e interessi	Lo studente vuole compilare un questionario preparato da un docente
Pre-condizioni	Lo studente ha effettuato il login ed esiste almeno un quiz pubblico nel sistema
Garanzia di successo	Il questionario è stato compilato correttamente
Scenario principale di successo	<ol style="list-style-type: none"> 1. Lo studente visualizza la lista di tutti i quiz presenti nella banca dati <ol style="list-style-type: none"> a. Si può ricercare il quiz tramite titolo tramite apposita barra di ricerca 2. Lo studente sceglie il quiz da compilare 3. Lo studente avvia la sessione di compilazione 4. Per ogni domanda del questionario: <ol style="list-style-type: none"> a. Lo studente visualizza la domanda e le possibili risposte b. Lo studente scrive/sceglie la risposta che vuole dare 5. Lo studente invia il quiz e il sistema notifica il completamento della compilazione 6. Il sistema aggiorna l'avanzamento dell'utente nelle missioni
Estensioni	<ul style="list-style-type: none"> - 4.1. Se lo studente si disconnette o torna alla home durante la compilazione, il sistema deve salvare la risposta di ogni domanda e permettere all'utente di riprendere da dove si era interrotto - 4.2. Lo studente può andare avanti e indietro tra le domande - 5.1. Lo studente può visualizzare il punteggio ottenuto e rivedere le risposte date per analizzare la sua compilazione
Extension points	<ul style="list-style-type: none"> - TakeChallengeQuiz: dopo aver completato il quiz, la sfida viene aggiornata con il punteggio dello studente. Se entrambi gli studenti hanno compilato il quiz, la sfida mostra il verdetto
Requisiti speciali	Il sistema deve garantire salvataggi parziali delle risposte date al questionario, in modo da salvare parte del lavoro in caso di disconnessioni
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Ininterrotta; per compilare un questionario successivo è necessario finire il precedente

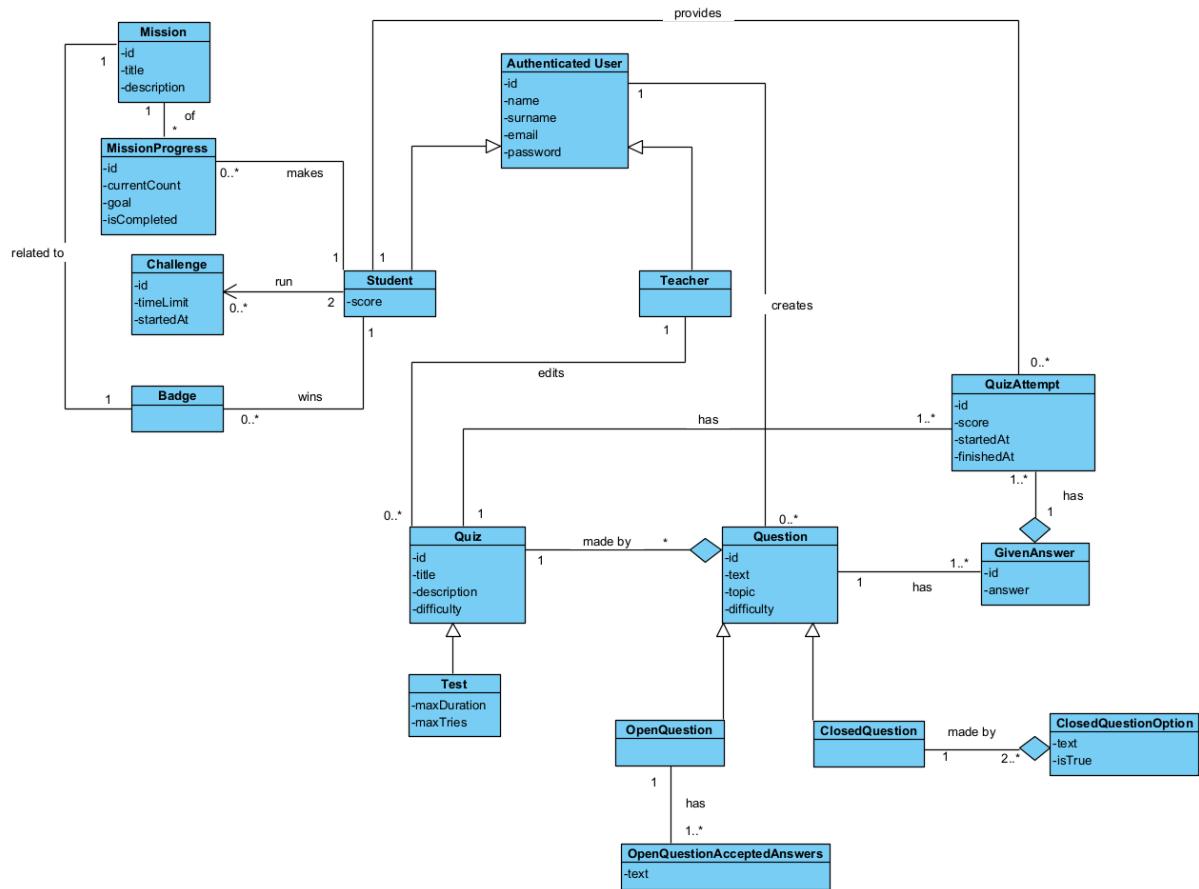
Nome del Caso d'Uso	CreateChallenge
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	<i>Student</i>
Parti interessate e interessi	Lo studente (<i>challenger</i>) vuole sfidare un altro studente (<i>opponent</i>) a completare un quiz. Entrambi gli studenti hanno interesse nel vincere (ottenere il punteggio più elevato possibile) e nel completare il questionario entro l'ora di scadenza della sfida
Pre-condizioni	Lo studente ha effettuato il login ed esiste almeno un quiz pubblico nel sistema
Garanzia di successo	La sfida è stata creata correttamente dallo studente sfidante e entrambi gli sfidanti hanno completato il quiz
Scenario principale di successo	<ol style="list-style-type: none"> 1. Lo studente sfidante sceglie l'opzione per sfidare un altro studente 2. Lo studente sfidante sceglie il quiz 3. Lo studente sfidante sceglie il numero di ore per la durata della sfida 4. Lo studente sfidante sceglie lo studente da sfidare 5. Il sistema notifica la creazione della sfida
Estensioni	Nessuna
Extension points	Nessuno
Requisiti speciali	Il sistema deve memorizzare la sfida e tenere le risposte di entrambi gli studenti in memoria fino a quando non è terminata
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Potenzialmente illimitata

Nome del Caso d'Uso	EditQuizAndTest
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	<i>Teacher</i>
Parti interessate e interessi	L'insegnante vuole aggiornare le informazioni di un quiz/test esistente
Pre-condizioni	Il docente ha effettuato il login e il quiz/test da modificare deve esistere nel sistema ed essere accessibile dall'insegnante (l'insegnante deve essere il suo autore).
Garanzia di successo	Le modifiche effettuate verranno salvate nel sistema. La versione aggiornata sarà quella mostrata agli studenti da quel momento in poi.
Scenario principale di successo	<ol style="list-style-type: none"> 1. L'insegnante accede e visualizza la lista dei propri quiz 2. L'insegnante sceglie il quiz che vuole modificare 3. Il sistema mostra i dettagli del quiz e le domande associate ad esso 4. L'insegnante effettua le modifiche necessarie 5. L'insegnante decide di salvare le modifiche 6. Il sistema valida i nuovi dati (ad esempio dati obbligatori) 7. Il sistema salva le modifiche e notifica l'insegnante che l'operazione è avvenuta con successo
Estensioni	<ul style="list-style-type: none"> - 6.1. Le modifiche effettuate non sono valide (es. il docente lascia campi vuoti o vuole rendere pubblico un quiz senza domande). Si torna al punto 4.
Extension points	<ul style="list-style-type: none"> - DeleteTest: il docente preme il pulsante “delete” per cancellare il test - AddQuestion: il docente aggiunge una domanda ad un quiz - RemoveQuestion: il docente rimuove una domanda dal quiz - ConfigureQuiz: il docente modifica titolo, descrizione o privacy del quiz
Requisiti speciali	Controllo di correttezza degli input del docente
Elenco delle variabili tecnologiche e dei dati	Nessuno
Frequenza di ripetizione	Potenzialmente illimitata

Nome del Caso d'Uso	Register
Portata	Sistema EduQuest
Livello	Obiettivo utente
Attore primario	<i>Guest</i>
Parti interessate e interessi	L'utente non registrato vuole registrarsi
Pre-condizioni	Nessuna
Garanzia di successo	L'utente si è registrato correttamente
Scenario principale di successo	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per registrarsi 2. Il sistema chiede di inserire i dati (nome, cognome, email e password) per la registrazione e la scelta del ruolo (Studente o Docente) 3. L'utente inserisce i dati richiesti e conferma 4. Il sistema verifica che i dati siano validi, crea un nuovo profilo nel database e invia una conferma del successo
Estensioni	<ul style="list-style-type: none"> - 4.1. I dati inseriti dall'utente non sono validi. Si torna al punto 3.
Extension points	Nessuno
Requisiti speciali	Il sistema deve garantire la privacy dell'utente e l'unicità dell'account
Elenco delle variabili tecnologiche e dei dati	Nessuna
Frequenza di ripetizione	Una volta sola per utente non registrato

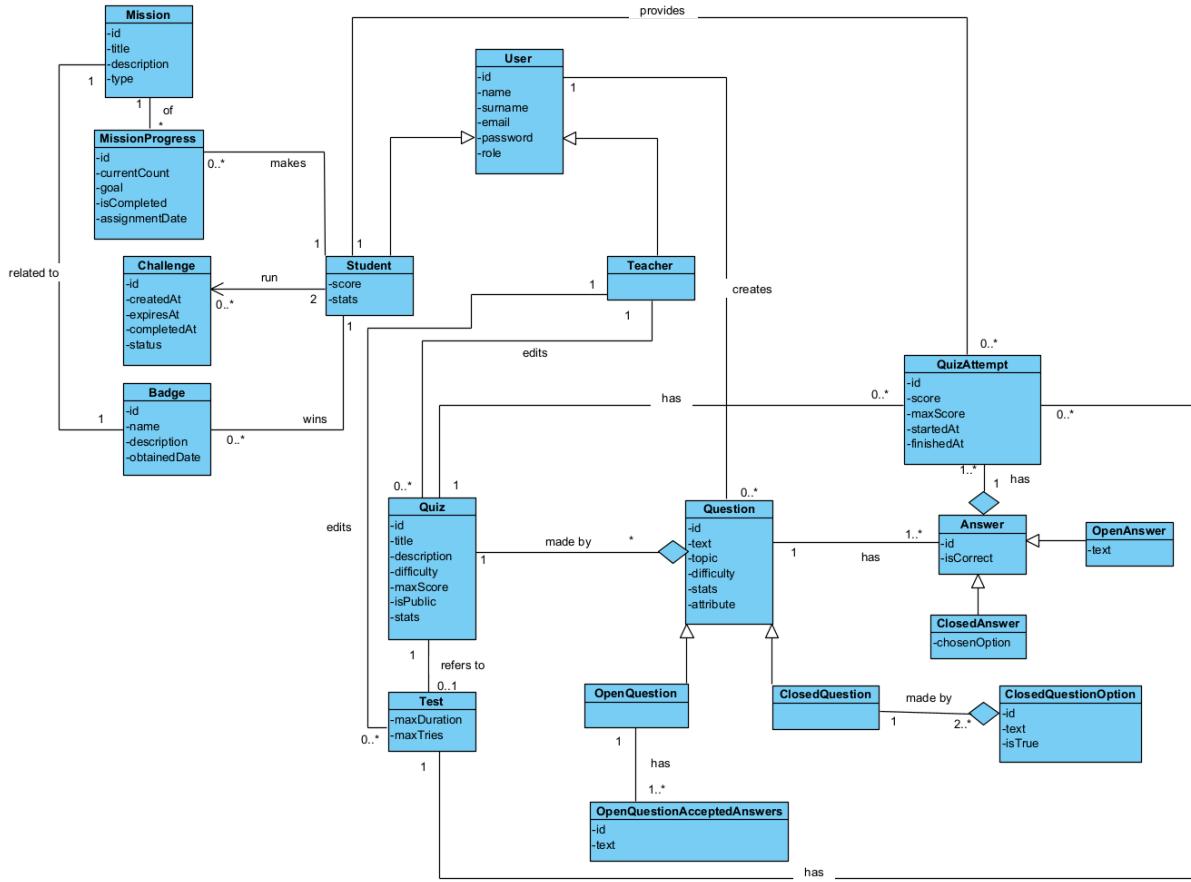
Modello di dominio

Prima iterazione



Nella prima iterazione, avevamo modellato l'entità *Quiz* come *generalizzazione* di *Test* (sottoclasse di *Quiz*). La nostra interpretazione iniziale, infatti, è stata quella di una relazione “*IS-A*”: un test appariva concettualmente come un tipo di quiz più strutturato. Tuttavia, procedendo con l'implementazione della logica di business, ci siamo resi conto che l'ereditarietà creava un forte accoppiamento che non rispecchiava la realtà del dominio. Nella nostra applicazione, infatti, il *Quiz* è un'entità indipendente e “giocabile”: rappresenta un'esercitazione libera, priva di vincoli formali, che lo studente può compilare in autonomia. Il *Test*, invece, può rappresentare una valutazione formale. Nella pratica, avremmo complicato inutilmente la trasformazione da quiz a test. In più, abbiamo aggiunto la possibilità ai docenti, tramite l'attributo *isPublic*, di rendere intercambiabilmente un quiz privato o pubblico a seconda delle necessità, permettendo quindi, se voluto, di tenere un quiz “nascosto” e pubblicarlo solamente come test.

Seconda iterazione



Le classi che stanno alla base del modello di dominio sono gli utenti, che rappresentano gli attori del sistema. La classe *User* rappresenta l'utente autenticato e contiene le sue informazioni (identificatore univoco, nome, cognome, email, password e ruolo).

Questa classe presenta due classi derivate:

- *Teacher*: ogni insegnante è responsabile della creazione e gestione di *Quiz*, *Test* e *Answer*
- *Student*: ogni studente si occupa di compilare *Quiz* e *Test*, creare domande (*Answer*), partecipare a sfide (*Challenge*) ed effettuare missioni (*Mission*) per guadagnare *Badge*

I due ruoli sono ben distinti: un insegnante non potrà eseguire quiz e uno studente non potrà crearne di nuovi, ad esempio. Questa distinzione, per quanto limitante, ci ha consentito di modellare con maggiore semplicità il sistema e le differenze fra gli utenti rispetto a un sistema più aperto in cui ogni utente potesse essere sia insegnante (potendo quindi creare quiz e test) che studente (partecipando a quiz e test, guadagnando badge...). Inoltre, questa scelta è coerente con i requisiti funzionali di partenza.

Abbiamo scelto di consentire sia a insegnanti che a studenti la creazione di domande: sebbene questa scelta fornisca un'estrema libertà agli studenti, con il rischio concreto di riempire il database con domande di basso livello, gli insegnanti sono liberi di utilizzare le domande che preferiscono per popolare i propri quiz, ignorando quelle considerate inopportune; inoltre, il numero di domande salvate nel sistema può crescere in modo molto veloce grazie al contributo di tutti gli utenti.

I Quiz sono le entità al centro del sistema EduQuest: ogni quiz consiste in un insieme di domande, e ogni studente può compilare un quiz ed ottenere un punteggio, basato sulla bontà delle risposte date. Ogni domanda può essere di due tipologie: aperta, in cui lo studente può inserire la risposta con un testo di lunghezza arbitraria, oppure chiusa, in cui lo studente può scegliere tra un numero limitato di opzioni.

- Nelle domande aperte, le risposte giuste devono essere scelte dal creatore della domanda in fase di creazione; questo significa che il docente deve prevedere le possibili risposte giuste fornite dallo studente in fase di compilazione, ed ogni risposta diversa da quelle scelte verrà considerata sbagliata
- Nelle domande chiuse, è possibile specificare le opzioni disponibili e scegliere l'opzione corretta. In particolare, abbiamo permesso agli utenti di creare domande con 2, 3 o 4 risposte possibili, includendo in questa categoria, quindi, anche le domande vero/falso

I Test sono delle entità che sono composte da un Quiz e una serie di limitazioni: la durata massima, cioè il tempo massimo entro cui il test deve essere completato dal momento in cui la sua compilazione viene iniziata, e il numero massimo di tentativi a disposizione di uno studente. Una volta esauriti i tentativi, lo studente non potrà più eseguire il test. Inoltre, abbiamo deciso di far proseguire lo scorrere del tempo del timer anche se lo studente esce volontariamente oppure in caso di *crash* per prevenire il *cheating* da parte degli studenti.

Quando uno studente inizia la compilazione di un quiz, il sistema memorizza lo stato di questa sessione all'interno della classe QuizAttempt. Questa classe contiene:

- Informazioni generali del tentativo, quali il tempo di inizio e il punteggio ottenuto, calcolato quando il tentativo viene terminato
- Le risposte fornite ad ogni domanda: questa struttura permette al sistema di memorizzare le risposte ogni volta che vengono inviate durante la compilazione, consentendo al sistema di salvare lo stato intermedio di un tentativo e riprenderlo in un momento successivo

Inoltre, lo studente può decidere di scorrere tra le domande del quiz/test, permettendogli completa flessibilità durante la compilazione.

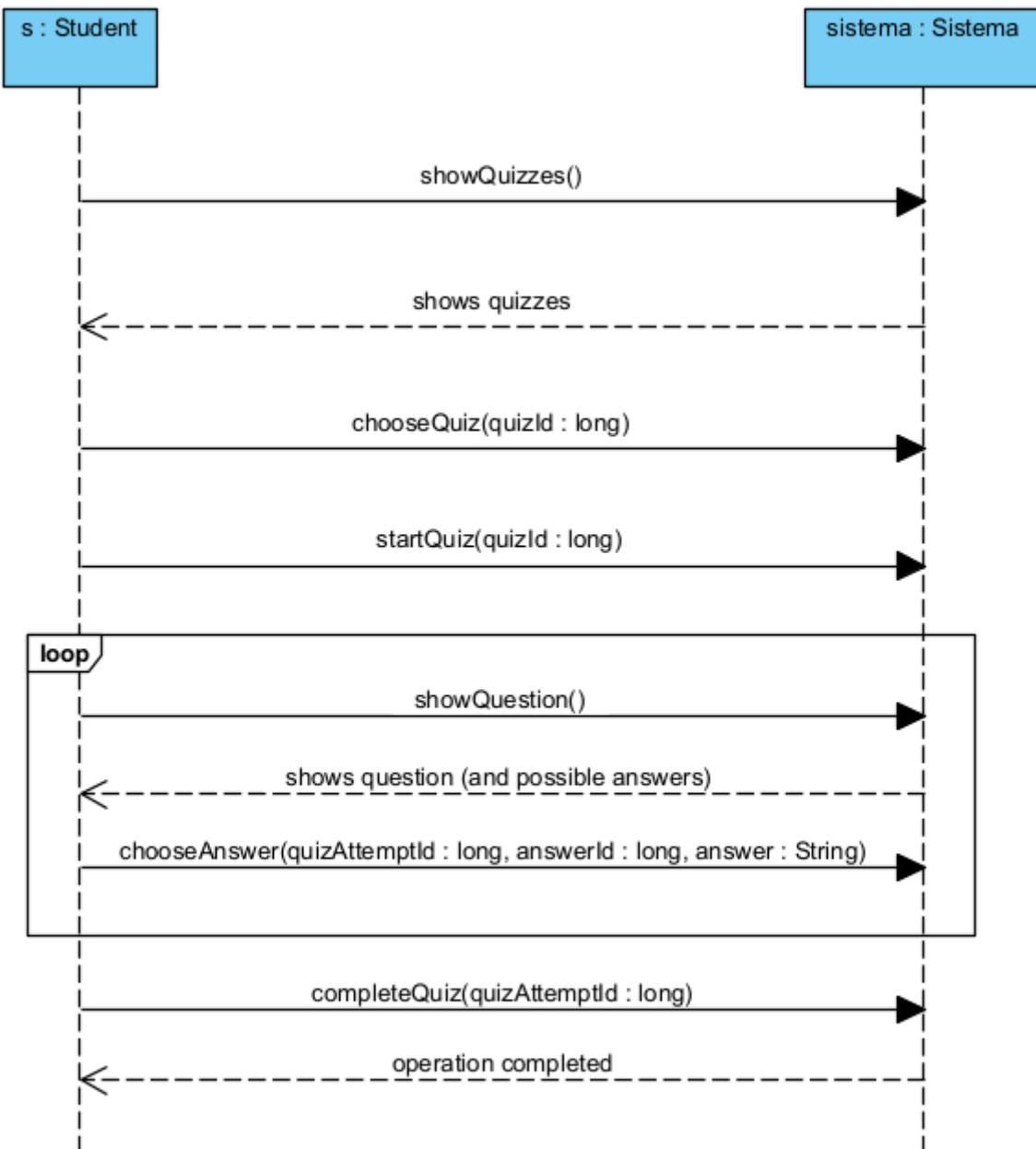
Il sistema di *gamification* è composto da una serie di entità che hanno l'obiettivo di incentivare la partecipazione dello studente ai quiz e ai test.

Il sistema mette a disposizione degli studenti una serie di missioni, modellate dalla classe *Mission*; viene memorizzato il progresso di ogni studente per ogni missione, in modo da aggiornarlo alla compilazione di ogni questionario. Quando una missione viene completata, lo studente guadagna un badge.

Ogni studente ha la possibilità di sfidare un altro studente attraverso le sfide, che consistono nella compilazione di un quiz entro un tempo limite. Quando entrambi gli studenti avranno completato le missioni, verrà scelto come vincitore lo studente che avrà ottenuto il punteggio più alto.

Diagrammi di sequenza di sistema (SSD)

TakeQuiz Prima iterazione



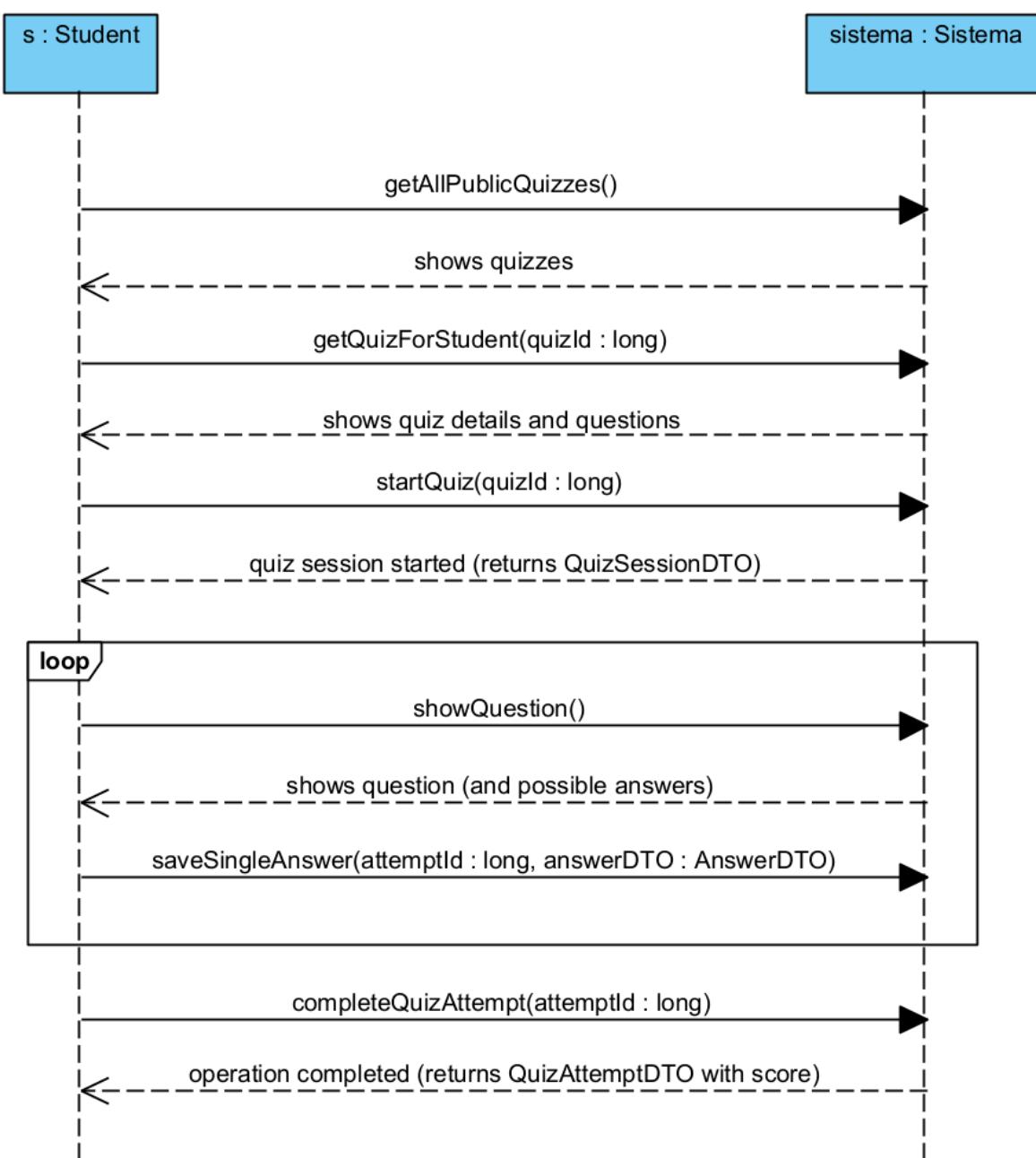
TakeQuiz: questo diagramma illustra la sequenza di interazioni tra lo studente e il sistema per svolgere la compilazione di un quiz. Lo studente chiede al sistema l'elenco dei quiz, dopodiché ne sceglie uno ed inizia la sua compilazione. Qui risiedono alcune differenze tra prima e seconda iterazione:

- Come anticipato, abbiamo aggiunto la possibilità ai docenti di modificare la pubblicazione dei quiz, permettendo loro di renderli privati. A differenza della prima versione, il backend ora filtra le risorse alla radice, restituendo al client esclusivamente i quiz aventi l'attributo *isPublic* a true. Questa validazione lato server garantisce che lo studente non possa mai richiedere l'esecuzione di un quiz privato

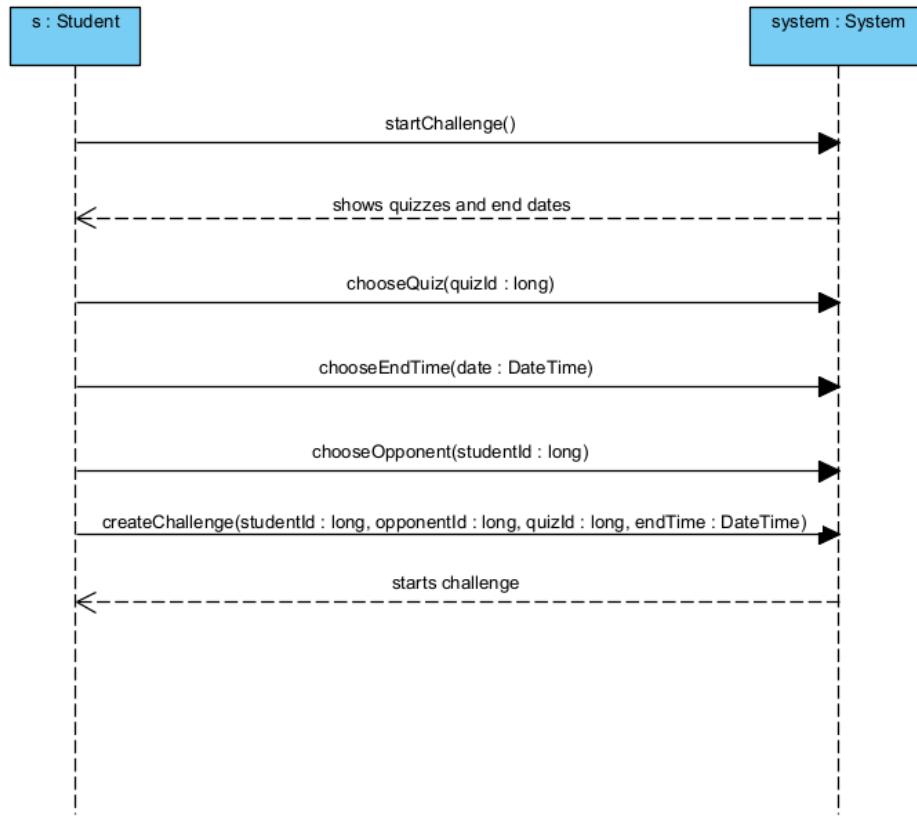
- Inoltre, per prevenire il cheating da parte di studenti più esperti (come l'ispezione dei pacchetti JSON in transito tramite il tab *network* dei developer tools del browser), tramite il metodo *getQuizForStudent* abbiamo applicato l'*information hiding*: il backend non invia le entità di dominio originali, bensì dei DTO "ciechi". I campi contenenti le soluzioni corrette vengono volutamente offuscati, impostando sistematicamente a *false* o rendendo *null* i campi risposta

Dopodiché, fino a quando ci sono domande rimaste, il sistema mostra una domanda all'utente, che fornisce una risposta e quest'ultima viene memorizzata dal sistema. Una volta finita la compilazione del quiz, lo studente chiede al sistema di completare il tentativo.

Seconda iterazione

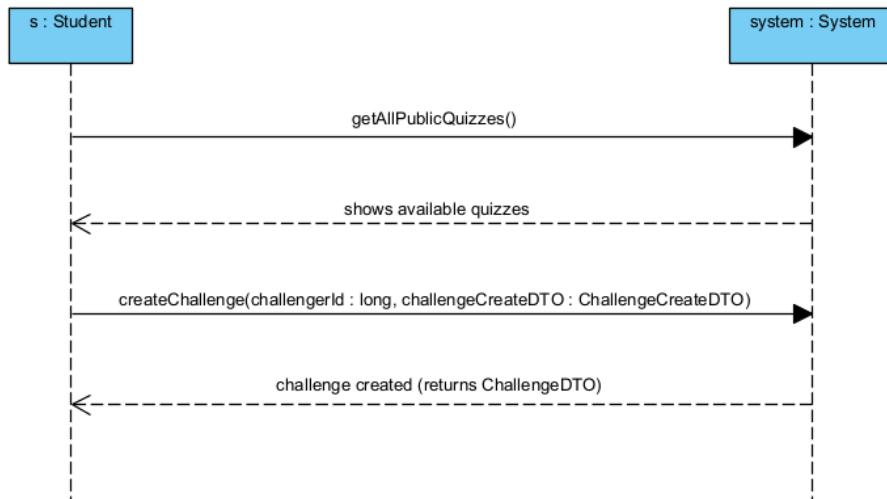


CreateChallenge Prima iterazione

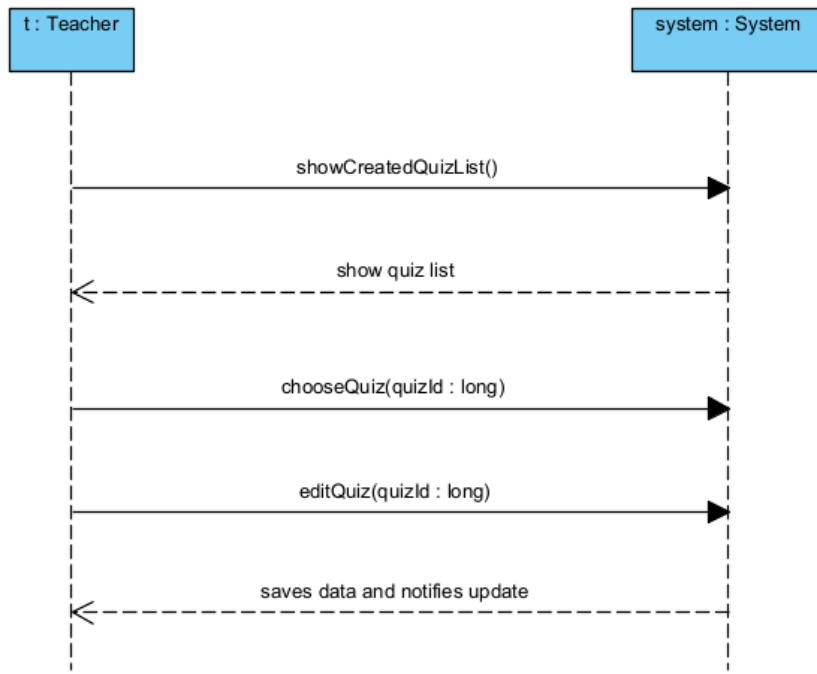


CreateChallenge: questo diagramma illustra la sequenza di interazioni tra lo studente e il sistema per creare una sfida tra lo studente richiedente ed uno sfidante. Lo studente chiede al sistema la lista dei quiz disponibili, dopodiché chiede al sistema di creare la sfida, che sarà svolta sul quiz e con l'avversario scelto dallo sfidante.

Seconda iterazione

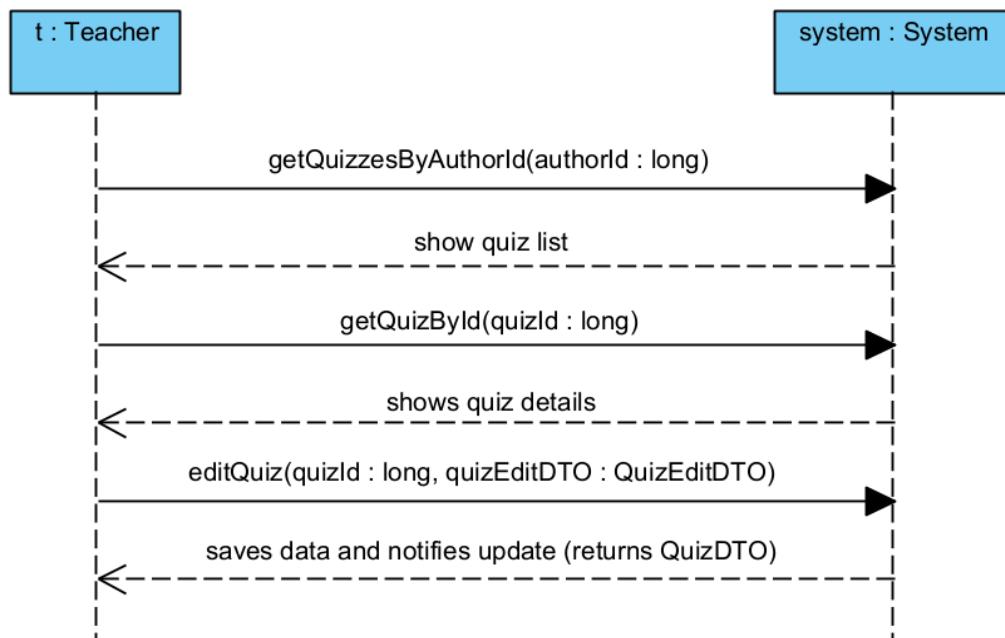


EditQuiz Prima iterazione

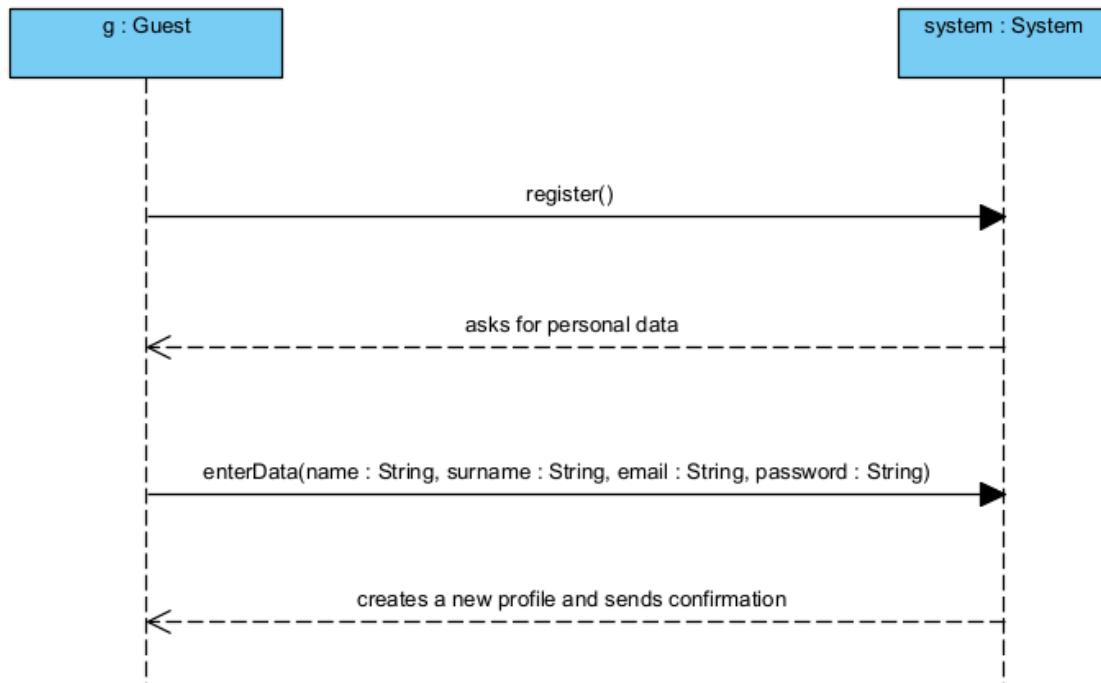


EditQuiz: questo diagramma illustra la sequenza di interazioni tra l'insegnante e il sistema per modificare le informazioni di un quiz. L'insegnante chiede al sistema di visualizzare tutti i quiz di cui è l'autore, dopodiché sceglie il quiz di cui vuole modificare le informazioni e ottiene i suoi dettagli. Infine l'insegnante chiede al sistema di modificare le informazioni (titolo, descrizione, visibilità, domande) di quel quiz.

Seconda iterazione

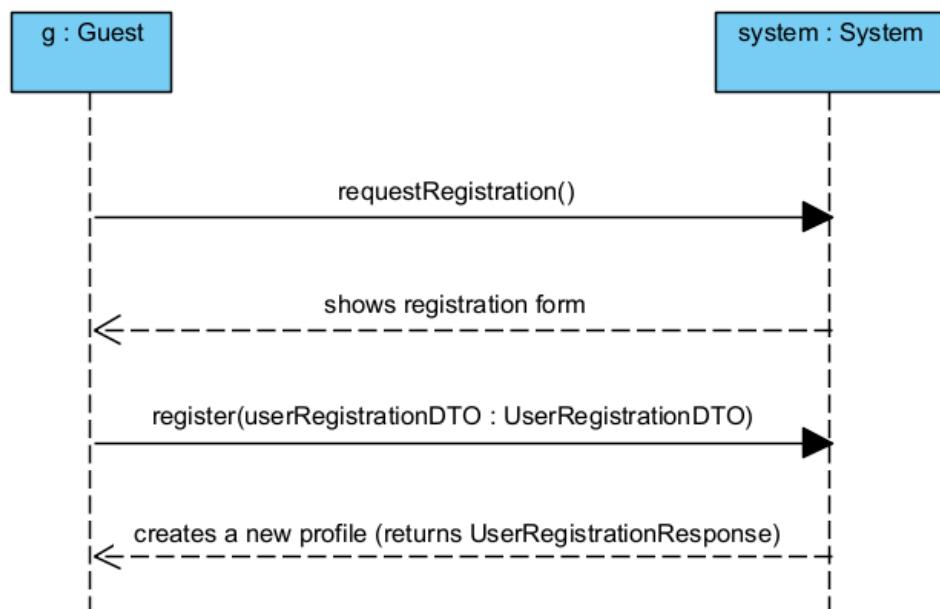


Register Prima iterazione



Register: questo diagramma illustra la sequenza di interazioni tra l'utente non registrato e il sistema per effettuare la registrazione. L'utente richiede al sistema di effettuare la registrazione, e inserisce i suoi dati. Il sistema riceve i dati e crea un profilo per l'utente.

Seconda iterazione



Contratti

Abbiamo deciso di specificare i contratti per le seguenti operazioni di sistema, in modo da avere una visione più dettagliata sugli effetti di queste operazioni.

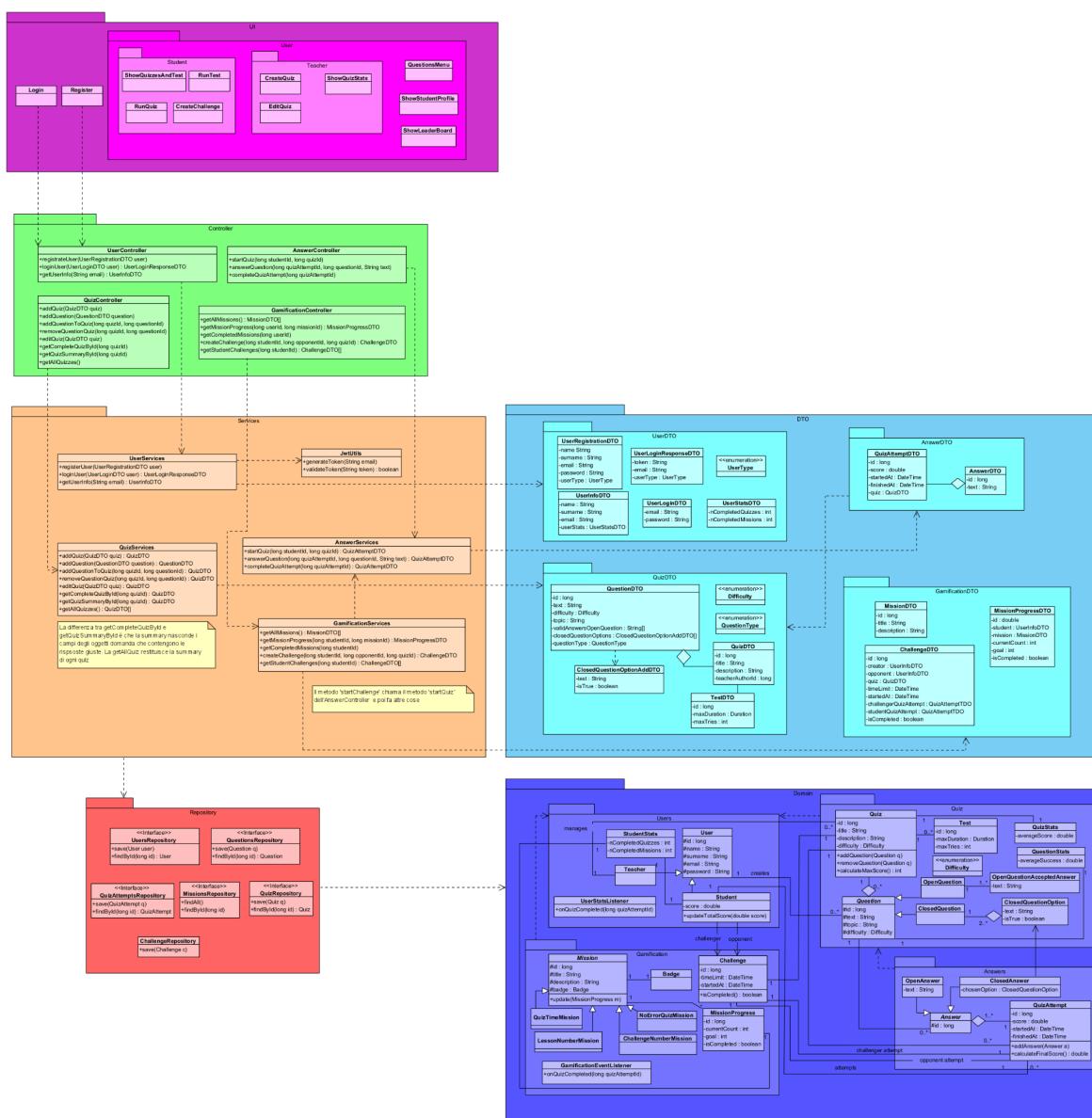
Contratto chooseAnswer	
Operazione	chooseAnswer(long quizAttemptId, long answerId, String answer)
Riferimenti casi d'uso	TakeQuiz
Precondizioni	<ul style="list-style-type: none"> - Lo studente ha richiesto e selezionato il quiz da effettuare - Il sistema sta mostrando una domanda del quiz - Lo studente ha selezionato o scritto la risposta alla domanda - È stata creata e inizializzata un'istanza di <i>QuizAttempt qa</i>, con identificatore <i>quizAttemptId</i>, relativa alla sessione in corso
Postcondizioni	<ul style="list-style-type: none"> - È stata creata un'istanza di <i>Answer a</i> e sono stati inizializzati i suoi attributi - L'attributo <i>answers</i> di <i>QuizAttempt qa</i> è stato aggiornato per includere <i>a</i>

Contratto completeQuiz	
Operazione	completeQuiz(long quizAttemptId)
Riferimenti casi d'uso	TakeQuiz
Precondizioni	<ul style="list-style-type: none"> - Lo studente ha terminato la compilazione di un quiz
Postcondizioni	<ul style="list-style-type: none"> - L'attributo <i>score</i> dell'istanza di <i>QuizAttempt a</i>, ottenuta tramite <i>quizAttemptId</i>, è stato aggiornato - Il progresso delle missioni dell'utente è stato aggiornato dal sistema

Contratto createChallenge	
Operazione	createChallenge(long studentId, long opponentId, long quizId, DateTime endTime)
Riferimenti casi d'uso	CreateChallenge
Precondizioni	<ul style="list-style-type: none"> - Lo <i>Student s</i> ha richiesto al sistema di creare una sfida - Lo <i>Student s</i> ha selezionato lo sfidante, il quiz e la durata della sfida
Postcondizioni	<ul style="list-style-type: none"> - È stata creata un'istanza di <i>Challenge c</i> e i suoi attributi sono stati inizializzati con i dati forniti

Diagramma delle classi software

Prima iterazione



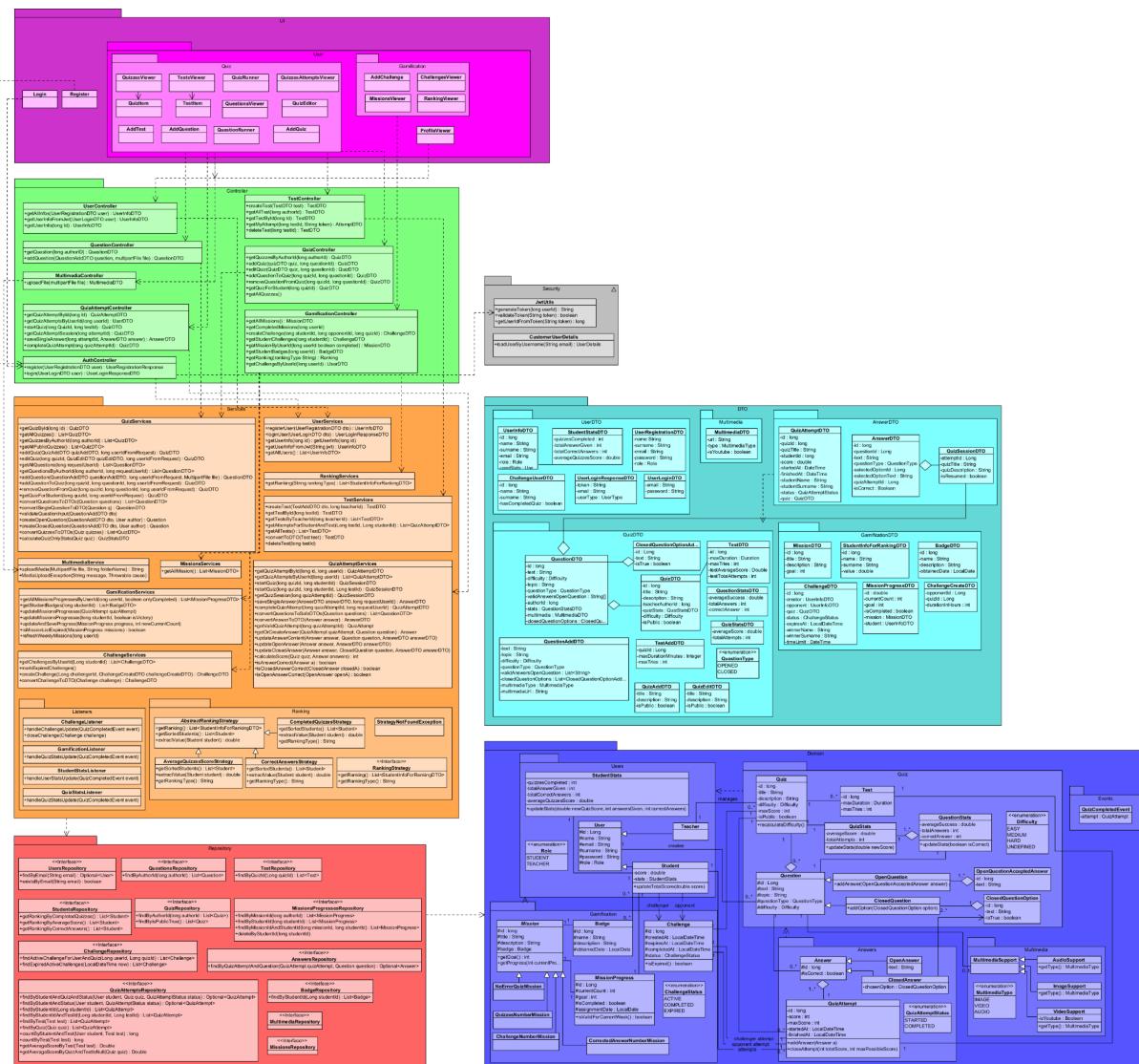
Per fornire una panoramica completa del lavoro svolto, riportiamo a confronto i diagrammi delle classi software globali (comprensivi dei layer architetturali) della prima e della seconda iterazione. Piuttosto che condurre un'analisi sulle singole classi, il confronto visivo tra i due modelli permette di evidenziare alcune macro-dinamiche fondamentali che hanno guidato la maturazione del sistema:

- Scalabilità della layered architecture: il diagramma finale risulta visibilmente più denso, riflettendo la massiccia aggiunta di funzionalità (gamification avanzata, multimedia, ranking). Tuttavia, grazie alla rigorosa divisione in layer (UI, Controller, Service, Repository, Domain, DTO), il sistema ha assorbito la complessità senza collassare, mantenendo i flussi di dipendenza ordinati
 - Estrazione del package *Security*: nella prima iterazione, le logiche di autenticazione (ad esempio, *JwtUtils*) erano parzialmente mischiate all'interno del layer dei services (blocco

arancione). Nell'iterazione finale, in piena ottica di *single responsibility principle* e di *separation of concerns*, è stato introdotto un package dedicato esclusivamente alla *Security* (blocco grigio)

- Inoltre, il layer dei *DTO* (blocco azzurro) ha subito una forte espansione speculare a quella del *Domain* (blocco blu scuro). Questo conferma la rigorosa adozione del pattern *DTO*: all'aumentare delle entità di dominio, abbiamo costantemente implementato i rispettivi oggetti di trasferimento per garantire l'*information hiding* e non esporre mai il modello interno ai client esterni.

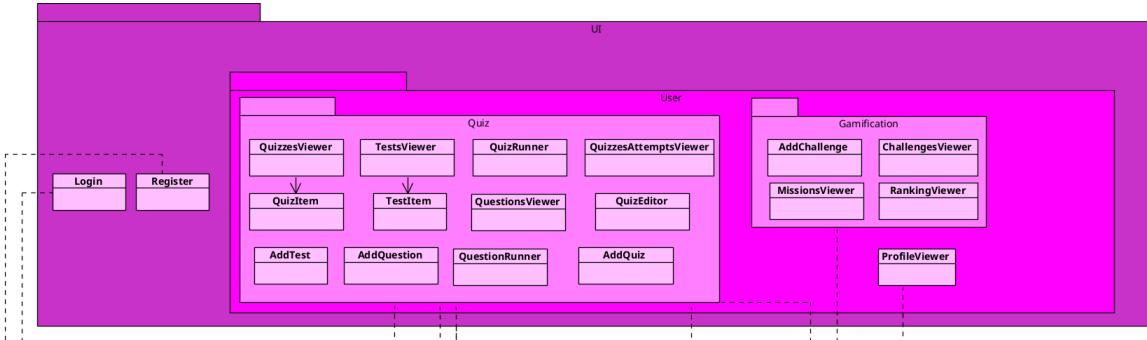
Seconda iterazione



L'architettura software di EduQuest è un'architettura a strati: ogni strato richiede un servizio agli strati sottostanti ed eroga servizi agli strati superiori. Questa struttura è ampiamente diffusa soprattutto in applicazioni web, vista la sua manutenibilità (è possibile modificare un livello dell'applicazione senza intaccare gli altri livelli) e facilità di testing (ogni livello può essere testato in autonomia).

Nei prossimi paragrafi analizzeremo nel dettaglio il funzionamento e la composizione di ogni livello dell'architettura.

UI package



Questo package rappresenta il livello client dell'applicazione, ossia l'interfaccia grafica che permette all'utente di interagire con il sistema. Lo strato di UI comunica direttamente con lo strato di controller per poter svolgere le operazioni di sistema.

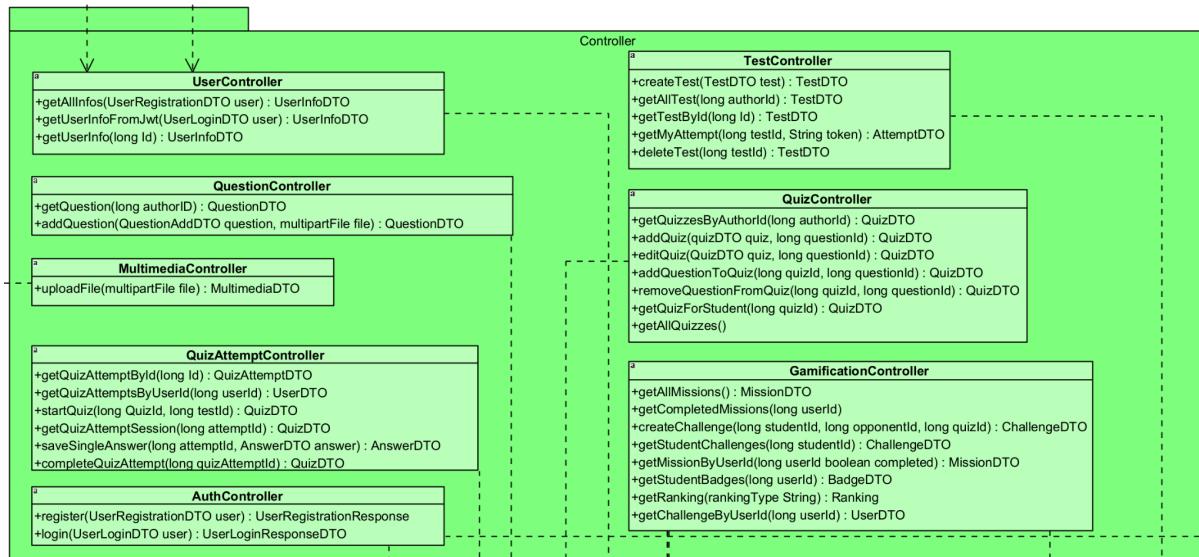
La realizzazione di questo livello dell'applicazione si basa sulla tecnologia dei *Web Components*, che consistono in un'insieme di funzionalità per creare elementi personalizzati e riutilizzabili nelle applicazioni. Ogni componente incapsula una funzionalità dell'interfaccia grafica, e permette di supportare la granularità del sistema anche senza l'utilizzo di un framework strutturato.

Possiamo dividere i componenti in alcune macro categorie:

- *AddChallenge, AddQuiz, AddQuestion, AddTest...*: si occupano di raccogliere i dati relativi a nuove challenge, quiz, domande e test, e di inviare questi dati al sistema, per la loro validazione e per la creazione di nuovi oggetti nel database
- *ChallengesViewer, MissionsViewer, QuizzesViewer, QuizzesAttemptsViewer...*: si occupano di raccogliere e visualizzare i dati relativi a challenge, missioni, profili ecc... Alcuni componenti possono anche integrare funzionalità aggiuntive, ad esempio per iniziare la compilazione di uno dei quiz visualizzati da *QuizzesViewer*
- *QuizRunner*: questo componente si occupa dell'esecuzione di un *QuizAttempt*, ossia di una compilazione di un quiz o di un test; al suo interno vengono visualizzate le domande del quiz e le opzioni disponibili per ogni domanda, assieme ai vari pulsanti necessari all'esperienza utente

La comunicazione tra i componenti e il sistema è intermediata da una serie di classi *Services* contenute all'interno del package frontend: queste classi si occupano di incapsulare l'invio della richiesta HTTP al backend, in modo che il componente non debba occuparsi dei dettagli tecnici della richiesta (token, endpoint...).

Controller package



Questo package contiene le classi *Controller*, ossia le prime classi del sistema che ricevono i comandi dall’utente (principalmente attraverso il livello di UI). Questi componenti si occupano esclusivamente di orchestrare le richieste ai livelli sottostanti, senza implementare la logica di business.

I metodi di questi controller possono essere invocati tramite delle API RESTful che comunicano tramite il linguaggio *JSON* e il protocollo *HTTP*. Per rappresentare gli oggetti scambiati attraverso le richieste sono stati utilizzati i *DTO*, evitando di esporre direttamente gli oggetti del livello di dominio. La realizzazione dei controller è stata semplificata grazie all’utilizzo delle funzionalità messe a disposizione da Java Spring, attraverso le annotazioni:

- `@RequestController`: informa il framework che la classe definita successivamente andrà trattata come un *controller*
- `@RequestMapping`: permette di specificare l’*endpoint* al quale dovranno essere inviate le richieste per interagire con i vari metodi del controller

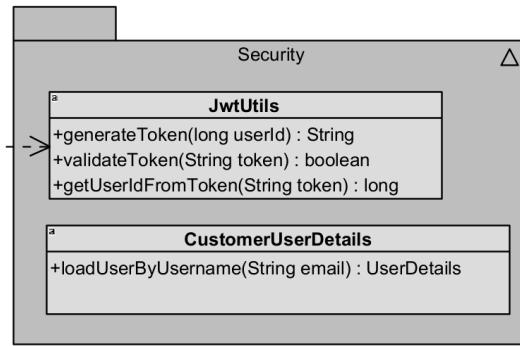
L’autenticazione è stata gestita usando i *Json Web Token*: ad ogni richiesta è allegato un token che contiene le informazioni dell’utente. Questo token è necessario per garantire l’accesso controllato alle risorse, ad esempio per fare in modo che solo gli utenti registrati (e quindi in possesso di un token valido) possano ottenere l’elenco dei quiz.

È stato creato un Controller per ogni area dell’applicazione:

- **UserController**: orchestra le richieste relative alle informazioni degli utenti
- **QuestionController**: gestisce le richieste relative alla creazione di domande
- **QuizController**: orchestra le richieste relative ai quiz, che vanno dal recupero delle informazioni fino alla creazione e modifica di nuovi quiz
- **TestController**: è un controller analogo a QuizController, ma si occupa della gestione dei Test
- **AuthController**: si occupa delle richieste di registrazione e login degli utenti
- **GamificationController**: gestisce le richieste delle funzionalità di gamification dell’applicazione, quali missioni, sfide tra studenti e ranking
- **QuizAttemptController**: orchestra le richieste relative all’esecuzione di quiz e test, come il salvataggio di una risposta o il completamento della compilazione di un quiz da parte di uno studente

- MultimediaController: necessario per il caricamento di file multimediali, utilizzato per gli allegati alle domande

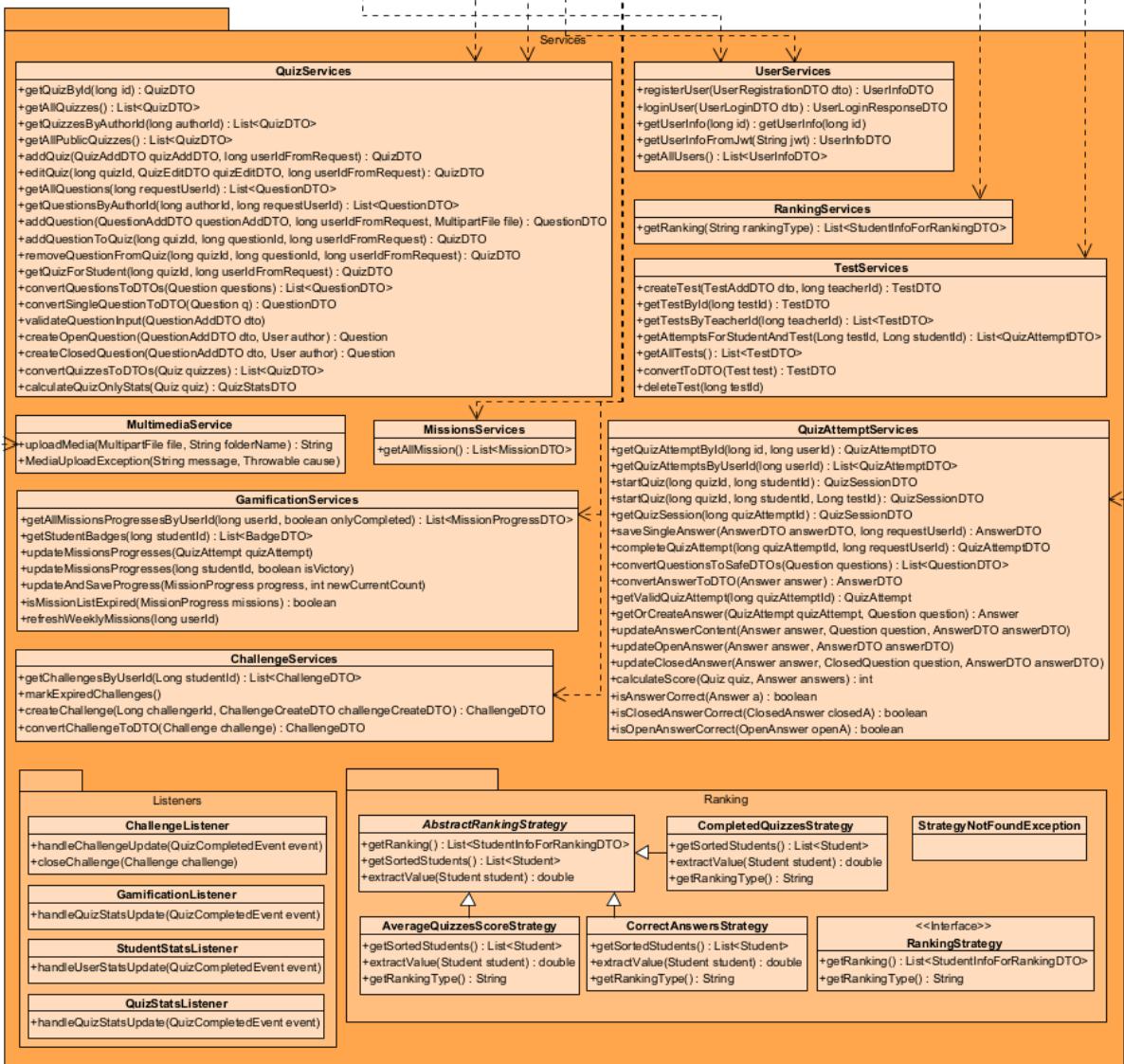
Security package



Contiene le classi necessarie alla gestione della sicurezza dell'applicazione. Avendo utilizzato il framework Java Spring, è stato sufficiente definire alcune classi per specificare il comportamento dell'applicazione in termini di sicurezza, mentre il framework si è occupato dell'implementazione a basso livello di queste specifiche. In particolare, abbiamo definito le seguenti classi:

- *JwtUtils*: come citato nella sezione dedicata al package *Controller*, l'autenticazione dell'applicazione è stata gestita utilizzando i Json Web Token. Questa classe contiene i metodi necessari a generare e validare questi token, insieme a un metodo per estrarre le informazioni contenute in un token
- *JwtRequestFilter*: questa classe contiene un metodo che si comporta da filtro durante l'invio di una richiesta al controller; questo metodo ha il compito di verificare che il token contenuto nella richiesta HTTP sia valido ed estrarre l'identificatore dell'utente a cui appartiene il token
- *CustomUserDetailsService*: funge da ponte tra il framework di sicurezza e il database, caricando i dettagli dell'utente necessari per l'autorizzazione
- *SecurityConfig*: questa classe si occupa di definire una configurazione di sicurezza riguardo all'accesso alle risorse dell'applicazione; ad esempio, vengono specificati i permessi relativi alle varie risorse e i filtri da applicare, tra cui il filtro dei Jwt definito precedentemente

Service package



Rappresenta il nucleo funzionale del sistema, in cui le classi di cui è composto (i *service*) implementano le regole di business e garantiscono l'integrità transazionale delle operazioni. Questi oggetti si occupano anche di effettuare le opportune verifiche sulla validità degli input forniti dall'utente, garantendo un accesso sicuro alle risorse private di ogni utente e generando errori che vengono intercettati e riportati agli utenti dal livello di controller.

Come avvenuto per i controller, sono state sfruttate le funzionalità di Java Spring per realizzare queste classi, in particolare grazie all'annotazione `@Service`.

Il core didattico dell'applicazione è formato da:

- Creazione di domande (studenti e insegnanti)
- Creazione di quiz e test (insegnanti)
- Aggiunta e rimozione di domande da un quiz (insegnanti)
- Modifica delle informazioni generali di un quiz (insegnanti)

- Esecuzione di quiz e test (studenti)

Le classi che compongono il core didattico sono:

- *QuizServices*: rappresenta il nucleo della logica di business per la gestione dei quiz. Coordina le operazioni di recupero dati interfacciandosi con il repository e gestisce l'aggiornamento strutturale delle entità (come la modifica dei dettagli del quiz e l'inserimento di nuove domande). Infine, si occupa di mappare le entità di dominio nei rispettivi DTO, garantendo l'information hiding e alleggerendo il traffico di rete
- *TestServices*: il gestore delle sessioni relative ai test, si occupa di creare un test con regole specifiche (durata, tentativi massimi), di interrogare il repository per recuperare un test e, in generale, tutte le operazioni relative ai test
- *QuizAttemptServices*: motore dell'esecuzione, gestisce il ciclo di vita dinamico della sessione per lo studente con operazioni per iniziarla e salvarla, validare le risposte date e calcolare il risultato

Nel modulo *gamification* vengono gestite tutte le funzionalità che hanno l'obiettivo di rendere l'apprendimento divertente e stimolante. Le funzionalità fornite da questo sotto-package si riferiscono principalmente agli studenti, e sono:

- Visualizzazione dello stato delle missioni di ogni studente
- Visualizzazione delle classifiche secondo diverse statistiche
- Creazione e visualizzazione delle sfide tra due studenti

Le classi che compongono questo package sono:

- *RankingServices*: mostra il calcolo del punteggio ordinato seguendo un sistema di punti
- *GamificationServices*: funge da motore principale per tutte le meccaniche di *gamification* dell'applicazione. Incapsula la logica di business necessaria a monitorare il progresso degli utenti e gestisce il ciclo di vita delle missioni. Inoltre, sfrutta un approccio a eventi: viene infatti invocato in modo reattivo da appositi listener in risposta a specifici trigger di sistema (ad esempio, la consegna di un test), aggiornando punteggi e badge
- *ChallengeServices*: gestisce l'aspetto competitivo dell'applicazione, controllando le sfide 1vs1, la loro creazione e scadenza

Per modellare i diversi tipi di ranking abbiamo adottato i design pattern *Strategy* e *Template Method*:

- **Pattern strategy**: è stato utilizzato per sfruttare diversi tipi di ordinamento e calcolo della classifica. La classe *RankingServices* funge da *context*, mantenendo un riferimento alla strategia selezionata e delegando il calcolo alla strategia iniettata a Runtime. La scelta di questo pattern ci consente di rendere il sistema di ranking facilmente estendibile, in quanto è sufficiente definire una nuova classe che implementa un nuovo algoritmo di ordinamento per poter realizzare una nuova classifica
- **Pattern template method**: è stato sfruttato per definire la struttura di base di un algoritmo di ranking (e quindi di una strategia) all'interno di una classe astratta. Questa classe astratta implementa la chiamata ad un metodo per ottenere gli studenti ordinati, per poi occuparsi di trasformare gli studenti nei corrispondenti DTO e restituire quindi la lista ordinata; in ogni strategia, è quindi necessario implementare esclusivamente 3 metodi:
 - Estrazione dei dati dal repository degli studenti
 - Estrazione del valore, per ottenere il valore da inserire nel DTO (ad esempio il numero di quiz completati)
 - Definizione del nome della strategia

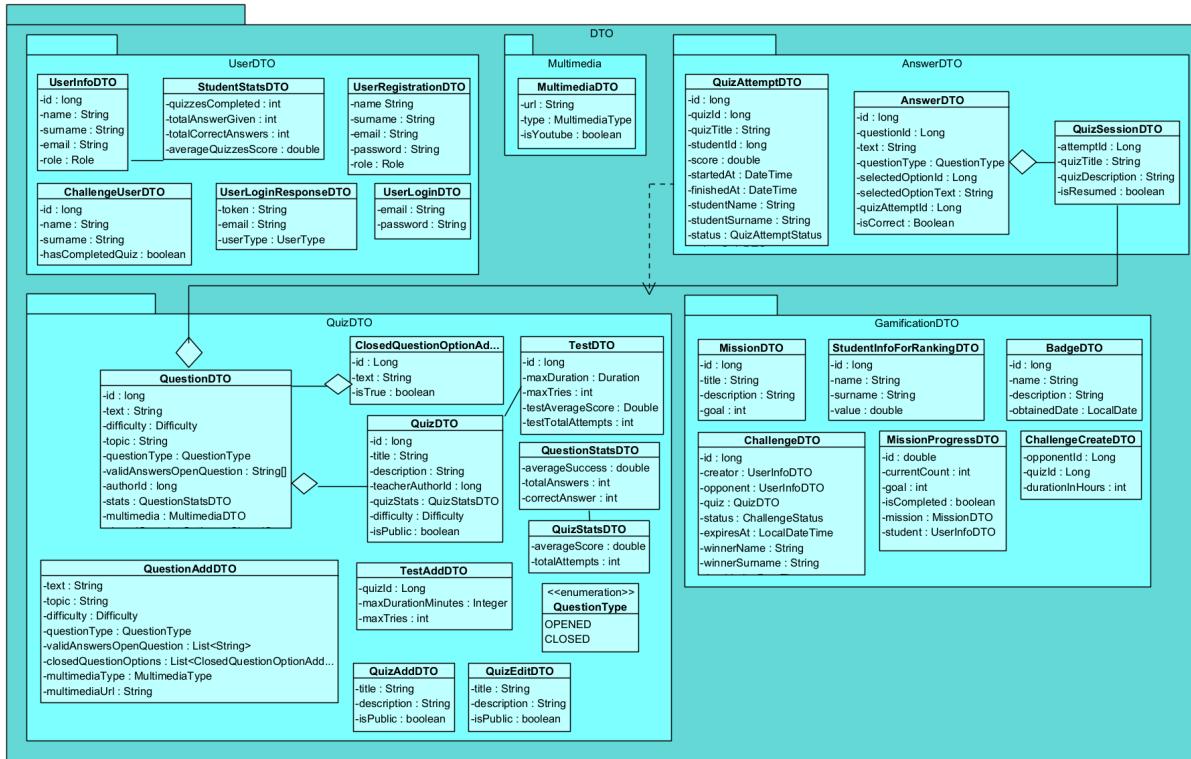
Inoltre, al fine di ottimizzare le risorse di sistema, un'importante scelta progettuale ha riguardato la gestione del ciclo di vita delle missioni settimanali assegnate agli studenti. Invece di adottare un approccio basato su task schedulati (ad esempio, una scansione notturna ogni domenica per rinnovare le missioni di tutti gli studenti), il team ha optato per l'applicazione del **pattern lazy load**, cioè per una generazione *on-demand*. La logica di validazione temporale risiede nell'entità di dominio (*MissionProgress*, tramite il metodo *isValidForCurrentWeek()*), ma è il *GamificationServices* che si occupa di istanziare e salvare le nuove missioni esclusivamente quando l'utente si autentica o richiede di visualizzare i propri progressi. Questa scelta garantisce un'elevata scalabilità del sistema, evitando query massive e inutili sul database per utenti inattivi.

Abbiamo inserito una classe all'interno dei *services* per supportare la gestione degli utenti. Le funzionalità implementate sono contenute all'interno della classe *UserServices*, che consente di registrare nuovi utenti, autenticare un utente tramite email e password, e recuperare le informazioni degli utenti.

Infine, la classe *MultimediaServices* si occupa della gestione dei file collegati alle domande e gli errori di upload.

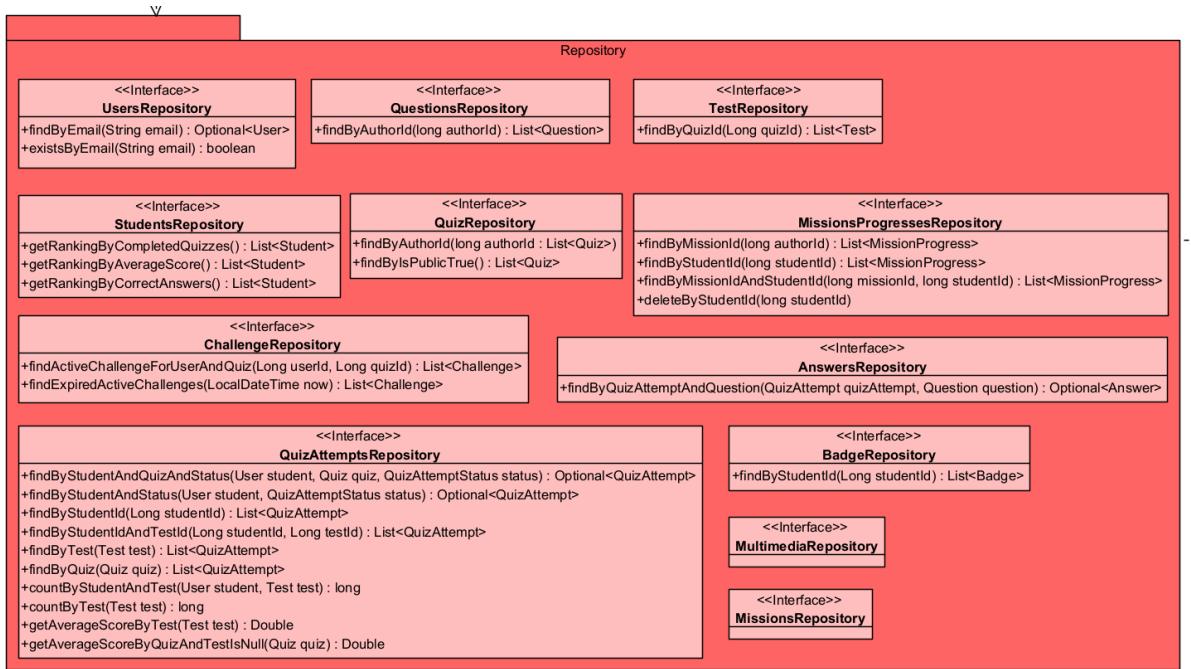
Alcune parti dell'applicazione, come l'aggiornamento delle statistiche e delle missioni degli studenti, sono state implementate utilizzando l'*observer pattern*, realizzato sfruttando le potenzialità del framework Spring. Quando la compilazione di un quiz viene completata, la classe *QuizAttemptServices* si occupa di generare un evento che viene intercettato da una serie di classi *Listener*. Queste classi, una volta captato l'evento, si occuperanno di aggiornare le informazioni dell'utente coinvolto con la compilazione del quiz: vengono aggiornate le sue statistiche, i progressi nelle missioni e nelle sfide, e infine vengono aggiornate le statistiche del quiz. Grazie a questo pattern è stato possibile svincolare la logica di aggiornamento di queste informazioni dal *QuizAttemptServices*, che invece è stata inserita all'interno dei listener.

DTO package



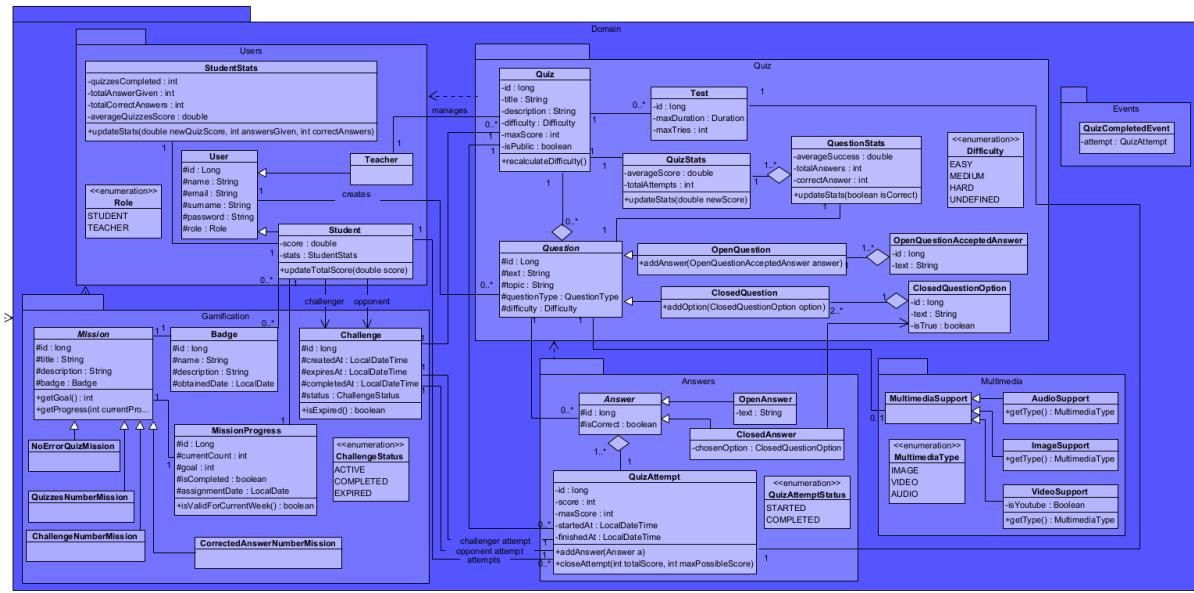
All'interno di questo package sono stati definiti tutti i *Data Transfer Object*, utilizzati per implementare un livello di astrazione tra il package *domain* e il livello di *UI*. Grazie a questo pattern, l'utente non interagisce direttamente con gli oggetti del sistema, ma con rappresentazioni ottimizzate che permettono di garantire la sicurezza dei dati, consentendo, ad esempio, di oscurare alcuni dati che non dovrebbero in alcun modo essere forniti all'utente (come la password di uno studente).

Repository package



Questo livello si occupa di astrarre l'accesso al database da parte dei livelli superiori, utilizzando il pattern DAO (*Data Access Object*) tramite le interfacce di Spring JPA. Grazie alle potenzialità del framework Spring, è stato sufficiente definire una serie di interfacce per ogni entità del dominio memorizzata nella base di dati, mentre tutta la logica di accesso e query è stata gestita dal framework.

Domain package



Questo package contiene tutte le classi che rappresentano il dominio dell'applicazione. La sua struttura è molto simile a quanto analizzato nel modello di dominio, con l'aggiunta di diverse classi ausiliarie che sono state fondamentali nella realizzazione dell'applicazione. Tra le aggiunte più importanti, segnaliamo la classe *StudentStats*, che contiene le statistiche di uno studente. Invece che calcolare ad ogni richiesta le statistiche di uno studente analizzando i suoi *QuizAttempt*, abbiamo preferito creare una classe separata in modo da ridurre le operazioni necessarie al calcolo delle statistiche (come l'accesso alla base di dati). Come illustrato nel livello *services*, queste statistiche vengono aggiornate al termine del completamento di un quiz da parte dello studente, attraverso l'uso del pattern *observer*. Lo stesso pattern è stato applicato anche per le statistiche relative a quiz e domande, memorizzate nelle classi *QuizStats* e *QuestionStats*.

Una nota importante: durante l'analisi del modello di dominio, abbiamo accennato al fatto che l'insegnante deve specificare l'elenco delle risposte giuste nelle domande aperte. Per rendere l'esperienza più flessibile, abbiamo implementato un codice che verifichi la risposta corretta tralasciando eventuali spazi vuoti iniziali/finali e utilizzando il metodo Java `equalsIgnoreCase()`.

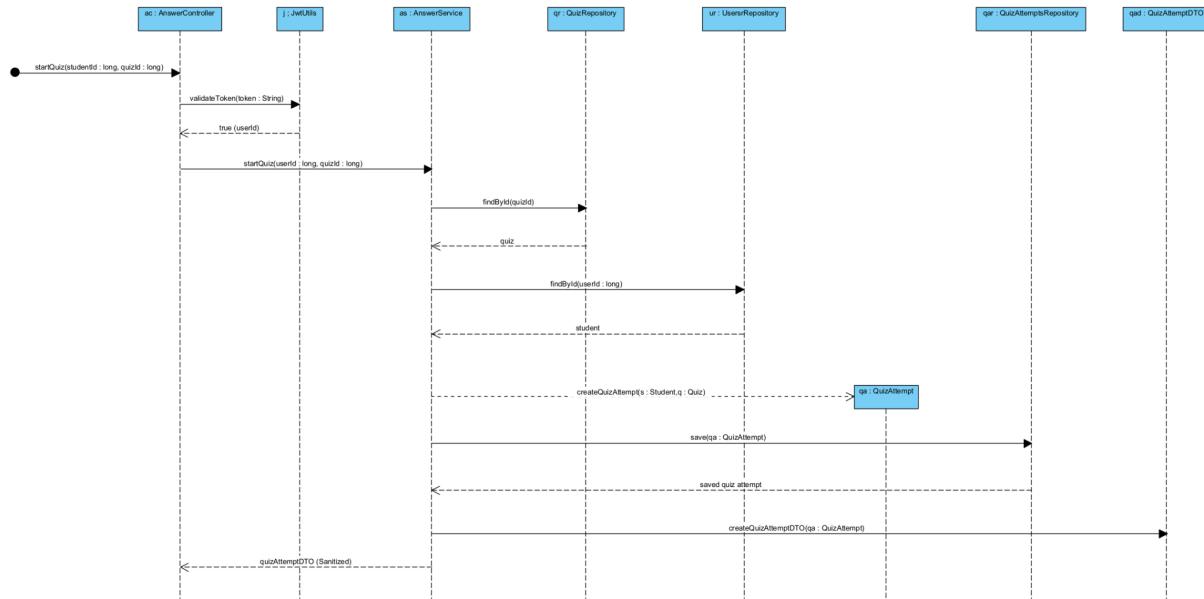
In questo package è stato implementato nuovamente il pattern *template method*: è stata definita una classe astratta *Mission*, che contiene la struttura di base di una missione, e che è stata estesa in diverse classi che implementano solo un numero ristretto di metodi necessari a differenziare le diverse missioni.

Diagrammi di sequenza (SD)

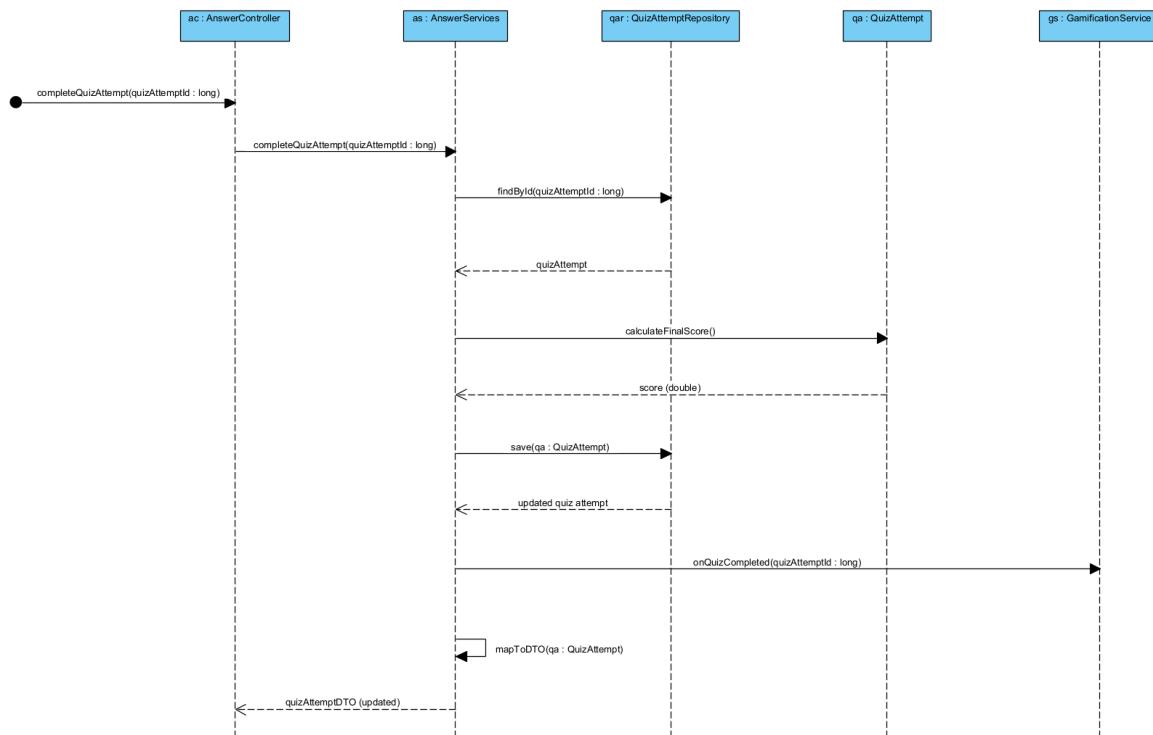
Prima iterazione

Riportiamo di seguito la prima versione dei diagrammi di sequenza, cui seguirà la versione finale della seconda iterazione.

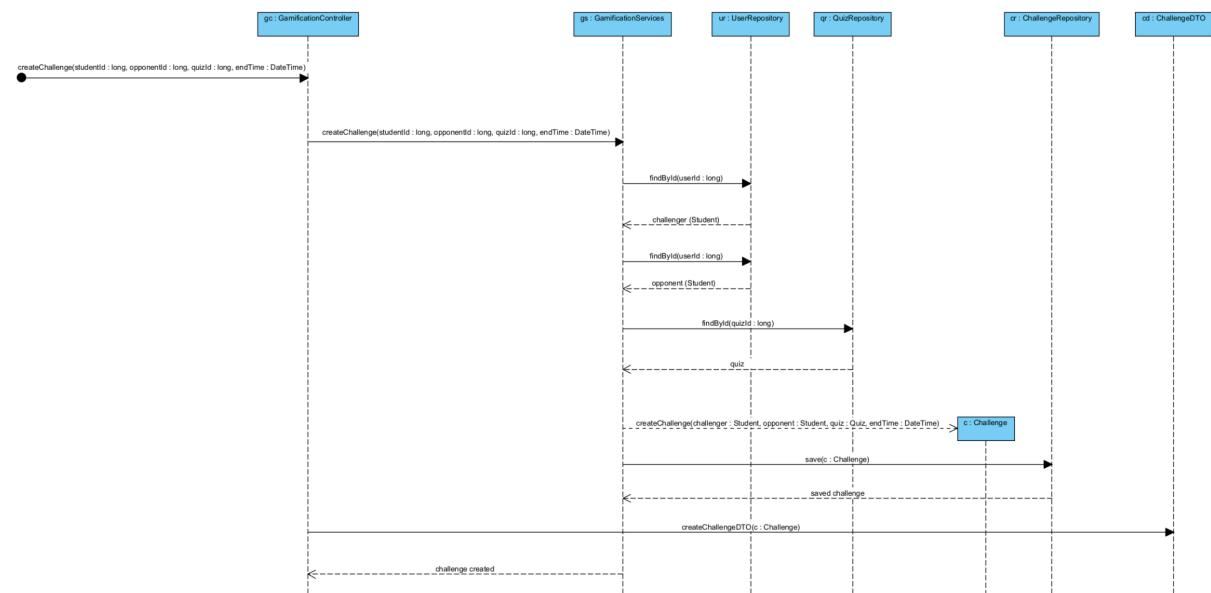
StartQuiz



CompleteQuizAttempt



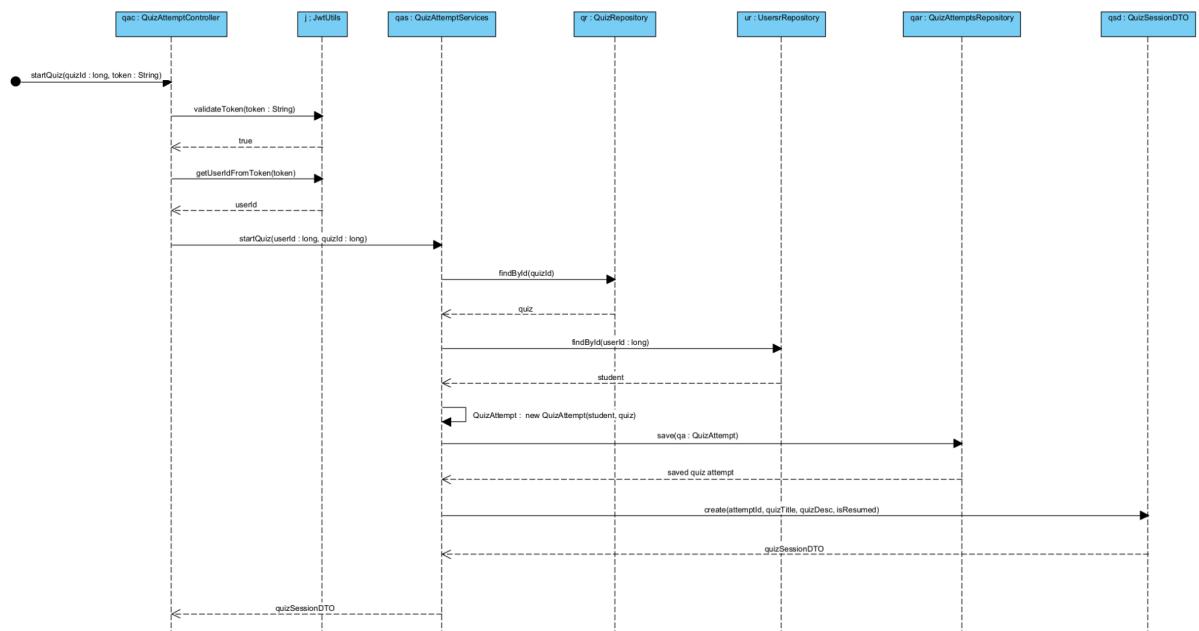
CreateChallenge



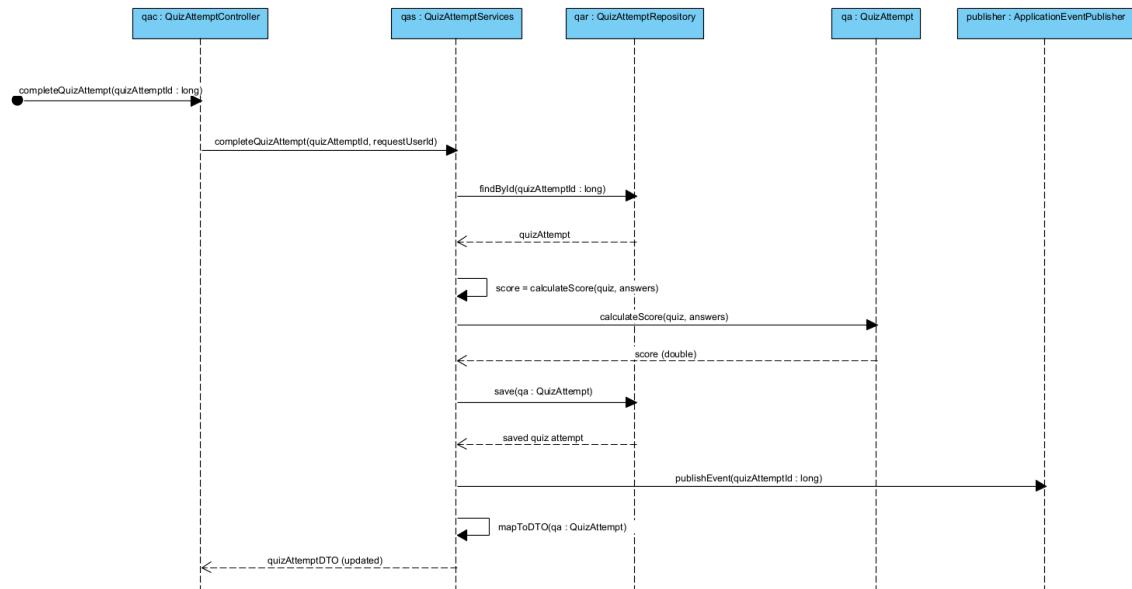
Seconda iterazione

StartQuiz: durante l'esecuzione di questa operazione, la richiesta arriva al controller che si occupa di comunicare con *JwtUtils* per validare l'autenticazione dell'utente che sta richiedendo l'operazione e recuperare l'identificatore dell'utente. Successivamente, il controller comunica con il *QuizAttemptServices* per effettuare le operazioni necessarie all'inizio della compilazione del quiz, collaborando con i repository di *User* e *QuizAttempt*. Infine, viene restituito al controller un *QuizSessionDTO* che contiene le informazioni sulla sessione appena creata.

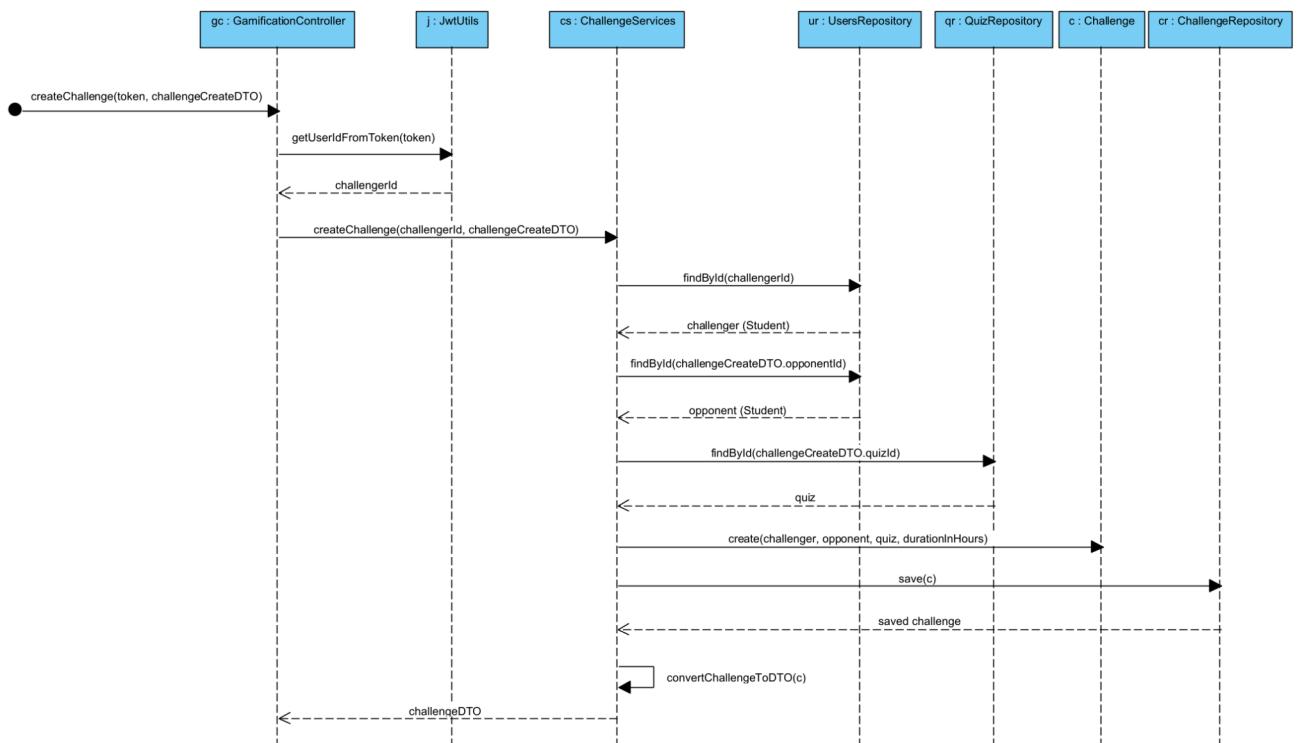
N.B. La chiamata al metodo *validateToken* della classe *JwtUtils* verrà data per scontata nei diagrammi successivi, in quanto fondamentale per garantire un accesso sicuro al sistema.



CompleteQuizAttempt: questo diagramma mostra gli effetti della chiamata al *QuizAttemptController* per terminare l'esecuzione in un quiz. Possiamo notare meglio l'applicazione del pattern *observer*: il *QuizAttemptServices*, dopo aver collaborato con il repository, richiede la pubblicazione di un evento *QuizCompletedEvent*, che verrà intercettato dai listener per aggiornare le statistiche dello studente, dei quiz e delle domande.

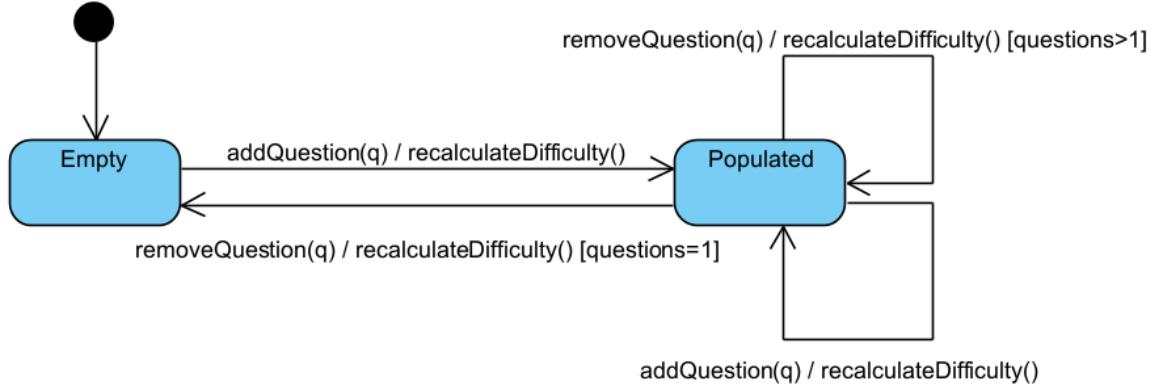


CreateChallenge: grazie a questo diagramma di sequenza è possibile comprendere meglio come avviene la creazione di una challenge da parte di uno studente. Per eseguire questo processo, il *ChallengeServices* si occuperà di comunicare con i repository di *User*, *Quiz* e *Challenge*.



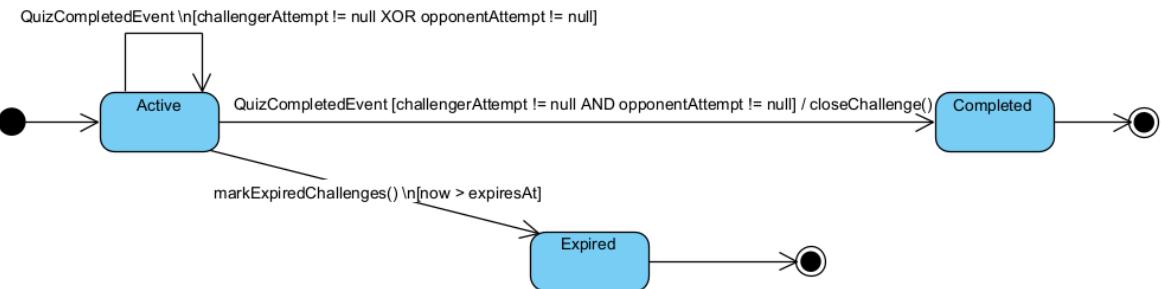
Diagrammi delle macchine a stati (STM)

Quiz: quando viene creato, un quiz si trova nello stato *empty*, cioè non contiene alcuna domanda. Nel momento in cui viene aggiunta una domanda, il suo stato diventa *populated* e rimane tale fino a quando vengono rimosse tutte le domande da cui è composto.



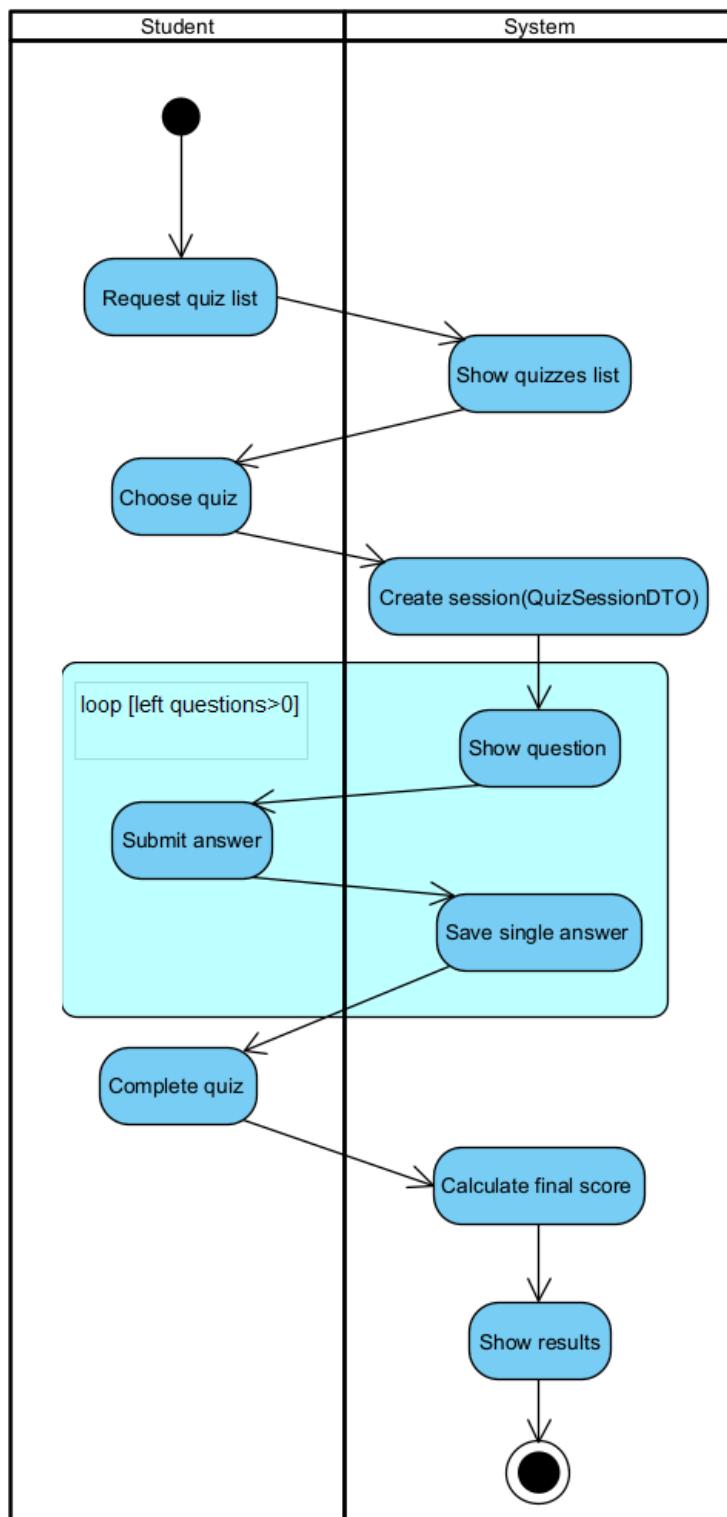
Challenge: quando viene creata, una challenge si trova nello stato *active* e ci rimane fino a quando:

- Entrambi gli studenti compilano il quiz contenuto nella challenge, e lo stato diventa *completed*
- Almeno uno dei due studenti non completa il quiz e il limite di tempo per completare la sfida scade, e lo stato diventa *expired*



Diagrammi delle attività (AD):

Inseriamo anche un diagramma delle attività che rappresenta la sequenza di operazioni svolte per la compilazione di un quiz.



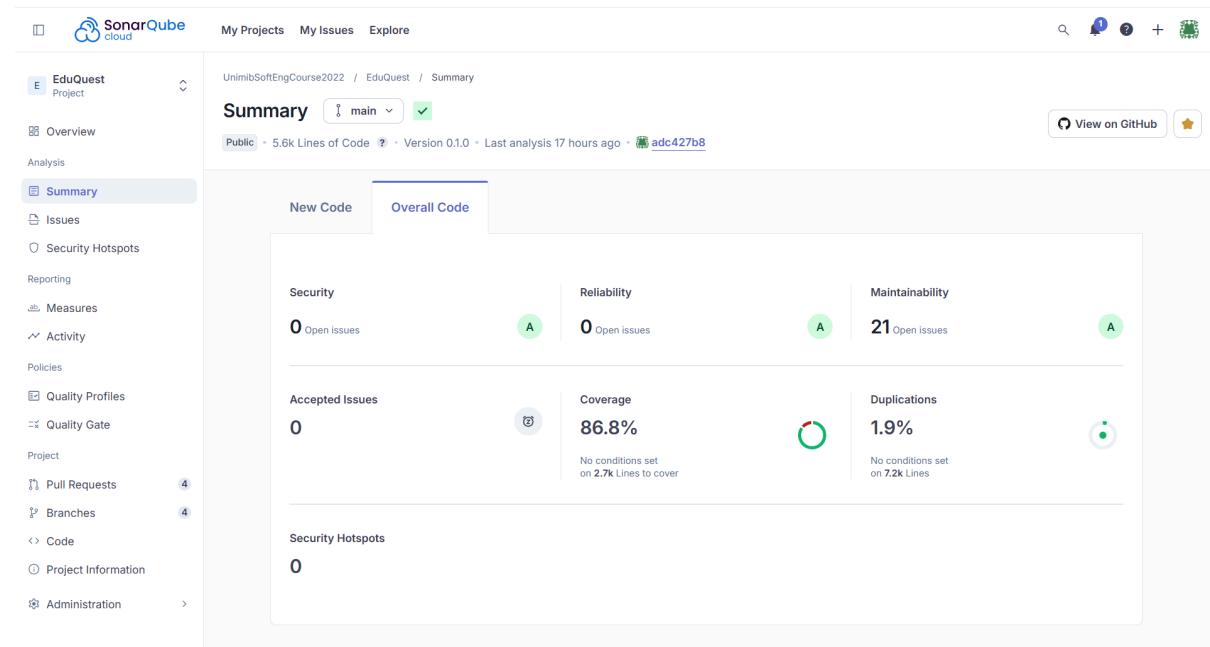
Analisi statica e qualità del codice

Per valutare la manutenibilità e la solidità dell'architettura, il progetto è stato sottoposto ad analisi statica utilizzando due strumenti complementari: SonarCloud (per l'analisi continua di bug, vulnerabilità e code smells) e SciTools Understand (per l'estrazione di metriche architettoniche e l'analisi del coupling).

Sonarcloud

Per quanto concerne l'analisi di Sonarcloud, abbiamo deciso di applicare una separazione delle responsabilità (*separation of concerns*) e di dividere quindi l'analisi di frontend e backend, per avere, oltre ad un'interfaccia più pulita, un quadro complessivo più chiaro sui problemi e sulla qualità di ciascuna parte del software.

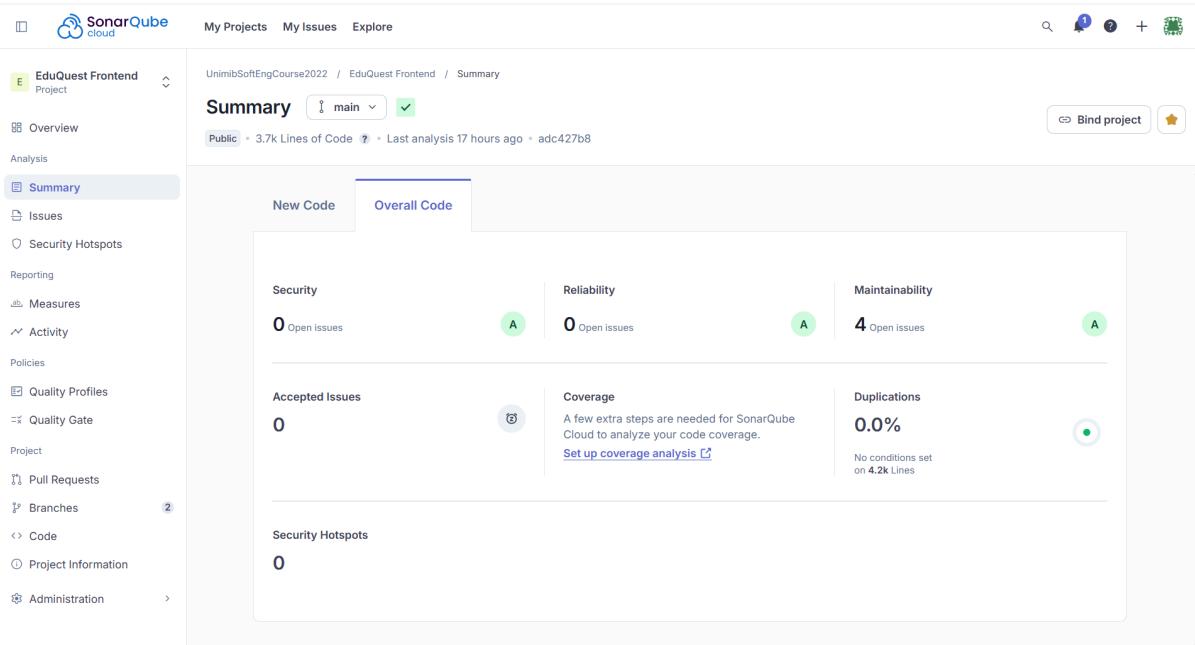
Iniziamo dall'analisi del backend:



Come si può evincere da questa schermata (la dashboard riassuntiva), il progetto ha raggiunto un rating A in tutte le sue categorie principali, quali *security*, *reliability* e *Maintainability*. Per quanto concerne la sicurezza, siamo riusciti a risolvere tutti i security hotspots e ad azzerare i problemi rilevati. Il debito tecnico residuo è anch'esso prossimo allo zero, con solamente 21 open issues, di cui la maggior parte ha una gravità minore, risolvibile con poco sforzo. Le poche (4) issues più critiche sono state, invece, analizzate dal team e classificate come accettabili e frutto di precise scelte architettoniche e che, comunque, non compromettono il massimo ranking del codice. Inoltre, l'analisi mostra che la percentuale di codice duplicato è inferiore al 2%, attestando la corretta applicazione del principio DRY (*Don't Repeat Yourself*). Infine, concentrando l'attenzione sull'affidabilità, si evidenzia un'eccellente copertura, pari ad oltre l'86%, dei test automatizzati, che garantisce stabilità e riduce il rischio di regressioni.

Durante lo sviluppo, infatti, abbiamo cercato di dare importanza alla parte di testing, che è stata sviluppata parallelamente al software.

Proseguiamo, poi, con l'analisi del frontend:



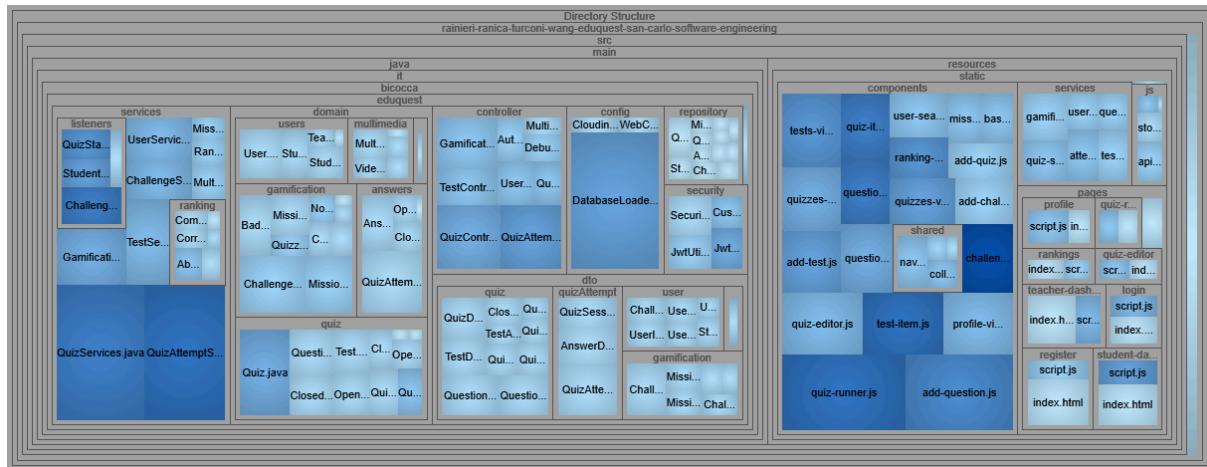
Anche questa analisi si è rivelata molto positiva, con rating A in tutte le categorie principali: il frontend non presenta alcun problema di sicurezza (i security hotspots sono pari a zero) e la parte di maintainability è anch'essa molto valida, presentando solamente 4 issues. Tre di queste sono legate ad una complessità cognitiva leggermente sopra il massimo valore accettabile, che, dopo un'attenta indagine, abbiamo deciso di accettare come debito tecnico fisiologico e controllato. Infatti, si è valutato che un'ulteriore frammentazione di tali componenti avrebbe disperso la logica visiva, riducendo paradossalmente la leggibilità e la coesione del codice. Inoltre, abbiamo eliminato completamente qualsiasi duplicazione di codice (come si può vedere nella dashboard, è 0,0%). Come si può notare, infine, dalla sezione *coverage*, per il codice JavaScript abbiamo deciso di non creare test unitari automatizzati: sebbene questo possa sembrare un debito tecnico notevole, in realtà si è trattata di una specifica scelta ingegneristica basata sul risk management e sul ritorno di investimento delle ore di sviluppo. Infatti, abbiamo adottato un approccio in cui la logica di business (esecuzione dei quiz, gamification...) è tutta incentrata nel backend, mentre il frontend funge solamente da strato di presentazione, e lo sviluppo di test UI automatizzati avrebbe sottratto tempo e risorse allo sviluppo e al testing di funzionalità centrali dell'applicazione. In ogni caso, l'assenza di test automatizzati sul client è stata compensata da sessioni di test esplorativo e test funzionali end-to-end eseguiti manualmente dal team durante le fasi di controllo.

Understand

Per valutare la solidità dell'architettura e l'eventuale presenza di antipattern strutturali, il codice sorgente è stato sottoposto ad analisi statica tramite il tool SciTools Understand. L'indagine si è concentrata sull'estrazione delle metriche di complessità ciclomatica e sull'analisi dei grafi di dipendenza (fan-in e fan-out), al fine di mappare i difetti implementativi con gli antipattern descritti in letteratura.

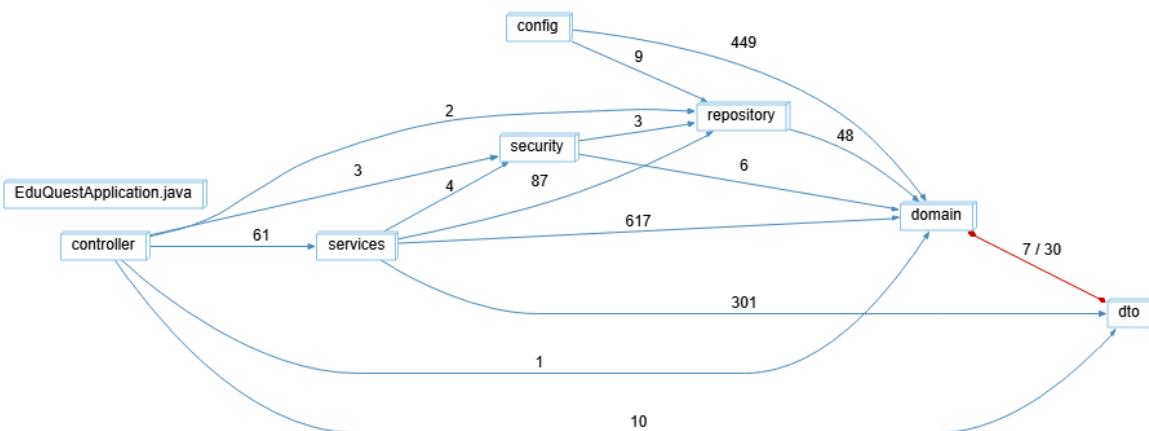
Coerentemente con la separazione architetturale discussa in precedenza, l'analisi topologica e strutturale condotta è stata focalizzata principalmente sul backend. Essendo il framework adottato (Spring Boot) basato sul paradigma Object-Oriented, si presta in modo ideale alla rilevazione di metriche di accoppiamento, coesione e design smells. Al contrario, il frontend, delegando la logica di business al server, rende meno applicabili le metriche classiche di ereditarietà e polimorfismo.

Comunque, per ottenere una visione d'insieme, abbiamo generato una *metrics treemap* dell'intero repository. Nel grafico, l'area dei rettangoli è proporzionale al volume del file (misurato in linee di codice), mentre l'intensità della colorazione blu indica il livello di complessità ciclomatica massima (tonalità più scure corrispondono a complessità maggiori).

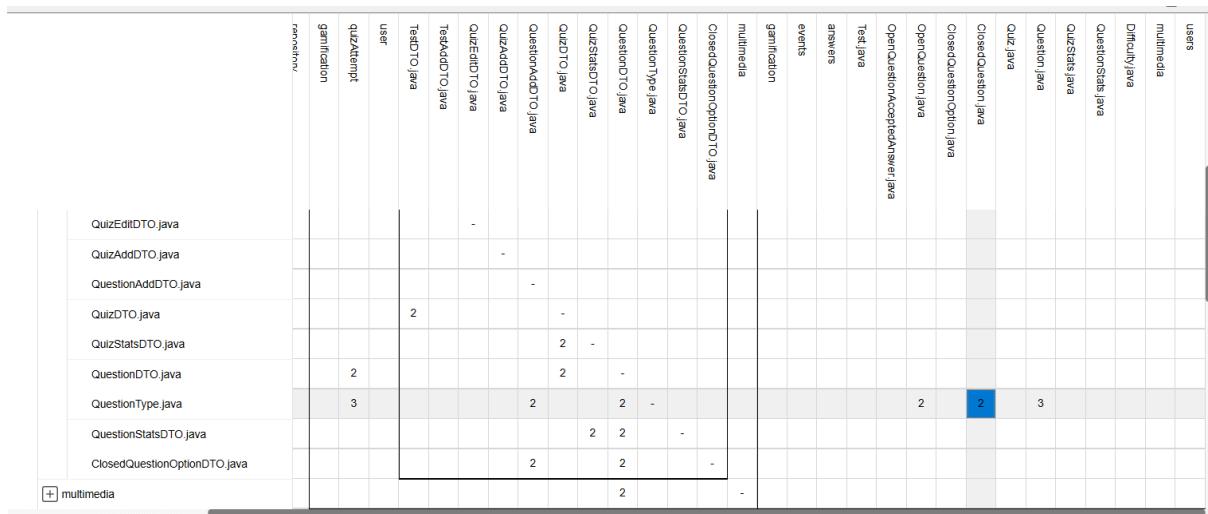


- Frontend (quadrante a destra): la mappa mostra una distribuzione fisiologicamente più frammentata, tipica delle architetture a componenti. Ciononostante, emergono moduli come challenges-viewer.js o quiz-runner.js, la cui colorazione scura conferma specularmente le segnalazioni di SonarCloud relative ai picchi di complessità cognitiva per alcune funzioni
- Backend (quadrante a sinistra): notiamo un forte ingombro per le classi DatabaseLoader.java, QuizServices.java e QuizAttemptServices.java. Questi ultimi due, in particolare, si rivelano abbastanza problematici a causa del loro colore blu abbastanza intenso, che evidenzia un forte accentramento della logica di business. Ciò verrà discusso dettagliatamente in questo paragrafo

In primo luogo, l'analisi con Understand è stata molto utile per identificare una dipendenza ciclica critica (antipattern *breakable*) tra il pacchetto di dominio e il pacchetto dei DTO, di cui altrimenti probabilmente non ci saremmo mai accorti.

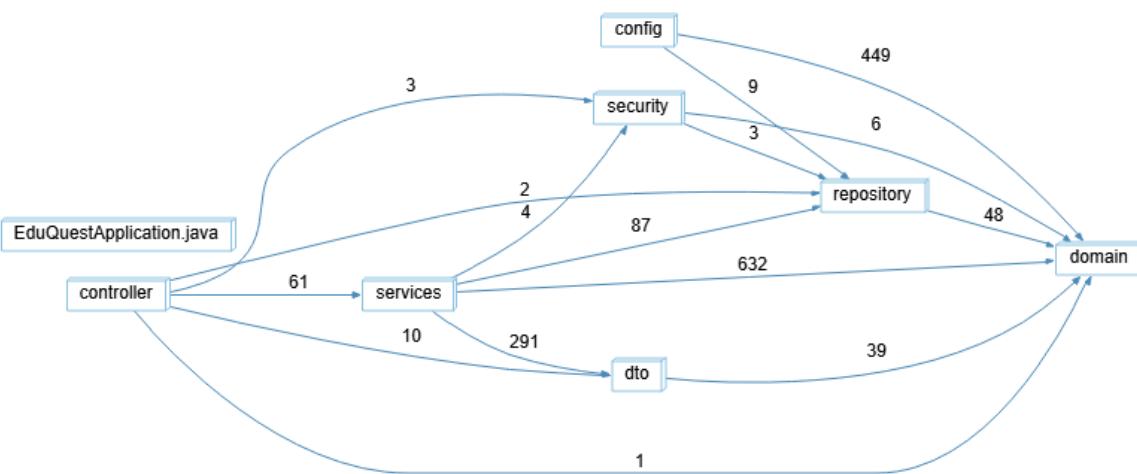


Analizzando a fondo, ci siamo resi conto che si trattava semplicemente di un posizionamento errato dell'enum `QuestionType.java`: invece di essere collocato, come avevamo fatto per gli altri `enum`, all'interno del pacchetto di dominio di riferimento (in questo caso, il sottopacchetto `domain.quiz`), era stato inserito all'interno del corrispettivo pacchetto DTO (il sottopacchetto `dto.quiz`).



Ciò comportava un'incoerenza nella logica fino a quel momento seguita, perché doveva essere il DTO a fare chiamate (compresi gli import) al dominio, non viceversa. Tuttavia, è stato prontamente risolto semplicemente spostando quella classe all'interno del pacchetto corretto `domain.quiz`.

Ciò ci ha permesso di avere una visione completa corretta rispetto a come avevamo pensato l'architettura, illustrata dal seguente grafo delle dipendenze (a livello di pacchetto):



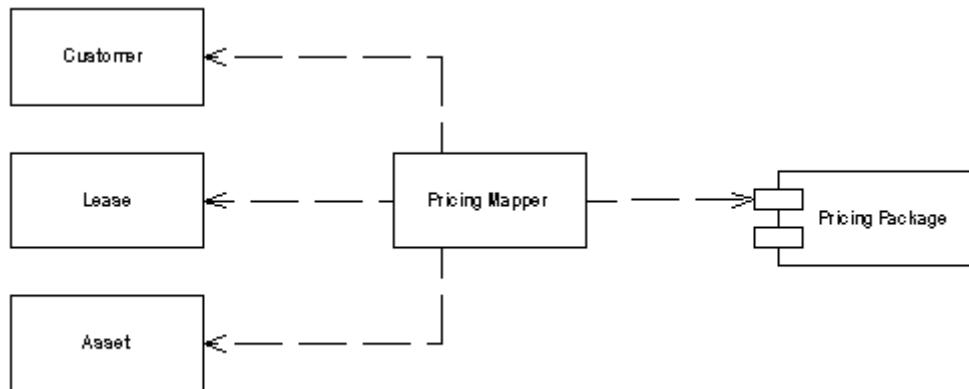
Analizzando il grafo sopraindicato, abbiamo notato la presenza di alcuni antipattern strutturali. Per identificare quali, abbiamo modellato la nostra analisi basandoci sulle slide riguardanti l'argomento riportate nel moodle del corso (per completezza, inseriamo qui il [link](#)).

- Il pacchetto `domain` si presenta come il classico *external butterfly*, caratterizzato da un altissimo fan-in e un fan-out nullo. Ciò implica che la modifica di classi al suo interno può compromettere il funzionamento globale del sistema, in quanto tutti i pacchetti puntano ad esso e lo utilizzano. Tuttavia, si tratta di una conseguenza inevitabile (e voluta) dell'architettura che abbiamo progettato, dove il modello di dominio rappresenta il fulcro

centrale e indipendente dell'applicazione. Seguendo i principi della layered architecture, è corretto che i layer esterni (controller per l'I/O, services per la logica, repository per la persistenza) dipendano fortemente dalle entità di business per poterle manipolare, mentre il dominio rimane isolato dalle logiche infrastrutturali. Pertanto, pur assumendo le sembianze di un antipattern strutturale per via dell'elevato accoppiamento in ingresso, lo classifichiamo come un rischio architetturale calcolato e fisiologico, tipico dei sistemi data-centrici

- Inoltre, segnaliamo la presenza del code smell *data class* all'interno del package *domain*: le classi fungono di fatto da contenitori di dati, e solo un numero molto ristretto di queste implementano dei metodi che si occupano della logica di business, che è stata lasciata maggiormente all'interno del package *services*. Per sfruttare al meglio la potenzialità delle classi sarebbe opportuno spostare parte della logica di business all'interno di questo package, per alleggerire il carico sui *services*
- Il pacchetto *services* si configura come un *external breakable*. Analizzando i flussi, è architetturalmente corretto che tale modulo riceva chiamate esclusivamente dai controller (61 dipendenze). Tuttavia, per evadere la logica di business, il package dei servizi presenta un fan-out molto elevato, dipendendo, in modo particolare, da *domain* (632 interazioni) e *dto* (291). Ciò è coerente con la nostra logica architetturale, tuttavia rende il sistema instabile: un cambiamento nelle dipendenze potrebbe causare la rottura o il malfunzionamento del programma, costringendo ad un continuo refactoring delle classi interessate (cosa che ci è capitato di fare durante lo sviluppo dell'applicazione)

Per quanto riguarda il secondo punto, già al termine della prima iterazione il team aveva iniziato a pensare all'utilizzo del pattern architettonale *mapper*. Come definisce Martin Fowler, un *mapper* è un oggetto il cui scopo esclusivo è stabilire una comunicazione tra due sottosistemi indipendenti, permettendo loro di scambiarsi dati pur rimanendo completamente ignari l'uno della struttura dell'altro. Eccone una visualizzazione grafica:



Nel contesto della nostra applicazione, il *mapper* potrebbe fungere da traduttore isolato tra il modello di dominio (le entità) e i DTO, centralizzando in classi helper dedicate la logica ripetitiva di conversione e copia dei campi (mappatura entità-DTO). Applicando dunque questo pattern alla nostra architettura software, potremmo risolvere il problema di external breakable dei services, in quanto:

- verrebbero eliminate le funzioni di traduzione da entità a DTO, in linea con il principio SOLID di singola responsabilità (SRP)

- inoltre, delegando l'invocazione dei mapper esclusivamente al livello dei controller, si otterrebbe un disaccoppiamento totale tra la business logic e i dati inviati al frontend, eliminando completamente la dipendenza con il pacchetto dei DTO

Sebbene, come anticipato, avessimo già delineato questa soluzione architetturale, in ottica di sviluppo iterativo si è preferito adottare un approccio pragmatico. Infatti, si è data priorità alla validazione della logica di business e alla stabilità del sistema, posticipando l'introduzione dei layer intermedi. L'analisi con Understand ha poi confermato tale fragilità che, in ottica di uno sviluppo futuro, sarebbe stata risolta tramite l'uso, appunto, dei mapper.

Testing e collaudo del sistema

Al termine della fase di sviluppo, e parallelamente all'analisi statica del codice (condotta tramite SonarCloud e Understand), il sistema EduQuest è stato sottoposto a una rigorosa fase di collaudo dinamico. L'obiettivo di questa fase è stato validare la correttezza della logica di business, l'integrità dei flussi di dati e la robustezza delle API esposte dal backend.

Le funzionalità centrali dell'applicazione, come l'autenticazione, la creazione e compilazione dei quiz e l'aggiornamento in tempo reale delle classifiche, sono state verificate combinando test di unità (eseguibili tramite la build di Maven) e collaudi end-to-end condotti direttamente sull'interfaccia utente. Questa validazione incrociata ha confermato la corretta integrazione tra il frontend e i controller RESTful.

Collaudo delle logiche temporali tramite Postman

Durante la fase di testing, è emersa la problematica del collaudo delle funzionalità dipendenti dallo scorrere del tempo. Verificare manualmente il corretto aggiornamento settimanale delle missioni o lo scadere del timer di una sfida attendendo il naturale decorso temporale era ovviamente un approccio impraticabile. Per superare questo ostacolo, abbiamo adottato una tecnica di *state manipulation* a fini di test, progettando un apposito *DebugController*. All'interno degli endpoint, abbiamo sfruttato direttamente *JdbcTemplate* per lanciare query che sovrascrivono forzatamente i dati all'interno del database, nel nostro caso retrodatandoli. In questo modo, è stato possibile simulare istantaneamente il passaggio di settimane o mesi. Poiché gli endpoint di debug non possiedono (e non devono possedere) alcun riferimento nell'interfaccia grafica utente, per il loro collaudo e utilizzo ci siamo affidati a **Postman**. Attraverso questo potente client API, abbiamo configurato e salvato le richieste HTTP POST dirette a questi endpoint. Questo ci ha permesso, tramite un semplice click su Postman, di retrodatare lo stato della banca dati, consentendoci di verificare istantaneamente la corretta reazione del sistema. In particolare, siamo riusciti a verificare nel giro di pochi secondi la regolare scadenza delle sfide e la corretta rigenerazione delle missioni settimanali, implementate tramite la tecnica di *lazy initialization*.

Attualmente, gli endpoint utilizzati per il collaudo dinamico (quelli del *DebugController*) coesistono con la logica di business. Sebbene non siano esposti nell'interfaccia grafica, la loro presenza in una release finale può rappresentare un potenziale rischio di sicurezza. In uno sviluppo futuro, sarebbe opportuno separare i contesti di esecuzione tramite i **Profile** del framework *Spring* (definendo gli ambienti *dev* e *prod*). Grazie all'uso di annotazioni dedicate come `@Profile("dev")`, sarebbe possibile istanziare i controller di test e le API di manipolazione dello stato esclusivamente nell'ambiente *"developer"*. Invece, avviando l'applicazione nel profilo *"user"*, tali backdoor verrebbero completamente escluse dal processo di compilazione e caricamento, assicurando così la sicurezza del database.

Conclusione

Lo sviluppo dell'applicazione Eduquest è stato un momento di maturazione, affinamento delle nostre competenze e coronazione del nostro cammino universitario. Siamo riusciti a completare gli obiettivi prefissati all'inizio, in particolare:

- Abbiamo realizzato un'interfaccia utente semplice ma funzionale, ideale per un primo prototipo di applicazione, come quello realizzato
- Siamo riusciti a progettare e realizzare un backend solido e che garantisce le funzionalità richieste, anche se, come osservato durante l'analisi del codice, migliorabile
- Abbiamo imparato ad applicare i concetti chiave del *software engineering* e a pensare da ingegneri, sia nella fase di analisi e progettazione che nella parte di implementazione, applicando i vari pattern (DP, architetturali) e principi (SOLID, Martin) e sfruttando gli strumenti di analisi statica del codice (tra i vari, SonarCloud e Understand)
- Ci siamo resi conto dell'importanza dell'analisi e della progettazione del software, grazie alla quale il lavoro è stato organizzato in modo ordinato e rigoroso: ciò ha semplificato molto la scrittura del codice, nonostante abbiano dovuto adattare quest'ultimo a nuove esigenze emerse durante l'implementazione vera e propria
- Inoltre, abbiamo imparato a lavorare in team in modo agile, dividendo il lavoro in due iterazioni

Come già accennato, l'applicazione è ampiamente estensibile, sia dal punto di vista delle funzionalità che dal punto di vista architettonico. Ad esempio, abbiamo deciso di non implementare le sfide in tempo reale tra due studenti: essendo una funzionalità decisamente avanzata, che coinvolge l'utilizzo di tecnologie come le *websocket*, abbiamo preferito, anche per limiti temporali, evitare la loro implementazione, favorendo uno studio più approfondito delle altre funzionalità scelte. Inoltre, avevamo anche pensato di implementare una sezione dedicata per permettere ai docenti di caricare materiale multimediale, come ad esempio lezioni videoregistrate o simili, ma ci avrebbe tolto una considerevole quantità di risorse, considerato anche che non si trattava di una funzionalità richiesta. Tuttavia, avendo già integrato e configurato l'infrastruttura di cloud storage tramite Cloudinary, la futura implementazione di questa sezione risulterà un'estensione naturale e a bassissimo impatto per il team di sviluppo.

In conclusione, siamo molto contenti di aver avuto la possibilità di lavorare allo sviluppo di Eduquest: questo progetto ci ha offerto l'opportunità di metterci in gioco e di tradurre in un'applicazione concreta concetti fino ad ora affrontati solo a livello teorico. Non sono mancati, tuttavia, momenti di difficoltà e divergenze tra di noi, ma abbiamo imparato a gestirli trovando soluzioni di compromesso efficaci. Questa è probabilmente l'insegnamento più importante, insieme al lato tecnico, che ci accompagnerà nel nostro futuro universitario e lavorativo.

Ringraziamo il lettore per l'attenzione e il tempo dedicato alla lettura di questa relazione.

Luca Rainieri, Andrea Ranica, Mattia Turconi, Chenhao Wang