Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

TESTING AUTOMATICO DI APPLICAZIONI WEB

Relatore:
Chiar.mo Prof.
Paolo Ciancarini

Presentata da:
Daniele Ferrari

Correlatore:
Dott. Stefano Gozzi

Sessione III
Anno Accademico 2016-2017

"It's not a bug, it's a feature!"

Introduzione

Una ricerca di Tricentis di gennaio 2017 ha scoperto che il danno economico dovuto a errori software in tutto il mondo nel 2016 si aggira intorno a 1,1 bilioni di dollari americani. In totale gli errori in circa 363 aziende hanno colpito 4,4 milardi di clienti e si stima complessivamente a 315 il numero di anni lavoro sprecati[13].

Si parla quindi di una grande quantità di tempo e denaro.

Un errore all'interno di un software crea, tendenzialmente, una varietà enorme di danni collaterali. Anche il più innocuo dei bug in un sistema di gestione dei flussi monetari in un'azienda, per esempio, può creare dei danni di entità molto maggiore sia all'azienda stessa che a tutte quelle che usufruiscono dei suoi servizi. Potrebbe crearsi, insomma, una concatenazione di malfunzionamenti o incongruenze potenzialmente infinita.

Scenari catastrofici a parte, tutti la letteratura è d'accordo nell'affermare che la spesa maggiore nel ciclo di vita di un software è quella di manutenzione. La continua ricerca e correzione di errori e l'adeguamento a nuovi standard rappresentano un lavoro continuo e protratto nel tempo che assorbe grandi quantità di denaro e forza lavoro.

Nell'ambito delle applicazioni web queste problematiche sono ancora più ampie. Un software web deve interfacciarsi con una grandissima quantità di sistemi e configurazioni diverse oltre ai numerosi e continui aggiornamenti delle eventuali piattaforme ospite.

Ma come fare, quindi, per ridurre gli errori presenti in un software?

Sono molti gli aspetti che influiscono sulla qualità di un software e interi rami dell'informatica sono dedicati a questo.

In questa tesi verrà discusso un tema, a volte trascurato, che può ridurre di molto la presenza di errori in un programma: il testing automatico.

Dopo una breve analisi del software testing in generale, analizzeremo una tipologia di testing di web application ancora poco utilizzata, il testing automatico end-to-end.

Verrano presentate le principali caratteristiche di questa tipologia di testing, i vantaggi e le difficoltà nell'utilizzarla, alcuni strumenti per scrivere test automatici con relativi pro e contro e, infine, un'analisi approfondita su un caso reale: la creazione di una suite di test automatici tramite Cypress su un'editor di wireframe web (Balsamiq Mockup) sottoforma di plugin per Atlassian Confluence.

Indice

Bibliografia **55**

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Software Testing

In questo capitolo analizzeremo alcuni dei concetti fondamentali nel campo del software testing. Nella prima parte verranno spiegati i motivi per cui il testing rappresenta una parte di fondamentale importanza in ogni sviluppo software. Nella seconda parte verrà analizzato l'end-to-end testing su web application, facendo particolare attenzione alle tipologie di testing più utilizzate durante lo svolgimento del caso reale in esame.

1.1 Definizione di software testing

Il software testing consiste nell'eseguire una componente software per valutarne una o più proprietà di interesse. In generale, queste proprietà indicano se il componente in esame:

- soddisfa i requisiti decisi in fase di progettazione e sviluppo;
- risponde correttamente a tutti i tipi di input;
- esegue la sua funzione in un ragionevole lasso di tempo;
- è sufficientemente facile da usare;
- è installabile e utilizzabile nell'ambiente per cui è stato sviluppato

- soddisfa i risultati richiesti dagli stakeholder

Dato che il numero di possibili test, anche per componenti software molto semplici, è praticamente infinito, tutti i processi di testing utilizzano delle strategie per selezionare dei test adeguati alle risorse e al tempo a disposizione. Ciò significa che il testing può essere usato per verificare la presenza di errori nel codice ma non per garantirne la totale l'assenza. In altre parole: il testing non può dimostrare che un prodotto funziona a qualsiasi condizione, può solo dimostrare che non funziona come dovrebbe sotto specifiche condizioni.

È per questo motivo che molto spesso, in ambito di qualità software, si parla di code coverage dei test, ovvero il rapporto tra il codice coperto e il codice non coperto da test. L'obiettivo dello sviluppatore di test deve quindi essere quello di raggiungere una code coverage adeguata utilizzando il minor numero possibile di test.

Il testing è in ogni caso da considerare non come una singola attività ma come un processo che si articola durante tutto il ciclo di vita del software; è un processo iterativo: la risoluzione di un bug potrebbe svelarne altri, ancora più nascosti, o addirittura crearne di nuovi.

1.2 Importanza dei test nello sviluppo software

Il processo di sviluppo software è uno degli aspetti più complessi e vasti delle discipline informatiche; ingloba elementi economici e sociali, necessita della collaborazione di un team, spesso eterogeneo, ed è uno degli elementi più importanti per il successo di un software.

Sviluppare un software in maniera strutturata significa aumentarne la solidità e la facilità di manutenzione; significa avere documentazione attendibile e architetture solide su cui costruire funzioni complesse; senza queste basi è difficile che il prodotto risulti appetibile e, in ogni caso, sarebbe complicato da manutenere [1].

Il testing rappresenta uno degli aspetti più importanti all'interno del ciclo di vita di un software: permette di individuare gli errori durante la fase di sviluppo, aumenta la qualità del software e quindi la sicurezza e la fiducia da parte dei cliente

e degli stakeholder verso l'azienda [10], è necessario per applicazioni di alta qualità che vogliono mantenere un basso costo di manutenzione ed è fondamentale per riuscire a mantenersi sul mercato.

La difficoltà di individuazione e il costo di risoluzione di un bug aumentano notevolmente con l'avanzare del ciclo di vita del software (come si può vedere nella figura 1.1). Per questo motivo il testing dovrebbe essere effettuato a partire dalle prime fasi di produzione del codice. Generalmente è il metodo di sviluppo software scelto per il progetto a determinare quando e come viene condotto il testing. Per esempio, nei metodi a cascata, gran parte del testing viene svolto dopo la definizione dei requisiti di sistema. Al contrario, negli approcci agili, analisi dei requisiti, programmazione e testing vengono spesso fatti in parallelo.

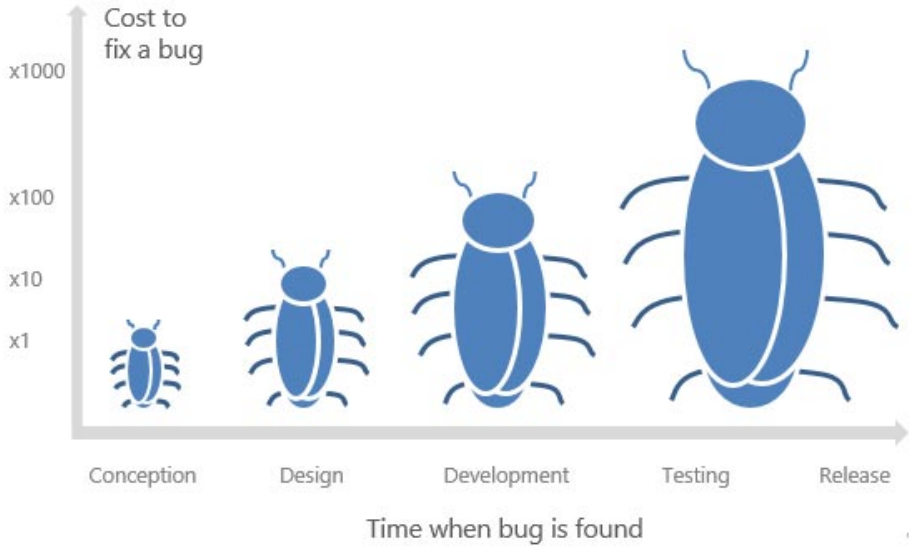

Figura 1.1: Costo di risoluzione di un bug [16]

1.2.1 Metodi agili e Test Driven Development

Tra i modelli di sviluppo software attualmente utilizzati dobbiamo evidenziare quella famiglia che prende il nome di "Test Driven Development" (TDD). L'idea di base è quella di scrivere i test di verifica prima ancora del vero e proprio codice funzionale del software. In questo modo ogni singola feature del prodotto finito,

verrà scritta con la consapevolezza dei possibili problemi che potrebbe creare. Questa strategia è utilizzata principalmente nei metodi di sviluppo agile e può vantare un Defect Removal Efficency del 95% superando i metodi waterfall (87%), così come i metodi agili che non la utilizzano (92,50%) [9].

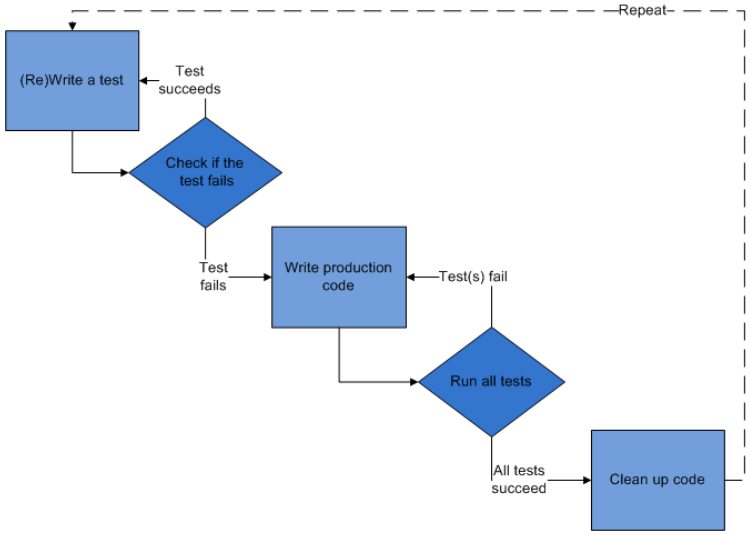

Figura 1.2: Ciclo TDD [17]

Il tipico processo di TDD (come descritto nel libro [9] e rappresentato in figura 1.2) è composto da cinque passi che vanno ripetuti fino al completo soddisfacimento dei requisiti funzionali richiesti:

1. Nella prima fase, detta "fase rossa", il programmatore scrive un test per verificare la nuova funzionalità da sviluppare, il che lo costringe a capirne nel dettaglio requisiti ed eventuali problemi. Il nuovo test viene quindi inserito nella suite di quelli già creati.

2. Vengono lanciati tutti i test (compreso quello nuovo) per controllare che non ci siano conflitti. Ovviamente il test dovrà fallire, in quanto la funzione non è stata ancora implementata.

3. A questo punto si apre la cosiddetta "fase verde", in cui il programmatore sviluppa la quantità minima di codice necessaria a far passare il test scritto in precedenza.

4. Si rilanciano tutti i test per verificare che il codice scritto funzioni correttamente. In caso contrario si corregge il test o il codice e si reitera il passaggio.

5. Infine, nella "fase grigia" si esegue il refactoring del codice per adeguarlo a determinati standard di qualità e si rilancia la suite di test per verificare che non siano stati introdotti nuovi errori.

Essendo basata sul testing, questa strategia risulta essere molto meno incline ad errori ma anche più complessa e onerosa da gestire.

Capitolo 2

Principali tipologie di testing su web application

Oggi, gran parte dei software tendono ad essere sviluppati come applicazioni web. Questo fenomeno si verifica sia nei software commercializzati che in quelli sviluppati per un uso interno alla stessa azienda che li ha prodotti. I vantaggi sono ovviamente molti, ma il poter essere utilizzati su tantissimi dispositivi e browser diversi incrementa notevolmente la probabilità di guasti o malfunzionamenti. Risulta quindi necessario costruire una struttura di test solida che verifichi l'usabilità del prodotto, da lanciare possibilmente dopo ogni aggiornamento (sia del prodotto che dei browser).

2.1 Unit Test

Questa tipologia di test serve per collaudare singole unità software, ovvero le minime componenti di un programma dotate di funzionamento autonomo. Lo unit test viene svolto normalmente dagli sviluppatori e può essere eseguito anche come *"glass box"*, ovvero basato esplicitamente sulla conoscenza dell'architettura e del funzionamento interno di un componente oltre che sulle sue funzionalità esternamente esposte.

Nelle applicazioni web lo unit test permette anche di verificare che una determinata porzione di codice continui a funzionare anche quando le librerie o i tool che utilizza vengono aggiornati.

Come descritto nell'articolo *"Beautiful Testing As the Cornerstone of Business Success"* di Lisa Crispin nel libro *"Beautiful Testing"* [10], lo unit test automatico è sicuramente il più redditizio in merito al ROI (Return of Investment). Nello stesso articolo viene anche spiegato che introdurre l'utilizzo di unit test automatici all'interno di un team non esperto in Test Driven Development risulta un processo molto difficile. Inizialmente lo sforzo risulta essere molto alto ma con il passare del tempo diminuisce notevolmente, come si può vedere nella figura 2.1, che mostra la "Rampa del dolore" definita da Brian Marick.

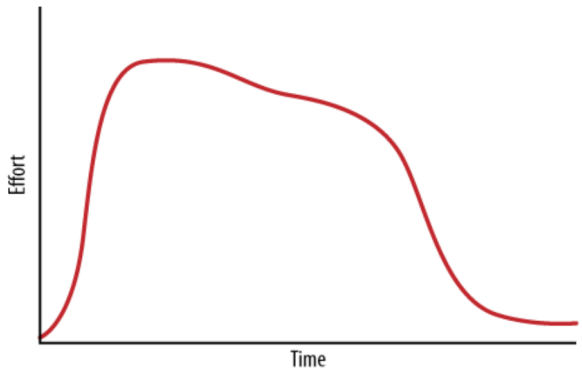

Figura 2.1: La "Hump of Pain" [10]

2.2 Graphic User Interface Test

I Graphic User Interface (GUI) Test sono i test effettuati sull'aspetto estetico del programma. Questi tipi di test verificano che tutti i tasti, le caselle di testo, le immagini, le icone delle interfacce utente vengano visualizzati in maniera corretta e permettano all'applicazione di soddisfare le sue specifiche.

A differenza dei sistemi con interfacce a linea di comando, una GUI ha molte più funzionalità da testare. Anche un programma relativamente piccolo può

avere diverse centinaia di possibili operazioni eseguibili tramite GUI. Ovviamente, aumentando le dimensioni del programma, la quantità di operazioni aumenta esponenzialmente, rendendo i test sempre più complessi da scrivere e manutenere.

Un problema dei test su GUI è riconducibile al fatto che, molto spesso, è necessario eseguire una serie di comandi (cliccare su un'icona, selezionare una voce nel menù a tendina e così via) per riuscire a raggiungere una determinata interfaccia del programma. Logicamente, se in un passaggio precedente a quello da verificare qualcosa è andato storto, tutta la restante parte del test risulterà fallimentare.

Altro grande problema nei test sulle GUI di applicazioni web nasce dalla natura multipiattaforma di questo tipo di applicazioni. La possibilità di funzionare su differenti browser e dispositivi aumenta notevolmente il numero di stati in cui si può trovare l'applicazione rendendo ancora più numerosi i casi di test.

Questo genere di test è principalmente manuale: un tester esegue una serie di operazioni sull'applicazione e controlla visivamente se il risultato ottenuto è quello desiderato. In questo ambito, i test automatici ancora non sono perfetti. Esistono strumenti (Screenster, vedi paragrafo 4.3) in grado di effettuare dei confronti grafici tra il risultato voluto e quello reale dopo ogni operazione. La difficoltà nell'uso di questi strumenti è dovuta alla scelta del grado di precisione nel confronto: se la soglia di errore è troppo alta si rischia di trovare dei falsi positivi, in caso contrario ci saranno molto falsi negativi. Inoltre, effettuando dei confronti per immagini, lo svolgimento dei test risulta spesso rallentato.

2.3 Integration Test

I test di integrazione vengono effettuati per verificare che, dopo aver unito diversi moduli di un programma, questo funzioni correttamente e non ci siano conflitti tra i suoi componenti. Il test di integrazione è sicuramente uno dei più completi e sicuri, perchè verifica una serie di passaggi consecutivi che non riguardano solamente l'aspetto visivo dell'applicazione ma anche quello funzionale. E' una delle tipologie di test più dispendiose e complesse; tra le aziende produttrici

di software che investono sui test, infatti, l'integration testing è quello che assorbe il maggior sforzo aziendale (tra il 50% e il 70%) [7].

Le principali strategie di integration testing sono tre:

- **Big Bang**, in cui si uniscono tutti (o comunque la maggior parte) dei componenti del software prima di fare il test.
- **Top-Down**, si parte dal test dei moduli di più alto livello per continuare con gruppi di moduli dipendenti a livelli inferiori.
- **Bottom-Up**, il test parte dai moduli di livello basso, e unisce man mano, componenti sempre di più alto livello, fino a raggiungere il livello più alto della gerarchia dei componenti.

Nell'ambito delle applicazioni web questa tipologia di testing può essere utilizzata per verificare i risultati intermedi in punti chiave all'interno dell'architettura dell'applicazione. Questo può essere fatto tramite l'analisi dello stato dell'applicazione valutato sia tramite i dati persistenti sia tramite quelli mantenuti in memoria dall'applicazione o da qualsiasi suo componente distribuito.

2.4 Regression Test

I regression test sono quella famiglia di test utili a verificare che il software, a seguito di un aggiornamento (anche minimo) di uno dei suoi componenti, non presenti conflitti che non permettano di utilizzarlo.

I regression test accertano anche che i bug conosciuti non rimangano inavvertitamente dopo l'aggiornamento che li avrebbe dovuti rimuovere.

Questo genere di test è molto comune nelle produzioni software che utilizzano metodi agili o, comunque, cicli di release rapidi, ed è fondamentale dopo qualsiasi aggiornamento, per evitare conflitti indesiderati.

Circa il 95% dei progetti analizzati (circa 12,000) nel libro *"The Economics of Software Quality"* [8] utilizzano i regression test.

Proprio perchè devono valutare molti aspetti dell'applicazione, questi test sono generalmente eseguiti in maniera automatica dopo ogni aggiornamento: gestirli manualmente è complicato e sicuramente molto più oneroso.

2.5 Smoke Test

Gli smoke test vengono utilizzati per verificare degli aspetti critici e basilari del software. Si verifica, per esempio, se si riesce a lanciare il programma, se l'interfaccia grafica viene caricata, se il click sui pulsanti principali funziona correttamente. Gli smoke test servono, quindi, a verificare se abbia o meno senso proseguire con i successivi, più specifici, test: se lo smoke test va a buon fine si può continuare, altrimenti ci sono problemi più gravi da risolvere prima di proseguire. Vengono generalmente eseguiti prima di ogni release, in modo da evitare di pubblicare un aggiornamento in grado di creare danni a tutto il software.

2.6 End-to-end Testing

Con questo termine si intende l'attività di testing dell'applicazione per come la vedono gli utenti, dall'inizio alla fine. L'end-to-end testing comprende il testing di interfacce e delle dipendenze esterne, come l'ambiente in cui viene eseguito e il backend. Con questa famiglia di test non vengono solo verificate proprietà funzionali ma possono essere anche verificate proprietà non funzionali come prestazioni e stabilità.

Questo genere di test viene usato per verificare che lo scopo per cui l'applicazione è stata progettata venga soddisfatto senza problemi dall'inizio alla fine. L'obiettivo dei test end-to-end è proprio quello di identificare le dipendenze nel sistema, assicurandosi che le informazioni scambiate tra i vari suoi componenti o con altri sistemi siano corrette. L'intera applicazione è quindi testata in uno scenario reale in cui per esempio interagisce con un utente o con un database.

Un semplice percorso effettuato da un test end-to-end su un servizio web di posta elettronica potrebbe essere il seguente:

1. Lancia un'istanza di un browser
2. Inserisci l'URL del servizio e aspetta il caricamento
3. Fai log-in nel servizio di posta inserendo le credenziali nel form della pagina
4. Accedi alla posta in arrivo per verificare che il contenuto sia visualizzato correttamente
5. Accedi alla posta letta e non letta e verifica che il contenuto sia visualizzato correttamente
6. Fai log-out dall'applicazione

Nelle applicazioni web, e in particolare nel caso reale in esame, il testing end-to-end è una delle famiglie di testing più utile. Questo genere di testing, che racchiude tutti quelli precedentemente descritti, permette di simulare scenari realistici di interazione utente-applicazione rendendo i test molto più vicini al caso reale.

Capitolo 3

Testing automatico end-to-end di web application

In questo capitolo analizzeremo le caratteristiche, i vantaggi e le difficoltà nell'utilizzo di testing automatico nell'ambito dell'end-to-end testing di web application. Il testing automatico è ancora poco diffuso tra le aziende per vari motivi ma la principale ragione è sicuramente riconducibile al fatto che lo sforzo di progettazione, scrittura e manutenzione di una suite di test automatici è decisamente più oneroso del corrispettivo necessario per i test manuali.

A differenza dei test manuali, lo sviluppo di test automatici, infatti, è facilmente paragonabile a quello di un software. Prima di scrivere un test automatico bisogna conoscere la struttura interna del software da testare, i suoi punti critici e molti dettagli implementativi. Onde evitare la creazione di test illeggibili e difficili da manutenere è necessario commentare i passaggi complessi e le funzioni utilizzate; bisogna mantenere degli standard e seguire delle linee guida come l'utilizzo di red test. Infine è necessario che il test sia parlante, che faccia capire al tester, in caso di errore, dove si sia riscontrato e quale potrebbe essere. Proprio per questi motivi gli sviluppatori di test automatici devono avere competenze molto più elevate rispetto a quelle necessarie per progettare ed eseguire test manuali.

Come la programmazione di software, anche quella dei test automatici quindi è un'arte difficile da padroneggiare.

3.1 Vantaggi e difficoltà

I principali vantaggi dell'utilizzo di test automatici sono:

- Possibilità di testare molti aspetti di un software in poco tempo
- Riusabilità di test su differenti applicazioni
- Possibilità di lanciare test in parallelo per valutare le capacità di carico o la real time collaboration
- Sicurezza del risultato (dando per scontato che il test sia scritto bene)
- Bassa percentuale di falsi positivi o negativi (a patto che il test sia solido)

Il motivo principale per cui si vuole costruire una suite di test automatici è sicuramente perchè si vuole eseguire dei test ripetutamente dopo ogni nuova release. Se un test deve essere eseguito una sola volta, lo sforzo per automatizzarlo potrebbe superare i benefici offerti.

Dal punto di vista puramente teorico l'introduzione dei test automatici nel ciclo di vita di un software è un passo obbligatorio per un'azienda perchè, a lungo termine, semplifica notevolmente tutto il testing; scontrandosi con la realtà economica però ci si rende conto che la costruzione di una suite di test automatici sia un lavoro, soprattutto inizialmente, dispendioso e complicato. Per scrivere test automatici solidi sono necessarie grandi conoscenze nel campo della programmazione, oltre a quelle riguardanti il testing; un'azienda potrebbe, quindi, fare fatica a rinunciare a del personale qualificato investibile in ambiti come la progettazione o lo sviluppo per assegnarlo ai test automatici.

Nell'articolo *"When to automate your testing (and when not to)"*[5] di Oracle del 2013 vengono spiegati alcuni concetti chiave per comprendere perché e quando sviluppare test automatici. Una delle domande a cui l'articolo trova risposta è perché molte aziende continuino ad affidarsi principalmente al testing manuale. I vantaggi del testing manuale rispetto a quello automatico sono diversi:

- Tempo: i team di testing potrebbero non avere tempo da investire in alternative al testing manuale, nell'apprendimento dell'utilizzo degli strumenti e/o nello sviluppo e manutenzione di suite di test automatici.

- Complessità: alcune applicazioni potrebbero essere troppo complesse e non indicate per il testing automatico.

- Abilità: alcuni tester (business analyst, ecc) potrebbero mancare di alcune abilità necessarie all'utilizzo di strumenti per i test automatici.

- Costo: alcune aziende potrebbero non possedere strumenti di test automatico e di denaro da investire per comprarli (o per insegnare ai tester come utilizzarli)

- Affidabilità: le organizzazioni di tester o quality assurance potrebbero avere molta esperienza e sicurezza nei test manuali ed essere quindi intimorite del passaggio a quelli automatici.

- Disinformazione: è possibile che alcune aziende non siano a conoscenza dell'esistenza e della possibilità di automatizzare i test.

La situazione dal 2013 ad oggi è sicuramente migliorata, ma il fatto che molte aziende non adottino ancora i test automatici è sicuramente imputabile a uno o più di questi fattori.

Anche nella nona edizione del *"World Quality Report 2017-2018"*, che raccoglie dati proveniente da 1660 aziende sparse in 32 stati nel mondo [4] si può notare che solo il 15% delle più comuni attività di testing sono effettuate tramite strumenti automatici. Le principali sfide del testing automatico, secondo la ricerca effettuata da Capgemini e Sogeti, nascono dalla disponibilità dei dati e degli ambienti di testing e dalla mancanza di supporto per il testing su dispositivi mobile (come si può vedere dal grafico in figura 3.1. Allo stesso tempo, il 41% delle aziende non ha un approccio solido ai test che includa delle automazioni standardizzate.

Investire sui test automatici significa agevolare la manutenzione del software, rilasciare aggiornamenti più sicuri e affidabili, ridurre notevolmente la probabilità

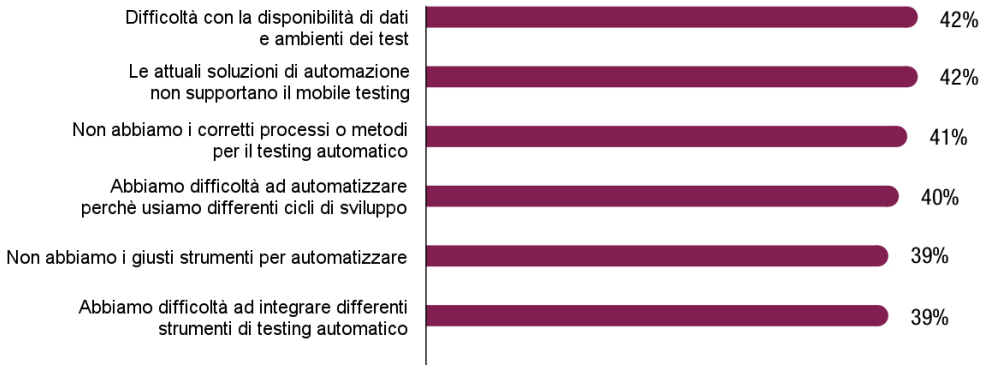

Figura 3.1: Difficoltà nell'automatizzazione dei test

che il cliente trovi un bug nel prodotto e ridurre, a lungo termine, il tempo investito nel processo di testing. Anche se, sulla carta, può sembrare una spesa superflua e facilmente evitabile, l'investimento in test automatici costituisce un passo fondamentale per un'azienda che vuole produrre un software di qualità, aggiornato e sicuro nel tempo.

Tempo e denaro

Uno dei principali motivi per automatizzare un test è il risparmio di tempo a lungo termine. Molti test vengono lanciati più volte a distanza di tempo. Ogni volta che viene modificata una parte di codice nel software, il test dovrebbe essere ripetuto. Effettuare un test manuale ogni volta è sicuramente più dispendioso in termini di tempo che scrivere un test automatico e programmarne il lancio.

Inoltre, se l'applicazione è presente su differenti browser o sistemi operativi, la quantità di test da lanciare ad ogni modifica aumenta notevolmente. Una volta creato, un test automatico può invece essere lanciato infinite volte senza nessun costo aggiuntivo.

Lo svolgimento di un test automatico è sicuramente più veloce di quello di un test manuale, il che riduce ulteriormente il tempo di testing dopo ogni aggiornamento, permettendo all'azienda di risparmiare tempo e denaro.

Precisione

Anche il più attento tester può fare degli errori durante il monotono svolgimento del testing manuale. I test automatici effettuano esattamente gli stessi passi ogni volta che vengono lanciati, senza dimenticare di registrare dettagliatamente i risultati ottenuti. I tester, che non devono più eseguire i test manuali, hanno più tempo da dedicare alla scrittura di nuovi test automatici o all'implementazione di funzionalità più complesse.

Test coverage

Il testing automatico può aumentare notevolmente la profondità e lo scope dei test, aiutando a incrementare la qualità del software. Test lunghi, spesso evitati nel testing manuale, possono essere sviluppati e lanciati più agevolmente tramite i test automatici. Possono essere lanciati anche su differenti browser o dispositivi, con differenti configurazioni. I test automatici possono entrare all'interno dell'applicazione e guardare i contenuti della memoria, i dati, il contenuto dei file e lo stato interno del software determinando se il software si sta comportando correttamente. Essi possono inoltre eseguire centinaia di differenti e complessi casi di test contemporaneamente, fornendo una copertura del codice impossibile per i test manuali.

Utenti contemporanei

Anche il più grande team di Quality Assurance non potrebbe eseguire un test di web application controllato con migliaia di utenti. I test automatici possono simulare facilmente decine, centinaia o migliaia di utenti virtuali che interagiscono contemporaneamente con una rete, un software o un'applicazione web.

Nonostante l'utilità e la potenza dei test automatici, questi non riescono (ancora) a soddisfare appieno tutte le necessità di testing di un'applicazione. In molti casi, infatti, solo i test manuali sono considerabili consistenti e sicuri. Uno di questi casi è quello delle interfacce grafiche e di tutto ciò che riguarda la visualizzazione a schermo.

Dato che i test automatici lavorano direttamente sul codice sorgente dell'applicazione, non riescono ad accorgersi, in alcune situazioni, di errori dovuti solamente alla visualizzazione. Tipici problemi di questo genere si possono verificare a causa della non compatibilità tra diversi browser o sistemi operativi e sono difetti non facilmente riscontrabili analizzando solamente il codice sorgente.

Nonostante ciò, si stanno sviluppando tecniche di analisi visuale che potrebbero automatizzare anche questi tipi di test. Allo stato dell'arte, però, ancora siamo lontani dal poter creare delle suite di test sicure, consistenti ed efficaci che riescano ad individuare questi errori senza sollevare inutili falsi negativi.

Nella rivista *"World Quality Report 2017-2018"* [4], già citata precedentemente, è presente un grafico che indica quali sono i benefici dei test automatici secondo le 1660 aziende intervistate (Figura 3.2). Questi dati confermano i benefici elencati precedentemente portando in luce anche altri vantaggi, come il miglioramento sul controllo e la trasparenza delle attività di testing. Questo significa che, tramite il testing automatico, un'azienda è in grado di supervisionare più facilmente il processo di testing e può permettere anche al cliente di osservarne lo svolgimento.

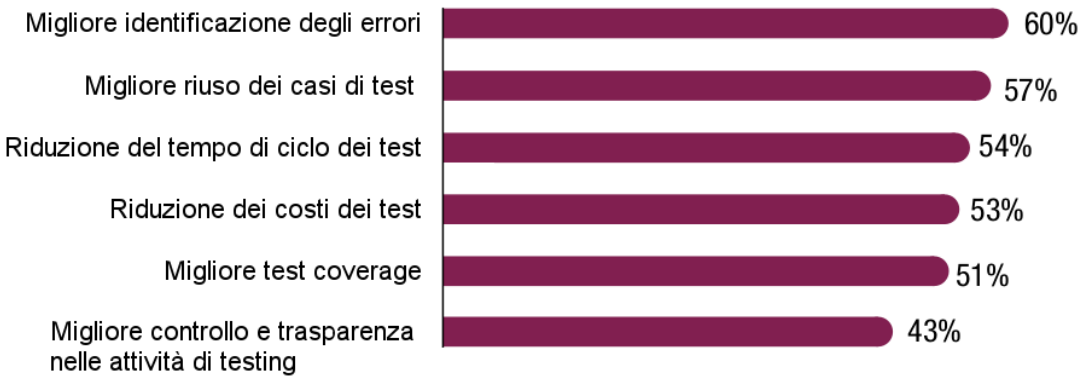

Figura 3.2: Benefici dei test automatici [4]

3.2 Regole di buona scrittura

In questa sezione verranno presentate una serie di regole per la buona scrittura di test, sia generali che specifiche dei test automatici su web application.

1. **Nomenclatura**

   Anche se può sembrare banale è importante dare ad ogni test un nome che abbia senso sia per chi lo scrive che per chi dovrà utilizzarlo in futuro. È buona norma dare un nome che rappresenti il modulo, la funzione o l'insieme di funzioni che il test andrà a verificare.

2. **Descrizioni**

   La descrizione del test deve spiegare, più dettagliatamente possibile, che cosa il test andrà a fare, qual è il suo obiettivo, qual è l'ambiente in cui dovrà essere eseguito, quali dati verranno utilizzati, ecc. È meglio inserire all'interno della descrizione quante più informazioni possibili.

3. **Assunzioni e precondizioni**

   Si dovrebbero includere nella descrizione del test tutte le assunzioni applicate e ogni precondizione che deve essere soddisfatta prima che il test venga lanciato. Andrebbe specificato se il test è dipendente dalle interazioni con l'utente, dall'ambiente in cui lo si esegue o da eventuali altri test da eseguire prima, dopo o durante il test in questione.

4. **Risultati attesi**

   Un test ben scritto dovrebbe specificare dettagliatamente, ad ogni passo, il risultato atteso. Per esempio se deve apparire una pagina bianca, o uno specifico cambiamento dopo un determinato passo.

5. **Copertura**

   Una caratteristica fondamentale di un buon test è il fatto che debba verificare ogni aspetto della funzione o del modulo in esame. Il test non deve solamente eseguire dei passaggi che simulino il flusso di azioni di un utente, ma deve

anche verificare che ogni passaggio restituisca il risultato voluto. Deve cercare quindi di analizzare il modulo o la funzione il più ampiamente possibile.

6. **Leggibilità e comprensione**

   Come in ogni ambito della programmazione, la chiarezza è fondamentale. Bisogna considerare che il test non sarà eseguito solamente da una persona, quindi deve essere di facile comprensione, leggibile e diretto. L'obiettivo è scrivere test semplici e facilmente comprensibili a tutti, che arrivino al punto senza girarci intorno. Nonostante ciò, un test deve comunque coprire tutti gli aspetti del modulo o della funzione che sta verificando, come specificato nella regola 5.

7. **Riusabilità**

   Come per la precedente, questa regola vale per tutta la programmazione in generale. Scrivere test troppo specifici è sicuramente un errore (a meno che non si tratti di unit test); l'idea è quella di scrivere test facilmente riutilizzabili in futuro, riducendo gli sprechi di tempo e facilitando la scrittura dei test successivi.

8. **Progettare il test prima di svilupparlo**

   È sempre una buona pratica valutare l'obiettivo e lo scenario del test prima di svilupparlo. Scrivere subito il test rischia di portare lo sviluppatore a concentrarsi solo alla creazione di uno script che dia risultato positivo valutando solamente le condizioni favorevoli e non pensando a differenti scenari possibili. Inoltre, non si deve ridurre lo scope del test solo per fare in modo che funzioni e dia risultato positivo.

9. **Rimuovere le incertezze dei test automatici**

   Una delle caratteristiche principali del testing automatico è l'abilità di dare risultati consistenti, tali da essere certi che qualcosa sia andato storto se il test è fallito.

Se un test automatico passa a un lancio e fallisce al successivo, senza nessuna modifica al software in esame, non saremo mai certi che il fallimento sia causato dall'applicazione o che sia dovuto ad altri fattori.

Quando un test fallisce bisogna analizzare i risultati per capire cosa è andato storto e, quando si hanno molte inconsistenze nei risultati o falsi positivi, il tempo per farlo aumenta notevolmente.

Non bisogna avere timore di cancellare dei test instabili ma, piuttosto, ci si deve concentrare sui risultati consistenti e puliti dei quali poterci fidare.

10. **Non affidarsi troppo ai test**

    Dopo un aggiornamento il risultato positivo di un test automatico potrebbe dare al team di sviluppo una certa sicurezza che il nuovo codice o la modifica non presenti bug. Anche se tutti i test risultassero positivi, non avremmo comunque la certezza che non siano presenti dei difetti. Non bisogna quindi affidarci totalmente ai test automatici; come già detto in precedenza, il testing aiuta nella ricerca di bug ma non ne garantisce l'assenza.

11. **Evitare di automatizzare tutti i test**

    La copertura totale del codice tramite test non è possibile perchè le combinazioni di scenari e azioni è enorme. I test che vengono eseguiti sono sempre un sottoinsieme di tutti quelli possibili. Questo principio è valido anche per i test automatici. Creare un test automatico costa tempo e fatica, necessita di numerose risorse e in alcuni casi, per diverse ragioni, è addirittura impossibile. È però possibile utilizzare un approccio basato sul rischio che determina quali test dovrebbero essere automatizzati. Per raggiungere il massimo valore da un'automazione è bene automatizzare soltanto gli scenari e i casi di maggior valore economico. Inoltre, un alto numero di test automatici aumenta notevolmente i costi e la difficoltà di manutenzione della suite.

    Un'altra considerazione da tenere a mente è che non tutti i test possono essere automatizzati. Alcuni test, infatti, hanno una natura troppo complessa,

necessitano di troppi controlli e possono risultare inconsistenti. In quei casi è meglio lasciare che il test venga fatto manualmente.

12. **Selettori**

    Per identificare un oggetto all'interno di una pagina web si può ricorrere a diversi metodi. Il più solido è sicuramento utilizzare l'ID o il nome dell'oggetto da selezionare; utilizzare questi parametri è facile, efficiente e incrementa notevolmente prestazioni e leggibilità del codice. In alcune situazioni l'unico modo per raggiungere un elemento è utilizzando il suo XPath; anche se più lento permette di verificare che un elemento sia nella corretta posizione all'interno della gerarchia della pagina. Per raggiungere gli elementi dinamici è sempre meglio utilizzare gli XPath dato che, generalmente, il loro ID o nome viene generato dinamicamente. È bene quindi mappare i selettori in variabili dei test, in modo da poterli cambiare nel caso in cui siano fragili o vengano modificati.

13. **Attese**

    È fortemente consigliato non utilizzare dei comandi di sleep perchè potrebbero creare dei problemi all'interno del test, a partire dall'aumento del tempo di esecuzione. Quando strettamente necessario è più indicato utilizzare dei wait, o ancora meglio dei fluent wait ovvero delle attese deterministiche che sollevano un'eccezione nel caso in cui l'elemento non venga trovato. La migliore pratica è quella di utilizzare metodi di polling (descritti brevemente nella sezione 4.1).

Capitolo 4

Alcuni strumenti per scrivere test automatici a confronto

In questa sezione effettueremo un'analisi di alcuni strumenti per la scrittura e gestione di test automatici. Partiamo dal presupposto che nessuno strumento è in grado di soddisfare tutte le esigenze di una determinata azienda o, ancora più nello specifico, di un determinato progetto. Ogni strumento ha dei pro e dei contro e la scelta dipende dalle necessità del progetto. In questo capitolo verranno analizzati alcuni degli strumenti più conosciuti e interessanti per la scrittura e gestione di test automatici di siti e applicazioni web, descrivendo i pro e i contro di ognuno nell'ottica di costruire una suite di test end-to-end.

4.1 Selenium

Selenium è il più diffuso tool per l'automazione di browser in commercio [3]. Nonostante la sua principale funzione non sia quella di web testing automatico, esso viene utilizzato principalmente per questo. Quelli che da ora in poi chiameremo per semplicità "test", in realtà, sono da intendere come sequenze di comandi che permettono l'automazione di un browser. Selenium permette quindi di specificare istruzioni per automatizzare le interazioni con il sito o l'applicazione web;

non fornisce nativamente comandi come le asserzioni nè, tantomeno, strumenti di gestione dei test.

Si tratta di un prodotto open-source nato in seguito alla Google Test Automation Conference del 2009 dall'unione di due progetti (uno del 2004 e uno del 2007). Nel 2012 è stato riconosciuto come internet standard dal W3C e oggi è allo stadio di Candidate Recommendation.

Il suo principale componente è Selenium WebDriver, l'ambiente di sviluppo che permette di registrare, modificare e debuggare test per web application. È implementato come un driver per browser e non necessita di un server per eseguire i comandi. Ciò gli permette di interagire con i browser più diffusi come Firefox, Chrome, Internet Explorer e Microsoft Edge. I Test sono registrati in *Selenese*, uno speciale linguaggio di scripting ideato appositamente per Selenium.

Un altro elemento molto importante è il Selenium Grid, un server che permette ai test di utilizzare delle istanze di web browser su macchine remote. Con questo sistema un server agisce da hub e viene contattato dai test per accedere alle istanze dei browser. Selenium Grid permette, quindi, di lanciare test in parallelo su diverse macchine e di gestire differenti versioni e configurazioni di browser in maniera centralizzata (invece che per ogni singolo test).

Vantaggi

Oltre ad essere un software open source con molti anni di esperienza alle spalle, e dunque supportato da una vasta documentazione, Selenium permette di lanciare test su diversi sistemi operativi e browser, anche in parallelo. Inoltre permette l'integrazione dei test su sistemi come Jenkins, TestNG e Maven agevolando i processi agili e di continuous integration.

Svantaggi

Proprio per il fatto che Selenium nasce con l'intento di automatizzare dei comandi su browser, manca di alcuni elementi fondamentali per uno strumento di creazione e gestione di test automatici. Infatti, come precedentemente accennato, Selenium non fornisce un sistema di gestione dei test che permetta di programmare

i lanci o salvare i risultati ottenuti né la possibilità di effettuare asserzioni. Per rimediare a questa mancanza è necessario accompagnare Selenium a dei framework. La necessità di installare questi componenti extra e quindi di doverli configurare, rende la fase di inizializzazione molto complessa e insidiosa.

Un'altra caratteristica negativa di Selenium è la necessità di tempi di attesa espliciti per testare le applicazioni AJAX. Questo è dovuto al fatto che l'esecuzione di un comando su un elemento (per esempio un click su un pulsante) avviene senza considerare se l'elemento in questione sia o meno presente all'interno della pagina. Ciò significa che il test fallirà sia se l'elemento non è presente nella pagina per un errore, sia se invece è semplicemente in fase di caricamento. Questo meccanismo può facilmente diventare un problema nel caso si stia lavorando su pagine web AJAX.

La soluzione consiste nell'utilizzo del "polling". Se l'elemento selezionato non è disponibile si riprova ad accedervi ad intervalli di tempo regolari, finché l'elemento non è pronto oppure non si raggiunge il timeout. Questa tecnica potrebbe comunque risultare una perdita di tempo nel caso in cui l'elemento selezionato non sia effettivamente presente nella pagina. Inoltre aumenta notevolmente la complessità nella scrittura dei test perchè deve essere gestita all'interno del suo stesso codice.

Conclusioni

Selenium è un prodotto maturo, nel bene e nel male; a fianco di molti suoi sostenitori si trovano altrettanti sviluppatori che non lo apprezzano affatto. Alcune scelte tecniche risultano ancora ostiche e di difficile gestione. Installazione e configurazione sono molto più complicate della maggior parte dei tool sul mercato e rendono questo prodotto poco appetibile.

4.2 Cypress

Si tratta di un tool open source per il testing end-to-end di siti e applicazioni web nato nel 2016 e disponibile su GitHub [12]. L'idea alla base del progetto è quella di riunire tutti i componenti necessari a un tool di testing automatico

tradizionale (framework, libreria di asserzioni ecc) in un unico strumento, efficiente e facile da usare. I test sono interamente scritti in JavaScript con funzioni della libreria di Cypress che permettono di interagire con il tool.

Vantaggi

Installazione e configurazione di Cypress risultano essere molto semplici e veloci. Uno dei punti di forza del prodotto risiede nella chiarezza e semplicità dei comandi. La struttura delle frasi è studiata per essere parlante e di facile comprensione. Cypress implementa nativamente Chai, una libreria JavaScript per le asserzioni, e Mocha, un framework che facilita la gestione dei test asincroni; due tool molto familiari alla gran parte degli sviluppatori web.

A differenza di Selenium, Cypress viene eseguito nello stesso contesto dell'applicazione da testare, e non come una serie di comandi remoti al browser. Questo gli permette di comunicare con i processi Node.js costantemente. Avere accesso a entrambe le parti del sistema permette di rispondere agli eventi dell'applicazione in tempo reale e, allo stesso tempo, di lavorare all'esterno del browser con funzioni che richiedono privilegi più alti.

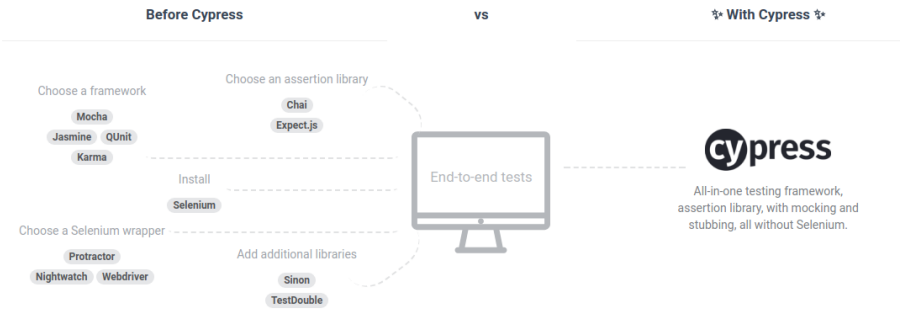

Figura 4.1: Vantaggi Cypress [18]

Cypress controlla ogni parte del processo di automazione assicurandosi la comprensione di qualsiasi cosa succeda all'interno e all'esterno del browser.

Dato che Cypress lavora all'interno dell'applicazione da testare, ha accesso nativo ad ogni suo singolo oggetto. I test quindi, potranno accedere a qualsiasi oggetto al quale può accedere la stessa applicazione.

Cypress lancia i test all'interno di un'istanza di Chrome in incognito, istanza che viene avviata all'inizio del processo di testing. Questo permette allo sviluppatore di test di utilizzare tutti gli strumenti presenti sul browser, come l'inspector o la console, per interagire dinamicamente con la pagina durante l'esecuzione dei test. Inoltre, dato che il browser viene lanciato in incognito, ogni nuova istanza di test sarà libera da ogni residuo dei precedenti lanci, come cronologia, dati di navigazione o cookies.

Cypress dispone anche della funzione di "viaggio nel tempo": tutti i passaggi di test e le interazioni con la pagina vengono registrati in un elenco nella parte sinistra del browser. È possibile interrompere in qualsiasi momento l'esecuzione del test e navigare avanti e indietro tra i passi di test precedentemente effettuati semplicemente cliccando sugli elementi dell'elenco.

Questo permette al tester di verificare manualmente, in caso di errore, tutti i passaggi che hanno portato il test al fallimento.

Cypress permette anche di registrare un video dell'intero test e salvare uno screenshot della pagina nel caso in cui si verifichi un errore. Inoltre dispone di una sezione per la gestione programmata dei test che permette di decidere quando e quali test eseguire, salvandone i risultati in un file di log.

Un'altra grande differenza con Selenium sta nel fatto che Cypress, grazie a Mocha, implementa nativamente la gestione asincrona degli elementi della pagina, rendendo molto più semplice la gestioni di applicazioni o siti AJAX. La tecnica utilizzata è quella dei "promise": particolari oggetti JavaScript che assicurano che un'operazione ancora non completata si risolverà in futuro. L'utilizzo di questi oggetti evita l'introduzione di attese esplicite che appesantirebbero il test, aumentandone invece la stabilità.

Infine, un ultimo punto forte di questo prodotto risiede nella folta community di utenti, molto attiva sia su GitHub che su Gitter [15] (un sistema di web chat ideato per mettere in collaborazione utilizzatori di repository GitHub), e nella

velocità ed efficienza del team di sviluppo, sia nelle risposte ai feedback che nella gestione delle issue su GitHub.

Svantaggi

La principale limitazione nell'uso di Cypress riguarda il fatto che è compatibile solamente con Chrome, ma il team di sviluppo sta già lavorando all'integrazione del tool su altri browser.

Come Selenium, anche questo prodotto necessita di capacità di programmazione JavaScript piuttosto elevate e non può essere utilizzato da tutti.

Essendo un prodotto piuttosto giovane, manca di alcune feature che tool come Selenium hanno già implementato. Una di queste riguarda l'interazione con gli iframe, che approfondiremo nel capitolo 6.1. Inoltre, anche se la documentazione sul sito è molto valida, manca di alcuni dettagli che, in determinate situazioni, possono tornare molto utili.

4.3 Screenster

Screenster è un software di visual regression testing per applicazioni web nato nel 2016. È disponibile in versione server o cloud con licenza a pagamento. A differenza dei prodotti descritti in precedenza, Screenster è uno strumento di test visuale e rappresenta un grande passo verso il futuro dei regression test automatici.

Questo tool permette di registrare un test semplicemente riproducendone i passaggi. Durante questa fase, dopo ogni azione, il software memorizza uno screenshot. Conclusa la registrazione il programma è in grado di verificare la buona riuscita del test riproducendo i passaggi registrati e confrontando, ad ogni passo, gli screenshot del caso base con quelli del test.

È possibile selezionare alcune zone della pagina da non confrontare con il caso base. Questo permette di alleggerire l'esecuzione dei test e, in alcuni casi, evita il presentarsi di falsi negativi.

A differenza del testing tradizionale, quindi, il visual testing confronta il risultato sperato e quello in esame solamente da un punto di vista grafico. Il te-

sting è quindi molto più vicino ad un'interazione simulata utente-applicazione e rappresenta il massimo livello di testing end-to-end finora raggiunto.

Per evitare errori dovuti alla scelta di selettori imprecisi, il programma memorizza ogni discendente, figlio e attributo HTML di ogni elemento DOM della pagina. In questo modo se durante il test un selettore non dovesse funzionare, Screenster lo scambierà con un altro.

Vantaggi

La semplicità di utilizzo del software rappresenta sicuramente un punto a suo favore. L'utilizzo di visual testing tramite screenshot è un elemento innovativo nell'ambito del software testing e costituisce, a mio parere, il futuro dei testing end-to-end di applicazioni web e non.

Altro vantaggio di questo prodotto risiede nella possibilità di gestire tutta la suite di test comodamente dal servizio di cloud fornito dall'applicazione. Tramite questo servizio è possibile registrare, organizzare e lanciare comodamente i test.

Svantaggi

A differenza degli altri due software questo è disponibile solo in versione freemium (versione gratuita per un utente con 50 test al mese) con abbonamento mensile. L'integrazione con JIRA e il supporto tecnico sono disponibili solo per

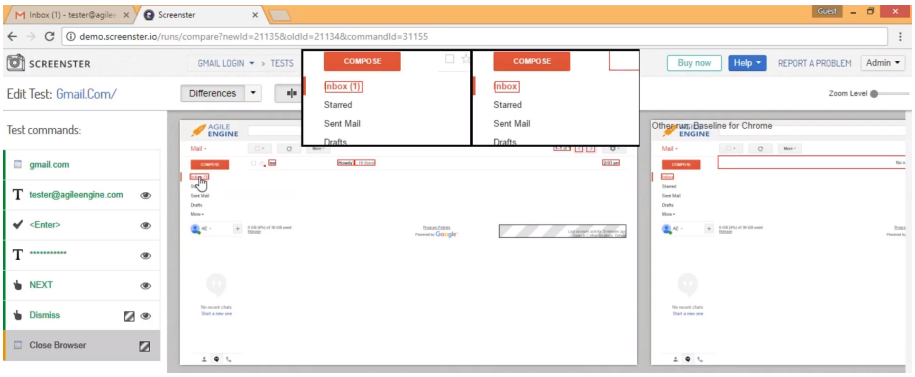

Figura 4.2: Demo test Sreenster su gmail [19]

gli abbonamenti di fascia alta. Anche la possibilità di lanciare test in parallelo è limitata alle versioni in abbonamento.

Oltre agli aspetti economici il prodotto presenta anche dei problemi in merito a stabilità e velocità. Gran parte della community riporta di numerosi crash e rallentamenti durante la registrazione e il lancio dei test.

Inoltre, come già precedentemente accennato, gli algoritmi di confronto di immagini rischiano di causare numerosi falsi negativi (o positivi) per la sensibilità utilizzata. Questo problema può essere in parte risolto grazie alla possibilità di selezionare aree di pagina da escludere durante il confronto, ma comunque rimane un aspetto piuttosto aleatorio dei test.

4.4 Altri tool

Sul mercato sono presenti numerosi altri tool per il testing automatico di applicazioni web, gran parte molto validi. Sono stati scelti tre dei tool più simbolici e significativi che possono essere considerati come rappresentanti dei prodotto con caratteristiche simili.

Descriverò brevemente alcuni degli altri tool presi in esame sottolineando che la maggior parte di questi sono disponibili solo in versione freemium e pochi sono open source. La maggior parte dei prodotti open source sono stati scartati per la poca popolarità e per la mancanza di documentazione adeguata.

Sahi

Sahi è un tool di web testing automatico disponibile in versione freemium. Funziona su tutti i principali browser senza necessità di riscrittura dei test; gestisce contenuti AJAX senza l'utilizzo di attese esplicite; permette il lancio di test in parallelo ed è dotato di una serie di strumenti per la gestione e analisi dei risultati dei test. Questo tool presenta delle caratteristiche assimilabili a quelle di Cypress (ad eccezione della possibilità di testing su diversi browser). Un articolo del 2017, che mette a confronto Sahi e Selenium [2], ci fa capire quanto le differenze tra questi tool siano irrilevanti ai fini di costruire una suite di test stabile. Sahi

è più immediato e di facile utilizzo rispetto a Selenium, che necessita invece di competenze di programmazione molto più elevate.

TestCafè

TestCafè è un altro valido tool per la registrazione, modifica e gestione di test automatici di applicazioni web. Anche questo prodotto è disponibile in versione freemium. A differenza di Sahi e Cypress, questo tool permette di registrare un test semplicemente eseguendo manualmente le operazioni da inserire nel test. Grazie ad una toolbox è possibile osservare i passaggi memorizzati e modificarli o cancellarli durante la registrazione. Screenster è il rappresentante di questo tool che però, a differenza del primo, non effettua testing visuale.

Un grande problema di TestCafè è il fatto che non riesca a gestire gli iframe (i dettagli del problema nel caso di Cypress sono spiegati nel cap 6.1), cosa assolutamente fondamentale per il progetto in esame.

4.5 Confronto tra i tool

L'analisi comparativa effettuata si è basata sia sulle necessità a breve termine del progetto sia sulle possibili evoluzioni future. Abbiamo cercato di scegliere un tool con lungimiranza, basandoci anche su aspetti come il supporto tecnico e i feedback degli utenti. Selenium è stato scartato per la sua non immediata installazione e configurazione e per i feedback non troppo positivi. La strategia di testing di Screenster è sicuramente la più innovativa tra i tool proposti, ma ancora non è del tutto affidabile e risulta pesante da gestire. Cypress è stato quindi il tool che meglio aderiva alle nostre necessità. È un software giovane (con i pro e i contro che ciò comporta), open source, con una folta community di utenti e sviluppatori molto attiva e disponibile. È un prodotto in continua evoluzione che si distacca dalla struttura, ormai piuttosto vecchia, di Selenium.

	Selenium	**Cypress**	**Screenster**
Licenza	Open-source	Open-source	Commerciale
Testing multi-browser	si	no	si
Design dei test case	no	si	si
Testing visuale	non nativamente	non nativamente	si
Servizio cloud	no	no	si
Supporto tecnico	no	no	si
Test recording	no	no	si
Continuous integration	si	si	si
Rewind durante testing	no	si	no
Lancio programmato	no	si	si

Tabella 4.1: Comparazione strumenti di testing automatico

Capitolo 5

Caso reale: Balsamiq Mockup

Durante la mia esperienza di tirocinio ho avuto modo di confrontarmi con un problema reale, di un'azienda reale, che mi ha permesso di imparare parte della difficile arte dei test automatici. Il risultato è una suite di test automatici, in grado di verificare alcune delle funzionalità del prodotto, che probabilmente verrà integrata dall'azienda. Tutto il lavoro svolto presso Balsamiq è stato strutturato seguendo i principi base del software testing (vedi sezione 3.2), analizzando attentamente l'applicazione e progettando i test in maniera strutturata, aiutati dall'esperienza di testing dell'azienda.

5.1 L'azienda

Balsamiq è un'azienda nata nel 2008 con l'obiettivo di aiutare le persone a creare software puliti e semplici da usare. Il prodotto sviluppato dall'azienda è un web editor grafico che permette di creare wireframe di interfacce per siti web, applicazioni desktop, web e mobile. L'azienda si è espansa in tutto il mondo riscuotendo un grande successo e riuscendo a creare una folta e variegata community (che conta più di 500.000 utenti). Oltre alla grande passione e competenza in merito alla User Experience, Balsamiq crede molto nella cultura aziendale, proponendo un modello strutturale e gerarchico molto moderno e famigliare.

5.2 Il software

Balsamiq Mockup è un editor di wireframe disponibile in versione cloud, desktop (per Windows e Mac) e sotto forma di plug-in su diversi siti come Google Drive e Atlassian Confluence, tramite abbonamento mensile.

Si tratta di uno strumento di wireframing in grado di replicare in digitale l'esperienza di disegnare bozzetti di interfacce utente su carta e penna. Questo prodotto permette a chiunque (dallo sviluppatore al cliente finale) di costruire graficamente bozze di interfacce di un'applicazione o di un sito web. L'editor mette a disposizione dell'utente una serie di elementi grafici che possono essere inseriti nel foglio di lavoro per comporre la propria applicazione. Ogni elemento ha delle caratteristiche a seconda del suo ruolo: un bottone avrà una forma e un colore, un'area di testo avrà delle parole scritte al suo interno e così via.

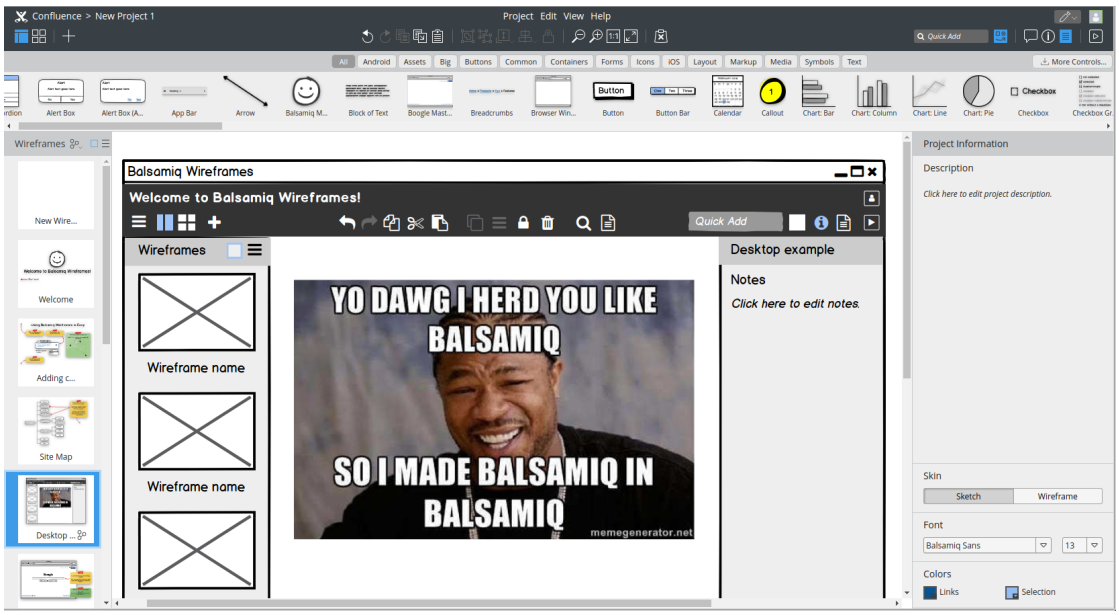

Figura 5.1: Editor Balsamiq creato tramite l'editor di Balsamiq

L'idea di base è quella di agevolare la progettazione dell'applicazione focalizzando l'attenzione sulle interfacce e accantonando tutti gli aspetti tecnici e implementativi. La semplicità e immediatezza del software lo rendono utile anche nel facilitare i rapporti tra cliente e sviluppatore: entrambe le parti possono discutere

del lavoro osservandone l'aspetto finale; la maggior parte dei dettagli implementativi saranno gestiti dallo sviluppatore che potrà invece progettare le interfacce insieme al cliente.

Balsamiq, come precedentemente accennato, è presente sotto forma di plug-in su diverse applicazioni web, tra cui Confluence. È stata proprio la versione su Confluence quella testata durante il progetto e di cui parleremo nel capitolo 6.

Confluence è un'applicazione web della suite Atlassian che aiuta a gestire la collaborazione all'interno di un team. L'applicazione permette di creare uno spazio di lavoro in cui ogni membro del team può inserire delle pagine di documentazione o discussione. Questo software permette, in breve, di creare delle Wiki interne ai team di sviluppo per agevolare la gestione di progetti e la creazione e condivisione di documentazione. È possibile implementare dei plug-in che permettono di inserire contenuti e macro nelle pagine, tra cui, appunto, Balsamiq Mockup.

Il progetto Balsamiq viene inserito nella pagina sotto forma di macro. Questa può essere configurata per visualizzare l'anteprima di uno o tutti i wireframe nel progetto.

È stato proprio questo il contesto in cui abbiamo costruito i test.

5.2.1 Caratteristiche tecniche

Balsamiq Mockup conta circa 165.000 linee di codice. La maggior parte (circa l'80%), comune a tutte le versioni, è scritta in Javascript. La versione browser è interamente implementata in Javascript mentre le versioni MacOs/iOS e Windows hanno una parte (circa il 20%) scritta rispettivamente in Swift e C#. Per svincolarsi da qualsiasi framework, è stata implementata un'interfaccia (chiamata native.js) che il codice esterno utilizza per comunicare con il core. Questa interfaccia gestisce SQLite, i font e le dimensioni dei testi, i PDF, le chiamate asincrone, le immagini, i timer, l'interfaccia utente e la real-time collaboration.

Il backend è gestito da un server interno all'azienda (chiamato BAS) che dialoga con il database e i filesystem. Esistono due versioni di BAS: una in JavaScript per la versione cloud e una per i plugin server Atlassian (javaBAS) scritta in Java (la versione Desktop non la utilizza). La prima utilizza librerie Node.js e comunica

con MySQL; la seconda usa Active Object, una libreria Java messa a disposizione da Atlassian per astrarre i database.

Gli stati di errore critico durante l'esecuzione dell'applicazione si manifestano con il *Red Screen of Death*, una pagina interamente rossa che indica il crash dell'applicazione e la sequenza di comandi che lo ha causato (mostrato qui sotto in figura 5.2).

Figura 5.2: Esempio di Red Screen of Death

5.2.2 Prototipazione e wireframing

Prima di analizzare concretamente il lavoro svolto è bene spendere due parole per definire uno dei concetti fondamentali del software testato: la differenza tra prototipazione e wireframing.

I due termini sono spesso confusi ma in realtà presentano delle differenze significative.

Per prototipazione si intende l'attività di creazione di un prototipo di un'applicazione. Tramite il prototipo è possibile mostrare in maniera realistica il design delle interfacce e le loro interazioni; a volte viene utilizzato anche per testare l'usabilità e come presentazione da mostrare al cliente.

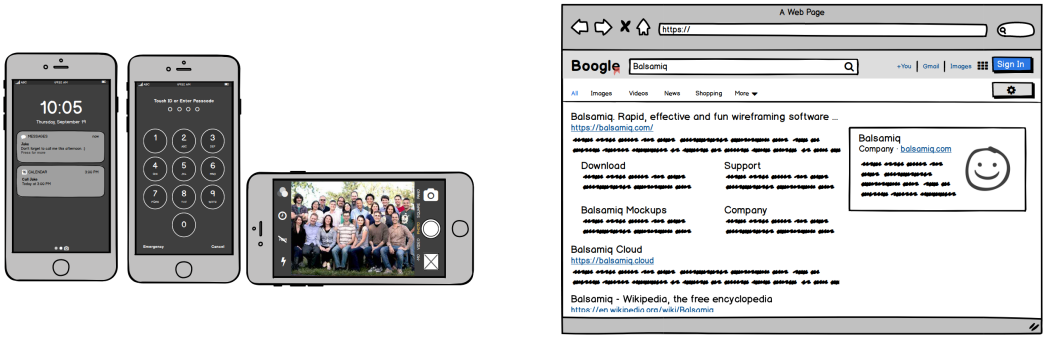

Figura 5.3: Esempi di wireframe

Il wireframing è il processo di creazione di una rappresentazione grafica a bassa fedeltà di un'interfaccia utente (il wireframe appunto) che ha lo scopo di concentrare l'attenzione sugli aspetti base del concept, sul layout, e sull'aspetto generale che dovrà aver il prodotto finito. Un wireframe, al contrario di un prototipo, non è interattivo e non mostra troppi dettagli, ma deve creare un semplice design in grado di guidare lo sviluppo del progetto [14]. Esso non è altro che uno scheletro grafico di un'interfaccia utente, che gli sviluppatori utilizzano come immagine chiara e pulita degli elementi che dovranno poi implementare. Viene utilizzato nella fase di progettazione del software, prima che qualsiasi linea di codice venga scritta. Essendo una rappresentazione grafica, e non funzionale come il prototipo, ogni aspetto implementativo viene completamente tralasciato permettendo quindi una più facile contrattazione sviluppatore-cliente.

5.3 Obiettivi dei test

L'obiettivo dei test sviluppati è quello di verificare il funzionamento del plug-in di Balsamiq su una pagina Confluence simulando le interazioni di un utente dall'inizio alla fine, ovvero puro e semplice end-to-end testing.

Il primo test sviluppato è stato il terreno di prova per prendere confidenza con il tool; abbiamo deciso di scrivere un piccolo smoke test che valutasse le funzioni base dell'applicazione.

Il test, dopo aver fatto log-in in Confluence, crea una nuova pagina e entra nell'editor di Balsamiq. A questo punto rinomina il wireframe e il progetto, aggiunge un elemento al wireframe, lo duplica, lo cancella, effettua l'undo e quindi esce dall'editor e pubblica la pagina.

Sin da subito, ci siamo scontrati con il problema principale del progetto: la presenza di due iframe (uno all'interno della pagina Confluence, uno all'interno dell'editor Balsamiq) ha ostacolato la ricerca degli elementi in essi contenuti (rimandiamo alla sezione 6.1 i dettagli).

Risolto il problema abbiamo progettato nel dettaglio tutti i test della suite; alcuni non sono stati ancora effettivamente implementati per problematiche tecniche o per limiti del software di testing.

5.3.1 Test implementati

Sono stati sviluppati cinque test, ognuno focalizzato su un aspetti diversi dell'editor Balsamiq.

- **Import-export**

  All'interno dell'editor Balsamiq è possibile importare dei wireframe sottoforma di JSON, inserendo all'interno di una casella di testo il codice JSON del wireframe. Inoltre, l'editor permette di esportare in PNG e PDF i progetti e inserirli come allegati alla pagina Confluence.

  Il test doveva verificare che queste funzionalità non dessero problemi.

- **Fullscreen**

  Oltre alla modalità editor, Balsamiq permette di visualizzare il wireframe a schermo intero, il che risulta molto utile quando si deve presentare il progetto. Nella modalità a schermo intero non è possibile modificare il wireframe e il puntatore viene trasformato in una grande freccia blu che consente di indicare gli oggetti del wireframe più facilmente.

  Aprendo una pagina Confluence che già contiene un wireframe si può accedere alla modalità fullscreen in due modi: direttamente dalla pagina Confluen-

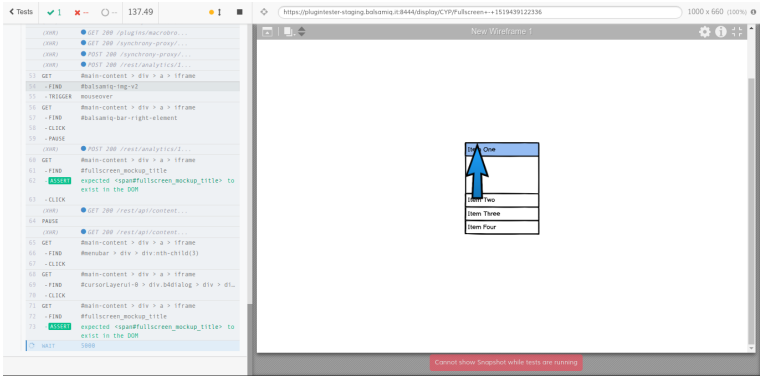

Figura 5.4: Test fullscreen

ce cliccando sull'icona sopra all'anteprima del wireframe, oppure dall'editor. Anche quest'ultimo può essere raggiunto da una pagina che già contiene un wireframe, dalla modalità fullscreen oppure cliccando su "modifica" o facendo doppio click sull'anteprima del wireframe nella pagina Confluence.

Questo test doveva verificare che tutti questi passaggi da editor a fullscreen e viceversa funzionassero correttamente.

- **Gestione wireframe e cestino**

  Ogni progetto Balsamiq può contenere uno o più wireframe. Quelli eliminati vengono memorizzati in un cestino interno all'applicazione da dove possono essere successivamente ripristinati (anche dopo la chiusura dell'editor).

  Questo test, dopo aver aggiunto due wireframe al progetto, doveva cancellarne uno verificando che fosse stato correttamente inserito nel cestino e eliminato dal progetto. Poi doveva tornare indietro di un'operazione con l'undo, verificare quindi che il cestino si fosse svuotato e che il wireframe fosse nuovamente nel progetto.

- **Testi, commenti e note**

  Un problema ricorrente nelle applicazioni e nei siti web è l'inserimento di caratteri speciali come input dell'utente; a volte, infatti, può succedere che alcuni caratteri speciali creino conflitti con il codice o, più semplicemente,

vengano visualizzati male dal browser. Uno degli obiettivi di questo test è quello di verificare che, inserendo una stringa di caratteri speciali nei principali elementi testuali dell'editor (titolo progetto, titolo wireframe, casella di testo, commenti e note) non si creino conflitti o problemi nell'applicazione.

Altre due funzioni verificate in questo test sono le note e i commenti. Balsamiq permette di inserire delle note sul wireframe, comuni e visibili a tutti gli utenti in modifica.

I commenti vengono visualizzati nell'apposito pannello con l'username dell'autore, il tasto per rispondere, l'icona per mettere il "like" e il timestamp di quando è stato inserito il commento.

Inoltre, per associare il commento all'elemento al quale si riferisce, è possibile inserire un segnalino (callback). Questo verrà visualizzato nella posizione corretta quando un utente farà passare il cursore sopra al commento.

Infine Balsamiq implementa anche un sistema di versioning dei wireframe che permette di creare delle versioni alternative del progetto corrente senza modificare l'originale.

Il test in esame doveva quindi verificare tutte queste funzionalità: modificare i testi inserendo sequenze di caratteri speciali, modificare le note, inserire e modificare un commento, una callout e aggiungere una versione alternativa.

5.3.2 Test non implementati

Tra i test in progetto ce ne sono stati alcuni che non abbiamo potuto implementare per limitazioni, temporanee e non, del tool di testing.

- **Import-export**

  Questo test, oltre agli obiettivi elencati nella sezione precedente, doveva verificare l'importazione di immagini, file e wireframe salvati localmente e l'esportazione e il download del progetto in PDF. Purtroppo però, il tool utilizzato non permette di interagire con le funzioni di sistema e tutte queste

funzionalità necessitano l'apertura di un file navigator per selezionare i file da importare o la cartella in cui scaricare il progetto.

- **Real-time collaboration**

  L'editor Balsamiq implementa un sistema di real-time collaboration (RTC) che permette a più utenti di lavorare contemporaneamente al progetto senza nessun conflitto. Inoltre è disponibile anche una chat istantanea (la cui cronologia viene cancellata dopo l'uscita dall'editor) per permettere agli utenti di comunicare durante la creazione dei wireframe.

  Per testare queste funzionalità è necessario lanciare due test in parallelo, cosa che il tool scelto per il testing non implementa ancora. Dato che la RTC è una funzionalità secondaria dell'editor, non è stato un problema evitare di sviluppare questo test.

- **Gruppi e simboli**

  All'interno dell'editor è possibile selezionare una serie di elementi e raggrupparli per gestirli come fossero un solo. È anche possibile trasformare degli elementi raggruppati, in un simbolo; in questo modo quel gruppo di elementi sarà disponibile in tutto il progetto e potrà essere inserito velocemente dalla barra degli elementi (nella sezione dedicata ai simboli). In pratica si tratta di aggiungere un elemento personalizzato a quelli già presenti nell'editor.

  Purtroppo anche queste due funzioni non sono state testate perchè, per i problemi relativi agli iframe che discuteremo nella sezione 6.1, non è stato possibile selezionare gli elementi all'interno del wireframe.

Capitolo 6

Cypress: applicazione a caso reale

In questo capitolo descriverò nel dettaglio i principali problemi riscontrati durante la stesura dei test e le soluzioni adottate. Nell'ultima parte verranno descritte le considerazioni in merito all'esperienza di utilizzo di Cypress, le qualità del tool e le mancanze che abbiamo osservato.

6.1 Problemi riscontrati

Il problema più grande durante l'utilizzo di Cypress è stato la gestione degli iframe. Tutto l'editor Balsamiq si trova all'interno di un iframe (come anche l'interno della pagina Confluence).

Per selezionare un elemento della pagina (e quindi utilizzarlo come soggetto di altri comandi) Cypress utilizza la funzione `cy.get()` che prende come parametro il selettore JQuery dell'elemento e permette di concatenare comandi.

Per esempio, per cliccare sull'elemento con ID "wireframe" basterà scrivere:

```
cy.get('#wireframe').click()
```

Per cercare un elemento all'interno di un altro si utilizza invece la funzione `find()` che agisce in modo analogo alla `cy.get()` ma deve essere concatenata a una di queste.

Il problema sta nel fatto che queste funzioni non supportano la selezione o l'accesso ad elementi interni ad un iframe perchè li considerano separati dal DOM

dato che si aspettano un document genitore diverso. Nonostante questa limitazione (che il team sta già tentando di superare), sulla pagina GitHub del programma è presente un workaround in grado di risolvere il problema.

Abbiamo quindi ridefinito la funzione `cy.get()` per cercare gli elementi all'interno dell'iframe mantenendo una sintassi simile. In realtà sono state create tre funzioni, una per ogni iframe possibile: quello di Confluence, quello dell'editor Balsamiq in una nuova pagina e quello dell'editor aperto da una pagina in modifica (Codice in Appendice B).

In questo modo, per cercare l'elemento con ID `exitButton` all'interno dell'iframe possiamo scrivere semplicemente:

```
cy.getBIF('#exitButton')
```

Nonostante questo, parte del problema risultava comunque irrisolta.

Le funzioni `cy.get()` e `find()`, infatti, implementano nativamente un sistema di polling automatico: se l'elemento non viene trovato al primo tentativo, le funzioni riproveranno più volte a intervalli di tempo regolari. Se l'elemento continua a non essere trovato, dopo una certa quantità di tempo definita nella configurazione di Cypress, verrà sollevato l'errore.

Avendo ridefinito la funzione `cy.get()` questo meccanismo è venuto a mancare costringendoci a inserire delle attese e delle pause esplicite all'interno del codice.

Il team di Cypress, come già precedentemente accennato, è però al lavoro per sistemare il problema e cercare di implementare una funzione per "entrare" e "uscire" dagli iframe. Per questo motivo abbiamo deciso, nonostante questo problema, di continuare ad utilizzare lo strumento cercando di scrivere test modulari e facilmente modificabili in vista dell'introduzione della nuova funzionalità.

Il problema degli iframe ci ha anche impedito di selezionare gli elementi grafici all'interno del foglio di lavoro di Balsamiq. Non siamo riusciti a capirne il motivo ma probabilmente, dopo l'aggiornamento sopracitato, anche questo verrà risolto.

Un altro problema riscontrato durante la stesura dei test è stata l'impossibilità di interazione con il sistema esterno al browser che non ci ha permesso di importare dei file locali all'interno di un wireframe.

Questa azione dovrebbe aprire una finestra di sistema (un file manager) che permette di selezionare il file da inserire nel wireframe. Purtroppo non siamo riusciti ad aprire tale finestra e quindi aggiungere un file esterno.

Infine, un ulteriore problema derivante dagli iframe è stata l'impossibilità di "viaggiare nel tempo" durante l'esecuzione dei test, funzionalità molto utile per il debugging in caso di errore. L'anteprima dell'interno degli iframe, infatti, non veniva visualizzata quando provavamo a selezionare un passo già concluso del test.

6.2 Conclusioni e risultati ottenuti

Nonostante i problemi sopracitati il tool si è rivelato un'ottima scelta. La documentazione è molto ricca e la community molto attiva (anche su Gitter, una chat di supporto ai progetti GitHub).

La cosa che ci ha colpito maggiormente di questo tool è stata la facilità di comprensione e semplicità del codice dei test (vedi Appendice B). Gli statement sono decisamente parlanti e la lettura di un test risulta essere molto semplice vista la loro discorsività e chiarezza.

Inoltre la possibilità di utilizzare gli strumenti da sviluppatore di Chrome è stata molto utile. Purtroppo non abbiamo usufruito molto della funzione di "viaggio nel tempo" ma è da considerarsi una funzionalità molto utile che potremo sfruttare in futuro.

La suite di test risulta essere ancora un pochino lenta, per via delle attese sopracitate, ma comunque efficiente e sicura. I test che non presentano troppe attese o pause vengono eseguiti molto velocemente e la possibilità di programmare i lanci e di salvare gli screenshot in caso di errore è molto comoda.

Scegliere Cypress è stata una scommessa, un investimento su un prodotto giovane e molto valido. Nonostante i problemi incontrati durante questa esperienza non ci siamo pentiti della scelta e abbiamo ottenuto ottimi risultati. Abbiamo sviluppato cinque test completi, che andranno sistemati dopo l'aggiornamento per la gestione degli iframe da parte di Cypress, ma che già ora funzionano più che bene.

Probabilmente Screenster, anche se con licenza a pagamento, rappresenta un degno rivale di tutti gli strumenti per il testing automatico end-to-end ma la possibilità di effettuare unit test, la solidità strutturale e la licenza open-source di Cypress lo rendono una scelta migliore.

Conclusioni

Abbiamo valutato diversi aspetti del testing automatico end-to-end nell'ambito di applicazioni web. I tool presi in esame rappresentano dei validi candidati per costruire buone suite di testing. In particolare la nostra scelta, Cypress si è rivelata essere molto valida, nonostante i problemi riscontrati, ancora giovane e con delle grandi mancanze in termini di features, ma in continua evoluzione ed aggiornamento. L'esperienza nell'utilizzo di questo strumento mi ha permesso di capire le difficoltà di sviluppo di una suite di testing solida e funzionale ed alcuni problemi generali del testing di web application, descritte in precedenza.

Il testing automatico end-to-end è ancora in fase di sviluppo e deve scontrarsi con delle problematiche non facili da superare. La soluzione a queste difficoltà, a mio parere, risiede nel testing visuale che strumenti come Screenster stanno cercando di fornire. La tecnologia non è ancora a livelli tali da garantire la giusta affidabilità e solidità degna di una suite di test ma, una volta raggiunte, il futuro del software testing sarà l'automazione e il testing manuale diventerà sempre meno conveniente.

Appendice A

Lista completa dei test

A.1 Smoke test

1. Apri editor da nuova pagina
2. Rinomina wireframe con caratteri speciali
3. Rinomina progetto con caratteri speciali
4. Aggiungi un elemento visuale al wireframe
5. Duplica l'elemento appena inserito
6. Cancella l'ultimo elemento inserito ed esegui l'undo
7. Esci dall'editor e torna sulla pagina Confluence in modifica
8. Verifica che la macro di Balsamiq esista e sia consistente

A.2 Import-export

1. Importa wireframe tramite JSON
2. Esporta PDF come allegato alla pagina Confluence
3. Esporta PNG come allegato alla pagina Confluence

4. Download progetto come PNG
5. Esci dall'editor e controlla che la pagina Confluence abbia gli allegati

A.3 Fullscreen

1. Apri pagina Confluence in modifica
2. Apri fullscreen da Confluence
3. Apri editor da fullscreen
4. Aggiungi un elemento per verificare di essere nell'editor
5. Chiudi editor e torna sulla pagina Confluence
6. Apri editor da Confluence
7. Apri fullscreen da editor
8. Chiudi editor e torna su modifica pagina
9. Apri editor da modifica pagina con doppio click
10. Chiudi editor e torna su modifica pagina
11. Apri editor cliccando sul tasto "modifica"
12. Chiudi editor

A.4 Gestione wireframe e cestino

1. Crea nuova pagina Confluence e apri editor
2. Aggiungi un wireframe al progetto
3. Aggiungi un altro wireframe

4. Sposta il primo wireframe nel cestino
5. Undo - Redo
6. Svuota il cestino
7. Chiudi editor

A.5 Testi, commenti e note

1. Crea nuova pagina Confluence
2. Aggiungi titolo alla pagina con caratteri speciali
3. Apri editor
4. Rinomina progetto con caratteri speciali
5. Inserisci un elemento di testo nel wireframe
6. Modifica il testo contenuto nell'elemento appena inserito
7. Apri commenti, scrivi un nuovo commento, inserisci una callout, modifica il commento inserito
8. Apri note, scrivi una nuova nota, crea una versione alternativa del progetto
9. Chiudi editor

Appendice B

Esempi codice Cypress

```
// Semplice test che entra in un sito, cerca l'elemento che contiene
    "type" e ci clicca sopra, quindi controlla che l'url della pagina
    sia cambiata correttamente, cerca l'elemento con classe
    "action-email" e verifica che abbia il contenuto giusto.
describe('My First Test', function() {
  it("Gets, types and asserts", function() {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

    // Should be on a new URL which includes '/commands/actions'
    cy.url().should('include', '/commands/actions')

    // Get an input, type into it and verify that the value has been
        updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
  })
})
```

```
// Ogni metodo Cypress e' equivalente alla sua controparte JQuery
cy.get('#main-content')
  .find('.article')
  .children('img[src^="/static"]')
  .first()

// Funzioni per il workaround descritto nel capitolo 6
// add "iframe" command on cypress to get the iframe in the document
Cypress.Commands.add('iframe', { prevSubject: 'element' }, $iframe => {
  return new Cypress.Promise(resolve => {
    resolve($iframe.contents().find('body'))
  })
})

// use it to get the element "selector" in the Balsamiq iframe
Cypress.Commands.add('getBIF', {}, $selector => {
  return
      (cy.get('#balsamiq_mockups_editor_iframe').iframe().find($selector))
})

// use it to get the element "selector" from the Confluence new page
    iframe
Cypress.Commands.add('getNIF', {}, $selector => {
  return (cy.get('#wysiwygTextarea_ifr').iframe().find($selector))
})

// use it to get the element "selector" from the Confluence edit page
    iframe
Cypress.Commands.add('getEIF', {}, $selector => {
  return (cy.get('#main-content > div > a >
      iframe').iframe().find($selector))
})
```

Bibliografia

Articoli

[1] Nazia Islam, *A Comparative Study of Automated Software Testing Tools*, Culminating Projects in Computer Science and Information Technology, 12, 2016, http://repository.stcloudstate.edu/csit_etds/12.

[2] Arjun Satheesh, Monisha Singh, *Comparative Study of Open Source Automated Web Testing Tools: Selenium and Sahi*, Indian Journal of Science and Technology,Vol 10(13), 2017, 10.17485/ijst/2017/v10i13/109048.

[3] Laurent Christophe, Reinout Stevens, Coen De Roover, *Prevalence and Maintenance of Automated Functional Tests for Web Applications*, Software Maintenance and Evolution (ICSME), 2014, 10.1109/ICSME.2014.36.

[4] Capgemini, Micro Focus, Sogeti, *World Quality Report: 2017 2018 Ninth Edition*, 2017.

[5] Joe Fernandes, Alex Di Fonzo, *When to Automate Your Testing (and When Not To)*, 2013.

[6] S.S.Riaz Ahamed, *Studying the Feasibility and Importance of Software Testing: An Analysis*, IJEST Volume 1 Issue 3 2009, 119-128, 2009.

[7] W.T. Tsai, Xiaoying Bai, Ray Paul, Weiguang Shao, Vishal Agarwal, *End-to-End Integration Testing Design*, 2011.

Libri

[8] Capers Jones, Olivier Bonsignour, *The Economics of Software Quality*, Addison-Wesley, Massachusetts, 2011.

[9] Capers Jones, *Software Methodologies: A Quantitative Guide*, Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, 2018.

[10] Tim Riley, Adam Goucher, *Beautiful Testing: Leading Professionals Reveal How They Improve Software*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2010.

Siti

[11] Wikipedia, "Software Testing", `https://en.wikipedia.org/wiki/Software_testing`

[12] GitHub, "Cypress", `https://github.com/cypress-io/cypress`

[13] Wikipedia, *"Report: Software Failures cost $1.1 trillion in 2016"*, `http://servicevirtualization.com/report-software-failures-cost-1-1-trillion-2016/`

[14] UX Mastery, "Wireframing for beginners", `https://uxmastery.com/wireframing-for-beginners/`

[15] Pagina Gitter di Balsamiq, `https://gitter.im/cypress-io/cypress`

Immagini

[16] Tabella cost bug fix, `http://www.qalab.co/blog/tips_to_avoid_the_huge_cost_of_fixing_a_software_bug`

[17] Grafico Test Driven Development, `https://it.wikipedia.org/wiki/Test_driven_development`

[18] Vantaggi Cypress, `https://www.cypress.io/how-it-works`

[19] Demo test Screenster, `https://screenster.io/documentation/demo-automation-of-gmail-with-screenster`

[20] Architettura Selenium Web Driver, `https://www.edureka.co/blog/what-is-selenium`

Ringraziamenti

Per prima cosa ringrazio i miei genitori, Liviana e Luciano, per avermi dato la possibilità di raggiungere questo traguardo, per avermi sempre sostenuto nonostante le difficoltà, per avermi riempito le valige di cibo ad ogni rientro e per avermi reso quello che sono.

Ringrazio i miei nonni per avermi cresciuto: in particolare mio nonno Giulio e mia nonna Rosetta, due dei miei punti di riferimento, per avermi tramandato la loro esperienza e "semplicità", per avermi insegnato ad essere onesto e generoso, per avermi aspettato ogni giorno con un piatto caldo dopo scuola e per avermi trasmesso l'amore per la famiglia.

Ringrazio mio fratello, Mattia, per avermi offerto consulenze informatiche quando ne avevo bisogno e per tutte le torture che mi ha inflitto (un pezzo di puzzle nel lettore floppy è effettivamente difficile da perdonare). Ringrazio anche Debora che lo sopporta.

Ringrazio la mia ragazza, Maria Clara, per avermi sopportato in questi meravigliosi anni e nei momenti più carichi di ansia e nervosismo (leggi "sempre").

Ringrazio Francesca e Fausto per tutti gli esami preparati insieme, per le serate di svago e per il continuo supporto tecnico e psicologico di questi anni.

Ringrazio Andre e Seba per esserci da sempre, punti fissi nelle serate alla play.

Ringrazio la mia Balla, la famiglia in trasferta, il Convento de li Frati Gaudenti, per avermi cresciuto e aver arricchito la mia permanenza a Bologna; con voi ho passato delle serate fantastiche che mai scorderò.

Ringrazio tutti i miei professori per avermi fatto soffrire e per avermi guidato al meglio in questo cammino.

Ringrazio Balsamiq per la bellissima esperienza di tirocinio offertami e per avermi dato una grande dimostrazione di cultura aziendale.

Ringrazio tutti gli amici e i parenti che hanno creduto in me e che mi vogliono bene, ognuno è stato fondamentale per rendermi la persona che sono.

Grazie!