Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

END-TO-END TESTING PER UN'APPLICAZIONE GESTIONALE SU WEB

Relatore:
Chiar.mo Prof.
FABIO VITALI

Presentata da:
MARKUS AURICH

Sessione III
Anno Accademico 2013/2014

"Failure is not an option.
It comes bundled with the software."

ANONIMO

Indice

Introduzione

"Hai testato che tutto sia funzionante?" mi chiese il project manager. "Sí sí, funziona tutto", rispondo io. Due ore dopo il cliente si lamenta che su Internet Explorer 9 il bottone non invia il form. Ovviamente lo avevo testato, ma non su quel browser in esattamente questa versione.

Scene come queste succedono spesso nel mondo del web developing. Testare manualmente è faticoso, non efficace, ed è un spreco di tempo. Non sarebbe utile automatizzare il processo di testing dell'interfaccia grafica? Lo è, end-to-end testing è nato con questo obiettivo. Lo scopo di questo elaborato è dimostrare le potenzialità e il funzionamento dell'end-to-end testing e valutarne la praticabilità in efficienza ed efficacia applicandolo ad un caso concreto.

Per dimostrare la tesi si descriverà prima di tutto l'ambito scientifico in cui l'elaborato si sviluppa. Nel corso degli ultimi quindici anni, lo sviluppo - e il conseguente successo - del web ha cambiato drasticamente il mondo dell'informatica. Una conseguenza di questo radicale cambiamento é stata la nascita dei paradigmi di programmazione web che, nonostante le critiche di molti sviluppatori della vecchia scuola, hanno contribuito al successo del web.

Ad oggi, nel 2015, Javascript è uno dei linguaggi di programmazione più conosciuti e utilizzati [Was15]: diversi software e applicazioni, una volta esclusivamente stand-alone, sono oggi disponili anche sotto forma di applica-

zioni web. Grazie all'end-to-end testing, esclusivamente applicabile nel web a questo livello di automatizzazione, applicazioni online come Microsoft Office 365, GMail e Spotify, sono capaci di sostituire le classiche versioni desktop. A conferma e prova di ciò questo elaborato è stato redatto usando esclusivamente applicazioni web: ShareLatex come editor latex, Truben.no per le tabelle e Draw.io per disegnare i grafici.

Approfondendo in dettaglio l'end-to-end testing, si é riscontrato come sia un metodo del software testing, in specifico del system testing, che è un procedimento utilizzato per individuare le carenze di correttezza, completezza ed affidabilità delle componenti software, del sistema intero. Oltre al system testing, il software testing può essere suddiviso in unit e integration testing. Mentre con unit testing si intende la verifica di un singolo unit - un singolo componente del software - con integration testing si verificano tanti unit come gruppo. Nessuno di questi livelli è da considerare più importante dell'altro e non è efficace applicare, per esempio integration testing, non avendo verificato ogni singolo unit con unit testing.

A partire già dal 1988, alcuni programmatori iniziavano a scrivere dei test per verificare i loro software e i concetti del software testing venivano applicati da poco nel mondo web: Testing frameworks come QUnit, Jasmine e Mocha, che si analizzeranno nel capitolo 1, sono stati introdotti a partire dal 2008 per rendere possibile unit testing anche su Javascript.

Negli ultimi anni, il web e la sua potenzialità sono cresciuti molto. Così, anche nell'ambito del testing, l'end-to-end testing rappresenta un grande passo in avanti. End-to-end testing è infatti un approccio che consente di valutare l'efficacia di un'interfaccia grafica tramite un processo automatizzato: un notevole vantaggio se si considera che, fino a pochi anni fa, queste operazioni potevano essere svolte solo da persone fisiche con elevati costi in termini di tempo e denaro. Grazie a WebDriver, un'interfaccia HTTP, in specifico un RESTful web service rilasciato nel 2011 che permette il controllo di un browser col fine di eseguire test automatizzati, al giorno d'oggi possia-

mo sfruttare l'end-to-end testing per sviluppare software web di alta qualità. Occorre specificare che un sistema analogo non é disponibile in ambienti stand-alone.

Esistono molteplici interfacce in vari linguaggi per WebDriver, ma nessuna offre un testing tool completo. Dal 2013 in poi sono stati introdotti nuovi tool per l'end-to-end testing: Protractor e Nightwatch, entrambi basati su WebDriver e scritti in JavaScript. Protractor, il tool più completo dei due e creato per AngularJS, riesce a risolvere alcune problematiche dovute all'architettura di single-page-applications.

Per provare a dimostrare l'obiettivo della tesi in un caso concreto, si è scelto di applicare end-to-end testing con Protractor a Buudis, un gestionale e-commerce realizzato con AngularJS dall'autore di questo elaborato analizzandone la sua efficienza ed efficacia. In seguito si elenca una sintesi dei risultati:

- Il grado di difficoltà per impostare ed applicare l'end-to-end testing utilizzando Protractor è adeguato, nonostante sia richiesta una conoscenza ampia di Javascript.
- I costi di manutenzione, che si creano adattando i test per modifiche o estensioni dell'applicazione possono essere alti. Questo dipende molto dalla portata delle modifiche all'applicazione.
- Il debugging è complesso: solo se l'implementazione del test stesso causa l'errore, lo stacktrace aiuta a capire quale sia il problema.
- Con end-to-end testing si possono far emergere errori non identificabili attraverso l'unit testing. Questi sono, nel caso del gestionale, tutti di natura integrativa a causa di dipendenze esterne, del backend o di errori integrativi dei moduli.
- Il risparmio di tempo è significativo.

L'affidabilità dei risultati dipende principalmente dalla scelta dei test case. Anche impostando tutti i possibili test case, i risultati possono mostrare

solo la presenza di errori ma non la loro assenza. Infatti il testing può solo aiutare ad aumentare la qualità del software.

Nel primo capitolo si analizzeranno alcune basi del software testing e delle applicazioni web che ci permetteranno di introdurre l'end-to-end testing nel secondo capitolo dove se ne analizzeranno il funzionamento tecnico e le sue prospettive. Successivamente, nel terzo capitolo, verrà presentato l'uso dell'end-to-end testing applicandolo ad un caso concreto, il gestionale Buudis, per poi dimostrarne la sua efficacia ed efficienza nell'ultimo capitolo.

Elenco delle figure

Elenco delle tabelle

Elenco listati di codice

Capitolo 1

Testing di single-page applications

In questo capitolo si analizzeranno alcune definizioni della letteratura del software testing e di come si applicano test a progetti di software. Inoltre si approfondirà il concetto di single-page application, un modo per strutturare applicazioni web, esaminando come e con quali strumenti vengono applicati Unit Test alle single-page application.

1.1 Definizione di single-page application

Una single-page application (SPA) è un'applicazione web o sito web che consiste in una sola pagina HTML in cui i contenuti vengono caricati dinamicamente con AJAX, una tecnologia per caricare contenuti in modo asincrono mentre che la pagina HTML è visualizzata [MD07]. Mentre Jesse James Garrett, nel suo articolo "A new approach to Web Applications" analizzando Ajax, intende soltanto comunicazione HTTP [Gar+05], in browser moderni la comunicazione può avvenire anche tramite socket.

Come dimostra la figura 1.1, un'applicazione realizzata con una single-page application sfrutta un API REST che fornisce i dati in formato JSON e interagisce direttamente con il Database, mentre un sito normale (figura

1.2) prende direttamente i dati dal database. Un esempio importante di una single-page application è GMail o, in parte, anche Facebook.

Un'applicazione web tradizionale o un sito web è un'insieme di tante pagine HTML linkate tra loro. Per visualizzare altri contenuti è necessario caricare un'altra pagina HTML.

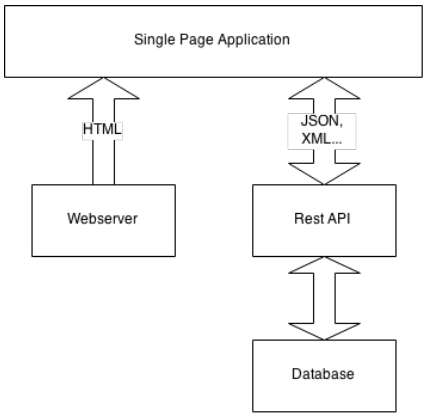

Figura 1.1: Architettura di una single-page application

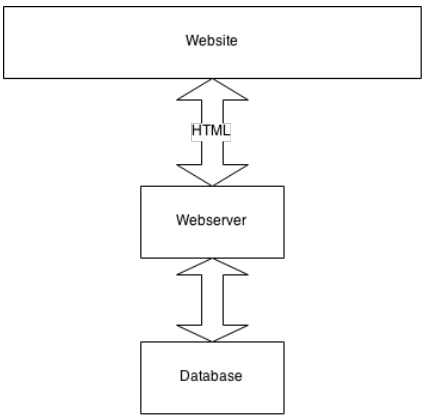

Figura 1.2: Architettura di un sito web o un'applicazione web tradizionale

La nascita delle single web applications risale al 2005 [Gar+05] e quella dei framework, che aiutano a crearli, al 2006: solo negli ultimi cinque anni la loro diffusione ha avuto un forte incremento. Con GWT, uno dei primi framework come si deduce dalla tabella 1.1, si possono creare single-page applications scrivendo il codice in Java che viene compilato in HTML e Javascript, mentre gli altri framework sono tutti scritti in Javascript puro.

Nome	Primo release
AngularJS	2009
BackboneJS	2010
emberJS	2011
ExtJS	2007
GWT	2006

Tabella 1.1: Panoramica di alcuni single-page application frameworks (dati presi dalle loro repository)

Si analizzeranno ora le definizioni di software testing, utili alla comprensione dell'obiettivo di questo elaborato.

1.2 Definizione di software testing

Con software testing si intende un procedimento utilizzato per individuare le carenze di correttezza, completezza e affidabilità delle componenti software.[Pan99]

> Program testing can be used to show the presence of bugs, but never show their absence!
>
> *Edsger W. Dijkstra*

Il software testing può essere utilizzato per mostrare la presenza di errori ma non ne può mai garantire l'assenza: questo avviene perché i test case vengono scelti e non possono mai coprire tutto il codice in tutte le combinazioni di input possibili. È per questo motivo che, in ambito di qualità del software, si parla anche di code coverage, la relazione tra codice coperto da test e codice non coperto da test. Per lo sviluppatore, la difficoltà sta nel raggiungere una grande copertura del codice (code coverage) creando il minor numero di test case [TH02].

Si procede ora ad analizzare i vari livelli di testing del software.

1.3 Livelli di testing

Nell'ambito del software testing, i test possono essere suddivisi in base dell'obiettivo (Objective Testing) o in base dell'oggetto da testare (Target Testing) [Per14, Chapter 4.2]. Si analizzeranno ora le varie suddivisioni di testing.

1.3.1 Objective Testing

Objective Testing va impostato in base all'obiettivo richiesto. Gli obiettivi possono essere funzionali, che verificano il corretto funzionamento del programma, e non funzionali. Proprietà non funzionali possono essere, per esempio, performance, affidabilità o usabilità che non sempre possono essere testate da macchine.

I seguenti obiettivi sono quelli tra i più citati in letteratura: [Per14, Sezione 4.2.2]

- Acceptance Testing
- Installation Testing
- Alpha and Beta Testing
- Reliability Achievement and Evaluation
- Regression Testing
- Performance Testing
- Security Testing
- Stress Testing
- Back-to-Back Testing
- Recovery Testing
- Interface Testing

- Configuration Testing
- Usability and Human Computer Interaction Testing

1.3.2 Target Testing

Nel target testing, l'oggetto da testare può essere un singolo unit del software, tanti singoli unit messe insieme come gruppo o un intero sistema. Nessuno di questi livelli è da considerare più importante dell'altro [Per14, Sezione 4.2.1].

Unit testing

Con unit testing si intende l'attività di testing di singoli piccoli unit software isolate che possono essere singoli metodi o classi [Soc01]. Per la complessità limitata di unit, su questo livello, con unit testing si riesce a raggiungere una copertura di codice (quasi) completa che forma la base per l'integration testing. Nell'ambito degli unit test, le dipendenze con altre unita o sistemi di backend si devono simulare. Se ne vedranno ora alcuni metodi:

- Dummy
  Un oggetto che viene passato nel codice per riempire parametri con valori.
- Fake
  Un oggetto con implementazione limitata nel modo che non possa essere utilizzato in ambienti di produzione. Un esempio tipico di un Fake è un oggetto che simula un Database.
- Stub
  Un oggetto che contiene un metodo che indipendentemente dall'input torna sempre lo stesso valore.
- Mock
  Un oggetto in cui vengono definite delle funzioni che per certi valori

di input si ottengono dei valori di output predefiniti. Per creare un oggetto mock di solito vengono usati Mock Framework.

- Spy
  Un oggetto che protocolla chiamate di metodi e i suoi valori di input. Per creare un oggetto Spy di solito vengono estesi oggetti Fake, Stub o Mock.

- Shim, Shiv
  Una libreria che blocca richieste ad un'API e li tratta con un'implementazione di un oggetto Fake, Stub o un Mock. Nel caso può reindirizzare le chiamata anche ad altri metodi.

Integration testing

Integration Testing è la fase del software testing in cui i vari moduli vengono messi insieme e testati come gruppo. Con integration testing si verificano le interazioni tra le piccole unit di software che, nel caso migliore, sono già stati testati con unit testing [Per14, Capitolo 4.2.1.2].

Le strategie classiche per effettuare Integration testing sono:

- Big Bang
  Un approccio in cui la maggior parte o tutti i componenti del software vengono messi insieme per testare.

- Bottom Up Testing
  Un approccio in cui vengono prima testati gruppi di componenti a livello basso per facilitare i test successivi a livelli più alti. Questo processo va ripetuto finché è raggiunto un livello più alto della gerarchia.

- Top Down Testing
  Un approccio in cui vengono prima testati gruppi di componenti a livelli alti per continuare con gruppi di componenti dipendenti a livelli più bassi.

System testing

Con system testing si intende l'attività di testing del sistema completo. Successivamente, fatto un unit e integration testing effettivo, che ha già eliminato vari difetti, il system testing può comprendere anche interfacce esterne, dispositivi hardware e circostanze esterne come il sistema operativo [Per14, Sezione 4.2.1.3].

Il system testing viene effettuato per verificare le proprietà non funzionali e quelle funzionali. Un metodo del system testing è l'end-to-end testing che esegue il programma come se lo facessero veri utenti controllando in modo automatico l'interfaccia grafica.

Dopo aver analizzato i livelli di testing, si procede ora con la descrizione di processi di sviluppo legati al testing, utile a comprendere la panoramica in cui si colloca l'elaborato.

1.4 Processi di sviluppo legati al testing

Per quanto riguarda l'ambito informatico legato al testing, negli ultimi cinque anni si sono diffusi soprattutto due processi di sviluppo: test-driven development e behaviour-driven-developement. Se ne esamineranno ora le caratteristiche.

1.4.1 Test-driven development (TDD)

Grazie al Test-driven development, un processo di sviluppo agile, i test vengono scritti prima dell'implementazione della componente. Questo processo di sviluppo permette al sviluppatore di ragionare bene sulle specifiche, prima di confrontarsi con i dettagli dell'implementazione. Inoltre assicura che, per ogni parte del programma, siano scritti tutti i test [Amo15].

Il test-driven development è diviso in 5 fasi, come esemplifica la figura 1.3, che vengono applicate per ogni singolo unit di un software:

- Implementazione del test per il problema richiesto
- Esecuzione del test
- Implementazione dell'unita per il problema richiesto
- Ri-esecuzione del test
- Pulizia del codice

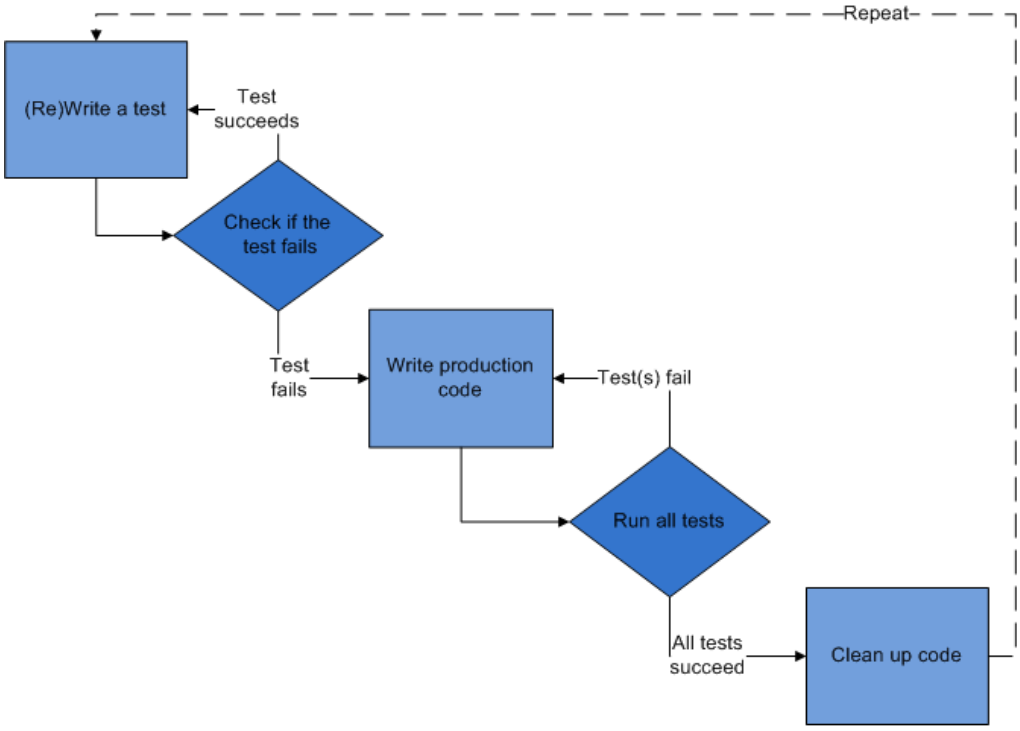

Figura 1.3: Flowchart delle fasi di sviluppo nel Test-driven development
fonte: `http://en.wikipedia.org/wiki/File:Test-driven_development.png`

1.4.2 Behavior-driven development (BDD)

Il Behavior-driven development è un processo di sviluppo agile basato sul Test-driven development. Mentre il TDD è focalizzato sul codice e sulle sue interazioni, questo processo di sviluppo punta sulle specifiche del test che devono essere scritte in maniera chiara, corta e facilmente leggibile. Il codice e l'output del test sono frasi comprensibili, per esempio "La calcolatrice calcola 1 + 1 che deve essere 2". Applicando il BDD si scrivono automaticamente test migliori e più comprensibili rispetto al TDD [Hab13].

```
describe Hash do
  let(:hash) { Hash[:hello, 'world'] }

  it 'includes key' do
    hash.keys.include?(:hello).should be true
  end
end
```

Listing 1.1: Un test rispettando i principi del BDD scritto in pseudo codice

Come si può notare dal listing 1.1 si può vedere un unit test che rispetta i principi del BDD in pseudo codice. Questo test verifica se la componente Hash contiene una stringa della variabile `:hello`.

In queste pagine si sono analizzate le basi teoriche del software testing e delle single-page applications, si vedranno ora gli strumenti per scriverli ed eseguirli su single-page applications.

1.5 Unit testing frameworks per Javascript

In letteratura gli unit testing framework più utilizzati sono Jasmine, Mocha e QUnit. Tutti e tre sono nati indipendenti dal DOM o da altri framework e possono essere eseguiti sia su browser sia in altri ambienti Javascript, co-

me per esempio NodeJS. Si procede ora con una breve panoramica su alcuni framework:

Nome	Stars su GitHub	Contributori	Primo release
Jasmine	8310	89	2010
mocha	6032	197	2012
QUnit	3106	89	2008

Tabella 1.2: Panoramica di alcuni unit testing tool per Javascript (dati presi da github.com il 14.02.2015)

1.5.1 Jasmine

Jasmine, [Gro14] nato nel 2010 come JsUnit, è un testing framework che è molto adatto per il Behavior-driven development per la sua sintassi descrittiva.

```
describe('A suite', function() {
    it('contains spec with an expectation', function() {
        expect(foo).toBe(true);
    });
});
```

Listing 1.2: Esempio di un Unit test scritto con Jasmine

In listing 1.2 si mostra un esempio di un test scritto con Jasmine. Il primo parametro di `describe()` descrive la componente del software che viene testato mentre il primo parametro di `it()` descrive il test case. Con la condizione `expect(foo).toBe(true)` viene verificato se `foo` sia uguale a `true`. Solo se si verificano tutte le condizioni presenti dentro `it()` il test case ha successo.

1.5.2 Mocha

Mocha [Gro14] è un testing framework, nato nel 2012, adatto per Behaviour-driven development e per Test-driven development per le sue interfacce (op-

zionali) legate a questi due sistemi. È molto simile a Jasmine ma un po' meno completo.

```
describe('A suite', function(){
    it('contains spec with an expectation', function(){
        foo.should.equal(true);
    });
});
```

Listing 1.3: Esempio di un unit test sincrono scritto con mocha

In listing 1.3 si mostra un esempio di un test scritto con Mocha. Coma già visto per Jasmine, il primo parametro di `describe()` descrive la componente del software che viene testato mentre il primo parametro di `it()` descrive il test case. Solo la sintassi delle condizioni è diversa: `foo.should.equal(true)`

1.5.3 QUnit

QUnit è nato nel 2008 ed era parte del progetto di jQuery. Nello stesso anno fu estratto diventando indipendente: fu uno dei primi unit testing framework per Javascript. Le prime release erano ancora dipendenti dal DOM finché nel 2009 è diventato completamente stand-alone [She13].

```
QUnit.test('A suite', function (assert) {
  assert.ok(foo, 'contains spec with an expectation');
});
```

Listing 1.4: Esempio di un unit test scritto con QUnit

Lo snippet in listing 1.4 evidenzia un esempio di un test scritto con QUnit e verifica se la condizione `foo` è vera. Dato che QUnit non nasce per il Behaviour-driven development, la sintassi è molto diversa rispetto a Mocha e Jasmine.

Dopo aver elencato vari framework con cui si scrivono i test, si analizzeranno ora alcuni tool per eseguirli.

1.6 Unit test runner per Javascript - Una panoramica di alcuni tool

I Test runner sono essenziali per un sviluppo continuo. Quelli moderni, i cui principali sono Karma e Test'em 'Scripts, possono eseguire i test su uno o più browser normali o headless sia in locale che in remoto. Con "browser nativi" si intendono browser con interfaccia grafica, mentre con "headless browser" si intendono browser senza interfaccia grafica che possono essere controllati dalla linea di comando o altre interfacce. Integrando i test runner in un ambiente di sviluppo con task runner, come Grunt o Gulp, i test vengono eseguiti in continuo e lo sviluppatore comprende immediatamente se un test fallisce oppure no.

Si vedranno ora le caratteristiche dei principali test runner usati per Javascript.

Nome	Stars su GitHub	Contributori	Primo release
Karma	4699	134	2012
Test'em 'Scripts!	1875	74	2012

Tabella 1.3: Panoramica di alcuni unit test running tools per Javascript (dati presi da github.com il 14.02.2015)

1.6.1 Karma

Karma [Kar13] è un test runner sviluppato dal team di AngularJS, non soddisfatto delle soluzioni di test runner esistenti. Karma, rilasciata la prima volta nel 2012, si basa sulla libreria Socket.io e NodeJS e supporta Jasmine, Mocha e QUnit come testing framework ed è supportato da Continous Integration Tools come Jenkins o Teamcity.

1.6.2 Test'em 'Scripts!

Test'em 'Scripts! [Ho12] è un testing framework che si basa su Node-JS e Socket.io e supporta Jasmine, QUnit, Mocha e Buster.js come testing frameworks. Test'em 'Scripts!, rilasciato per la prima volta nel 2012, è supportato da Continuous Integration Tools come Jenkins o Teamcity.

A conclusione di queste pagine introduttive sulle basi del software testing, il funzionamento di una single-page application ed alcuni strumenti per unit testing su Javascript, si può dedurre che questi strumenti, applicati nel mondo del web solo dal 2006, aprono molte possibilità di sviluppo di software e contribuiscono alla professionalizzazione di applicazioni web.

Si procederà nel secondo capitolo allo studio di un metodo del system testing di recente sviluppo: l'end-to-end testing. Si è scelto di analizzare l'end-to-end testing perché è un metodo nuovo, che racchiude in sé un potenziale di sviluppo ancora non conosciuto nel web. E non solo. Pertanto lo studio di questo elaborato vuole essere un contributo alla ricerca in questo ambito.

Capitolo 2

End-to-end testing in single-page applications

Questo capitolo, dedicato all'end-to-end testing, analizzerà le nuove prospettive legate a questo metodo descrivendone le basi tecniche che lo rendono possibile.

2.1 Definizione di end-to-end testing

Con end-to-end testing si intende l'attività di testing dell'interfaccia grafica come la vedono gli utenti del programma dal inizio alla fine [Vic14].

End-to-end testing fa parte del System testing e verifica i flussi e il funzionamento corretto del programma intero. Questo metodo comprende il testing di interfacce e dipendenze esterne come l'ambiente in cui viene eseguito o il backend. Con end-to-end testing, non solo vengono verificate proprietà funzionali, ma possono essere anche verificate proprietà non funzionali come performance o affidabilità.

Esempi di test cases di end-to-end testing possono essere:

- Aggiungere al carrello un prodotto in un online shop
- Avvio corretto di un'intera single-page application

- Flusso di navigazione
- Validazione e invio di un form
- Processi come il login, la registrazione o il check-out in un online shop
- Corretto funzionamento di un form (salva tutti i i dati inseriti nel modo giusto?)

Si passerà ora a descrivere alcuni aspetti più tecnici legati all'end-to-end testing su web per meglio comprenderne le potenzialità applicative.

2.2 WebDriver - La base dell'end-to-end testing per applicazioni web

WebDriver è la base dell'end-to-end testing per applicazioni web ed è un RESTful web service che si appoggia sul protocollo HTTP permettendo il controllo di un browser col fine di eseguire test automatizzati [Sel13b].

WebDriver nasce dal progetto Selenium, che vede il suo inizio nel 2004 grazie a Jason Huggins. Selenium era un tool interno all'azienda per automatizzare il testing di applicazioni web ed è stato rilasciato open source nello stesso anno [Ste07].

Nome	Versioni supportate
Chrome	12+
Firefox	tutti
Internet Explorer	6, 7, 8, 9, 10, 11
Opera	12.x
Safari	5.1+

Tabella 2.1: Browser supportati da WebDriver [Sel13a]

Per sfruttare le potenzialità di WebDriver è necessario installare uno dei driver, in cui è implementato il web service e la comunicazione col brow-

ser. Ogni browser elencato nella tabella 2.1 ha un suo driver che dev'essere installato separatamente.

Come mostra la figura 2.1, per rendere possibile test automatizzati su più browser, o su browser installati in remoto, esiste la possibilità di connettersi a Selenium Server, un server in cui è implemento il web service di WebDriver che manda e raccoglie le richieste e risposte a vari browser definiti.

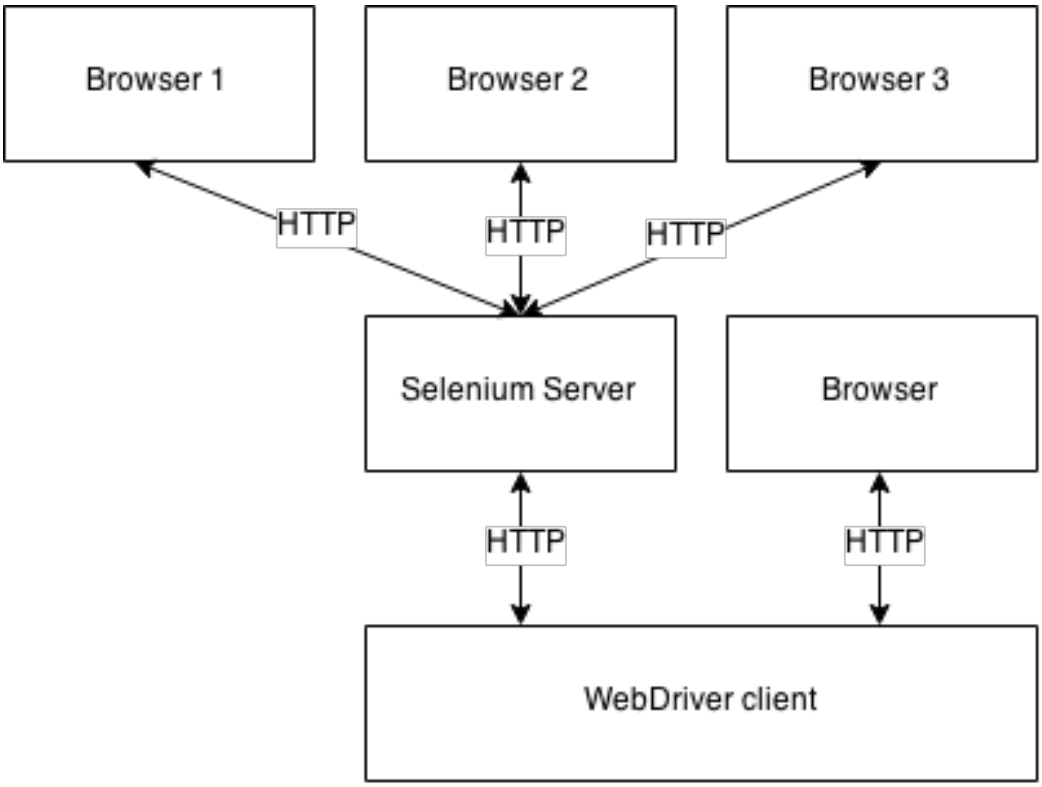

Figura 2.1: I flussi di comunicazione di WebDriver

Si procederà all'analisi delle prospettive di WebDriver in relazione all'end-to-end-testing senza approfondirne l'elevata complessità tecnica, che meriterebbe approfondimenti che non possono essere studiati, per ragioni di spazio, in questo elaborato.

2.3 Nuove prospettive dell'end-to-end testing su web

Nel passato le applicazioni sono state sviluppate solo in maniera nativa. Per quanto riguarda l'end-to-end testing per applicazioni native con interfaccia grafica è difficile automatizzare il processo di testing poiché non esistono tool paragonabili a WebDriver. L'unica possibilità di testare applicazioni native è quella di usare un software per registrare tutti gli input (tastiera, mouse, ecc.) e rieseguire la sequenza per testare. In un secondo momento, cambiando solo la posizione di un bottone, il test dev'essere adattato. Il dispendio di manutenzione dei test è molto alto usando tool di registrazione ed eseguire i test su più dispositivi richiede altri tool di alta complessità.

Con il successo delle applicazioni web, l'end-to-end testing apre nuove prospettive del software testing: sfruttando WebDriver è possibile controllare un browser con comandi sia da locale che da remoto. Di conseguenza l'end-to-end testing diventa facile da gestire, ripetibile e completamente automatizzabile. L'end-to-ending testing tools, come possono essere Protractor e Nightwatch, danno la possibilità di controllare WebDriver con interfacce Javascript definendo così i test con testing frameworks come Jasmine, di cui si é descritto le caratteristiche nei paragrafi precedenti.

Un'ulteriore prospettiva d'innovazione dell'end-to-end testing é la possibilitä di eseguire i test anche in ambienti su server, sfruttando WebDriver e i servizi in cloud, come mostra la tabella 2.2, che offrono la possibilità di eseguire i test su centinaia di combinazioni tra browser e sistemi operativi in versioni diverse. Aumentando il numero di ambienti in cui poter testare, la qualità del testing sulle applicazioni web aumenta.

Dopo aver accennato ad alcune delle prospettive dell'end-to-end testing si vedranno di seguito le problematiche relative al single-page applications.

Nome	Combinazioni browser/OS
Browserstack	700+
Saucelabs	350+
Testdroid	350+
TestingBot	128
Equafy	100+

Tabella 2.2: Panoramica dei servizi in cloud per eseguire test da remoto

2.4 Problematiche di end-to-end testing di single-page applications

Nonostante non vi sia letteratura in merito a questo tema, si puó affermare che vi siano alcune problematiche legate all'end-to-end testing. La piú significativa tra le problematiche riguarda il cambio da una pagina all'altra che non ricarica la pagina HTML e quindi manca un segnale importante per l'end-to-end testing tool. Sono due i principali segnali sintomo della problematica: DocumentReady e WindowLoad. "DocumentReady" viene chiamato nel momento in cui il browser ha finito di costruire il DOM, il modello a oggetti del documento, avendo caricato e eseguito tutto il Javascript e caricato il CSS. Invece l'evento "WindowLoad" viene chiamato dopo il "DocumentReady" avendo caricato anche tutti gli immagini e contenuti. Un end-to-end testing tool deve aspettare almeno l'evento "DocumentReady" per eseguire comandi sul HTML.

DocumentReady e WindowLoad, in single-page applications, vengono chiamate una sola volta all'avvio dell'applicazione. Per ogni cambio di pagina non ci sono eventi standardizzati che vengono chiamati, perchè i contenuti vengono caricati con Ajax e con logiche dipendenti dal framework utilizzato. Protractor, l'end-to-end testing tool sviluppato dal team di AngularJS di cui si analizzeranno alcuni aspetti nel paragrafo 2.5.1, offre una soluzione a questa problematica.

Dopo aver analizzato le basi tecniche e le problematiche principali del

testing di single-page applications, si passerà ora alla descrizione degli strumenti con cui possiamo applicare end-to-end testing ad applicazioni Javascript.

2.5 Panoramica di alcuni end-to-end testing tools

Gli end-to-end testing tool che descriveremo nelle prossime pagine si basano, come già anticipato, su WebDriver; Protractor e Nightwatch sono stati rilasciati di recente, come mostra la tabella 2.3, data la recente nascita di WebDriver nel 2011.

Nome	Contributori	Primo release
Protractor	126	2013
Nightwatch	28	2014

Tabella 2.3: Panoramica di alcuni end-to-end testing tool per applicazioni web (dati presi da github.com il 15.02.2015)

2.5.1 Protractor

Protractor è un end-to-end testing framework scritto in NodeJS per applicazioni AngularJS che può essere utilizzato anche per qualsiasi altra applicazione web [Ang14]. L'architettura di Protractor si basa su WebDriver come é mostrato nella figura 2.2.

Con Protractor è stato risolta una delle problematiche del end-to-end testing su single-page applications: in una single-page application non ci sono segnali che indicano che la pagina ha finito di caricare perchè gli eventi DocumentReady e WindowLoad vengono eseguiti una volta sola all'avvio dell'applicazione. Protractor interagisce direttamente con AngularJS comunicando con la componente `$http`, responsabile per tutte le chiamate Ajax,

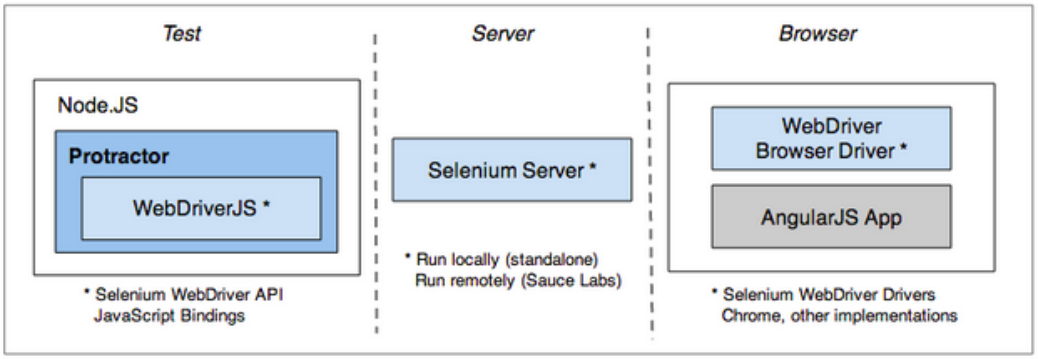

Figura 2.2: Rappresentazione grafica dell'architettura di Protractor
fonte: `http://angular.github.io/protractor/#/infrastructure`

e la componente responsabile per il rendering del HTML. Perciò Protractor riesce a gestire tutti i periodi di attesa tra i cambiamenti di stato da solo [Sch14b].

Inoltre Protractor fornisce un'interfaccia per selezionare elementi DOM secondo l'approccio di AngularJS sfruttando il data-binding. Per esempio con l'espressione `element(by.model('user.email'))` di Protractor si prende l'elemento a cui è legato la variabile scope `$scope.user.email` nell'applicazione AngularJS [Sch14a].

```
describe('angularjs homepage', function() {
  it('should add one and two', function() {

    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();

    expect(element(by.binding('latest')).getText()).toEqual('
        5'); // This is wrong!
  });
});
```

Listing 2.1: Esempio di un end-to-end test utilizzando Protractor

In Listing 2.1 vediamo un esempio di un end-to-end test utilizzando Protractor come test runner e Jasmine (vedi 1.5.1) come testing framework. Questo test apre la pagina web *juliemr.github.io/protractor-demo/* e verifica se dopo aver compilato i campi "first" e "second" con 1 e 2 e inviato il modulo cliccando sul bottone "gobutton", nell'elemento DOM "binding" sia scritto 5.

2.5.2 Nightwatch

Nightwatch è un end-to-end testing framework scritto in NodeJS aperto a qualsiasi applicazione web. Come Protractor, anche Nightwatch si basa su WebDriver. Mentre Protractor funziona anche interagendo direttamente con un browser (driver), Nightwatch pretende Selenium Server come trasmettitore in mezzo.

```
module.exports = {
  "Demo test Google" : function (client) {
    client
      .url("http://www.google.com")
      .waitForElementVisible("body", 1000)
      .assert.title("Google")
      .end();
  }
};
```

Listing 2.2: Esempio di un end-to-end test utilizzando Nightwatch

In Listing 2.1 vediamo un esempio di come un test scritto in Nightwatch verifica se, aprendo il sito www.google.com e aspettando finche l'elemento DOM `<body>` è visibile, il titolo della pagina sia "Google".

Dopo aver studiato e descritto il funzionamento dell'end-to-end testing, le sue potenzialità e le base tecniche per applicarlo, si procede nell'elaborato a fare un'analisi di un caso pratico a cui è stato applicato.

Capitolo 3

End-to-end testing applicato sul gestionale Buudis

In questo capitolo si procederà a studiare un caso pratico comprendendo come e con quali strumenti è stato applicato end-to-end testing alla single-page application "Gestionale Buudis", sviluppato dall'autore di questo elaborato.

3.1 L'idea di Buudis

"Buudis" è una piattaforma web, creata da una start-up austriaca, per fare shopping in gruppo sfruttando sconti di quantità. Ogni prodotto ha tre livelli di prezzo legati ad uno sconto ed il prezzo si abbassa in base al numero degli acquirenti. Sulla piattaforma si possono registrare sia clienti che commercianti. Una volta inserito e pubblicato un prodotto da un commerciante, l'offerta sarà disponibile per 30 giorni, dopo di che agli acquirenti viene addebitato l'importo del prodotto e la merce viene spedita. Dall'altra parte anche i clienti possono esprimere i loro prodotti desiderati facendo delle "richieste" che vengono pubblicate e possono essere votate da altri utenti della piattaforma. Un commerciante può rispondere ad una richiesta facendo un'offerta del prodotto desiderato.

3.2 Il gestionale Buudis

Il gestionale di Buudis è una single-page application realizzata con AngularJS, pensata esclusivamente per amministratori del sistema e protetta da un sistema di login. Sfruttando un'API Rest riesce a manipolare tutti i dati presenti sul database. L'obiettivo del gestionale, disponibile in inglese e tedesco, è quello di dare la possibilità agli amministratori di intervenire quando offerte o richieste sono state inserite male o quando non rispecchiano le condizioni generali di contratto di Buudis. Inoltre certe funzioni, come assegnare più di un utente allo stesso commerciante, sono esclusivamente disponibili sul gestionale.

Le entità principali da gestire sono:

- address
  Indirizzi degli utenti e dei commercianti
- category
  Categorie per le offerte
- company
  Commercianti che possono inserire i loro prodotti
- demand
  Richieste inserite dai clienti
- image
  Immagini di prodotti e richieste
- notification
  Notifiche inviate via mail
- offer
  Prodotti inseriti dai commercianti
- page
  Pagine di contenuto statiche del portale

- user
  Utenti dell'applicazioni

- user_role
  Ruoli degli utenti

La base dell'applicazione sono le liste, come mostra la figura 3.1, di ogni entità che sono realizzate con ui-grid, un modulo AngularJS posto per gestire grandi dimensioni di dati in tabelle. Le liste possono essere ordinate e filtrate per ogni colonna: facendo doppio click sulla riga si entra nel dettaglio dell'entità dove si trovano sia il modulo dell'entità che le altre liste di entità con cui è connesso.

Un possibile sviluppo futuro del gestionale Buudis è l'implementazione di una sezione "statistiche" in cui l'amministratore potrà vedere l'andamento del portale.

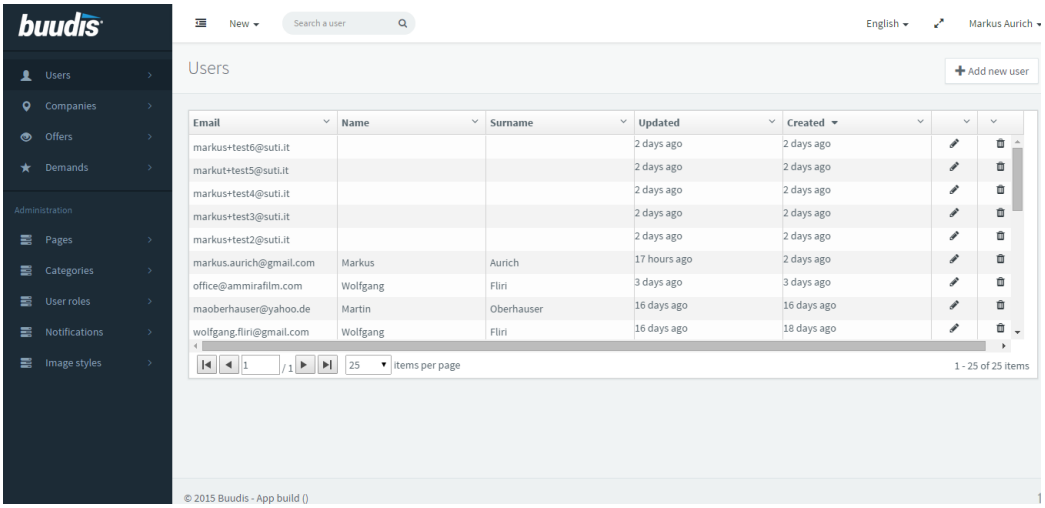

Figura 3.1: Una lista di entità sul gestionale Buudis

3.3 Specifiche tecniche del gestionale

Il gestionale è implementato utilizzando e sfruttando tanti framework e moduli già esistenti, con i quali è stato possibile sviluppare l'applicazione in

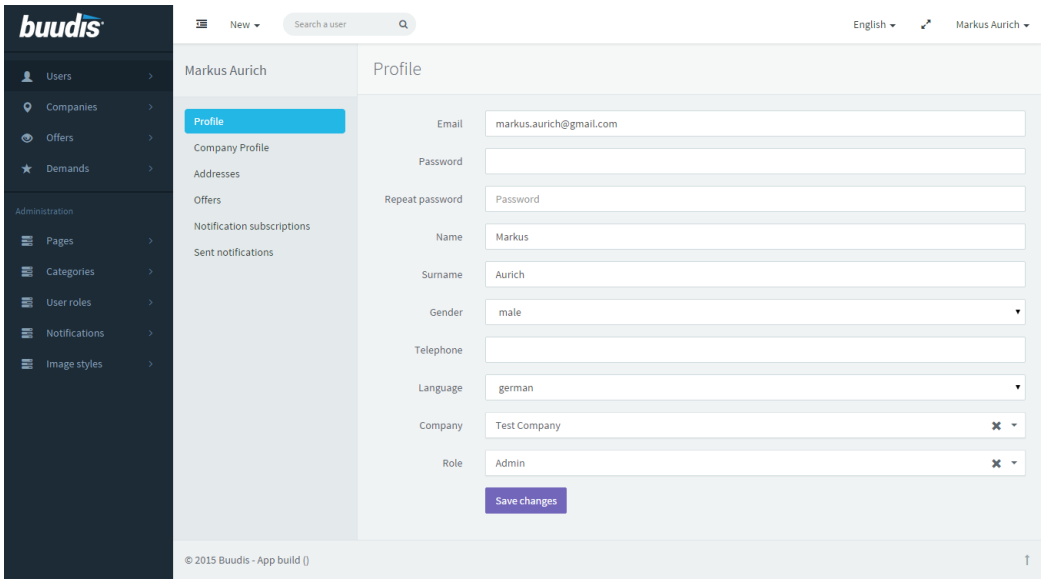

Figura 3.2: La pagina dettaglio di un'entità

breve tempo. Il lavoro principale dell'implementazione consisteva in configurare e impostare i vari framework, nel modo che interagissero tra di loro, e strutturare tutta l'applicazione per le esigenze richieste.

I principali framework e moduli con cui il gestionale "Buudis" è stato sviluppato:

- **Angular v1.3**
  Single-page application framework
- **ui-router v0.2**
  Modulo AngularJS per gestire le rotte, il flusso, dell'applicazione
- **ng-resource v1.3**
  Modulo AngularJS per gestire le chiamata ad un'API Rest
- **ui-grid v3.0**
  Modulo AngularJS per visualizzare i dati sulle tabelle
- **Angulr - Bootstrap Admin Web App v2.0**
  Template di un gestionale per Angular

- **Bootstrap v3.3**
  Frontend framework
- **Grunt v0.4**
  Javascript Task Runner utilizzato per il building dell'applicazione e per eseguire vari compiti durante lo sviluppo in automatico
- **Karma v0.12**
  Javascript Test Runner per eseguire i Unit Test
- **Jasmine v2.0**
  Javascript unit testing framework
- **Protractor v1.0**
  Javascript end-to-end testing framework per Angular

3.4 Automatizzazione e Task runner

Per automatizzare tutti i passi necessari per eseguire i test è stato scelto GruntJS, un Task Runner basato su NodeJS con cui si possono impostare compiti ripetitivi per velocizzare e automatizzare certi processi, come il processo di testing, building o sviluppo.

In particolare, per eseguire tutti compiti necessari per il testing è necessario:

- cancellare tutti i file nella cartella temporanea
- gestire le dipendenze dalla configurazione di bowser
- compilare il css scritto con less
- lanciare un web server
- validare tutto il Javascript per errori e regole di stile
- lanciare tutti i unit test con Karma

- lanciare tutti gli end-to-end test con Protractor

Implementare questi processi in GruntJS, per cui sono disponibili migliaia di moduli, ha inoltre il vantaggio che lo stesso procedimento di compiti può essere eseguito sia in locale che sul server diventando utile se si usano tool di Continous Integration.

```
grunt.registerTask('test', [
  'clean:server',
  'wiredep',
  'less:development',
  'concurrent:test',
  'autoprefixer',
  'connect:test',
  'jshint:all',
  'karma:test',
  'protractor:test'
]);
```

Listing 3.1: Configurazione Grunt per eseguire i compiti necessari per il processo di testing dell'applicazione

Per quanto riguarda lo sviluppo sono stati impostati dei compiti in più. In particolare:

- lasciare Karma acceso in background
- lanciare un servizio che ispeziona tutti i file del progetto e lanciare dei compiti in automatico in caso di salvataggio
  - ricompilare less se è stato salvato un file less
  - validare Javascript se è stato salvato un file Javascript
  - lanciare i Unit Test se è stato salvato un file Javascript
  - eseguire livereload

```
grunt.registerTask('serve', [
  'clean:server',
  'wiredep',
  'less:development',
  'concurrent:server',
  'autoprefixer',
  'connect:livereload',
  'karma:server',
  'watch'
]);
```

Listing 3.2: Configurazione Grunt per eseguire i compiti che facilitano lo sviluppo

Dopo aver analizzato il gestionale Buudis, si passerà ora alla descrizione degli Unit test e end-to-end test implementati.

3.5 Unit testing applicato al gestionale Buudis

L'unit testing da definizione verifica il funzionamento di piccoli unit del software. La condizione ideale degli unit per applicarlo é che quest'ultimi gestiscano un minor grado di complessità. L'implementazione di Buudis prevede, in gran parte, di gestire configurazioni di altri moduli, mettendo insieme tutti i componenti e controllando i flussi. Per questo, presumendo che tutti i componenti esterni siano già stati testati, un'applicazione di questo genere non è molto adatta per applicare unit testing. Rimangono poche le funzioni che devono essere testate.

3.5.1 Testing framework e test runner

Come framework per l'unit testing è stato usato Jasmine, un tool nato nel 2010 con una sintassi molto descrittiva. Come test runner invece è stato usato Karma in combinazione con PhantomJS, un browser headless senza interfaccia grafica. Questo set-up permette di eseguire i test in continuo e

integrandolo bene con GruntJS, l'esecuzione avviene automatica quando si aggiorna un qualsiasi file Javascript del progetto.

3.5.2 Implementazioni

Si mostra di seguito un'implementazione di un Unit test per testare l'assegnazione automatica della lingua:

```
describe('Controller: AppCtrl', function () {

  beforeEach(module('app'));

  var AppCtrl,
    scope;

  beforeEach(inject(function ($controller, $rootScope) {
    scope = $rootScope.$new();
    AppCtrl = $controller('AppCtrl', {
      $scope: scope
    });
  }));

  it('should set the language to de', function () {
    expect(scope.lang).toBe('de');
  });

});
```

Listing 3.3: Unit test per testare l'assegnazione automatica della lingua

3.6 End-to-end testing applicato sul gestionale Buudis

Dato che il gestionale Buudis è molto dipendente dall'API, ed è fatto utilizzando tanti framework diversi implementando soltanto configurazioni e i flussi dell'applicazione, il gestionale è predisposto per l'end-to-end testing.

Sono state impostate 85 rotte, 16 formulari e 22 liste che si adattano, anche dinamicamente, in base a dove vengono integrati. La complessità dell'applicazione non sta su algoritmi, classi o funzioni complicate, ma nella struttura e nei suoi flussi. Facendo delle modifiche, o aggiornando delle dipendenze esterne, si possono facilmente inserire errori che é difficile scoprire perché non sempre escono subito sulla console del browser. Anche applicando unit testing in un'applicazione di questo genere, non si scoprono errori di integrazione perché appena un unit è dipendente da un'altra, sono richiesti dei mock per simulare le dipendenze e l'errore d'integrità non verrà mai scoperto, almeno che non si facciano test manuali controllando ogni singola pagina e testando ogni singola funzione in ogni modo possibile. L'end-to-end testing può aiutare molto per assicurare che, almeno i test case case definiti, siano funzionanti.

3.6.1 Cosa testare e cosa non testare?

La scelta di cosa testare è la parte principale di tutto il processo di testing. Se si scelgono test cases che non rispiecchano gli use case di utenti veri o che gli rispecchia solo in parte, i risultati possono essere assurdi in confronto alla realtà. Invece se si scelgono troppi test case il dispendio di adattare i test dopo modifiche o estensioni all'applicazione è troppo alto.

Sul gestionale Buudis gli use case principali sono:

- autenticarsi
- aggiungere le entità principali
- aggiornare le entità principali
- rimuovere le entità principali

I test case scelti per il gestionale Buudis rispecchiano questi tre use e, inoltre, testano la validazione dei formulari.

3.6.2 Task- e test-runner

Per implementare gli end-to-end test è stato scelto Protractor, che è un end-to-end testing framework molto adatto per AngularJS. Protractor usa WebDriver per mandare i comandi al browser, ma è aperto per tutti browser populari incluse le versione vecchie. Per ragioni di spazio e di analisi, é stato testato solo su Chrome (versione attuale), Firefox (versione attuale), Safari (versione attuale) e Internet Explorer (versione attuale più due versioni precedenti). Per un testing più ampio, i servizi in cloud come BrowserStack o SauceLabs sarebbero delle soluzioni migliori perché offrono una diversità enorme di combinazioni browser, versioni e sistema operativo.

3.6.3 Implementazioni

Si riporta di seguito una scelta di alcuni implementazioni di end-to-end test sul gestionale Buudis scritto con Protractor e Jasmine:

Il test in Listing 3.4 verifica il corretto funzionamento del meccanismo di login e i reindirizzamenti a lui connessi:

```
describe('login mechanism', function() {

  it('/#access should redirect to login page when token is
      invalid', function() {
    browser.executeScript('localStorage.setItem("ls.token",
        "123INVALIDTOKEN123");');
    browser.get('/#access');
    expect(browser.getLocationAbsUrl()).toBe(browser.baseUrl
        + '/#/access/signin');
  });

  it('/#access should autologin', function() {
      browser.get('/#access');
    expect(browser.getLocationAbsUrl()).toBe(browser.baseUrl
        + '/#/user/list');
  });
```

```
  it('/#offer/list should autologin and return to the same
      path', function() {
    browser.get('/#offer/list');
      expect(browser.getLocationAbsUrl()).toBe(browser.
          baseUrl + '/#/offer/list');
  });

});
```

Listing 3.4: End-to-end test del mecanismo di login e autologin

I test in Listing 3.5 verificano il corretto funzionamento del modulo per aggiungere e aggiornare utenti. In dettaglio i test verificano:

- Se compilando solo un indirizzo email il bottone per inviare il modulo sia disabilitato ancora
- Se compilando tutti i campi obbligatori il bottone per inviare il modulo sia abilitato
- Se dopo aver compilato tutti i campi obbligatori l'invio del modulo sia funzionante

```
describe('user add form', function() {

  it('submit button should be disabled', function() {
    browser.get('/#user/add');
    element(by.model('user.email')).sendKeys('markus.
        aurich@gmail.com');
    expect(element(by.css('form[name=userform] button[type=
        submit].add')).isEnabled()).toBe(false);
  });

  it('submit button should be enabled', function() {
    browser.get('/#user/add');
    element(by.model('user.email')).sendKeys('markus.
        aurich@gmail.com');
    element(by.css('select[ng-model="user.language"] option[
        value=de]')).click();
```

```
    expect(element(by.css('form[name=userform] button[type=
        submit].add')).isEnabled()).toBe(true);
  });

  it('should submit successfully', function() {
    browser.get('/#user/add');
    element(by.model('user.email')).sendKeys('markus.aurich+'
        + randChars() + '@gmail.com');
    element(by.css('form[name=userform] select[ng-model="user
        .language"] option[value=de]')).click();
    element(by.css('form[name=userform] button[type=submit].
        add')).click();
    expect(browser.getLocationAbsUrl()).not.toBe(browser.
        baseUrl + '/#/user/add');
  });

});
```

Listing 3.5: End-to-end del modulo per aggiungere e aggiornare utenti

In queste pagine si è provato a descrivere il gestionale Buudis in tutte le sue caratteristiche: si procederà nell'ultimo capitolo a valutare l'efficienza ed efficacia degli end-to-end test sul gestionale Buudis.

Capitolo 4

Valutazione dell'end-to-end testing applicato al gestionale Buudis

In questo capitolo si procederà con la valutazione dell'end-to-end testing applicato al gestionale Buudis, cercando di dimostrarne la sua efficienza ed efficacia.

4.1 Grado di difficoltà

Analizzando la difficoltà del processo di setup del tool Protractor e la sua applicazione pratica, sono state riscontrate le seguenti caratteristiche:

- L'installazione e l'aggiornamento dei driver necessari per il funzionamento di WebDriver avviene con il package-manager di NodeJS (npm), il cui utilizzo è molto intuitivo.

- Protractor offre un'interfaccia semplificata di WebDriver, riducendone la complessità al minimo.

- Problematiche legate all'end-to-end testing di una single-page application, come cambiamenti di stato o richieste Ajax, sono risolti da Protractor, che li gestisce in modo automatico.

- Tutti i metodi di Protractor sono implementati come "Promise": una tecnica per gestire al meglio l'asincronicità.

- Per selezionare oggetti del DOM su Javascript solitamente si usano selettori CSS. Creando la propria applicazione con AngularJS, i selettori CSS non vengono utilizzati, infatti i layouts (views) e la logica implementativa (controller) sono gestiti in modo separato. Protractor offre la possibilità di esprimere selettori anche in base a modelli di dati che sono legati ad oggetti del DOM (data-binding). In questo modo anche i test possono rispecchiare "the angular way" e il legame tra struttura HTML e logica implementativa si riduce al minimo.

- Sfruttando Jasmine come framework per il testing, possono essere usati tutti i suoi metodi di utilità, come `beforeEach()` e `afterEach()`, che facilitano l'implementazione. Inoltre la sua sintassi è specializzata per il Behaviour-driven development (vedi sezione 1.4.2), semplificando sotto tutti i punti di vista la scrittura e lettura del codice.

Avendo analizzato il grado di difficoltà si procede ora con un'analisi dei costi di manutenzione, che sono molti legati alle caratteristiche elencato in questa sezione.

4.2 Costi di manutenzione

Modificando ed estendendo l'applicazione anche i test si devono adattare. Una delle critiche all'end-to-end testing automatizzato, è l'alto costo di manutenzione dei test stessi.[Ral14]

I costi di manutenzione nel caso del gestionale Buudis, si possono suddividere in due gruppi:

- Modifiche di piccoli unit, che riguardano parti specifiche dell'implementazione, ma non il flusso o la struttura completa dell'applicazione
- Modifiche di grande portata che riguardano il flusso o la struttura dell'applicazione

Le modifiche di piccola portata (che non riguardano la struttura dell'applicazione), non producono dei costi alti per adattare i test, perchè andranno ad influenzare sempre pochi test case.

Invece le modifiche che riguardano il flusso o la struttura dell'applicazione, possono essere anche piccole modifiche, ma rischiano di generare delle gravi conseguenze, producendo dei costi di adattamento molto alti: per esempio una ristrutturazione dei path incide tutti i test case dell'end-to-end testing. Qui il costo dell'adattamento dei test diventa più alto del costo dell'implementazione della modifica.

4.3 Debugging

Un'altra critica all'end-to-end testing è il debugging complesso.[Ral14] L'output della console evidenzia in maniera chiara solo quale dei test falliscono, ma non fornisce informazioni aggiuntive. Solo se l'implementazione del test stesso causa l'errore, lo stacktrace aiuta a capire quale sia il problema.

```
Finished in 242.07 seconds
43 tests, 43 assertions, 5 failures

Shutting down selenium standalone server.
[launcher] 0 instance(s) of WebDriver still running
>>
```

Figura 4.1: Esempio di un riassunto dei test in cui 4 verifiche sono fallite

Confrontandolo col testing manuale c'è una perdita di qualità sul debugging. Nel caso un test fallisca, l'unica soluzione è verificare manualmente, attraverso un browser, tutti i passi eseguiti dal test durante la sua esecuzione.

```
6) login mechanism /#access should redirect to login page when token is invalid
 Message:
   Expected 'http://localhost:9001/#/access' to be 'http://localhost:9001/#/access/signin'.
 Stacktrace:
   Error: Failed expectation
  at [object Object].<anonymous> (C:\Users\Markus\buudis-backoffice\test\spec\e2e\login-mechanism.js:390:39)
  at C:\Users\Markus\buudis-backoffice\node_modules\protractor\node_modules\jasminewd\index.js:94:14
  at [object Object].webdriver.promise.ControlFlow.runInNewFrame_ (C:\Users\Markus\buudis-backoffice\node_mo
  at [object Object].webdriver.promise.ControlFlow.runEventLoop_ (C:\Users\Markus\buudis-backoffice\node_mod
```

Figura 4.2: Esempio di un stacktrace di un test fallito

Nonostante che il debugging in parte sia complesso, sono stati emersi alcuni errori, con cui si procede nella sezione successiva.

4.4 Errori emersi

Grazie all'end-to-end testing si possono emergere alcuni errori non visualizzabili attraverso l'unit testing.

Avendo fatto un Unit e Integration testing completo, nell'end-to-end testing ci si poteva concentrare sugli use case principali dell'applicazione (vedi sezione 3.6.1). Gli errori che sono emersi, erano tutti di natura integrativa: moduli che dopo aggiornamenti non funzionavano più come previsto, l'API REST che aveva cambiato la struttura del JSON e causava il malfunzionamento di un modulo, la logica dei token di autenticazione che era cambiata e non si riusciva più ad effettuare il login.

Molto comoda è stata l'implementazione dell'autologin che funziona salvando il token sul localStorage del browser. Nel momento in cui si entra nell'applicazione il token dev'essere validato e l'utente va reindirizzato nella pagina giusta. Ma come reagisce l'applicazione se il token non risulta valido? Per questa casistica sono stati scritti due test specifici prima di implementare la soluzione. La gestione del processo di sviluppo in questa maniera ha portato ad una soluzione veloce ed efficace.

Avendo mostrato gli errori emersi si procederà con un discorso molto importante dell'end-to-end testing: il risparmio di tempo.

4.5 Risparmio di tempo

Con l'end-to-end testing si serializza il lavoro manuale: processi di testing che prima venivano fatti da persone o da team interi, possono essere automatizzati. La serializzazione impedisce errori, risparmia risorse e di conseguenza contribuisce positivamente alla qualità del software.

In questa sezione si prova a dimostrare il risparmio di tempo facendo un calcolo stimato in base al gestionale Buudis:

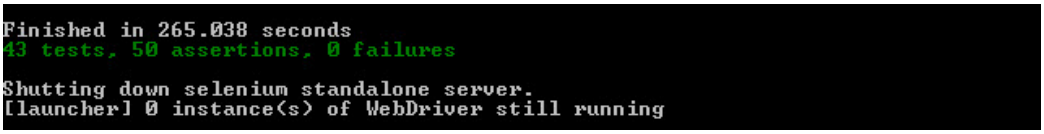

Figura 4.3: Esempio dell'output della console nel caso di successo di tutti i test

Provando ad esprimere il risparmio di tempo dopo aver applicato l'end-to-end testing automatizzato al gestionale Buudis, si possono fare le seguenti considerazione:

Considerati 43 test case manuali, il tempo stimato per il loro completamento, si aggira intorno ai ~13 minuti, che includono tutti i test su un singolo browser. Per ogni browser aggiuntivo la cifra aumenta linearmente, ed effettuando i test, con un setup molto basilare: Chrome (versione attuale), Firefox (versione attuale), Safari (versione attuale) e Internet Explorer (versione attuale più due versioni precedenti), già si arriva a ~78 minuti (vedi tabella 4.1).

Componente	No. di test	Tempo di esec.	Totale
autenticazione	6 x 3	25 sec	75 min
aggiungere un record (per 10 entità diversi)	6 x 10 x 2	20 sec	~27 min
aggiornare un record (per 10 entità diversi)	6 x 10	15 sec	15 min
rimuovere un record (per 10 entità diversi)	6 x 10	15 sec	15 min
			~78 min

Tabella 4.1: Stima per eseguire i 43 test case del gestionale Buudis manualmente su sei browser diversi

Il lavoro stimato per l'end-to-end testing automatizzato, composto dal setup e dall'implementazione dei 43 test, si arriva a sette ore totali come mostrato nella tabella 4.2.

Descrizione	Tempo
setup iniziale (Protractor, WebDriver, Grunt)	3 ore
implementazione di 43 test	4 ore
	7 ore

Tabella 4.2: Stima per setup dell'ambiente per end-to-end test automatizzati

Per valutare i due numeri basta aggiungere un altro fattore: quello del numero di esecuzioni durante il processo di sviluppo dell'applicazione. Non avendo numeri esatti si stima il valore in base al numero di commits fatti sul progetto, dal punto in cui il cliente ha avuto accesso al gestionale (18 in totale).

Il risparmio di tempo stimato, utilizzando la metodologia dell'end-to-end testing applicato al gestionale Buudis, ammonta a ~16 ore, cone è possibile verificare dalla tabella 4.3.

	Set-up	Esecuzione	Totale
Testing manuale	0	23 ore	23 ore
Testing automatizzato	7 ore	0	7 ore

Tabella 4.3: Confronto del testing manuale con il testing automatizzato

Dopo aver dimostrato il risparmio di tempo, finiamo la valutazione con un discorso della scelta test case e dell'affidabilità dei risultati.

4.6 Scelta dei test case e affidabilità dei risultati

> Just because you've counted all the trees doesn't mean you've seen the forest.
>
> *Rameshwar Vyas*
> *CEO di Ranosys Technologies*

L'affidabilità dei risultati dipende principalmente dalla scelta dei test case. Anche impostando tutti i possibili test case, come già abbiamo mostrato nelle pagine precedenti, i risultati possono solo mostrare la presenza di errori ma non la loro assenza. Il testing può solo aiutare ad aumentare la qualità del software.

I 43 test case sul gestionale Buudis, sono stati scelti per rispecchiare quello che dovrebbe essere un comportamento reale nell'utilizzo dell'applicazione. L'end-to-end testing ha contribuito ad aumentare in maniera significativa la qualità del software prodotto, infatti, archiviate queste casistiche (perchè correttamente funzionanti), lo sviluppatore è libero di focalizzarsi sull'implementazione del codice che dovrebbe essere il suo obiettivo primario.

Conclusioni

In questo elaborato è stato presentato l'end-to-end testing applicato al gestionale Buudis, una single-page application con cui è possibile amministrare un sistema e-commerce. Con l'analisi di tale metodo, e la sua applicazione ad un caso concreto, si è voluto evidenziare il potenziale, l'efficenza e l'efficacia dell'utilizzo di questo recente metodo di testing.

L'idea dell'end-to-end testing non è nuova: si tratta di testare l'interfaccia grafica come la vedono gli utenti del programma dall'inizio alla fine. Fino a poco fa questo lavoro era eseguito da persone fisiche, con tutte le problematiche ad esso legate: il lavoro del testing richiede molto tempo e di conseguenza viene fatto solo in parte.

L'end-to-end testing su web è una piccola rivoluzione del software testing. Per la prima volta il grado d'automatizzazione raggiunge un livello in cui l'uso del testing manuale può essere tenuto molto ridotto. Questo metodo di testing non verifica solo il funzionamento delle componenti, ma verifica anche l'interazione corretta di tutte le dipendenze, incluso il backend.

Analizzando l'affidabilità dei risultati si può riscontrare che questa dipende principalmente dalla scelta dei test case. I 43 test case sul gestionale Buudis sono stati scelti per rispecchiare quello che dovrebbe essere un comportamento reale nell'utilizzo dell'applicazione. Avendoli testati con Protractor, un tool utilizzato per single-page applications create con AngularJS, in combinazione con Jasmine, sono emersi alcuni errori di natura integrativa: per esempio alcuni moduli non funzionavano più come previsto oppure l'API REST cambiava la struttura del JSON e causava il malfunzionamento di un

modulo.

A livello di valutazione dell'end-to-end testing applicato al gestionale, si considerano come svantaggiosi il debugging complesso e i costi di manutenzione dei test e vantaggiosi il risparmio di tempo e l'aumento significativo della qualità del software, soprattutto per quanto riguardo lo sviluppo di applicazioni web di una certa complessità.

Per quanto riguardo lo sviluppo di scenari futuri, un primo sviluppo possibile è quello di estendere i test case dell'end-to-end testing. Al momento, solo gli use case principali dell'applicazione sono tradotti in test. Nel caso che gli use case cambino o l'applicazione venisse estesa, anche i test verranno adattati; sempre con l'obiettivo di stabilire la qualità dell'applicazione ad un livello adeguato.

Inoltre il processo di building (pacchettizzazione), testing e deploy (pubblicazione) potrebbe essere automatizzato con un continous integration tool, che esegue quest'ultimi su server e manda notifiche allo sviluppatore nel caso che uno dei processi fallisca. In questo modo, il testing dell'applicazione verrebbe per forza sempre eseguito dopo ogni modifica e solo se il testing andrà a buon fine, la nuova versione potrà essere pubblicata. Di conseguenza il grado di automatizzazione aumenterà di nuovo, garantendo che la versione pubblicata sia testata nel modo più efficace possibile.

In conclusione, grazie all'utilizzo dell'end-to-end testing, aumenteranno la qualità del software e il successo delle applicazioni web.

Bibliografia

Libri

[Hab13] Evan Habn. *JavaScript Testing with Jasmine*. 1st edition (April 4, 2013). O'Reilly Media, 2013. ISBN: 9781449356378.

[She13] Dmitry Sheiko. *Instant Testing with QUnit*. 1st edition (August 2013). PACKT Publishing, 2013. ISBN: 9781783282173.

[Per14] Richard E. (Dick) Fairley Perre Bourque. *Swebok - Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2014. ISBN: 9780769551661. URL: `http://www.computer.org/web/swebok/index`.

[Amo15] Enrique Amodeo. *Learning Behavior-driven Development with JavaScript*. 1st edition. PACKT Publishing, 2015. ISBN: 9781784392642.

Articoli scientifici

[Pan99] Jiantao Pan. «Software testing». In: *Dependable Embedded Systems* 5 (1999), p. 2006.

[TH02] Mustafa M Tikir e Jeffrey K Hollingsworth. «Efficient instrumentation for code coverage testing». In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), pp. 86–96.

[Gar+05] Jesse James Garrett et al. «Ajax: A new approach to web applications». In: (2005). URL: https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf.

[MD07] A. Mesbah e A. van Deursen. «Migrating Multi-page Web Applications to Single-page AJAX Interfaces». In: (mar. 2007), pp. 181–190. ISSN: 1534-5351.

Online

[Soc01] British Computer Society. *Glossary of terms used in software testing*. [ultima vista 14.02.2015]. 2001. URL: http://www.testingstandards.co.uk/bs_7925-1_online.htm.

[Ste07] Simon Stewart. *GTAC 2007: Simon Stewart - Web Driver*. [ultima vista 12.02.2015]. 2007. URL: https://www.youtube.com/watch?v=tGu1ud7hk5I.

[Ho12] Toby Ho. *Test'em 'Scripts! Readme*. [ultima vista 14.02.2015]. 2012. URL: https://github.com/airportyh/testem/blob/master/README.md.

[Kar13] Karma. *Karma Readme*. [ultima vista 14.02.2015]. 2013. URL: https://github.com/karma-runner/karma/blob/master/README.md.

[Sel13a] Selenium. *Platforms Supported by Selenium*. [ultima vista 16.02.2015]. 2013. URL: http://docs.seleniumhq.org/about/platforms.jsp.

[Sel13b] Selenium. *Selenium WebDriver*. [ultima vista 16.02.2015]. 2013. URL: http://docs.seleniumhq.org/projects/webdriver/.

[Ang14] Angular. *Protractor - How It Works*. [ultima vista 15.02.2015]. 2014. URL: http://angular.github.io/protractor/#/infrastructure.

[Gro14] Kevin Groat. *Which Javascript test library should you use? QUnit vs Jasmine vs Mocha*. [ultima vista 14.02.2015]. 2014. URL: http://www.techtalkdc.com/which-javascript-test-library-should-you-use-qunit-vs-jasmine-vs-mocha/.

[Ral14] Julie Ralph. *End-to-end Angular Testing with Protractor*. [ultima vista 23.02.2015]. 2014. URL: https://www.youtube.com/watch?v=aQipuiTcn3U.

[Sch14a] Gabriel Schenker. *Angular JS–Part 14, End to end tests*. [ultima vista 07.02.2015]. 2014. URL: http://lostechies.com/gabrielschenker/2014/04/18/angular-jspart-14-end-to-end-tests/.

[Sch14b] Benjamin Schmid. *End-to-End-Tests mit Protractor*. [ultima vista 15.02.2015]. 2014. URL: http://www.heise.de/developer/artikel/End-to-End-Tests-mit-Protractor-2461535.html.

[Vic14] Ramon Victor. *Protractor for AngularJS - Writing end-to-end tests has never been so fun*. [ultima vista 15.02.2015]. 2014. URL: http://ramonvictor.github.io/protractor/slides/.

[Was15] Todd Wasserman. *15 programming languages you need to know in 2015*. [ultima vista 24.02.2015]. 2015. URL: http://mashable.com/2015/01/18/programming-languages-2015/.