Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Testing integrato di architetture micro front-end

Sviluppo e testing di un'applicazione Angular in un contesto DevOps

Relatore:
Prof. Luca Ardito

Candidato:
Isabella Romita

Anno accademico 2019/20
Torino

Sommario

***Contesto*:** Nell'ambito dell'ingegneria del software, il testing ricopre un ruolo fondamentale. E' un'attività costosa e che richiede anche molto tempo, a prescindere dal fatto che venga svolta in un contesto aziendale o meno. Nel corso degli anni infatti, il software testing è diventato sempre più complicato, a causa della vasta gamma di linguaggi di programmazione, sistemi operativi e piattaforme hardware che si sono evolute rapidamente. Praticamente tutto ciò che ci circonda possiede una componente software, ed è proprio per questa ragione che diventa essenziale il ruolo del testing.

***Obiettivo*:** Lo scopo di questa tesi è analizzare le varie tecniche di testing a livello front-end, focalizzando l'attenzione su test unitari e su test integrati, con l'obiettivo finale di capire come tali tecniche si sposino con un contesto DevOps.

***Metodo*:** Partendo dallo sviluppo di un'applicazione Angular, la tesi realizza una Continuous Integration/Continuous Delivery pipeline tramite le *GitHub Actions* messe a disposizione da GitHub stesso. Il flusso di lavoro parte dal testing statico, per poi focalizzare l'attenzione su test unitari, effettuati tramite il framework *Jest*, e test end-to-end, realizzati tramite *Cypress*, e infine terminare con il deploy su Firebase, una piattaforma di hosting online, solo in seguito al superamento di tutti i test.

***Risultati*:** La pipeline realizzata è in grado di separare perfettamente l'ambiente di testing da quello di sviluppo, con processi di testing che riescono a raggiungere una code coverage totale superiore al 90%. Inoltre tutta la reportistica in merito ai risultati ottenuti è disponibile online.

***Conclusioni*:** Gli strumenti utilizzati, nonostante il loro recente ingresso nel mercato, si sono rivelati adatti alla realizzazione di una CI/CD pipeline. Per quanto riguarda la fase finale di Deploy, potrebbe essere anche utilizzata una piattaforma diversa, o anche la versione a pagamento di quella presa qui in esame, per permettere l'hosting di applicazioni più grandi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Il software testing

L'attività di testing è una delle fasi fondamentali appartenenti al ciclo di vita di un software, ed ha come scopo quello di elevarne la qualità e verificare che il comportamento sia il più in linea possibile con quanto richiesto. Il superamento di tutti i test non garantisce che il software, una volta rilasciato, funzionerà come dovrebbe, ma è utile per ridurne la difettosità.

Un'errata percezione comune del testing è che consista solo nell'esecuzione di test, cioè nell'esecuzione del software e nel controllo dei risultati. Tuttavia questo processo include molte attività differenti, e quella appena citata è solo una di queste. Altre sono la pianificazione, l'analisi, la progettazione e l'implementazione dei test, il controllo dell'avanzamento e i risultati, e la valutazione della loro qualità.

Alcuni test richiedono l'esecuzione del componente o del sistema in fase di test: questo tipo di testing è detto *dinamico*. Altri invece consistono in un'analisi preliminare del codice, senza la necessità di eseguirlo: questo tipo di testing è invece *statico*.

Un'altra credenza sbagliata è quella che prevede che il testing si concentri interamente sulla verifica dei requisiti, delle user stories o altre specifiche, ma questa è solo una delle sue principali attività. Il testing comprende anche la *validazione*, ovvero la verifica che il sistema soddisfi sia i bisogni dell'utente sia degli altri stakeholders nel proprio ambiente operativo.

Al giorno d'oggi, qualsiasi tipo di software viene realizzato in team, questo significa che è necessaria una costante comunicazione tra tutti i suoi membri, ogni qualvolta viene apportata una modifica. Per evitare che delle modifiche successive compromettano delle funzionalità già precedentemente testate e funzionanti, si ricorre al *Regression Testing* che consiste nella ri-esecuzione, totale o parziale, dei test precedentemente scritti, per assicurarsi che i cambiamenti più recenti non abbiano introdotto nuovi bugs (in caso contrario siamo appunto in presenza di una *regressione*).
Essendo quella del Regression Testing una tipologia di testing che può richiedere molto tempo, è diventato sempre più comune il processo di automazione della stessa all'interno del contesto di *Continuous Integration* e *Continuous Delivery*. Entrambi i concetti si traducono nella creazione di una pipeline che prima esegue i test sulle nuove funzionalità, e solo in seguito al loro superamento, è in grado di integrarle all'interno dell'applicazione.

Componenti chiave di questo concetto sono i test automatici. E' necessario quindi, a tale scopo, scrivere degli opportuni script di test, ovvero un insieme di istruzioni che vengono eseguite sul sistema, per verificarne il corretto comportamento. La corretta scrittura di questi test permette di risparmiare sia tempo che risorse, nel momento in cui il sistema cambia, in seguito ad una modifica, e uno o più test falliscono. In questo modo sarà più semplice per lo sviluppatore, risolvere il problema o, perlomeno, identificarne la fonte di provenienza. Questa fase può essere molto critica, in quanto le correzioni effettuate a seguito di un test non funzionante, potrebbero causare il fallimento di un test che invece prima non dava problemi. A tal proposito, i test di regressione possono quindi diventare una componente considerevole del progetto, in termine di righe di codice.

1.1.1 Benefici e rischi dell'automazione dei test

Ogni nuovo tool introdotto in un'organizzazione richiede uno sforzo per ottenere benefici reali e duraturi. L'utilizzo di tools per eseguire i test comporta potenziali vantaggi e opportunità, ma ci sono anche rischi. I potenziali vantaggi dell'utilizzo di strumenti a supporto dell'esecuzione dei test includono:

- Riduzione del lavoro manuale ripetitivo (ad esempio: esecuzione di test di regressione, creazione e distruzione dell'ambiente di testing, inserendo nuovamente gli stessi dati di prova e controllando gli standard di codifica), che si traduce in un risparmio in termini di tempo.

- Maggiore coerenza e ripetibilità (ad esempio: i dati dei test sono creati in modo coerente, i test sono eseguiti da uno strumento nello stesso ordine con la stessa frequenza, e i test sono costantemente basati sui requisiti).

- Valutazione più obiettiva (ad esempio: misure statiche, coverage).

- Più facile accesso alle informazioni sui test (ad esempio: statistiche e grafici sui progressi dei test, sui tassi di difettosità e sulle prestazioni).

Ci sono anche una serie di rischi:

- Le aspettative per lo strumento possono essere irrealistiche (inclusa la funzionalità e la facilità d'uso).

- Il tempo, il costo e l'impegno per l'introduzione iniziale di uno strumento possono essere sottovalutati (compresa la formazione e le competenze esterne).

- Il tempo e l'impegno necessari per ottenere benefici significativi e continui dallo strumento possono essere sottovalutati (compresa la necessità di modificare il processo di prova e di migliorare continuamente il modo in cui lo strumento viene utilizzato).

- Si tende a fare troppo affidamento sul tool (visto come un sostituto per la progettazione o l'esecuzione del test, o l'uso di test automatizzati dove il test manuale sarebbe migliore).

- Il fornitore dello strumento può cessare l'attività, ritirare lo strumento o vendere lo strumento a un altro fornitore.

- Il fornitore può fornire una risposta inadeguata per il supporto, gli aggiornamenti e la correzione dei difetti.

- Un progetto *open source* può essere sospeso.

- Una nuova piattaforma o tecnologia potrebbe non essere supportata dallo strumento.

- Potrebbe non esserci una chiara proprietà dello strumento (ad esempio, per il tutoraggio, gli aggiornamenti, ecc.). ("Foundation Level Syllabus, ISTQB", 2019)

1.1.2 I diversi tipi di test

Per quanto riguarda i test automatici, (Pittet, 2019) distingue diverse tipologie:

- *Test unitari:* sono quelli di livello più basso. Consistono nel test di singoli metodi e funzioni delle classi, dei componenti o dei moduli utilizzati dal software, isolati dal sistema di cui fanno parte. Per fare ciò vengono spesso utilizzati i *test doubles*, che permettono di testare il comportamento dell'unità in reazione a specifici stimoli, al fine di verificare che corrisponda a quello previsto.

- *Test di integrazione:* verificano che i diversi moduli o servizi utilizzati dall'applicazione funzionino bene insieme. Ad esempio, si può testare l'interazione con il database o assicurarsi che i microservizi funzionino insieme come previsto. Questi tipi di test sono più costosi da eseguire in quanto richiedono l'esecuzione di più parti dell'applicazione.

- *Test funzionali:* si concentrano sui requisiti di business di un'applicazione. Verificano solo l'output di un'azione e non controllano gli stati intermedi del sistema durante l'esecuzione di tale azione.

- *Test end-to-end:* replicano il comportamento di un utente con il software in un ambiente applicativo completo e permettono di verificare che diversi scenari funzionino come previsto. Questi possono essere semplici come il login all'interno di una pagina Web o molto più complessi come la ricezione di notifiche tramite email, pagamenti online e così via.

- *Test di accettazione:* sono test formali eseguiti per verificare se un sistema soddisfa i requisiti di business, infatti richiedono che l'intera applicazione sia attiva. Si focalizzano maggiormente sulle funzionalità più sollecitate da parte degli utenti e/o quelle con più alto valore di business. Possono anche andare oltre e misurare le prestazioni del sistema, ed eventualmente non approvare le modifiche se determinati obiettivi non vengono raggiunti.

- *Test di performance:* Verificano i comportamenti del sistema quando è sotto stress. Questi test non sono funzionali e possono essere eseguiti in vari modi per stabilire l'affidabilità, la stabilità e la disponibilità della piattaforma. Ad esempio, si possono osservare i tempi di risposta durante l'esecuzione di un numero elevato di richieste, o vedere come il sistema si comporta con una quantità significativa di dati.

1.2 Agile testing

L'agile testing è una tipologia di testing dinamico, che permette di svolgere le attività di test in maniera regolare durante tutto il processo di sviluppo. Per questo è importante individuare subito l'approccio da applicare: quali test eseguire, come e quando eseguirli. La soluzione può essere ricercata attraverso il concetto di *agile testing quadrants*, che verrà spiegato a breve. Questo strumento, delineato da Brain Marick, divide l'intera metodologia di test agile in quattro quadranti e dispone i tipi di test da eseguire su livelli diversi, per adattarsi al meglio a quello che viene considerato il *manifesto agile.*

1.2.1 Il *Manifesto Agile*

Il Manifesto Agile contiene quattro dichiarazioni di valori:

- Individui e interazioni *al di sopra* di processi e strumenti
- Software funzionante *al di sopra* di una documentazione completa
- Collaborazione con il cliente *al di sopra* della negoziazione del contratto
- Rispondere al cambiamento *al di sopra* del seguire un piano

Il manifesto sostiene che, sebbene i concetti a destra abbiano valore, quelli a sinistra hanno un valore maggiore.

Individui e interazioni. Lo sviluppo agile è molto incentrato sulle persone. Sono i team a costruire il software, ed è attraverso la comunicazione e l'interazione continue, piuttosto che tramite strumenti o processi, che i team possono lavorare nel modo più efficace.

Software funzionante. Dal punto di vista del cliente, un software funzionante è molto più utile e prezioso di una documentazione troppo dettagliata, e fornisce l'opportunità di dare al team di sviluppo un rapido feedback. Inoltre, poiché il software funzionante, anche se con funzionalità ridotte, è disponibile molto prima nel ciclo di vita dello sviluppo, il metodo Agile può conferire un significativo vantaggio in termini di *time-to-market*. Lo sviluppo agile è, quindi, particolarmente utile in ambienti di business in rapida evoluzione dove i problemi e/o le soluzioni non sono chiare o dove l'azienda desidera innovare in nuovi settori problematici.

Collaborazione con il cliente. I clienti spesso hanno grandi difficoltà ad entrare nel dettaglio del sistema che richiedono. Collaborare direttamente con il cliente migliora la possibilità di capire esattamente quali sono i suoi bisogni. Per quanto sia importante chiudere un contratto con il cliente, lavorare in regolare e stretta collaborazione con loro, è più probabile che porti successo al progetto.

Rispondere al cambiamento. Il cambiamento è inevitabile nei progetti software. L'ambiente in cui opera l'azienda, la legislazione, l'attività dei concorrenti, i progressi tecnologici e altri fattori possono avere importanti influenze sul progetto e sui suoi obiettivi. Questi fattori devono essere adattati dal processo di sviluppo. In quanto tale, avere flessibilità nelle pratiche di lavoro per abbracciare il cambiamento è più importante del semplice aderire rigidamente a un piano. ("Foundation Level Extension Syllabus Agile Tester, ISTQB", 2014)

1.2.2 Agile Testing Quadrants

La figura 1.1 rappresenta i 4 quadranti del testing Agile.

Il funzionamento di ciascun quadrante è spiegato di seguito (ProfessionalQA, 2019):

- **Primo quadrante (Q1):** riguarda la qualità del codice e dei componenti. Coinvolge gli sviluppatori nell'implementazione delle tecnologie e dell'automazione, ovvero test basati sulla tecnologia per supportare il team. Copre i test unitari e i test dei componenti del sistema a livello unitario. Tuttavia, la partecipazione degli sviluppatori al Q1 non limita l'ingresso e il lavoro di

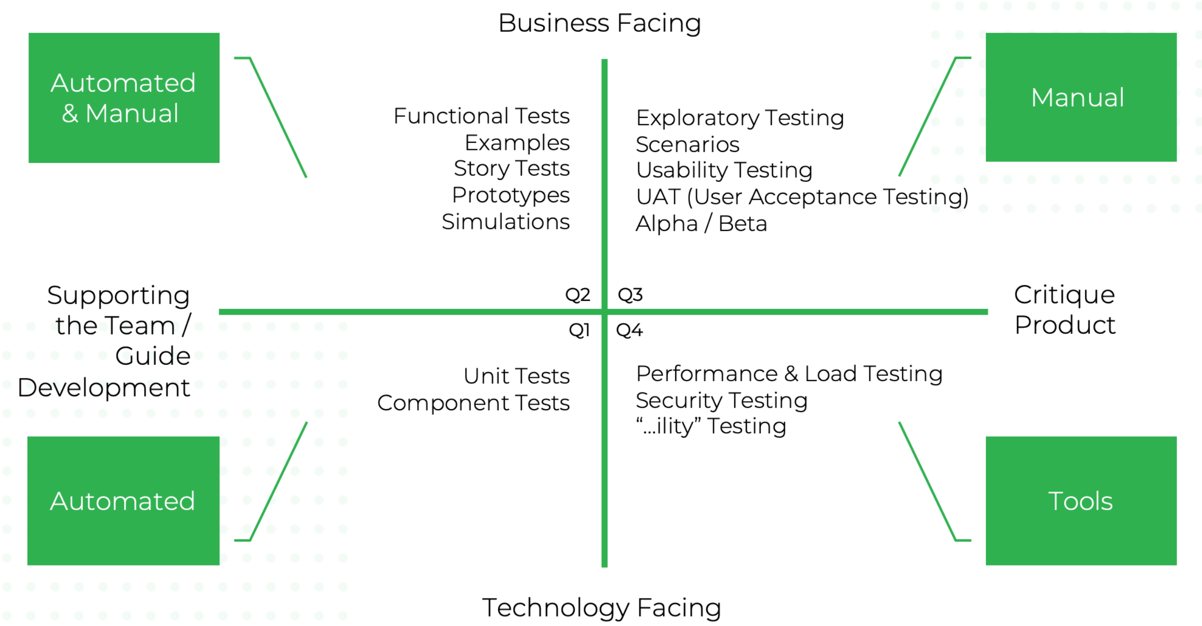


Figura 1.1. Agile Testing Quadrants.

un tester. Un tester può passare attraverso il lavoro di documentazione per comprendere meglio la struttura interna del prodotto software che può aiutare nella progettazione di test efficaci.

- **Secondo quadrante (Q2):** in questo quadrante è possibile che vengano implementati test orientati al business per supportare il team. Fondamentalmente, l'obiettivo principale di questo quadrante sono i requisiti di business. Generalmente, implica l'uso di test sia manuali che automatizzati ed è costituito da test funzionali, story test, prototipi e simulazioni. Un tester interagisce continuamente con il cliente o le parti interessate per raccogliere e implementare i requisiti richiesti nei test.

- **Terzo quadrante (Q3):** in questo quadrante i feedback e le recensioni delle fasi di test o dei quadranti precedenti, insieme alla reale esperienza dell'utente, sono la base per testare il prodotto software, utilizzando più scenari. Il testing manuale viene utilizzato per valutare l'applicazione software in base ai requisiti dell'utente e al pensiero logico di un tester. Questo quadrante viene utilizzato per costruire e acquisire fiducia nell'usabilità del prodotto. I tipi di test coperti da questo quadrante sono test esplorativi, test di usabilità, test di accettazione, test alfa e beta.

- **Quarto quadrante (Q4):** prevede l'utilizzo della tecnologia, ad esempio strumenti, per automatizzare il processo di valutazione del prodotto software a livello di accettazione operativa, per soddisfare i requisiti non funzionali, quali affidabilità, sicurezza, compatibilità, manutenibilità, interoperabilità, recupero, ecc. I tipi di test coperti da questo quadrante consistono in test delle prestazioni, test di carico, test di stress, test di manutenibilità, test di sicurezza, test di affidabilità, test di recupero e altri tipi di test, utilizzati per verificare e validare i requisiti non funzionali del prodotto software.

1.2.3 Differenze tra il testing tradizionale e quello Agile

Una delle principali differenze tra i cicli di vita tradizionali e quelli agili è l'idea di iterazioni molto brevi, ognuna delle quali si traduce in un software funzionante che offre caratteristiche di valore per i business stakeholders. All'inizio del progetto c'è un periodo di pianificazione del rilascio. Questo è seguito da un sequenza di iterazioni. All'inizio di ogni iterazione, c'è un periodo di pianificazione dell'iterazione. Una volta che viene stabilito l'ambito di iterazione, vengono sviluppate le user stories selezionate, integrate con il sistema, e testate. Queste iterazioni sono altamente dinamiche, con lo sviluppo, l'integrazione e le attività di test che avvengono ad ogni iterazione, con la presenza di un notevole parallelismo e sovrapposizione. Le attività di test hanno luogo durante tutta l'iterazione, e non come attività finale.

Tester, sviluppatori e stakeholder aziendali hanno tutti un ruolo nei test, come per i cicli di vita tradizionali. Gli sviluppatori eseguono test unitari mentre sviluppano le caratteristiche delle user stories. I tester testano poi queste caratteristiche. Anche gli stakeholders aziendali testano le storie durante l'implementazione. Possono utilizzare test cases scritti, ma possono anche semplicemente sperimentare e utilizzare la funzionalità per fornire un rapido feedback al team di sviluppo.

In alcuni casi, si verificano periodicamente iterazioni di indurimento o stabilizzazione per risolvere eventuali difetti persistenti e altri problemi tecnici. Tuttavia, la migliore pratica è che nessuna caratteristica è considerata completata fino a quando non è stata integrata e testata con il sistema. Un'altra buona pratica è affrontare i difetti rimasti dalla precedente iterazione all'inizio di quella successiva. Tuttavia, alcuni lamentano che questa pratica porti ad una situazione in cui il lavoro totale da svolgere nell'iterazione corrente è sconosciuto, e sarà quindi più difficile stimare in quanto tempo le caratteristiche rimanenti possono essere implementate. Alla fine della sequenza di iterazioni, ci possono essere una serie di attività di rilascio per preparare il software alla consegna, anche se in alcuni casi questa avviene alla fine di ogni iterazione.

Quando viene utilizzato il *risk-based testing* come strategia di test, si verifica un'analisi del rischio di alto livello durante la pianificazione del rilascio, con i tester che spesso guidano l'analisi. Tuttavia, i rischi specifici associati ad ogni iterazione sono identificati e valutati nella pianificazione dell'iterazione. Questa analisi dei rischi influenza la sequenza di sviluppo, nonché la priorità e la profondità dei test delle caratteristiche. Influenza anche la stima del *test effort* richiesto per ogni caratteristica.

In alcune pratiche Agile (ad esempio, *Extreme Programming*), si usa il *pairing*. Questo può coinvolgere i tester lavorando insieme a due a due per testare una caratteristica, oppure un tester che lavora in collaborazione con uno sviluppatore per creare e testare una funzione. L'automazione dei test a tutti i livelli di testing avviene in molti team Agile, e questo può significare che i tester passano il tempo a creare, eseguire, monitorare e mantenere i test e i risultati automatizzati.

Un principio fondamentale del metodo Agile è che il cambiamento può avvenire nel corso del progetto. Pertanto, è preferibile produrre una documentazione leggera dei prodotti di lavoro. Le modifiche alle caratteristiche esistenti hanno implicazioni nel testing, in particolare nei test di regressione. L'uso di test automatizzati è un modo per gestire la quantità di test effort associato al cambiamento. Tuttavia, è importante che il tasso di cambiamento non superi la capacità del team di gestire i rischi associati a tali cambiamenti.

1.3 Il concetto di Continuous Integration

Consiste nell'automatizzazione del processo di integrazione di eventuali modifiche del codice da parte di uno o più sviluppatori che stanno lavorando sullo stesso progetto. Il processo di CI comprende una serie di strumenti automatici che verificano la correttezza del codice prima che esso venga integrato. Per capire l'importanza di questo concetto, è utile discutere quali sono i problemi che potrebbero nascere dalla sua assenza. Innanzitutto sarebbe necessaria una forte coordinazione sia tra i membri del team che si occupa del progetto, sia con il resto dell'organizzazione, come per esempio con il team di produzione che deve periodicamente lanciare i nuovi rilasci.

La comunicazione all'interno di un ambiente non-CI rischia di diventare molto complessa, e ciò aggiungerebbe inutili costi burocratici al progetto. Questo porterebbe a rilasci più lenti con alte probabilità di fallimento, in quanto richiederebbe agli sviluppatori di essere molto più prudenti in merito alle integrazioni. Questi rischi ovviamente aumentano esponenzialmente nel momento in cui sia il team sia la dimensione del codice crescono. L'introduzione della CI nel suddetto scenario consente agli sviluppatori di software di lavorare in modo autonomo su più funzionalità in parallelo. Quando si è pronti ad unire queste funzionalità nel prodotto finale, lo si può fare in modo indipendente e rapido.

Il concetto di CI è generalmente utilizzato nell'ambito di *agile software development workflow*. Viene prima stilata una lista di tutte le attività necessarie al rilascio del prodotto. Queste attività vengono quindi distribuite tra i membri del team per la consegna. L'utilizzo della CI consente a queste attività di essere sviluppate in modo indipendente e in parallelo tra gli sviluppatori assegnati. Una volta completata un'attività, lo sviluppatore la introduce nel sistema CI e la integra con il resto del progetto.

1.3.1 Continuous Delivery e Continuous Deployment

La Continuous Integration, il Continuous Delivery e il Continuous Deployment sono le tre fasi di una pipeline di rilascio software automatizzata. Queste tre fasi portano il software dall'idea alla consegna all'utente finale. La fase di integrazione è il primo passo del processo ed è quella che è stata appena descritta.

La **Continuous Delivery** è la sua continuazione. La fase di consegna è responsabile del "confezionamento" del prodotto da rilasciare agli utenti finali. In questa fase vengono utilizzati tools di compilazione automatici e il prodotto dovrebbe essere pronto per essere distribuito agli utenti in qualsiasi momento.

Il **Continuous Deployment** è la fase finale della pipeline. E' responsabile dell'avvio e della distribuzione automatici del prodotto software agli utenti finali. Ciò avverrà tramite script o strumenti che lo spostano automaticamente su server pubblici o su altri meccanismi di distribuzione, come un app store.

1.3.2 GitHub e le GitHub Actions

GitHub è una delle più famose e utilizzate piattaforme per lo sviluppo collaborativo di software, utile quindi per sincronizzare più persone che lavorano contemporaneamente sullo stesso progetto. E' basato su Git, un software di controllo di versione, ovvero in grado di gestire gli aggiornamenti di un progetto senza sovrascriverne nessuna parte. Nel caso di modifiche che non danno l'effetto sperato, è sempre possibile tornare indietro, alla versione precedente che era invece funzionante. GitHub tiene traccia di tutte le vecchie versioni nel proprio repository, così da poterle recuperare in caso di necessità. Inoltre, per ogni utente al lavoro viene creata una differente versione del progetto, così da non creare fastidiose sovrapposizioni o sovrascritture.

Le GitHub Actions costituiscono una nuova funzionalità recentemente integrata in GitHub, che consente di automatizzare alcune delle operazioni che vengono effettuate più comunemente. E' quindi possibile, tramite un file yaml, elencare una serie di comandi da eseguire in sequenza, che ci permettono di realizzare in maniera semplice un workflow di tipo CI/CD.

1.4 DevOps

DevOps è una cultura che promuove la collaborazione tra il *Development* team e l'*Operations* team, per fare in modo che il software arrivi alla fase di produzione il più rapidamente possibile, in modo automatizzato e ripetibile. La parola "DevOps" è infatti la combinazione delle due parole *Development*, che sta per sviluppo, e *Operations*, ovvero operazioni.

DevOps aiuta ad aumentare la produttività di un'organizzazione che fornisce applicazioni e servizi. Consente alle organizzazioni di servire meglio i propri clienti e di competere più fortemente sul mercato. Può essere definito come un allineamento delle operazioni di sviluppo e IT attraverso una migliore comunicazione e collaborazione.

Perchè è necessario:

- Prima di DevOps, i due teams, di sviluppo e operativo, lavoravano in completo isolamento e, avendo tempistiche differenti, era difficile rimanere sincronizzati, e questo era causa di ulteriori ritardi.
- Il testing e la distribuzione erano attività isolate, eseguite dopo la progettazione, che quindi consumavano più tempo rispetto ai cicli di costruzione reali.
- Senza utilizzare DevOps, i membri del team trascorrono gran parte del loro tempo a testare, distribuire e progettare, invece di costruire il progetto.
- Effettuando il deploy manualmente, è più facile commettere errori nella fase di produzione.

C'è una forte richiesta per l'aumento del tasso di consegna del software da parte delle aziende. Secondo uno studio della Forrester Consulting, solo il 17% dei team è in grado di soddisfare queste tempistiche. E questo dimostra quanto è importante lo strumento che stiamo analizzando.

1.4.1 Perchè utilizzare DevOps

Nella tabella 1.1 sono riassunte le differenze principali tra la metodologia DevOps e quella tradizionale.

DevOps consente quindi ai team di sviluppo *agile* di implementare i meccanismi di Continuous Integration e Continuous Delivery, velocizzando il lancio dei prodotti sul mercato.

Altri motivi per cui utilizzare DevOps sono elencati di seguito:

- **Prevedibilità**: DevOps offre un tasso di errore dei nuovi rilasci notevolmente inferiore .

Tradizionale	DevOps
Dopo aver effettuato un ordine per nuovi server, il team di sviluppo lavora sui test. Il team operativo lavora sulle pratiche burocratiche come richiesto dalle aziende per implementare l'infrastruttura.	Dopo aver effettuato un ordine per nuovi server, il team di sviluppo e quello operativo lavorano insieme sui documenti per impostarli. Ciò si traduce in una migliore visibilità dei requisiti di infrastruttura.
Le proiezioni su failover, ridondanza, ubicazioni dei data center e requisiti di archiviazione sono distorte in quanto non sono disponibili input da sviluppatori che hanno una profonda conoscenza dell'applicazione.	Le proiezioni su failover, ridondanza, disaster recovery, ubicazioni dei data center e requisiti di archiviazione sono piuttosto precise a causa degli input degli sviluppatori.
Il team operativo non ha idea dei progressi del team di sviluppo e, di conseguenza, organizza un piano di monitoraggio basato su quello che riesce a comprendere.	Il team operativo è completamente consapevole dei progressi che gli sviluppatori stanno facendo, e sviluppano insieme un piano di monitoraggio adatto alle esigenze IT e aziendali, utilizzando anche strumenti avanzati di monitoraggio delle prestazioni delle applicazioni (APM).
Prima del rilascio, il test di carico crea un arresto anomalo dell'applicazione. Il rilascio è ritardato.	Prima del rilascio, il test di carico rende l'applicazione un pò lenta. Il team di sviluppo risolve rapidamente i colli di bottiglia, e l'applicazione viene rilasciata in tempo.

Tabella 1.1. Differenze tra il metodo tradizionale e quello DevOps

- **Riproducibilità**: è importante creare sempre una nuova versione, in modo che la quella precedente possa essere ripristinata in qualsiasi momento.
- **Manutenibilità**: processo di ripristino senza sforzo in caso di arresto anomalo di una nuova versione o disattivazione del sistema corrente.
- **Time-to-market**: DevOps riduce il *time-to-market* fino al 50% grazie alla consegna semplificata del software. Questo è particolarmente rilevante per le applicazioni digitali e mobili.
- **Maggiore qualità**: DevOps aiuta il team a fornire una migliore qualità dello sviluppo delle applicazioni in quanto include anche i problemi di infrastruttura.
- **Rischio ridotto**: DevOps incorpora aspetti di sicurezza nel ciclo di vita della consegna del software, aiutando a ridurne i difetti.
- **Resilienza**: lo stato operativo del software è più stabile, sicuro e le modifiche possono essere verificate.

- **Efficienza dei costi**: DevOps offre efficienza dei costi nel processo di sviluppo del software, che è sempre un'aspirazione della gestione delle aziende IT.
- **Suddivisione del codice**: DevOps si basa sul metodo di programmazione agile, pertanto consente di suddividere pezzi di codice più grandi in blocchi più piccoli e gestibili.

1.4.2 Ciclo di vita

DevOps è un'integrazione tra sviluppo e operazioni.

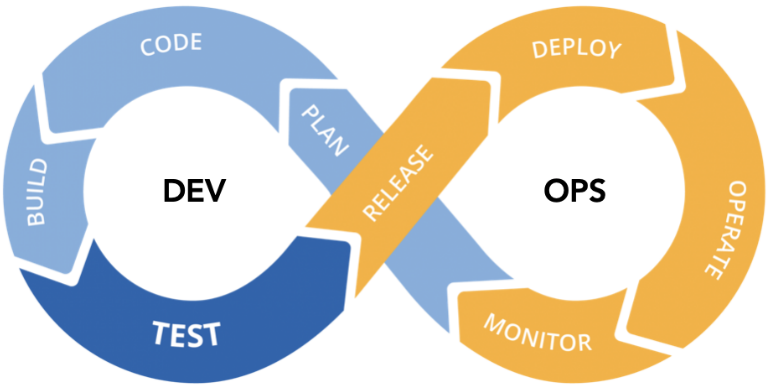


Figura 1.2. DevOps lifecycle.

Il suo ciclo di vita è composto dalle seguenti fasi, illustrate nella figura 1.2:

1. ***Plan***. In questa fase avviene l'analisi dei requisiti, e la suddivisione del lavoro in tasks.
2. ***Code***. Si procede con lo sviluppo del codice, e la pianificazione dei test.
3. ***Build***. Indica la fase in cui il codice appena implementato viene compilato, attraverso strumenti di Continuous Build/Continuous Integration.
4. ***Test***. Vengono eseguiti i test tramite tools specifici, per identificare e correggere i bug nella nuova porzione di codice.

5. ***Release***. In questa fase, le nuove funzionalità sono integrate con il codice precedente, grazie ai processi di CI che si rivelano fondamentali.

6. ***Deploy***. Il software viene finalmente distribuito (CD). Il processo viene eseguito in modo tale che eventuali modifiche apportate in qualsiasi momento al codice, non pregiudichino il funzionamento del sito ad alto traffico.

7. ***Operate and Monitor***. In queste fasi, il team operativo si prenderà cura del comportamento inappropriato del sistema o dei bug riscontrati in produzione.

1.4.3 DevOps vs Agile

Al contrario di quanto si possa pensare, Agile e DevOps non sono due metodologie contrapposte, anzi la seconda può essere considerata un vero e proprio miglioramento della prima, avvenuto nel corso del tempo.
La grande novità è quella di riuscire a far collaborare i due settori cruciali dello sviluppo di un'applicazione, ovvero i *Developers*, che si occupano della modifica delle funzionalità, e le *Operations*, che ne controllano la stabilità, e spingono verso un miglioramento generale dei servizi.

La tabella 1.2 riassume le principali differenze tra i due metodi.

Agile	**DevOps**
Enfatizza l'abbattimento delle barriere tra sviluppatori e management.	DevOps riguarda la distribuzione del software e i team operativi.
Risolve il divario tra le esigenze dei clienti e i team di sviluppo.	Affronta il divario tra lo sviluppo e il team operativo.
Si concentra maggiormente sulla prontezza funzionale e non-funzionale.	Si concentra sulla prontezza operativa e commerciale.
La metodologia agile riguarda principalmente il modo in cui lo sviluppo è concepito dall'azienda.	DevOps pone l'accento sulla distribuzione del software nei modi più affidabili e sicuri che non sono necessariamente sempre i più veloci.
Pone una grande enfasi sulla formazione di tutti i membri del team affinché abbiano una varietà di competenze simili. In questo modo, quando qualcosa va storto, qualsiasi membro del team può ottenere assistenza da qualsiasi altro, in assenza del leader.	DevOps utilizza il principio del *"divide and conquer"*, diffondendo le competenze tra il team di sviluppo e quello operativo, mantenendo una comunicazione costante.
Lo sviluppo agile si basa sugli "sprint". Ciò significa che i tempi di sviluppo sono molto brevi (meno di un mese), e le diverse funzionalità devono essere prodotte e rilasciate in quel periodo.	DevOps si impegna per scadenze e benchmark consolidati con le versioni principali, piuttosto che con quelle più piccole e più frequenti.

Tabella 1.2. Differenze tra il metodo Agile e quello DevOps

Si può pensare alla differenza tra i due metodi in questo modo: se l'Agile cerca di innovare il processo di sviluppo sfruttando i privilegi della tecnologia, il DevOps punta ad un obiettivo più ampio, ovvero seguire un insieme di principi che permettano di guidare il cambiamento verso un software più stabile e sicuro, che arrivi senza troppi intralci alla fase di produzione, risolvendo così un intero problema di business. (Salgarelli, 2017)
DevOps riesce quindi ad andare oltre gli ideali dell'Agile, con l'obiettivo di ottimizzare l'applicativo a livello globale, piuttosto che locale.

1.4.4 I principi fondamentali

Qui di seguito sei principi che sono essenziali per l'adozione di DevOps:

1. *Focus sul cliente*: il team DevOps deve intraprendere un'azione incentrata sul cliente per cui deve costantemente investire in prodotti e servizi.

2. *Responsabilità end-to-end*: il team DevOps deve fornire supporto alle prestazioni per il loro intero ciclo di vita. Ciò migliora il livello di responsabilità e la qualità dei prodotti progettati.

3. *Miglioramento continuo*: la cultura DevOps si concentra sul miglioramento continuo per ridurre al minimo gli sprechi. Accelera continuamente il miglioramento del prodotto o dei servizi offerti.

4. *Automatizzare tutto*: l'automazione è un principio vitale del processo DevOps, non solo per lo sviluppo del software, ma anche per l'intero panorama dell'infrastruttura.

5. *Lavorare come una squadra*: nella cultura DevOps i ruoli di designer, sviluppatore e tester sono già definiti. Tutto quello che devono fare è lavorare come una squadra con una collaborazione continua.

6. *Monitorare e testare tutto*: è molto importante per il team DevOps disporre di solide procedure di monitoraggio e testing.

1.5 Obiettivi

Lo scopo di questa tesi, realizzata in collaborazione con l'azienda *Alten Italia Spa*, è quello di approfondire le varie tecniche di testing in ambito front-end, partendo dallo sviluppo di un'applicazione Angular, risaltando le tecniche di routing e di 'lazy loading' per il caricamento delle pagine, messe a disposizione dal framework stesso. In particolare verranno analizzati due tools di testing: **Jest** per i test unitari, e **Cypress** per quelli di integrazione. L'obiettivo finale è quello di realizzare una CI/CD pipeline completa, che esegue in automatico sia la compilazione, sia il testing, sia il deployment finale su Firebase [1], il tutto tramite GitHub Actions.

Il resto di questa tesi sarà strutturato come segue:

- Il Capitolo 2 tratterà dello sviluppo dell'applicazione in Angular, spiegandone i dettagli in merito alla struttura e al funzionamento, e presenta anche un accenno all'analisi statica realizzata mediante SonarQube;
- Il Capitolo 3 è focalizzato sul testing unitario tramite il tool *Jest*, prestando anche particolare attenzione alla code coverage e alla sua pubblicazione sulla piattaforma *Codecov*;
- Il Capitolo 4 è incentrato sul testing end-to-end tramite il tool *Cypress*, anche affiancato a *Cucumber*;
- Il Capitolo 5 presenta le conclusioni e un breve approfondimento sul risparmio, sia in termini di tempo che di denaro, dovuto all'utilizzo di questo approccio.

[1] https://booking-portal-3edee.web.app/

Capitolo 2

Architettura e implementazione

L'applicazione realizzata è stata sviluppata in Angular 9, un framework open source per lo sviluppo di web app a singola pagina, che utilizza TypeScript come linguaggio di programmazione. Sono state implementate tutte le caratteristiche fondamentali dell'ambiente Angular:

- **Moduli**. Costituiscono i blocchi principali dell'applicazione. Il modulo comprende non solo i componenti dell'applicazione, ma anche i relativi servizi e pipes, rendendoli eventualmente pubblici tramite la proprietà di *export*, in modo tale che anche altri componenti possano utilizzarli.

- **Componenti**. Definiscono le varie classi dell'app. Ogni classe contiene i propri dati e la propria logica, ed è associata ad un template HTML che ne definisce la vista.

- **Forms**. Servono per gestire e memorizzare gli input esterni dati dall'utente. In questo caso saranno utilizzati per gestire la registrazione di un nuovo utente, per effettuare il login e per salvare una prenotazione o un messaggio.

- **Servizi**. Sono un particolare tipo di classe, singleton, che implementa funzionalità condivise tra i vari elementi di un'applicazione. La stessa istanza può quindi essere utilizzata da più componenti. In questo caso i servizi verranno utilizzati per recuperare i dati da un server, quindi sono tutti gestiti tramite chiamate HTTP asincrone verso il server, sfruttando il meccanismo degli Observable, che permette di effettuare alcune operazioni sui dati che arrivano.

- **Routing**. Permette di definire percorsi diversi all'interno dell'applicazione, associando gli URL alle viste piuttosto che alle pagine.

2.1 Contesto

La web app realizzata consiste in un piccolo portale di prenotazione per stanze d'hotel. Si divide in 3 sezioni:

1. **Login**: è richiesto l'inserimento di username e password se si è già registrati, altrimenti è possibile creare un nuovo profilo attraverso un form di registrazione.

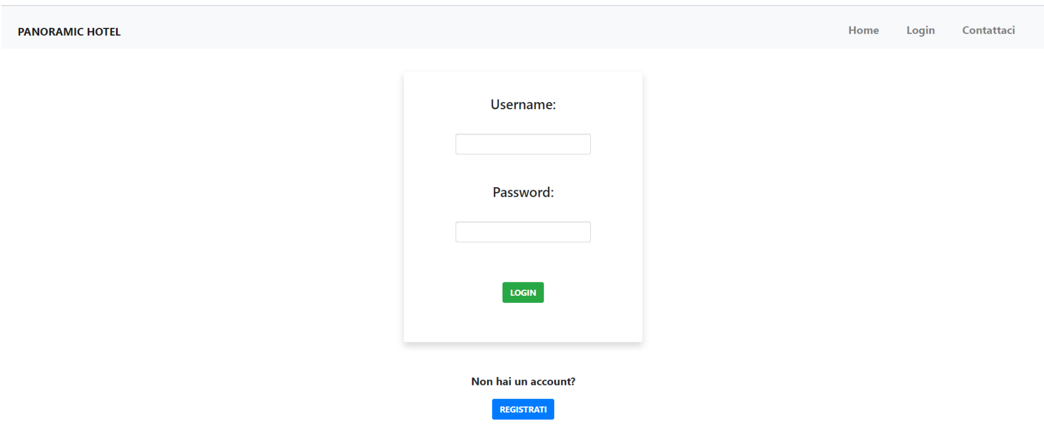


Figura 2.1. Pagina di login.

2. **Dashboard**: homepage del sito, divisa in ulteriori 3 sezioni:
   - *a sinistra*: card delle varie tipologie di stanze, con descrizione, prezzo e rating. Al click sulla stanza si apre una pagina che ne specifica i dettagli e un bottone porta ad un form di prenotazione.
   - *al centro*: carousel con foto varie dell'hotel che scorrono automaticamente, ognuna con la propria descrizione, e in basso descrizione generale dell'hotel.
   - *a destra*: sezione dedicata ad un preventivo in base al periodo del soggiorno e del numero di persone.

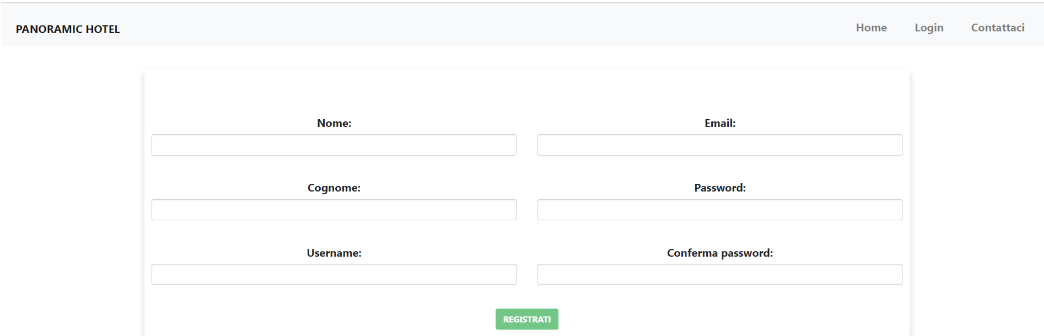


Figura 2.2. Pagina di registrazione.

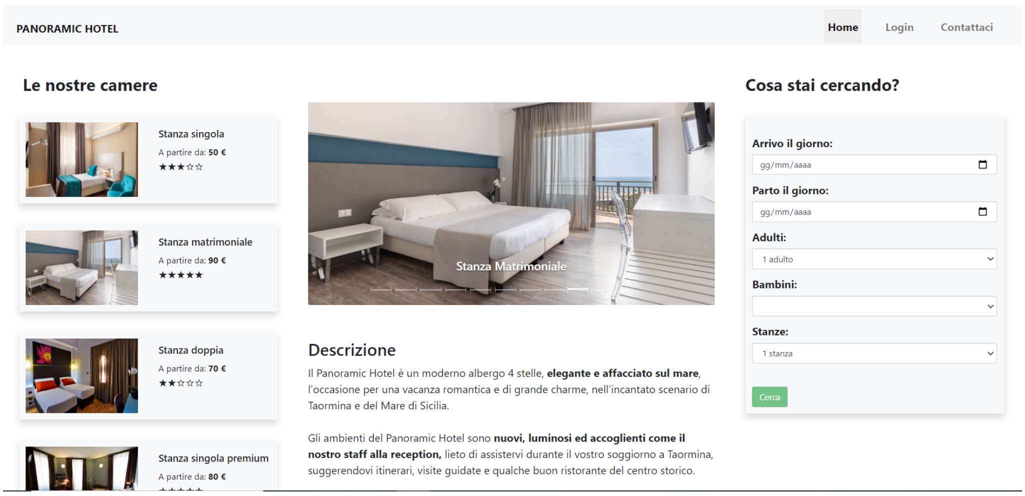


Figura 2.3. Pagina principale.

3. **Contattaci**: tramite questo form è possibile contattare l'hotel per eventuali dubbi o domande.

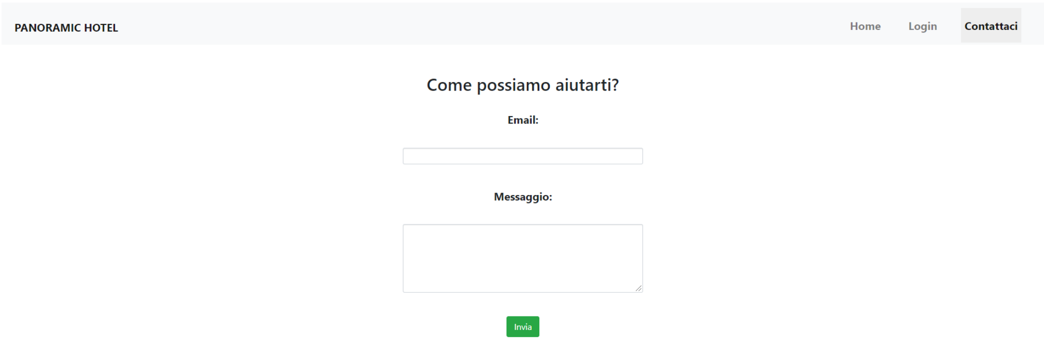


Figura 2.4. Pagina dei contatti.

2.1.1 Bootstrap

Bootstrap è uno dei framework CSS più famosi e più utilizzati per lo sviluppo di interfacce web. E' considerato ad oggi una sorta di standard de facto in ambiti come la creazione di template HTML preconfezionati, soprattutto in un ottica responsive (HTML.it, 2018). Nell'ottica di rendere l'applicazione fruibile e navigabile anche da dispositivi mobile, è stata aggiunta questa libreria all'interno del progetto, implementando la logica opportuna in tutte le pagine.

2.2 Struttura

La figura 2.5 descrive la struttura del progetto.

In particolare abbiamo:

- *app.module*: il modulo di default che viene generato al momento della creazione del progetto, detto anche "root module". Vengono importati tutti i componenti necessari all'avvio dell'applicazione.

- *app-routing.module*: quello che ci consente di navigare tra più viste, dichiarando a livello di applicazione la mappatura tra i percorsi e i rispettivi componenti.

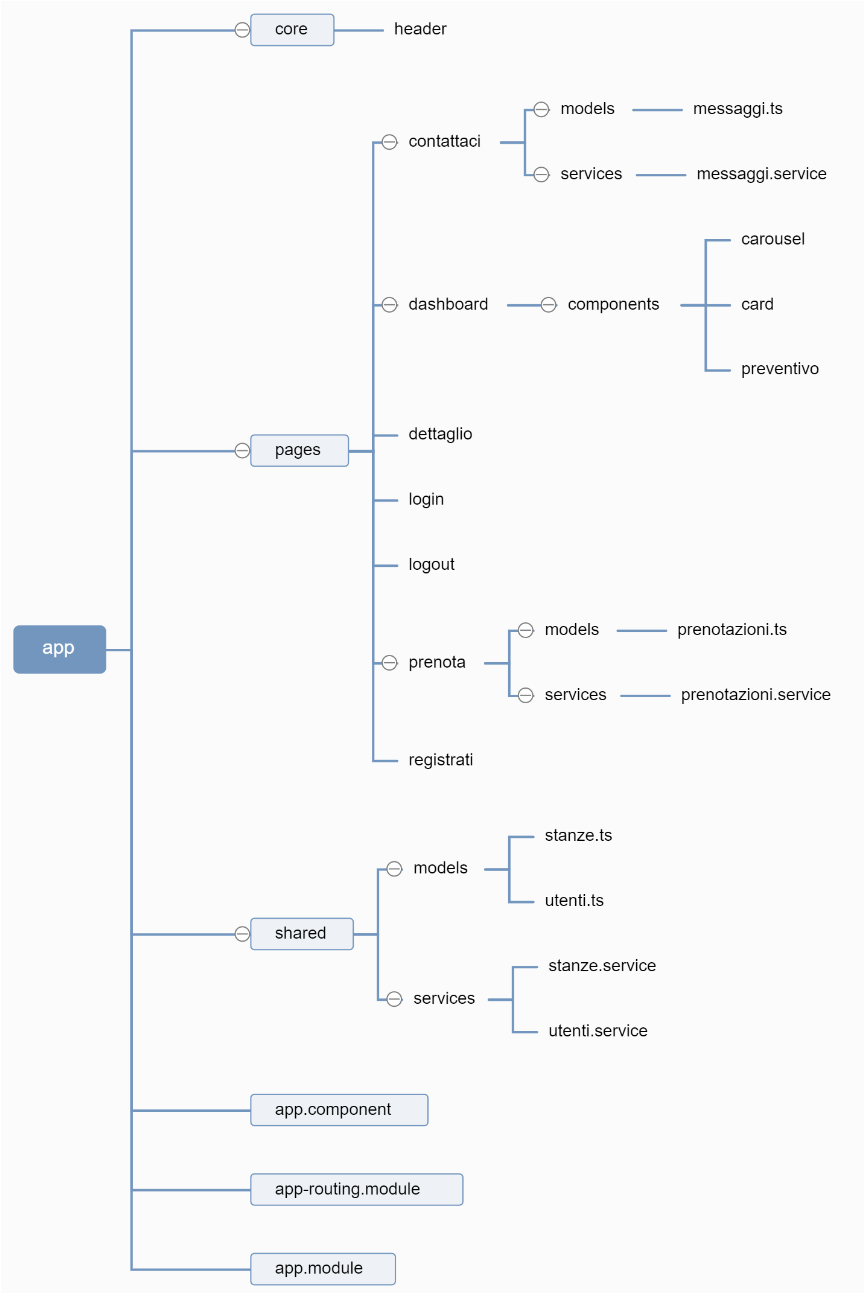


Figura 2.5. Rappresentazione modulare dell'applicazione.

- *app.component*: è il componente base dell'applicazione, la cui vista cambia a seconda del percorso in cui ci troviamo.

- *shared*: è una cartella che contiene i modelli e i servizi condivisi da più componenti. In questo caso ci sono quelli relativi alle stanze e agli utenti.

- *pages*: è una cartella che contiene tutti i componenti raggiungibili tramite routing. In particolare, *dashboard* è formato da ulteriori 3 sottocomponenti, che quindi fanno parte di una sua sottocartella.

- *core*: contiene l'unico componente che si ripete in ogni vista, ovvero la barra di navigazione.

2.2.1 Simulazione del database

A questo scopo è stato creato il file *db.json*, contenente tutti i dati necessari per interagire con l'applicazione. Il file è composto da 4 arrays, ciascuno corrispondente a un elenco degli oggetti di seguito elencati:

1. *stanze*: ogni stanza è caratterizzata da ID, tipologia, prezzo, rate e url.

2. *utenti*: ogni utente è caratterizzato da ID, nome, cognome, password e indirizzo mail.

3. *messaggi*: ogni messaggio è caratterizzato da ID, indirizzo mail, e testo.

4. *prenotazioni*: ogni prenotazione è caratterizzata da ID, nome, cognome e indirizzo mail di chi prenota, giorno di arrivo e di partenza, numero di adulti, di bambini e di stanze richieste, tipologia di stanza.

Il file è stato importato su Firebase, implementando cosi un vero e proprio *Realtime Database*, tramite il quale è possibile leggere, recuperare e aggiornare i dati in tempo reale. Per far comunicare l'applicazione con il database sono state quindi implementate delle chiamate HTTP asincrone verso il server, sfruttando il meccanismo degli *Observable*, che nell'ambito specifico di Angular possono essere visti come uno stream di dati su cui è possibile effettuare delle elaborazioni.

2.2.2 Servizi HTTP e Observables

Come detto in precedenza, per elementi a cui non si vuole associare una vista, si crea un servizio, ovvero una particolare classe che sfrutta il meccanismo di *Dependency Injection*, una tecnica che Angular ha implementato in maniera efficiente per semplificare la fase di sviluppo e rendere più flessibili i componenti, rendendo disponibili le dipendenze esterne di un oggetto tramite il costruttore, senza istanziare altri oggetti.

I servizi si occupano, quindi, di gestire il modello dei dati. Solitamente il recupero delle informazioni da un server è gestito tramite una web API, per questo motivo sfrutteremo i servizi HTTP basati sugli Observable, ovvero delle "collezioni di strumenti" che permettono di “osservare” dei dati, o meglio restare in attesa in modo passivo di molteplici valori. (Faghy, 2019)
I servizi, infatti, contengono metodi il cui valore restituito è un Observable. Quando questi vengono chiamati all’interno del componente, è necessario registrare l’azione da eseguire sullo stream di dati, tramite il metodo *subscribe()*, ed eventualmente anche gestire gli errori molto facilmente ricorrendo alla callback *error()*.
Sono state quindi implementate delle chiamate REST (GET, POST, PATCH) in tempo reale al Firebase Database, rendendo possibile la registrazione di un nuovo utente e il salvataggio di nuove prenotazioni e nuovi messaggi.

2.2.3 Change-Detection strategy

Angular esegue il rilevamento delle modifiche su tutti i componenti, dall’alto verso il basso, ogni volta che rileva un cambiamento nell’applicazione, dovuto per esempio ad un evento utente o a dati ricevuti da una richiesta di rete. Il rilevamento delle modifiche è molto performante, ma man mano che l’app diventa più complessa e la quantità di componenti aumenta, inizia a richiedere troppe risorse. Per aggirare il problema, è possibile passare alla strategia *onPush*, forzando l’ascolto solo di determinati componenti. In questo modo Angular rileverà solo le loro modifiche e quelle dei loro figli, unicamente quando vengono passati loro nuovi riferimenti e non più quando i dati vengono semplicemente mutati.

Tuttavia in questo modo, quando i dati mutati saranno di tipo Observable, i cambiamenti non verranno rilevati. E’ necessario, quindi, forzare, un ulteriore controllo chiamando il metodo *markForCheck()* della classe *ChangeDetectorRef* ogni qualvolta viene effettuata una *subscribe()*. In questo modo Angular saprà che dovrà restare in ascolto di un cambiamento anche per quel particolare oggetto. (DigitalOcean, 2017)

2.3 Testing statico

L'analisi statica consiste nel valutare un intero sistema, o un suo modulo, in base al suo contenuto, alla sua documentazione e alla sua struttura, quindi senza la necessità che esso sia in esecuzione.

Solitamente questo tipo di testing è il primo ad essere effettuato, in quanto rappresenta un controllo preliminare, sia dei requisiti applicativi che del codice stesso. Ha come scopo, infatti, quello di portare subito alla luce le anomalie del prodotto (Lavecchia, 2017), in modo tale da risolvere i bug minori il prima possibile, risparmiando cosi tempo, denaro e risorse.
Un esempio immediato di testing statico è la rilettura del codice da parte del programmatore stesso o, preferibilmente, di qualcun altro. Tramite questa operazione ci si può facilmente accorgere di loop infiniti, cicli inutili, blocchi di codice duplicati, codice mal strutturato e così via.

Questo tipo di analisi può essere anche effettuata in modo automatizzato, tramite alcune piattaforme disponibili sul mercato, come SonarQube, che utilizzeremo per analizzare l'applicazione in esame.

2.3.1 SonarQube

E' una piattaforma open-source, utilizzata per il controllo continuo della qualità del codice. Tramite l'analisi statica, è in grado di rilevare velocemente, bug, "code smells" e vulnerabilità su oltre 20 linguaggi di programmazione.

Una volta installato, è possibile lanciare il server locale tramite il file *StartSonar.bat*, ed in seguito accedere alla pagina principale, che mostra tutti i progetti analizzati. Per implementarlo all'interno del nostro progetto in Angular è sufficiente installare il package *SonarScanner* e, tramite il file *sonar-project.properties* impostare una serie di proprietà di configurazione. A questo punto è possibile lanciare l'esecuzione dell'analisi statica di tutto il codice tramite la CLI. I risultati saranno accessibili direttamente sul server Sonar, nella sezione dedicata ai progetti, come mostrato in figura 2.6.

I problemi rilevati possono essere di diverso tipo: *bug*, *vulnerability*, *smell code*, *coverage* o *duplication*. Ogni categoria ha un corrispondente numero di problemi o un valore percentuale.

Inoltre, i problemi possono avere uno dei cinque diversi livelli di gravità: *blocker*, *critical*, *major*, *minor* e *info*. Proprio davanti al nome del progetto c'è un'icona che

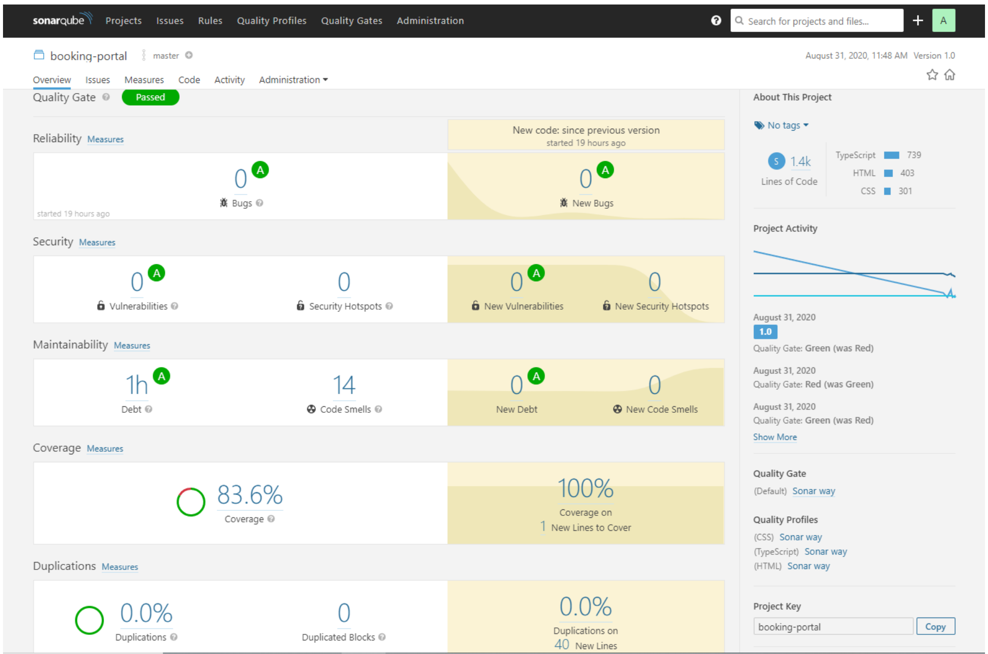


Figura 2.6. Schermata di SonarQube dopo l'analisi statica.

mostra lo stato del **Quality Gate**: superato (verde) o non riuscito (rosso).
Il Quality Gate è un insieme di condizioni che il progetto deve soddisfare prima di poter essere considerato idoneo per il rilascio.

Garantire la qualità del "nuovo" codice mentre si risolvono i problemi già esistenti nella versione precedente è un buon modo per mantenere una buona base di codice nel tempo. Il Quality Gate facilita l'impostazione di regole per la convalida di ogni modifica durante l'analisi successiva ed influenza anche i segmenti di codice rimasti uguali, in modo tale da favorire la risoluzione della maggior parte dei problemi nel corso del tempo.
E' per questo che SonarQube riporta sempre il confronto con l'analisi effettuata in precedenza, identificando il cosiddetto *Leakage Period*, ovvero il lasso di tempo che intercorre tra due analisi o versioni del progetto.

La configurazione predefinita per SonarQube contrassegna il codice come "fallito" se:

- la coverage del codice aggiunto è inferiore all'80%;
- la percentuale di righe duplicate nel codice aggiunto è maggiore di 3;

- la manutenibilità, l'affidabilità o la valutazione della sicurezza è peggiore del grado A.

E' anche possibile inserire l'analisi all'interno della pipeline, in modo tale da condizionare il suo completamento al soddisfacimento del Quality Gate, e per fare questo utilizzeremo SonarCloud, ovvero la versione cloud-hosted di SonarQube.

2.4 Testing dinamico

L'analisi dinamica è il processo di valutazione di un intero sistema, o di un suo componente, basato sull'osservazione del suo comportamento in esecuzione. Un prerequisito fondamentale per svolgere questo tipo di testing è conoscere il risultato atteso, in modo tale da poter effettuare un confronto con il comportamento osservato. Per essere in grado di fare ciò, è necessario quello che viene definito *oracolo*, ovvero un oggetto che conosce nel dettaglio ogni caso di test, ed è quindi in grado di prevederne il comportamento. L'oracolo può essere di due tipi: umano o automatico. Il primo si basa sulla conoscenza dei requisiti o su congetture personali, mentre il secondo prende forma grazie alla definizione di specifiche formali o sulla base dei risultati di uno stesso software, che ha origini differenti.

L'analisi dinamica si divide principalmente in due categorie, che differiscono per l'approccio che adottano. La prima è quella del *black-box testing*, o testing funzionale. Si basa sulla sola conoscenza dei requisiti di sistema, e si preoccupa di analizzare gli output generati dal sistema, o dal componente che si sta testando, in risposta ad input specifici. La seconda è definita *white-box testing*, o testing strutturale, in quanto richiede una profonda conoscenza della struttura del software, e quindi dell'intero codice. Sulla base di queste informazioni vengono poi definiti i casi di test, gli input associati e i risultati attesi. (Lavecchia, 2017)

Nel prossimo capitolo analizzeremo nel dettaglio il testing dinamico, in particolare approfondiremo quello basato sui test unitari, realizzati mediante l'utilizzo del tool Jest.

2.5 Costruzione della CI/CD pipeline

A questo scopo si è deciso di sfruttare le *Actions* messe a disposizione da GitHub, in modo tale da automatizzare il *workflow* di operazioni direttamente nel repository [1]. Per farlo è necessario creare un file YAML, in cui si indica prima l'evento scatenante le operazioni, e successivamente l'elenco dei vari *jobs* che devono essere eseguiti, ognuno dei quali può essere composto da uno o più *steps*. Tutti i jobs vengono eseguiti in parallelo, a meno che non si specifichino delle dipendenze. Nel nostro caso l'evento trigger sarà l'evento *push*, e i jobs saranno 6:

1. **build**. Come prima cosa estrae il codice sorgente dal repository, tramite la action *checkout*. Successivamente si memorizzano in cache i moduli del nodo (questa configurazione serve per tenere traccia dei cambiamenti dei package *npm* in un file json, in modo tale da velocizzare il processo di compilazione). Infine impostiamo l'ambiente Node da usare nelle actions ed installiamo i pacchetti richiesti, per poi creare il package build.

2. **sonarcloud**. In questa fase viene sfruttato SonarCloud per sostituire il server locale di SonarQube. Per fare ciò è sufficiente creare un account sulla piattaforma [2]. Sarà cosi possibile consultare il report aggiornato a seguito di ogni nuova *push*.

3. **jest**. Anche qui viene estratto il codice sorgente e poi viene semplicemente lanciato il comando che esegue i test unitari utilizzando il framework JEST.

4. **coverage**. Questo job è conseguente al precedente, ovvero viene eseguito solo dopo che l'altro è finito. Attraverso una action predefinita, *codecov-action*, viene pubblicato il report della code coverage sulla piattaforma Codecov.

5. **cypress**. Esegue i test end-to-end su un server locale, il cui indirizzo viene specificato tramite il parametro *wait-on*. In questo modo viene separato l'ambiente di sviluppo da quello di testing.

6. **deploy**. L'obiettivo della pipeline è eseguire il deploy solo se l'esecuzione dei test è andata a buon fine, motivo per il quale quest'ultimo job verrà eseguito

[1] https://github.com/LisaRomita/booking-portal

[2] https://sonarcloud.io/dashboard?id=LisaRomita_booking-portal

solo se *jest*, *cypress* e *sonarcloud* avranno avuto successo. Sarà necessario utilizzare un token per l'autenticazione, in modo tale da consentire a GitHub l'accesso alle risorse di Firebase.

La figura 2.7 mostra una schermata parziale di quello che appare su GitHub, nella sezione *Actions*, a seguito di un commit avvenuto con successo.

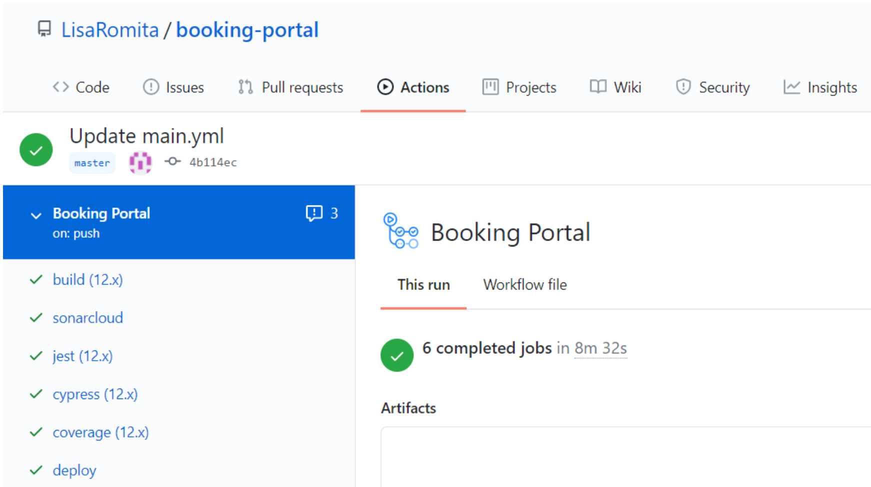


Figura 2.7. Workflow completato con successo.

Capitolo 3

Test unitari con Jest

Lo Unit Testing o, letteralmente, testing di unità, ha come scopo quello di verificare il corretto funzionamento dei singoli moduli di cui il sistema è composto, quindi isolati da tutto il resto. E' un'attività che è stata riconosciuta come modello standardizzato da IEEE (1008-1987), e può essere vista come un insieme delle seguenti fasi:

1. Pianificare il metodo da utilizzare, quindi individuare quali saranno le risorse richieste e le tempistiche previste.
2. Individuare le funzionalità da testare.
3. Determinare, quindi realizzare, l'insieme degli script di test.
4. Eseguire i test.
5. Verificare se è necessario scrivere ulteriori test.
6. Valutare i risultati finali.

E' quindi importante prestare una particolare attenzione ai servizi che dipendono da componenti esterni, in quanto stiamo andando a testare la singola unità, andando ad isolarla dall'intero sistema. (Lavecchia, 2019)

3.1 Test Doubles

Ci sono diversi modi in cui è possibile simulare un elemento da cui dipende un componente, o isolare un singolo modulo:

- *Dummy*: è un oggetto che non viene effettivamente utilizzato, la cui presenza è però necessaria affinché il metodo venga eseguito, e se ne possa così verificare la correttezza.

- *Stub*: fornisce risposte cablate per causare particolari comportamenti del sistema sotto verifica, per i soli casi previsti dalle procedure di test. L'esito è determinato solo al termine dell'interazione tra il sistema e il test double.

- *Spy*: è un particolare tipo di stub che, in più, registra le chiamate effettuate dall'oggetto sotto test, per permettere di verificare a posteriori la corretta comunicazione con il componente originale.

- *Fake*: è un oggetto che sostituisce il componente originale per ragioni che esulano dalla verifica del buon funzionamento del sistema.

- *Mock*: è uno stub che verifica il funzionamento del sistema, essendo in grado di stabilire l'esito della verifica durante lo svolgimento dell'interazione stessa.

Questo significa che la realizzazione di una buona suite di test non implica semplicemente la scrittura di test cases validi, ma anche di classi *mock* che andranno a sostituire le dipendenze durante l'esecuzione dei test.

3.1.1 Ts-mockito

E' una libreria che permette di creare mock per qualsiasi tipo di codice TypeScript. E' inoltre molto semplice e intuitiva da utilizzare, essendo ispirata alla libreria *Mockito* di Java. Nonostante sia meno popolare di altre già presenti sul mercato, come *Typemoq*, che presenta sicuramente funzionalità anche più avanzate, è caratterizzata da una sintassi molto più compatta, ed è per questo che sta acquisendo notevole interesse.

La figura 3.1 rappresenta l'andamento dei download del package ts-mockito, rispetto a quello di typemoq, negli ultimi 6 mesi, secondo i dati raccolti da *npm trends*.

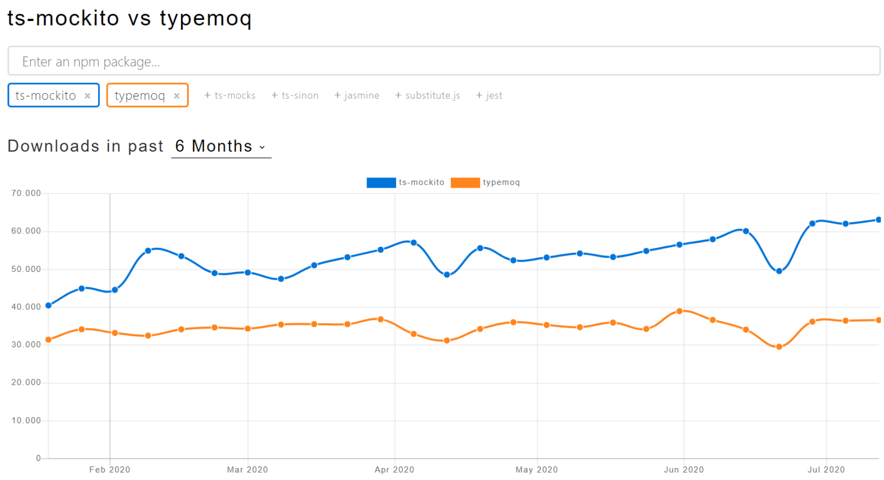


Figura 3.1. ts-mockito vs typemoq, npm trends.

Nell'ambito di questo lavoro di tesi, ts-mockito verrà utilizzato all'interno dei test unitari dei componenti, per simulare servizi e interfacce. Il semplificare la scrittura dei mock, permette di concentrarsi sulla realizzazione di test cases solidi e funzionali, in modo tale da cercare di rendere il codice più corretto possibile.

3.2 Jest

Jest è un *JavaScript test runner*, ovvero una libreria che permette di scrivere, eseguire e strutturare i test in modo molto semplice. Riassumiamo alcune delle sue caratteristiche principali:

- **Configurazione**. Molto semplice da configurare, basta installare il pacchetto in qualsiasi tipo progetto, e tutte le funzionalità di Jest saranno subito disponibili.
- **Sicuro**. Si accerta che tutti i test girino in un proprio processo, riuscendo così ad eseguirne di più in parallelo.
- **Veloce**. Per accelerare la fase di testing, Jest esegue prima quelli che hanno fallito in precedenza, e riorganizza gli altri in base al tempo di cui necessitano.
- **Code Coverage**. Genera in automatico un report sulla code coverage, se il comando viene eseguito con il flag *–coverage*, senza bisogno di ulteriori installazioni. Approfondiremo questo aspetto più avanti.

- **Debug**. E' molto semplice capire il motivo per cui un test è fallito, in quanto Jest mette a disposizione un log di errore ricco ed esaustivo, che rende il lavoro di debug rapido ed efficace.

Noi utilizzeremo Jest per testare ogni singolo componente, insieme a ts-mockito. Mentre per quanto riguarda il testing dei servizi, e quindi delle chiamate HTTP, ci serviremo di un modulo fornito direttamente da Angular, *HttpClientTestingModule*.

3.2.1 Jest vs Karma

Anche Karma è un test runner, creato dal team di AngularJS, infatti è quello usato di default per testare progetti scritti in Angular. Per eseguire gli script di test, Karma apre il proprio browser, e fa girare tutti test che sono specificati nel file chiamato *karma.conf.ts*, il che è un vantaggio, ma dall'altro lato non è possibile ottenere un report sulla code coverage, a meno che non si installino dei plugin aggiuntivi.

Jest è invece basato su Jasmine, un altro framework che permette, tramite alcune funzioni come *equal* e *toBe*, di verificare se un determinato comportamento sta funzionando come previsto, e che supporta anche il mocking, rendendo possibile l'isolamento dei test. (Rhettiarachchi, 2020) suggerisce i seguenti motivi per cui usare Jest al posto di Karma:

- **Jest è da 2 a 3 volte più veloce**. Il motivo è che Karma utilizza un vero browser per eseguire i test, mentre Jest utilizza la riga di comando, per cui il tempo necessario viene dimezzato. Ciò è particolarmente importante quando si utilizza una pipeline CI/CD. Essendo i test più veloci, anche i tempi di esecuzione verranno ridotti.

- **Istantanee**. Quando viene eseguito un test per la prima volta, tramite il metodo *toMatchSnapshot()* è possibile creare un'istantanea del componente e memorizzarla in una cartella. Ogni volta che testiamo l'applicazione, Jest genera istantanee per tutti i componenti e le abbina a quelle precedenti. Questa è un'ottima funzionalità perché garantisce che l'interfaccia utente non cambi in modo imprevisto.

- **Procedura di installazione**. Una delle filosofie portate avanti dai creatori di Jest, come già detto, è quella della *zero-configuration experience*, che si dimostra essere vera, in quanto Jest funziona benissimo con la maggior parte delle applicazioni JavaScript, almeno nelle sue funzionalità di base. A differenza di Karma, che invece necessita di molte più configurazioni iniziali.

Tuttavia evidenzia anche alcuni svantaggi:

- Poichè Jest non gira su un vero browser, ma usa jsdom, c'è il rischio che questo differisca da quello che andremo poi ad utilizzare.
- Il debugging, a livello visivo, è più difficile rispetto a Karma.

Jest supporta quindi tutte le funzionalità di Jasmine, e integra anche quelle di Karma, motivo per il quale andremo a disinstallare quest'ultimo, insieme a tutte le sue dipendenze, cosi da avere un codice più pulito.

3.2.2 Il file *jest.config.ts*

Nonostante Jest sia in grado di supportare le funzionalità di base senza necessità di alcuna configurazione, in questo caso abbiamo bisogno di implementare delle caratteristiche aggiuntive, per cui serve un file di configurazione specifico. In particolare, faremo in modo che Jest:

- cerchi i file *.spec.ts* nella cartella *src* del nostro progetto,
- legga il file di configurazione TypeScript *tsconfig.json* per tutti gli alias TypeScript cosi da renderli comprensibili,
- compili il codice TypeScript in memoria prima di eseguire i test,
- raccolga informazioni sulla code coverage e le scriva in una cartella specifica, utilizzando più formati diversi.

Andando poi a modificare lo script *"test"* all'interno del file *package.json* sarà possibile eseguire i test tramite Jest direttamente con il comando *npm test*, e verrà anche automaticamente generato un file HTML con il report sulla code coverage.

3.3 I test

L'attività di tesi ha prodotto 16 suite di test, per un totale di 38 test. Per ogni componente sono state testate le funzioni principali, come quelle di manipolazione dei dati e di sottomissione dei forms. Per quanto riguarda invece i servizi, abbiamo utilizzato il modulo di testing di Angular, *HttpTestingClientModule*, che facilita, appunto, il testing delle richieste HTTP, che sono le funzionalità principali dei servizi dell'applicazione in questione.

3.3.1 Componenti

Ogni qualvolta sia presente una dipendenza all'interno del componente, si ricorre a ts-mockito per simulare soprattutto servizi e interfacce, in particolare ai metodi *when()* e *thenReturn()*, utilizzati per forzare un determinato comportamento. Ogni suite di test inizia con la chiamata alla funzione *describe()* che prende in ingresso due parametri: una stringa, che è il nome della suite di test, e una funzione, che racchiude tutti i test specifici di quella suite. Ogni test è quindi, a sua volta, caratterizzato dalla chiamata alla funzione *test()*, che riceve in ingresso la stessa tipologia di parametri della funzione *describe()*, con la differenza che la funzione passata questa volta, conterrà il vero e proprio test, quindi prima le assegnazioni, poi le azioni, e infine le asserzioni. All'interno della suite troviamo anche il blocco *beforeEach()*, che è una funzione che viene chiamata prima di ogni blocco *test()*. Ciò è possibile grazie all'API TestBed di Angular, che contiene una serie di metodi statici che effettuano una sorta di reset delle variabili. Viene quindi ripristinato l'ambiente di testing iniziale con il metodo *configureTestingModule()*, viene ricompilato il modulo attraverso la funzione *compileComponents()*, e infine viene creata una nuova istanza del componente con il metodo *createComponent()*.

La tabella 3.1 elenca nel dettaglio tutti i test realizzati per i componenti.

Componente	**# Test**	**Descrizione test**
Header	1	Toggle Login/Logout
Contattaci	3	Autocompilazione del campo mail
		Testo corretto
		Submit corretto
Dashboard	1	Testo corretto
Card	1	Creazione del componente
Carousel	1	Creazione del componente
Preventivo	5	Creazione del componente
		Calcolo della prima tipologia di preventivo
		Calcolo della seconda tipologia di preventivo
		Calcolo della terza tipologia di preventivo
		Calcolo della quarta tipologia di preventivo
Dettaglio	1	Creazione del componente
Login	3	Creazione del componente
		Controllo dell'utente
		Login corretto
Logout	1	Logout corretto
Prenota	2	Autocompilazione dei campi utente
		Submit corretto
Registrati	4	Corretto riempimento del vettore utenti
		Registrazione corretta
		Utente già esistente
		Password non corrispondenti
App	2	Creazione del componente
		Titolo corretto

Tabella 3.1. Descrizione nel dettaglio dei test dei componenti

3.3.2 Servizi

Nei servizi ritroviamo la stessa struttura, con la differenza che per testare le chiamate HTTP sono necessari ulteriori accorgimenti. Vengono inizialmente creati dei mock, ovvero i risultati che ci aspettiamo. Vanno importati i moduli *HttpTestingClientModule* e *HttpTestingController*, che facilitano il mock delle chiamate REST. Attraverso la utility *inject()*, inietteremo il servizio che ci interessa nel nostro test. Viene poi chiamato il metodo *subscribe()*, che ci aspettiamo ritorni un risultato uguale a quello di cui abbiamo precedentemente creato un mock. Successivamente ci assicuriamo che sia stata effettuata una sola richiesta al servizio presente all'url indicato, chiamando il metodo *expectOne()*. Poi verifichiamo che la richiesta non sia stata cancellata e che la risposta sia di tipo json, e completiamo la richiesta con la chiamata al metodo *flush()*. Infine ci accertiamo che non ci siano richieste in sospeso, attraverso la funzione *verify()*.

La tabella 3.2 elenca nel dettaglio tutti i test realizzati per i servizi.

Servizio	**# Test**	**Descrizione test**
Utenti	6	Creazione del servizio
		GET del singolo utente
		GET di tutti gli utenti
		POST di un utente
		Getter funzionante
		Setter funzionante
Stanze	3	Creazione del servizio
		GET di una stanza
		GET di tutte le stanze
Messaggi	2	Creazione del servizio
		POST di un messaggio
Prenotazioni	2	Creazione del servizio
		POST di una prenotazione

Tabella 3.2. Descrizione nel dettaglio dei test dei servizi

La figura 3.2 mostra quello che appare sulla riga di comando quando i test sono in esecuzione.

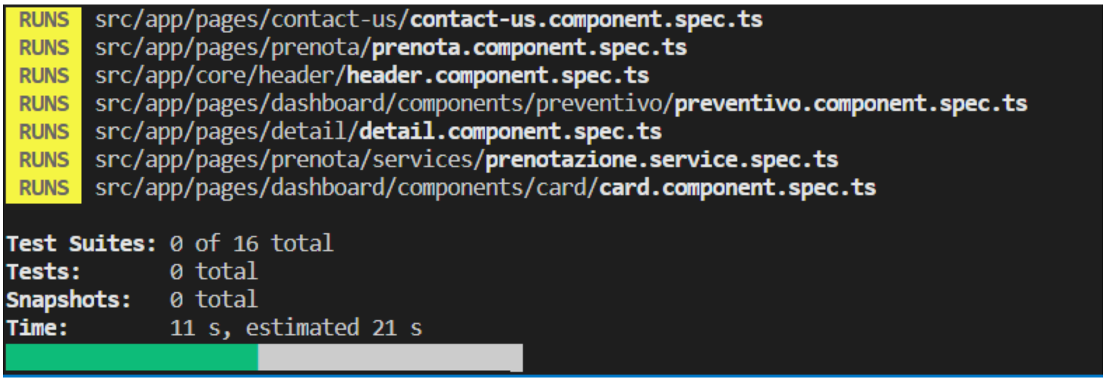


Figura 3.2. Test in esecuzione sulla CLI.

La figura 3.3 mostra invece la schermata che appare alla fine, dopo che tutti i test hanno avuto successo.

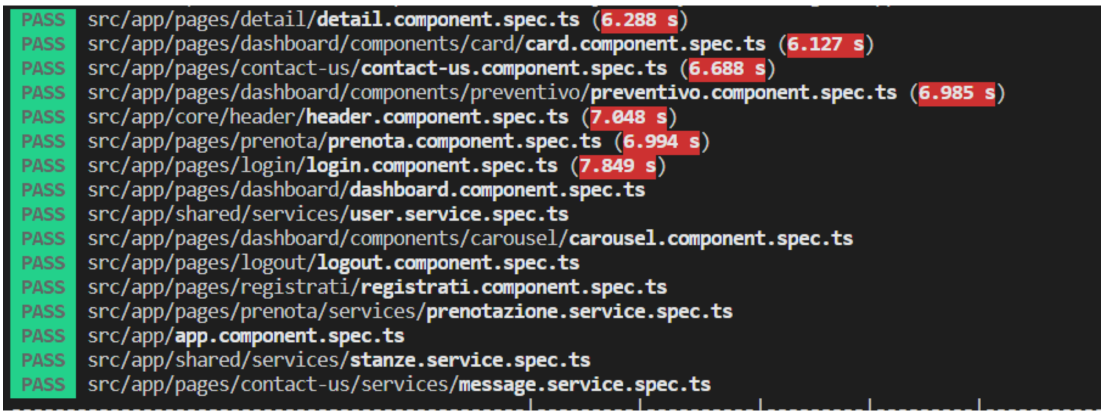


Figura 3.3. Test eseguiti.

3.4 White-box Testing

I test di tipo *white-box*, ovvero "a scatola bianca" devono il loro nome al fatto di essere basati sulla struttura interna o sull'implementazione del sistema, che può includere codice, architettura, flussi di lavoro o flussi di dati all'interno del sistema.

Un primo vantaggio del collaudo a scatola bianca è che permette di sondare dettagliatamente tutte le casistiche che possono presentarsi, e lo fa in modo facile ed efficiente grazie all'automazione.
Un altro è la semplificazione del debugging, cioè il passaggio dal malfunzionamento alla correzione del difetto, in quanto la segnalazione del bug normalmente indica il punto del codice e i valori delle variabili per cui questo si è manifestato.
Le tecniche di testing strutturale hanno in genere l'obiettivo di coprire quanto più codice possibile, andando ad analizzare tutte le istruzioni, procedure e porzioni di codice che compongono la struttura del programma. Generalmente l'obiettivo è quello di raggiungere un'ampia *coverage*, quindi di coprire più codice possibile. Per fare ciò è necessario definire un insieme di test, e di conseguenza una serie di dati di input, per fare in modo che tutte le componenti del sistema, e quindi tutte le sue funzionalità, siano coperte almeno una volta dall'esecuzione dei test.

3.4.1 Code coverage

Come accennato nel paragrafo precedente, è una metrica che permette di verificare la percentuale del codice che è stata testata, ovvero il numero di righe di codice che viene eseguito mentre i test girano. Utilizzando degli strumenti opportuni, come in questo caso Jest, è possibile anche sapere con esattezza quali sono le parti che non sono ancora state testate, cosi da integrare la nostra suite di test con quelli mancanti. Nello specifico ci sono 4 parametri che la code coverage prende in considerazione:

- *Statements*: indica la percentuale di statements eseguiti.
- *Branches*: indica la percentuale di path intrapresi.
- *Functions*: indica la percentuale di funzioni testate.
- *Lines*: indica la percentuale di righe di codice eseguite tramite i test.

Il 100% di *Statement Coverage* assicura che tutte le dichiarazioni eseguibili nel codice siano state testate almeno una volta, ma non garantisce che sia stata testata tutta la logica decisionale. Quando si raggiunge invece il 100% di *Decision Coverage*, si eseguono tutti i risultati delle decisioni (branches), che includono sia il caso dell'esito positivo (true) che di quello negativo (false), anche quando non c'è un'esplicita dichiarazione falsa (nel caso di un *if* senza il corrispondente *else*). Questo tipo di coverage aiuta a trovare difetti nel codice che altri test non sono in grado di garantire. Il raggiungimento del 100% di questo tipo di coverage garantisce anche la copertura totale della Statement coverage (ma non viceversa).

La figura 3.4 mostra il il report in formato html della code coverage, risultante dall'esecuzione dei test tramite Jest.

All files

92.96% Statements 264/284 **63.08%** Branches 41/65 **86.15%** Functions 56/65 **91.38%** Lines 212/232

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
src/app		100%	6/6	100%	0/0	100%	1/1	100%	4/4
src/app/core/header		100%	13/13	100%	2/2	100%	2/2	100%	10/10
src/app/pages/contact-us		100%	22/22	100%	4/4	100%	4/4	100%	17/17
src/app/pages/contact-us/services		100%	8/8	100%	0/0	100%	2/2	100%	6/6
src/app/pages/dashboard/components/card		100%	10/10	100%	0/0	100%	2/2	100%	8/8
src/app/pages/dashboard/components/preventivo		100%	23/23	100%	15/15	100%	3/3	100%	18/18
src/app/pages/detail		100%	17/17	100%	2/2	100%	4/4	100%	15/15
src/app/pages/logout		100%	8/8	100%	0/0	100%	2/2	100%	6/6
src/app/pages/prenota		100%	27/27	100%	4/4	100%	4/4	100%	22/22
src/app/pages/prenota/services		100%	8/8	100%	0/0	100%	2/2	100%	6/6
src/app/shared/services		96.97%	32/33	0%	0/1	90.91%	10/11	96.3%	26/27
src/app/pages/dashboard		94.74%	18/19	100%	4/4	80%	4/5	93.33%	14/15
src/app/pages/registrati		92.86%	26/28	70%	7/10	100%	7/7	91.67%	22/24
src/app/pages/login		90%	27/30	50%	3/6	100%	6/6	87.5%	21/24
src/app/pages/dashboard/components/carousel		61.9%	13/21	0%	0/13	50%	2/4	57.89%	11/19
src		54.55%	6/11	0%	0/4	16.67%	1/6	54.55%	6/11

Figura 3.4. Jest Coverage Report.

In ogni caso, la code coverage è una buona misura della quantità di codice testato, ma non è affatto indicativa per quanto riguarda la qualità dei test, per quello sono richiesti altri strumenti che non verranno approfonditi in questo contesto.
Per rendere disponibile il report online, è necessario ricorrere ad un ulteriore piattaforma, Codecov, che riesce ad accedere al repository originale e, così facendo, ad aggiungere la pubblicazione del report [1] alla CI/CD pipeline.

3.4.2 La piattaforma Codecov

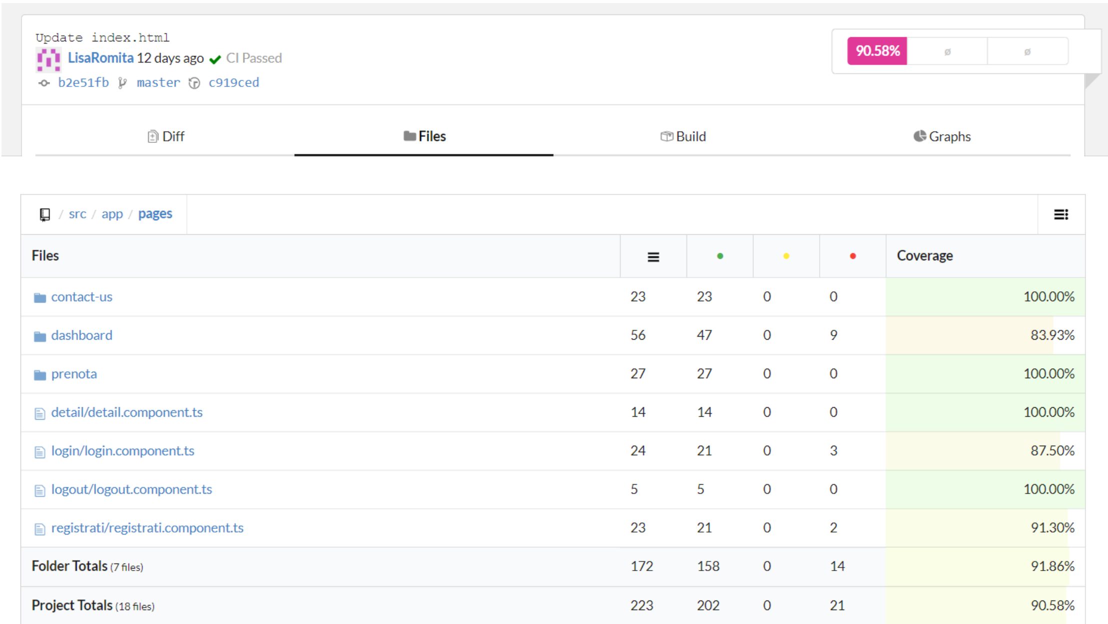
Codecov utilizza 3 termini per descrivere una linea di codice che è stata eseguita:

- **hit**: indica che il codice è stato eseguito dalla suite di test
- **partial**: indica che il codice è stato parzialmente eseguito, ma ci sono dei branch che non sono stati testati

[1]https://codecov.io/gh/LisaRomita/booking-portal

- **miss**: indica che il codice non è stato eseguito dalla suite di test

Il rapporto su cui si basa Codecov è $hits/(hits + partial + miss)$. Questo tipo di strumenti incoraggiano gli sviluppatori a scrivere test e ad aumentare la code coverage. Durante il processo di scrittura dei test, lo sviluppatore potrebbe scoprire nuovi bug o problemi di sintassi nel codice sorgente che sono importanti da risolvere prima di rilasciare l'applicazione. Codecov, a differenza dei prodotti open source e a pagamento, si concentra sull'integrazione e sulla promozione di pull request corrette. Integra le metriche direttamente nel flusso di lavoro per promuovere una maggiore copertura del codice, in particolare nelle richieste pull in cui si verificano comunemente nuove funzionalità e correzioni di bug.



Files	≡	●	●	●	Coverage
contact-us	23	23	0	0	100.00%
dashboard	56	47	0	9	83.93%
prenota	27	27	0	0	100.00%
detail/detail.component.ts	14	14	0	0	100.00%
login/login.component.ts	24	21	0	3	87.50%
logout/logout.component.ts	5	5	0	0	100.00%
registrati/registrati.component.ts	23	21	0	2	91.30%
Folder Totals (7 files)	172	158	0	14	91.86%
Project Totals (18 files)	223	202	0	21	90.58%

Figura 3.5. Schermata principale della piattaforma Codecov.

La figura 3.6 mostra il grafico realizzato da Codecov per rappresentare visivamente la percentuale di test coverage raggiunta. Il cerchio più interno è l'intero progetto, allontanandosi dal centro sono le cartelle e, infine, i singoli file. Le dimensioni e il colore di ogni sezione rappresentano rispettivamente il numero di istruzioni eseguite e la coverage. Vi sono delle parti in rosso, che non è stato possibile testare e che costituiscono il motivo per il quale non è stato raggiunto il 100%.

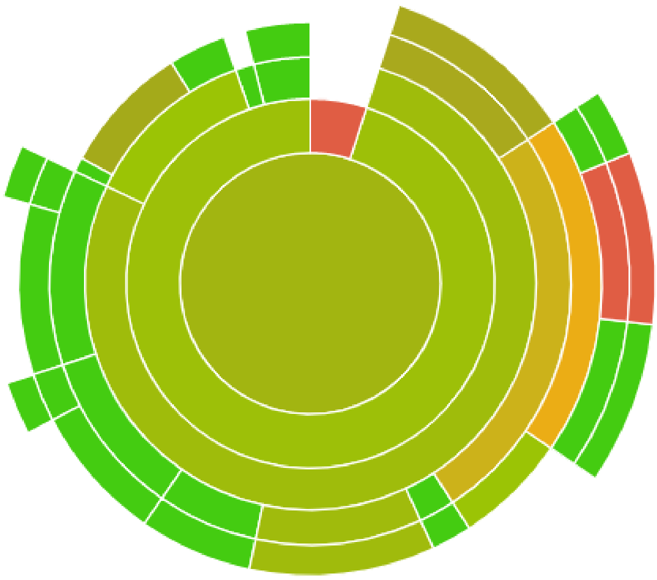


Figura 3.6. Grafico Sunburst.

Capitolo 4

Test end-to-end con Cypress

Con il testing end-to-end si intende quell'attività di testing dell'interfaccia grafica, cosi come la vedono gli utenti finali. In altre parole rappresenta una metodologia utilizzata per verificare se il flusso di un'applicazione si sta comportando come previsto dall'inizio alla fine, senza che vengano rilevati dei bugs che andrebbero ad inficiare sulla qualità dell'applicazione stessa.

L'end-to-end testing fa parte del System testing (test di sistema) e verifica i flussi e il funzionamento corretto del programma intero. Questa fase comprende il testing di interfacce e dipendenze esterne, come l'ambiente in cui il software viene eseguito o le componenti di back-end.

Lo scopo dell'esecuzione di test end-to-end consiste nell'identificare le dipendenze del sistema e garantire che vengano trasmesse le informazioni corrette tra i vari componenti. (Lavecchia, 2018)

4.1 Cypress

Cypress è un framework di testing end-to-end basato su Mocha, un altro framework JavaScript ricco di funzionalità in esecuzione nel browser.
Con Cypress è possibile inizializzare i test, scriverli, eseguirli e farne il debug. E' composto da un *Test Runner* open source, che viene installato localmente, e da un *Dashboard Service* che permette di registrare i test.
L'installazione è semplice e veloce, e include anche un set di esempi che facilitano molto la comprensione del linguaggio. Non è necessario configurare servers, drivers o altre dipendenze. Le caratteristiche principali di Cypress sono:

- *Time Travel*: durante l'esecuzione dei test, Cypress cattura degli snapshot e registra dei video, rendendo così il debug molto semplice da effettuare.
- *Automatic Waiting*: i test vengono eseguiti solo dopo che gli elementi diventano visibili nel DOM, evitando inutili attese.
- *Network-free*: Cypress gira direttamente nel browser, senza coinvolgere la rete, implementando i propri meccanismi di manipolazione del DOM.
- *Real-time reloads*: dopo la modifica di un test, Cypress lo rilancia in automatico.

La maggior parte dei tool di testing (come Selenium, di cui parleremo a breve) esegue i test al di fuori del browser, lanciando comandi da remoto attraverso la rete. Cypress fa esattamente l'opposto, in quanto viene eseguito nello stesso ciclo di esecuzione dell'applicazione.

Dietro Cypress, lato server, gira un processo Node. Cypress e Node comunicano, sincronizzano ed eseguono costantemente attività per conto proprio. Avendo accesso ad entrambe le parti (front-end e back-end), è possibile rispondere agli eventi dell'applicazione in tempo reale, lavorando allo stesso tempo al di fuori del browser, per attività di più alto livello.

Cypress opera anche a livello di rete leggendo e alterando il traffico web al volo: ciò gli consente non solo di modificare tutto ciò che entra ed esce dal browser, ma anche di modificare il codice che potrebbe interferire con l'automazione del browser.

Cypress controlla l'intero processo di automazione dall'alto verso il basso, il che lo rende capace di comprendere tutto ciò che accade all'interno e all'esterno dal browser. Ciò significa che i risultati che è in grado di fornire sono più coerenti rispetto a qualsiasi altro tool di testing.

Inoltre, essendo installato localmente sul computer, ha accesso al sistema operativo per le attività di automazione. Ciò rende possibili attività come l'acquisizione di schermate, la registrazione di video, operazioni generali sul file system e operazioni di rete.

4.1.1 Cypress vs Selenium

Selenium è uno dei più famosi framework in commercio, che si occupa di automatizzare il testing delle applicazioni web su diversi browser.

Non è un singolo strumento, ma una suite di diversi software, ognuno pensato per soddisfare le diverse esigenze di test di un'organizzazione.

Il Selenium *Integrated Development Environment* (IDE) è il framework più semplice della suite Selenium. È un plugin per Firefox che si installa tanto facilmente quanto un normale plugin. Proprio per questo viene utilizzato solo come strumento di prototipazione. Se si vogliono creare casi di test più avanzati, bisognerà ricorrere a Selenium RC o WebDriver.

Selenium *Remote Control* (RC) è stato per lungo tempo il framework di punta dell'intero progetto, in quanto è in grado di supportare diversi linguaggi di programmazione, come Java, C#, Ruby, Phyton e altri, consentendo all'utente di scegliere quello che preferisce.

Il *WebDriver* si dimostra migliore dei precedenti sotto molti aspetti, implementando un approccio più moderno e stabile nell'automazione delle azioni del browser. WebDriver infatti, a differenza dell'RC, non si basa su JavaScript per l'automazione, ma controlla il browser comunicando direttamente con esso. Questo gli permette di interagire con i più diffusi come Chrome, Firefox, Edge ed Explorer.

Infine c'è il Selenium *Grid*, uno strumento utilizzato insieme all'RC per eseguire test paralleli su macchine e browser diversi, portando così ad un risparmio in termini di tempo. Una caratteristica fondamentale è che agisce da hub, fungendo da fonte centrale dei comandi per ogni nodo ad esso collegato. In questo modo è in in grado di lanciare i test contemporaneamente e gestire le varie configurazioni in maniera centralizzata, piuttosto che per il singolo test. (Guru99, 2018)

Rispetto a Cypress, Selenium presenta degli svantaggi considerevoli:

- se si vogliono salvare i risultati dei test o fare asserzioni è necessario accompagnare Selenium ad altri framework, che quindi complicano e rallentano la fase di inizializzazione,

- i tempi di attesa sono lunghi, in quanto l'esecuzione di un comando su un elemento avviene senza considerare se l'elemento in questione sia o meno presente all'interno della pagina,

- utilizza un protocollo che manda json attraverso la rete (anche se i test sono eseguiti in locale), il che può comportare notevoli ritardi di esecuzione.

La tabella 4.1 mostra i due tools a confronto.

	Cypress	**Selenium**
Browser	Google Chrome	Google Chrome,Internet Explorer Firefox, Microsoft Edge, Safari
Linguaggi	JavaScript	JavaScript, Java, C#, Ruby, R, Python
Framework	Mocha	Mocha, Behave, TestNG, Karma e altri
Target	Sviluppatori	Testers
Esecuzione	Sincrona	Asincrona
Documentazione	Semplice da leggere	Lunga e complessa
Debug	Semplice	Complesso
Community	In crescita	Robusta e consolidata
Setup	Semplice e completa	Lunga e laboriosa

Tabella 4.1. Differenze tra Cypress e Selenium

Per gli scopi di questo lavoro di tesi, ci è sembrato più conveniente utilizzare Cypress, in quanto le caratteristiche a cui siamo maggiormente interessati sembrano essere molto più funzionali in tale ambiente.

4.2 I test

E' stato creato un file JavaScript per ognuna delle pagine dell'applicazione, in cui vengono testate le interazioni principali che un utente può avere con l'app, più uno generale per testare la visibilità dell'applicazione da diversi dispositivi mobile. Ogni suite di test è costituita dalla chiamata alla funzione *context()*, che prende in ingresso il nome della suite e la funzione *beforeEach()*, che indica la pagina da visitare prima di eseguire ogni test. Ogni test, a sua volta, è identificato dal blocco *it()*, che riceve come parametri il nome del test e le istruzioni da eseguire. Tramite il comando *npx cypress run* i test vengono lanciati ed eseguiti sulla CLI, mentre con *npx cypress open* si apre l'interfaccia grafica di Cypress, ed è possibile scegliere quali test eseguire.

La figura 4.1 mostra quello che appare sulla CLI quando viene lanciato il comando *npx cypress run*. E' presente l'elenco di tutte le suite di test trovate, che vengono lanciate in sequenza.

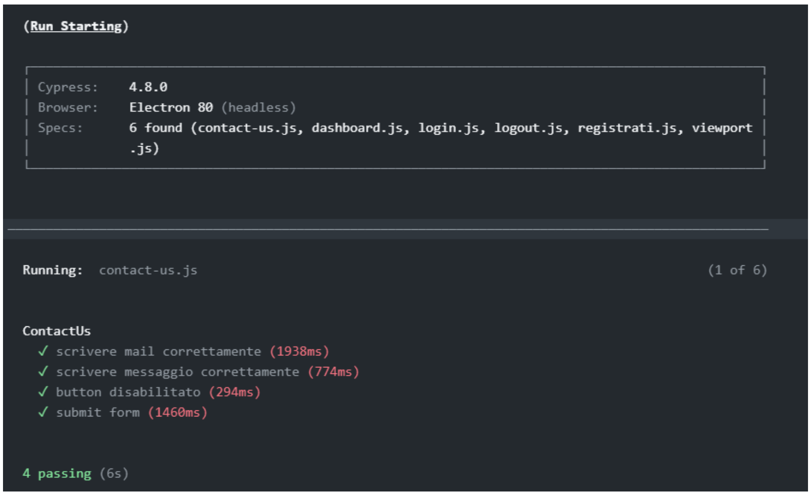


Figura 4.1. Esecuzione di un test su CLI.

Per ogni test viene poi mostrata la durata dell'esecuzione e, come mostrato in figura 4.2, viene salvato anche il video. In caso di test fallito, viene anche memorizzato uno snapshot dell'istante in cui è avvenuto l'errore.

La figura 4.3 mostra il sommario finale di tutti i test eseguiti.

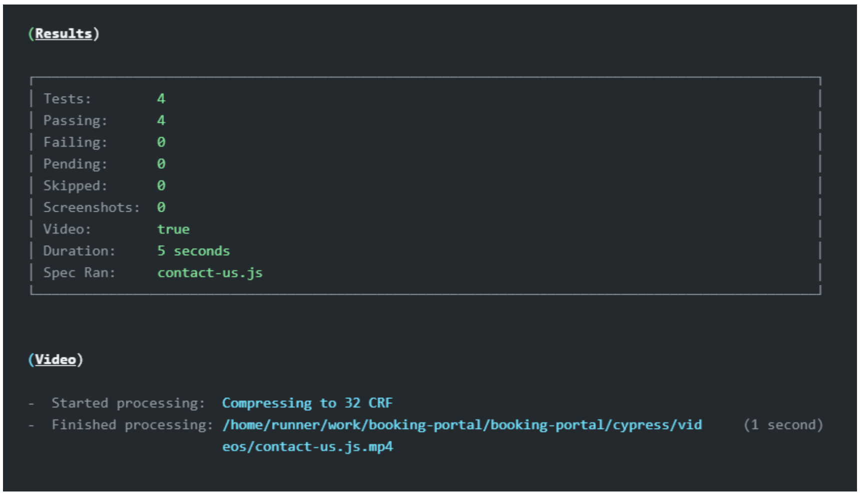


Figura 4.2. Risultati di un test su CLI.

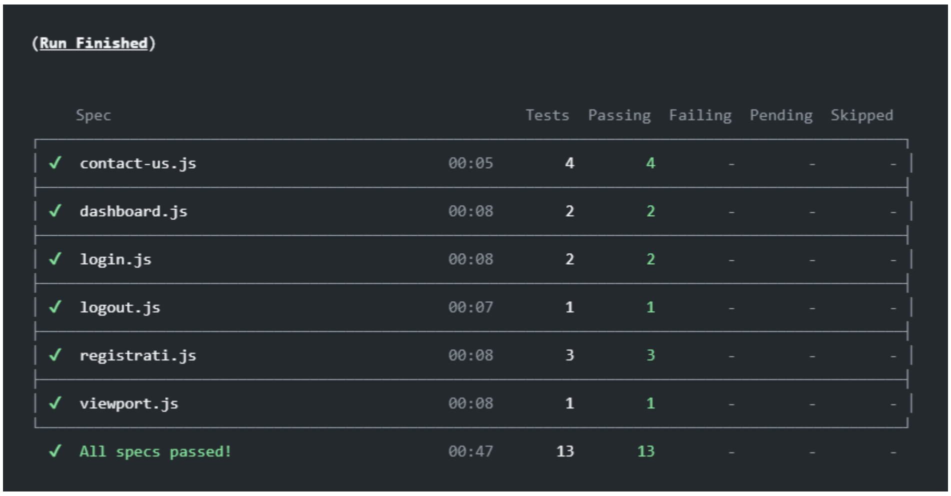


Figura 4.3. Sommario dei test su CLI.

La figura 4.4 mostra invece l'esecuzione di un test su browser, a seguito del comando *npx cypress open*. Nella sezione di sinistra sono presenti nel dettaglio tutte le operazioni eseguite, mentre a destra avviene la simulazione automatica vera e propria.

La tabella 4.2 mostra nel dettaglio i test end-to-end realizzati per le varie pagine dell'applicazione.

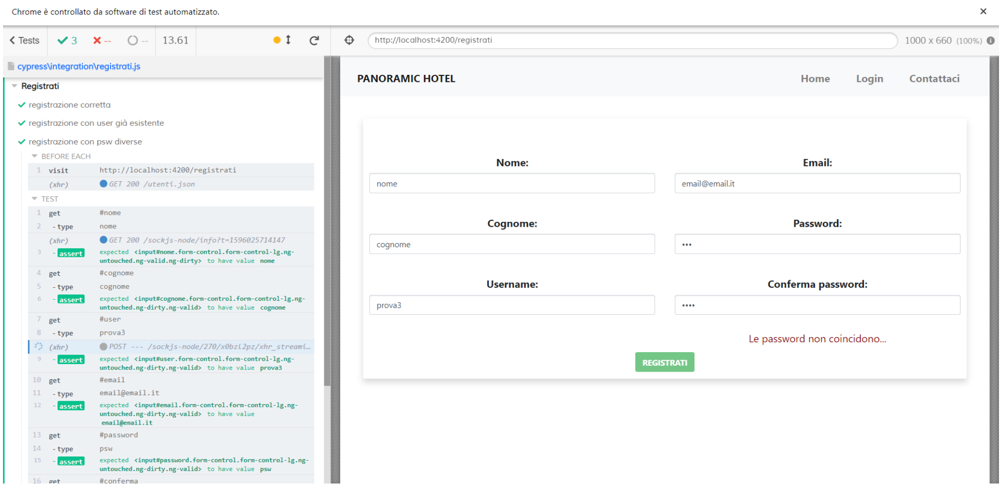


Figura 4.4. Esecuzione di un test su browser.

Pagina	**# Test**	**Descrizione test**
Contattaci	4	Scrivere la mail correttamente
		Scrivere messaggio correttamente
		Bottone disabilitato
		Submit del form
Dashboard	2	Ricerca di un preventivo
		Prenotazione di una stanza singola
Login	2	Login corretto
		Login con password errata
Logout	1	Logout corretto
Registrati	3	Registrazione corretta
		Registrazione con password diverse
		Registrazione con username già esistente

Tabella 4.2. Descrizione nel dettaglio dei test end-to-end

4.3 TDD vs BDD

Il Test-Driven Development è una metodologia tipica della filosofia agile, che consiste nello scrivere i test prima delle funzioni effettive, con l'obiettivo di avere maggior

controllo sull'intero processo di sviluppo.
Il TDD comprende infatti 4 fasi: analisi, design, sviluppo e testing. Prima si analizza il problema, si progetta un'applicazione per risolverlo, poi la si sviluppa e infine la si testa.
Il pattern seguito è del tipo Red-Green-Refactor, ovvero:

- *Red*: scrivere un test che è destinato a fallire, in quanto la funzione che lo soddisfa non è ancora stata implementata.

- *Green*: scrivere il codice minimo per fare passare il test, senza aggiungere elementi che esulano dal suo scopo.

- *Refactor*: una volta che il test ha avuto successo, è possibile andare a modificare il codice scritto in precedenza, rendendolo più efficiente e mantenibile. A questo punto si può scrivere un altro test e si ricomincia da capo.

Il vantaggio del TDD è che costringe a pensare prima alle interfacce e, solo dopo, alla loro implementazione, permettendo cosi di ottenere un'architettura migliore.

Il Behaviour-Driven Development fa sempre parte della filosofia agile e ha lo scopo di migliorare la comunicazione tra sviluppatori e clienti finali.
Questo tipo di approccio sfrutta quelle che vengono definite *user stories* per descrivere le funzionalità, e successivamente si concentra sui criteri di accettazione. Ogni user story è composta da diversi possibili scenari, in cui quel particolare comportamento può verificarsi. I singoli scenari rappresentano i veri e propri test di accettazione, ovvero quelli volti a stabilire se l'applicazione soddisfa i requisiti richiesti.

Per raggiungere il suo scopo, il BDD sfrutta un linguaggio particolare, detto Gherkin, per scrivere sia le user stories che gli scenari. La sua particolarità è quella di utilizzare una terminologia naturale, e quindi comprensibile a tutti i membri del team che si occupa del progetto. (Laramind, 2018)

Quindi prima viene realizzato il *feature file* secondo il pattern Given-When-Then, che descrive gli scenari, e in seguito lo *step file*, in cui si definiscono i vari steps a livello di codice.
In Angular, per girare test di questo tipo, viene utilizzato Cucumber, un tool di cui parleremo a breve, che esegue test automatici scritti, appunto, in Gherkin.

4.3.1 Il pattern Given-When-Then

E' un tipico approccio della filosofia BDD, che consiste nel seguire uno schema preciso quando si descrive uno scenario, utilizzando queste tre parole chiave:

- *Given*: rappresenta lo stato in cui ci si trova prima di cominciare ad eseguire le azioni descritte nello scenario.
- *When*: descrive l'evento che scatena il comportamento che stiamo testando.
- *Then*: presenta cosa avviene nel sistema in seguito al comportamento descritto, e come reagisce l'applicazione.

La figura 4.5 mostra il test scritto per uno scenario in cui si vuole verificare la correttezza di un form. Possiamo notare la presenza della parola chiave ***And***, che viene utilizzata quando ci sono più eventi che innescano il comportamento.

```
Feature: Usare il form contact-us

@contattaci
Scenario: Inviare un messaggio
Given I navigate to the Contact Us section
When I enter "fake@email.com" in email field
And I enter "Hello" in text field
And I click submit button
Then div should contain "Grazie, risponderemo il prima possibile"
```

Figura 4.5. Esempio di feature file.

L'equivalente nella filosofia TDD è il pattern **Arrange-Act-Assert**, in cui le tre azioni corrispondono in ordine, rispettivamente, a quelle appena descritte. La differenza sta nel fatto che, in questo caso, non sono considerate parole chiave e non facilitano il dialogo tra cliente e sviluppatore, in quanto non viene utilizzato un linguaggio diverso. Questo tipo di pattern è utile solo al programmatore, nel momento in cui deve organizzare la struttura del test, e renderne più semplice la comprensione.

4.3.2 Cucumber

E' un tool di testing che supporta l'approccio BDD, e viene quindi impiegato per scrivere test cases mirati a verificare il comportamento di una determinata funzione, servendosi di un linguaggio che risulta di facile comprensione sia a chi scrive il codice, sia all'utente finale che andrà ad utilizzare l'applicazione, ovvero il *Gherkin*. Cucumber è stato inizialmente sviluppato in linguaggio Ruby, per poi diventare in grado di supportare una serie di numerosi altri linguaggi, tra cui anche JavaScript.

I test di Cucumber vengono scritti parallelamente allo sviluppo del codice del software, e vengono definiti *steps*. Per prima cosa, Cucumber legge gli steps descritti all'interno del *feature file*, poi cerca la corrispondenza esatta di ciascun passaggio nello *step file* e, quando la trova, esegue il test case.

4.3.3 I test con Cucumber

Per testare l'applicazione seguendo l'approccio BDD è necessario installare una libreria appositamente creata per rendere Cucumber compatibile con Cypress, ovvero *cypress-cucumber-preprocessor*.
E' stato poi creato un feature file per ogni componente da testare. In questi file andranno descritti i vari scenari seguendo il pattern Given-When-Then. Per ognuno di questi file è stata anche creata una cartella con lo stesso nome, all'interno della quale è presente il file JavaScript che contiene le definizioni dei vari step.

Per fare in modo che vengano eseguiti solo questi test, è sufficiente aggiungere il comando *{"testFiles": "**/*.feature"}* all'interno del file *cypress.json*.

La figura 4.6 mostra come appare ora la schermata di apertura di Cypress, in cui i test precedenti non sono più visibili, ma compaiono solo quelli scritti secondo l'approccio BDD.

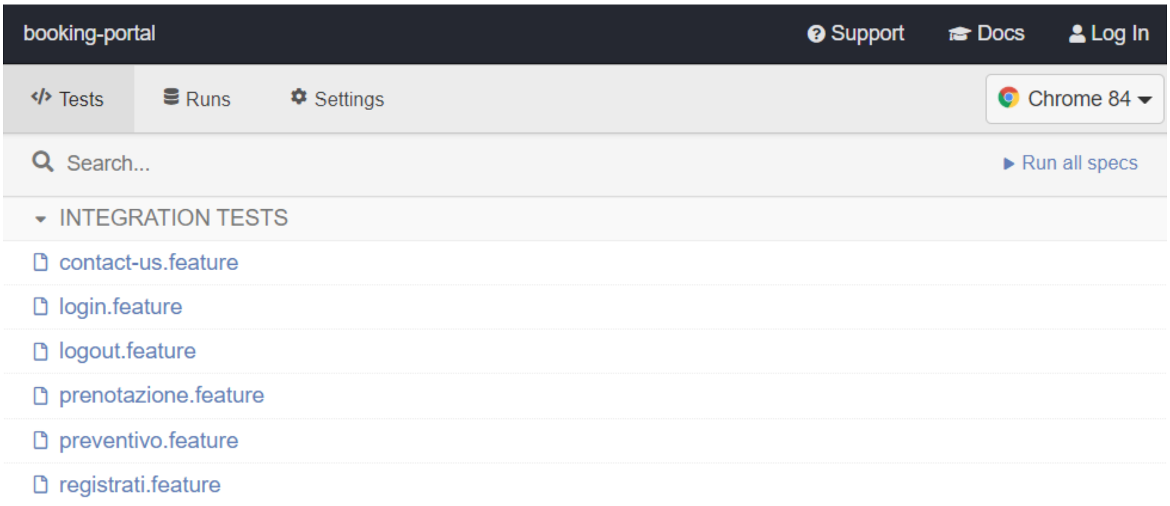


Figura 4.6. Schermata principale di Cypress.

Capitolo 5

Conclusioni

La fase di testing è di fondamentale importanza nel ciclo di vita di un software. E' importante assicurarsi che tutte le funzionalità principali dell'applicazione che sta per essere rilasciata siano il più possibile prive di errori, i quali abbasserebbero di molto la qualità del prodotto e anche l'affidabilità dell'azienda produttrice.
Dal momento che lo sviluppatore non è quasi mai solo, bensì fa parte di un team, in questo contesto avere delle buone metodologie, degli standard e un'ottima comunicazione con gli altri membri aiuta di gran lungo il processo di sviluppo del software, in tutte le sue fasi, garantendo anche maggiore autonomia a ogni membro del team.

L'automazione dei test semplifica al massimo questo aspetto, in quanto permette di introdurre delle nuove funzionalità, senza rischiare di andare ad inficiare su quelle già presenti e collaudate. Così facendo si è più sicuri che ciò che viene aggiunto in un secondo momento non danneggi l'applicazione, evitando così di introdurre delle regressioni potenzialmente molto dannose per il software.

L'obiettivo della CI/CD pipeline, infatti, è proprio quello di creare un flusso di lavoro che esegua una sequenza di azioni ogni qualvolta avvenga una modifica, in modo tale da bloccare la versione del software a quella precedente nel caso in cui una qualsiasi di queste azioni fallisca.

La tesi propone quello che potrebbe essere un modello da seguire durante la fase di testing di un'applicazione, dall'inizio alla fine: partendo dalla realizzazione del software e analizzandone varie metodologie di testing, fino ad arrivare al rilascio finale.

5.1 Risultati

Il lavoro di tesi comprende in totale 38 test unitari, e 12 end-to-end, elencati nel dettaglio nella tabella 5.1, che presenta anche le percentuali di code coverage di ciascun componente.

Componenti	**Unit tests**	**E2E tests**	**Coverage**
Header	1	/	100%
Contattaci	3	4	100%
Dashboard	1	2	92.86%
Card	1	/	100%
Carousel	1	/	55.56%
Preventivo	5	/	100%
Dettaglio	1	/	100%
Login	3	2	87.5%
Logout	1	1	100%
Prenota	2	/	100%
Registrati	4	3	91.3%
App	2	/	100%
Servizi			
Utenti	6	/	88.89%
Stanze	3	/	100%
Messaggi	2	/	100%
Prenotazioni	2	/	100%
Totale	**38**	**12**	**90.58%**

Tabella 5.1. Sommario dei test

Gli strumenti che sono stati utilizzati sono abbastanza recenti sul mercato. Sia Jest che Cypress sono nati da poco, con l'obiettivo di superare i limiti che i grandi nomi, come Karma e Selenium, ormai molto popolari, si portavano dietro. Si sono dimostrati molto semplici da comprendere e da implementare, rivelandosi delle ottime alternative rispetto ai framework più consolidati.

Sfruttando le GitHub Actions, messe a disposizione da GitHub, la tesi porta alla realizzazione di una pipeline in grado di separare perfettamente l'ambiente di testing da quello di sviluppo. Così facendo si raggiungono ottimi risultati in termini di code coverage, la cui reportistica è disponibile interamente online.

La piattaforma di hosting Firebase ha risposto perfettamente alle nostre esigenze, soprattutto grazie alla presenza di un database realtime, che ha reso possibile l'implementazione di una vera e propria REST API.

5.2 Il costo degli errori

L'efficienza dei test, nonché la possibilità di ridurre i tempi ed i costi complessivi del progetto, dipendono in gran parte dall'accuratezza della formulazione dei requisiti. E' per questo che è importante testare il prodotto durante l'intero periodo di lavoro sul progetto, in quanto più tardi si trova un bug, maggiore sarà il costo della sua risoluzione.

Inoltre, cambiamenti significativi apportati nelle prime fasi del progetto non porteranno a notevoli cambiamenti di budget. È molto più economico cambiare il prodotto all'inizio del ciclo di vita di sviluppo rispetto alle fasi finali del progetto, in cui sarebbe necessario cambiare caratteristiche già esistenti, riscrivere il codice sorgente, e far girare altri testi per assicurarsi che l'errore sia stato adeguatamente corretto.
Secondo i dati raccolti dalla società *Hewlett Packard*, il costo per correggere un bug rilevato nell'ultima fase del ciclo di vita del progetto è fino a 100 volte superiore al costo dello stesso bug rilevato nella prima fase, (Rayskiy, 2017), come mostrato in figura 5.1.

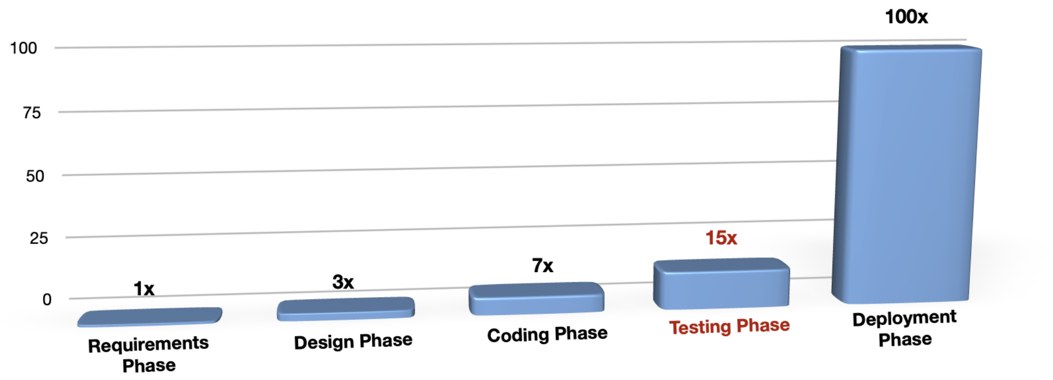


Figura 5.1. Costo della riparazione di un defect.

Possiamo concludere che il costo per riparare un errore all'interno del progetto dipenda fortemente dalla fase in cui viene individuato: più ci avviciniamo alla fine e più aumenta. Nella fase trattata in questo lavoro di tesi, ovvero quella di testing, il costo è ancora accettabile. Per evitare di eccedere il budget prestabilito, sfruttando le metodologie precedentemente descritte, è quindi altamente consigliato coinvolgere un team di testers già nella fase di definizione dei requisiti.

Riferimenti bibliografici

Aibin, M. (2019). *Code analysis with sonarqube.* Retrieved from https://www.baeldung.com/sonar-qube

Bharadwaj, S. (2018). *Why should you switch to cypress for modern web testing?* Retrieved from https://dzone.com/articles/why-should-you-switch-to-cypress-for-modern-web-testing

Bogard, J. (2008). *Arrange act assert and bdd specifications.* Retrieved from https://lostechies.com/jimmybogard/2008/07/24/arrange-act-assert-and-bdd-specifications/

Chandan, A. (2020). *Cypress with cucumber.* Retrieved from https://medium.com/@amarr1143l/cypress-with-cucumber-2791eb345cfd

Ciubotaru, C. (2019). *Test driven development in an angular world — part 1.* Retrieved from https://itnext.io/test-driven-development-in-an-angular-world-92c0c42a54d0

Crispin, L. (2011). *Using the agile testing quadrants.* Retrieved from https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

Deksnis, H. (2019). *Testing your typescript code with ts-mockito....* Retrieved from https://medium.com/passionate-people/testing-your-typescript-code-with-ts-mockito-ac439deae33e

Deschryver, T. (2018). *Integrate jest into an angular application and library.* Retrieved from https://medium.com/angular-in-depth/integrate-jest-into-an-angular-application-and-library-163b01d977ce

DigitalOcean. (2017). *Understanding change detection strategy in angular.* Retrieved from https://www.digitalocean.com/community/tutorials/angular-change-detection-strategy

Ende, M. (2018). *Mock angular providers with ts-mockito.* Retrieved from https://medium.com/@markusende/mock-angular-providers-with-ts-mockito-a44aab871095

Faghy. (2019). *81 rx – cos'è un observable.* Retrieved from https://www.bloccoappunti.it/2019/06/17/81-rx-cose-un-observable/

Fayock, C. (2020). *What are github actions and how can you automate tests and slack notifications?* Retrieved from https://www.freecodecamp.org/

news/what-are-github-actions-and-how-can-you-automate-tests-and-slack-notifications/

Fortunato, F. (2017). *Unit testing angular applications with jest.* Retrieved from https://izifortune.com/unit-testing-angular-applications-with-jest/

Foundation level extension syllabus agile tester, istqb (2014th ed.) [Computer software manual]. (2014, 9).

Foundation level syllabus, istqb [Computer software manual]. (2019, 11).

Fowler, M. (2013). *Givenwhenthen.* Retrieved from https://martinfowler.com/bliki/GivenWhenThen.html

Greenman, C. (2019). *Coverage reporting in angular.* Retrieved from https://medium.com/razroo/coverage-reporting-in-angular-8cae30dae2e4

Gupta, S. (2020). *Automate your deployment with github actions.* Retrieved from https://blog.kiprosh.com/automate-deployment-with-github-actions/

Guru99. (2018). *What is selenium? introduction to selenium automation testing.* Retrieved from https://www.guru99.com/introduction-to-selenium.html

Guru99. (2020). *Devops tutorial: Complete beginners training.* Retrieved from https://www.guru99.com/devops-tutorial.html

HTML.it. (2018). *Guida bootstrap.* Retrieved from https://www.html.it/guide/guida-bootstrap/

Kihl, C.-J. (2018). *Get your npm-package covered with jest and codecov.* Retrieved from https://www.freecodecamp.org/news/get-your-npm-package-covered-with-jest-and-codecov-9a4cb22b93f4/

Kinsbruner, E. (2019). *Cypress vs. selenium: What's the right cross-browser testing solution for you?* Retrieved from https://www.perfecto.io/blog/cypress-vs-selenium-whats-right-cross-browser-testing-solution-you

Krief, M. (2019). *Learning devops.* Packt Publishing.

Laramind. (2018). *Bdd - behaviour driven development.* Retrieved from https://www.laramind.com/blog/bdd-behaviour-driven-development/

Lavecchia, V. (2017). *Testing del software: Tecniche di analisi statica e dinamica.* Retrieved from https://vitolavecchia.altervista.org/testing-del-software-tecniche-analisi-statica-dinamica/

Lavecchia, V. (2018). *Tipologie di testing software: Il test end-to-end (end-to-end testing).* Retrieved from https://vitolavecchia.altervista.org/tipologie-di-testing-software-il-test-end-to-end-end-to-end-testing/

Lavecchia, V. (2019). *Differenza tra unit test, regression test, integration test, system test e acceptance test.* Retrieved from https://vitolavecchia.altervista.org/differenza-tra-unit-test-regression-test-integration-test-system-test-e-acceptance-test/

Moreira, P. S. (2017). *From 0 to 100% coverage real quick.* Retrieved from https://hackernoon.com/from-0-to-100-coverage-real-quick-d71dda7069e5

Nalakath, N. (2020). *Building angular apps using github actions.* Retrieved from https://medium.com/better-programming/building-angular-apps-using-github-actions-bf916b56ed0c

Olsen, G. M. (2019). *Getting started with github actions - ci/cd firebase deploy.* Retrieved from https://dev.to/gautemeekolsen/getting-started-with-github-actions-ci-cd-firebase-deploy-5g87/

Palmer, J., Cohn, C., Giambalvo, M., & Nishina, C. (2018). *Testing angular applications.* Manning Publications.

Petrelius, M. (2019). *Types are not tests and tests are not types.* Retrieved from https://lesscodeismore.dev/types-are-not-tests/

Pittet, S. (2019). *The different types of software testing.* Retrieved from https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

ProfessionalQA. (2019). *Agile testing quadrants.* Retrieved from https://www.professionalqa.com/agile-testing-quadrants

Rayskiy, A. (2017). *Why should testing start early in software project development?* Retrieved from https://xbsoftware.com/blog/why-should-testing-start-early-software-project-development/

Rhettiarachchi, C. (2020). *Why use jest over karma for angular testing?* Retrieved from https://medium.com/@charith.rhettiarachchi/why-use-jest-over-karma-for-angular-testing-b56ffa82f8

Rizzo, N. (2016). *Angular bdd testing made easy.* Retrieved from https://blog.donkeycode.com/angular-bdd-testing-made-easy-ea0be523c4be

Salgarelli, A. (2017). *Metodo agile e metodologia devops, definizioni e differenze.* Retrieved from https://medium.com/4devops/metodo-agile-e

-metodologia-devops-definizioni-e-differenze-2b4aeb87ee0f/

Sall, A. (2018). *Unit testing in angular: Stubs vs spies vs mocks.* Retrieved from https://www.amadousall.com/unit-testing-angular-stubs-vs-spies-vs-mocks/

Sall, A. (2019). *How to set up angular unit testing with jest.* Retrieved from https://www.amadousall.com/how-to-set-up-angular-unit-testing-with-jest/

Schaniel, R. (2018). *Getting started with cypress e2e testing in angular.* Retrieved from https://medium.com/@ronnieschaniel/getting-started-with-cypress-e2e-testing-in-angular-bc42186d913d

Singhal, K. (2018). *How to read test coverage report generated using jest.* Retrieved from https://medium.com/@krishankantsinghal/how-to-read-test-coverage-report-generated-using-jest-c2d1cb70da8b/

Smart, J. F. (2017). *The 3 "a"s for building a great test automation suite.* Retrieved from https://blog.testproject.io/2017/09/04/3-building-great-test-automation-suite/

Stammerjohann, M. (2020). *Github action deploying angular app to firebase hosting.* Retrieved from https://fireship.io/snippets/github-actions-deploy-angular-to-firebase-hosting/

Stocki, M. (2017). *When it comes to mocking in typescript. . . .* Retrieved from https://medium.com/@michal.m.stocki/when-it-comes-to-mocking-in-typescript-be8531d39327/

Tayan, G. (2019). *Cypress vs selenium webdriver: Better, or just different?* Retrieved from https://medium.com/@applitools/cypress-vs-selenium-webdriver-better-or-just-different-2dc76906607d