UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

REPORT PROGETTO - SOFTWARE DEPENDABILITY

Pokèdex-Social-2.0

https://github.com/FrancescoPinto02/Pokedex-Social-2.0

DOCENTE

Prof. Dario Di Nucci

Università degli Studi di Salerno

AUTORE

Francesco Alessandro Pinto

0522501981

Anno Accademico 2025-2026

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Scopo del Documento

Lo scopo di questo report è descrivere il lavoro svolto nell'ambito del progetto per il corso di Software Dependability, evidenziando come le tecniche e gli strumenti presentati durante il corso siano stati integrati nel ciclo di sviluppo di una Web Application.

La web application selezionata per il progetto è **Pokedex-Social-2.0**, una piattaforma sviluppata dal sottoscritto per esercitarsi con l'utilizzo di tecnologie moderne. L'applicazione permette agli utenti di consultare l'intero Pokédex, visualizzando ogni Pokémon e i relativi dettagli, e integra funzionalità social quali registrazione, autenticazione e creazione di team personalizzati da condividere con la community. Una componente particolarmente interessante è la generazione automatica di team ottimizzati mediante un algoritmo genetico, che introduce un ulteriore livello di complessità applicativa.

1.2 Struttura del Progetto

L'intero progetto Pokedex-Social-2.0 è organizzato come una monorepo pubblicata su GitHub (link repo). La struttura complessiva è suddivisa in tre componenti principali:

- **Backend**: il backend è sviluppato in Java utilizzando Spring Boot 3.5.5 e Java 17, con sistema di build Maven. Esso fornisce tutti i servizi necessari alla logica applicativa, gestisce la persistenza dei dati tramite il framework ORM Hibernate e espone una serie di endpoint REST che possono essere invocati dal frontend mediante chiamate HTTP.

- **Frontend**: il frontend è realizzato con React 19, Typescript e Vite 7.2.4. La sua funzione principale è gestire la logica di presentazione, richiedendo dati al backend in formato JSON e visualizzandoli dinamicamente nelle interfacce utente. Al momento alcune funzionalità implementate nel backend, come la creazione e condivisione dei team, non sono ancora completamente disponibili nel frontend, ma la struttura è predisposta per integrarle progressivamente.

- **Database**: il sistema utilizza un database PostgreSQL, orchestrato tramite Docker fin dalle prime fasi di sviluppo. All'avvio del container il database viene popolato automaticamente mediante file CSV contenenti i dati necessari per il funzionamento dell'applicazione.

CAPITOLO 2

Strumenti e Tecniche Utilizzate

2.1 JML e OpenJML

In fase di progettazione delle specifiche formali si è inizialmente valutata la possibilità di applicare **JML** alle classi **service** del backend, in particolare a quelle coinvolte nella gestione dell'autenticazione e del Pokédex (eg. ricerca filtrata, paginazione). Tale scelta è stata tuttavia rapidamente abbandonata, poiché queste componenti dipendono fortemente dall'infrastruttura di Spring Boot: l'uso estensivo di annotazioni, dependency injection, proxy dinamici e l'interazione continua con il contesto applicativo rende i metodi poco adatti alla verifica automatica tramite OpenJML.

Per tali ragioni l'attenzione è stata rivolta alle componenti dell'algoritmo genetico (anche esso molto importante per il sistema), progettate in modo modulare e prive di dipendenze da framework esterni. Le specifiche sono state applicate direttamente alle classi astratte che definiscono la struttura e il comportamento di base del modulo genetico, così che tutte le implementazioni concrete ereditino automaticamente i contratti formali associati, garantendo uniformità e coerenza semantica. In questo contesto sono state quindi fornite specifiche formali per alcune classi del package optimizer.ga (link), in particolare per le seguenti classi:

- **Individual.java**: classe astratta che definisce il comportamento base degli individui dell'algoritmo genetico.
- **Population.java**: classe astratta che rappresenta una popolazione di individui ed estende **HashSet<Individual>**, includendo, tra le altre, una funzione per il calcolo della fitness media.
- **GeneticOperator.java**: classe astratta che rappresenta la base di tutti gli operatori genetici e include il metodo **apply**, responsabile dell'applicazione dell'operatore a una popolazione.
- **CrossoverOperator.java**: classe astratta che rappresenta la base degli operatori di crossover, introducendo inoltre la classe interna **Pairing** e la funzione **makeRandomPairings** per l'accoppiamento degli individui.

L'obiettivo iniziale era quello di estendere progressivamente le specifiche fino a coprire anche la classe orchestratrice dell'algoritmo genetico, inclusa la componente dedicata alla funzione di fitness. Tuttavia, l'utilizzo congiunto di JML e OpenJML ha evidenziato alcune limitazioni che hanno impedito di completare una specifica formale coerente per tutte le parti del sistema. In particolare:

1. Molte classi dell'algoritmo utilizzano funzioni della libreria standard Java prive di specifiche formali fornite da OpenJML. In alcuni casi, come **Collections.max()** o **Collections.min()**, il problema è stato aggirato tramite implementazioni custom equivalenti e più facilmente specificabili. Per funzioni più complesse, come **Collections.shuffle()**, la definizione di una specifica formale risulta invece estremamente difficile, sia per la natura non deterministica del metodo sia per la complessità delle assunzioni necessarie.
2. Alcune classi sovrascrivono metodi ereditati, come **clone()**. Pur trattandosi di implementazioni semplici (generate automaticamente dall'IDE), la relativa specifica formale non è stata realizzabile, poiché la versione standard del metodo presenta specifiche incomplete o errate nei file JML della libreria. OpenJML non è quindi in grado di verificarne correttamente il comportamento. Per evitare che tali metodi interferissero con la restante analisi, si è fatto ricorso alle annotazioni **skipesc** e **skiprac**, che permettono di escludere il metodo dalla verifica.

3. Una criticità particolarmente rilevante riguarda l'impossibilità di specificare clausole **assignable** riferite a campi privati degli elementi contenuti in strutture dati come **HashSet**. La classe **Population** estende **HashSet<Individual>** e ogni **Individual** contiene un campo **fitness**. Nella funzione **evaluate(Population)** della classe **FitnessFunction** sarebbe stato necessario specificare che le uniche modifiche ammesse riguardavano esattamente tutti i campi **fitness** degli individui presenti nella Popolazione. OpenJML, tuttavia, non offre un meccanismo per esprimere assegnabilità vincolata ai campi interni di una collection, rendendo impossibile definire una specifica completa e accurata per questo metodo.

La verifica delle specifiche è stata condotta utilizzando entrambe le modalità offerte da OpenJML, ovvero Extended Static Checking (ESC) e Runtime Assertion Checking (RAC). La Figura 2.1 riporta il riepilogo dell'esecuzione di ESC sulle classi annotate. Si osserva che alcune classi non risultano verificate: ciò è dovuto all'utilizzo delle annotazioni **skipesc** e **skiprac** su metodi come **clone()**.

```
Summary:
  Valid:        15
  Invalid:      0
  Infeasible:   0
  Timeout:      0
  Error:        0
  Skipped:      5
 TOTAL METHODS: 20
 Classes:       3 proved of 5
 Model Classes: 0
 Model methods: 0 proved of 0
 DURATION:         42.3 secs

Note: ./src/main/java/com/pokedexsocial/backend/optimizer/ga/population/Population.java uses unchecked or unsafe operati
ons.
Note: Recompile with -Xlint:unchecked for details.
1 warning
```

Figura 2.1: ESC: Execution Summary

2.2 Testing e Analisi della Copertura

2.2.1 Situazione iniziale

All'avvio del progetto il repository non conteneva alcun test automatizzato. L'assenza di una suite di test rendeva complesso verificare la correttezza delle componenti critiche del backend, limitava la possibilità di introdurre strumenti di analisi della copertura e impediva di utilizzare tecniche di mutation testing. Si è pertanto deciso

di procedere con la definizione di un insieme esteso di test di unità, con l'obiettivo di coprire in modo sistematico la logica applicativa del sistema.

2.2.2 Generazione Unit Test

Per accelerare lo sviluppo della suite di test e garantire una copertura ampia ed eterogenea, si è scelto di supportare la generazione dei casi di test mediante un LLM, in particolare GPT 5.1. A tal fine è stato realizzato un prompt dedicato, ottenuto attraverso un processo iterativo di **prompt engineering**. La base iniziale era un prompt progettato per generare test per applicazioni web sviluppate in TypeScript; tale prompt è stato progressivamente raffinato, anche con l'assistenza dello stesso LLM, fino a ottenere una versione coerente con le convenzioni e i requisiti di un progetto Java basato su Spring Boot. Il prompt finale, fornito al modello e riportato nel repository del progetto (link), è strutturato in più sezioni che definiscono:

- le linee guida preliminari per l'analisi del codice;
- i vincoli architetturali relativi all'uso di JUnit 5 e Mockito;
- la struttura esatta delle classi di test e le convenzioni di naming;
- le regole di progettazione dei test;
- un formato di output standardizzato che il modello deve rispettare.

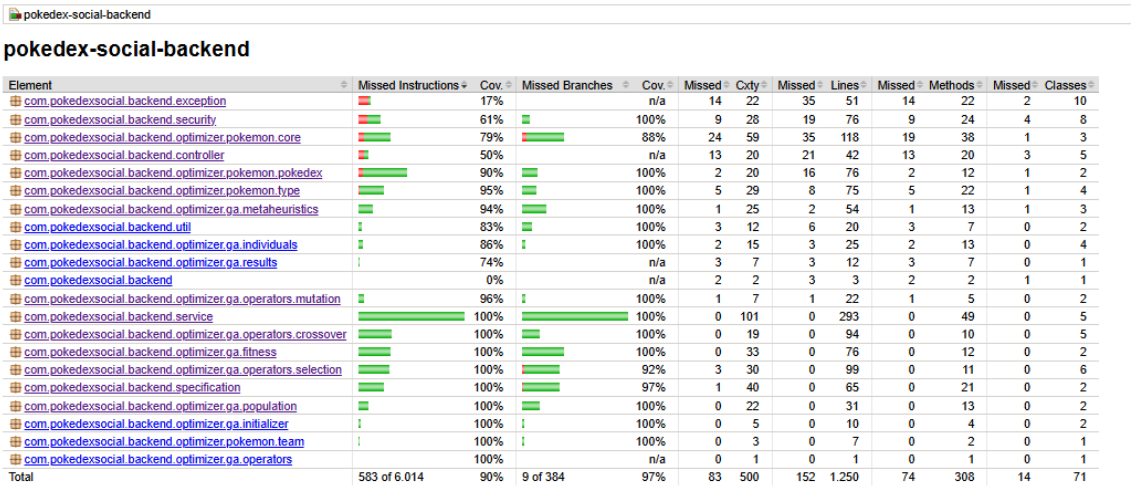
L'approccio seguito per la generazione dei test è di tipo **white-box**: il modello è stato guidato affinché generasse test in grado di esercitare tutti i branch condizionali delle componenti del backend. In particolare, è stata data maggiore priorità a le classi responsabili dell'autenticazione e della validazione degli utenti, i service che implementano la logica di business dell'applicazione e le classi che compongono l'algoritmo genetico.

2.2.3 Analisi Copertura

Per analizzare la copertura dei test è stato utilizzato **JaCoCo**, integrato nel progetto tramite plugin Maven. Lo strumento è stato impiegato in modo progressivo durante

la fase di generazione dei test: dopo la produzione dei primi casi di test mediante LLM, la copertura è stata analizzata per individuare eventuali branch non esercitati e valutare la necessità di integrare ulteriori test all'interno della suite. La Figura 2.2 mostra il risultato dell'ultima esecuzione completa dei test. Come si può osservare, la copertura dei branch raggiunge il 97% e i pochi branch non coperti rientrano principalmente in tre categorie:

- metodi autogenerati dall'IDE, come **equals** e **hashCode**, non rilevanti per la logica applicativa;
- rami complessi da esercitare negli operatori di selezione dell'algoritmo genetico, che richiederebbero mock articolati o configurazioni non naturali;
- branch di fatto non raggiungibili (unfeasible) in condizioni operative normali.



pokedex-social-backend

pokedex-social-backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.pokedexsocial.backend.exception		17%		n/a	14	22	35	51	14	22	2	10
com.pokedexsocial.backend.security		61%		100%	9	28	19	76	9	24	4	8
com.pokedexsocial.backend.optimizer.pokemon.core		79%		88%	24	59	35	118	19	38	1	3
com.pokedexsocial.backend.controller		50%		n/a	13	20	21	42	13	20	3	5
com.pokedexsocial.backend.optimizer.pokemon.pokedex		90%		100%	2	20	16	76	2	12	1	2
com.pokedexsocial.backend.optimizer.pokemon.type		95%		100%	5	29	8	75	5	22	1	4
com.pokedexsocial.backend.optimizer.ga.metaheuristics		94%		100%	1	25	2	54	1	13	1	3
com.pokedexsocial.backend.util		83%		100%	3	12	6	20	3	7	0	2
com.pokedexsocial.backend.optimizer.ga.individuals		86%		100%	2	15	3	25	2	13	0	4
com.pokedexsocial.backend.optimizer.ga.results		74%		n/a	3	7	3	12	3	7	0	1
com.pokedexsocial.backend		0%		n/a	2	2	3	3	2	2	1	1
com.pokedexsocial.backend.optimizer.ga.operators.mutation		96%		100%	1	7	1	22	1	5	0	2
com.pokedexsocial.backend.service		100%		100%	0	101	0	293	0	49	0	5
com.pokedexsocial.backend.optimizer.ga.operators.crossover		100%		100%	0	19	0	94	0	10	0	5
com.pokedexsocial.backend.optimizer.ga.fitness		100%		100%	0	33	0	76	0	12	0	2
com.pokedexsocial.backend.optimizer.ga.operators.selection		100%		92%	3	30	0	99	0	11	0	6
com.pokedexsocial.backend.specification		100%		97%	1	40	0	65	0	21	0	2
com.pokedexsocial.backend.optimizer.ga.population		100%		100%	0	22	0	31	0	13	0	2
com.pokedexsocial.backend.optimizer.ga.initializer		100%		100%	0	5	0	10	0	4	0	2
com.pokedexsocial.backend.optimizer.pokemon.team		100%		100%	0	3	0	7	0	2	0	1
com.pokedexsocial.backend.optimizer.ga.operators		100%		n/a	0	1	0	1	0	1	0	1
Total	583 of 6.014	90%	9 of 384	97%	83	500	152	1.250	74	308	14	71

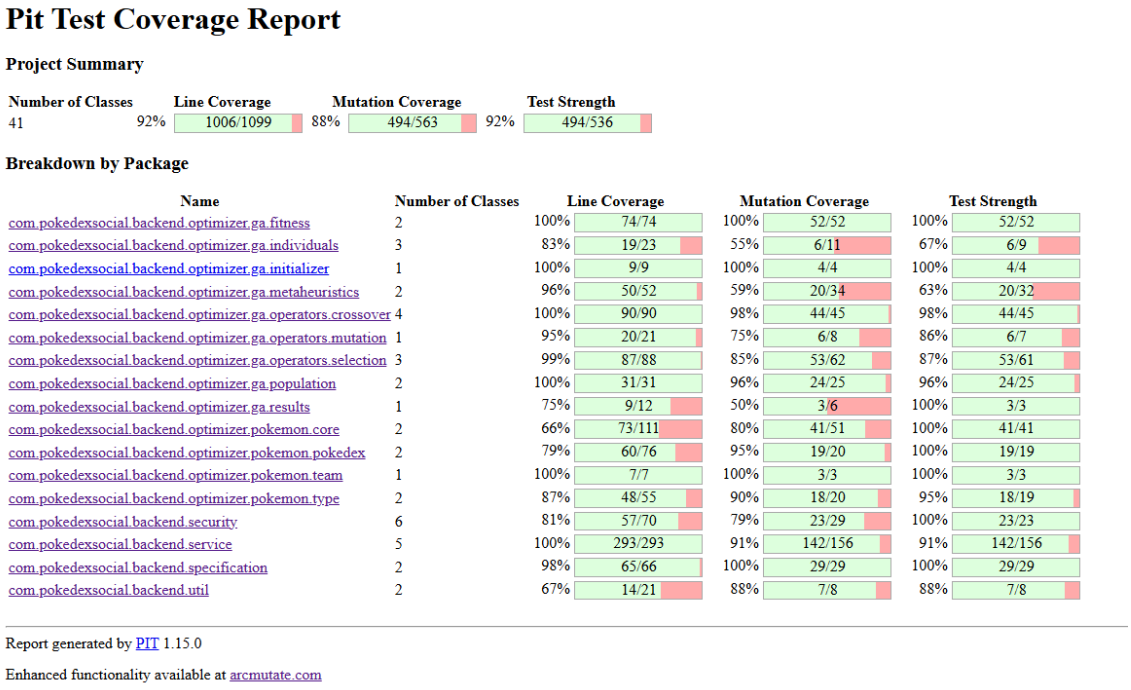
Figura 2.2: JaCoCo: Coverage Report

2.2.4 Mutation Testing

Per valutare in modo più approfondito la qualità della suite di test è stato utilizzato PiTest, configurato con il set predefinito di operatori di mutazione. Come mostrato in Figura 2.3, la prima esecuzione di PiTest ha evidenziato una test strength pari al 92%. Questo risultato conferma che la strategia di generazione dei test, si è dimostrata già nella fase iniziale particolarmente efficace. Tuttavia l'analisi ha anche messo in luce

la presenza di un numero significativo di mutanti sopravvissuti in alcuni package rilevanti, in particolare:

- **metheuristic**, contenente gli orchestratori dell'algoritmo genetico;
- **selection**, che include gli operatori di selezione;
- **service**, responsabile della logica applicativa principale.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
41	92%	1006/1099	88%	494/563	92%	494/536

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.pokedexsocial.backend.optimizer.ga.fitness	2	100%	74/74	100%	52/52	100%	52/52
com.pokedexsocial.backend.optimizer.ga.individuals	3	83%	19/23	55%	6/11	67%	6/9
com.pokedexsocial.backend.optimizer.ga.initializer	1	100%	9/9	100%	4/4	100%	4/4
com.pokedexsocial.backend.optimizer.ga.metaheuristics	2	96%	50/52	59%	20/34	63%	20/32
com.pokedexsocial.backend.optimizer.ga.operators.crossover	4	100%	90/90	98%	44/45	98%	44/45
com.pokedexsocial.backend.optimizer.ga.operators.mutation	1	95%	20/21	75%	6/8	86%	6/7
com.pokedexsocial.backend.optimizer.ga.operators.selection	3	99%	87/88	85%	53/62	87%	53/61
com.pokedexsocial.backend.optimizer.ga.population	2	100%	31/31	96%	24/25	96%	24/25
com.pokedexsocial.backend.optimizer.ga.results	1	75%	9/12	50%	3/6	100%	3/3
com.pokedexsocial.backend.optimizer.pokemon.core	2	66%	73/111	80%	41/51	100%	41/41
com.pokedexsocial.backend.optimizer.pokemon.pokedex	2	79%	60/76	95%	19/20	100%	19/19
com.pokedexsocial.backend.optimizer.pokemon.team	1	100%	7/7	100%	3/3	100%	3/3
com.pokedexsocial.backend.optimizer.pokemon.type	2	87%	48/55	90%	18/20	95%	18/19
com.pokedexsocial.backend.security	6	81%	57/70	79%	23/29	100%	23/23
com.pokedexsocial.backend.service	5	100%	293/293	91%	142/156	91%	142/156
com.pokedexsocial.backend.specification	2	98%	65/66	100%	29/29	100%	29/29
com.pokedexsocial.backend.util	2	67%	14/21	88%	7/8	88%	7/8

Report generated by PIT 1.15.0

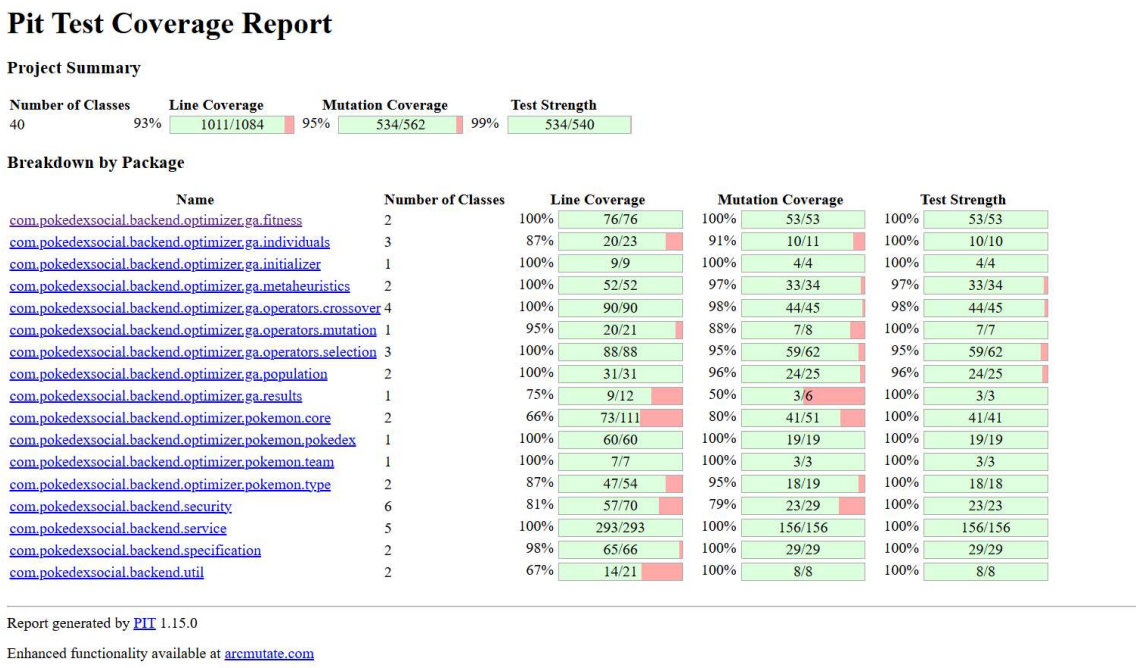
Enhanced functionality available at arcmutate.com

Figura 2.3: PiT: Report Iniziale

Questi risultati hanno guidato una fase di rafforzamento mirato della suite di test, concentrata proprio sulle componenti in cui era stata rilevata una minore efficacia. L'esecuzione finale di PiTest, riportata in Figura 2.4, mostra una test strength del 99%. I pochi mutanti residui appartengono principalmente a metodi complessi dell'algoritmo genetico, caratterizzati da logiche non deterministiche o strutture dati articolate, per cui risulta estremamente difficile costruire casi di test pienamente discriminanti.

2.3 Analisi delle prestazioni con JMH

Oltre alla valutazione della correttezza e della robustezza della test suite, è stato configurato Codecov per monitorare l'eventuale presenza di **flaky tests** e per identificare



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
40	93%	1011/1084	95%	534/562	99%	534/540

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.pokedexsocial.backend.optimizer.ga.fitness	2	100%	76/76	100%	53/53	100%	53/53
com.pokedexsocial.backend.optimizer.ga.individuals	3	87%	20/23	91%	10/11	100%	10/10
com.pokedexsocial.backend.optimizer.ga.initializer	1	100%	9/9	100%	4/4	100%	4/4
com.pokedexsocial.backend.optimizer.ga.metaheuristics	2	100%	52/52	97%	33/34	97%	33/34
com.pokedexsocial.backend.optimizer.ga.operators.crossover	4	100%	90/90	98%	44/45	98%	44/45
com.pokedexsocial.backend.optimizer.ga.operators.mutation	1	95%	20/21	88%	7/8	100%	7/7
com.pokedexsocial.backend.optimizer.ga.operators.selection	3	100%	88/88	95%	59/62	95%	59/62
com.pokedexsocial.backend.optimizer.ga.population	2	100%	31/31	96%	24/25	96%	24/25
com.pokedexsocial.backend.optimizer.ga.results	1	75%	9/12	50%	3/6	100%	3/3
com.pokedexsocial.backend.optimizer.pokemon.core	2	66%	73/111	80%	41/51	100%	41/41
com.pokedexsocial.backend.optimizer.pokemon.pokedex	1	100%	60/60	100%	19/19	100%	19/19
com.pokedexsocial.backend.optimizer.pokemon.team	1	100%	7/7	100%	3/3	100%	3/3
com.pokedexsocial.backend.optimizer.pokemon.type	2	87%	47/54	95%	18/19	100%	18/18
com.pokedexsocial.backend.security	6	81%	57/70	79%	23/29	100%	23/23
com.pokedexsocial.backend.service	5	100%	293/293	100%	156/156	100%	156/156
com.pokedexsocial.backend.specification	2	98%	65/66	100%	29/29	100%	29/29
com.pokedexsocial.backend.util	2	67%	14/21	100%	8/8	100%	8/8

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com

Figura 2.4: PiT: Report Finale

i test caratterizzati da tempi di esecuzione anomali. Come mostrato in Figura 2.5, tale analisi ha permesso di individuare potenziali colli di bottiglia che, qualora significativi, avrebbero potuto beneficiare di un'analisi prestazionale approfondita mediante micro-benchmarking con JMH. Dalla revisione dei test più lenti è emerso che la maggior parte di essi apparteneva a categorie intrinsecamente poco adatte all'ottimizzazione tramite JMH. In particolare:

- i test basati su **MockMvc**, rallentati dall'inizializzazione del contesto Spring e dalla simulazione del flusso HTTP;
- i test che fanno uso intensivo della **reflection**, la cui latenza è dovuta alla natura dinamica delle operazioni svolte;
- i test relativi all'**autenticazione**, basata su generazione e validazione di token JWT e su operazioni crittografiche non ottimizzabili senza compromettere la sicurezza. Essendo una componente standardizzata e non sostituibile, non rappresenta un candidato utile per analisi con JMH.

Alla luce di queste osservazioni, l'attenzione si è concentrata sulle parti più computazionalmente rilevanti del sistema: le componenti dell'algoritmo genetico. Tale

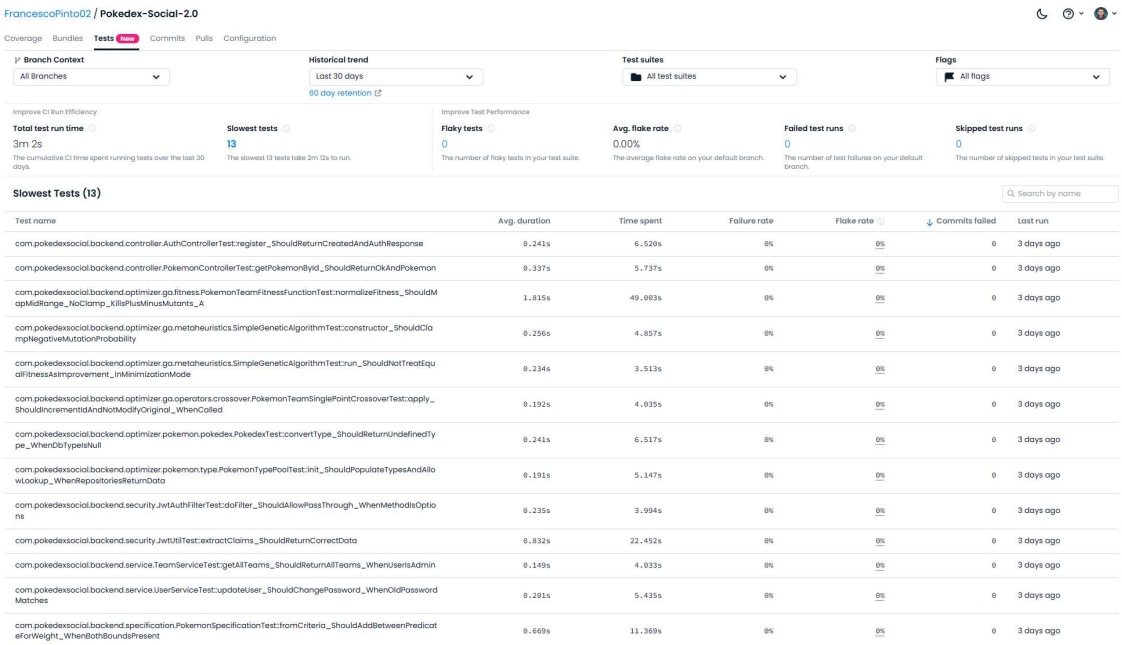


Figura 2.5: Codecov: Slowest Tests

algoritmo esegue ripetutamente operatori di selezione e di crossover, oltre che valutazione della fitness, con impatti diretti sia sui tempi di esecuzione sia sulla qualità delle soluzioni prodotte. Per questo motivo è stato utilizzato JMH al fine di confrontare le principali configurazioni dell'algoritmo e valutare quale rappresentasse il miglior compromesso tra efficienza e qualità dei risultati. È importante sottolineare che il benchmark ha misurato le prestazioni dell'**intero algoritmo genetico** e non dei singoli operatori isolati, così da catturare il comportamento reale del sistema nelle condizioni effettive di utilizzo.

I risultati in Tabella 2.1 mostrano differenze insignificanti tra gli operatori di crossover, mentre la strategia di selezione incide in modo più significativo sia sulle prestazioni sia sulla qualità delle soluzioni. Le tecniche **Roulette Wheel** e **k-Tournament** risultano le più efficienti, con tempi compresi tra 7,5 e 8 ms per operazione e un numero ridotto di iterazioni. La selezione **Rank**, pur essendo più lenta (circa 10 ms per operazione), produce fitness medie nettamente superiori, evidenziando una maggiore capacità esplorativa. Nel contesto attuale, in cui la funzione di fitness non è particolarmente onerosa, la differenza nei tempi risulta trascurabile ed è preferibile adottare configurazioni in grado di garantire una qualità più elevata. In prospettiva

futura, qualora la funzione di fitness dovesse diventare più complessa o computazionalmente pesante, JMH potrà essere nuovamente impiegato per individuare la configurazione ottimale rispetto al nuovo carico computazionale.

Selection	Crossover	Tempo (ms/op)	Media Fitness	Iterazioni Medie
roulette	uniform	7,995	88,104	28,64
ktournament	uniform	7,805	87,999	25,52
rank	uniform	10,328	95,418	39,97
roulette	single point	7,990	88,466	30,67
ktournament	single point	7,750	87,871	28,21
rank	single point	10,458	95,274	39,97
roulette	two point	7,479	88,258	29,50
ktournament	two point	7,788	87,877	27,41
rank	two point	10,163	95,158	39,90

Tabella 2.1: Risultati del benchmark JMH per diverse configurazioni dell'algoritmo genetico.

Il codice utilizzato per realizzare il benchmark dell'algoritmo genetico tramite JMH può essere trovato nella directory jmh del progetto (link).

2.4 Analisi della sicurezza e qualità del codice

2.4.1 SonarCloud

Per valutare la qualità complessiva del codice e individuare eventuali vulnerabilità, è stato utilizzato **SonarCloud** (integrato nella pipeline di CI come verrà mostrato in seguito). La prima esecuzione dell'analisi, come si può osservare nella Figura 2.6, ha fornito risultati complessivamente molto positivi: l'intero progetto presenta infatti **0 vulnerabilità di sicurezza**, **2 security hotspots**, **4 issue di reliability**, **203 code smells di maintainability** e un valore di duplicazione pari al **5%** su circa **6.6k** linee di codice.

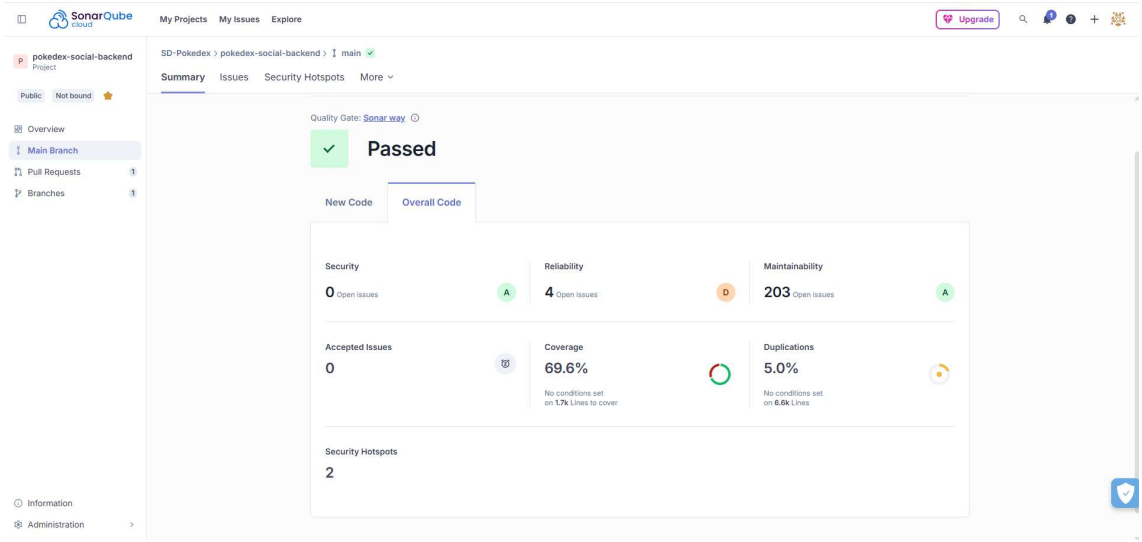


Figura 2.6: SonarCloud: Report Iniziale

I due **security hotspots** individuati da SonarCloud erano relativi all'uso della classe **Random** in alcune componenti interne dell'algoritmo genetico. L'uso non controllato di generatori pseudo-casuali può rappresentare un rischio in contesti sensibili, poiché la prevedibilità della sequenza generata potrebbe essere sfruttata per ricostruire o alterare il comportamento del sistema. Nel caso specifico, tuttavia, tali occorrenze sono risultate essere falsi positivi: le istanze di **Random** erano infatti variabili private, utilizzate esclusivamente all'interno di operatori genetici che non gestiscono dati critici né influiscono sulla sicurezza dell'intero sistema. Le issue di **reliability** riguardavano principalmente la mancanza di cast espliciti in alcune espressioni, potenzialmente fonte di ambiguità o warning a tempo di compilazione. Tali problemi sono stati risolti introducendo cast chiari e tipizzati, rendendo il codice più robusto e maggiormente aderente alle best practice di Java. Il risultato dell'analisi effettuata alla fine del progetto può essere osservato nella Figura 2.7. È opportuno notare che la coverage segnalata è minore rispetto a quella di JaCoCo perchè fa riferimento alla Line Coverage e non alla Branch Coverage.

2.4.2 Snyk

Per l'analisi delle dipendenze è stato utilizzato Snyk, che ha identificato un totale di **4 vulnerabilità** nel Backend (Figura 2.8), di cui **2 ad alta severità** e **2 a severità media**. Tutte le vulnerabilità rilevate erano introdotte indirettamente attraverso gli *starter*

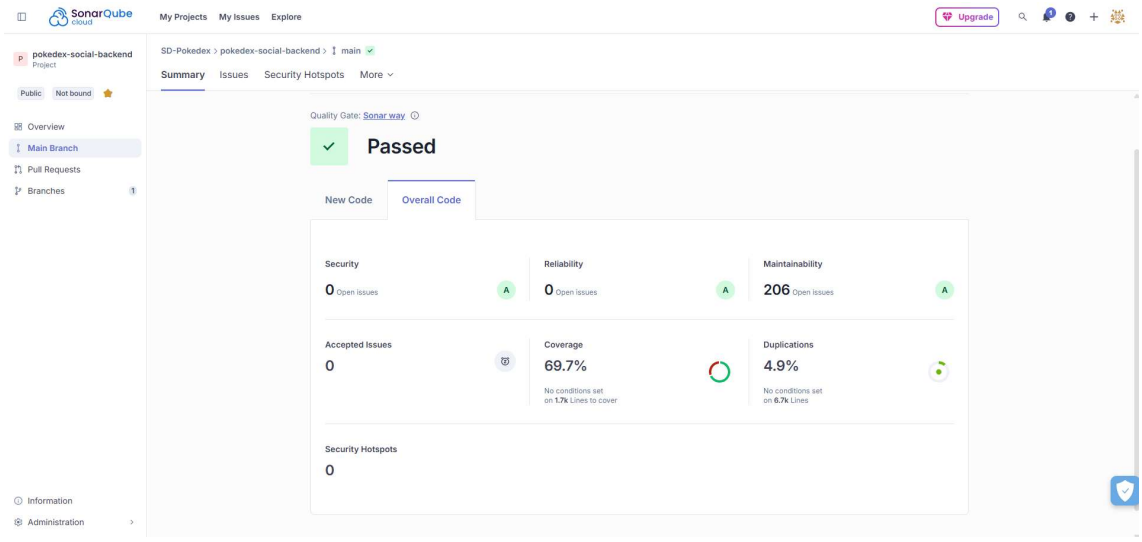


Figura 2.7: SonarCloud: Report Finale

di Spring Boot utilizzati dal progetto e riguardavano librerie comuni come **Tomcat**, **Spring Core** e **Logback**.

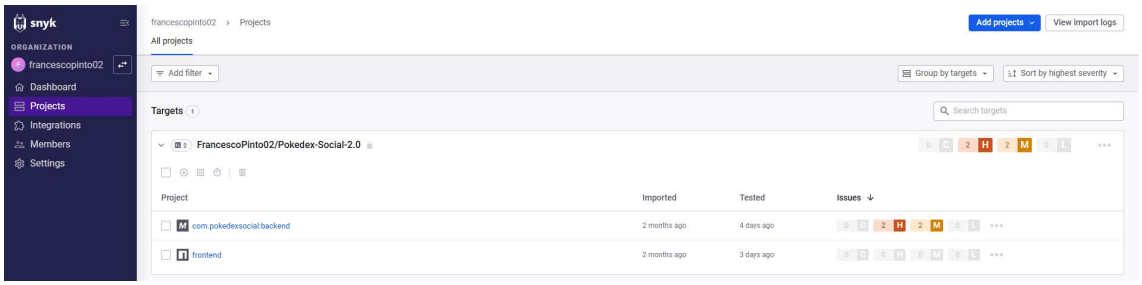


Figura 2.8: Snyk: Report Iniziale

Poiché tali vulnerabilità erano già state corrette nelle versioni successive dei pacchetti, è stato sufficiente aggiornare la dipendenza principale di Spring Boot dalla versione **3.5.5** alla versione **3.5.8** per risolvere interamente i problemi segnalati. Dopo l'aggiornamento, una nuova scansione Snyk non ha riportato ulteriori vulnerabilità (Figura 2.9).

2.4.3 GitGuardian

Per prevenire la presenza di credenziali o informazioni sensibili all'interno del repository, è stato utilizzato **GitGuardian**, uno strumento specializzato nel rilevamento di **Secret Leaks**. L'analisi ha individuato diversi **secret incidents**, risultati tuttavia quasi

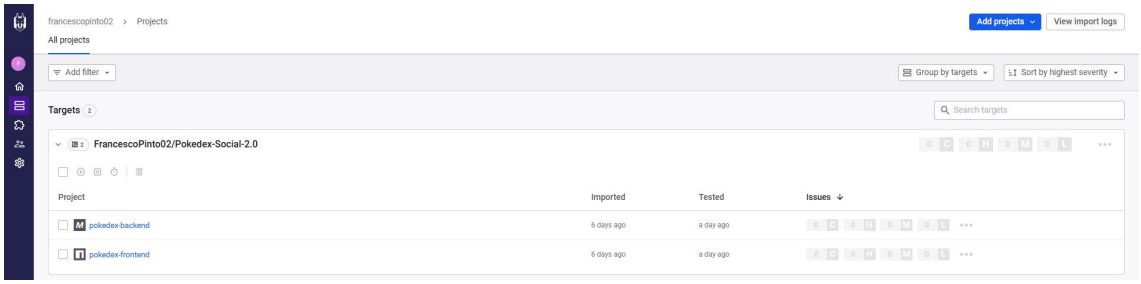


Figura 2.9: Snyk: Report Finale

interamente **falsi positivi**. In particolare, molte segnalazioni riguardavano credenziali fittizie utilizzate nei casi di test o credenziali di accesso al database impiegate unicamente in ambiente locale tramite Docker.

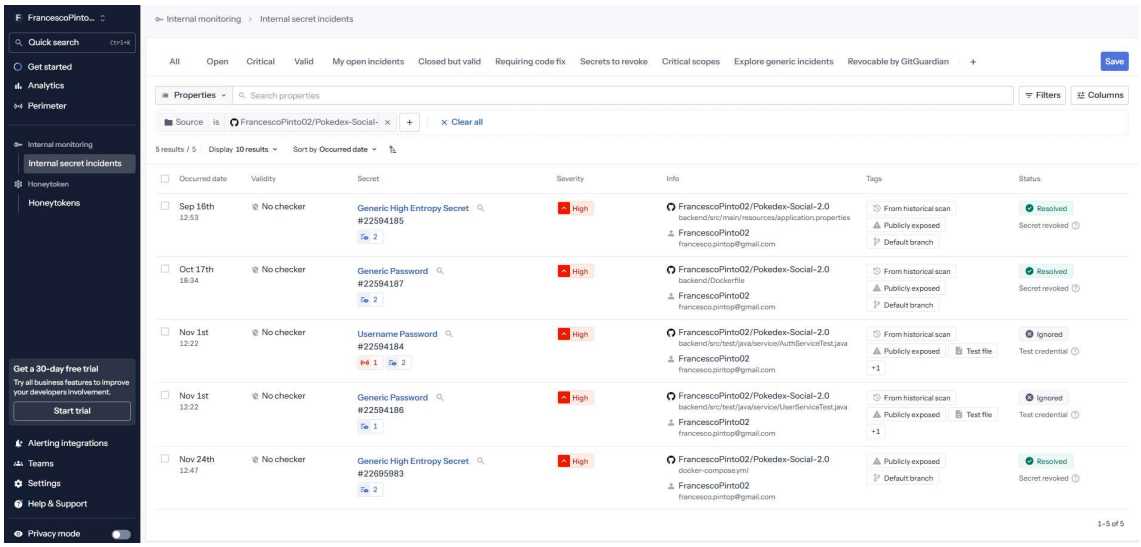


Figura 2.10: GitGuardian: Report

L'unica segnalazione meritevole di maggiore attenzione ha riguardato un **Generic High Entropy Secret**, relativo alla presenza di una chiave segreta utilizzata per la generazione dei token di autenticazione, riportata in chiaro sia nel file **application.properties** sia nel **Dockerfile**. Sebbene tale chiave fosse impiegata esclusivamente per scopi di sviluppo locale e non costituisse un reale rischio operativo, è stato comunque opportuno intervenire a scopo didattico e per mantenere buone pratiche di sicurezza. La chiave è stata quindi rimossa dal codice e sostituita con una gestione tramite **variabili d'ambiente**, applicata sia alla configurazione Spring sia al Dockerfile. Il report completo di GitGuardian può essere osservato nella Figura 2.10.

2.4.4 Dependabot

Per garantire l'aggiornamento continuo delle dipendenze e prevenire l'introduzione di vulnerabilità note attraverso librerie obsolete, è stato abilitato Dependabot all'interno del repository GitHub. Al primo ciclo di analisi, Dependabot ha rilevato due vulnerabilità di severità moderata all'interno del **frontend**: una relativa a **Vite** (vulnerabilità diretta) e una riguardante **js-yaml**, introdotta in modo transitivo da altre librerie. È stato sufficiente aggiornare le versioni delle librerie corrispondenti nel file package.json per correggere entrambe le vulnerabilità (Figura 2.11)

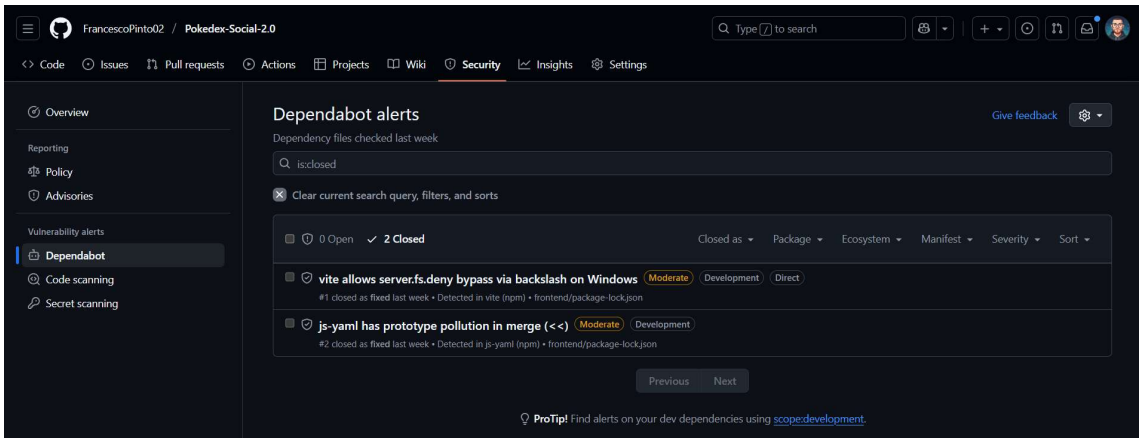


Figura 2.11: Dependabot: Report

2.5 Containerizzazione con Docker e Docker Compose

L'intera applicazione è stata containerizzata mediante Docker, realizzando un'immagine separata per ciascuna delle tre componenti principali: **backend**, **frontend** e **database**. Tutti i Dockerfile sono stati progettati seguendo le best practice moderne, in particolare l'uso di build **multi-stage** per ottenere immagini finali leggere, sicure e facilmente distribuibili.

2.5.1 Backend

Il backend Spring Boot è stato containerizzato attraverso un Dockerfile multi-stage composto da una fase di build basata su Maven e una fase di runtime basata su

eclipse-temurin 17. Durante la fase di build vengono recuperate in anticipo le dipendenze e viene compilato l'intero progetto. L'immagine finale include unicamente il **jar** prodotto, evitando la presenza di Maven e riducendo significativamente la dimensione complessiva.

2.5.2 Frontend

Anche il frontend è stato containerizzato mediante un approccio multi-stage: una prima immagine **Node.js** esegue la fase di build tramite **Vite**, mentre la fase finale utilizza una immagine Node preridotta che installa esclusivamente il tool **serve**. Solo la cartella **dist**, contenente gli asset statici già compilati, viene inclusa nell'immagine finale, eliminando completamente **node_modules** e garantendo un'immagine estremamente compatta.

2.5.3 Database PostgreSQL

A differenza di molti progetti analoghi, è stato definito un Dockerfile anche per il database PostgreSQL. Pur utilizzando come base l'immagine ufficiale, sono stati aggiunti gli script SQL e i dataset necessari all'inizializzazione del database, copiandoli nella directory `/docker-entrypoint-initdb.d/`:

```
FROM postgres:15

COPY db_scripts/ /docker-entrypoint-initdb.d/
COPY dataset/ /docker-entrypoint-initdb.d/dataset/
```

Questa scelta permette di distribuire un'immagine del database completamente autonoma. In questo modo, chiunque desideri eseguire l'applicazione non deve scaricare manualmente i file CSV o gli script di creazione delle tabelle: l'intero database viene inizializzato automaticamente alla prima esecuzione del container. Ciò semplifica enormemente l'avvio dell'applicazione tramite Docker Compose e consente la pubblicazione integrale dell'architettura su Docker Hub, offrendo un'esperienza "one-command setup" per utenti e sviluppatori.

2.5.4 Orchestrazione con Docker Compose

Le tre immagini vengono infine orchestrate tramite un file **docker-compose.yml**, che definisce i servizi, le dipendenze, le variabili d'ambiente e le porte esposte. Docker Compose consente di avviare l'intera applicazione con un unico comando, garantendo un ambiente consistente e pienamente riproducibile, particolarmente utile sia per lo sviluppo locale sia per le dimostrazioni del progetto. Le immagini costruite sono state inoltre caricate su Docker Hub, rendendo possibile eseguire l'intera applicazione senza necessità di compilare manualmente alcun componente. Il file `docker-compose.yml` è disponibile nel repository GitHub del progetto (link).

Per poter avviare correttamente il sistema tramite Docker Compose è necessario definire una variabile d'ambiente contenente la *secret key* utilizzata per la generazione dei token JWT. A titolo esemplificativo, è possibile generare una chiave tramite un apposito JWT Secret Generator (link).

2.6 Continuous Integration con GitHub Actions

L'intero progetto è supportato da una pipeline di Continuous Integration realizzata tramite GitHub Actions. La pipeline integra quasi tutti gli strumenti utilizzati nel corso e automatizza le attività di build, testing, analisi della qualità del codice, scansione di sicurezza e distribuzione delle immagini Docker.

2.6.1 Build, Test e Coverage

Il workflow principale si occupa della compilazione del backend e del frontend, dell'esecuzione dei test e della generazione dei report di copertura. In particolare, per il backend vengono eseguite automaticamente:

- la build Maven con esecuzione dei test JUnit;
- la generazione del report JaCoCo;
- l'upload dei report di test e coverage su Codecov;

- l'esecuzione del mutation testing con PiTest;
- la pubblicazione dei report JaCoCo e PiTest su GitHub Pages.

Il frontend viene invece costruito tramite Node.js e il risultato della fase di build viene archiviato come artefatto. Questo workflow garantisce che ogni modifica a main o ogni pull request venga validata tramite un processo di compilazione completo, analisi dei test e valutazione della copertura.

2.6.2 Analisi di Sicurezza e Qualità

La pipeline integra i diversi strumenti dedicati alla sicurezza e alla qualità del codice mostrati in precedenza:

- **SonarCloud**: eseguito ad ogni push o pull request, analizza bug, vulnerabilità e code smells del backend Spring Boot.
- **Snyk**: esegue una scansione delle dipendenze sia del backend sia del frontend, identificando vulnerabilità note (CVE) e monitorando automaticamente l'evoluzione della sicurezza nel tempo.
- **GitGuardian**: effettua una scansione completa del repository per rilevare eventuali segreti o credenziali sensibili accidentalmente committate.

2.6.3 Build e Distribuzione delle Immagini Docker

Una volta completato con successo il workflow di build e test, viene avviato automaticamente il workflow dedicato alla containerizzazione. Questo job:

- costruisce le immagini Docker di backend, frontend e database tramite Docker Build;
- effettua il login su Docker Hub tramite credenziali protette (Github Secrets);
- pubblica le immagini aggiornate con il tag **latest**.

Questa automazione consente di mantenere un registro di immagini sempre aggiornato e rende possibile avviare l'intera applicazione tramite Docker Compose senza alcuna operazione manuale.

CAPITOLO 3

Conclusioni

3.1 Risultati ottenuti

Il progetto ha permesso di integrare in modo efficace diversi strumenti e metodologie orientati alla software dependability. Le specifiche formali JML hanno garantito una descrizione rigorosa di alcune componenti critiche, mentre la definizione di una test suite completa, supportata da JaCoCo e PiTest, ha migliorato la robustezza e la qualità del codice. L'analisi prestazionale tramite JMH ha fornito indicazioni utili sull'efficienza dell'algoritmo genetico, permettendo di confrontare le principali configurazioni operative.

Gli strumenti di sicurezza e qualità del codice (SonarCloud, Snyk, GitGuardian, Dependabot) hanno assicurato un monitoraggio continuo e automatico delle vulnerabilità e delle dipendenze. La containerizzazione con Docker, insieme all'orchestrazione tramite Docker Compose, ha reso l'applicazione facilmente distribuibile e riproducibile. Infine, la pipeline di CI realizzata con GitHub Actions ha permesso di automatizzare l'intero processo di build, test, analisi e distribuzione.

3.2 Sviluppi futuri

Tra i possibili sviluppi futuri si possono includere:

- l'estensione delle specifiche JML alle restanti componenti dell'algoritmo genetico, qualora le limitazioni di OpenJML lo consentano;
- l'ampliamento della test suite con ulteriori casi dedicati alle componenti del frontend e ai flussi end-to-end;
- l'esecuzione periodica di benchmark JMH per valutare l'impatto di eventuali evoluzioni della funzione di fitness;
- l'integrazione di strumenti per il monitoraggio in ambiente di produzione, qualora venisse previsto un deploy reale dell'applicazione.

Bibliografia