POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Progettazione ed implementazione di test end-to-end per l'applicazione mobile per gli studenti del Politecnico di Torino

Relatori
prof. Marco Torchiano
prof. Riccardo Coppola

Candidato
Giorgio Ferraro

Marzo 2025

Indice

Elenco delle tabelle

Elenco delle figure

Ringraziamenti

Ringrazio i professori Marco Torchiano e Riccardo Coppola per avermi dato la possibilità di realizzare questa tesi. Ringrazio i miei cari, che mi hanno sempre sostenuto nel corso dei miei studi.

Capitolo 1

Introduzione

Il software è diventato una componente essenziale e onnipresente della vita moderna. Dall'uso quotidiano degli smartphone e dei social media, fino ai sistemi complessi che gestiscono infrastrutture critiche come l'energia, i trasporti e la sanità. Grazie alla digitalizzazione crescente, gran parte delle nostre attività personali e professionali sono facilitate e automatizzate da programmi e applicazioni. In questo contesto, il software è uno strumento che supporta l'intera infrastruttura della società contemporanea.
Durante gli anni i programmi diventano sempre più complessi per adattarsi a ogni situazione e per supportare qualsiasi attività. La crescita di complessità determina un aumento dei problemi legati allo sviluppo di codice e un'esigenza nell'eliminare questi malfunzionamenti per assicurare il corretto svolgimento delle attività. Per questo è nato il testing, un nuovo processo in grado di garantire il corretto funzionamento degli applicativi. Il testing permette di individuare e rimuovere gli errori durante la fase di sviluppo, aumenta la qualità del software e la fiducia da parte del cliente verso l'azienda. È necessario per applicazioni di alta qualità che vogliono mantenere un basso costo di manutenzione ed è fondamentale per riuscire a mantenersi sul mercato.

Nonostante quest'attività risulti dispendiosa in termini di tempo e costo da parte delle aziende, è importante notare come la mancanza di test o la superficialità nel condurli, abbia determinato situazioni disastrose sia dal punto di vista economico che in termini di vite umane. Esistono molti casi in letteratura che ne dimostrano l'importanza come i seguenti [2] :

- Nel 1994 in Scozia, un errore nel sistema di un elicottero Chinook ha portato allo schianto di quest'ultimo e alla morte di 29 passeggeri.

- Nel 1996 un razzo europeo Ariane 5 esplose poco dopo essere stato lanciato a causa di un difetto nel software. La superficialità nel condurre i test ha determinato una perdita di 370 milioni di dollari.

Il testing è diventato fondamentale anche nelle applicazioni mobili, dato che i telefoni sono i dispositivi più utilizzati sul mercato. Alcuni degli aspetti fondamentali delle applicazioni mobili sono l'interfaccia grafica e l'esperienza utente. Questi due fattori nella maggior parte dei casi determinano il successo dell'app stessa. In questo contesto nasce il testing di interfaccia utente, detto *GUI testing*. La tecnica ancora non è molto matura, per via delle difficoltà nei test delle applicazioni mobili, perciò si tende a preferire un test di tipo manuale, soprattutto per i progetti più piccoli. Nei progetti più grandi si implementano degli script per automatizzare il test, per ridurre l'errore umano delle verifiche manuali e per velocizzare il processo. Nonostante sia possibile automatizzare questa attività, la tipologia in sé presenta molte sfide come la fragilità, dovuta alla miriade di dispositivi mobili con sistemi operativi e hardware diversi. Inoltre la rapidità con cui cambia l'interfaccia utente determina un continuo aggiornamento dei test, quindi un costo che non tutti sono disposti a pagare.

Lo studio si concentra sul testing di interfaccia nell'applicazione per gli studenti del Politecnico di Torino, per garantire il corretto funzionamento della UI, attraverso lo strumento detto Appium.
Quest'ultimo è un framework open-source multi piattaforma utilizzato per automatizzare il test di applicazioni mobili. I test sviluppati con Appium sono di tipo black box, quindi non comportano modifiche dell'app, consentendo di verificare l'app come utente finale.
Inoltre si utilizza una pipeline di Continuous Integration and Continuous Deployment per automatizzare completamente il processo di test.

La struttura della tesi si articola in 7 capitoli:

- Background: spiega lo stato attuale del testing, e approfondisce in particolare quello di interfaccia nel contesto mobile.
- Strumenti utilizzati: descrive gli strumenti necessari al progetto.
- Contesto di applicazione: illustra l'ambito in cui si svolge questo studio.
- Descrizione progetto: definisce il progetto in tutte le sue parti.

- Pipeline CI/CD: descrive l'integrazione della suite di test in una pipeline di sviluppo.
- Limitazioni dell'approccio: espone le considerazioni sui problemi e i limiti del lavoro svolto e dello strumento usato.
- Conclusioni: rappresenta i possibili miglioramenti e le sfide ancora da risolvere relativi al testing di interfaccia per app mobili.

Capitolo 2

Background

2.1 Testing

Il test è un processo fondamentale nel ciclo di vita del software. È necessario per verificare che il comportamento del software rispetti i requisiti definiti in fase di progettazione. Lo scopo del testing è di individuare e rimuovere gli errori, al fine di minimizzare la probabilità che il software distribuito presenti dei difetti nella normale operatività. Costituisce un aspetto fondamentale per la riduzione del costo di sviluppo e il conseguente successo in termini economici per l'azienda e in termini di soddisfazione dell'utente nell'adottare quell'applicativo. Il processo di testing è ormai diventato parte integrante dello sviluppo, attraverso le metodologie *Agile*, che prevedono dei rilasci sempre più ravvicinati. Questo approccio assicura lo sviluppo di un software che soddisfi certi requisiti funzionali e di performance, per garantire un' elevata qualità.
I malfunzionamenti ("failure") sono comportamenti del software difformi dai requisiti espliciti o impliciti. In pratica si verificano quando il sistema non fa quello che l'utente si aspetta. I difetti ("bug" o "defect") sono sequenze di istruzioni che quando eseguite generano un malfunzionamento. Di solito nella fase di testing si tenta di ridurre a zero la probabilità che il software abbia dei malfunzionamenti. Ovviamente è impossibile ridurre a zero questa probabilità, in quanto le possibili combinazioni di input validi sono enormi e non possono essere riprodotte in un tempo ragionevole. Tuttavia questo processo è necessario e fondamentale per consegnare all'utente finale un prodotto accettabile. La fase di testing diventa fondamentale nei casi detti *life-critical*, cioè in cui un difetto può mettere a rischio la vita umana. In questi casi le probabilità di malfunzionamento devono essere ridotte al minimo, pertanto i

test sono più rigorosi, approfonditi e lunghi rispetto agli altri casi [1].

2.1.1 Fase di verifica e validazione

La fase di test può essere suddivisa in due parti: la fase di verifica e la fase di validazione.
La verifica serve a stabilire se il software è stato costruito rispettando i requisiti e le specifiche. Un esempio tipico di verifica è il controllo che le specifiche del software o di una sua parte siano rispettate sia come interfaccia che come funzionalità. Di solito la verifica è affidata al team di supporto qualità (QA team) e prende il nome di test statico, perché non prevede l'esecuzione di codice.
La validazione, invece, si concentra sul controllo del comportamento del software rispetto alle esigenze e aspettative dell'utente finale. È orientata al prodotto finito e verifica se il software soddisfa i requisiti funzionali e non funzionali. L'obiettivo della validazione è assicurarsi che il prodotto finale sia utile, funzionante e pronto per l'uso. Questa attività richiede l'esecuzione di codice, perciò è detta test dinamico e viene svolta dal team che si occupa del testing.

2.1.2 Tipologie di test

Il software testing può essere suddiviso in diverse categorie in base al livello di dettaglio del test stesso. Esistono tre tipologie principali di test detti rispettivamente *unit test*, *integration test* e *end-to-end test*. Come mostrato in Figura 2.1, i test seguono la seguente struttura piramidale per varie ragioni:

- La struttura implica un certo rigore sulla scrittura dei test. Prima si procede con la scrittura degli unit test, poi con gli integration test, e infine con gli end-to-end test. Si segue questo approccio perché è necessario verificare il funzionamento delle singole parti, prima di controllare l'intero sistema.

- I livelli hanno dimensioni diverse per una differenza nel numero di test scritti, perché le parti da verificare nei livelli inferiori sono in numero maggiore rispetto a quelli superiori.

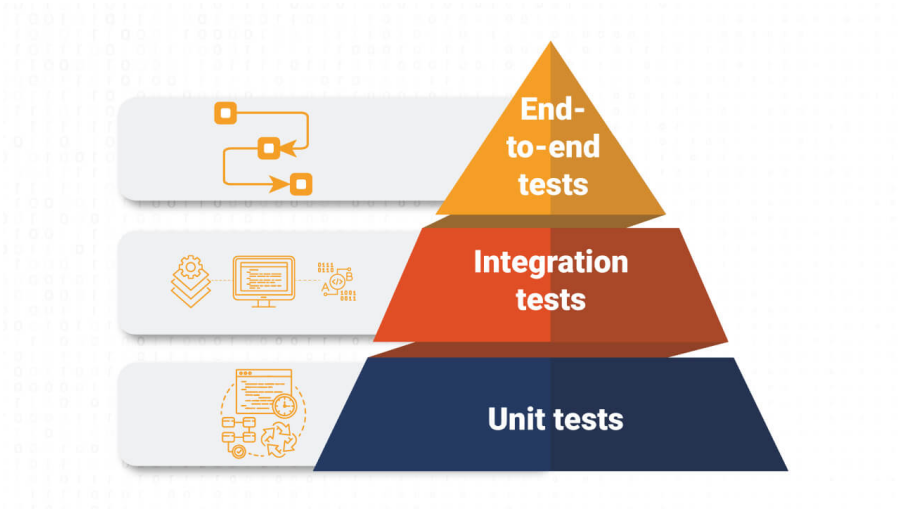

Figura 2.1. Test Pyramid. Immagine tratta da [13]

Unit test

Questa tipologia verifica le più piccole parti testabili di un'applicazione, dette anche unità, che nel caso della programmazione ad oggetti sono le singole classi e i loro metodi. Si controlla che ciascuna unità funzioni correttamente in modo isolato, senza alcuna interazione con altri moduli. I test di unità sono molto specifici e importanti, perché garantiscono una certa qualità, indispensabile per le altre tipologie di testing di più alto livello. Le unità che sono mal progettate o non verificate adeguatamente richiedono uno sforzo maggiore, perché si devono riprogettare da zero o comunque notevolmente modificare e testare. Di conseguenza l'impatto non è indifferente sia in termini di tempo nello sviluppo, sia in termini di costo sostenuto dall'azienda.

Integration test

L'Integration Test (test di integrazione) è una fase del software testing che verifica il corretto funzionamento dell'interazione tra diversi moduli o componenti di un'applicazione. Dopo che ogni modulo è stato testato individualmente (unit testing), il test di integrazione si concentra sull'assicurarsi che questi moduli collaborino correttamente quando combinati.
Obiettivi del test di integrazione:

- Verificare che i moduli comunichino correttamente tra loro.

- Identificare problemi legati all'interfaccia tra componenti (ad esempio, errori nei passaggi di dati o nei protocolli di comunicazione).

- Assicurare che l'integrazione di nuovi moduli non introduca bug in quelli già esistenti.

End to end test

Il test end-to-end (E2E) è una tipologia di test che verifica il funzionamento di un'applicazione nella sua totalità. Questo metodo verifica l'intero sistema dalla prospettiva utente e simula degli scenari reali per controllare che il software si comporti come previsto. I test end to end sono importanti e utili per controllare che l'applicazione soddisfi i requisiti e le specifiche utente e per prevenire problemi che possono manifestarsi nella fase di produzione.

2.2 Testing di applicazioni mobili

L'industria degli smartphone è cresciuta molto velocemente ed è ad oggi uno dei settori più fiorenti, con milioni di dispositivi venduti e utenti. Questo successo è dovuto alle migliaia di applicazioni, che soddisfano ogni esigenza del consumatore e dall'evoluzione di hardware sempre più performante, che ha reso questi dispositivi sempre più appetibili per l'utente finale. Le dimensioni ridotte dei dispositivi hanno reso necessario lo sviluppo del touch-screen, delle interfacce grafiche e di altri sensori come metodo per interagire con il sistema. La maggior parte del software sfrutta l'interazione con un'interfaccia grafica, detta GUI, per svolgere qualunque tipo di operazione. Da qui la necessità sempre maggiore di assicurare il corretto funzionamento attraverso il testing di questa componente. In questo contesto, il testing della GUI è importante, perché esistono migliaia di app con le stesse funzionalità, quindi l'interfaccia e l'esperienza utente diventano degli aspetti fondamentali per l'adozione e il successo della stessa.

Il GUI testing usa l'interfaccia grafica per verificare l'intero software e tutti i possibili servizi di back-end. L'approccio è di solito black-box, quindi senza l'accesso al codice sorgente e spesso senza una conoscenza dettagliata dell'architettura e implementazione del sistema. È un test di tipo end to end, che si concentra sui requisiti funzionali, le caratteristiche e i comportamenti del sistema. Fornisce anche delle opportunità per il testing di performance e robustezza [4].

L'enorme interesse di mercato per le applicazioni mobili ha consentito lo sviluppo di nuovi software o framework che supportano il testing d'interfaccia, assicurando la qualità delle app. Tuttavia le limitazioni nelle soluzioni di testing automatizzato e manuale hanno impedito la realizzazione di un approccio efficace, comprensivo e pratico per il testing. Queste restrizioni sono dovute alle difficoltà intrinseche del sistema da verificare, dato che presenta delle sfide non banali come la frammentazione, la fragilità dei test e tante altre. Il testing di applicazioni mobili rimane un'attività per lo più manuale e questo costa all'industria un enorme sforzo in termini di tempo e denaro [5].
La seguente sezione descrive lo stato dell'arte del testing di applicazioni mobili, quindi le sfide, gli strumenti e i servizi a disposizione per gli sviluppatori. La descrizione degli strumenti per il test delle applicazioni mobili è tratta dall'articolo *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing* [5]

2.2.1 Sfide

Le difficoltà del testing derivano dalle proprietà specifiche dei dispositivi mobili, dai vincoli aziendali e dalla mancanza di funzionalità importanti negli strumenti. Queste sono le motivazioni per cui il testing manuale è preferito rispetto a quello automatizzato.
Le app mobili accettano input dall'utente e da diversi sensori e componenti hardware. Risulta molto difficile emulare in un ambiente controllato i vari scenari di input. Inoltre le metodologie di sviluppo software *Agile* hanno accorciato la finestra temporale di rilascio, perciò è sempre più difficile verificare le app con molte configurazioni diverse, che sono rappresentative delle condizioni reali.

Frammentazione

La frammentazione è un problema derivante dal contesto mobile, caratterizzato dalla presenza di numerose configurazioni dovute ai milioni di dispositivi diversi disponibili sul mercato. Il testing di applicazioni deve considerare diverse configurazioni. Le configurazioni possono essere rappresentate come una matrice di test, che combina diverse variazioni di sistemi operativi, versioni e dispositivi. Inoltre durante il ciclo di vita del dispositivo, lo stesso può essere aggiornato con una versione più recente del sistema operativo. Questo comporta un ulteriore aumento delle possibili combinazioni definite nella

matrice di test, dato che versioni diverse del software di sistema potrebbero generare dei malfunzionamenti.

Figura 2.2. Frammentazione. Immagine tratta da [17]

La frammentazione rappresenta una delle più grandi sfide nel testing di applicazioni mobili ed è un problema che viene esacerbato col tempo, perché ogni anno vengono immessi nuovi telefoni sul mercato. Il problema è più evidente nel caso di Android, dato che esistono un'infinità di dispositivi con configurazioni diverse disponibili per i consumatori. Nel caso di iOS, la frammentazione è minore, perché il numero di dispositivi è limitato e controllato da Apple e i consumatori di solito posseggono l'ultimo smartphone con l'ultima versione del sistema operativo. Le possibili soluzioni sono i servizi cloud o crowd-based. I primi permettono agli sviluppatori di provare le loro app su molti sistemi fisici e virtuali. Quelli crowd-based impiegano vari tester per verificare il corretto funzionamento del software. In ogni caso questi servizi sono a pagamento, quindi solo alcune aziende e team possono permetterseli. La problematica principale legate a queste soluzioni riguarda le tempistiche troppo elevate nell'esecuzione dei test. Pertanto sono incompatibili con le pratiche DevOps e i rilasci sempre più ravvicinati.

Instabilità dei test

L'instabilità dei test si verifica quando l'attività di verifica fallisce o ha successo in modo impredicibile, senza nessuna modifica al codice. Di conseguenza, un test instabile è caratterizzato da un comportamento non deterministico. I test instabili presentano dei risultati inconsistenti tra esecuzioni

multiple, per questo correggerli è un compito non banale. Alcuni studi hanno stimato che il tempo necessario per correggerli è pari al 50% in più rispetto a test non instabili, e il costo annuo è di circa qualche milione di dollari. La correzione di questa problematica richiede più esecuzioni e un debugging approfondito per capire il motivo del fallimento e quindi più tempo rispetto a test non instabili. L'instabilità deve essere risolta per assicurare la qualità del software e per ridurre lo spreco di tempo e risorse [3].

Cause dell'instabilità dei test:

- **Problemi di concorrenza:** si verificano quando più test accedono alle risorse in maniera concorrente. Questo costituisce il cosiddetto problema delle corse critiche.
- **Dipendenze esterne:** l'interazione con sistemi esterni, come database, introduce dell'instabilità dovute a fattori come tempi di risposta variabile e latenze di rete.
- **Comportamento non deterministico:** il test dipende da elementi randomici o impredicibili come date e input utente.
- **Ambiente instabile:** l'ambiente di test non è isolato, controllato o consistente. Questo determina variazioni di prestazione, disponibilità e configurazione.
- **Asserzioni insufficienti:** il test non controlla tutti gli aspetti rilevanti del comportamento o del risultato atteso, lasciando spazio a falsi positivi o negativi.
- **Logica fallace:** il test contiene difetti, errori di battitura o errori nel codice che ne influenzano la funzionalità o la validità.

2.2.2 Framework di automazione

I framework di automazione sono strumenti essenziali per gli sviluppatori di applicazioni mobili, utilizzati frequentemente per raccogliere informazioni sui componenti dell'interfaccia e per simulare l'interazione dell'utente con il dispositivo. Questi framework forniscono delle interfacce di programmazione, dette API (Application Programming Interface), per scrivere test per la GUI di applicazioni mobili attraverso script manuali o registrati. Gli script

specificano una serie di azioni che devono essere eseguite sui vari componenti della GUI e li controllano tramite asserzioni.

Con il passare del tempo, gli strumenti di automazione si sono evoluti, tanto che attualmente è possibile identificare tre generazioni. La prima si basa sulle coordinate dei componenti di interfaccia sullo schermo. Questa generazione è la più fragile tra le tre, perché il minimo cambiamento della posizione dei componenti, come il cambio di risoluzione, determina il fallimento dei test. Significa che lo stesso test eseguito su un dispositivo con esito positivo, potrebbe fallire in un altro, semplicemente per le diverse dimensioni dello schermo. La seconda generazione sfrutta le API per individuare i componenti o widget dell'interfaccia. Questo approccio risulta più robusto rispetto al precedente e più accurato nell'estrazione dei componenti rispetto al livello successivo. Tuttavia presenta un problema dovuto all'identificazione degli elementi nella gerarchia dell'interfaccia. Questi elementi di solito sono individuati tramite degli identificativi. Il cambiamento di questi identificativi tra una versione dell'app e la successiva può comportare il fallimento dei test. La terza e ultima generazione si basa sul cosiddetto Visual GUI Testing (VGT). Sfrutta il riconoscimento parziale delle immagini dell'interfaccia e le catture dello schermo per verificare la correttezza della GUI. In alcuni casi il VGT è più robusto ai cambiamenti del layout, ma è più dipendente dalla rappresentazione grafica degli elementi. Se l'icona di un bottone cambia, il relativo test deve essere riscritto [4].

2.2.3 Strumenti di registrazione e riproduzione

Gli strumenti di registrazione e riproduzione (C&R) catturano l'interazione manuale dell'utente con il sistema da testare e creano degli script di test per riprodurre in modo automatizzato la sessione manuale. Sono molto apprezzati perché permettono di creare dei test automatizzati in modo semplice e veloce e riducono lo sforzo manuale per aggiornare gli script di test. Le suddette applicazioni permettono di catturare anche azioni specifiche e complesse e questo li rende molto accurati nelle simulazioni utente. Le versioni più recenti sono in grado di verificare se il comportamento dell'ultima versione è cambiato rispetto a una registrazione di una versione precedente. Di solito ogni test corrisponde a una versione di interazione manuale e deve essere registrata separatamente. Gli script di test creati con questi strumenti replicano l'interazione manuale con l'interfaccia, ma non sono in grado di verificarne automaticamente il corretto funzionamento. Per questo motivo,

è fondamentale integrare gli script con asserzioni, al fine di verificare che l'AUT operi come previsto.

2.2.4 Tecniche automatizzate per la generazione di input di test

L'Automated Test Input Generation (AIG) è una tecnica di testing del software che prevede la generazione automatica di sequenze di input per collaudare le applicazioni. È una metodologia particolarmente utile per le applicazioni complesse, dove l'interazione utente può essere molto variegata e difficile da coprire manualmente. L'obiettivo principale dell'AIG è identificare bug, problemi di usabilità e altri difetti nel software in modo efficiente ed efficace. Inoltre l'approccio velocizza la generazione dei casi di test e riduce lo sforzo sostenuto da sviluppatori e tester. Gli input possono essere dati, eventi (come clic o digitazioni), o sequenze di azioni che il software deve elaborare. Gli input possono essere generati utilizzando diversi approcci e algoritmi, ciascuno con caratteristiche specifiche. I metodi più comuni sono:

- Model-Based Input Generation
- Random Input Generation
- Systematic Input Generation

La Model-Based Input Generation (generazione di input basata su modelli) utilizza un modello formale che rappresenta il comportamento atteso del sistema o dell'interfaccia utente. Questo modello guida la creazione degli input, assicurando che i test coprano in modo efficace i diversi stati e transizioni del sistema. I modelli di riferimento sono diagrammi di stato, automi a stati finiti, diagrammi UML o modelli matematici.
La Random Input Generation (generazione casuale di input) consiste nel creare input in modo casuale, senza seguire un modello predefinito. Questo approccio è spesso utilizzato per stress testing o fuzz testing, dove si cercano comportamenti anomali o vulnerabilità inviando al sistema dati imprevedibili.
La Systematic Input Generation (generazione sistematica di input) crea input seguendo un approccio metodico e predefinito. Questo metodo garantisce una copertura completa o quasi completa di tutti i possibili input o combinazioni.
La scelta tra Model-Based, Random e Systematic Input Generation dipende

dal contesto del progetto, dalle risorse disponibili e dagli obiettivi specifici del test. Spesso, una combinazione di questi approcci può offrire la migliore copertura e affidabilità del software.

Esistono ancora delle sfide illustrate in molti lavori di ricerca come le seguenti:

- **Generazione di eventi di sistema**: gli eventi di sistema includono notifiche push, chiamate in arrivo, cambiamenti di rete, ecc. Questi eventi possono influenzare il comportamento dell'app. La mancata gestione degli eventi di sistema può ritardare l'individuazione di bug.
- **Costo nel riavviare un'app**: riavviare l'app dopo ogni test può essere necessario per garantire l'indipendenza dei test. Il riavvio dell'app aumenta i tempi di esecuzione dei test e può ritardare il rilascio del software.
- **Bisogno di input manuali specifici per interazioni complesse**: alcune interazioni utente sono complesse e richiedono input specifici che non possono essere facilmente generati automaticamente. Questo necessità aumenta i costi e il tempo necessario per i test.
- **Effetti collaterali in diverse esecuzioni**: si verificano quando l'esecuzione di un test influisce sul comportamento dei test successivi. Gli effetti collaterali determinano risultati di test inaccurati e difficili da replicare.
- **Bisogno di casi riproducibili**: è essenziale che i casi di test siano riproducibili per permettere il debugging e la verifica delle correzioni. La mancanza di riproducibilità complica il processo di risoluzione dei bug.
- **Simulazione di servizi e comunicazione tra app**: le app mobili spesso interagiscono con servizi esterni. La mancanza di simulazioni adeguate può portare a test non affidabili e incompleti.
- **Mancanza di supporto per la generazione di scenari di test su più dispositivi**: le app devono essere testate su una vasta gamma di dispositivi con diverse caratteristiche hardware e software. Questo aspetto può portare a una copertura di test insufficiente e a problemi non rilevati su alcuni dispositivi.

2.2.5 Strumenti per il controllo degli errori e dei difetti

Questi strumenti permettono di monitorare e riportare i bug, gli arresti anomali e il consumo di risorse durante l'esecuzione. Forniscono agli sviluppatori informazioni dettagliate come video, script di test sul fallimento con dei passi per riprodurre l'errore o tracce dello stack. Inoltre possono fornire delle informazioni sulle interazioni standard dell'utente con l'app o assistere l'utente finale nel costruire dei report di difetti utili agli sviluppatori. Le procedure di segnalazione sono possibili solo se il software è integrato all'interno dell'app. L'integrazione è importante, perché i feedback delle recensioni utente nei marketplaces non contengono abbastanza informazioni per individuare e risolvere il difetto. Di solito questi strumenti hanno un'utilità limitata perché forniscono soltanto dei report di arresti anomali.

2.2.6 Servizi crowd-based per il test di applicazioni mobile

I servizi crowd-based di testing mobile impiegano un gruppo di esperti finanziati per verificare le applicazioni mobili. Il gruppo di esperti di solito compie verifiche manuali sull'applicazione da testare (Application Under Test o AUT). Il testing manuale è ancora preferito a quello automatizzato proprio per i limiti e i problemi di quest'ultimo. Esistono varie tipologie di test per questi servizi come: il testing funzionale crowd-sourced, di usabilità, di sicurezza e di localizzazione. Il primo impiega gli esperti per trovare e riportare difetti nell'app. Gli esperti vengono compensati per il numero di bug veri trovati. La verifica di usabilità si concentra sul design dell'interfaccia e sull'esperienza utente per valutare la facilità d'uso e intuitività dell'app. I collaudi di sicurezza cercano di individuare le falle di progettazione in un'app che potrebbero compromettere la sicurezza utente. Il test di localizzazione si assicura che un'app funzioni in maniera appropriata in regioni geografiche diverse con diversi linguaggi in tutto il mondo.

2.2.7 Strumenti per lo streaming di dispositivi

Gli strumenti per lo streaming di dispositivi usano dispositivi virtuali o controllati da remoto con internet per realizzare il mobile testing. Supportano lo streaming sicuro di smartphone per beta tester crowd-sourced e forniscono l'accesso a un insieme di dispositivi fisici e virtuali dell'azienda stessa a un

team di controllo qualità. L'accesso a queste risorse permette di affrontare meglio il problema della frammentazione.

Capitolo 3

Strumenti utilizzati

3.1 Appium

Appium è uno strumento in grado di implementare l'automazione di interfacce su diverse piattaforme (mobile, web, desktop, ecc.) e con diversi linguaggi di programmazione (JS, Java, Python, ecc.). Le funzionalità appena descritte lo rendono molto appetibile rispetto ad altri strumenti. Il framework si compone di tre parti fondamentali che ne definiscono l'architettura e le API: Appium core, clients, drivers.
Il processo di automazione, illustrato nella Figura 3.1, è definito dalle interazioni tra quattro componenti: il client Appium (WebDriverIO), il server Appium, il driver di automazione (UiAutomator2) e il dispositivo Android o iOS.

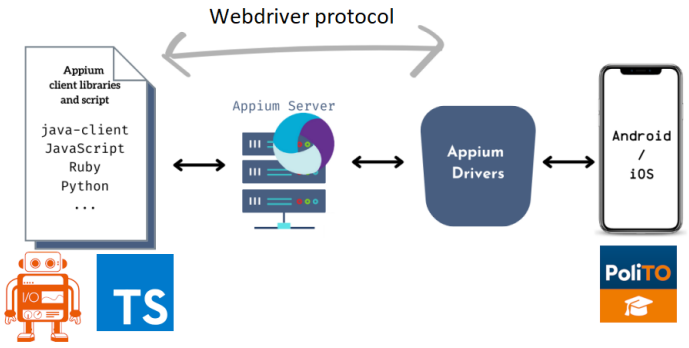

Figura 3.1. Processo d'automazione. Immagine tratta da [14]

La Figura 3.2 mostra le trasformazioni del codice nei vari moduli

Figura 3.2. Comando d'automazione

3.1.1 Appium core

L'appium core è il modulo che definisce le API di Appium. Queste provengono da Selenium, un software pioniere nell'ambito dell'automazione. Il team di Selenium insieme ai principali fornitori di browser e al consorzio W3C (World Wide Web Consortium) ha definito uno standard detto, *WebDriver specification*, per l'automazione nei browser. Nel tentativo di utilizzare e mantenere un unico standard, il team di Appium ha deciso di adottare la WebDriver specification come protocollo per qualunque piattaforma. La scelta di questa specifica è dovuta alla somiglianza delle interfacce tra browser e applicazioni mobili, nonostante le differenze nelle modalità d'interazione utente dei dispositivi. Le API di Appium sono usate dal client come modo per realizzare l'automazione sul dispositivo specifico. Alcuni dei comandi più comuni riguardano l'individuazione degli elementi di interfaccia e le azioni eseguite su di essi.

Inoltre un aspetto fondamentale dovuto proprio al protocollo WebDriver riguarda l'architettura di Appium. La specifica WebDriver è un protocollo

basato su richieste HTTP, quindi il software adotta una struttura client-server, che consente la modularità e le capacità multipiattaforma e multi-linguaggio.
L'architettura impone il rispetto delle linee guida nello sviluppo dei moduli client e driver. Il client deve essere in grado di utilizzare le API di Appium. Il driver deve ricevere le richieste del server e convertirle in comandi d'automazione per la piattaforma specifica. L'implementazione del driver è possibile solo se esiste una tecnologia di automazione sottostante e se il linguaggio possiede librerie HTTP. A parte questo, il client può essere sviluppato in qualunque linguaggio e si può realizzare un driver per qualunque piattaforma [6].

3.1.2 Client

Il client si occupa di inviare richieste e ricevere risposte dal server, usando le API del framework. Le risposte sono importanti per capire se il comando ha avuto successo o per ottenere informazioni sullo stato dell'AUT. I comandi disponibili al client vengono stabiliti dal server e dai plugin usati in una data sessione. I comandi base che hanno a disposizione i vari client, sono parte del protocollo WebDriver. Ad esempio il comando base *Find Element* corrisponde a un richiesta POST HTTP inviata all'endpoint HTTP */session/:sessionid/element*. Questo aspetto è rilevante per gli sviluppatori che lavorano con il protocollo WebDriver. Non è particolarmente utile per le persone che si limitano a scrivere i test con Appium. Quando si scrivono test, si usano le librerie client Appium, che convertono le funzioni scritte nel linguaggio di programmazione specifico in richieste HTTP, comprensibili al server. Le librerie forniscono una serie di comandi nativi per i vari linguaggi di programmazione, in questo modo sembra di scrivere codice Python, JS o Java. In molti casi, i client per un dato linguaggio sono costruiti sopra il client Selenium, perciò alcuni riportano nella documentazione solo le caratteristiche aggiunte rispetto all'implementazione base di Selenium [7].

WebDriverIO

Il client Appium selezionato per il progetto è WebDriverIO [9]. È un modulo integrato perfettamente con Appium, nonostante non faccia parte dei client mantenuti direttamente dal team di Appium. Di seguito sono elencati alcuni degli aspetti che contraddistinguono il software:

- Ottimo per l'automazione su dispositivi mobili e browser.

- Consente di creare e configurare un progetto in modo semplice e rapido.
- Offre vari strumenti per la scrittura di codice per l'automazione:
  - **reporters**: indispensabili per l'attività di test e debugging. Si occupa di registrare tutti i comandi scambiati tra il client e il server durante una sessione di test, fornendo un resoconto dettagliato dell'esito del test, soprattutto in caso di errore. Il client mette a disposizione vari reporter tra cui scegliere. Il progetto adotta tre reporter diversi.
    * **spec** (Figura 3.3): è il reporter standard e mostra tutto quello che avviene nella sessione direttamente nel terminale. È utile perché costituisce il metodo più rapido per gli sviluppatori di visualizzare e correggere eventuali errori nei test.

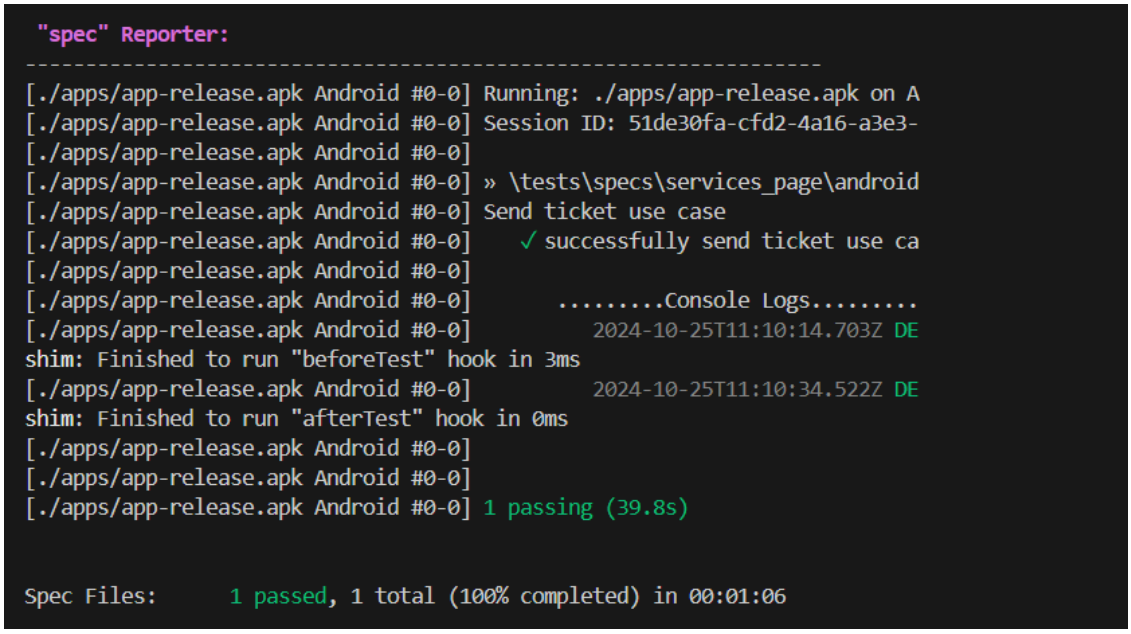

Figura 3.3. Spec reporter

    * **allure** (Figura 3.4): visualizza il report in un formato grafico dettagliato con molti diagrammi e screenshot. Rappresenta uno dei reporter più completi per le funzionalità che offre.
    * **ctrf-json** (Figura 3.5): è meno dettagliato di allure, però è fondamentale perché si integra perfettamente in una pipeline di

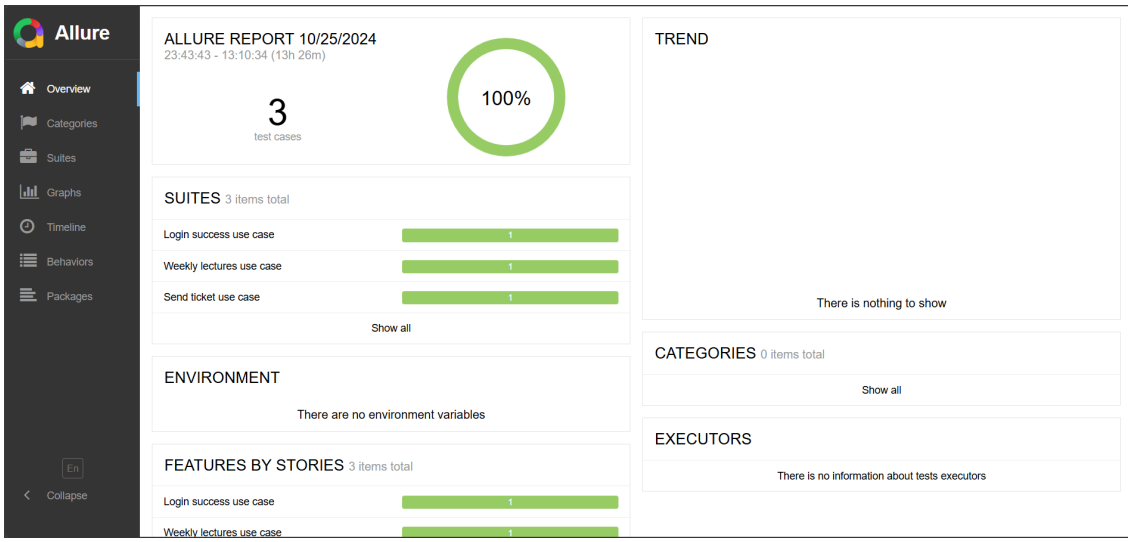

Figura 3.4. Allure reporter

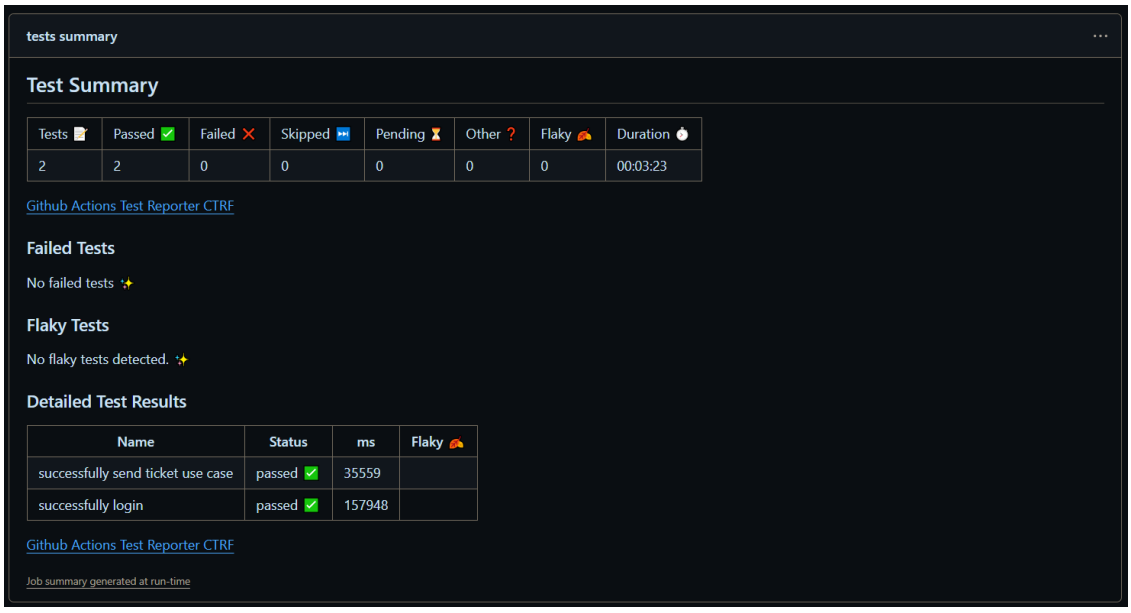

Figura 3.5. Ctrf-json reporter

Continuous Integration and Continuous Deployment (CI/CD) come quella delle Github Actions.

– **services**: estendono le funzionalità di base di WebDriverIO. Lo studio usa uno di questi servizi, ovvero il @wdio/appium-service.

Quest'ultimo è un servizio per WebDriverIO che semplifica l'uso di Appium. Si occupa di avviare e gestire automaticamente un server Appium, eliminando la necessità di eseguirlo manualmente o di configurare script personalizzati per avviarlo.

3.1.3 Driver

I driver realizzano l'automazione di interfaccia usando le tecnologie sottostanti di automazione della specifica piattaforma. Ad esempio i driver iOS usano la tecnologia Apple, detta XCUITest. Questi moduli sono semplicemente delle cassi Node.js che estendono una classe speciale, detta BaseDriver. Il driver eredita tutti gli attributi e i metodi della superclasse e implementa il protocollo WebDriver.
Per scrivere il codice utile al funzionamento del modulo, è necessario implementare del codice per i metodi del protocollo WebDriver. Il contenuto del metodo dipende dalla piattaforma specifica su cui si vuole realizzare l'automazione [8].

3.2 Appium Inspector

Appium Inspector è essenzialmente un client Appium, basato su WebDriverIO, con un'interfaccia grafica e funzionalità aggiuntive. È utile per lo sviluppo di test di automazione, poiché offre la possibilità di analizzare visivamente l'AUT. L'analisi dell'app richiede l'avvio del server Appium, del dispositivo fisico o emulato e infine della sessione con Appium Inspector. È necessario seguire l'ordine specificato precedentemente perché il client può creare la sessione e connettersi al server e al dispositivo solo se questi sono già attivi.
L'interfaccia del programma è composta da due parti: il Session Builder (Figura 3.6) e il Session Inspector (Figura 3.7).

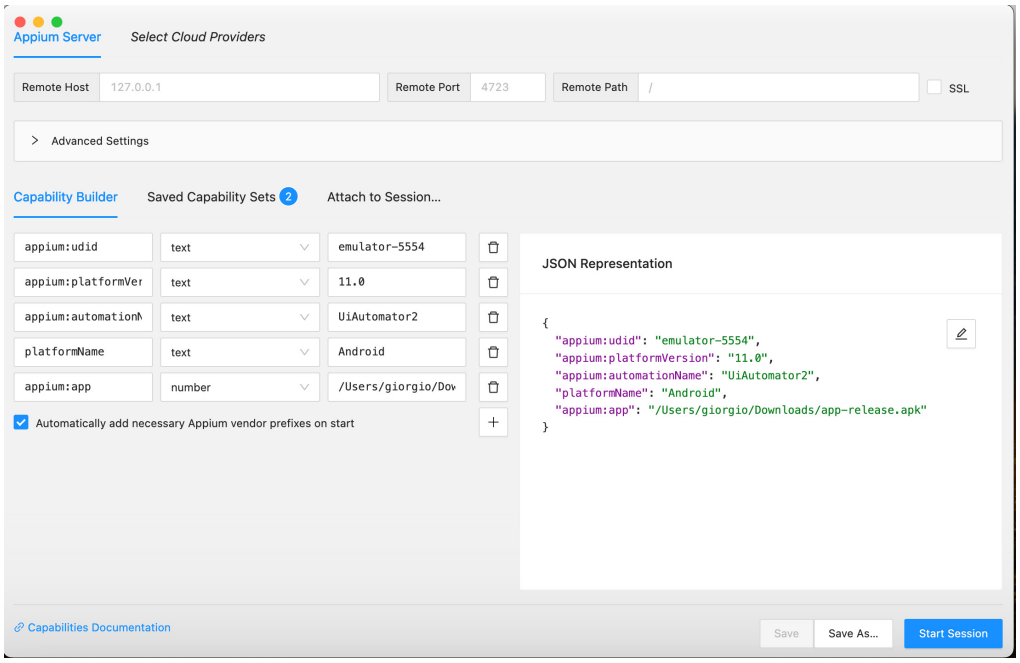

Figura 3.6. Session Builder

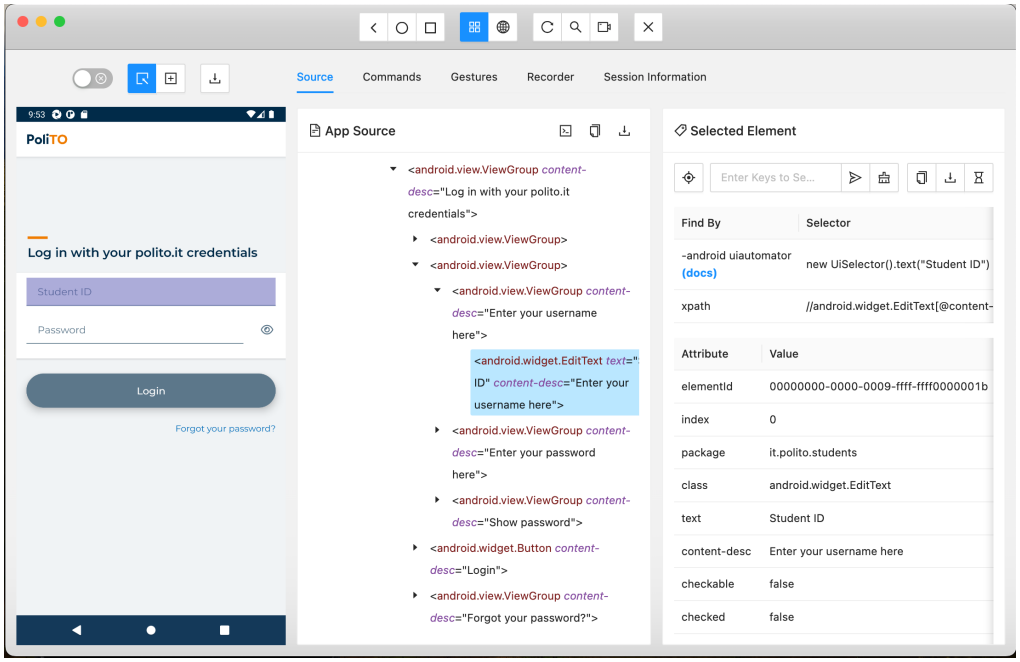

Figura 3.7. Session Inspector

3.2.1 Session Builder

Il Session Builder consente di avviare sessioni locali e remote per i test su dispositivi diversi (Figura 3.8). L'avvio di una sessione di analisi richiede l'impostazione di alcuni parametri fondamentali, detti *desired capabilities*, e di ulteriori parametri nel caso di connessioni remote. Le capabilities sono definite da un nome, una tipologia, che determina i valori che può assumere il parametro, e un valore. I tipi disponibili sono: text, boolean, number e JSON object. Ogni sessione ha bisogno di alcuni attributi fondamentali come: la piattaforma, Android o iOS, il driver di automazione, l'app da analizzare e il dispositivo fisico o emulato.
Inoltre il Builder permette di registrare e utilizzare configurazioni diverse per verificare l'app su dispositivi e sistemi operativi differenti. Una volta salvate, le configurazioni sono accessibili dal pannello *Saved Capability Sets*.

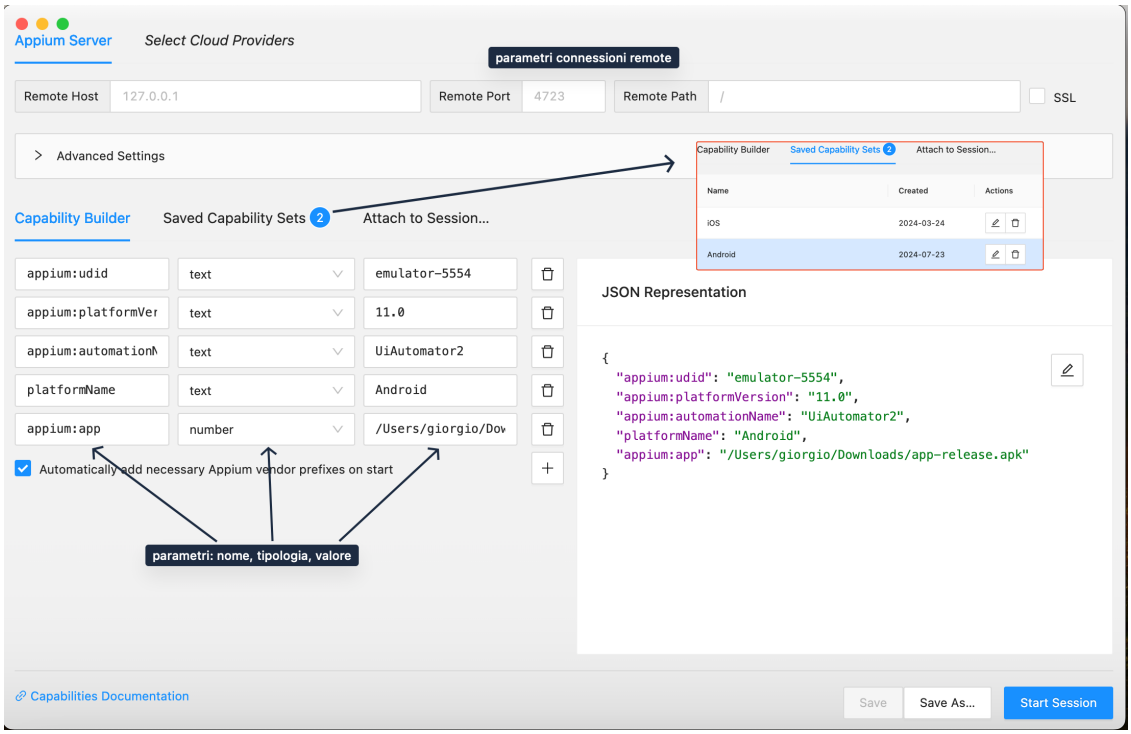

Figura 3.8. Funzioni e parametri Session Builder

3.2.2 Session Inspector

Il Session Inspector rappresenta l'interfaccia dell'applicazione, disponibile solo dopo l'avvio di una sessione. È formata da vari pannelli e sezioni, ognuno dei quali offre funzionalità diverse.

- **Header** (Figura 3.9): contiene dei comandi per interagire con il dispositivo e con la sezione relativa al codice XML. Alcuni dei comandi più utili riguardano la possibilità di cercare gli elementi attraverso i selettori e la capacità di registrare l'interazione manuale dell'utente.
- **Screenshot panel** (Figura 3.10): visualizza la schermata attuale dell'AUT. È possibile interagire con lo screenshot per selezionare e individuare i componenti dell'interfaccia. Una volta selezionato il componente, l'app visualizza il codice XML e i dettagli corrispondenti in due sezioni distinte.
- **Source tab** (Figura 3.11): è un pannello formato dall'Application Source panel e dal Selected Element panel. Il primo mostra il codice relativo alla GUI dell'app in formato XML e con una struttura gerarchica. Il secondo permette l'interazione con l'elemento selezionato e mostra i selettori e i dettagli del componente. Questo pannello è importante perché consente di ricavare gli identificativi necessari alla scrittura dei test.
- **Commands tab** (Figura 3.12): consente di eseguire alcuni comandi e script per ottenere informazioni di stato e per eseguire delle azioni sull'app e sul dispositivo. I comandi si riferiscono principalmente al driver Android (UiAutomator2), quindi alcuni di questi potrebbero non essere supportati su iOS.
- **Gestures tab** (Figura 3.13): è una scheda per creare e provare diverse gestures sull'AUT. Ogni gesture è formata da vari comandi come: Pointer Down, Pointer Up, Move e Pause. Le sequenze di queste istruzioni opportunamente disposte creano interazioni complesse come lo swipe, il pinch to zoom, e tanti altri.
- **Recorder tab** (Figura 3.14): è una sezione che viene popolata a seguito delle interazioni manuali dell'utente con l'AUT. La scheda contiene il codice generato dall'utilizzo dell'utente con l'app in diversi linguaggi di programmazione. La potenzialità di questo strumento risiede nella capacità di poter scrivere dei test di automazione in modo

semplice e veloce. Questa velocità determina una riduzione dei tempi necessari alla scrittura del codice di automazione e quindi rende i test automatizzati più appetibili. In generale, lo strumento è particolarmente efficace per gestire interazioni semplici su un'interfaccia che subisce frequenti modifiche.

- **Session information tab** (Figura 3.15): contiene le informazioni sulla sessione attiva, come l'ID, l'url, le desired capabilities e una sezione con il codice per poter ricreare la sessione attuale.

Figura 3.9. Header

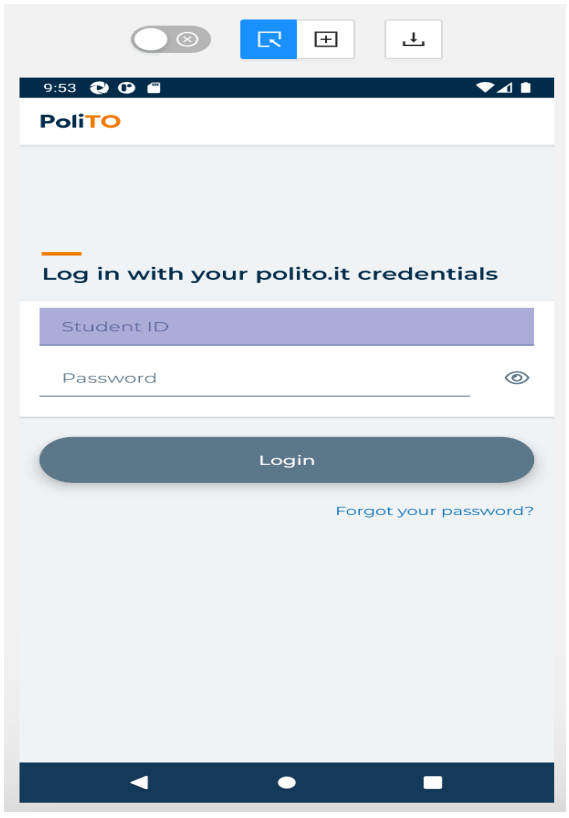

Figura 3.10. Screenshot Panel

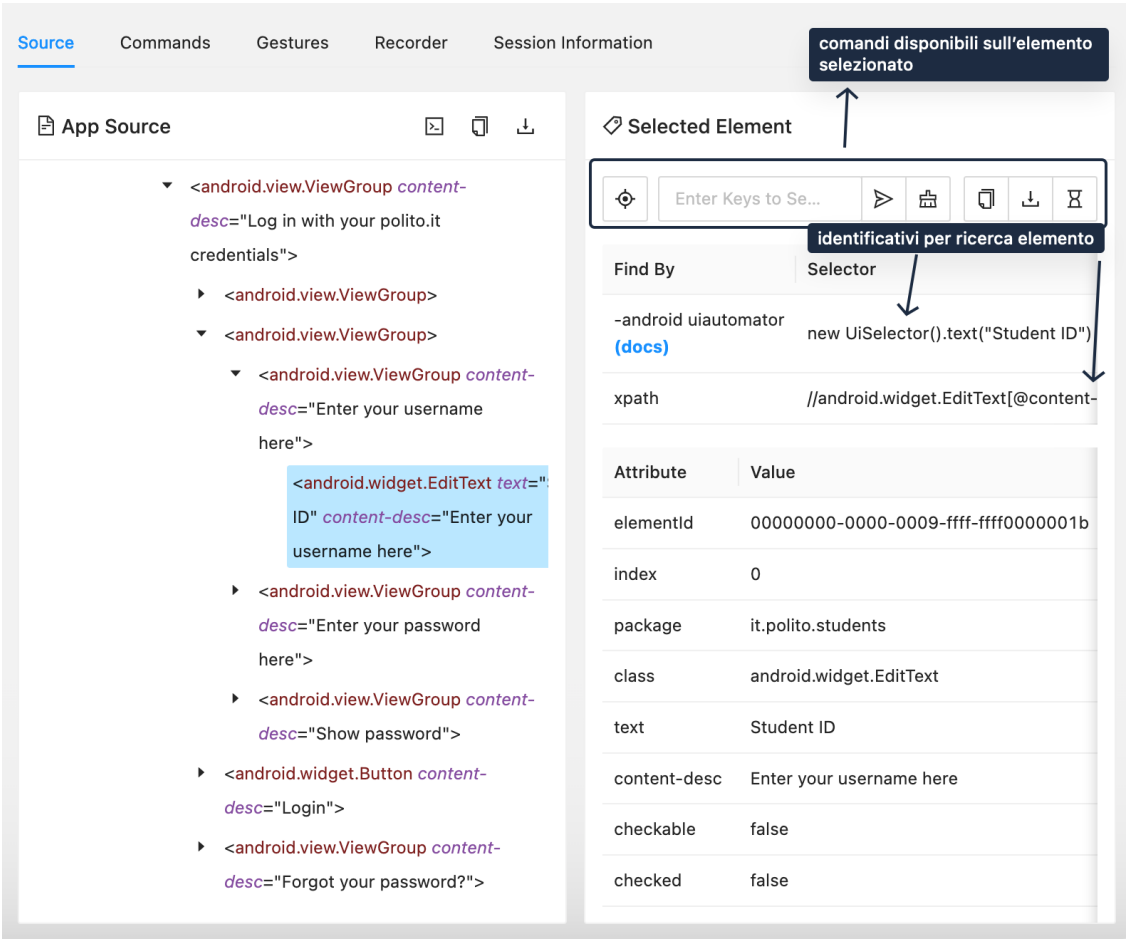

Figura 3.11. Source tab

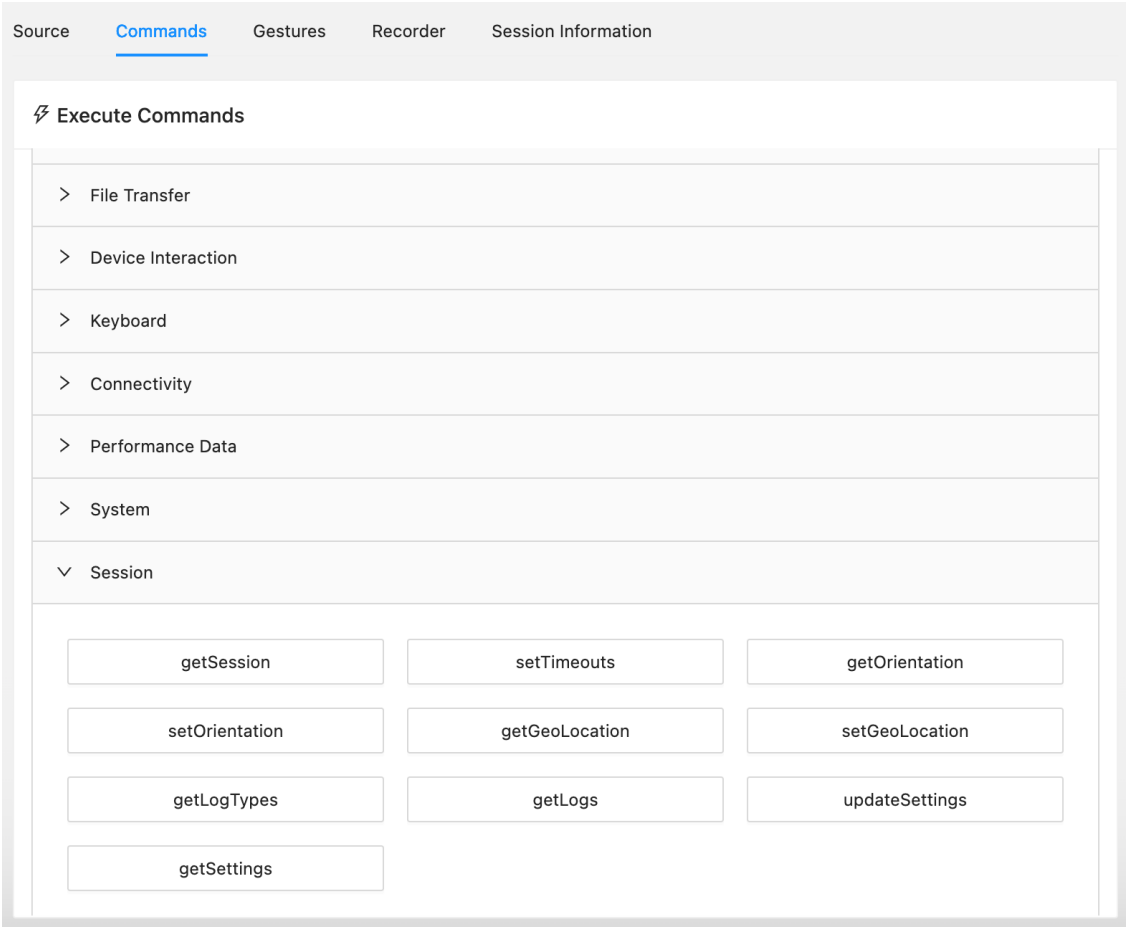

Figura 3.12. Commands tab

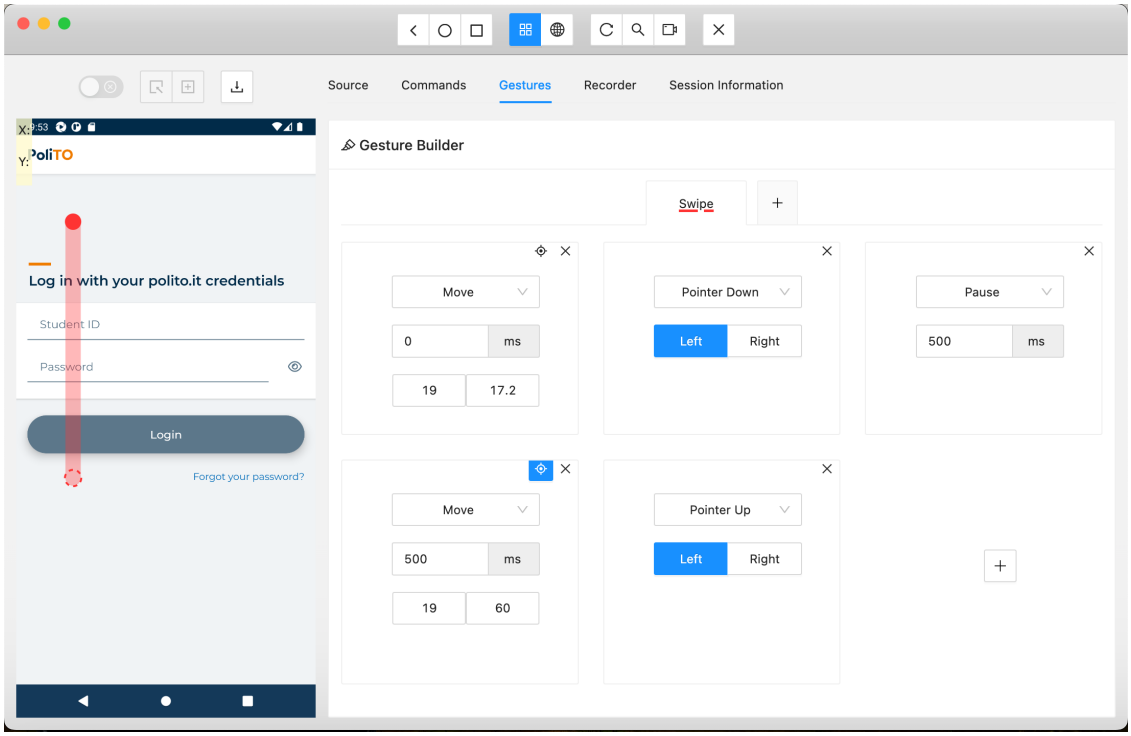

Figura 3.13. Gestures tab

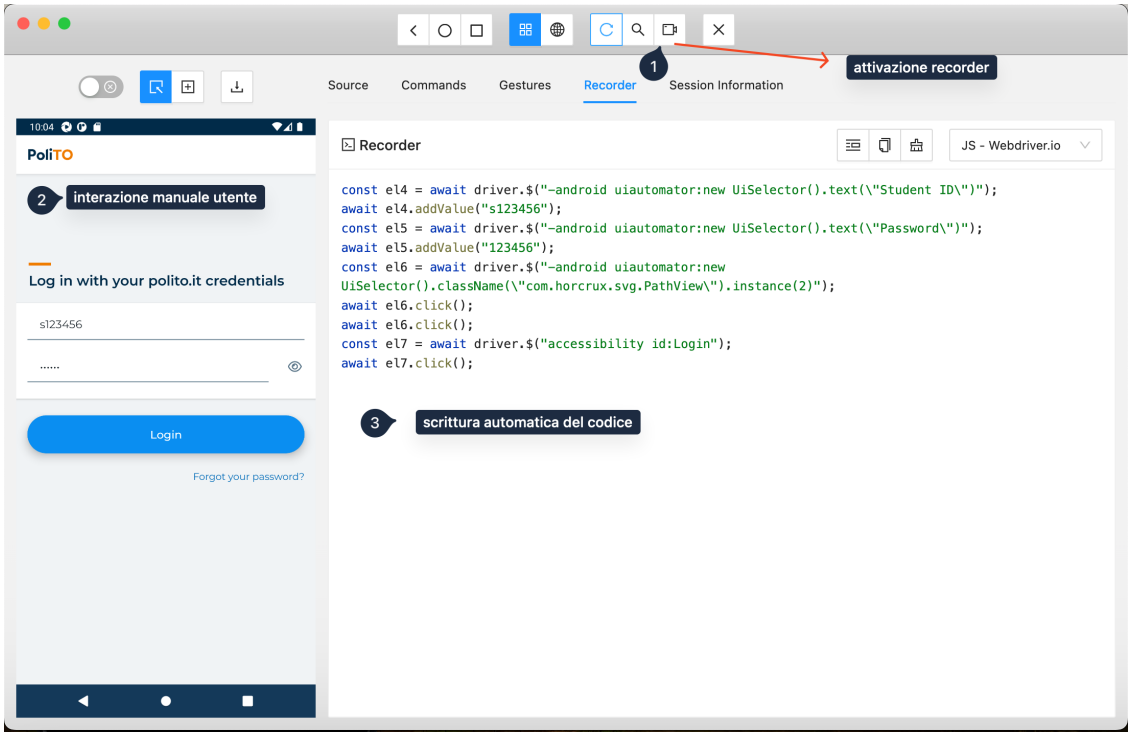

Figura 3.14. Recorder

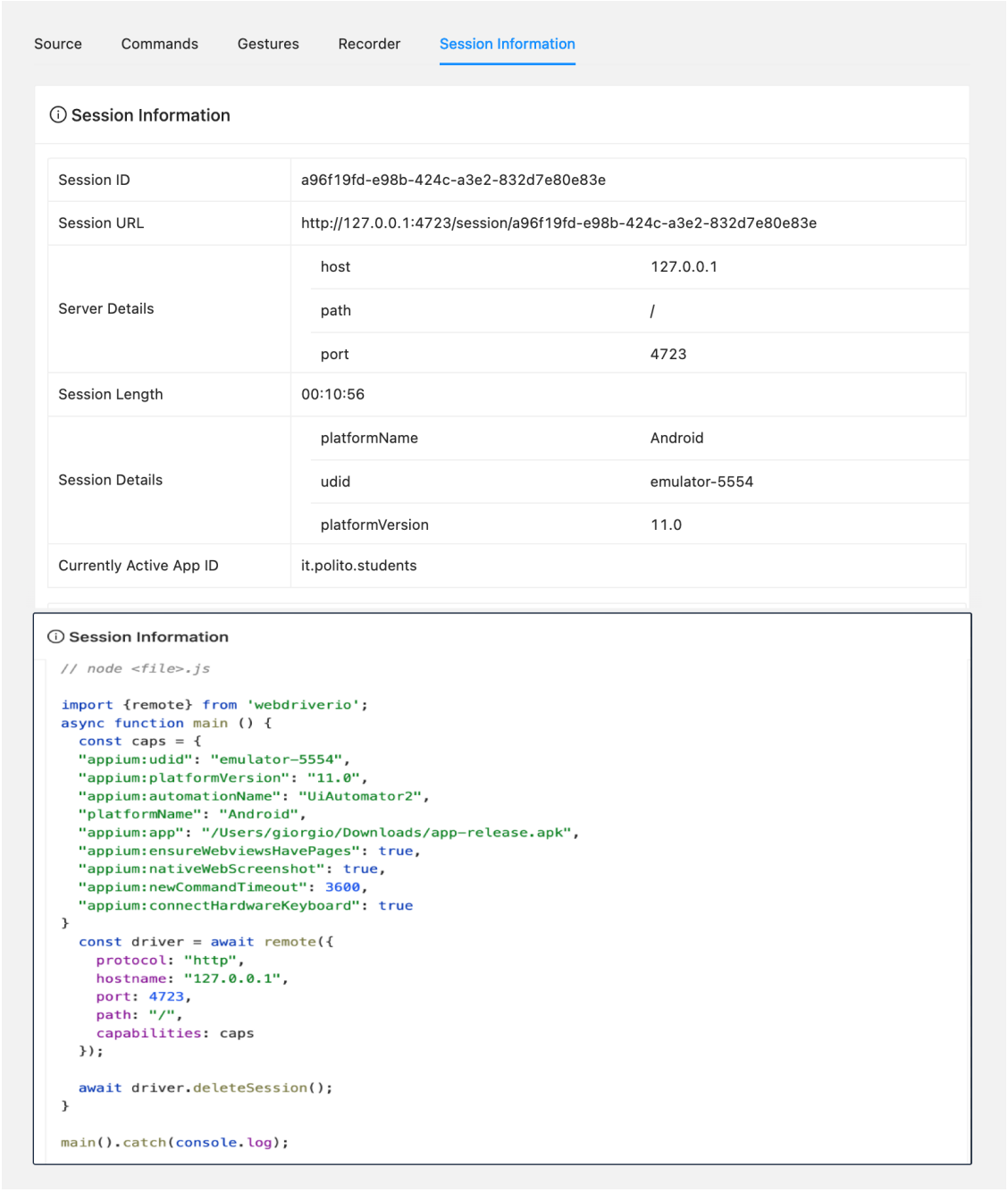

Figura 3.15. Session information tab

Capitolo 4

Contesto di applicazione

I test verificano l'applicazione per gli studenti del Politecnico di Torino, Polito Students App[1].
L'app consente agli studenti di accedere rapidamente a molte informazioni e servizi organizzati in cinque schermate diverse:

- **Didattica**: mostra le informazioni sui corsi, sugli appelli e un riepilogo generale dell'andamento della carriera dello studente.

- **Agenda**: mostra le lezioni, gli appelli e le prenotazioni.

- **Luoghi**: fornisce una mappa per raggiungere le aule, i laboratori e ogni altra struttura all'interno del Politecnico.

- **Servizi**: permette l'accesso a funzioni come l'email, le prenotazioni di aule studio e laboratori, le offerte di lavoro e tanto altro.

- **Profilo**: contiene le informazioni sulla carriera dello studente e gestisce le impostazioni dell'app.

[1]https://github.com/polito/students-app

Figura 4.1. Polito Students App. Immagine tratta da [15]

L'applicazione sfrutta il linguaggio di programmazione Typescript e il framework React Native per la compatibilità su entrambe le piattaforme iOS e Android. La caratteristica principale del framework è la possibilità di scrivere codice condiviso per entrambe le piattaforme, garantendo comunque prestazioni simili a quelle di un'app nativa.

Le caratteristiche principali di React Native sono elencate di seguito:

- Utilizza componenti simili a quelli di React (come <View>, <Text>, e <Button>) per costruire l'interfaccia utente.
- Sfrutta il JavaScript Bridge, che consente al codice JavaScript di comunicare con il codice nativo (Objective-C, Swift per iOS; Java, Kotlin per Android). Grazie a questo bridge, React Native può inviare istruzioni ai componenti nativi in tempo reale.
- Supporta l'hot reloading, che consente agli sviluppatori di vedere i cambiamenti in tempo reale senza dover ricompilare l'intera applicazione.
- Offre API predefinite per accedere a funzionalità del dispositivo, come fotocamera, geolocalizzazione, e sensori. È possibile scrivere moduli nativi in caso di esigenze avanzate.

Il progetto utilizza lo stesso linguaggio e il client per le API fornito dagli sviluppatori dell'app, con l'obiettivo di integrarsi al meglio e fornire un ulteriore strumento per verificare la correttezza dell'app.

Lo studio si concentra sull'analisi dell'interfaccia per individuare le funzioni più comunemente usate dagli studenti e costruire dei test mirati, che garantiscano una corretta esperienza utente. Quindi lo sviluppo dei test considera solo alcune delle schermate precedentemente descritte. L'unica pagina esclusa dalla verifica è Profilo, perché non ha informazioni o funzioni rilevanti nell'uso giornaliero.

Capitolo 5

Descrizione progetto

Il progetto è stato organizzato per adattarsi alle funzionalità offerte dall'app e al client WebDriverIO.

5.1 Struttura

Il client WebDriverIO ha imposto una certa struttura al codice per rispettare alcune buone pratiche della programmazione. La struttura si articola in tre cartelle principali:

- **apps**: contiene l'app del Politecnico di Torino per gli studenti, polito-students.apk.
- **config**: contiene i file di configurazione necessari per creare una sessione di test. Alcuni dei parametri (Figura 5.1) più rilevanti includono **gli specs**, che determinano quali test eseguire, **la piattaforma** (Android o iOS), **l'app** su cui eseguire l'automazione, **la tecnologia di automazione** (UiAutomator2) e **il dispositivo fisico o virtuale** su cui i test vengono eseguiti.
- **tests**: contiene tutti gli script di test nella cartella *specs* e le pagine nella cartella *pageobjects*.

5.2 Procedura per lo sviluppo dei test

La seguente sezione descrive i test e tutti gli elementi necessari alla loro creazione con i concetti di Use Case Narrative e pagina

```
const config = {

  specs: ['../tests/specs/android.spec.login.ts'],

  capabilities: [
    {
      platformName: 'Android',

      'appium:app': './apps/app-release.apk',

      'appium:automationName': 'UiAutomator2',

      'appium:udid': 'emulator-5554',
    },
  ],
};
```

Figura 5.1. Configuration file

5.2.1 Use Case Narrative

Lo sviluppo dei test ha richiesto una fase preliminare di progettazione, che sfrutta il concetto di Use Case Narrative [11]. Una narrazione di caso d'uso è una descrizione testuale dettagliata di come un sistema o un'applicazione interagisce con gli utenti o altri sistemi per raggiungere un obiettivo specifico. Serve a chiarire il comportamento atteso del sistema in situazioni specifiche, ed è uno strumento importante nel processo di analisi e progettazione del software. Gli elementi della narrativa sono descritti di seguito.

- **Use Case**: nome descrittivo del caso d'uso.
- **Scope**: nome del sistema interessato dal caso d'uso.
- **Level**: livello d'astrazione. Di solito assume il valore di *user goal*, per azioni specifiche degli utenti. Serve per organizzare i casi d'uso in base al contesto e alla granularità.

- **Intention in Context**: indica l'obiettivo principale dell'attore.
- **Primary Actor**: attore principale che interagisce con il sistema per raggiungere l'obiettivo.
- **Support Actor**: altri attori che supportano il caso d'uso (es. sistema di verifica e-mail per la conferma dell'account).
- **Stakeholders' Interests**: descrive gli interessi delle parti coinvolte.
- **Precondition**: condizioni che devono essere garantite prima che il caso d'uso possa iniziare.
- **Minimum Guarantees**: lo stato minimo che il sistema deve garantire anche in caso di fallimento del caso d'uso. Definisce il livello base di affidabilità del sistema.
- **Success Guarantees**: lo stato finale del sistema in caso di successo.
- **Trigger**: l'evento o la condizione che avvia il caso d'uso.
- **Main Success Scenario**: descrive la sequenza di azioni principali che conducono al completamento con successo del caso d'uso, raggiungendo gli obiettivi stabiliti senza errori o deviazioni.
- **Extensions**: scenari alternativi che possono verificarsi durante l'esecuzione del caso d'uso.

Questo è un esempio di use case narrative adottato nel progetto, che descrive in modo esaustivo la funzionalità di login dell'app.

Use Case: Login

Scope: Authentication System

Level: User goal

Intention In Context

- Student: A university student wants to securely access their academic records, course materials, and other educational resources through the mobile app.

- System: Must verify the student's identity and protect their personal and academic data from unauthorized access.

Primary Actor: Student

Support Actor

- Email Service: Sends notifications or links for password recovery.
- Authentication system: Manage verification of student credentials.

Stakeholders' Interests

- Registered student: Wants to quickly and securely access their personal information.
- Application Developer: Wants to provide a smooth and secure login experience for students.
- System Administrator: Wants to ensure the security of credentials and the authentication system.

Precondition

- The student must have an existing account with valid credentials (studentID and password).
- The student's mobile device must be connected to the Internet.

Minimum Guarantees

- The system logs every login attempt, whether successful or unsuccessful.
- The unauthenticated student cannot access the protected features of the application.

Success Guarantees

- The student is successfully authenticated and can access the application's features.

- The system logs the successful login attempt for auditing purposes.

Trigger

- The student launches the mobile application and initiates the login process.

Main Success Scenario

1. The student login with credentials (studentID and password).
2. The system verifies the student's credentials against those stored in the authentication system.
3. The system authenticates the student and loads the main interface of the application.
4. The student can now access the application's features.

Extensions

2a Incorrect studentID or password:

- The system detects that the studentID or password entered do not match the stored credentials.
- The system displays an error message and allows the student to re-enter the correct credentials.
- Return to step 1 of the Main Success Scenario

1a The student selects the "Forgotten password" option.

- The system guides the student through the password recovery process via a link sent to the registered email address.
- The student follows the instructions received to reset the password.
- The system confirms the password change and allows the student to log in with the new password.
- Return to step 1 of the Main Success Scenario

1b No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 1 of the Main Success Scenario.

I test sono creati a partire dalle seguenti narrative dei casi d'uso:

- login
- ricerca luoghi
- visualizzazione lezioni settimanali
- visualizzazione informazioni corso
- prenotazione esami
- caricamento elaborato
- prenotazione aule studio
- creazione ticket
- visualizzazione offerte di lavoro
- visualizzazione offerta formativa

5.2.2 Pagina

Il progetto usa il *Page Object Pattern*, ovvero un modello di programmazione che si adatta bene al caso in esame. Questo schema fa parte dei cosiddetti *design pattern* e rappresenta una soluzione standard per risolvere problemi comuni nello sviluppo software. Il pattern prevede l'implementazione di classi, denominate pagine, che rappresentano le diverse schermate dell'AUT. Ogni pagina contiene metodi specifici per eseguire azioni sull'interfaccia utente. I test automatizzati si avvalgono di questi metodi per interagire con l'AUT, garantendo una chiara separazione tra la logica dei test e il codice relativo all'interfaccia. Il principale vantaggio di questo approccio risiede nella sua facilità di manutenzione: in caso di modifiche all'interfaccia dell'AUT, è sufficiente aggiornare il codice della relativa pagina, senza la necessità di intervenire direttamente sui singoli test. Questo riduce significativamente la duplicazione del codice, poiché i

metodi che gestiscono l'interazione con l'interfaccia sono centralizzati in un'unica classe, evitando la loro ripetizione all'interno dei vari test [12]. Sono state definite 24 pagine con un numero di righe compreso tra 10 e 50, a seconda della complessità della schermata. Tutte le pagine derivano da una classe base detta Driver, che definisce i metodi per la ricerca degli elementi nell'interfaccia, sfruttando i selettori di WebDriverIO. Di conseguenza, le altre pagine ereditano tutti i suoi metodi e li utilizzano per individuare e interagire con i componenti dell'interfaccia. La figura 5.2 mostra i 4 metodi per la ricerca dei componenti. Ogni funzione utilizza un approccio diverso per la ricerca e accetta come parametro una stringa, che rappresenta un selettore specifico.

```
export default class Driver {

  public async findByAccessibilityId(accessibilityId: string) {
    return await $(`~${accessibilityId}`);
  }

  public async findById(id: string) {
    return await $(`id:${id}`);
  }

  public async findByUiSelector(uiSelector: string) {
    return await $(`android=${uiSelector}`);
  }

  public async findByXpath(xpath: string) {
    return await $(`${xpath}`);
  }

}
```

Figura 5.2. Driver page

Nella tabella 5.1 sono riportati i metodi di ricerca del driver, suddivisi per tipologia.
Come riportato nella documentazione di WebDriverIO, ogni selettore effettua la ricerca in modo diverso e ha delle tempistiche differenti. La ricerca per *accessibility id* e *id* individua l'elemento in base all'attributo omonimo. Sono i metodi migliori per la ricerca del componente per

Ricerca per	Funzione di ricerca
Accessibility ID	$('~Login')
ID	$('id:android:id/login')
UiAutomator	$('android=new UiSelector().description("Login")')
XPath	$('//android.widget.Button[@content-desc="Login"]')

Tabella 5.1. Metodi di ricerca del driver

tempistiche e perché individuano univocamente un singolo elemento all'interno della GUI. La documentazione privilegia l'uso della ricerca tramite accessibility id, poiché garantisce un livello di accessibilità che rende l'applicazione inclusiva e utilizzabile da chiunque. La ricerca per UiAutomator individua l'elemento in base ad alcune proprietà, ovvero la classe (className), il testo (text) e la descrizione (content-desc) dell'elemento. Con className si intende la classe dell'elemento come *android.widget.Button*. Il testo e la descrizione sono due attributi del componente. È possibile creare un selettore più specifico che utilizza più attributi contemporaneamente (es. className e text, className e content-desc). La tabella 5.2 mostra le tre tipologie di ricerca per UiAutomator.

La ricerca per XPath è la meno consigliata per le sue tempistiche e per la sensibilità alle modifiche all'interfaccia dell'app. La sostituzione di un componente con un altro può causare il fallimento della ricerca dell'elemento e per estensione del test. Questo tipo di ricerca individua gli elementi basandosi sulla loro posizione nella gerarchia della struttura XML. Un'espressione XPath rappresenta un percorso che punta a uno o più elementi in un documento. Come indicato dalla tabella 5.3, il percorso specificato può essere assoluto o relativo.

className	new UiSelector() .className("android.view.ViewGroup").instance(12)
text	new UiSelector().text("Student ID")
content-desc	new UiSelector().description("Enter your username here")

Tabella 5.2. Ricerca con UiSelector

percorso assoluto	/android.widget.FrameLayout /android.widget.LinearLayout /android.widget.Button[@text="Conferma"]
percorso relativo	//android.widget.Button[@text="Conferma"]

Tabella 5.3. XPath

Esempio pagina

La pagina *createTicket* (Figura 5.3) definisce i metodi necessari per la creazione e l'invio di un ticket.
Il metodo *selectTopic* (Figura 5.4) della pagina *createTicket* realizza la selezione di uno specifico argomento per il ticket, in base a un parametro passato come argomento della funzione. Il metodo esegue varie azioni. Cerca il componente dell'interfaccia, attraverso la funzione *findByUiSelectorText()*. Una volta trovato il componente, esegue un click su di esso (*topicElement.click()*), in modo da attivare il menu a tendina. Le due operazioni, appena descritte, sono ripetute anche per le opzioni del menu e sono necessarie per selezionare un'opzione corrispondente all'argomento del metodo.
Le uniche funzioni leggermente diverse della pagina *createTicket* sono la *writeTicket* e *writeTicketSubject* perché permettono l'inserimento del testo

```
import Driver from "../driver";
import data from "../../../../strings/index";

class CreateTicketPage extends Driver {
  Tabnine | Edit | Test | Explain | Document | Ask
  async selectTopic(topicName: string) {
    const topic = data["createTicketScreen"]["topicDropdownLabel"];
    const topicElement = await this.findByUiSelectorText(topic);
    await topicElement.click();

    const topicOptionElement = await this.findByUiSelectorText(topicName);
    await topicOptionElement.click();
  }
  Tabnine | Edit | Test | Explain | Document | Ask
  async selectSubTopic(subTopicName: string) {
    const subTopic = data["createTicketScreen"]["subtopicDropdownLabel"];
    const subTopicElement = await this.findByUiSelectorText(subTopic);
    await subTopicElement.click();

    const subTopicOptionElement = await this.findByUiSelectorText(subTopicName);
    await subTopicOptionElement.click();
  }
  Tabnine | Edit | Test | Explain | Document | Ask
  async writeTicketSubject(subject: string) {
    const subjectTicket = data["createTicketScreen"]["subjectLabel"];
    const subjectTicketElement = await this.findByUiSelectorText(subjectTicket);
    await subjectTicketElement.setValue(subject);
  }
  Tabnine | Edit | Test | Explain | Document | Ask
  async writeTicket(text: string) {
    const reply = data["ticketScreen"]["reply"];
    const replyTicket = await this.findByUiSelectorText(reply);
    await replyTicket.setValue(text);
  }
  Tabnine | Edit | Test | Explain | Document | Ask
  async sendTicket() {
    const sendTicket = data["createTicketScreen"]["sendTicket"];
    const sendTicketBtn = await this.findByAccessibilityId(sendTicket);
    await sendTicketBtn.click();
  }
}

export default new CreateTicketPage();
```

Figura 5.3. Ticket Page

dell'utente in componenti detti, TextArea. Come prima, i metodi individuano l'elemento in cui scrivere e utilizzano la funzione *setValue* per

Figura 5.4. Funzione Select Topic

inserire il contenuto nell'area.

5.2.3 Test

Ogni test[1] riproduce lo scenario principale dei casi d'uso, simulando l'interazione dell'utente con l'interfaccia, al fine di garantire il corretto funzionamento dell'app.
I test si dividono in 2 categorie principali:

- La prima verifica la correttezza delle informazioni già presenti nell'AUT (es. ricerca luogo, visualizzazione lezioni settimanali, ecc.).
- La seconda controlla il corretto inserimento di nuove informazioni da parte dell'utente (es. login, creazione ticket, prenotazione aule studio e esami, caricamento elaborati).

Ogni test svolge le seguenti azioni:

- **Navigazione** (Figura 5.5): è una attività necessaria per raggiungere la pagina dove si svolge il test. Si verifica nei blocchi before e beforeEach e costituisce una fase preliminare all'esecuzione del test vero e proprio. La navigazione sfrutta i metodi definiti nelle pagine.

[1] test progetto github

```
beforeEach(async () => {

  await BottomBar.navigateToServicesPage();

  await ServicesPage.navigateToOffering();

});
```

Figura 5.5. Fase 1: Navigazione

- **Esecuzione** (Figura 5.6): azioni nell'interfaccia utente per ottenere i risultati previsti (es. invio ticket, prenotazioni esami, ecc.). Le interazioni con i componenti sono definite nei metodi delle pagine.
- **Verifica** (Figura 5.7): confronto dei risultati ottenuti dall'interfaccia con quelli ricavati dalle API. Dalla verifica si ottengono tre risultati diversi, test fallito, test eseguito con successo, test saltato.

Esempio test

Il test dedicato all'invio del ticket[2] sfrutta i metodi definiti nella pagina *createTicket*[3] e rappresenta lo scenario principale del caso d'uso.
Le immagini 5.8, 5.9 e 5.10 mostrano il caso d'uso, il test sviluppato dal caso d'uso e infine la simulazione automatizzata dell'esperienza utente.
Esiste una certa corrispondenza tra lo scenario principale del caso d'uso e il test (Figura 5.8 e 5.9). La correlazione considera unicamente le fasi relative alle azioni dell'utente, ovvero il primo, il terzo e il quinto punto dell'elenco, perché il test simula un'interazione manuale. La prima fase riguarda la navigazione nella pagina di creazione del ticket ed è rappresentata dal blocco *beforeEach* del test. Ogni funzione invocata in questo blocco si limita a individuare e interagire con elementi specifici nell'interfaccia dell'app, che implementano la navigazione verso le pagine successive. La

[2] test invio ticket

[3] pagina createTicket

```
it('check first bachelor', async () => {
  const degreeName = bachelor.degrees[index].name;

  await OfferingPage.selectFirstBachelor(degreeName);

  const degreeNameElement = await DegreePage.findByUiSelectorText(degreeName);
  const degreeNameText = await degreeNameElement.getText();

  const info = data.degreeScreen.info;
  const infoElement = await DegreePage.findByUiSelectorText(info);
  const infoText = await infoElement.getText();

  const degreeId = await bachelor.degrees[index].id;
  const degree = await offeringApi.getOfferingDegree({
    degreeId,
    year,
  });
  const degreeData = await degree.data;

  const locationValue = await degreeData.location;
  const locationString = await `${location}: ${locationValue}`;
  const locationElement = await DegreePage.findByUiSelectorText(
    locationString,
  );
  const locationText = await locationElement.getText();

  const departmentValue = await degreeData.department.name;
  const departmentString = await `${department}: ${departmentValue}`;
  const deparmentElement = await DegreePage.findByUiSelectorText(
    departmentString,
  );
  const departmentText = await deparmentElement.getText();

  const facultyValue = await degreeData.faculty.name;
  const facultyString = await `${faculty}: ${facultyValue}`;
  const facultyElement = await DegreePage.findByUiSelectorText(facultyString);
  const facultyText = await facultyElement.getText();

  const durationValue = await degreeData.duration;
  const durationString = await `${duration}: ${durationValue}`;
  const durationElement = await DegreePage.findByUiSelectorText(
    durationString,
  );
  const durationText = await durationElement.getText();

  const className = await degreeData._class.name;
  const classCode = await degreeData._class.code;
  const codeString = await `${degreeClass}: ${className} (${classCode})`;
  const codeElement = await DegreePage.findByUiSelectorText(codeString);
  const codeText = await codeElement.getText();
```

Figura 5.6. Fase 2: Esecuzione

seconda fase corrisponde alla scrittura e all'invio del ticket. Queste due operazioni avvengono solo a fronte della selezione di argomento, sottoargomento e della scrittura dell'oggetto del ticket e sono rappresentate

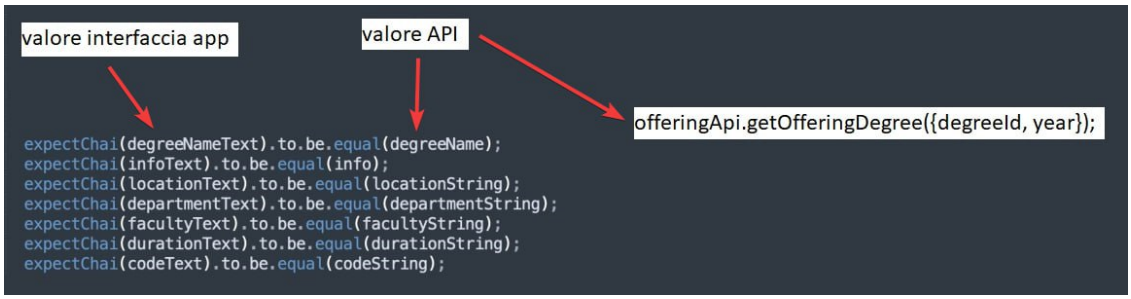

Figura 5.7. Fase 3: Verifica

Main Success Scenario

1. The student navigates ticket section
2. The system shows option for open new ticket
3. The student writes and sends ticket
4. The system elaborates and open ticket
5. The student sees new opened ticket in ticket section

Figura 5.8. Scenario principale caso d'uso creazione ticket

dalla funzioni omonime. Infine l'ultimo passaggio riguarda la possibilità dell'utente di vedere il ticket appena creato. L'ultima fase corrisponde alla parte cruciale del test che verifica il successo del caso d'uso ed è descritto nelle ultime righe, successive all'invio del ticket. Il codice ottiene le informazioni del ticket appena creato nella GUI e le confronta con quelle inserite durante il processo di creazione, se sono uguali, il processo è andato a buon fine, altrimenti si è verificato un errore.

Lo studio ha effettuato complessivamente 12 test per 10 casi d'uso. I test hanno una lunghezza compresa tra 50 e 150 righe di codice. Il numero e la lunghezza dei test risultano adeguati alla complessità dell'AUT, in quanto i

```
describe('Send ticket use case', () => {

  const ticketApi = new TicketsApi();

  beforeEach(async () => {
    await BottomBar.navigateToServicesPage();

    await ServicesPage.navigateToTicket();

    await TicketPage.navigateToFaqs();

    await FaqsPage.searchFaq('abc');

    await FaqsPage.navigateToWriteTicket();
  });

  it('successfully send ticket use case', async function (this: Mocha.Context) {
    const topics = await ticketApi.getTicketTopics();
    const topicData = await topics.data[0];

    if (topicData.subtopics.length === 0) {
      this.skip();
    }

    await CreateTicketPage.selectTopic(topicData.name);

    await CreateTicketPage.selectSubTopic(topicData.subtopics[0].name);

    const subject = `Test Ticket ${setup.timestamp}`;
    await CreateTicketPage.writeTicketSubject(subject);

    const text = `test write ticket ${setup.timestamp}`;
    await CreateTicketPage.writeTicket(text);

    await CreateTicketPage.sendTicket();

    const subjectTextElement = await TicketPage.findByUiSelectorText(subject);
    const subjectText = await subjectTextElement.getText();

    const textTicketElement = await TicketPage.findByUiSelectorText(text);
    const textTicket = await textTicketElement.getText();

    expectChai(subjectText).to.be.equal(subject);
    expectChai(textTicket).to.be.equal(text);
  });

});
```

Figura 5.9. Create ticket test

10 casi d'uso coprono gran parte delle funzionalità dell'app. Ogni test verifica almeno lo scenario principale del relativo caso d'uso. La differenza tra il numero di test e il numero dei casi d'uso è spiegata dalla presenza di due test aggiuntivi: la ricerca dei luoghi senza risultati e la visualizzazione dell'offerta formativa delle lauree magistrali. Le ragioni nell'introduzione dei due test sono da ricercare nell'interfaccia dell'app e nell'estensione dei

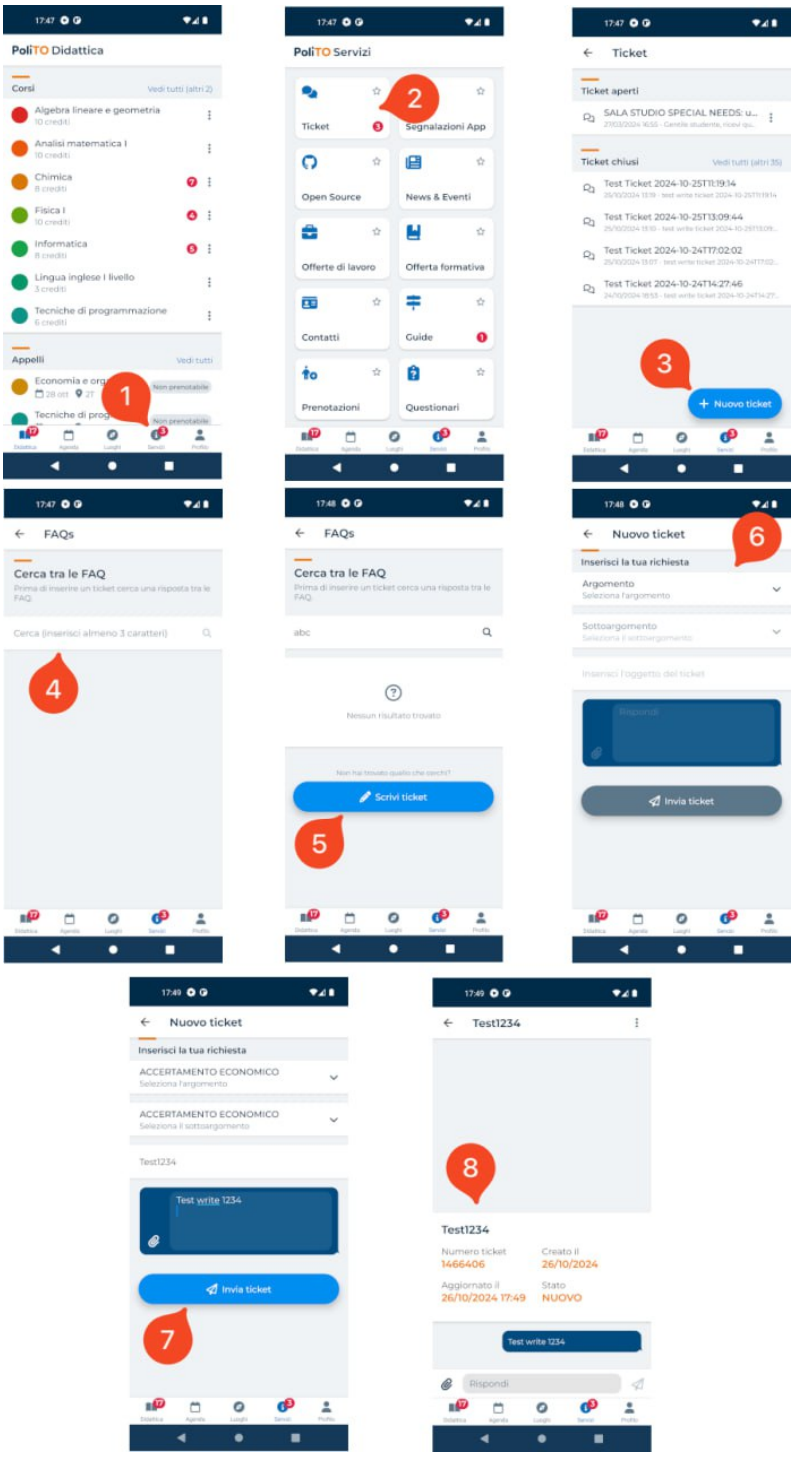

Figura 5.10. Esecuzione creazione ticket test su Polito Students app

casi d'uso. La GUI dell'app per le offerte formative è divisa in due sezioni, una per le lauree triennali e l'altra per le magistrali. Di conseguenza, il file di test è suddiviso in due parti, ciascuna dedicata alla verifica delle informazioni di uno dei due percorsi di studio.
La ricerca dei luoghi senza risultati è un scenario alternativo molto semplice da realizzare. Inoltre, a differenza dello scenario principale, questo test non presenta problemi legati alla lingua dei risultati delle API ed è quindi sempre eseguito.

Inoltre ogni test utilizza dati geografici e temporali:

- **Localizzazione**: è possibile eseguire i test in diverse lingue, perché vengono utilizzati file specifici per ciascuna lingua (es. it.json, en.json).
- **Fuso orario**: i test usano il fuso orario *Europe/Rome*, a causa di alcuni problemi riscontrati nell'app.

Capitolo 6

Integrazione dei test in una pipeline CI/CD

L'acronimo sta per Continuous Integration and Continuous Delivery/Deployment e si riferisce alle pratiche di integrazione e rilascio continuo del software. La CI/CD aiuta nel mantenimento di un ciclo continuo di sviluppo e aggiornamento del software e riduce la possibilità di rilasciare codice contenente bug. Con l'aumentare delle dimensioni delle applicazioni, le funzionalità di CI/CD possono aiutare a ridurre la complessità, aumentare l'efficienza e semplificare i flussi di lavoro. Il processo automatizza l'intervento manuale per il rilascio di nuovo codice in produzione, quindi minimizza il tempo di inattività e velocizza il rilascio del codice. La Figura 6.1 mostra il processo di Continuous Integration e Continuous Delivery/Deployment. La spiegazione dei concetti di CI/CD è tratta dall'articolo *What is CI/CD?* [10].

Figura 6.1. Pipeline CI/CD. Immagine tratta da [10]

6.1 Continuous Integration

La Continuous Integration è un processo automatizzato per gli sviluppatori che facilita l'integrazione di cambiamenti del codice all'interno di ambienti condivisi (repository), come Github. L'aggiornamento del codice determina delle fasi di test automatizzate per garantire l'affidabilità del codice inserito. Nello sviluppo delle applicazioni moderne, gli sviluppatori lavorano simultaneamente su diverse funzionalità della stessa app, in aree isolate dette branch. Occasionalmente è necessario unire questo nuovo codice proveniente da branch diversi nella versione principale del codice presente nel branch principale (main o master). L'unione del codice è una procedura che può diventare tediosa, manuale, lunga e soggetta a errori. Gli errori sono dovuti ai conflitti che possono emergere durante la procedura di unione. L'integrazione continua può costituire una soluzione al problema di avere troppi branch di un'app che vanno in conflitto l'uno con l'altro. Un processo di CI corretto prevede una validazione automatizzata del codice inserito. La validazione include la generazione dell'applicazione (build) e l'esecuzione di vari test, come unit test e integration test, per verificare che le modifiche non abbiano compromesso il funzionamento dell'app. Uno dei principali vantaggi dell'integrazione continua è la possibilità di individuare immediatamente eventuali bug nel nuovo codice.

6.2 Continuous Delivery

Il continuous delivery automatizza il rilascio di codice validato in un ambiente esecutivo, a seguito del build e dell'esecuzione dei test avvenuti nella fase di continuous integration. Questa procedura risulta davvero efficace solo se l'attività di CI è già implementata nella pipeline di sviluppo. Di solito in questa fase, il codice del programmatore viene automaticamente verificato e rilasciato in un ambiente di produzione dal team operativo. L'obiettivo è garantire che il codice sia sempre pronto per il rilascio, riducendo al minimo lo sforzo necessario per sviluppare nuove funzionalità.

6.3 Continuous Deployment

Lo stadio finale di una pipeline di CI/CD è il continuous deployment. È un'estensione del processo di delivery e rende le modifiche dell'ambiente di produzione disponibili agli utenti finali. Riduce il sovraccarico dei team

operativi derivante dalle operazioni manuali, velocizzando il processo di rilascio dell'applicazione.

6.4 Github Actions

Lo studio utilizza una pipeline di CI/CD per adottare le migliori strategie in termini di sviluppo del software, per integrarsi al progetto dell'applicazione e assicurare una maggiore affidabilità del software. In particolare i test sfruttano la pipeline di Github, ovvero le Github Actions. Le actions sono ormai indispensabili nei progetti in cui si vuole garantire una certa qualità durante il rilascio del software.
Le actions realizzano l'automazione dei flussi di lavoro (workflow) direttamente all'interno di un repository. Sono utili per eseguire attività come test, build, deployment, ogni volta che si verifica un evento specifico, come un commit, una pull request o il rilascio di una nuova versione. Ecco i concetti fondamentali delle GitHub Actions:

- **Workflow**: è una configurazione che definisce una sequenza di azioni da eseguire. È descritto in un file YAML che si trova nella directory *.github/workflows/.*
- **Events**: i workflow si attivano in base a eventi definiti nel file YAML, come push, pull_request, e tanti altri.
- **Jobs**: rappresenta un insieme di passaggi che vengono eseguiti in sequenza in un ambiente isolato. Ogni job può essere eseguito su una diversa macchina virtuale o container.
- **Steps**: sono i singoli comandi o azioni da eseguire all'interno di un job. Alcuni riguardano azioni predefinite come *actions/checkout* per clonare il repository sul runner, altri sono comandi personalizzati, ad esempio *npm install*.
- **Runner**: è la macchina (fisica o virtuale) su cui vengono eseguiti i job. Ogni runner utilizza una versione specifica del sistema operativo tra Linux, Windows e MacOS.

Il workflow replica le impostazioni locali necessarie all'automazione ed esegue i test appena si verifica un evento di push nel repository principale. La Figura 6.2 mostra le fasi del workflow eseguite su un runner, che utilizza l'ultima versione di Ubuntu.

Come si evince dalla Figura 6.2, gli step sono eseguiti in questo ordine:

1. Download del codice per la simulazione dei test automatizzati d'interfaccia.

2. Download dell'applicazione Polito Students.

3. Attivazione dell'accelerazione hardware, necessaria al funzionamento dell'emulatore Android su Linux.

4. Configurazione del client delle API tramite *Github token*. Il client è fondamentale per l'utilizzo delle API nei test di automazione.

5. Installazione di Java 21.

6. Installazione di Node.js 20.17.0 e dei suoi pacchetti (node_modules).

7. Salvataggio dei pacchetti Node.js. Questa operazione diminuisce i tempi delle simulazioni future, perché le esecuzioni successive non devono scaricare nuovamente i pacchetti, ma si limitano a importarli da un archivio.

8. Avvio dell'emulatore Android ed esecuzione dei test.

9. Creazione e caricamento del report dei test e di eventuali screenshot su Github. Gli screenshot sono caricati su Github solo in caso di fallimento dei test di automazione. Sono essenziali perché forniscono una panoramica dello stato dell'app nel momento in cui si verifica l'errore, rendendo più semplice la correzione del codice.

6.4.1 Risultati test di automazione nella pipeline di sviluppo

I tempi della simulazione completa tramite GitHub Actions possono essere suddivisi in diverse fasi, per mettere in evidenza le 3 operazioni principali eseguite nel workflow. Le fasi includono la configurazione iniziale, l'avvio dell'emulatore e l'esecuzione dei test. La configurazione iniziale fa riferimento all'installazione di tutte le dipendenze necessarie (Node.js, Java, ecc.) per l'esecuzione dei test. La tabella 6.1 mostra la durata delle tre fasi e il tempo totale di simulazione.

Tempo	**Valore**
Configurazione Github Action	37s
Avvio emulatore Android	3m 14s
Test	5m 55s
Simulazione	9m 46s

Tabella 6.1. Tempi simulazione

La durata della simulazione è legata ai parametri della configurazione delle Github Action e all'emulatore Android. Di seguito sono riportati i parametri con tutte le versioni di sistemi operativi, software e hardware usati.

Sistema operativo	Ubuntu 22.04 LTS
Versione Java	21
Versione Node.js	20.17.0
API level	34
Immagine Android	Default Android System Image
Architettura processore	x86_64
Dispositivo virtuale	Pixel 3A XL

Tabella 6.2. Parametri simulazione

L'unica strategia per ridurre il tempo di esecuzione complessivo prevede l'uso dell'azione *actions/cache*, che abbrevia la durata del processo di configurazione di circa 30 secondi. Altre modifiche alla fase di configurazione non sono state valutate, in quanto la durata è già

significativamente ridotta rispetto al tempo complessivo della simulazione. (configurazione / simulazione = 37s / 9m 46s). I tempi richiesti per l'esecuzione dei test e l'avvio dell'emulatore non possono essere ridotti a causa della complessità dell'automazione e dello stesso emulatore.

Nella Figura 6.3 sono mostrati i risultati e le tempistiche dei test all'interno della pipeline di Github. L'esecuzione dei test ha una durata accettabile, considerando la durata per l'avvio dell'emulatore Android e la complessità dei test eseguiti. L'immagine 6.3 evidenzia due test non eseguiti: la ricerca dei luoghi e la prenotazione degli esami, a causa di impostazioni errate della lingua e della finestra temporale.
I test con una durata maggiore rispetto a tutti gli altri sono:

- **Login**: deve chiudere una schermata informativa subito dopo aver effettuato l'accesso. La scheda impiega molto tempo a caricarsi perché deve elaborare e visualizzare un video.
- **Ricerca dei luoghi**: impiega molto tempo per il caricamento della mappa.
- **Invio del ticket**: richiede una lunga fase di navigazione e un'interazione non banale nella fase di scrittura e caricamento del ticket.
- **Caricamento dell'elaborato**: esegue la scansione e l'invio del documento tramite la fotocamera. L'interazione dell'app con la fotocamera costituisce un'operazione che richiede molto tempo soprattutto su un dispositivo virtuale.

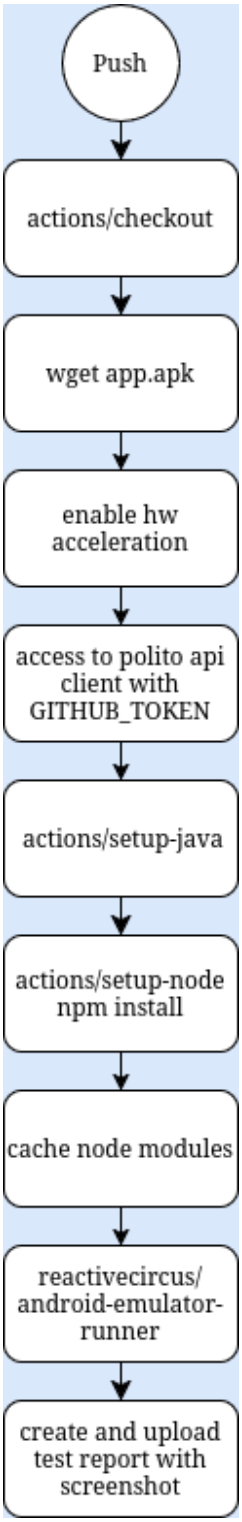

Figura 6.2. Github Actions workflow

Detailed Test Results

Name	Status	ms	Flaky
check first bachelor	passed	14027	
check first master	passed	7243	
check first job offer	passed	19434	
successfully booking study room special needs	passed	11304	
successfully send ticket	passed	35344	
check weekly lectures	passed	16177	
successfully login	passed	43458	
successfully search place	skipped	0	
no place found	passed	37945	
upload assignment with scan file option	passed	48612	
successfully book exam	skipped	0	
check course info page	passed	12925	

Tests	Passed	Failed	Skipped	Pending	Other	Flaky	Duration
12	10	0	2	0	0	0	00:05:55

Figura 6.3. Risultati test

Capitolo 7

Limitazioni dell'approccio

Il capitolo affronta le problematiche emerse durante lo sviluppo del progetto legate allo strumento e all'ambiente di test.

7.1 Appium

Di seguito sono elencate alcune problematiche dello strumento riscontrate direttamente nell'attività di testing.

- **Overhead di risorse**: l'uso di Appium richiede un livello significativo di risorse del sistema, il che può rallentare il computer su cui è in esecuzione.

- **Interazioni avanzate**: Appium ha delle difficoltà nel gestire gesture complesse o funzionalità specifiche delle applicazioni native, come le notifiche push o le animazioni. Di conseguenza i test non ricorrono a nessuna funzionalità nativa e usano soltanto gesture semplici come lo *swipe*. Altre interazioni come il *pinch to zoom* sono più difficili da realizzare.

- **Supporto per iOS**: la configurazione per l'automazione delle app iOS richiede Xcode, certificati di provisioning, e configurazioni complesse a causa delle restrizioni imposte da Apple.

- **Test cross-platform**: anche se Appium è progettato per supportare Android e iOS, ci possono essere differenze significative nella configurazione e nelle API disponibili per ciascuna piattaforma. Quindi i test possono variare tra una piattaforma e l'altra.
  Il problema appena esposto dipende principalmente dalla tecnologia di automazione.

- **Log poco chiari**: i messaggi di log generati da Appium non sono sempre chiari, specialmente nelle pipeline di sviluppo, il che può rendere il debugging più complesso e dispendioso in termini di tempo.

- **Controllo visuale**: Appium non è ideale per test basati sulla verifica di elementi grafici o layout complessi. Non offre funzionalità avanzate di confronto visivo.

- **Dispositivi multipli**: configurare e gestire test paralleli su dispositivi multipli richiede risorse aggiuntive e una configurazione accurata, soprattutto per evitare conflitti con porte e sessioni.

- **Limitazioni dei driver di automazione**: Appium si basa su driver specifici per Android (UIAutomator2) e iOS (XCUITest). Questi ultimi non espongono delle informazioni dettagliate sugli elementi dell'interfaccia. Ad esempio non è possibile ottenere informazioni riguardo le immagini, il colore dei widget e tanti altri.
  Il comportamento descritto è mostrato dalle proprietà esposte da Appium Inspector nell'immagine 7.1.

Attribute	Value
elementId	00000000-0000-0009-ffff-ffff0000007
index	1
package	it.polito.students
class	android.view.View
text	Teaching
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
long-clickable	false
password	false
scrollable	false
selected	false
bounds	[218,97][451,162]
displayed	true

Figura 7.1. Proprietà elementi UI

7.2 Test

Lo sviluppo del progetto ha richiesto sia interventi comuni a più test o all'ambiente di test in generale, sia interventi mirati per i singoli test, data la complessità delle viste e il comportamento di Appium.

7.2.1 Navigazione schermate app

Quasi tutti i test richiedono una fase di navigazione per raggiungere la schermata dove si svolge il test. Il processo di navigazione è mostrato nelle Figure 5.10 e 5.5. L'accesso alle varie sezioni dell'app è richiesto dato che l'app non implementa i deep link. Questi ultimi costituiscono un metodo per avviare l'AUT direttamente in una pagina specifica. Risultano molto utili nella fase di test per snellire il codice e per ridurre i tempi di esecuzione della verifica.

7.2.2 Estensioni casi d'uso

Alcuni dei test presentano del codice in più rispetto a quello richiesto per eseguire lo specifico caso d'uso. Questo è dovuto all'impossibilità di disporre di un ambiente di test completamente configurabile. Quindi per evitare eventuali errori nell'esecuzione ripetuta dell'intera suite di test, si è deciso di aggiungere porzioni di codice per saltare i test, nel caso in cui le condizioni attuali fanno riferimento a uno scenario diverso da quello principale. Queste sezioni di codice costituiscono in alcuni casi degli scenari alternativi a quello principale, dette anche estensioni. Un esempio di quanto descritto è presente nel test di creazione del ticket (Figura 7.2). Si verifica se l'utente vuole creare due ticket con lo stesso argomento e sottoargomento. Il sistema impedisce la creazione del secondo ticket e l'esecuzione del test è saltata (Figura 7.3). Ovviamente nei suddetti scenari, i reporter mostrano lo stato *test skipped*.
Le estensioni non sono implementate sotto forma di test alternativi per due motivi:

- Configurazione iniziale dell'ambiente di test non disponibile.
- Mantenere i tempi di esecuzione bassi.

Le estensioni non sono presenti in tutti i test e in ogni caso esistono poche varianti del caso d'uso principale. Nonostante l'aggiunta delle derivazioni nello scenario principale, il codice risulta compatto e il test è più robusto ad eventuali modifiche dell'ambiente di test.
Le Figure 7.2 e 7.3 mostrano l'estensione precedentemente descritta e il relativo codice per la creazione di un ticket.

Extensions

3a Create two ticket with same information:

- The system prevents the user from creating a new ticket.
- The students selects an other option for new ticket
- Return to step 3 of the Main Success Scenario.

Figura 7.2. Estensione creazione ticket

```
if (topicData.subtopics.length === 0) {
  this.skip();
}
```

Figura 7.3. Codice estensione creazione ticket

7.2.3 Isolamento

Ogni test inizializza una nuova sessione di WebdriverIO e termina l'AUT in modo da garantire un certo isolamento tra una verifica e l'altra. Questo comportamento evita alcune problematiche, come la fragilità dei test. Nonostante gli accorgimenti appena citati, non è stato possibile garantire un isolamento totale per due motivi:

- Impossibilità di fornire un token di autenticazione e caricare l'AUT direttamente nella schermata principale. Quest'ultima è disponibile soltanto in seguito al login.

- Riduzione dei tempi di esecuzione incompatibili in contesti di Continuous Integration and Continuous Deployment.

La maggior parte dei test richiede una sessione autenticata per poter accedere alle funzionalità dell'app e eseguire i vari casi d'uso. Per questa ragione il test di login viene eseguito per primo e poi tutti gli altri a seguire. L'esecuzione dei test in questo ordine è garantita da una specifica sequenza indicata nel file di configurazione di WebDriverIO (Figura 7.4).

```
const config = {
  ...baseConfig,

  specs: [
    "../tests/specs/android.spec.login.ts",
    "../tests/specs/android.spec.place.ts",
    "../tests/specs/android.spec.lectures.ts",

    //TEACHING
    "../tests/specs/teaching_page/android.spec.course.ts",
    "../tests/specs/teaching_page/android.spec.book_exam.ts",
    "../tests/specs/teaching_page/android.spec.assignment.ts",

    //SERVICES
    "../tests/specs/services_page/android.spec.room.ts",
    "../tests/specs/services_page/android.spec.ticket.ts",
    "../tests/specs/services_page/android.spec.job_offering.ts",
    "../tests/specs/services_page/android.spec.degree.ts",
  ],
```

Figura 7.4. Suite di test

Nonostante l'approccio risulti formalmente errato, l'esecuzione corretta dei test richiederebbe un reset dell'app e un login per ogni test che accede al portale oppure la possibilità di saltare il login tramite il token di autenticazione. Dato che la seconda opzione non è disponibile, la prima risulta corretta, ma non ottimale in termini di velocità per la pipeline delle Github Actions.

7.2.4 Versioni Android

I test sviluppati con questa tecnologia risentono molto delle prestazioni generali dell'ambiente in cui si sviluppa. Il processo di testing in locale usa un hardware non molto prestante e una versione vecchia di Android, ovvero la 11.0. Questo perché l'emulatore Android costituisce un sistema molto pesante in cui eseguire le verifiche. Di conseguenza l'esecuzione senza errori dei test richiede delle funzioni come *pause()* necessarie per la completa elaborazione delle schermate dell'app.

Alcune proprietà come i timeout della sessione, relativi al file di configurazione di WebDriverIO, sono stati allungati per garantire la corretta esecuzione dei test.
I test sfruttano delle immagini di sistema con diverse configurazioni di Android su dispositivi emulati e fisici (Tabella 7.1 e Figura 7.5).

Release Name	**API level**	**Target**
R	30	Android 11.0 (Google APIs)
R	30	Android 11.0 (Default)
UpsideDownCake	34	Android 14.0 (Default)

Tabella 7.1. Versioni di Android per immagini di sistema

Lo sviluppo dei test in locale utilizza la versione di Android 11 con e senza le Google APIs.
Le Github Actions sfruttano la versione 14 senza Google APIs, perché risulta più leggera, veloce e meno problematica nell'esecuzione dei test della sua controparte.

7.2.5 Test lezioni settimanali

Nel test riguardante le lezioni settimanali[1], alcuni elementi grafici non sono visibili. Questi elementi sono stati esclusi dal test perché lo strumento non fornisce identificativi validi nell'interfaccia. I componenti ignorati fanno riferimento a lezioni tenute nello stesso giorno e nella stessa ora (Figura 7.6).

7.2.6 Test prenotazione aule studio

Un altro test molto sensibile al funzionamento di Appium riguarda la prenotazione delle aule studio[2]. Sono presenti due problemi per questo test. Uno riguarda gli elementi della vista che sono invisibili a meno di utilizzare

[1] test lezioni settimanali

[2] test prenotazione aule studio

Figura 7.5. Dispositivi fisici e virtuali. Immagine tratta da [16]

azioni complesse come le Gestures. Il secondo è collegato al primo e riguarda il comportamento di Appium per gli elementi non presenti a schermo. Lo strumento non carica questi componenti, quindi risulta impossibile individuarli. Di conseguenza il test esegue la prenotazione delle aule studio in tutti le fasce orarie, tranne l'ultima (Figura 7.7).
Un'ulteriore criticità riguarda il corretto render dell'interfaccia in base al fuso orario impostato. Quindi il progetto configura il fuso orario *Europe/Rome*, prima dell'esecuzione della suite di test.

7.2.7 Test prenotazione esami

Il test per la prenotazione degli esami[3] funziona perfettamente, ma dipende dalla data in cui si esegue. Il test eseguito in una certa finestra temporale,

[3] test prenotazione esami

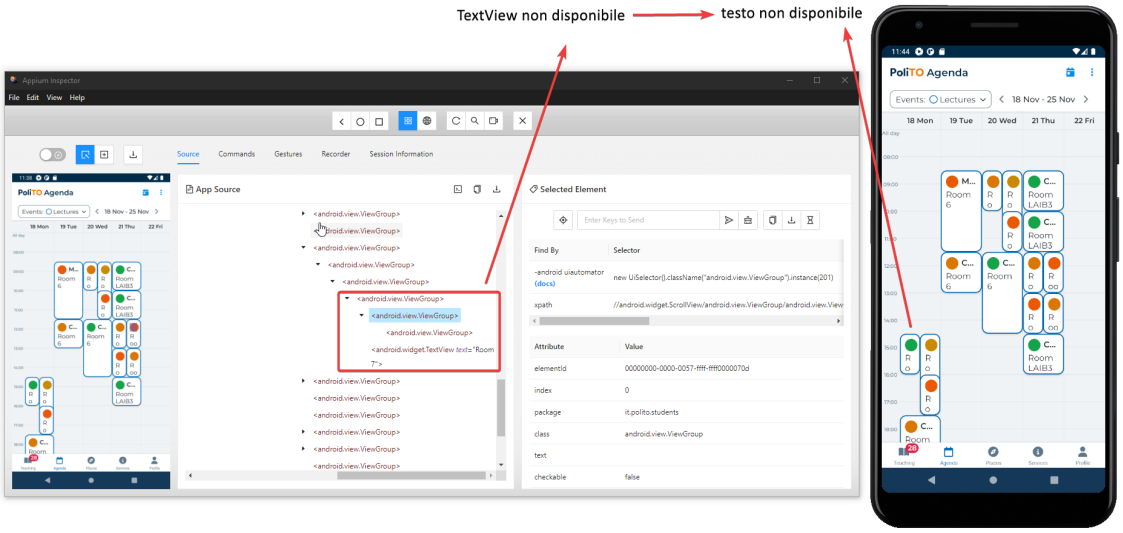

Figura 7.6. Vista lezioni settimanali

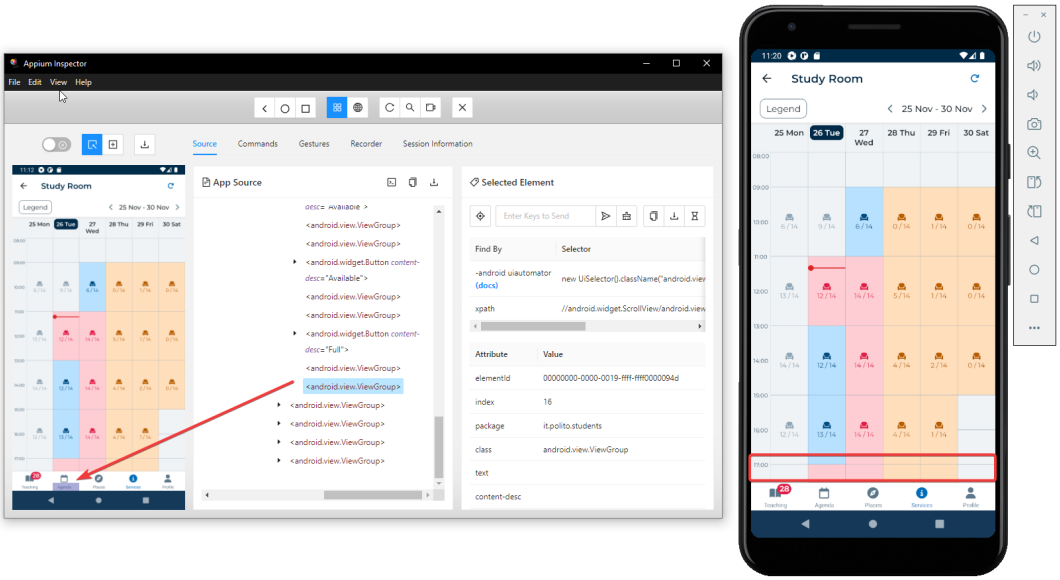

Figura 7.7. Vista prenotazioni aule studio

ha successo, altrimenti è semplicemente ignorato e quindi viene contrassegnato dallo stato *skipped*. La causa di questo comportamento è legato alla disponibilità dei dati in una finestra temporale specifica. Di

conseguenza, l'utilità del test è ridotta a causa della sua esecuzione intermittente. Il problema appena trattato è rappresentato nella Figura 7.8.

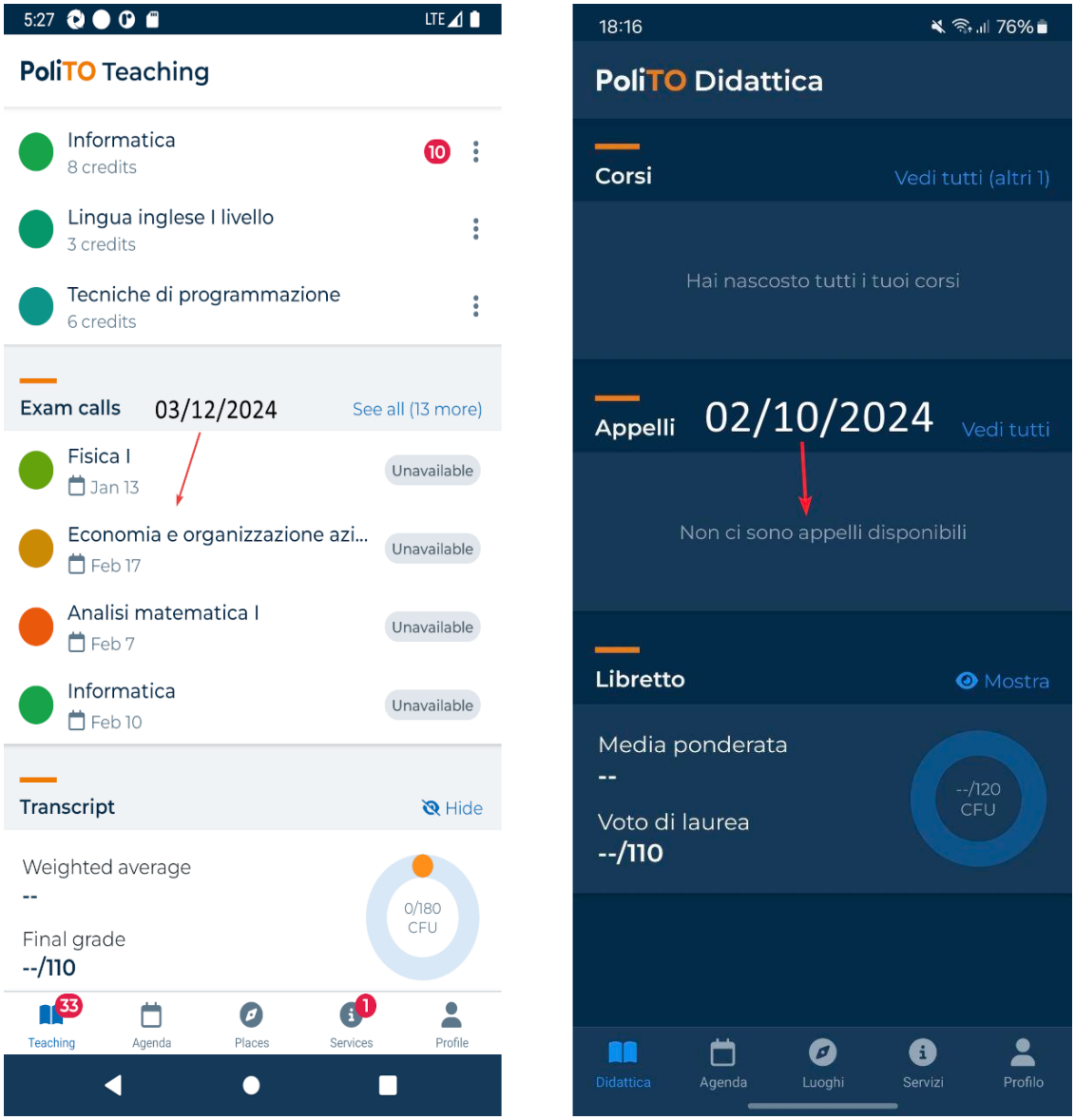

Figura 7.8. Disponibilità prenotazione esami

La capacità di configurare l'ambiente di test in modo flessibile è necessaria per verificare la funzionalità delle prenotazioni esami. In particolare sono necessari i seguenti elementi:

- Un utente di test con dati fittizi per le sessioni d'esame.

- Alcune API in grado di restituire e modificare i dati fittizi.

Naturalmente, gli elementi indicati devono essere sempre disponibili. In questo modo, il test potrebbe effettuare la prenotazione degli esami, poiché gli appelli sarebbero accessibili nell'interfaccia.

7.2.8 Test ricerca luoghi

Il test relativo alla ricerca dei luoghi[4] presenta problemi di fragilità. Questo si verifica quando si utilizzano le versioni di Android senza le Google APIs e l'opzione dell'emulatore, che disabilità le animazioni.
Il problema riguarda l'interfaccia dell'AUT, dato che si verificano problemi con le animazioni legate alla mappa e il render della barra di ricerca. Senza quest'ultima è impossibile eseguire il test di ricerca. Inoltre non è possibile nemmeno ricorrere a soluzioni alternative che interagiscono direttamente con i luoghi presenti sulla mappa, perché troppo complesse e legate alle dimensioni dei dispositivi.
Il test presenta un altro problema legato alla localizzazione. È presente un'incompatibilità tra la lingua dei risultati ottenuti dalle API e la lingua dei componenti dell'interfaccia. Questo comportamento si verifica unicamente con la lingua italiana. Il test è quindi ignorato in italiano e eseguito in inglese.

[4] test ricerca luoghi

Capitolo 8

Conclusioni

Questo lavoro di tesi si focalizza sulla progettazione e sull'implementazione di test di automazione dell'interfaccia utente per l'applicazione degli studenti del Politecnico di Torino, con l'obiettivo di migliorare l'affidabilità e l'efficienza del processo di verifica del software. I test di automazione dell'interfaccia consentono di simulare e automatizzare le interazioni tipiche dell'utente finale con l'AUT, riducendo così il margine di errore umano e velocizzando il processo di testing. L'approccio adottato in questa tesi prevede l'uso della GUI per eseguire una serie di operazioni automatizzate che replicano i comportamenti degli utenti reali. Questi test sono stati progettati per coprire vari scenari d'uso dell'AUT, al fine di garantire un'ampia copertura funzionale e rilevare eventuali bug o malfunzionamenti. L'obiettivo principale è verificare che l'AUT risponda correttamente alle diverse interazioni, mantenendo un comportamento stabile e coerente in tutte le sue funzionalità.
L'automazione dei test è stata resa possibile grazie all'utilizzo del framework Appium. Quest'ultimo è stato configurato per interagire con l'interfaccia grafica dell'AUT, permettendo la generazione di una suite di test automatizzati che coprono le funzionalità principali dell'AUT, come la navigazione tra le schermate, l'inserimento di dati nei moduli e la verifica delle risposte dell'AUT.
I test sviluppati non solo verificano il corretto funzionamento dell'AUT, ma contribuiscono anche a identificare rapidamente eventuali regressioni introdotte durante lo sviluppo. Questo è particolarmente importante in ambienti di sviluppo agili, dove le modifiche al codice sono frequenti e rapide. Per massimizzare l'efficacia del processo di testing, il progetto è stato integrato all'interno di una pipeline di Continuous Integration and

Continuous Deployment. L'integrazione dei test nella pipeline CI/CD consente di automatizzare l'intero processo di build, test e rilascio dell'AUT. Ogni modifica al codice sorgente attiva automaticamente l'esecuzione dei test, assicurando che eventuali errori vengano rilevati e corretti tempestivamente prima che il software venga distribuito agli utenti finali. Questo approccio non solo riduce il tempo necessario per il rilascio di nuove versioni dell'applicazione, ma aumenta anche la qualità del prodotto finale, garantendo un'applicazione più stabile e affidabile.
Il lavoro di tesi ha dimostrato come l'implementazione di test di automazione dell'interfaccia, supportata da strumenti come Appium e integrata in una pipeline CI/CD, possa significativamente migliorare l'efficienza del processo di sviluppo software. Questo approccio contribuisce a garantire il rilascio di un'applicazione funzionante e di alta qualità, riducendo al minimo il rischio di malfunzionamenti in produzione e aumentando la soddisfazione degli utenti finali.

8.1 Considerazioni

In conclusione, i test condotti hanno mostrato risultati promettenti, evidenziando il potenziale degli strumenti usati per l'automazione dei test su dispositivi mobili (Appium, WebDriverIO, Typescript). L'utilizzo delle tecnologie ha permesso la costruzione di un'architettura di test buona, nonostante alcune delle problematiche citate precedentemente riguardanti Appium e l'AUT. I risultati complessivi indicano che queste sfide possono essere risolte con aggiornamenti e migliorie dell'app e delle tecnologie usate. L'utilizzo di TypeScript, in particolare, ha contribuito a migliorare la robustezza del codice, grazie alla sua forte tipizzazione e agli strumenti di debugging avanzati, riducendo il rischio di errori a runtime. WebDriverIO ha dimostrato di essere una libreria versatile per la gestione di interfacce di test automatizzati, mentre Appium si è confermato una scelta eccellente per i test su Android.
In sintesi, questi strumenti, se configurati e utilizzati correttamente, rappresentano una soluzione potente per l'automazione dei test.

La tipologia di test presenta ancora molte sfide come:

- **Frammentazione**: l'unico modo per mitigare il problema della moltitudine di dispositivi con hardware e software diversi consiste nell'utilizzo di vari emulatori e dispositivi fisici.

- **Integrazione in una pipeline di sviluppo**: la configurazione del workflow per i test automatizzati può essere una sfida, soprattutto per quanto riguarda la gestione dei file di configurazione di Appium e l'uso di funzionalità indispensabili, come la cattura di screenshot in caso di errori. Un'ulteriore complessità è dovuta all'action dell'emulatore Android (ReactiveCircus/android-emulator-runner), che richiede una configurazione precisa per garantire un corretto funzionamento durante l'esecuzione dei test. Questi aspetti possono comportare numerosi tentativi prima di raggiungere la configurazione ottimale, con un conseguente aumento del tempo necessario per l'automazione. Di conseguenza si riduce l'efficacia delle soluzioni automatizzate, poiché la pipeline di CI/CD dovrebbe semplificare e velocizzare i processi, non aggiungere complessità e lunghe iterazioni di setup.
- **Analisi dei risultati e reporting**: la raccolta e la gestione dei report può risultare complicata e dispendiosa, soprattutto per test complessi su più dispositivi.

8.2 Sviluppi futuri

L'esperienza acquisita nel corso di questo progetto offre spunti preziosi per affrontare futuri sviluppi, con un focus particolare sulla risoluzione delle criticità elencate nelle sezioni precedenti riguardanti l'isolamento e la navigazione. Questi due miglioramenti garantiscono una maggiore correttezza e affidabilità dei test sviluppati, snelliscono il codice e garantiscono una riduzione dei tempi di esecuzione. Quindi sono sicuramente compatibili con le pipeline di CI/CD. Altri spunti per allargare e migliorare il progetto potrebbero riguardare i seguenti campi:

- **Frammentazione**: l'utilizzo di vari dispositivi permette lo sviluppo di app più robuste, affidabili e funzionali.
- **Visual Testing**: WebDriverIO offre un servizio di Visual Testing che insieme alla tipologia di test sviluppati in questo studio, completa i test sulla UI. Questo metodo consente di controllare la corretta visualizzazione dell'interfaccia confrontando immagini di riferimento con le immagini generate durante l'esecuzione dei test, identificando eventuali discrepanze visive.
- **Utilizzare runner locali per la pipeline di CI/CD**: è necessario per poter avere un controllo maggiore sull'esecuzione dei test, per

ridurre i costi associati all'impiego dei runner remoti e per riuscire a eseguire test su tanti dispositivi diversi.

Appendice A

Use Case Narratives

Di seguito sono riportate le Use Case Narrative utilizzate per progettare e documentare i test d'automazione svolti.

A.1 Login

Use Case: Login

Scope: Authentication System

Level: User goal

Intention In Context

- Student: A university student wants to securely access their academic records, course materials, and other educational resources through the mobile app.
- System: Must verify the student's identity and protect their personal and academic data from unauthorized access.

Primary Actor: Student

Support Actor

- Email Service: Sends notifications or links for password recovery.

- Authentication system: Manage verification of student credentials.

Stakeholders' Interests

- Registered student: Wants to quickly and securely access their personal information.
- Application Developer: Wants to provide a smooth and secure login experience for students.
- System Administrator: Wants to ensure the security of credentials and the authentication system.

Precondition

- The student must have an existing account with valid credentials (studentID and password).
- The student's mobile device must be connected to the Internet.

Minimum Guarantees

- The system logs every login attempt, whether successful or unsuccessful.
- The unauthenticated student cannot access the protected features of the application.

Success Guarantees

- The student is successfully authenticated and can access the application's features.
- The system logs the successful login attempt for auditing purposes.

Trigger

- The student launches the mobile application and initiates the login process.

Main Success Scenario

1. The student login with credentials (studentID and password).

2. The system verifies the student's credentials against those stored in the authentication system.
3. The system authenticates the student and loads the main interface of the application.
4. The student can now access the application's features.

Extensions

2a Incorrect studentID or password:

- The system detects that the studentID or password entered do not match the stored credentials.
- The system displays an error message and allows the student to re-enter the correct credentials.
- Return to step 1 of the Main Success Scenario

1a The student selects the "Forgotten password" option.

- The system guides the student through the password recovery process via a link sent to the registered email address.
- The student follows the instructions received to reset the password.
- The system confirms the password change and allows the student to log in with the new password.
- Return to step 1 of the Main Success Scenario

1b No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 1 of the Main Success Scenario.

A.2 Show course info

Use Case: Show course info

Scope: University Course Management System

Level: User goal

Intention In Context

- A student wants to view detailed information about a course.
- The system should provide an easy way to access and display this information.

Primary Actor: Student

Stakeholders' Interests

- Students: Interested in easily accessing up-to-date and comprehensive information about their courses.
- Professors: Want a reliable way to distribute course information and updates to students.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)
- The student is enrolled in the course

Minimum Guarantees

- The student sees the courses they are enrolled in

Success Guarantees

- The student successfully views the detailed course information.
- The system displays the course information accurately and completely.

Trigger

- Student navigates to course section

Main Success Scenario

1. The student searches and selects one course
2. The system shows all info about course
3. The student sees all info about selected course

Extensions

1a No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 1 of the Main Success Scenario.

A.3 Upload assignment

Use Case: Upload assignment

Scope: Course File Upload System

Level: User goal

Intention In Context

- A student needs to upload assignment to the appropriate subject section within the university application.
- The system should provide a straightforward process for assignment upload.

Primary Actor: Student

Stakeholders' Interests

- Students: Interested in a user-friendly and efficient way to find the correct subject and upload their files without confusion.
- Professors: Require an organized repository of student submissions for easy access and review.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)

Minimum Guarantees

- The system displays an error message if the file upload fails.

Success Guarantees

- The student successfully uploads the assignment for a selected course
- The system confirms the upload and the assignment is visible for student and professor

Trigger

- Student navigates to course section

Main Success Scenario

1. The student searches and selects one course for uploading assignment
2. The system shows assignment section
3. The student navigates in upload section and upload assignment
4. The system process assignment
5. The student sees uploaded assignment for that course

Extensions

1a No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 1 of the Main Success Scenario.

A.4 Book exam

Use Case: Book exam

Scope: University Exam Management System

Level: User goal

Intention In Context

- A student wants to book a session for an upcoming exam through the university application.
- The system should facilitate searching for available exam sessions and streamline the booking process.

Primary Actor: Student

Stakeholders' Interests

- Students: Interested in easily reserving available exam sessions to secure a spot for their exams.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)
- Time of the year: Student can book see and book exam only in a valid date period for exam calls.

- Compiled CPD: Students can book exam only if they have compiled CPD

Minimum Guarantees

- The student can see available exam sessions for courses.
- The system invalidate booking exam functionality if the CPD are not compiled.

Success Guarantees

- The student successfully books an exam session.
- The system confirms the booking and updates the availability status.

Trigger

- Student navigates to booking exam section

Main Success Scenario

1. The student navigates to booking exam section
2. The system shows available exam calls
3. The student selects one exam call for specific course
4. The system shows all info about exam calls and provide a way to book exam
5. The student books exam for selected course
6. The system books exam
7. The student sees booked exam

Extensions

1b No internet connection:

   – The system displays a warning message for no internet connection.
   – The student activates internet connection.

– Return to step 1 of the Main Success Scenario.

2a Booking exam disabled

– The system invalidate booking exam functionality if CPD are not compiled

– Return to step 2 of the Main Success Scenario.

A.5 Show lectures of the week

Use Case: Show lectures of the week

Scope: Weekly Schedule Display System

Level: User goal

Intention In Context

- Student sees all lectures of the week
- System must allow the student to see alle lectures of the week

Primary Actor: Student

Stakeholders' Interests

- Student:
  - Wants to have updated information about lectures, with all information like days, times and classrooms.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)
- Time of the year: Student must select a valid date period for lectures.

Minimum Guarantees

- User Feedback: The system provides adequate feedback to the user for any errors, such as connection issues

Success Guarantees

- Updated information about lectures: The system provides updated information about lectures

Trigger

- The student navigates to agenda

Main Success Scenario

1. The student navigates to agenda
2. The system shows info for week lectures and valid date option
3. The student selects a valid date for lectures
4. The system shows all lectures available for the week
5. The student sees all lectures of the week

Extensions

2a No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 2 of the Main Success Scenario.

A.6 Search place

Use Case: Search place

Scope: Map Search System

Level: User goal

Intention In Context

- Student: A university student wants to search and find a location in map through the mobile app.
- System: Must allow the student to search for and display specific locations and information about it on the map through the mobile application.

Primary Actor: Student

Support Actor

- Maps service: Provides the map, location information, and search functionalities.
- Device geolocation service: Uses the device's GPS to provide location data to the app.

Stakeholders' Interests

- Student:
  - Wants to quickly and easily find specific locations and obtain detailed information about them.
  - Wants location data to be accurate and up-to-date.

Precondition

- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data) to access the search and map visualization functionalities.
- Student Account: The student must have a registered account and be authenticated.
- Integrated Map Service: The app must be properly integrated with a map service to provide location search and visualization functionalities.

- Availability of Map Data: Map data and location information must be available and up-to-date to allow the user to find the necessary information.

Minimum Guarantees

- User Feedback: The system provides adequate feedback to the user for any errors, such as connection issues.
- Basic Operational Functionality: The search and map visualization functionalities must work at least in a limited way, even under suboptimal network conditions.

Success Guarantees

- Accurate Search and Display: The system provides accurate search results and correctly displays locations on the map.

Trigger

- The student navigates to places page

Main Success Scenario

1. The student navigates to places page
2. The system shows a map and search option for searching place
3. The student searches place
4. The system shows a list of result
5. The student selects the place from results
6. The system shows location and info of selected place using map service
7. The student sees exact location for that place in the map with all info

Extensions

2a No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 2 of the Main Success Scenario.

A.7 Open ticket

Use Case: Open Ticket

Scope: Ticket System

Level: User goal

Intention In Context

- Student opens new ticket.
- System helps student to write and open a new ticket.

Primary Actor: Student

Stakeholders' Interests

- Student opens new ticket.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Student must selects a topic and subtopic before write and open a new ticket
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)

Minimum Guarantees

- User Feedback: The system provides adequate feedback to the user for any errors, such as connection issues

Success Guarantees

- Student can open a ticket.

Trigger

- The student navigates to ticket section

Main Success Scenario

1. The student navigates ticket section
2. The system shows option for open new ticket
3. The student writes and sends ticket
4. The system elaborates and open ticket
5. The student sees new opened ticket in ticket section

Extensions

3a Create two ticket with same information:

- The system prevents the user from creating a new ticket.
- The students selects an other option for new ticket
- Return to step 3 of the Main Success Scenario.

A.8 Book study room

Use Case: Book study room

Scope: Booking system

Level: User goal

Intention In Context

- The student books study room for a specific date and time
- The system should facilitate searching for available classrooms and

seats, and streamline the booking process.

Primary Actor: Student

Stakeholders' Interests

- Students: Interested in easily finding and reserving available study room and seats to ensure they have a place to study or meet.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)

Minimum Guarantees

- User Feedback: The system provides adequate feedback to the user for any errors, such as connection issues or booking fails
- The student can access to calendar to see available study room and seats in a specific date and time.

Success Guarantees

- The student successfully books a study room and a specific seat for a specific date and time.
- The system registers every study room booked and show only available study room

Trigger

- Student navigates to bookings section

Main Success Scenario

1. The student navigates to bookings section
2. The system provides option for new booking
3. The student books study room specifying room, date, time and seat

4. The system process booking
5. The student sees study room booked with all info about date, time and seat

Extensions

1a No internet connection:

- The system displays a warning message for no internet connection.
- The student activates internet connection.
- Return to step 1 of the Main Success Scenario.

4a Classroom Already Booked

- If the selected study room becomes unavailable before the booking is confirmed, the system displays an error message.
- Return to step 3 of the Main Success Scenario.

A.9 Show degree info

Use Case: Show degree info

Scope: Degree Information Display System

Level: User goal

Intention In Context

- A student wants to view detailed information about a degree program, including list of course with all info about them.
- The system should provide an easy way to access and display this information.

Primary Actor: Student

Stakeholders' Interests

- Students: Interested in easily accessing up-to-date and comprehensive information about degree programs to make informed decisions.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)

Minimum Guarantees

- The student can search the degree program.

Success Guarantees

- The student successfully views the detailed degree information.
- The system displays the degree information accurately and completely.

Trigger

- Student navigates to degree section

Main Success Scenario

1. The student navigates to degree section
2. The system shows list all available degree
3. The student browses and select specific degree
4. The system shows all info about selected degree
5. The student sees all info about a selected bachelor/master degree

Extensions

1b No internet connection:

– The system displays a warning message for no internet connection.

– The student activates internet connection.

– Return to step 1 of the Main Success Scenario.

A.10 Show job offering info

Use Case: Show job offering info

Scope: Career Service System

Level: User goal

Intention In Context

- A student wants to view available job offers related to their field of study through the university application.
- The system should provide an easy way to access and display these job offers.

Primary Actor: Student

Support Actor

- Career Service: Provides and updates the job listings within the application.

Stakeholders' Interests

- Students: Interested in easily accessing up-to-date job offers to find employment opportunities that match their skills and career goals.

Precondition

- Student Account: The student must have a registered account and be authenticated.
- Active Internet Connection: The user must have an active Internet connection (Wi-Fi or mobile data)

- Job offers are regularly updated and maintained by career services.

Minimum Guarantees

- The student can access the job offers section

Success Guarantees

- The student successfully views the list of available job offers.
- The system displays the job offers accurately and completely.

Trigger

- Student navigates to job offering section

Main Success Scenario

1. The student navigates to job offering section
2. The system shows a list of available job offers
3. The student browses between all possible job offers
4. The student selects specific job offers
5. The system shows all info about job offer
6. The student sees all info about selected job offer

Extensions

1b No internet connection:

   – The system displays a warning message for no internet connection.
   – The student activates internet connection.
   – Return to step 1 of the Main Success Scenario.

Bibliografia

[1] *Collaudo del software*. URL: https://it.wikipedia.org/wiki/Collaudo_del_software.

[2] *List of software bug*. URL: https://en.wikipedia.org/wiki/List_of_software_bugs.

[3] *BrowserStack - What is Flaky Test?*. URL: https://www.browserstack.com/test-observability/features/test-reporting/what-is-flaky-test.

[4] Pekka Aho, Emil Alégroth, Rafael A. P. Oliveira, Tanja E. J. Vos. *Evolution of Automated Regression Testing of Software Systems Through the Graphical User Interface*. In: *The First International Conference on Advances in Computation, Communications and Services (ACCSE)*. 2016. URL: https://www.thinkmind.org/articles/accse_2016_2_20_90031.pdf

[5] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing*. URL: https://www.cs.wm.edu/~denys/pubs/ICSME%2717-CEL.pdf

[6] *How Does Appium Work?*. URL: https://appium.io/docs/en/latest/intro/appium/.

[7] *Appium client*. URL: https://appium.io/docs/en/latest/intro/clients/.

[8] *Appium drivers*. URL: https://appium.io/docs/en/latest/intro/drivers/.

[9] *Why Webdriver.IO?*. URL: https://webdriver.io/docs/why-webdriverio/.

[10] *What is CI/CD?*. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[11] *Use Case Narrative Components* URL: https://www.cplusoop.com/uml/module3/use-case-narrative.php

[12] *Page object models* URL: https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/
[13] *The Testing Pyramid: The Key to Efficient Software Testing*. URL: https://images.app.goo.gl/VQyxXADaoaD5b3kY8.
[14] *Appium philosophy and it's architecture - qavalidation* URL: https://www.google.com/imgres?q=appium&imgurl=https%3A%2F%2Fqavalidation.com%2Fwp-content%2Fuploads%2F2022%2F09%2FAppoium-intro-2.x.png&imgrefurl=https%3A%2F%2Fqavalidation.com%2F2017%2F01%2Fappium-philosophy-and-its-architecture.html%2F&docid=24Nh13BLjWhGdM&tbnid=LlnBEl81unkLBM&vet=12ahUKEwjCxL6sjoyLAxVl9bsIHacVKzMQM3oECE8QAA..i&w=1280&h=720&hcb=2&ved=2ahUKEwjCxL6sjoyLAxVl9bsIHacVKzMQM3oECE8QAA
[15] *Github Polito Students app* URL: https://github.com/polito/students-app/blob/main/assets/readme-hero.png
[16] *samsung s21* URL: https://images.samsung.com/is/image/samsung/p6pim/ph/galaxy-s21/gallery/ph-galaxy-s21-5g-g991-sm-g991bzvgphl-368315535?720_576_JPG
[17] *Frammentazione* URL: https://www.folio3.com/mobile/wp-content/uploads/2022/08/PngItem_291349.png