Teammates Manager: una web application per la gestione del proprio team

Paolo Innocenti and Stefano Vannucchi

Università degli Studi di Firenze, Florence, Italy

1 Introduzione

Entrando nell'ottica moderna, in cui si ricerca un'elevata flessibilità e la possibilità di riutilizzare le proprie informazioni e servizi in più contesti, risulta predominante il modello di architettura *RESTful*. Grazie a quest'ultimo, infatti, è possibile spostare le operazioni computazionalmente onerose in carico ad un server, promuovendo quindi la separazione client-server a vantaggio di un client più leggero per i dispositivi utente e rendendo fruibili i servizi ad una varietà di dispositivi e applicazioni, sia mobili che non.
Partendo da tale idea, questo progetto si basa sulla realizzazione di una web application composta da un back-end implementato con l'utilizzo di *Spring Boot framework*, ed un front-end, creato con *Vue.js*, che consuma i servizi esposti dal primo.

Il tema trattato è quello relativo alla gestione (inserimento, visualizzazione, cancellazione e aggiornamento) di "membri del proprio team", definiti nell'applicativo come *teammates*, ai quali è possibile attribuire una serie di competenze, denominate *skills*. Tali *skills* possono essere sia selezionate tra quelle già esistenti, che aggiunte attraverso l'inserimento in una *multiselect* che permette l'operazione di "tagging".
Il modello di dominio individuato risulta, quindi, composto da due entità: *Teammate* e *Skill*. A livello di database, per il quale è stato scelto *MySQL*, questo modello viene realizzato con due tabelle per le entità ed un'ulteriore tabella intermedia per realizzare la relazione molti a molti presente tra le due entità.

2 Tecnologie utilizzate

L'intero progetto è stato svolto seguendo il modello di sviluppo **test driven development**, che prevede la stesura dei test automatici prima della scrittura del *System under test* stesso. In questo modo è possibile creare un applicativo robusto e ben testato in ogni suo *path*, capace di rispondere bene alle esigenze dell'utente finale e garantendo la corretta fruizione delle sue funzionalità. Inoltre i test stessi potranno essere consultati in seguito ed utilizzati come documentazione "attiva" del software.
Oltre l'applicazione della *TDD*, durante lo sviluppo degli applicativi è stato fatto

uso di numerose tecnologie, volte al miglioramento e al perfezionamento del risultato finale. Il progetto, mantenuto in un *repository* dedicato su *GitHub*, è stato integrato con altri servizi quali *Codecov* e *SonarCloud*.

Il primo è un tool online utile per il collezionamento della *code coverage*, intesa come percentuale di linee di *production code* coperte durante l'esecuzione dei *tests*.
Il secondo, invece, rappresenta la versione disponibile in cloud del noto strumento di *code quality analysis*, *SonarQube*. Quest'ultimo risulta particolarmente utile essendo in grado di notificare allo sviluppatore eventuali mancanze o miglioramenti che possono essere introdotti nel codice, al fine di generare un risultato più stabile e soprattutto più manutenibile nel corso del tempo.
Le integrazioni tra il *repository* e i vari strumenti sopra citati sono state rese possibili grazie all'utilizzo di un *Continuous Integration server*, nella fattispecie *Travis CI*. Mediante esso è, infatti, possibile sia automatizzare l'intero processo di compilazione del progetto, con conseguente esecuzione dei tests, che definire una serie di operazioni da eseguire una volta completata la *build*, tra cui contattare *Codecov* e *SonarCloud*, inviando contestualmente i report ottenuti nella precedente fase di testing.

Scendendo più nel dettaglio delle tecnologie utilizzate, risulta necessario concentrarsi in primo luogo sulla struttura generale del progetto, nonché operare successivamente una distinzione tra back-end e front-end.

Il progetto finale risulta composto da un front-end nidificato all'interno del back-end; scelta operativa necessaria al fine di rendere facilmente riproducibile l'intero processo di compilazione in locale e la distribuzione del software all'utilizzatore finale. Inoltre, seguendo il principio di inversione delle dipendenze, il risultato è quello di un back-end non dipendente dalla *user interface*, con, allo stesso tempo, un front-end fortemente dipendente dalla *business logic*.
Ruolo chiave per l'intero progetto è svolto da *Maven*. Esso rappresenta uno strumento utile per la gestione di progetti basati su *Java*, andando a semplificare il processo di compilazione dei sorgenti e introducendo un servizio di gestione delle dipendenze. Il risultato è un processo di *build* che automatizza sia il download delle dipendenze che la compilazione del progetto con conseguente esecuzione dei tests.
Nel contesto specifico l'utilizzo di suddetto strumento ha reso possibile la creazione di appositi *profiles*, necessari per l'esecuzione dei tests sia di front-end che di back-end.

Per quanto riguarda il back-end, data la necessità di persistere le informazioni relative ai *teammates* e alle loro *skills*, è stato scelto di utilizzare un database relazionale, in particolare un database *MySQL*, eseguito in un container *Docker*. Quest'ultimo costituisce una valida soluzione alternativa per la virtualizzazione di server, in quanto, eseguendo tutti i containers sullo stesso *kernel* (quello

dell'host) fa sì che diventino molto più "leggeri" rispetto alle classiche macchine virtuali.
Sempre in relazione al back-end, la stesura dei tests è stata fatta sfruttando le potenzialità di *JUnit*, testing framework per eccellenza quando si parla del linguaggio di programmazione *Java*, abbinando le sue classiche funzionalità con quelle messe a disposizione da altre opzioni quali *Hamcrest*, *AssertJ* e *REST Assured*.
Infine, per verificare l'efficacia degli unit test è stata predisposta una "sessione" di *mutation testing* grazie all'utilizzo di *PIT*. Questa tipologia di test ha l'obiettivo di generare dei "mutanti", ovvero variazioni del *production code* tramite l'inserimento di piccole modifiche, verificando conseguentemente l'esito dei tests. In questo frangente il successo viene calcolato in maniera opposta: si supera il test se la mutazione introdotta fa fallire almeno uno dei tests coinvolti.

Per quanto riguarda il front-end, invece, l'idea di sviluppo alla base permane la medesima del back-end, con l'adozione di tecnologie analoghe disponibili in ambiente *Javascript*. In particolare per gestire le dipendenze e automatizzare i processi di build e test, è stato sfruttato *npm*, principale *Node package manager*. Per la stesura degli *unit* e *integration* tests, è stato adottato *Jest*, testing framework sviluppato e messo a disposizione da *Facebook Inc.*, mentre per gestire la sessione di *mutation testing* è stato fatto uso del framework *Striker*, disponibile per vari ambienti di sviluppo tra cui *Javascript*.
Infine, per gli *e2e tests*, è stato utilizzato *Cypress* in combinazione con *Cucumber*, dandoci la possibilità di scrivere i test in linguaggio naturale e incrementando la loro forza esplicativa da un punto di vista di documentazione del codice.

Tornando in merito all'intero progetto, per il collezionamento della *code coverage* si è scelto di lavorare con *Codecov*, a discapito del noto strumento *Coveralls*, a causa della necessità di collezionare ed unificare più files di report, creati separatamente durante l'esecuzione dei tests di back-end e front-end. Sebbene i notevoli sforzi non è stato possibile utilizzare *Coveralls*, sostituito da *Codecov* che è risultato capace di soddisfare l'esigenza riscontrata.

Elemento finale, non per importanza, è il deploy automatizzato sulla piattaforma *Heroku*, una volta che la fase di build e tutti i tests automatici hanno avuto successo. Questo serve per rendere disponibile online l'applicativo, in maniera tale che il suo funzionamento possa essere testato anche da un operatore umano.

3 Il modello di dominio

Ripartendo dal modello di dominio, esso è formato, come abbiamo precedentemente detto, da due entità, *Teammate* e *Skill*, tra le quali si sviluppa una relazione molti a molti: un *teammate* potendo logicamente avere diverse competenze specialistiche, può essere legato a più di una *skill* contemporaneamente, mentre la relazione delle *skills* con i *teammate* è ovvia.

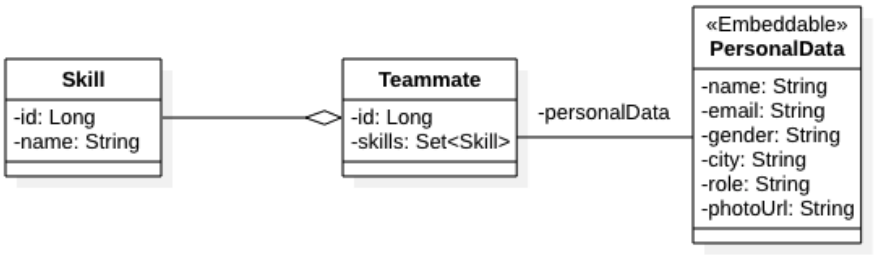

Fig. 1. Modello di dominio dell'applicazione

Questo modello è stato tradotto a livello di back-end mediante un accurato mapping realizzato sfruttando le annotazioni messe a disposizione dalla specifica *Jpa*. In particolare è stata creata una classe *Embeddable* denominata *PersonalData*, contenente tutte le informazioni principali del *teammate*, integrata all'interno della *Entity* chiamata *Teammate*. All'interno di queste informazioni è contenuta anche l'indirizzo mail del *teammate*, che è stato definito come un campo *unique* all'interno del database. Tale unicità è stata quindi verificata sia nelle fasi di inserimento che di update, in modo tale da evitare la generazione di eccezioni direttamente dal database e l'innesco di situazioni di *rollback*, e gestita tramite l'utilizzo di un'eccezione, *TeammateAlreadyExistsException*, definita estendendo l'eccezione *RuntimeException*, propria di *Java*.
Oltre che le informazioni appena citate, la classe Teammate contiene anche un *Set<Skill>*, inteso come insieme delle competenze del singolo *teammate*.
Tali *skills*, rappresentate con una seconda *Entity*, mappano esattamente l'omonima entità presente nel modello di dominio, tramite l'utilizzo di un id, automaticamente generato, e di un nome.

3.1 Struttura dell'applicazione

3.2 Backend

Per quanto riguarda invece la realizzazione della struttura del back-end, si è scelto di sviluppare i vari *layers* in maniera separata come fossero ognuno una feature distinta dell'applicativo, identificando quindi 3 layers: *persistence layer*, *service layer* e *controller layer*. Nel primo layer è stata trattata la relazione del back-end con il database, definendo il punto di collegamento tra la logica del *service layer* e i dati. Nel *service layer* invece è stata definita la logica dei servizi esposti, identificando e costruendo i servizi richiesti dall'applicativo. Infine, nel *controller layer*, sono stati realizzati gli end-point esposti esternamente all'applicativo, basati sui servizi definiti nel layer precedentemente descritto. Per permettere lo sviluppo disaccoppiato dei singoli layer, per ciascuno è stato creato un branch dedicato nel repository e sono stati scritti tutti gli unit test facendo uso di *mocks* e *fake implementations* per gli oggetti di competenza degli altri

layers.
Una volta sviluppati i vari layers in isolamento è seguita la fase di integrazione delle varie componenti. Anche questa operazione è stata realizzata in un *branch* dedicato, integrando due layer per volta, verificando che gli unit test continuassero ad avere successo anche dopo il merge e scrivendo i relativi integration tests.
Infine, dopo aver completato tutto il processo di integrazione, sono stati aggiunti alcuni e2e tests, basati su *REST Assured*, per verificare l'utilizzabilità del back-end esternamente al contesto applicativo basato sul front-end.

3.3 Frontend

Il front-end, invece, consiste di un'applicazione single page che permette la fruizione di tutte le funzionalità necessarie alla gestione dei *teammates*. Il design semplice e accattivante è stato realizzato sfruttando il css *Semantic UI* in unione con alcune regole di stile personalizzate sulla base delle necessità riscontrate.
Seguendo la versatilità del framework alla base del progetto, sono stati realizzati

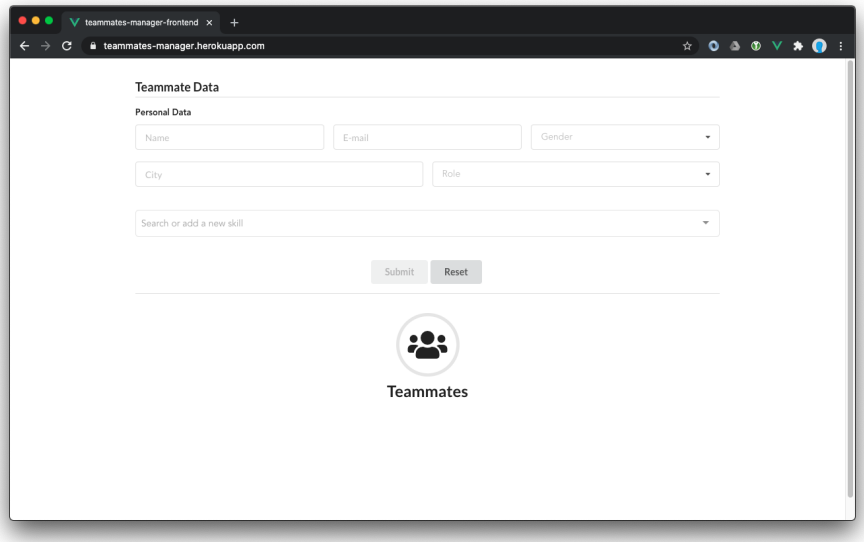

Fig. 2. Immagine del frontend

vari componenti, successivamente integrati nella pagina principale dell'applicativo. L'interfaccia grafica realizzata è costituita da un form, utilizzabile per l'inserimento e la modifica di un teammate, e da una lista di teammates. All'interno del form,

posto nella parte alta della pagina, troviamo elementi di stile propri del css utilizzato, quali *dropdown*, ed un componente *multiselect*, integrato grazie al gestore di dipendenze npm. Quest'ultimo componente è stato inserito all'interno del componente *TagMultiselect* in modo tale da poterlo meglio adeguare al nostro caso d'uso, ed è stato sfruttato l'evento tag che esso genera per richiamare il metodo di aggiunta di nuove skills.
Nella parte della pagina sottostante il form troviamo invece la lista dei teammates correntemente persistiti all'interno dell'applicazione. Ogni singolo teammate viene visualizzato all'interno di una card girevole, dove nella sua faccia anteriore sono riportate le sue informazioni generali, mentre nel lato posteriore sono elencate le varie skills oltre che ubicati i pulsanti per la cancellazione e la modifica del teammate corrente. La rotazione animata della card, che avviene automaticamente portando il mouse sulla stessa, è stata realizzata mediante l'applicazione di regole di stile ad hoc necessarie per ottenere un effetto gradevole.
Da un punto di vista di design del codice, nella fase di sviluppo si è scelto di separare l'applicazione in due parti, form e card, essendo le due parti logicamente disaccoppiate e tali da prestarsi bene, quindi, ad essere sviluppate contemporaneamente in maniera separata. Inoltre si è scelto di separare la logica di gestione delle informazioni dei vari teammates dalla parte di esecuzione delle richieste http necessarie alla comunicazione con il back-end, riducendo al minimo il punto di accoppiamento tra back-end e front-end, in modo tale da richiedere il minimo intervento sul front-end in caso di modifiche del back-end. A tal proposito è stato quindi introdotto un service layer dedicato contenente la classe *ApiService* con in carico i metodi per le chiamate ai REST endpoints. Tale scelta è state effettuata anche per isolare il componente terzo, *axios*, utilizzato per tali chiamate, che essendo tale non necessitava di essere testato in questo contesto.
Altra scelta implementativa rilevante, in merito alla realizzazione del front-end, riguarda la politica introdotta per gestire la comunicazione tra i vari componenti Vue che formano l'intera applicazione. In particolare, utilizzando *Vue.js*, è possibile creare un canale di comunicazione diretto tra componente padre verso uno specifico componente figlio, ma tale collegamento è percorribile nella sola direzione specificata. Talvolta, però, può essere necessario dover inviare informazioni in senso inverso.
Nel presente progetto, ciò è risultato necessario per dare notifica al componente principale, contenente tutta la logica dell'applicazione, dell'avvenuta pressione del tasto di eliminazione o aggiornamento di uno specifico teammate.
Per arrivare all'obiettivo prefissato, si è scelto di utilizzare la metodologia di "emissione di eventi" fornita dal framework stesso. In questo modo è possibile emettere uno specifico evento, al quale può essere connesso anche un eventuale payload, intercettabile soltanto dal componente padre.
Vue.js mette a disposizione anche un'ulteriore libreria ideata per la gestione dello stato di un'applicazione, denominata *Vuex*. Attraverso quest'ultima è possibile, infatti, generare uno stato globale e condiviso tra tutti i vari componenti.
Quest'ultima risulta una valida alternativa all'emissione di eventi, utile soprat-

tutto nel caso in cui risulti necessario condividere tra molti componenti le stesse informazioni aggiornate. Infine, un'attenzione particolare è stata data al design dell'applicativo, utilizzando *Semantic UI*, framework di sviluppo che semplifica la realizzazione di uno stile moderno. L'utilizzo di questo framework ha reso possibile modificare lo stile di alcune componenti di base, come le select, gli input, le icone e la griglia, tramite l'utilizzo di semplicissime classi css, evitando anche la necessità di dover aggiungere manualmente il numero elevato di classi e regole di stile che sarebbe stato richiesto.

3.4 Problematiche emerse

Durante lo svolgimento del progetto sono state tuttavia incontrate alcune problematiche che hanno richiesto un'analisi e gestione particolari.

Per quanto riguarda il back-end, l'unico problema incontrato è stato nel testing dei metodi del controller che producevano eccezioni. Più precisamente, inizialmente si era fatto affidamento sul fatto che Spring Boot propagasse l'eccezione e la fornisse anche come output del servizio richiamato tramite URI. Tuttavia, sebbene nella pratica succedesse questo, i test relativi fallivano perché veniva generata un'eccezione non gestita. Per rimediare a questo, è stato definito un *TeammateRestControllerExceptionHandler*, estendendo la classe *ResponseEntityExceptionHandler*, con il compito di intercettare le eccezioni da noi definite e di produrre un output idoneo, sia sotto il punto di vista del codice HTTP che sotto il punto di vista del messaggio dato all'utente.
Per quanto riguarda il front-end invece, vi sono state più problematiche, dovute anche all'inesperienza con il framework.
Il primo problema emerso è stata la necessità di integrare *JQuery* all'interno dell'applicativo, in quanto richiesto da *Semantic UI* per il corretto funzionamento delle componenti. Tale problematica è emersa non solo all'interno dei componenti dell'applicativo ma anche in fase di testing, fallendo qualunque test che coinvolgesse il componente principale App. Per risolverlo, è stato necessario esportare globalmente *$* e *jQuery* da *JQuery* e richiamare il file di *Semantic UI*.
Un altro problema è stato causato dal modo asincrono in cui *Vue.js* propaga alcune modifiche e dalla presenza di metodi asincroni, quali le chiamate HTTP, con cui veniva contattato il back-end. Per quanto riguarda il primo aspetto, inizialmente, in fase di test, anche settando i dati o le props di un componente, esse non riflettevano il valore atteso in fase di asserzione, facendo fallire i test. Per rimediarvi, si è dovuto fare ampio uso del costrutto *async-await*, marchiando come async l'intera funzione e poi aggiungendo la keyword await al metodo critico in cui venivano cambiati i valori delle variabili. Per il secondo invece si è fatto ricorso al package *flush-promises*. Infatti il problema era originato dal fatto che, anche se mocked, i metodi di *axios* restituivano una promise che doveva essere gestita e il cui esito era fondamentale per valutare la corretta esecuzione del codice. Sebbene fosse possibile usare anche in questi casi il costrutto async-await,

è parsa una scelta più elegante e leggibile utilizzare un package predisposto esattamente per questa funzionalità.

Altri problemi si sono avuti nella scelta e nella relativa impostazione dei framework per gli e2e tests. In particolare sono intervenute una serie di difficoltà che hanno contribuito alla costituzione della configurazione attuale. Infatti, in prima battuta si era scelto di utilizzare *Nightwatch* come framework per la stesura degli e2e tests, sostituito nel giro di poco tempo con l'attualmente utilizzato *Cypress* a causa dell'intervenuta difficoltà di integrazione con *Cucumber*.
Questa integrazione è stata resa possibile grazie all'utilizzo di una dipendenza, denominata *cypress-cucumber-preprocessor*, capace di pre-processare i feature file, collegando i vari scenari alle loro implementazioni concrete. In realtà è stato richiesto uno sforzo maggiore, rispetto a quanto detto sino ad adesso, poiché è risultato necessario integrare il tutto tramite l'utilizzo di webpack, specificando nel relativo file di configurazione le modalità di applicazione del cucumber-preprocessor sopra citato.
Un ultimo punto che merita di nota, in relazione alle configurazioni per gli e2e tests, è stato il tentativo di conversione dell'intero progetto Javascript, alla relativa versione staticamente tipizzata, *Typescript*. Tale tentativo non è andato a buon fine a causa di problematiche di integrazione dei componenti, particolarmente rilevanti in ambito degli e2e tests, ed ha reso necessaria un'operazione di rollback.

4 Aspetti interessanti

Grazie agli studi effettuati, è stata acquisita una notevole maturità in merito ai framework utilizzati, che ha permesso la riuscita nello sviluppo del progetto, sfruttando tutte le tecnologie prefissate.

Non sono mancati neanche alcuni tentativi fallimentari, che nonostante abbiano, talvolta, obbligato ad un cambio radicale di direzione, hanno sicuramente contribuito alla realizzazione delle applicazioni così come si trovano allo stato attuale.
Passaggi particolarmente interessanti, per quanto riguarda il back-end, concernono la modalità di gestione delle eccezioni, la gestione di teammates e skills in carico al service layer e la gestione delle *race conditions* e delle transazioni a livello di database.

Il primo, piuttosto rilevante in ambito di testing, è risultato necessario per ottenere il controllo completo degli stati in cui poteva trovarsi l'applicazione, gestendo eventuali errori e lanciando eccezioni personalizzate. Tutto ciò si ritrova ampiamente all'interno della classe *TeammateRestControllerTest*, ospitante gli unit tests per il controller dei teammates. In questo frangente, seguendo il pattern della *dependency injection*, viene iniettato nella classe un oggetto di tipo *TeammateRestControllerExceptionHandler*. In questo modo è possibile specifi-

care quali eccezioni intercettare, per poi rilanciare eccezioni personalizzate ed asserire sul comportamento del controller fornita in input qualsiasi combinazione di dati.
Il secondo invece, seppur non strettamente necessario, è stato fondamentale per ottenere un risultato finale più corretto e funzionale. Per quanto riguarda le skills infatti, al fine di evitare l'esistenza della stessa skill scritta con formato diverso, si è deciso di non inserire ogni nuova skill immediatamente ed invece effettuare prima un controllo all'interno del database per verificare la presenza di una skill il cui nome corrispondesse. Quindi si sono impostati i test in modo tale per cui, all'inserimento di una skill, se con un nome nuovo, questa venisse aggiunta al database e restituita, mentre venisse restituita la skill già precedentemente salvata altrimenti.
Per quanto riguarda i teammate, e in particolare la duplicazione delle mail, il problema è stato affrontato in modo analogo, producendo però un output diverso. L'idea alla base è stata quella di prevenire la duplicazione delle mail, prevenendo l'inserimento di un teammate nel caso di una mail già esistente e prevenendo allo stesso modo l'aggiornamento di un teammate nel caso la mail aggiunta in fase di update fosse già appartenente ad un altro teammate. Per realizzare questo si è quindi dovuto testare che, al verificarsi delle condizioni sopra previste, venisse prodotta l'eccezione corretta, sia come classe che come messaggio, in modo tale che poi il *TeammateRestControllerExceptionHandler* potesse intercettare le eccezioni previste e fornire l'output corretto del servizio per questi casi.

Infine, particolare attenzione ha richiesto l'analisi delle transazioni e delle possibili race conditions presenti all'interno dell'applicativo. In particolare, sono state individuate possibili situazioni critiche in 3 occasioni: inserimento di un teammate, aggiornamento di un teammate e inserimento di una skill. Tali *race conditions* derivavano, in tutti e 3 i casi, dalla necessità di mantenere degli attributi unici. A tal fine era quindi stata introdotta una verifica nei metodi, verifica che tuttavia non aveva un lock e che quindi poteva essere superata da più threads contemporaneamente. Per ovviare a ciò si sono quindi definite delle politiche di gestione. Per quanto riguarda l'inserimento delle skill si è scelto di restituire, in caso di violazione dell'unicità del nome della skill, la skill già persistita, restituendo quindi la stessa skill a tutti gli utenti partecipanti all'inserimento parallelo. Per quanto riguarda l'inserimento e l'aggiornamento di un teammate, si è invece scelto di intercettare le eccezioni generate a livello di database al momento della violazione dell'unicità della mail, in caso di inserimento o aggiornamento contemporaneo di teammates con stessa mail, e restituire un messaggio di errore, comunicando quindi l'esistenza della mail utilizzata. In questa ottica è stato importante definire le transazioni e fare uso dell'annotazione *@Transactional* di Spring Boot per annotare i metodi di inserimento e aggiornamento di un teammate. Con tale annotazione infatti, i metodi annotati vengono trattati come un'unica transazione, permettendo quindi di effettuare il rollback di tutte le operazioni avvenute all'interno della transazione nel caso di fallimento di una di esse. Questo ha permesso di effettuare un rollback automatizzato delle skill inserite

prima dell'inserimento del teammate, nel caso di un fallimento dovuto al verificarsi di un inserimento parallelo di teammates con stessa mail. Tale funzionalità di Spring Boot è stata utilizzata anche nella cancellazione di un teammate. Infatti, durante la rimozioni di un teammate, viene effettuata anche la cancellazione delle skill orfane. Tuttavia, essendo tale operazione basata sul reperimento della lista di teammates avente una determinata skill, la mancata definizione di una transazione produceva un errore dovuto all'impossibilità di inizializzare tale lista in maniera *lazy*, errore derivante dal reperimento di dati all'interno di un PersistenceContext chiuso. Annotando quindi il metodo *deleteTeammate* della classe *TeammateService* con l'annotazione *@Transactional*, si è riusciti a mantenere il *Persistence Context* aperto fino al termine della transazione e, di conseguenza, fino al termine della cancellazione delle skills orfane.

Per quanto riguarda il front-end invece, di particolare interesse è stata la scrittura dei test riguardanti il design. A differenza del back-end infatti, in cui era stata testata solo la logica, nel front-end è emersa la necessità di verificare che tutte le componenti ed i campi di input rispettassero le varie regole e, nel caso delle select, avessero le opzioni previste. È quindi possibile notare come, nei vari file di unit tests, il numero di questi test di stile sia molto elevato. È altresí bisognoso dire che questi test non sono tutti indispensabili al fine del funzionamento del programma, ma comunque richiesti al fine di garantire il mantenimento del design costruito, inteso come struttura html dei vari componenti della singola pagina.
Un'analisi accurata è stata anche necessaria al fine di testare il componente *TagMultiselect*. Qui infatti è stato necessario capire quali fossero gli eventi generati dal componente terzo inserito *vue-multiselect* nel caso di aggiunta di una skill già esistente e nel caso di rimozione, in modo tale da poter verificare che le skills del teammate venissero effettivamente rimosse quando rimosse dalla select. Per fare questo ci siamo serviti di un plugin, *Vue.js devtools* disponibile per vari browsers, in cui è stato possibile osservare tutti gli eventi generati nell'applicativo. Questo ci ha permesso di scoprire che, in fase di rimozione, il componente realizza in realtà un input dei campi rimasti ed è quindi stato possibile replicare il comportamento in fase di test per verificare che le skills rimosse, venissero rimosse anche dal teammate.

Infine, un ultimo aspetto meritevole di nota è la parte di logica realizzata per la validazione di un teammate in caso di inserimento o modifica. In questo contesto è importante poter specificare un insieme di regole, possibilmente distinte per ogni campo da validare, da considerare in seguito al momento dell'eventuale salvataggio delle informazioni inserite. Per ovviare a questa necessità è stato creato, a livello di "configurazione", un array di regole, specificando per ogni campo da validare un'espressione regolare.
Subito prima del salvataggio, i dati del teammate sono sottoposti ad un controllo, svolto nell'apposito metodo teammateIsValid, dove ciclando sulle regex precedentemente definite, si procede con il controllo dell'effettivo soddisfacimento. In

caso di fallimento di una o più regole, vengono presentati graficamente i relativi errori, bloccando il corrente salvataggio e stimolando l'utente a correggere le eventuali mancanze.
Sempre in relazione al tema di validazione dei campi, risulta interessante il metodo submitDisabled, individuabile sotto al blocco computed all'interno del file *App.vue*. "Computed" rappresenta una keyword propria di *Vue.js*, che fornisce la possibilità di specificare funzioni automaticamente calcolate che restituiscono un valore in relazione allo stato attuale degli oggetti di tipo data propri di una *vue instance*.
Nel nostro ambito tale funzionalità viene utilizzata per verificare il riempimento dei campi obbligatori del form, impossibilitando la pressione del tasto submit se non completamente riempiti.