POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Realizzazione full stack di un social network con Ruby on Rails, Angular, Xamarin e servizi cloud

Relatore
Prof. Giovanni Malnati

Candidato
Marco Sanfilippo Frittola
Matricola 229996

A.A. 2016/2017 - Ottobre 2017

A mio Padre, a mia Madre…

Grazie.

"Your most unhappy customers are your greatest source of learning…"
Bill Gates

Indice

1. Introduzione

Il presente lavoro di Tesi è stato redatto per il conseguimento del titolo di Laurea Magistrale in Ingegneria Informatica presso il Politecnico di Torino. Si articola in sette Capitoli e descrive la realizzazione di un social network, idea di business di una startup sportiva. La questione principale riguarda la possibilità di dare vita ad una buona idea come prodotto software, nel più breve tempo possibile e con il minor consumo di risorse. La realizzazione deve essere completa, ossia deve consistere nella creazione di un'applicazione web, due applicazioni per cellulare (Android e iOS) e l'intera architettura server per il deployment del backend. Tutto deve essere strutturato sin dal principio per poter essere facilmente mantenuto nel tempo, è necessario realizzare una soluzione software che possa fronteggiare senza grosse difficoltà un numero di utenti molto ampio, senza causare sprechi di alcun tipo nelle risorse utilizzate.

Il progresso tecnologico migliora di giorno in giorno strumenti e servizi già esistenti e porta alla realizzazione di nuovi, un tempo impensabili. È ormai abitudine quotidiana utilizzare servizi online per svolgere un gran numero di azioni, da quelle personali come pagamenti, acquisti o prenotazioni, a ricerche e recensioni di vario tipo o utilizzo di servizi multimediali. Siamo letteralmente invasi da migliaia e migliaia di applicazioni volte ad offrire una vastità di servizi differenti, attraverso app stand-alone o applicazioni web, di messaggistica, giochi e altro. Queste nascono spesso in tre ambienti differenti: "smanettoni", startup e aziende già avviate. La prima categoria racchiude coloro che hanno una idea "geniale" per un qualche servizio o funzionalità, create spesso come applicazioni mobile per monetizzare e poi lasciate sui market a morire. Le aziende ben consolidate invece si affacciano ad un modello di business differente dal solito, per facilitare la fornitura dei servizi ai loro clienti attraverso strumenti sempre più d'uso quotidiano, come pc e smartphone. L'idea in sé è già presente e avviata, magari viene migliorata la fruizione dei servizi attraverso queste nuove tecnologie. Il caso delle startup è una via di mezzo, un'idea geniale che vuole essere utilizzata come base aziendale per costruire un prodotto da lanciare sul mercato. Spesso però ci si ritrova ad avere poche risorse economiche ed umane, pur trattandosi di un'idea brillante non è facile trovare investitori disposti a coprire le spese e i rischi del caso, per cui è necessario mettere in piedi un sistema di sviluppo efficiente ed efficace sia in termini di tempo che di prodotto finale. Obbiettivo del lavoro di tesi svolto e descritto nel presente documento è quello di far arrivare sul mercato una buona soluzione software figlia di un'idea di business brillante, con i minimi tempi di sviluppo e con risorse molto limitate.

1.2 Startup

"Startup" è il termine utilizzato per identificare una nuova impresa nelle forme di un'organizzazione temporanea o una società di capitali. Tali aziende sono in cerca di soluzioni organizzative e strategiche che siano ripetibili e possano crescere indefinitamente. L'avvio di un'attività imprenditoriale non scalabile (come l'apertura di un ristorante) non coincide dunque con la creazione di una startup ma di una società tradizionale. Negli ultimi anni il termine startup è diventato inflazionatissimo, forse sulla scia delle esperienze americane della Silicon Valley e di New York (come Mark Zuckerberg e altri), in alcuni paesi è cresciuto a dismisura il numero di giovani che decidono di non aspettare di trovare un lavoro ma di crearselo da soli. Una idea innovativa spesso è accompagnata dalla creazione di una startup, società solitamente con un potenziale occupazionale elevato, ma i problemi e gli errori commessi sono non pochi, soprattutto

perché si confonde il fine ultimo di queste: non si deve creare una startup con l'obiettivo di diventare ricchi, bisogna creare una startup con l'obiettivo di portare valore, bisogna creare ricchezza. La vita di una startup può essere suddivisa in varie fasi: discovery, validation, efficiency, growth. In ognuna di queste sono presenti vari ostacoli, problematiche comuni sono la creazione di un team all'altezza per il prodotto pensato, accumulo e gestione delle risorse finanziarie, messa a punto di un MVP (Minimum Viable Product) e lo sbarco sul mercato, oltre che le difficoltà nel trovare i primi clienti e le questioni legali. Seconda e terza fase riguardano la stabilizzazione vera e propria, in termini di risorse finanziarie, staff e modello di business. Le startup sono più mature nell'ultima fase, in cui hanno già ricevuto alcuni finanziamenti e la difficoltà consiste nel riuscire a espandersi a livello nazionale e internazionale, ossia a "scalare".

Gli ostacoli quindi sono rappresentati dai tempi, dalle strategie organizzative, dal livello di esperienza degli *startupper* e dai tipi di contratti sottoscritti con fornitori e clienti. Un dato di fatto sicuramente negativo è scoprire che la maggior parte delle startup che falliscono non arrivano neppure alla seconda fase, sia per problemi di concorrenza (spesso si copiano prodotti già esistenti), di pianificazione (medio e lungo periodo), rischio e incapacità di gestione dei possibili fallimenti. Un ruolo non marginale è giocato dal prodotto offerto, sia in chiave di successo iniziale che di crescita a lungo termine. È indispensabile soddisfare un bisogno del mercato, non basta una tecnologia efficiente o un staff preparato, ma un modello di business scalabile che risponda a una precisa esigenza. È necessario avere il team giusto, regola d'oro è che il team di partenza della startup riesca a essere autonomo nel poter realizzare il prodotto. Inoltre, il prodotto deve essere *user-friendly*, piacevole e semplice da usare. Marketing e target sono aspetti non pienamente considerati all'inizio, bisogna conoscere le caratteristiche quantitative e qualitative della propria *customer base*, tenendo conto dei feedback, altra regola fondamentale per il successo. Infine, è bene realizzare il prodotto nel momento giusto: se un prodotto viene realizzato troppo presto, i potenziali consumatori potrebbero riconoscerne l'utilità, ma voltare le spalle e rivolgere lo sguardo altrove, perché non esiste ancora un bisogno chiaro nel mercato; se il prodotto arriva troppo tardi, si è persa un'opportunità di business, arrivare al momento giusto è fondamentale.

1.3 Social Network

Una "rete sociale" per definizione consiste in un qualsiasi gruppo di individui connessi tra loro da diversi legami sociali. Tali legami possono essere di varia natura, come rapporti di lavoro, familiari, di amicizia, casuali o altro. Ciò sembra essere ben distante dal significato attribuito oggigiorno con la diffusione del web, infatti il termine *social network* ha subito alterazioni di semantica che hanno generato non poche ambiguità, pur essendo ormai d'uso comune. La rete sociale è infatti, storicamente e in primo luogo, una rete fisica: comunità di lavoratori, sportive, religiose, educative, ecc. Il termine corretto per identificare il "social networking" è invece "social media". I media sociali sono tecnologie informatizzate che facilitano la creazione e la condivisione di informazioni, idee, interessi di carriera e altre forme di espressione tramite comunità e reti (sociali) virtuali. I social media utilizzano tecnologie basate su web, computer desktop e tecnologie mobili (smartphone e tablet) per creare piattaforme interattive; attraverso queste gli individui, le comunità e le organizzazioni possono condividere, discutere o modificare contenuti generati dagli utenti. Ciò introduce cambiamenti notevoli e pervasivi nella comunicazione, caratteristiche importantissime di questi nuovi canali sociali sono l'interattività, l'usabilità, l'immediatezza e la permanenza delle informazioni: possono contribuire a migliorare il senso delle connessioni degli individui con le comunità, sia reali che online, possono essere un efficace strumento di comunicazione e di marketing per aziende, imprenditori, organizzazioni no-profit, poiché offrono un meccanismo per il raggiungimento di un gran numero di utenti sempre attivi (always on).

Si tratta di applicazioni interattive basate sul Web 2.0, il cui legame tra le parti è dato dai contenuti generati dagli stessi utenti, come i post di testo o i commenti, foto digitali e video, *social reactions* e vari dati generati nelle interazioni online. Gli utenti creano profili specifici del servizio per il sito o l'applicazione e all'interno della rete virtuale si creano delle relazioni sociali che si espandono con il collegamento di un profilo utente a quelli di altri individui o gruppi, in forma monodirezionale (follower, come in Twitter) o bidirezionale (amicizie, come in Facebook). Più in dettaglio, ogni social network ha una serie di caratteristiche che lo rendono unico e lo distinguono dagli altri, ad esempio Youtube come piattaforma per i video, Facebook con i profili utenti, chat, album e (tanto) altro, Twitter con i semplici tweet (molto simili ai post Facebook ma con dimensione limitata), Instagram come condivisione di foto o video di breve durata, Linkedin come punto di incontro tra offerta e domanda di lavoro, ecc. In generale i requisiti funzionali di un social network differiscono in base a quanto offerto, ma si ritrovano elementi comuni come profili e gruppi di utenti, amici o followers, elementi preferiti o raccomandati, rating e valutazioni, recensioni e tag, notifiche, chat e, sempre più presenti, servizi geolocalizzati di ricerca e servizi real-time. Alcuni social network offrono servizi utilizzando un modello basato su crediti virtuali, una sorta di moneta che può essere spesa all'interno del social stesso, spesso fallimentare o "auto restrittivo" poiché a pagamento per servizi di svago. La funzionalità di ricerca è l'elemento chiave per la diffusione del social, basti pensare che in Facebook la ricerca di profili utente, corrispondenti a profili reali della persona, è stata la base del suo successo. Questa funzionalità può essere basata su vari attributi, come nome o tipologia della risorsa, tag o altro. Un social network diventa sempre più appetibile man mano che l'evolversi degli eventi e azioni compiute da un utente è reso esplicito agli altri membri: un resoconto delle azioni salienti dell'utente, ad esempio un elenco dei suoi contenuti preferiti o dei rating che ha assegnato di recente, scambio di mi piace o commenti, rende gli altri utenti "curiosi" delle azioni altrui, il che scatena una consultazione costante delle applicazioni social. Non meno importante a tal punto è il meccanismo di notifiche, meglio se *real time*, non invasivo e soprattutto personalizzabile nei contenuti, come forma di sottoscrizione ad uno "stream" di interessi. Social network basati sul ranking dei contenuti, in base alle preferenze dell'utente, aiutano a promuovere le risorse migliori in modo naturale. È chiaro che lo spam è sempre un elemento da non trascurare, ma un buon social deve anche disporre di un sistema di segnalazione circa i contenuti ritenuti non appropriati dalla comunità sociale. Al rating si affiancano spesso i tag, sono parole chiave che vengono associate a un contenuto, rendendone possibile la classificazione e la ricerca. I tag possono essere scelti dagli utenti in modo informale e costituiscono una categorizzazione alternativa di un contenuto. Non a caso la diffusione di informazioni è sempre accompagnata da *hashtag* popolari che ne faciliteranno la ricerca tramite tag. Tutto torna. Infine, la possibilità di "postare" contenuti in bacheca, come testo, foto o video, commentare tali risorse anche tramite *emoji* (evoluzione pittografica delle vecchie *emoticon*, espressione simbolica delle emozioni realizzata tramite determinate combinazioni di caratteri) o gli hashtag, rende il social network piacevole da usare, oltre che utile.

1.4 L'idea

Il documento descrive una soluzione ragionevolmente buona per realizzare la volontà di una startup, la cui idea di business è un sistema di recensioni sportive, ricerca geolocalizzata di risorse e condivisione di contenuti, volta a facilitare l'accesso alla pratica sportiva agli utenti. Inoltre, vuole promuovere le categorie professionali come istruttori e nutrizionisti o entità sportive come centri, organizzazioni, società, ecc. L'insieme dei servizi viene offerto attraverso applicazioni web e mobile. Secondo quanto descritto in precedenza, la startup si trova attualmente nella fase di *discovery*, per cui si rende necessario affrontare i problemi di costituzione della stessa, creazione di un team all'altezza per la realizzare un MVP (Minimum Viable Product), accumulo e gestione delle

risorse finanziarie e lo sbarco sul mercato, oltre che trovare i primi clienti e le questioni legali. Il documento focalizzerà ovviamente soltanto la messa a punto di un MVP e, in conclusione, saranno affrontati gli aspetti legati al ciclo di vita della startup stessa rispetto al prodotto e al team, in termini di vantaggi e svantaggi del modello scelto, la sua sostenibilità nel tempo, evoluzione e crescita.

Prima di trattare lo stack tecnologico scelto, è bene focalizzare in dettaglio i requisiti funzionali del social network in questione, per poter comprendere i motivi alla base di certe scelte. Un sistema di recensioni in generale necessita della costituzione di un profilo utente, meglio se reale e confermato con un certo meccanismo (verifica email, sms o altro ancora); la presenza di utenti reali è un vantaggio non trascurabile circa la veridicità di alcune informazioni e sulla credibilità del social in generale, oltre che una buon motivo di popolarità per gli utenti che lo usano. Sono inoltre presenti i profili delle risorse, ossia gli elementi da recensire che variano in base alla natura del servizio stesso, come una struttura sportiva, un negozio, un soggetto, ecc. e sono associati ad un amministratore della pagina. Si parla di social network e non di un generico sito web o applicazione perché a tutti gli utenti è offerta la possibilità di creare dinamicamente dei contenuti in modo non vincolato alle semplici recensioni. In altre parole ogni membro ha la possibilità di condividere contenuti testuali, foto e video; non solo, queste informazioni possono essere associate ad una risorsa recensita (un luogo, una struttura o altro) e dunque essere segnalati come socialmente utili o non appropriati, attraverso un sistema simile ai "mi piace" dei vari social ben noti. A loro volta, attraverso le risorse recensite si possono condividere informazioni di vario tipo circa le loro attività e gli amministratori delle pagine avranno la possibilità di inserire tutte le informazioni possibili per personalizzare e dettagliare i profili. Tali informazioni saranno oggetto di ricerca e costituiranno il contenuto della pagina principale del social network, secondo un algoritmo basato su popolarità ed età dell'informazione. La ricerca delle risorse può essere geolocalizzata per posizione corrente o a scelta su una mappa (ad esempio un'altra città), per categoria (struttura sportiva, istruttore, negozio, ecc.) con filtri relativi alle esigenze degli utenti (ad esempio strutture aperte in determinate fasce orarie, fasce d'età, restrizioni, agevolazioni, ecc.). Nella prima fase di vita una startup solitamente non ha grandi risorse finanziarie né tantomeno investitori. Risorse finanziarie limitate si traducono inequivocabilmente in un team ristretto, oltre che nella scelta di servizi cloud economici, come compromesso costo-funzionalità. Nonostante le scelte per la realizzazione del prodotto siano fortemente vincolate, lo scopo è costruire un buon prodotto da inserire sul mercato per poter acquisire da subito visibilità. Ovviamente il tutto è stato pensato in grande sin dal principio, per evitare che una progettazione errata del sistema porti a dover riscrivere l'intera architettura applicativa, in caso di necessità legate a fattori improvvisi come una rapida espansione dell'utenza. Pensare in grande vuol dire eliminare i vincoli dell'approccio monolitico sin da subito. Le scelte riguardano un sistema di backend, sia per la gestione della persistenza dei dati che della logica operativa, e due frontend, web e mobile, come interfacce per la fruizione dei servizi offerti. In dettaglio, la persistenza delle informazioni sarà gestita da un insieme di DBMS relazionali, quali PostgreSQL e la sua estensione PostGIS per la conservazione delle informazioni con supporto per i dati spaziali, altri non relazionali, come Redis e MongoDB per gestire il caching e le informazioni non omogenee, ossia non descrivibili da schemi prefissati. Questi dati saranno manipolati attraverso un motore per la logica di business ideato con il framework Ruby on Rails, di sua natura basato sul pattern MVC. Come sarà descritto nelle apposite sezioni, Rails è abbastanza buono per applicazioni web dalla complessità contenuta ma, al crescere della complessità applicativa, le View generate server-side rischiano di sovraccaricare il backend in computazioni non sempre necessarie. Per questo motivo la scelta sarà quella di spostare totalmente la generazione delle viste sui client web e mobile. Il backend Rails esporrà delle API costruite con la filosofia REST e formato JSON per i dati, molto leggero e più flessibile rispetto all'XML, che saranno "consumate" dai client attraverso i frontend web e mobile; si tratta di buon scelte poiché i meccanismi sono nativi e ben supportati.

Per quanto riguarda i frontend, in ambito mobile si utilizzerà Xamarin per Android e iOS, in modo da sfruttare i vantaggi della programmazione mobile cross-platform, un buon compromesso in termini di risorse umane necessarie e prodotto finale. Inizialmente il modello di business della startup era prettamente mobile, ho deciso di estenderlo suggerendo anche una interfaccia web, facilmente accessibile rispetto a un'applicazione da scaricare e installare, per la quale è stato utilizzato il framework Angular (versione 5), che sfrutta proprio lo stesso backend ma con un set di API simili. Si tratta di un framework altamente scalabile e componibile, che consente di spostare totalmente la complessità delle viste sul client, ormai dotati di capacità computazionale non marginale. I vantaggi dell'interfaccia web sono anche da ricercare soprattutto nell'amministrazione dei profili delle risorse offerte attraverso il servizio, la cui gestione sarebbe davvero scomoda attraverso un'applicazione mobile, ma non è comunque esclusa in futuro un supporto di questo tipo. Il set di API sarà diversificato tra ambiente mobile e web perché vi possono essere differenti funzionalità, scenari di utilizzo e di sicurezza. Per la messa in produzione del prodotto software si utilizzerà un'architettura di backend basata su container ospitati in una infrastruttura esterna e non di proprietà, ossia un cloud piuttosto che un personal datacenter, sfruttando la piattaforma Docker. Il motivo è evidente, la startup in questa fase non ha risorse a sufficienza per poter avere un proprio datacenter e comunque si tratterebbe di un investimento rischioso, dato che il numero di utenti allo stato iniziale è pari a zero. Grazie alla tecnologia dei container, come sarà spiegato nel penultimo Capitolo del documento, sarà possibile costruire un backend con un buon grado di scalabilità, sicuramente necessaria in futuro. Il design dell'architettura si orienterà verso il modello a microservizi, per cui sarà possibile replicare soltanto le parti che lo necessitano. Queste scelte sono frutto di una progettazione accurata che vuole soddisfare la volontà della startup per creare un prodotto degno di un mercato ancora senza competitors.

2. Applicazioni web: lo stato dell'arte

In ambito informatico per applicazione web (web application o web app) si intende un insieme di servizi Internet usufruibili attraverso il Web per mezzo di una rete, come ad esempio una Intranet o attraverso la rete Internet, la cui architettura tipica è di tipo Client-Server. Tali servizi sono residenti su un server applicativo e l'utente accede alle funzionalità offerte dall'applicazione attraverso un client, normalmente un browser web appoggiandosi ai consueti protocolli di rete: invia le richieste tramite il protocollo applicativo HTTP(S) al web server che, da una parte interpreta e gestisce le interrogazioni al DBMS, dall'altra genera il risultato in un output diretto allo stesso browser; a sua volta il browser lo interpreta e lo restituisce all'utente sotto forma di pagina web visuale. Un'applicazione web si caratterizza essenzialmente per il trasferimento di informazioni da una parte all'altra, ovvero dal frontend al backend e viceversa, con aggiunta di eventuali elaborazioni lato server. In realtà l'architettura serve side può essere ben più complessa, soprattutto per grandi applicazioni che necessitano di un insieme di server di vario tipo: server applicativi, per lo storage, per il caching, gestione del meccanismo publish/subscribe, broker, mail server, ecc. Queste interazioni sono basate su una sintassi applicativa come REST o SOAP (obsoleto sì, ma ancora utilizzato). A loro volta queste "elaborazioni" possono coinvolgere servizi interni e/o esterni più o meno complessi, detti *web services*. Quando le prime applicazioni Client-Server (non basate sul web) cominciarono ad essere diffuse nel panorama informatico, richiedevano che per ognuna di essa fosse installata la parte client direttamente sul calcolatore di ciascun utente. Qualunque aggiornamento, patch o modifica dell'applicativo lato server (server-side) si trasformava nella esigenza di installare le versioni aggiornate su tutte le workstation degli utenti per l'applicativo stesso. Di contro, le applicazioni web generano il loro output in modo dinamico, pilotando un frontend composto da documenti web, destinati quindi ad un semplice web browser. L'opportunità di aggiornare ed evolvere a costo ridotto il proprio applicativo, senza essere costretti a distribuire numerosi aggiornamenti per i vari client, ha reso la soluzione piuttosto popolare per molti prodotti software. Questo modello applicativo è stato utilizzato a partire dalla fine degli anni novanta con la diffusione di Internet, come semplici siti web composti da documenti navigabili con collegamenti ipertestuali. Fu proprio l'impossibilità da parte dell'utente di interagire con i contenuti che spinse i ricercatori a cercare un'evoluzione: renderlo dinamico, permettendo all'utente di interagire con esso. Pian piano si è passati infatti da un web cosiddetto "statico" (web 1.0) ad uno "dinamico" (web 2.0) e le evoluzioni sono state di vario tipo: dalla grafica semplice ed esclusivamente testuale, ad una composta da immagini e animazioni, widget che eseguono frammenti di codice tramite plugin, ormai deprecati per la scarsa sicurezza che comportano, fino alla possibilità di inserire, modificare o cancellare contenuti, con una complessità architetturale che aumenta soprattutto lato server.

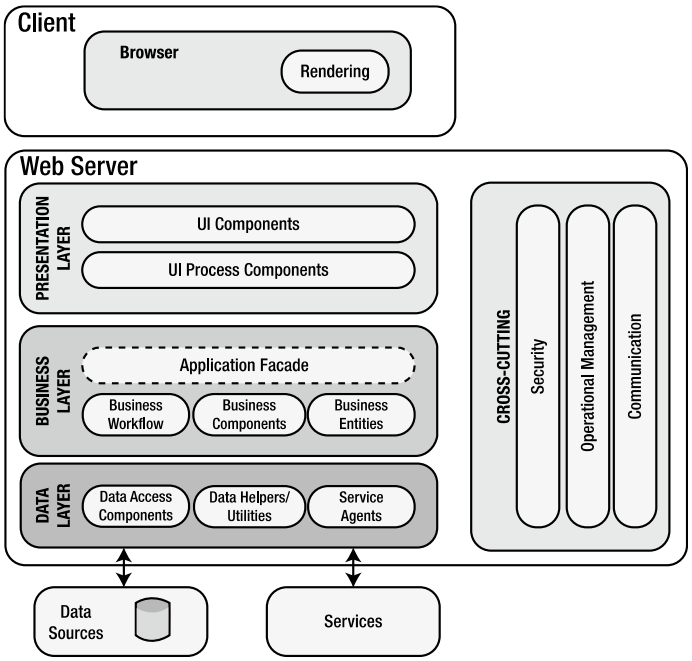


Figura 2.1: Architettura classica di una web app.

La figura 2.1 mostra l'architettura di un'applicazione web con contenuti generati dinamicamente. Tutti i meccanismi si trovano lato server, compreso quello per la generazione delle viste, ma ciò rende il server abbastanza carico e a lungo andare poco performante. Essendo il problema della scalabilità abbastanza importante, nel tempo si è diffuso un modello differente di sviluppo, si è passati dai semplici siti web con contenuti generati dinamicamente, con al più parti interattive nella pagina, alle Single Page Application (SPA), composte da un unico documento web in cui un complesso di file Javascript fa evolvere l'applicazione in base alle interazioni dell'utente. Non è più presente la navigazione tra pagine richieste interamente al server, ogni richiesta adesso consiste nel recupero di alcuni contenuti informativi che andranno a popolare parte della vista completa.

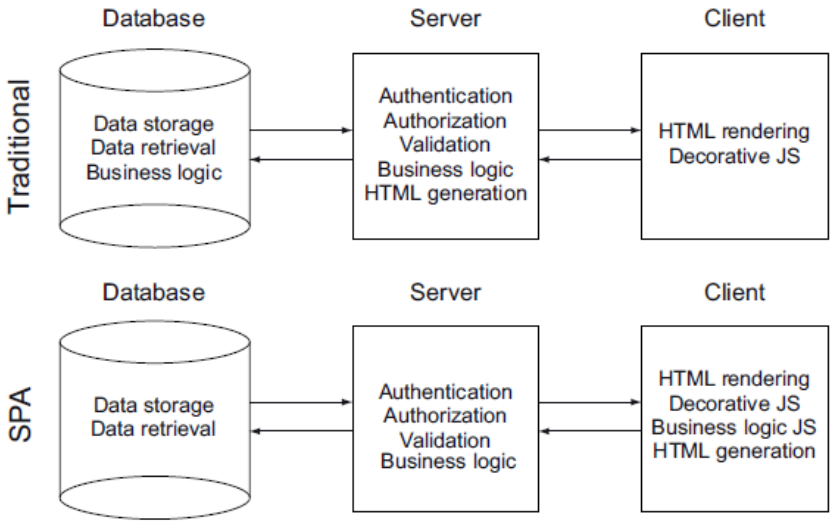


Figura 2.2: Architettura classica di una web app.

Come riporta la figura 2.2, la business logic applicativa migra in parte sul client, mentre la generazione delle viste vi migra totalmente, svincolando il server da tale onere. Ormai Javascript è alla base di potenti framework per le web app, basti pensare ad Angular, React, o Ember. Inoltre, i servizi offerti da un'applicazione web sono spesso forniti anche attraverso le applicazioni per il mobile (mobile app).

Anche se ciò è spesso confuso con la dicitura di "*full stack*", non si intende una web app con frontend sia attraverso browser che app nativa o *cross-platform* per cellulare. Per full stack si intende la capacità "a tutto tondo" di un programmatore nel campo dello sviluppo web, mobile e della gestione sistemistica, nonché le basi e i principi del *networking*, progettazione applicativa e integrazione nel tempo, capacità di progettare e realizzare UI, comprendere i problemi degli utenti e del mercato, ma all'interno di una grande realtà non sarà un unico soggetto ad occuparsene.

Per sua natura una web app può presentarsi con diverse strutture ed organizzazioni logiche, poiché di fatto racchiude in sé sia un modello tecnico che una filosofia di sviluppo. Sono divise in parti logiche dette tier, ognuna delle quali ha un ruolo ben preciso. Le app tradizionali (o stand-alone) semplici sono solitamente formate da un unico tier, che risiede sulla macchina client; se complesse, anche lato client si può avere un modello a più tier per la separazione di dati, logica applicativa e presentazione delle informazioni. Le applicazioni web invece sono di loro natura orientate verso un modello N-tier (architettura *multi-tier*), solitamente a tre parti (*three-tier*), con le varianti del caso. Quest'ultimo viene spesso mappato direttamente sulle parti logiche del modello MVC (Sez. 2.2), nella maggior parte dei casi è infatti possibile identificare logica di presentazione dei dati, logica di *business* e logica di persistenza dei dati, ma non è detto che le tre parti siano tutte collocate nella stessa architettura hardware e software. Sono possibili varie strategie di replicazione delle parti e approcci di costruzione monolitici (tier in un unico progetto applicativo) oppure fisicamente separate, ossia un progetto per l'insieme dati e logica (M, C), un altro progetto per le viste (V), eventualmente estremizzato come nel caso dei microservizi: le varie parti sono installate in macchine differenti (reali, virtuali o in container) e messe in comunicazione attraverso architetture come REST o protocolli proprietari. Ad esempio, l'architettura usata per costruire l'applicazione trattata dal presente documento è stata realizzata utilizzando un modello con backend unico che utilizza i container gestiti con Docker. Grazie al set di API offerto, il backend è svincolato dai frontend, che a loro volta sono diversificati tra app mobile e interfaccia web. Pensando ancor più in grande, in ogni singolo frontend è possibile ritrovare un pattern come MVC o MVVM (Sez. 4.3.1). Questo scenario client-side è molto comune per quei progetti abbastanza complessi in cui il grado di scalabilità richiesto non può essere soddisfatto con i soli meccanismi server-side, ma si aggiungono vantaggi come il caching e la possibilità di operare offline con contenuti da sincronizzare in un secondo momento.

2.1 Applicazioni mobile

Quello del mobile è un mondo in continua evoluzione, sia per l'esponenziale diffusione degli smartphone e dei tablet, che facilitano il compimento di tantissime operazioni prima impensabili, sia perché questo nuovo modo di comunicare ha reso necessario l'introduzione di un nuovo modello di marketing: il Mobile Marketing. Per raggiungere i propri clienti (in essere o potenziali che siano) uno degli strumenti più efficaci e diretti è rappresentato dalle applicazioni. Google, Facebook, Amazon, Twitter, Youtube, Spotify e tanti altri grandi nomi del mercato, ma anche startup, rilasciano versioni delle loro applicazioni per il mobile, come iOS e Android. Com'è chiaro, in ambito di applicazioni web le applicazioni mobile fanno da contorno, come interfaccia aggiuntiva per usufruire un insieme di servizi. Ci sono vari modi di interfacciare il frontend su piattaforma mobile: sviluppo nativo, cross-platform e ibrido.

Un'applicazione nativa è un'applicazione realizzata attraverso la scrittura del codice sorgente pensato appositamente per i diversi sistemi operativi, sviluppata con codice e librerie sia proprietarie che di terze parti. Si installano e si utilizzano direttamente sul dispositivo mobile, proprio come un'applicazione software tradizionale viene eseguita direttamente su un computer desktop, potenzialmente senza la necessità di connettività Internet.

Il vantaggio principale di questa forma di sviluppo è il poter interagire con le caratteristiche hardware e software del device, consentendo ottime prestazioni, sia come maggior velocità di calcolo che di interfacciamento con l'utente (migliore *User eXperience*). Vi è pieno controllo della sensoristica del dispositivo, infatti l'interazione diretta con le API messe a disposizione dal costruttore del sistema operativo garantirà accesso a tutte[1] le funzionalità del dispositivo, oltre a permettere prestazioni ottimali e migliorare sensibilmente l'usabilità. È anche possibile lo sviluppo di meccanismi per il funzionamento offline, a patto che il servizio offerto lo consenta, una migliore sicurezza dei dati e integrazione con backend di vario tipo (locale o remoto). Una delle strade percorribili quando si intende sviluppare un'applicazione mobile è quella di orientarsi verso lo sviluppo nativo, proprio per sfruttare tali vantaggi. Eppure questa tipologia di sviluppo implica conoscenze e competenze specifiche per i vari sistemi operativi. Queste app sono tipicamente scritte con linguaggio nativo della piattaforma, come in Java per dispositivi Android, Object-C o Swift per i dispositivi iOS e C# per Windows Phone.

Mentre un'app nativa è installata fisicamente sul dispositivo dell'utente, una web app per il mobile (sviluppo ibrido) è sostanzialmente un collegamento verso un applicativo remoto, scritto in un linguaggio indipendente dalla piattaforma come HTML5, il cui codice dell'interfaccia utente viene totalmente o parzialmente generato in remoto. Questa soluzione comporta delle importanti conseguenze in termini di funzionamento: il vantaggio principale di una web app mobile consiste nel fatto di non incidere in alcun modo (al più incide in maniera minima) sulle capacità di memoria del dispositivo e sulle sue capacità di calcolo, in quanto il nucleo di elaborazione e generazione dell'interfaccia utente risiede su server remoti, mentre sul client si ha la sola visualizzazione dei contenuti e qualche interazione, tutto ottimizzato in termini di design per essere visibile anche su piccole schermate e che funzionino bene con il touchscreen, attraverso le cosiddette *Web View*, componenti software per il rendering dei contenuti HTML. Tuttavia, per funzionare una web app richiede l'accesso a Internet e le sue prestazioni dipenderanno in modo sensibile dalla velocità del collegamento, lo stile è quello di un vero e proprio sito web con qualche variazione grafica sulla disposizione delle parti. Per la loro natura, queste applicazioni portano con sé limitazioni, visto che non possono facilmente interagire con l'hardware e il software del device. Framework noti sono Cordova, Ionic, Appcelerator Titanium e altri, è sufficiente la conoscenza di HTML, CSS e Javascript.

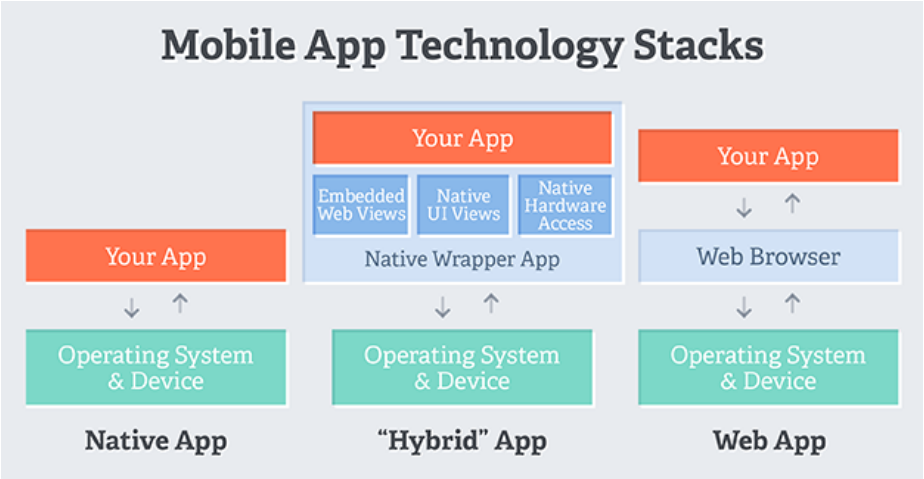


Figura 2.3: Confronto tra sviluppo nativo e web.

[1] Esclusi i casi in cui è necessario avere permessi di root o altro.

Come mostra la figura 2.3, le applicazioni ibride sono una via di mezzo tra nativo e web, infatti è necessario un interprete per accedere alle funzionalità del sistema operativo, comunque limitate. Il motivo è prettamente di sicurezza, si tratta di un codice che arriva dall'esterno e non si sa quali azioni possa effettivamente compiere sul sistema. Questa tecnica di sviluppo ibrido con linguaggio indipendente dalla piattaforma non deve essere confusa con lo sviluppo cross-platform. Infatti, la modalità di sviluppo citata pocanzi è la stessa utilizzata per i normali siti web, in cui la componente web view ha un compito analogo a quello di un browser. I framework per lo sviluppo multipiattaforma sono stati pensati per la scrittura di codice per implementare meccanismi e funzionalità condivisibili tra le varie piattaforme. Più in dettaglio, le API cross-platform vengono mappate sulle funzionalità specifiche del linguaggio madre, ossia sulle API native, perché ad esempio il sistema Android ha un modo diverso di esportare l'accesso alle funzionalità (Wi-Fi, GPS, rubrica e altro) rispetto ad iOS o Windows Mobile. Ovviamente serve un meccanismo che a partire dalla soluzione multipiattaforma consenta l'esecuzione del codice nativo. In tal modo però è possibile accedere alle funzionalità native di ogni singola piattaforma.

Applicazioni ibride e cross-platform hanno una base comune, trovano terreno fertile laddove lo sviluppo immediato è un'esigenza predominante. Ma sono due tecniche differenti con vantaggi e svantaggi differenti, l'ibrido è utile nei casi in cui performance e scalabilità non sono necessarie, ma essendo tali scenari molto ristretti, al più si passa ad una cross-platform come Xamarin (C#), React Native (Javascript e React), Flutter (Dart) e altri. Questi consentono la creazione di applicazioni per più piattaforme utilizzando però un unico linguaggio di programmazione, diverso da ognuno dei linguaggi nativi, a partire da un'unica soluzione software. I vantaggi sono principalmente legati al dover scrivere il codice una sola volta, per poi ricavare la versione dell'applicazione desiderata. Requisiti dell'applicazione e budget di sviluppo sono certamente fattori che influenzano la scelta, che a sua volta incide e non di poco sullo stile dell'applicazione stessa. Lo sviluppo nativo ammette infatti un insieme di vantaggi dal punto di vista grafico, assolutamente da non trascurare dato che per l'utente finale contano tantissimo l'impatto grafico e l'usabilità (UI - User Interface, UX – User eXperience) oltre che l'utilità. La visione dello sviluppatore è diversa da quella dell'utente finale, bisogna progettare e sviluppare un prodotto facile, utile e piacevole da usare, un'applicazione disinstallata causa cattiva esperienza raramente sarà scaricata di nuovo.

2.1.1 Xamarin vs Cordova

Xamarin o Cordova? Quesito ampiamente discusso, ma di per sé errato e secondo alcuni non andrebbe neppure posto. La risposta corretta in realtà non esiste, dipende fortemente dai requisiti applicativi specifici. Brevemente, Apache Cordova è un framework di sviluppo mobile open-source, consente di utilizzare le tecnologie Web standard - HTML5, CSS3 e Javascript per lo sviluppo di applicazioni. Queste vengono eseguite in "wrapper" e sono destinate a ciascuna piattaforma, basandosi su API compatibili con gli standard per accedere alle funzionalità di ciascun dispositivo come sensori, dati, stato di rete, ecc. È consigliata se si desidera distribuire o estendere un'app web su piattaforma mobile, senza doverla implementare in ogni piattaforma nativa.

Xamarin d'altro canto è davvero una piattaforma potente e completa, con esperienze di debugging e toolchain notevolmente migliori. Tuttavia non bisogna lasciarsi ingannare, c'è una curva di apprendimento piuttosto ripida, in particolare per gli sviluppatori che sono nuovi nello sviluppo mobile in generale. Non è necessario imparare solo la toolchain di Xamarin e il linguaggio C#, è inoltre necessario conoscere le piattaforme specifiche (iOS, Android e altri). Potrebbe non essere necessario imparare Java e Objective-C, ma bisogna conoscere quelle API perché in fin dei conti il *binding* è nativo su esse.

Per le applicazioni più semplici, dove il team ha una conoscenza minima degli ambienti mobile, è già confortevole usare HTML e Javascript, soprattutto se il contesto è prettamente dipendente dall'interfaccia e meno dipendente dalle prestazioni, allora Cordova potrebbe essere più che sufficiente. Molte aziende provano prima questa strada perché è certamente più economico (in termini di costo iniziale di investimento, non il costo del ciclo di vita complessivo) e soprattutto allettante per team di sviluppatori web.

Tuttavia è uno scenario abbastanza comune trovare aziende che hanno provato prima di Cordova e lo sostituiscono con una soluzione come Xamarin, passaggio legato a problemi di funzionalità e prestazioni, o perché ad esempio trovano troppo fastidioso il loro mantenimento in applicazioni complesse. Di solito le applicazioni ibride non corrispondono mai all'aspetto di una vera applicazione nativa, perché queste adottano automaticamente l'interfaccia più aggiornata. Il "segreto" di Xamarin è che il codice darà origine ad un'applicazione nativa di ogni piattaforma, tutto ciò che si può fare con un'applicazione nativa, lo si può fare con Xamarin, ma con il vantaggio aggiunto di una base di codice condivisa. Cordova fornisce maggiore abilità a costruire le applicazioni semplici, ma le prestazioni non sono chiaramente il punto forte, è una buona scelta per alcuni progetti facilmente riproducibili come frontend di siti web statici o animati. Xamarin è migliore perché è considerabile come nativo, di certo vi è dell'overhead legato al framework stesso, ma non tanto quanto con i wrapper delle web view, non vanta grandi numeri per le librerie di terze parti, è ancora un framework non ampiamente utilizzato, a differenza di Javascript che è ampiamente diffuso e supportato. Eppure anche Xamarin ha vantaggi non banali, se confrontato con Cordova, oltre al fatto che le applicazioni sono scritte interamente in C# e XAML (per il markup e lo styling). Xamarin offre anche una buona UX, come funzionalità e prestazioni notevolmente migliori. Si compila in codice nativo e consente l'accesso a API native tramite wrapper C#, rendendo le applicazioni di Xamarin, se ben scritte, dall'aspetto e funzionalità completamente nativi, poiché Xamarin è generalmente ritenuto essere il framework più vicino all'ambiente nativo. Ciò che è degno di nota è che Microsoft ha acquisito Xamarin e renderlo open source: tools e codice Xamarin sono ora disponibili gratuitamente, anche se per uso non commerciale. Con Cordova, lo sviluppatore ha bisogno solo di creare un'applicazione che apparirà simile su più piattaforme, può essere un vantaggio enorme se lo stesso UI (User Interface) è richiesto uguale su tutte le piattaforme, mentre perde alcuni dei suoi valori per la condivisione di codice quando l'UI dovrebbe essere diverso tra le piattaforme. Al contrario, uno dei vantaggi principali di Xamarin è la capacità di generare codice riutilizzabile - fino al 90% in termini di business logic, personalizzando la UI per la piattaforma target: è possibile scrivere del codice eseguito solo su una piattaforma e non in altre. Tuttavia, gli stessi livelli di condivisione di codice a Cordova possono essere ottenuti attraverso la potenza dei framework CSS, ma ciò rende pesante la computazione, risultando svantaggioso, ecco perché applicazioni complesse non usano Cordova. Altre caratteristiche come il multithreading differiscono tra le due tecniche, perché in Cordova è possibile usufruire di queste e altre funzionalità solo attraverso plug-in appositi, mentre in Xamarin è, di nuovo, tutto supportato in modo nativo. Quindi, supponendo che le persone su un team di sviluppatori siano fluenti in HTML, CSS e JavaScript, e si ha bisogno di costruire un'applicazione abbastanza semplice, Cordova si dimostra un'opzione migliore. Tuttavia, se è richiesta un'applicazione dalle alte prestazioni, Xamarin è decisamente da preferire. Detto ciò, argomentare ulteriormente la scelta di Xamarin rispetto ad altre sarebbe ripetitivo, nonostante non si tratti né di una scelta perfetta, né di una scelta sbagliata. Ognuno dei framework ha vantaggi e svantaggi, ma Xamarin è quello che più si avvicina ai vantaggi della soluzione nativa, un'alternativa può essere migliore dell'altra a seconda della situazione specifica, dei requisiti applicativi, del mercato e di molte altre variabili in gioco: risorse economiche e tempistiche sono entrambi vincoli stringenti, ma gli attuali requisiti applicativi non necessitano di caratteristiche specifiche della piattaforma, per cui Xamarin è al momento la soluzione ragionevole.

2.2 Il pattern MVC

Il modello MVC è un pattern architetturale alla base delle moderne applicazioni, sia server side che client side, per applicazioni web e stand alone. Anche se originariamente sviluppato per il desktop computing, l'architettura MVC è stata ampiamente adottata per applicazioni Internet nei principali linguaggi di programmazione. Sono stati creati diversi framework web commerciali e non commerciali che impongono il modello. Questi framework variano nelle loro interpretazioni, principalmente nel modo in cui le responsabilità di MVC sono divise tra il client e il server. I primi framework MVC web hanno adottato un approccio a client "sottile" o leggero, che ha collocato quasi l'intera logica di modello, vista e controller sul server. Ciò si riflette ancora nei framework popolari come Django, Rails e ASP.NET MVC. In questo approccio, il client invia richieste al server, un controllore le intercetta e le smista alla logica applicativa per la sua elaborazione, per poi ritornare al client una serie di dati, ossia una pagina web completa e aggiornata come vista. Il complesso di componenti per MVC esiste interamente sul server. Poiché le tecnologie dei client sono maturate, acquisendo capacità computazionale non marginale, sono stati creati framework client side quali Angular, React, Ember, Backbone e JavaScriptMVC, che consentono ad alcune parti del modello MVC di essere parzialmente replicati o spostati sul client, grazie al massivo utilizzo di AJAX, in modo da rendere continuativa la navigazione ed eliminare dal server l'onere della generazione delle viste.

2.2.1 L'approccio stratificato

Quando si progettano applicazioni web, necessariamente si opera in modo stratificato, partendo dallo strato più basso di accesso ai dati fino a quello più alto dell'interfaccia utente. Occorre agire in questo modo perché la complessità del paradigma software monolitico diventa insostenibile nel tempo e al crescere della complessità, del numero di utenti e della logica applicativa, legato a fattori come manutenibilità, scalabilità, e altri. Per tal motivo si passa ad un approccio sempre più distribuito, fino ad arrivare al modello a microservizi, se necessario. Pensando il software in modo stratificato, ogni strato dialoga con gli adiacenti attraverso un'interfaccia, in modo da non legarsi al componente specifico che la implementa. Ciò è importantissimo nelle fasi di sviluppo, debugging e testing, perché apre la strada ad una serie di possibili alternative per ogni strato software: ognuno può essere facilmente sostituito con una versione non finale ma comunque funzionante e valida per testare la logica degli altri strati software. Pur sembrando qualcosa di poco significativo, in realtà è utile per proseguire in parallelo nello sviluppo del software con altri eventuali sviluppatori coinvolti in un progetto. Questa separazione delle responsabilità (separation of concerns) viene in automatico con la modularità della soluzione, ogni parte è fatta (auspicabilmente bene) da altri soggetti e tra di loro non vi è alcun conflitto o dipendenza temporale. Nell'architettura web generica si ha uno stack server side, diviso in *Data Tier* e *Web Tier*. Tutto parte da uno *User Agent* che si interfaccia con le informazioni dell'applicazione attraverso un meccanismo di tipo *Request/Response* basato su HTTP. Tali informazioni sono presenti nel *Data Tier*, solitamente si tratta di dati gestiti da uno o più DBMS. Sebbene sia possibile scrivere un programma che, a partire dalla richiesta di alto livello, acceda alla base dati sottostante, è qualcosa che si vorrebbe evitare, per questo si parla di stack modulare formato da molteplici componenti software.

Il primo livello è il Data Access Layer (DAL) che ha la responsabilità di trasformare le informazioni presenti nella base dati nel formato del linguaggio di programmazione adoperato, dunque fa da ponte tra le due sintassi differenti. È importante notare che la presenza di una base dati relazionale o non relazionale non avrebbe fatto alcuna differenza dal punto di vista del DAL. Nel dettaglio, questo si occupa di trasferire informazioni dal processo applicativo (Java, Ruby, Python o C) al processo del DBMS, attraverso opportuni metodi di memorizzazione e recupero

delle informazioni. Di solito offre anche dei meccanismi di ricerca attraverso predicati che non siano la sola chiave primaria, utili nel trovare un gruppo di informazioni coerenti secondo dei criteri. Normalmente si implementa creando un set di oggetti per ciascuna entità che si vuole gestire, ad esempio la classe viene chiamata con il nome dell'entità e il suffisso "DAO" (Data Access Object), classe senza attributi che, attraverso dei metodi, mette in comunicazione l'ambiente di esecuzione con il DBMS; vale la pena ricordare che gli ORM (Sez. 2.3) inglobano già tale meccanismo.

Al di sopra del DAL vi è il Data Domain Layer (DDL), in cui le informazioni recuperate dallo strato sottostante sono adesso organizzate nella forma di oggetti. In altre parole, tutte le entità che entrano nel problema vengono rappresentate come oggetti del dominio da questo strato. Per capire, nel rappresentare una entità "persona" che ha un gruppo di "amici", si avrà nel database una tabella le cui righe rappresentano la relazione di amicizia attraverso una corrispondenza con la persona data dalla sua chiave primaria, come *foreign key*. Ciò nel mondo ad oggetti può essere rappresentato da una lista di amici all'interno dell'oggetto persona, senza alcuna chiave esterna. Il DDL si appoggerà al DAL per ottenere le informazioni o conservarle, salvando oggetti e interrogando la base dati. Inoltre, se le entità tra di loro hanno delle relazioni, anche gli oggetti saranno in qualche modo collegati dal punto di vista logico, infatti tale DDL fornisce dei metodi per l'acceso alle relazioni.

Il Service Layer (SL) contiene la *Business Logic*, ossia i moduli software procedurali che indicano la logica dell'applicazione, il cosa si fa su questi dati. Ad esempio, un market online avrà azioni come acquisti, pagamenti e altre che saranno implementate da uno o più metodi, ma non è obbligatorio che questo strato di servizio lavori necessariamente con dei dati, può anche avere solo dei metodi. In framework complessi come Rails o Spring ogni azione del controller, invocata quando si riceve una certa richiesta dal client, si compone internamente di uno o più metodi dei vari servizi.

Infine, il Presentation Layer (PL) genera una vista adeguata con cui l'utente può interagire tramite il suo user agent. In realtà questo strato ha due compiti principali: trasformare le richieste HTTP di un utente in un metodo di servizio, recuperato attraverso la corrispondenza con una particolare URL; inoltre, se non accade alcun tipo di errore, legato alle varie situazioni non valide[2] sia da parte dell'utente che del server, tale strato è responsabile della generazione di una o più parti per costruire una risposta adeguata per il client. È bene ricordare che la visione del progettista software è ben diversa da quella dell'utente finale, che userà l'applicazione con una sequenza probabilmente diversa da quanto pensato, per cui bisogna anche implementare una macchina a stati opportuna lato server per il servizio offerto.

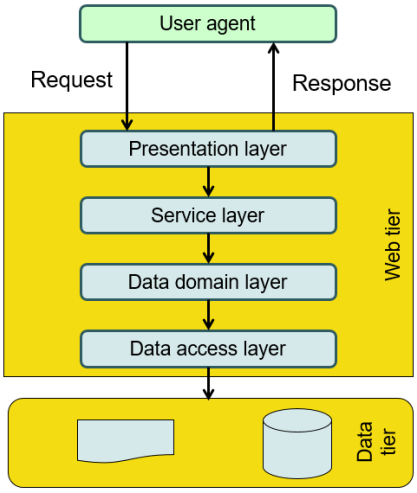


Figura 2.4: Stack server-side con i vari livelli.

[2] Involontarie o maliziose.

Non è intrinsecamente impedita la presenza di comportamenti scorretti, per cui questo strato deve inglobare funzionalità di validazione dei parametri e delle richieste ricevute, sia per sintassi che per semantica (ad esempio, un paramento numerico non può essere un insieme di lettere alfabetiche, così come una data di prenotazione per il futuro non può riferirsi al passato). È bene fare controlli lato client per una migliore user experience, ma non è sufficiente, per cui è necessario rifare lato server almeno gli stessi controlli eseguiti lato client e anche altri più accurati. Fatto ciò, ci si rivolge allo strato di servizio per far eseguire le dovute operazioni e ottenere una risposta, aggiornando di conseguenza la parte visiva ed evidenziando la presenza di errori.

Tutto ciò fa sì che lo strato del Presentation Layer diventi molto complesso, per tal motivo si adopera il modello di tipo MVC (Model-View-Controller), in cui vi sono ben tre responsabilità differenti, pensato per separare ciò che sta dentro da ciò che sta fuori. *Model* è la componente centrale del pattern architetturale, esprime il comportamento dell'applicazione in termini di dominio, indipendentemente dall'interfaccia utente. Gestisce direttamente i dati, la logica e le regole dell'applicazione. L'insieme di dati su cui l'applicazione opera sono le astrazioni del Domain Layer, spesso considerate parte del servizio. In realtà i dati sono del Domain Layer ma le operazioni da fare sui dati (logica operativa) sono stabilite dal servizio, nulla di HTTP. *View* è la rappresentazione visuale del modello o di una sua parte e a parità di modello si possono avere differenti rappresentazioni equivalenti (ad esempio HTML, JSON o XML). Anche qui, ancora nulla di HTTP. *Controller* è la parte che conosce HTTP, sa come intercettare le richieste in ingresso, farne il parsing e formulare le risposte attraverso lo strato di servizio. Le parti interagiscono tra loro nel seguente modo: il modello memorizza i dati recuperati in base ai comandi del controller e li mette a disposizione della vista. Una vista genera un output al client in base alle modifiche apportate al modello, non necessariamente passando per il controller. Un controller può inviare comandi al modello per aggiornarne lo stato, ma anche inviare comandi alla vista associata per modificare la presentazione della visualizzazione del modello.

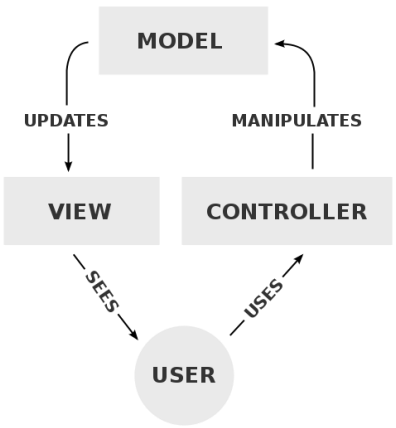


Figura 2.5: Schema operativo del pattern MVC.

Il modello deve essere il punto di partenza nello sviluppo di un progetto, sia web che non, la business logic nasce come conseguenza del modello ossia solo se si ha una chiara visione circa i dati su cui si può operare. Il vantaggio di MVC è la separazione delle responsabilità (separation of concerns), la vista non si deve preoccupare di come certe informazioni verranno recuperate, deve solo rappresentarle e viceversa, il controller, appoggiandosi allo strato di servizio, si occuperà di recuperare le informazioni senza dover provvedere ad una loro rappresentazione, ma passarne una copia alla vista e questa non deve poter modificare i dati del dominio. In Rails il concetto di "concern" è ben supportato, a favore dei vantaggi più evidenti del modello MVC, ossia lo sviluppo con riutilizzo di codice per snellire i modelli, oltre ché avere del codice trasversale e non ripetitivo. Poiché MVC disaccoppia i vari componenti di un'applicazione, gli sviluppatori sono in grado di

lavorare in parallelo su diversi componenti senza interferire e/o bloccarsi l'un l'altro. Ad esempio, una squadra potrebbe essere divisa in sviluppatori tra il frontend e il backend. Gli sviluppatori di backend possono progettare la struttura dei dati e come il client può interagirvi, senza necessità di avere l'interfaccia utente completata. Al contrario, gli sviluppatori frontend sono in grado di progettare e verificare il layout dell'applicazione prima che la struttura dei dati sia disponibile; se necessario, si utilizzano dei *mock*, come l'utilizzo di un server HTTP locale che emette dei file JSON come output di una elaborazione *fake*, simile a quella che produrrebbe il backend.

Pensando un'applicazione in questo modo, il suo sviluppo è facilitato anche grazie alla possibilità di testare le varie parti in modo indipendente tra loro, utilizzando i paradigmi di sviluppo TDD o BDD (Test-Driven Development o Behaviour-Driven Development), in cui rispettivamente la realizzazione del software procede con il voler ottenere un certo risultato specificato dal test, costruendo dei metodi adeguati per il conseguimento dello stesso, piuttosto che descrivere il comportamento di un oggetto o un'azione e costruire metodi di testing per verificarne il comportamento. Sono oggigiorno molto diffusi in ambito AGILE, Rails supporta bene sia il pattern BDD che TDD, il primo ampiamente utilizzato nello sviluppo trattato, ma la potenza di Rails è da ricercare anche negli ORM e ONRM, descritti nella successiva sezione e ampiamente utilizzati nel lavoro svolto.

2.3 ORM e ONRM

L'approccio ad oggetti si presta molto bene a rappresentare sistemi software complessi ma di solito manca di un meccanismo generalizzato per rendere persistenti le informazioni. Le basi di dati, relazionali e non, sono state pensate proprio per fornire un supporto alla persistenza nel tempo delle informazioni e sono tutt'oggi ben consolidate. Per adattare i contenuti di una base dati al modello a oggetti, è richiesto lo sviluppo di grandi quantità di codice da includere in uno o più moduli software, con differenze quasi assenti e dunque ripetitivo. Il meccanismo che consente di superare tali problemi è quello degli ORM (Object to Relational Mapping) per le basi di dati relazionali oppure ONRM (Object to Non-Relational Mapping) per quelle non relazionali; si tratta di uno strato software composto da un insieme di moduli che converte i tipi di dato da un formato ad oggetti ad uno conforme con la base dati in questione e viceversa, introducendo un meccanismo per cui oggetto e riga (o documento, struttura dati, ecc.) si mantengono sincronizzati in specifici momenti, senza doversi preoccupare dei dettagli di accesso alla stessa base dati per la sincronizzazione. La trattazione sarà fatta per le basi dati relazionali, ma si può estendere facilmente anche a quelle non relazionali.

Modellando sistemi software complessi, il programmatore è abituato a pensare ad oggetti, ed è giusto così. Si hanno un sacco di vantaggi legati al dominio ad oggetti, grazie ai quali è possibile esprimere in modo semplice i meccanismi elementari e le operazioni a questi associate. Con le basi dati relazionali siamo vincolati a non poter pensare ad oggetti, ma è necessario descrivere le informazioni in modo più semplice. Punto a favore del modello a oggetti è la possibilità di poterli incapsulare l'uno dentro l'altro con profondità arbitrariamente grande, mentre nel DB la conoscenza dell'oggetto può essere espressa attraverso una o più tabelle, le cui singole righe sono omogenee tra di loro dal punto di vista strutturale. Ogni riga della tabella contiene dati semplici come stringhe (con alcune variazioni sulla lunghezza massima possibile), formati temporali distinti in data, ora e timestamp (data e ora insieme), numeri e blob/clob (array di byte o char, non oggetto di predicati di ricerca). Vi sono anche riferimenti esterni alla tabella stessa, ossia stringhe o numeri che consentono di rappresentare un riferimento alla chiave primaria di altre righe. Ciò aiuta a modellare comportamenti abbastanza utili per la base dati, come ad esempio il *cascading*, ossia propagare una modifica di una entità sulle parti ad essa referenziate, modellare comportamenti e collegamenti

come le relazioni e la navigazione bidirezionale, quest'ultima nativa; è utile anche in fase di eliminazione, ad esempio nel cancellare un record "acquirente" dal DB si cancelleranno tutti i record che rappresentano i suoi "acquisti". In una base dati relazionale sono supportati dei tipi di dato ben più semplici rispetto a quelli offerti dai linguaggi ad oggetti, il vantaggio di avere un modello semplificato è legato alla normalizzazione dei dati, ossia all'interno della base dati l'entità non è replicata ed è conservata nella forma minima che consente di rappresentarla con le sue relazioni. Non vi è alcun tipo di ridondanza, che vuol dire nessuna contraddizione sulla rappresentazione del dato o problemi di coerenza tra le parti in caso di modifiche. Ciò implica inevitabilmente un modello piatto, in cui nel voler recuperare dei dati e rappresentarli in uno o più oggetti è necessario dover forzare i concetti sia della programmazione ad oggetti e dei principi delle basi dati relazionali, per conciliare i due mondi.

Per evitare che i programmatori debbano fare tutto ciò, è stato pensato il meccanismo dell'ORM oppure ONRM, una tecnica di programmazione che favorisce l'integrazione di sistemi software aderenti al paradigma della programmazione orientata agli oggetti con sistemi DBMS, spesso fornito tramite librerie. Il loro compito di base è trasformare un oggetto in memoria in una forma da rendere persistente in modo coerente con lo schema del database, senza distruggere la logica associata ai dati dell'oggetto stesso (relazioni e semantica) e viceversa, ossia recuperare da una o più tabelle le informazioni per rappresentare l'entità sotto forma di oggetto in memoria. Volendo fare un paragone tra i due mondi, il formato oggetti è ben più complesso e non facilmente interfacciabile con il mondo delle basi dati, ad esempio non si lavora con la chiave primaria ma la sua identità è data dal suo indirizzo in memoria. Gli oggetti vivono in grafi, cioè se un oggetto "A" contiene "B" che contiene "C", si crea un grafo delle dipendenze per il contenuto di "B" che allo stesso tempo contiene "C" ma è contenuto da "A". Quel che è presente in memoria è qualcosa di implicito che deve diventare esplicito nella base dati, sostituendo in qualche modo l'indirizzo di memoria con la chiave primaria del dato, tale chiave è qualcosa di assoluto nel contesto della base dati mentre l'indirizzo è qualcosa di effimero che può cambiare nel tempo.

Tutto ciò è molto complesso perché, come intuibile da quanto detto, la semantica del mondo ad oggetti è ben diversa di quella del mondo relazionale. Ad esempio vi è il problema legato ai paradigmi, il concetto di ereditarietà tra gli oggetti usato con il polimorfismo (un mezzo di trasporto più essere un'auto, bici o tram), non è così facile da ottenere come nel mondo ad oggetti, nel mondo delle basi di dati bisogna coinvolgere N tabelle differenti, anche se sono stati proposti alcuni approcci alternativi come quello del "*Single Table Inheritance*", un modo di accomunare informazioni in una tabella e referenziarne una riga nell'entità "figlia". Inoltre, la base dati può essere navigata in entrambi i versi, ad esempio nel caso delle vendite è possibile sapere chi ha fatto un determinato acquisto a partire da questo; viceversa, a partire da un acquirente, è possibile risalire ai suoi acquisti. Per natura sono delle relazioni navigabili in entrambi i versi, forse sarà necessario fare qualche *query* in più in caso di situazioni complesse, ma non è impedita la navigazione. Ciò non è così naturale nel mondo ad oggetti, perché il collegamento tra le parti è dato da un puntatore, che è intrinsecamente unidirezionale: se una classe "C" contiene una stringa "s", l'istanza "c" di "C" sa qual è la sua stringa "s", ma "s" non sa a quale istanza di "C" appartiene, e neppure sa di essere dentro una istanza, per la natura dei puntatori. Inoltre, nel mondo relazionale è possibile avere diversi tipi di relazioni, 1:1, 1:N, N:1, N:M, nel mondo ad oggetti invece è possibile avere relazioni 1:1, al più N:1 se un oggetto è referenziato da più parti. Per colmare la differenza tra le tipologie di relazioni bisognerebbe progettare del codice con puntatori incrociati, che non è semplice né dal punto di vista della programmazione stessa, nel dal punto di vista del contesto, perché spesso ciò va in conflitto con il lavoro svolto dal *garbage collector* (se presente) con i conseguenti rischi del caso, per questi motivi la generazione degli ORM è stato un processo abbastanza complicato. Dunque l'approccio è tale per cui per ogni riga della tabella nella base dati è possibile creare un

oggetto con le informazioni della riga associata e, viceversa, da un oggetto è possibile creare o aggiornare una riga della base dati, a partire dalle informazioni dell'oggetto stesso. Inoltre, mettono in atto un meccanismo tale per cui se l'oggetto cambia anche la riga cambierà, in modo da garantire un certo allineamento. Se questa corrispondenza garantita dall'ORM fosse stata a carico del programmatore sarebbe stata difficile da gestire. In realtà gli ORM non lo garantiscono in modo assoluto, ma entro una serie ben precisa di metodi o comandi, al di fuori dei quali questa corrispondenza è troncata per volontà del programmatore, dunque valida entro certi intervalli specifici.

Con ORM o ONRM allora si ottiene un modello generale per poter modellare e manipolare i dati in modo indipendente e trasparente alla specifica implementazione della base dati sottostante, in modo da poter esprimere le operazioni da compiere su questi in modo più naturale possibile: un approccio concettuale alla modellazione, interrogazione e trasformazione dei dati, che rende trasparenti le operazioni di conversione e rappresentazione a basso livello delle informazioni all'interno della base dati. È orientato ai fatti, da considerarsi non come una procedura per far qualcosa direttamente ma in modo intenzionale, si descrive attraverso dei metodi il cosa si desidera ottenere e non il come, la specifica implementazione delle procedure da eseguire per ottenere i risultati è data dall'ORM stesso, come strategia di elaborazione implementata da chi lo ha costruito. È importante, perché consente anche di cambiare il motore sottostante senza grosse difficoltà o eccessivi cambiamenti strutturali dell'applicazione. Tanto più si riesce ad essere dichiarativi, meglio è, così il singolo sviluppatore sarà responsabile di quel che accade a livello di operazioni svolte dal suo codice, altrimenti capire tutti i possibili casi ed intercettare ogni possibile fonte di errore o comportamento indesiderato provocherebbe spreco di tempo; sarà invece questo strato software che provvederà a rendere esplicite le dipendenze degli oggetti tra loro e la base dati.

Vantaggi degli ORM/ONRM sono molteplici: maggiore produttività, non si scrive del codice ripetitivo e le entità sono snelle nel contenuto (DRY – *Don't Repeat Yourself*). Al programmatore d'ora in avanti sarà chiesto di esplicitare il cosa fare, non il come, a meno di situazioni complesse in cui potrebbe essere necessario intervenire sull'effettiva introduzione di una porzione di codice appropriata per svolgere un certo compito. Altro vantaggio è legato alla possibilità di non dover creare le tabelle della base dati, infatti è possibile generarle automaticamente con opportuni comandi, schemi e/o annotazioni o, in alternativa, generare le classi a partire da un modello di persistenza già esistente. La generazione automatica a seguito di modifiche dello schema è da escludere in *production*, pena la rischiosissima perdita di dati con conseguenti danni vari. Accanto alla maggiore produttività vi è la manutenibilità, meno codice da scrivere per il programmatore e soprattutto utilizzo di codice che (solitamente) è ben controllato da una comunità di sviluppatori che forniscono librerie robuste e senza bachi software; per tal motivo è possibile sostituirla senza dover apportare gravi modifiche alla *business logic* dell'applicazione. Vi sono anche vantaggi in termini di prestazioni, chi ha prodotto quello specifico ORM/ONRM ha progettato un codice che genererà delle azioni ottimizzate, poiché sarà stato necessario aver studiato nel dettaglio quel particolare DBMS, in modo da poter sfruttare tutte le caratteristiche e gli algoritmi disponibili. Inoltre, con queste librerie viene spesso supportato automaticamente un modello di "*lazy evaluation*", in cui il codice può far riferimento ad un oggetto senza che questo sia stato effettivamente recuperato dalla base dati, in tal modo si risparmia molto in termini di *overhead*, nonostante dal punto di vista sintattico non cambi molto; sono inoltre implementate delle politiche di *caching* per le transazioni. Introducono anche un livello maggiore di astrazione sia dal particolare tipo di DBMS che da modifiche nella struttura della base dati, forniscono un interfacciamento elegante a supporto dei meccanismi che esistono già nel DBMS originario, ciò che non è supportato non è aggiunto. Come intuibile, garantire tutto ciò oltre a sporcare il codice avrebbe influito sulla produttività del singolo programmatore e sulla robustezza dell'applicazione.

Ma non è tutto oro quel che luccica, anche gli ORM hanno qualche svantaggio e vari limiti. Si tratta di uno strato software aggiuntivo, nonostante possa essere stato scritto con i migliori algoritmi per l'accesso alle basi dati. Non può far nulla a fronte di situazioni complesse in cui vi sono un gran numero di cicli macchina da impiegare per portare a termine un compito o alle troppe allocazioni in memoria, è una responsabilità del programmatore capire le situazioni onerose e cercare di risolverle con del codice ben fatto. La generazione della base dati è rischiosa in *production* e ovviamente gli ORM aiutano laddove il DBMS sottostante espone funzionalità, non può far miracoli su ciò che manca o non è ottimizzato. Si ha inoltre un accoppiamento (*coupling*) abbastanza forte tra database e logica dell'applicazione, l'unità di test di un oggetto senza una base dati corrispondente diventa difficile. Gli effetti negativi sulla testabilità dell'Active Record Pattern possono essere ridotti al minimo mediante l'uso di strutture per l'iniezione di dipendenza (*Dependency Injection*) o di sostituzione del dato reale con una sua simulazione (mock) ossia un qualcosa di fittizio utile solo a testare la logica applicativa.

Vi sono diversi tipi di ORM, distinti per framework e DBMS, con l'opportuno linguaggio e le diverse caratteristiche. Ad esempio, nel caso di Java vi sono Hibernate, implementazione dello standard JPA, che supporta però anche la modalità nativa (cambiano i nomi delle parti ma i concetti sono uguali), iBATIS di Apache, e altri. Nel caso di Rails vi sono "ActiveRecord" come ORM per qualsiasi base dati relazionale, uno strumento molto potente che prevede l'aggiunta di estensioni in base al tipo di database, come "PG" nell'usare PostgreSQL, "postgis" per usare l'estensione PostGIS. Vi sono ORNM per i DBMS NoSQL indipendenti da ActiveRecord, come "MongoID" per il noto database MongoDB, "Redis" per l'omonimo DBMS, altri per Cassandra e così via. Nel lavoro svolto, sono stati utilizzati simultaneamente ActiveRecord, MongoID, postgis e Redis, oltre che un mapper come ponte tra alcune risorse presenti sia in MongoDB che in PostgreSQL, utile per calcoli geografici, come dettagliato in Sez. 3.4.3.

2.4 DBMS NoSQL

Le basi dati relazionali nel tempo si sono evolute diventando potentissime, sono ormai parte di una grande quantità di applicazioni, sia web che stand alone. Le assunzioni che questi RDBMS (Relational DBMS) fanno sono molteplici, i dati sono organizzati in tabelle che contengono tuple omogenee tra loro nella forma, come collezione di valori interrogabili attraverso SQL. Il dato è normalizzato, ossia non si ha replicazione delle informazioni e qualunque risorsa faccia riferimento ad un'altra, la vedrà sempre aggiornata al suo ultimo stato e ogni modifica va fatta su un'unica entità. Il loro ordine non è rilevate, ogni tupla è identificata nel contesto del database da un identificativo univoco (chiave primaria) composta anche da più colonne, grazie al quale effettuare le dovute operazioni. Consentono anche di settare un insieme di regole di integrità che scattano in automatico al verificarsi di certi eventi (trigger) per garantire certi vincoli. La transazione è unità logica di lavoro, che consente di portare a termine un gruppo di operazioni o ripristinare la base dati nello stato presente prima dell'inizio della transazione stessa attraverso il rollback, se necessario. Vi sono le proprietà ACID alla base dei RDBMS, sicuramente importanti ma non la soluzione ad ogni problema. Uno dei fattori più limitanti è la necessità di avere dati strutturati e di definire uno schema per la loro memorizzazione, dopo aver creato la corrispondente tabella che ne descrive i tipi di dato nelle varie colonne. È ovvio che durante lo sviluppo iniziale del software non si abbia da subito una chiara idea dello schema definitivo, per cui si effettuano vari aggiustamenti. In modalità production ciò non è così semplice e potrebbe comportare disservizi fino a causare la perdita dei dati in una base dati mal progettata. Altro grosso svantaggio è la normalizzazione dei dati, che sicuramente aiuta nell'evitare repliche delle informazioni, ma comporta svantaggi computazionali che possono diventare insostenibili al crescere della complessità del dato e alla quantità di

informazioni (join annidati in interrogazioni complesse). Vi sono delle situazioni in cui è bene replicare parte delle informazioni, ad esempio la lista di interessi di un utente può contenere elementi che sono di interesse anche per altri utenti, per cui ha senso per ognuno avere già l'intera lista senza dover creare rappresentazioni semplici ed effettuare query complesse. Alcuni problemi sono già stati discussi nella sezione dedicata agli ORM, ma dal punto di vista della scalabilità orizzontale gli RDBMS non sono così ottimi, poiché progettati per garantire le proprietà ACID. Per tal motivo nascono i DBMS NoSQL, detti così perché abbandonano il modello ERM e quindi non è più necessario utilizzare SQL come linguaggio, ma di fatto espongono un set di metodi per le operazioni di accesso e modifica della base dati. Sono nativamente capaci di gestire dati non strutturati, solitamente raggruppabili in delle collezioni che inglobano informazioni più o meno simili tra loro, ma non si tratta di un fatto obbligatorio; inoltre, sono nativamente progettati per essere ospitati su cluster di elaboratori, offrendo dei meccanismi per la divisione del carico di lavoro, molto utile per la scalabilità orizzontale e in operazioni di data mining. Tali sistemi in realtà hanno qualche svantaggio rispetto ai sistemi RDBMS, poiché nei sistemi distribuiti non è possibile garantire contemporaneamente le proprietà di consistenza dei dati, disponibilità del servizio e la garanzia di poter funzionare anche se una delle componenti del sistema non funziona (Teorema CAP o Teorema di Brewer). Delle tre proprietà, al più, solo due possono essere garantite, per cui nella costruzione di un sistema distribuito è necessario tenerlo in considerazione. Cambia dunque anche il modello di consistenza dei dati, perché le informazioni sono distribuite e non è detto che in un certo istante tutte le parti abbiano la stessa copia aggiornata. È possibile negoziare un certo grado di visibilità, da cui nascono i modelli di consistenza *eventual, session, causal consistency* e *serializability* (dal più leggero al più forte). Dovendo "rinunciare" a qualcosa di ACID, il modello di proprietà che si adotta nei sistemi distribuiti NoSQL è il BASE (Basically Available, Soft State, Eventual Consistency), che garantisce il funzionamento del sistema anche se una delle componenti o dei nodi non è raggiungibile, un modello di consistenza dei dati in cui questi prima o poi si allineeranno in tutte le parti e la presenza di cambiamenti di questi senza input esterni, per tal motivo si parla di soft state. La conoscenza di tale modello è fondamentale perché altrimenti nella progettazione di un sistema si fanno delle assunzioni indebite basate su ACID che il DBMS NoSQL non può garantire.

2.5 REST

In questo contesto generale appare la filosofia REST (Representational State Transfer), pensata da Roy Fielding, alternativa ad altre specifiche di scambio messaggi nel computing distribuito come SOAP, volto a sfruttare alcune proprietà del software stratificato e caratteristiche del modello distribuito. Nel parlare di REST, il contesto è quello di un modello di interazione Client-Server in cui il client ha capacità computazionali non trascurabili, per cui si pensa ad una suddivisione di compiti tra le parti in modo da garantire la scalabilità del sistema, mantenendo al contempo la semplicità dell'implementazione. I server ospitano dei servizi, una raccolta omogenea di informazioni e processi a supporto delle attività dei client. Queste informazioni vivono all'interno di una base dati e possono essere recuperate attraverso un identificativo, che però è univoco soltanto nel contesto della base dati stessa, al di fuori non è detto che abbia un significato né che abbia universalità. Nel modello REST viene sfruttato tale identificativo per accedere in modo universale alle informazioni presenti nella base dati: l'informazione è resa globalmente univoca senza alcuna ambiguità attraverso l'aggiunta del nome di dominio associato a quel particolare contesto web, diventando così una URI (Uniform Resource Identifier). Ad esempio, considerando come nome di dominio "www.example.com" e una collezione di risorse chiamata "collection", una sua risorsa la cui chiave primaria è "33" può essere indirizzata in modo universale attraverso l'URI "www.example.com/collection/33". L'universalità della risorsa è data dall'aver anteposto ad un

valore univoco in contesto non globale (33) un altro valore univoco (www.example.com/collection) in contesto globale e in tal modo si acquisisce un vantaggio non banale: chiunque può fare riferimento a quella entità senza alcun tipo di ambiguità, cioè esisterà una sola risorsa identificata in quel modo. Esponendo i dati con un riferimento universale si possono esportare meccanismi per generalizzare le interazioni con l'informazione. Il dato ora si presta molto bene per essere coniugato con la semantica dell'HTTP, utilizzando i classici verbi per effettuare delle operazioni abbastanza intuitive: l'azione *GET* su *www.example.com/collection/33* fornisce il dato referenziato, l'azione *DELETE* su *www.example.com/collection/33* lo elimina, e così via. Pur essendo possibile rappresentare le quattro operazioni fondamentali della persistenza dei dati (CRUD - Create, Read, Update, Delete) mappandole su corrispondenti verbi HTTP, non è detto che queste debbano essere tutte esposte al mondo esterno: magari per alcune risorse sono ammesse operazioni di sola lettura, altre hanno senso globalmente piuttosto che specifiche della risorsa, ad esempio raramente è esposta la DELETE sull'intera collezione; a dirla tutta, la vera miniera d'oro di oggi è fatta proprio dai dati, per cui la DELETE spesso corrisponde a "disattivare" il dato stesso, senza realmente eliminarlo (al più viene eliminato dopo gli opportuni processamenti necessari). Il poter sfruttare la semantica di HTTP non riguarda soltanto le operazioni fondamentali che questo espone, ma riguarda anche il meccanismo di risposta che questo offre. Ha una rilevanza non banale, perché evita al programmatore il dover inventare un meccanismo (spesso sbagliato o poco efficace) per la gestione delle risposte e degli errori. È possibile sfruttare il meccanismo standardizzato di HTTP dei codici di stato per il risultato di ogni operazione: aderire pienamente all'HTTP aiuta ad eliminare quello strato di fantasia nel creare nuove sintassi e facilita l'interoperabilità anche tra parti umane.

In generale, non è escluso il non poter dare al client dei riferimenti alle entità che sono in qualche modo relazionate con l'entità su cui si effettua una certa operazione, lasciando a lui la responsabilità di utilizzare tali informazioni di contorno, se necessario. Il vantaggio non è solo evitare richieste aggiuntive per l'ottenimento di tali relazioni, ma evitare alle entità di terze parti che manipolano il risultato il non essere a conoscenza della semantica del dato stesso. REST è ben orientato ad essere *stateless*, quando si fa riferimento ad una risorsa questa viene elaborata senza tenere conto della storia passata per quello specifico client. Il server opera esclusivamente su quel che gli viene presentato dal client, infatti deve essere quest'ultimo, se necessario, a dover mantenere una storia per quell'entità o quell'operazione, in base al tipo di azione svolta. Pur se il modello REST è in accordo con il concetto di *stateless* di HTTP, i meccanismi *stateful* applicativi (come le sessioni) sono da gestire a parte, lato server. Altra assunzione del modello REST è che a parità di contenuto informativo sono possibili diverse forme di rappresentazione delle informazioni, tra di loro equivalenti, volte a facilitare il processamento dell'informazione, che non vuol dire fornire i dati in formato JSON piuttosto che XML (possibile grazie al meccanismo di negoziazione offerto da HTTP tramite *Header* appositi), ma ad esempio tradurre in diverse lingue lo stesso contenuto di base. Nel linguaggio del modello infatti si parla di "risorsa".

« *La separazione REST Client-Server degli interessi semplifica l'implementazione del componente, riduce la complessità della semantica del connettore, migliora l'efficacia dell'ottimizzazione delle prestazioni ed aumenta la scalabilità di componenti server puri. I vincoli di sistema a strati permettono di introdurre intermediari — proxy, gateway, e firewall — in vari punti della comunicazione senza cambiare le interfacce tra i componenti, consentendo loro di assistere nella traduzione della comunicazione o migliorare le prestazioni tramite cache condivisa di larga scala. REST consente la elaborazione intermedia vincolando i messaggi ad essere auto-descrittivi: l'interazione è priva di stato tra le richieste, i metodi di base ed i tipi di media sono utilizzati per indicare la semantica e scambiare informazioni e le risposte indicano esplicitamente la cachabilità* » - [Wikipedia].

Di REST è stato pensato anche un "Maturity Model" (Richardson) con quattro livelli (0-3), dal meno al più vicino alla reale filosofia REST. Richardson ha voluto evitare che l'enfasi del nuovo modello portasse ad una digressione più o meno ampia del suo significato. Nella pratica è bene attenersi al livello 3 di REST non per poter affermare che un certo software è "REST" e basta, ma perché porta dei vantaggi non trascurabili. Ad esempio, il livello 3 mappa ogni risorsa su una URL univoca, ogni azione su un verbo HTTP e ingloba il concetto di HATEOAS (Hypermedia as the Engine of Application State) grazie al quale si introducono riferimenti e relazioni nel risultato, sempre attraverso URL con stessa sintassi e semantica delle precedenti, volte a fornire le relazioni dell'entità in gioco con le altre entità. Ciò è molto interessante perché abilita un modello di riconoscimento semplificato delle interfacce e consente una evoluzione del software senza dover forzare (di solito) il cambiamento del client stesso. Ad esempio, un riferimento del tipo "*/help*" può essere mappato su "*/help/{id}*", ma se per qualche motivo si volesse cambiare in *"/helpPoint/{id}"* il riferimento per l'azione lato client sarà sempre "*/help*", con il vantaggio che questo sa come adattarsi. Anche l'evoluzione del server è resa più semplice, altrimenti il client dovrebbe avere cablati tutti gli *endpoint* e le azioni, limitando di fatto cambiamenti marcati per l'architettura software del server. Strutturando l'applicazione con il livello 0, di contro, si è ben distanti dal modello REST: si utilizza una sola URL per tutte le azioni e tutte le risorse, discriminandole attraverso i parametri passati al server e utilizzando POST come unico verbo HTTP di interazione. In tal caso, non si sfrutta pienamente la semantica di HTTP, che viene usato esclusivamente come "tunnel" per l'interazione Client-Server. I livelli 1 e 2 sono intermedi agli estremi spiegati, spesso si ritrova il livello 2 che consiste nell'implementare un livello 3 senza HATEAOS, mentre il livello 1 differenzia le URI per risorse senza tutti i verbi HTTP, ma solo l'azione POST con le specifiche nel body della richiesta.

Nei quattro Capitoli a seguire sarà descritto in dettaglio lo stack tecnologico scelto, come ognuna delle parti abbia contribuito al raggiungimento di un particolare obbiettivo, i problemi affrontati e le strategie utilizzate per risolverli.

3. Ruby on Rails

In ambito di sviluppo web si sono susseguiti nel tempo vari framework, tra cui spicca l'utilizzo massiccio di Ruby on Rails per la costruzione di applicazioni web dinamiche. Curva di apprendimento praticamente piatta e potenza del framework sono non gli unici ma due dei tanti vantaggi principali. Twitter, Basecamp, GitHub, Shopify, Airbnb e tante altre applicazioni web sono state costruite con tale framework. In questa sezione saranno analizzate le caratteristiche sia di Ruby che di Rails: di Ruby saranno trattati solo gli aspetti generici, senza focalizzare i dettagli del linguaggio in termini di programmazione; di Rails invece saranno descritte le parti utili all'implementazione di alcune strategie riguardanti le REST API, autenticazione, caching e il collegamento con le varie basi dati. Sarà inoltre descritto il meccanismo di logica applicativa per la ricerca georeferenziata dei dati.

3.1 Ruby

Cos'è Ruby? Un linguaggio open-source dinamico che dà particolare rilevanza alla semplicità e alla produttività, dotato di una sintassi elegante, facile da leggere e da scrivere. Non è soltanto "pulito ed elegante", il suo creatore Yukihiro "Matz" Matsumoto ha fuso insieme parti dei suoi linguaggi di programmazione preferiti (Perl, Smalltalk, Eiffel, Ada e Lisp) per creare un nuovo linguaggio in grado di bilanciare programmazione funzionale con programmazione imperativa (o procedurale). In un linguaggio procedurale l'enfasi è sui dati (variabili, strutture dati, oggetti) e sui modi per processarli tramite appunto procedure, funzioni, metodi. In pratica, si parte da un insieme di dati e si applicano a questi una serie di operazioni da eseguire in sequenza per produrre il risultato finale. Le operazioni da eseguire vengono descritte come una serie di istruzioni, comandi e ordini che il processore deve eseguire per ottenere il risultato. Il modo di pensare della maggior parte dei programmatori è fortemente orientato alla programmazione procedurale, si pensa un algoritmo come una sequenza di passi più o meno elementari. Alcuni esempi sono i linguaggi C/C++, C#, Java, Python, ecc.

I linguaggi funzionali, invece, pongono tutta l'enfasi sulle funzioni. I dati sono tuttavia ancora presenti, però le funzioni assumono una rilevanza ben più importante. Per fare alcuni esempi, in un linguaggio funzionale una funzione può essere argomento di un'altra funzione, invocata per ottenerne come parametro il suo valore di ritorno; data una funzione di due variabili, è possibile fare una proiezione su una delle due variabili (assegnandogli una valore) ottenendo una funzione di una variabile, per sfruttare il concetto di puntatore a funzione, e così via. Il principio di base è che il valore di una espressione dipende solamente dai valori delle sue sub-espressioni, se ne esistono. Di seguito è mostrato un esempio di linguaggio astratto imperativo per il calcolo del fattoriale di un numero:

```
1.  function fatt(n)
2.      prod = 1
3.      if (n > 1)
4.          for i in range(2, n+1)
5.              prod = prod * i
6.              n = n - 1
7.      return prod
8.  end
```

mentre il corrispondente in linguaggio funzionale viene così espresso:

```
1.  def fact(n)
2.    if n == 0
3.     return 1
4.    else
5.      return n * fact(n-1)
6.    end
7.  end
```

Come facilmente intuibile, la programmazione funzionale è fortemente orientata alla ricorsione. Naturalmente, anche nei linguaggi imperativi si può usare opzionalmente la ricorsione, anche mischiando le due modalità, mentre nella programmazione funzionale ai programmatori è praticamente impedito l'utilizzo di comandi imperativi: non è possibile definire variabili locali temporanee nelle funzioni o globali, bisogna ridurre tutto a delle funzioni.

Proprio questa mancanza di flessibilità dichiarativa ha portato Matz all'invenzione di un linguaggio che combina la programmazione funzionale con quella imperativa. Ma Ruby è apparentemente semplice, è tanto semplice per le cose semplici e tanto complesso per le cose complesse. È un linguaggio di altissimo livello, il che significa che astrae la maggior parte dei dettagli complessi della piattaforma e dei concetti di programmazione, aiutando però a creare rapidamente qualcosa da zero con un ridotto numero di linee di codice. Non è tutto, Ruby è stato concepito per essere il più naturale possibile, "*in un modo che rispecchia la vita*" – [Matz].

Ruby è un linguaggio di programmazione dinamico con una grammatica espressiva, varie librerie con funzionalità ricche e potenti, distribuite nelle cosiddette *gems* (gemme) di Ruby. Pur essendo adatto a stili di programmazione come quella procedurale e funzionale, è un linguaggio "Object-Oriented" puro. A differenza dei classici linguaggi orientati agli oggetti, in Ruby sono qualcosa di molto più dinamico, in quanto è possibile aggiungere o modificare metodi a runtime. Tutto in Ruby è un oggetto, persino i tipi di dato primitivi, i numeri o *nil* (il classico *null*) che indica l'assenza di un valore. È un linguaggio di scripting interpretato linea per linea da una virtual machine, non compilato ed eseguito. La sua sintassi è fortemente orientata all'espressione, è possibile ottenere lo stesso risultato in termini di operazioni strutturando il codice in modi differenti; esistono una serie di convenzioni operative che rendono il linguaggio non facilmente intuibile a primo impatto e, pur sembrando una caratteristica interessante, ciò è spesso fonte di confusione per chi approda a questo linguaggio di programmazione: ad esempio, non si usa il punto e virgola alla fine di ogni comando o dichiarazione, le parentesi per i blocchi di codice sono sostituite da tabulazione e parole chiave, i metodi possono essere chiamati senza le parentesi quadre e le assegnazioni possono essere fatte in modo condizionale, non esistono costruttori espliciti e obbligatori, e tanto altro. Come se non bastasse, tra i principi di programmazione e soprattutto tra i programmatori esperti, vi è uno sforzo massimo nella riduzione del numero di linee di codice da scrivere in Ruby per implementare un determinato algoritmo. Per tal motivo la sintassi di Ruby sembra criptica e per far chiarezza su quanto detto, viene ora mostrato un esempio di algoritmo generico per la ricerca del massimo di due numeri in Ruby:

```
1.  def max(a, b)
2.    if a > b
3.      return a
4.    else
5.      return b
6.    end
7.  end
```

In realtà, seguendo i principi dei "rubysti", si dovrebbe scrivere qualcosa del genere:

```
1.  def max a,b
2.    a if a > b else b end
3.  end
```

Come intuibile, sono omessi i return, Ruby restituisce l'ultima variabile o valore di ritorno in chiamata a metodo scritto per ultimo; ma un programmatore "esperto" avrebbe scritto lo stesso algoritmo nel seguente modo:

```
1.  def max a,b
2.    [a,b].max
3.  end
```

Questo esempio sottolinea la grande differenza con gli altri linguaggi di programmazione, oltre che la necessità di consultare la documentazione per diminuire la quantità di codice, se necessario. Pur non essendo facile a primo impatto, con l'esperienza si acquisirà maggiore facilità nel capire la sintassi e sono proprio quest'insieme di concetti che rendono Ruby affascinante e potente. Ruby è anche un linguaggio flessibile, dal momento che permette al programmatore di alterare liberamente le sue parti, infatti possono essere rimosse o ridefinite, senza dare troppi limiti agli sviluppatori. Per esempio, la somma è eseguita mediante l'operatore "+", ma è possibile usare qualcosa come:

```
1.  class Numeric
2.    def aggiungi(x)
3.      self.+(x)
4.    end
5.  end
6.
7.  y = 5.aggiungi  6
```

Ossia estendendo la classe base "Numeric" con il metodo "aggiungi", tutto ciò che Ruby può interpretare come un numero ha a disposizione il metodo definito. Ciò consente di focalizzare su un altro aspetto importantissimo di Ruby: il concetto di classe, la sua estensione e il supporto all'ereditarietà. A differenza di molti altri linguaggi ad oggetti Ruby ha solamente l'ereditarietà singola, volontariamente. Tuttavia, dispone del concetto di "modulo" (chiamato Categoria nell'Objective-C), ovvero un raggruppamento di metodi implementati per creare dei "miscugli" (*mixins*): le classi possono infatti includere al loro interno un modulo per ereditarne tutti i suoi metodi, automaticamente. Non esiste il concetto di interfaccia (Java) in Ruby.

Esempio:

```
1.  module MyModule
2.    module Alpha
3.      #methods here
4.    end
5.    module Beta
6.      #methods here
7.    end
8.  end
9.
10. class MyClassAlpha
11.   include MyModule::Alpha
12. end
13.
14. class MyClassBeta
15.   include MyModule::Beta
16. end
```

Ogni istanza della classe "MyClassAlpha" erediterà tutti i metodi definiti nel modulo "MyModule::Alpha", mentre la classe "MyClassBeta" tutti quelli definiti in "MyModule::Beta". La sintassi "::" serve per il *namespacing*. Generalmente gli appassionati di Ruby vedono questa pratica come molto più pulita e gestibile dell'eredità multipla, che è complessa e può essere troppo restrittiva in alcuni casi. Ruby ha un sacco di altre funzionalità, tra cui threading indipendente dal sistema operativo, in modo da renderlo disponibile su tutte le piattaforme in grado di eseguire il codice, a prescindere dal fatto che il sistema operativo lo supporti o meno; fa uso di un garbage collector e consente anche la gestione delle eccezioni come in Java o C++.

Non è tutto oro quel che luccica, Ruby ha anche dei difetti, e non pochi. È stato pensato nel 1990, rilasciato nel 1995, accettato "universalmente" nel 2006 e tutt'oggi è in continua evoluzione. Essendo un linguaggio interpretato, la sua virtual machine che fa da interprete del bytecode assorbe l'overhead del suo essere un linguaggio di altissimo livello, risultando a volte in prestazioni pessime per algoritmi molto complessi. Vi sono state varie VM nel tempo a supporto di Ruby, oggi si utilizza YARV (*Yet Another Ruby VM*), è un interprete che è stato sviluppato con l'obiettivo di ridurre notevolmente i tempi di esecuzione dei programmi Ruby, usa C come linguaggio di base e nel tempo ha subito notevoli cambiamenti soprattutto a favore delle prestazioni, inizialmente scadenti. La maggior parte degli svantaggi derivano dalle difficoltà di essere un nuovo linguaggio di programmazione tra i diversi concorrenti veterani. Pur avendo una grande comunità a supporto della sua evoluzione, non si può definire un vero e proprio concorrente di altri linguaggi come Java o C/C++. Ruby è abbastanza nuovo e ha il suo stile di programmazione unico, alcuni programmatori considerano questo uno svantaggio perché richiede del tempo solo per averne piena padronanza prima di utilizzarlo. Dal momento che l'apprendimento Ruby è come imparare un'altra lingua, molti programmatori preferiscono attenersi a ciò che già conoscono e possono sviluppare. Eppure il materiale (gratuito e non) è davvero completo, tra cui il libro scritto dallo stesso ideatore Matz: *The Ruby Programming Language* [1].

Ruby è un linguaggio libero e gratuito, può anche essere usato, copiato, modificato e distribuito liberamente. Nel sito ufficiale (https://www.ruby-lang.org) è possibile accedere alla documentazione e ai tutorial per imparare in tempi davvero brevi le nozioni di base per il suo utilizzo; bastano fino a 30 minuti per un apprendimento superficiale fino a qualche ora per nozioni avanzate. È stato progettato come un linguaggio di scripting generico e quindi ha un ampio supporto per una serie di applicazioni diverse. È stato utilizzato in vari ambiti, dalle applicazioni web ai server web stessi, alle librerie di grafica intelligente, ai motori di riconoscimento delle immagini, a cluster di database, utility di sistema di basso livello. Pur sembrando svantaggi abbastanza pesanti, Ruby viene utilizzato in contesti che è possibile ottimizzare in altro modo, soprattutto in ambito di applicazioni web. Infatti, nonostante l'indice TIOBE (https://www.tiobe.com/tiobe-index/) abbia riportato una crescita con alti e bassi nel tempo per questo linguaggio, Ruby non avrebbe mai preso così tanta popolarità se non grazie al framework web "Rails", meglio noto come Ruby on Rails, motore logico dell'applicazione ideata.

3.2 Rails: principi fondamentali

Prima di affrontare le strategie del caso e capire come Ruby on Rails aiuta a risolvere determinati problemi, è bene capire di che framework si tratta e il suo funzionamento. Ruby on Rails, o semplicemente Rails, è un framework per applicazioni web lato server scritto in Ruby sotto la licenza MIT. Rails è un framework MVC (Model-View-Controller), fornisce strutture predefinite per la gestione dei dati tramite DBMS e le pagine web attraverso un web server di default, sostituibile a piacimento. Facilita l'utilizzo di standard web quali JSON o XML per il trasferimento dei dati, e HTML, CSS e JavaScript per la visualizzazione dei contenuti. Oltre a MVC, Rails

sottolinea l'uso di altri noti modelli e paradigmi di ingegneria del software, tra cui "*Convention over Configuration*" (CoC), "*Don't Repeat Yourself*" (DRY) e "*Active Record Pattern*". Questi tre principi di base sono pilastri portanti del successo e della flessibilità di Rails, ma non sono gli unici.

3.2.1 Convention over Configuration

È un paradigma di programmazione fornito da vari framework che prevede una configurazione minima (o addirittura assente) a carico del programmatore, obbligandolo a configurare solo gli aspetti che si differenziano dalle implementazioni standard o che non rispettano particolari convenzioni. Questo tipo di approccio semplifica notevolmente la programmazione soprattutto agli stadi iniziali dello studio di un nuovo framework, senza necessariamente perdere flessibilità o possibilità di discostarsi dai modelli standard. Questo concetto è stato introdotto da David Heinemeier Hansson per descrivere la filosofia alla base del suo framework Ruby on Rails, ma non è legata esclusivamente a questo, anche il framework Spring Boot si basa su tale paradigma. È interessante perché quando la convenzione implementata dai vari tool corrisponde al comportamento applicativo desiderato, si è risolto un passo fondamentale nella costruzione dell'applicazione: non è stato necessario effettuare appositamente alcuna configurazione per quello scopo; solo quando il comportamento desiderato si discosta da quello ottenuto, è necessaria una configurazione esplicita. Utile al programmatore, perché non deve tener conto di vari aspetti di che farebbero altresì perdere del tempo non banale, ci sono un sacco di decisioni da prendere in applicazioni complesse legate alla configurazione del sistema e del codice, è buona cosa se fossero automaticamente risolte.

Un esempio classico è legato ai nomi dei modelli e alle corrispondenti tabelle del database, un modello chiamato "Product" (modello del prodotto) avrà corrispondenza (attraverso un apposito ORM) con la tabella "products" (tabella dei prodotti), ma se il programmatore necessita di chiamare la tabella in altro modo, deve farlo esplicitamente. Ruby on Rails ad esempio usa questo concetto anche sulla struttura delle cartelle del progetto, evitando file di configurazione XML o annotazioni immerse nel codice, come richiesto da altri framework come Spring. Anche GRails (Groovy on Rails), noto framework meno utilizzato rispetto a Ruby on Rails, ma con stessa filosofia di base, usa un approccio simile per la struttura del progetto. Il namespacing ad esempio è risolto guardando inizialmente le keyword del linguaggio, tra cui *module* e *class*, se un modulo o una classe non è trovato nel file corrente, si cerca nella gerarchia di cartelle in modo ricorsivo. Se non trovato, viene lanciata un'eccezione. La convenzione dei nomi non è casuale, nel codice si fa riferimento ad una classe (o modulo) in modalità "CamelCase", mentre gli elementi del file system sono nominati in modalità "snake-case": ad esempio, la classe *MyProduct*, definita in un file chiamato *my_product.rb* nella cartella *my_resources,* si può richiamare all'interno del codice come *MyResources::MyProduct*, e così via, grazie ad un meccanismo di traduzione dei nomi.

Alcuni framework richiedono più file di configurazione, ciascuno con molti settaggi da fare. Questi forniscono informazioni specifiche per ciascun progetto, che vanno dagli URL al mapping tra le classi e le tabelle del database. Un gran numero di file di configurazione con un sacco di parametri è spesso difficile da mantenere. Ad esempio, le versioni precedenti del mapper Hibernate per la persistenza in Java, le relazioni tra le entità o i campi principali dell'entità venivano descritti in file XML. La maggior parte di queste informazioni potrebbero essere descritte in modo convenzionale mappando i nomi delle classi alle tabelle di database, le cui colonne hanno una corrispondenza uno a uno con i campi dell'entità. In vari framework infatti le versioni successive hanno eliminato i file di configurazione XML e hanno impiegato convenzioni come queste. Ovviamente le configurazioni esplicite sono ancora possibili, ma raramente utilizzate.

Tuttavia il paradigma CoC porta svantaggi non banali, perché il framework stesso diventa una sorta di "black box", una scatola nera di cui non tutto è così immediato, se non attraverso una minuziosa conoscenza del framework stesso, che può risultare scomodo per quei programmatori che sin dall'inizio hanno bisogno di scrivere del codice abbastanza complesso, spesso risultando in un codice "sporco" e non facilmente leggibile. Altri framework per tal motivo prediligono la linea del "*explicit is better than implicit*", ossia le cose esplicite risultano essere migliori delle cose implicite. Purtroppo per chi approda a framework che usano tale paradigma, questi sono spesso visti come qualcosa di magico e non facilmente intuibili nel dettaglio, è il prezzo da pagare per ottenere un rapido setup di un'applicazione web.

3.2.2 Don't Repeat Yourself

In ingegneria del software, il principio *Don't Repeat Yourself* (non ripeterti) è un principio di progettazione e sviluppo secondo cui andrebbe evitata ogni forma di ripetizione e ridondanza logica nell'implementazione di un sistema software, spesso abbreviato come DRY e noto anche come *Single Point of Truth* (singolo punto di verità). Il principio venne inizialmente enunciato da Andy Hunt e Dave Thomas nel loro libro *The Pragmatic Programmer* (1999):

«Ogni elemento di conoscenza deve avere una sola, non ambigua, autorevole rappresentazione all'interno di un sistema.»

Il DRY viene spesso citato in relazione alla duplicazione del codice, ovvero nell'accezione stretta secondo cui il software non dovrebbe contenere sequenze di istruzioni uguali fra loro in punti diversi. Si tratta però di un concetto più ampio, che si applica a ogni parte di un sistema software, inclusi per esempio schemi di database, direttive di *build*, file di configurazione e persino la documentazione. L'applicazione completa del DRY implica logicamente che una modifica a un singolo elemento di un sistema non debba mai comportare la necessità di modificare altri parti di un sistema per apportare la modifica stessa. Ciò è molto importante in architetture multi-tier, in cui la stessa informazione è gestita a diversi livelli e attraverso diverse tecnologie (per esempio interfaccia utente, logica dell'applicazione, database). Questo rende particolarmente difficile evitare la duplicazione dell'informazione nei diversi livelli. Gli approcci possibili all'applicazione del DRY in questi contesti prevedono in genere l'uso di strumenti automatici per generare diversi artefatti (per esempio codice in diversi linguaggi e schemi di database) a partire da un'unica rappresentazione di partenza. Il principio DRY si ritrova anche nel framework Rails, sia per la scrittura del codice che nel concetto di "Separation of Concerns" (separazione delle responsabilità) per applicazioni multi-tier in cui presentazione, processamento e gestione dei dati sono funzionalità logicamente e separate in prima fase e, come potenzialmente accade nell'utilizzare servizi cloud, fisicamente separate. Quest'ultimo concetto legato al multi-tier sarà ripreso nel penultimo Capitolo, in cui sarà dettagliata la strategia di deployment per la costruzione di un'architettura facilmente scalabile.

3.2.3 Active Record Pattern

In ingegneria del software l'Active Record Pattern è un modello architetturale del software che memorizza gli oggetti presenti in memoria in una base dati relazionale. È stato così chiamato da Martin Fowler nel suo libro *Patterns of Enterprise Application Architecture* (2003). L'interfaccia di un oggetto conforme con questo modello di programmazione include funzioni che rispecchiano le azioni di *insert*, *update* e *delete*, oltre a metodi che rispecchiano le colonne del database sottostante a supporto della memorizzazione.

È un approccio per accedere ai dati di un database, le cui tabelle sono mappate in classi "*wrapper*" e le informazioni degli oggetti sono rappresentate nel database da una riga della tabella. Dopo la creazione di un oggetto, una nuova riga è aggiunta nella tabella in fase di salvataggio, mentre ogni oggetto caricato acquisisce i suoi valori corrispondenti direttamente dal database; quando un oggetto viene modificato, la corrispondete riga nel database viene ulteriormente aggiornata. La classe "*wrapper*" implementa dei metodi o proprietà per ogni colonna della tabella. Questo pattern è molto utilizzato da strumenti di persistenza dei dati e dagli ORM. Tipicamente, le chiavi esterne (*foreign key*) saranno esposte come istanza di un tipo appropriato e accessibili attraverso proprietà e metodi.

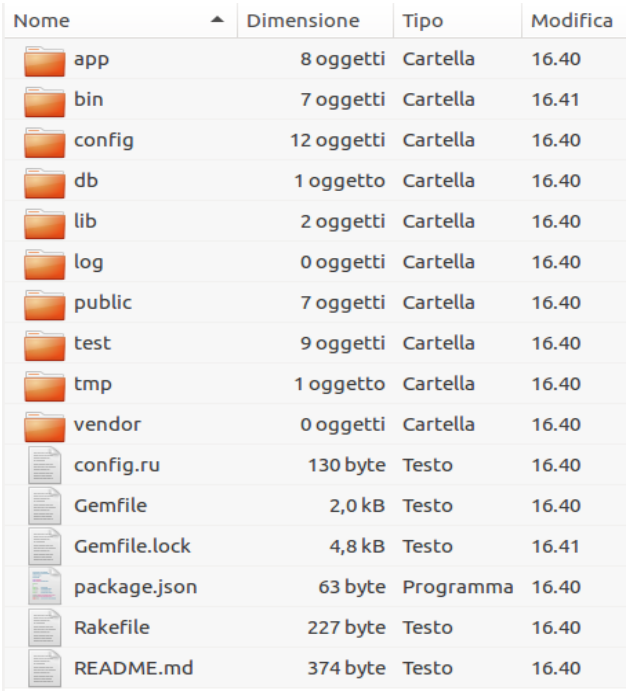
Il concetto di polimorfismo, assente di natura nel mondo delle basi di dati relazionali e non, viene supportato attraverso il concetto di *Single Table Inheritance* (STI), che consiste nell'aggiungere uno o più campi nella tabella o nel documento volto a specificare quale classe tale entità rispecchia. Nel caso di Rails, l'omonimo STI è supportato con la sintassi *entity_id* ed *entity_type*, rispettivamente foreign key e nome della classe dell'entità, mentre nel caso di Hibernate è chiamato "*Table-Per-Class Hierarchy"*. Questo concetto è stato fondamentale per la costruzione delle risorse dell'applicazione e sarà ripreso successivamente. Rails utilizza come ORM la gemma "ActiveRecord", che fornisce un modello di dominio persistente tra oggetti in memoria e tabelle del database, la cui logica e i dati sono correlati tra loro anche dall'aggiunta di meccanismi di ereditarietà e delle associazioni, non trattate di default nell'Active Record Pattern. L'ActiveRecord di Rails è l'ORM di default per il framework ma non è l'unico, vi sono ORM anche per linguaggi NoSQL (ONRM) come MongoID e altri (Sez. 3.4.3).

3.3 Ruby on Rails: struttura applicativa

Dopo aver compreso i principi di Rails, viene adesso spiegato il funzionamento del framework, per poi analizzare la sua utilità per il lavoro svolto. L'ambiente di Rails è da installare, vi sono varie guide che suggeriscono il come e non sarà oggetto di trattazione. È fortemente consigliato lo sviluppo in ambiente UNIX, a causa dei ritardi negli aggiornamenti delle gemme per Windows da parte della comunità. È bene però evidenziare un comando del setup, ossia "*gem install rails*": Rails è a tutti gli effetti una libreria scritta in Ruby e si presenta con una interfaccia CLI molto potente e completa, in grado di eseguire task per qualsiasi esigenza. Anche la documentazione di Rails (fortunatamente) è ben fornita, per cui ogni comando seguito da un *--help* trova sempre riscontro soddisfacente. Tra i comandi più importanti vi sono:

- rails console | c
- rails server | s
- rails generate | g [model, controller, ...]
- rails destroy | d [model, controller, ...]
- rails dbconsole | db
- rails new APP_NAME

Quest'ultimo è proprio il primo comando da utilizzare nella creazione di una web app in Rails, perché genera una struttura di file e cartelle minimale come base applicativa e che è possibile avviare senza alcun tipo di configurazione aggiuntiva.



Nome	Dimensione	Tipo	Modifica
app	8 oggetti	Cartella	16.40
bin	7 oggetti	Cartella	16.41
config	12 oggetti	Cartella	16.40
db	1 oggetto	Cartella	16.40
lib	2 oggetti	Cartella	16.40
log	0 oggetti	Cartella	16.40
public	7 oggetti	Cartella	16.40
test	9 oggetti	Cartella	16.40
tmp	1 oggetto	Cartella	16.40
vendor	0 oggetti	Cartella	16.40
config.ru	130 byte	Testo	16.40
Gemfile	2,0 kB	Testo	16.40
Gemfile.lock	4,8 kB	Testo	16.41
package.json	63 byte	Programma	16.40
Rakefile	227 byte	Testo	16.40
README.md	374 byte	Testo	16.40

Figura 3.1: Struttura di cartelle di un'applicazione Rails.

Così come in C è presente il "Make", in Rails è presente "Rake", a supporto di una serie di compiti molto utili, come quelli legati al setup del database, del testing e altro ancora. La struttura delle cartelle è fondamentale per capire alcune scelte progettuali legate al namespacing. Le più importanti sono "app" all'interno della quale vi sono "model" e "view", "controller". La cartella "config" invece contiene i file di configurazione per i database e gli ambienti, ad esempio nel file *development.rb* vi è la configurazione specifica per l'ambiente di sviluppo (magari non si necessita di un vero e proprio mail server, e così via). La cartella "lib" ospiterà tutti i file scritti dal programmatore come librerie, nel caso trattato sono state costruite delle librerie a supporto dell'autenticazione. La cartella "spec" conterrà i file di testing, la scelta è stata quella di utilizzare Rspec come framework di testing, molto potente e completo. Le altre cartelle sono utili per altri fini ma non verranno trattate. Un file importantissimo è il *Gemfile*, in cui sono specificate tutte le gemme da utilizzare nel progetto e automaticamente recuperate da un repository principale (rubygems.org) a meno che non sia stata specificata una sorgente differente, mentre il *Gemfile.lock* contiene la lista delle dipendenze per ogni gemma del Gemfile, generato automaticamente. La cartella "db" contiene un file *schema.rb* che descrive in modo astratto la configurazione del database: come spiegato nella Sez. 3.4.3, Rails supporta diversi database relazionali tra cui MySQL, SQLite e PostgreSQL, intercambiabili tramite comandi e senza dover rigenerare la struttura del database stesso, grazie all'interpretazione di tale file. Nel file *config/database.yml* è presente la configurazione in termini di URL, IP, porta e/o altre credenziali per l'accesso al server DBMS o al file system associato[3], unico punto di modifica in caso di passaggio da un DBMS ad un altro, nessuna modifica applicativa grazie al supporto completo che Rails offre con i vari ORM.

Con la creazione dell'applicazione viene configurato un web server di default sulla porta 3000, che si avvia con il secondo comando della precedente lista, l'applicazione di default consisterà di una semplice pagina statica, a dimostrazione del funzionamento. Vi sono 3 modalità di funzionamento dell'ambiente: development (default), test e production (lanciato con "-e production", magari accompagnato da "-b IP -p PORT" per il deployment); il web server non è valido per la fase di testing, che viene avviata con il comando "rake test" per i test scritti con il

[3] Solo nel caso di DBMS che fanno riferimento al file system, come SQLite.

supporto del framework "minitest", embedded in Rails, mentre framework più avanzati come Rspec, Cucumber o Capibara i test si lanciano con comandi specifici delle gemme (Sez. 3.4.7). Il comando "rails console" avvia una console Ruby con le gemme di Rails precaricate: ad esempio, una libreria Rails può non avere senso nel solo contesto di Ruby (operazioni su controller o database), per cui è presente una console Ruby all'interno di Rails. Il comando "generate" invece è utile alla generazione delle risorse applicative, tra cui model, view e controller; in realtà, per la generazione delle viste server-side è necessario generare un controller con le "action" desiderate e ad ogni action corrisponderà una vista: ad esempio, con il comando "rails g controller home index" sarà generato un controller con nome *home* e azione *index*, oltre che una vista *index.html.erb*. I file *.erb sono file contenenti script in Ruby che saranno interpretati server side e renderizzati in HTML, se generatori di output, come per le pagine PHP o JSP. Si ricorda che lo scopo del lavoro è la realizzazione di un prodotto software efficiente e capace di scalare il più possibile senza problemi, per cui la scelta è stata quella di non usare questo tipo di viste, spiegato in Sez. 5.2. Infine, è supportata la distruzione delle risorse create con il comando generate, cosa non sempre presente nei framework (Angular 4 con la sua CLI non lo supporta) e soprattutto non banale da fare manualmente, poiché spesso con la generazione delle risorse vengono aggiornati automaticamente dei file di configurazione; i vari tool di sviluppo codice come Git, Mercurial o Team Foundation Server sono praticamente indispensabili. Tra i vari file di configurazione generati ve ne sono alcuni importantissimi, tra cui la configurazione del router nel file *config/routes.rb*: in questo file sono presenti i mapping tra URL e controller + action, ossia è possibile specificare quale azione di quale controller dovrà essere eseguita quando viene contattata una certa URL con quel verbo HTTP, altrimenti viene restituito una generica risposta HTTP con codice di stato 404. Ad ogni generazione del controller, viene aggiornato tale file di routing; ovviamente tutto quanto detto è anche possibile senza i comandi di Rails, ma comporta uno spreco di tempo inutile, per cui si agisce manualmente solo se strettamente necessario. Rails nel tempo si è evoluto e al giorno d'oggi la versione più recente è la 5.1, già dalla 5.0 vi sono stati profondi cambiamenti rispetto alla 4.2, legati ai Websocket e altre funzionalità integrate. La sezione seguente descriverà la progettazione e la realizzazione del backend applicativo che espone esclusivamente REST API in formato JSON.

3.4 Ruby on Rails per il backend

Come già accennato, il backend è costituito da un motore scritto in Rails che espone un set di API diversificate tra ambiente web e mobile, perché sono possibili diversi scenari di utilizzo, non necessariamente uguali tra loro; inoltre, ne è stato fatto il *versioning*, ossia sono state costruite per poter evolvere senza compromettere eventuali sistemi che utilizzano le vecchie versioni, insieme alle nuove. Tali API sono state pensate con la filosofia REST livello 3, secondo il modello di maturità di Richardson, e rispondono ad un solo formato dati, JSON. È stato modificato il meccanismo di sicurezza di default, offerto dalla libreria Devise, per autenticare e autorizzare il loro utilizzo senza compromettere il sistema stesso, dato che inventare meccanismi di sicurezza non è mai ottimo: sono state scritte due librerie locali chiamate AuthManager e Tokenizer. E ancora, dal punto di vista funzionale sono stati utilizzati ben 3 DBMS differenti: Postgres con l'estensione PostGIS a supporto dei dati spaziali, MongoDB per la flessibilità offerta da collezioni di documenti schema-less e infine Redis per il caching, gestione delle sessioni, i websocket e il pattern *publish/subscribe* nel meccanismo delle notifiche. Redis, database NoSQL di tipo *key-value*, è molto utile grazie alla possibilità di storage con scadenza temporale, per questo viene utilizzato nei meccanismi di sessione, upload di contenuti multimediali e processamento delle notifiche. In un'ottica di alto livello, sono stati pensati dei modelli per le risorse in gioco: utente, nutrizionista, negozio, istruttore e centro sportivo ed entità sportiva; altri per le componenti del sistema quali post, commento, foto, video, album (collezione dei due precedenti); vi è anche un meccanismo per la

gestione dei followers e following: il modello del social network è infatti basato su una relazione monodirezionale, come Twitter o Instagram. Il tutto è stato testato grazie a Rspec, un framework di testing BDD (Behaviour Driven Development), grazie al quale ogni singola risorsa applicativa è descritta in termini di azione compiuta e risultato aspettato.

3.4.1 API: Rack, JSON, versioning, namespacing e routing

Rack è un software modulare che fornisce delle API minimali per mettere in comunicazione il server web e il framework web. Rails è un framework basato su Rack, ma questo supporta qualsiasi altro framework scritto in Ruby. Non è un semplice modo di filtrare ed elaborare una richiesta, ma è l'implementazione del modello di progettazione basato su pipeline dei web server, separa in modo netto le diverse fasi di elaborazione di una richiesta, la separazione delle responsabilità è un obiettivo fondamentale di tutti i prodotti software ben progettati. Le diverse fasi possono essere autenticazione, autorizzazione, decorazione di una richiesta, caching e monitoraggio, sia in entrata che in uscita, ed infine esecuzione vera e propria di un compito. Ogni operazione è composta da uno o più middleware, termine che si riferisce a qualsiasi componente o libreria software che contribuisce all'esecuzione di un certo compito, un po' come i filtri delle applicazioni JavaEE. Di default un'applicazione è configurata con tutti middleware di base, ciò risulta essere molto pesante se in fin dei conti non vengono utilizzati. Per tal motivo Rails ammette la creazione di un'applicazione con un set ridotto di middleware, è necessario specificare il parametro "--api" nel comando *rails new*, ciò in automatico provvederà al setup di un'applicazione senza interfaccia web HTML ma solo per le API, a cui seguirà necessariamente la definizione di una strategia di output (come JSON o altro formato); in alternativa, è possibile creare un'applicazione con set completo e personalizzarla tramite l'aggiunta o rimozione di middleware predefiniti o creati appositamente. È infatti possibile creare dei moduli software e l'insieme dei middleware che un'applicazione Rails usa in un certo momento è consultabile con il comando *rake middleware*.

Nel caso della modalità "--api", nel controller principale dell'applicazione, ossia *application_controller.rb* (dispatcher per tutte le richieste) viene inserito un metodo di configurazione importantissimo: *respond_to :json,* il quale indica che l'applicazione accetterà soltanto quelle richieste con header "*Content-Type: application/json*". Certamente tale comando può essere modificato se si necessita un altro formato, infatti il formato della risposta è a carico del programmatore, Rails mette a disposizione metodi in ogni oggetto come *.to_json* o *.to_xml*, per serializzare il contenuto di un oggetto nella struttura equivalente. Rails è ampiamente utilizzato grazie alla vastità di gemme (librerie) disponibili per certi compiti, nel caso della serializzazione è stata utilizzata una gemma molto potente, chiamata Active Model Serializer (AMS), che offre supporto aggiuntivo per operazioni non include da Rails. Tale gemma è semplice da usare, è sufficiente generare il "serializzatore" di un modello, si tratta di una classe che estende la classe base *ActiveModel::Serializer* e che mette a disposizione un solo metodo, chiamato *attributes*, che riceve come parametri la lista delle proprietà da serializzare.

```ruby
class PhotoFullDetail < ActiveModel::Serializer
  attributes :photo_id, :photo_url, :photo_description

  def photo_id
    object.id.to_s
  end

  def photo_url
    object.photo.url
  end
end
```

Figura 3.2: Esempio di serializzazione con AMS.

Ad esempio, per la risorsa *Photo* con campi *id, url* e *description*, il serializzatore corrispondente avrà la struttura riportata in figura 3.2. Come facilmente intuibile, *object* è l'istanza della classe Photo quando il serializzatore viene invocato, mentre i due metodi *photo_id* e *photo_description* vengono utilizzati come chiave nella struttura JSON: ciò vuol dire che possono anche essere parte di una elaborazione più complessa. L'invocazione del serializzatore avviene nel *return* di una action all'interno del controller, come mostrato nella figura seguente.

```ruby
return render json: photo,
              serializer: Mobile::V1::PhotoSerializer::PhotoFullDetail,
              root: 'photo_detail'
```

Figura 3.3: Esempio di *return* per la serializzazione.

Al metodo *render* sono passati tre parametri: l'oggetto da serializzare con un certo formato, l'attore per la serializzazione e opzionalmente una keyword per la struttura dati (*root*). Con poche e semplici configurazioni è possibile costruire il formato di risposta delle API come serializzazione di oggetti.

Per la configurazione di namespacing, versioning e routing è necessario agire sia a livello del file *config/routes.rb*, sia mantenere una struttura di cartelle ben precisa, poiché Rails cercherà in automatico le risorse appartenenti ad un preciso spazio di nomi nell'omonima cartella, navigandone la gerarchia. Mentre il routing serve principalmente a dichiarare quale action di un certo controller deve essere invocata quando si contatta una URL, e dunque si tratta di una configurazione obbligatoria, namespacing e versioning servono principalmente per aumentare la qualità del software, sia in termini di parametri interni come manutenibilità, riparabilità ed evolvibilità, sia come parametri di qualità del software esterni come usabilità e scalabilità; essendo punti importanti del suddetto lavoro, le API sono state costruite tenendo conto di tali fattori. Il versioning in realtà può essere considerato come un sotto caso del namespacing. Più in dettaglio, per namespacing si intende la creazione di uno spazio dei nomi per delle risorse, a cui corrispondono potenzialmente controller in modo non congiunto. Supponendo di voler creare i namespace "*web*" e "*mobile*" per un controller "*MyController*", vi corrisponderà un sistema di cartelle e file omonimi, in *app/controllers/**web**/my_controller.rb* e in *app/controllers/**mobile**/my_controller.rb*, le cui classi del controller avranno come modulo il namespace stesso, nella forma *Web::MyController* e *Mobile::MyController*. Non è tutto, perché in automatico il namespacing può essere mappato su delle URL simili alla struttura di cartelle precedente, ma è bene vedere prima come funziona la creazione reale del namespace per capire come fare: le keyword del framework sono *namespace, constraints, scope, resources*.

Namespace è l'opzione più semplice e completa, aggiunge alle risorse specificate internamente un prefisso omonimo e creerà un modulo all'interno del quale risiede il controller.

```
namespace :admin do
  resources :users
end
```

```
Prefix Verb   URI Pattern                    Controller#Action
admin_users GET    /admin/users(.:format)     admin/users#index
            POST   /admin/users(.:format)     admin/users#create
admin_user  GET    /admin/users/:id(.:format) admin/users#show
            PATCH  /admin/users/:id(.:format) admin/users#update
            PUT    /admin/users/:id(.:format) admin/users#update
            DELETE /admin/users/:id(.:format) admin/users#destroy
```

Figura 3.4: Esempio di uso del *namespace*.

La configurazione riportata creerà un mapping tra URI, Controller e Action in modo completo, per cui il sistema di cartelle sarà *app/controllers/**admin**/**, in tal caso con solo *users_controller.rb* avente *Admin::UsersController* come nome di classe e URI */**admin**/**. Scope è più complesso, ma consente la personalizzazione della configurazione da utilizzare soltanto in caso di "fine tuning" delle API.

```
scope module: 'admin', path: 'fu', as: 'cool' do
  resources :users
end
```

```
Prefix Verb   URI Pattern                Controller#Action
cool_users GET    /fu/users(.:format)     admin/users#index
           POST   /fu/users(.:format)     admin/users#create
cool_user  GET    /fu/users/:id(.:format) admin/users#show
           PATCH  /fu/users/:id(.:format) admin/users#update
           PUT    /fu/users/:id(.:format) admin/users#update
           DELETE /fu/users/:id(.:format) admin/users#destroy
```

Figura 3.5: Esempio di uso di *scope* con le opzioni.

Come si evince dal confronto tra le figure 3.4 e 3.5, scope ha impatto sull'URI ma non sul sistema di file e cartelle. Scope solitamente non si usa da solo, ma con le opzioni *module, path* e *as*: *module* aggiunge il modulo alla classe (dunque scope e module insieme formano il namespace), *path* aggiunge i prefissi aggiuntivi alle URI ed infine la keyword *as* consente di rinominare gli helper links. Brevemente, gli helper links sono utili nelle viste, ad esempio per navigare verso la home page si usa l'helper *root_path,* per navigare un utente si usa *user_path(user_id)*, ecc., ma non essendo le viste server-side oggetto del presente lavoro, gli helper links non saranno approfonditi. Allo scope di solito segue anche una configurazione di default, ad esempio utilizzando il comando *defaults: {format: :json}*, che specifica JSON come formato di risposta delle azioni raggruppate sotto tale blocco scope. Resources crea un mapping di tipo CRUD per il modello specificato, mappando sui corrispondenti verbi HTTP le varie azioni dei controller, secondo la filosofia REST. In caso di configurazione diversa, come ad esempio non esporre tutte le action o modificare i verbi, è necessario scrivere i mapping manualmente.

La sintassi è semplice: http_verb '*uri_path*', to: '*path_to_controller/controller#action*', as: '*custom_helper_link*'. Esempio completo della configurazione per il set di API è riportato dalla figura seguente.

```
constraints subdomain: 'api' do   #api.domain.com
  #scope 'prefix' give URL prefix as 'prefix/...'
  scope 'auth', module: :auth, defaults: {format: :json} do
    scope 'mobile', module: :mobile, defaults: {format: :json} do
      devise_scope :user do
        # sessions
        post 'users/signin', to: 'users/sessions#create'
        delete 'users/signout', to: 'users/sessions#destroy'
        # passwords
        patch 'users/password', to: 'users/passwords#update'
        post 'users/password', to: 'users/passwords#create'
        # registrations
        post 'users', to: 'users/registrations#create'
        patch 'users', to: 'users/registrations#update'
        delete 'users', to: 'users/registrations#destroy'
        # confirmations
        get 'users/confirmations', to: 'users/confirmations#show'
        post 'users/confirmations', to: 'users/confirmations#create'
      end
    end
end
```

Figura 3.6: Esempio completo di routing.

La figura per brevità non riporta le API per il web, ma risulta comunque chiaro come sia possibile costruire un set di API diversificato tra i due ambienti, con sottodominio "api" (nella forma *api.domain.com*) che ovviamente sarà opportunamente autenticato per l'esterno e da configurare in deployment con un opportuno reverse proxy; inoltre, il versioning stesso risulta essere semplicissimo, basta usare una strategia tra *scope* o *namespace* con la sintassi desiderata. Non esiste una regola bel precisa per il versioning, solitamente si differenziano con stringhe numeriche (sequenziali) come "1, 2, 3…" oppure "v1, v2, v3…" e le URI corrispondenti saranno nella forma *api.domain.com/1/...* piuttosto che *api.domain.com/v1/...* e così via; ad ogni versione corrisponderanno controller e action separati, per cui i vecchi sistemi che usano le vecchie versioni delle API possono essere ancora supportati (finché non si elimina totalmente il supporto applicativo) mentre l'architettura può crescere senza limitazioni. I parametri di qualità del software riportati in precedenza sono ampiamente soddisfatti: il codice può essere facilmente modificato senza coinvolgere altre parti del sistema, dunque riparato o fatto evolvere senza grosse difficoltà; ogni parte è a sé stante, per cui la scalabilità è garantita.

3.4.2 Devise: autenticazione e autorizzazione con JWT

Devise è la gemma più utilizzata per la gestione dell'autenticazione nelle applicazioni Rails. È una soluzione estremamente flessibile e personalizzabile, salvo il fatto che il grado finale di sicurezza è a rischio se le modifiche effettuate creano falle nel sistema. Devise si basa su un'altra gemma, chiamata Warden. Warden fornisce un meccanismo di autenticazione per le applicazioni Ruby basate su Rack, come appunto Rails. Consente a tutti i middleware a valle di lui (invocati in modo sequenziale) di condividere un meccanismo di autenticazione comune: ogni parte può accedere alle risorse di autenticazione o richiederla, utilizzando la stessa logica di sicurezza. Warden costituisce dunque un meccanismo minimale, per cui Devise può effettivamente essere

considerata una sua estensione (o decorazione). In generale, vi sono due possibili alternative in ambito di sicurezza applicativa: costruire dal nulla un meccanismo di autenticazione, implementando tutta la logica e le strategie (autenticazione basata su username o email e password, su token, ecc.) sfruttando Warden per la gestione di autenticazione e autorizzazione, oppure utilizzare Devise, personalizzando le componenti laddove necessario, sfruttando un sistema di strategie già testate e ben consolidate. Devise è un software open-source mantenuto da una comunità di sviluppatori, per cui è evidente che la seconda scelta è la più ovvia, soprattutto perché la creazione delle strategie da zero comporterebbe un enorme spreco di tempo, oltre che il rischio di tralasciare dettagli di sicurezza.

Devise è una soluzione modulare, ogni modello creato in Rails (User, Admin, Editor, ecc.) che necessita del suo supporto deve essere configurato con dei moduli. Devise si appoggia ad un database per la gestione dei modelli in gioco; tra i moduli disponibili, alcuni sono:

- Database Autenticatable: modulo per la verifica sicura dei parametri di autenticazione rispetto alle informazioni salvate nel database (email, hashed password, validità account, blocco account, ecc.).
- Omniauthable: aggiunge il supporto OmniAuth al modello (Facebook, Google+, ecc.).
- Confirmable: aggiunge il supporto per la verifica di un account, necessaria per usufruire del servizio attraverso l'accesso autenticato; consente l'invio di email dopo la registrazione con le istruzioni di verifica. La verifica si basa su un token random e monouso. La configurazione ammette anche l'utilizzo dell'applicazione senza conferma per un certo intervallo di tempo, se desiderato.
- Recoverable: utile per reimpostare la password, attraverso istruzioni per email.
- Registerable: supporto completo a registrazione e cancellazione di una risorsa nel sistema.
- Timeoutable: nuova richiesta di autenticazione dopo un certo intervallo di inattività.
- Lockable: monitoraggio degli accessi illeciti in caso di troppi tentativi di autenticazione falliti. L'account in questione potrebbe essere bloccato per un certo tempo; senza timeout, è necessaria una strategia di sblocco come la verifica dell'email associata o altro.

Dei suddetti moduli, non tutti sono direttamente utili per il set di API creato. È necessario riscrivere la logica per la gestione e la scadenza delle sessioni: in altre parole, Devise si basa su un meccanismo classico di autenticazione (email e password) mentre la gestione delle sessioni è fatta usando delle informazioni contenute nei cookie, settati dopo la corretta autenticazione. Tali cookie contengono una chiave che sarà utilizzata da Warden per il recupero della sessione, verificando se corrisponde server side ad una sessione valida di autenticazione, altrimenti viene lanciata una eccezione all'interno di Warden che rimanderà alla pagina di login. Dato che il progetto in questione non fa uso di pagine generate dinamicamente server side, è necessario costruire un meccanismo API-compliant, per cui non serve avere un *redirect* ma un generico HTTP 401, che client side sarà gestito per mostrare la pagina o la schermata di autenticazione. Tale meccanismo custom si basa sulla generazione di un token JWT (JSON Web Token): una parte viene rilasciata al client dopo la corretta autenticazione, mentre l'altra è conservata lato server in Redis. La chiave data al client, similmente a quanto fatto in precedenza da Devise, serve adesso per la ricerca del corrispondente token nel database. Non è detto tale corrispondenza sia soddisfatta, o perché il client ha fornito un token non valido, o perché il token in questione fa parte di una sessione scaduta. La scelta del database per la gestione delle sessioni ricade su Redis perché è un DBMS NoSQL di tipo *key-value*, offre la persistenza dei dati in memoria secondo una logica di caching che lo rende molto veloce ed efficiente; offre anche la persistenza dei dati a scadenza temporale: durante il salvataggio dell'informazione è possibile specificare un valore numerico (in millisecondi) che corrisponde al tempo dopo il quale il dato viene automaticamente eliminato dal DBMS, dunque non è necessario

far scadere le sessioni dalla logica applicativa, dunque estremamente utile nella situazione appena descritta. Appare chiaro dunque che il salvataggio delle sessioni corrisponde al salvataggio di una parte del token con una restrizione temporale. Infine, per diminuire l'overhead dato dalla generazione del JWT è stato deciso di fornire un token monouso solo per le richieste HTTP che non sono GET, mentre per tutte le altre richieste può essere riutilizzato più volte, essendo una azione sicura e idempotente. È stato fatto anche l'*override* del modulo Timeoutable di Devise, perché ogni nuova richiesta deve in qualche modo fare il "refresh" della sessione in corso, in modo da avere una scadenza relativa all'ultimo istante di utilizzo del sistema: la sua gestione è molto semplice, ad ogni verifica positiva della sessione in corso, viene settato un nuovo timeout per il token conservato in Redis.

JWT (Json Web Token) è un open standard basato su JSON per la creazione di token di accesso per affermare un certo numero di diritti o consensi. Ad esempio, un server potrebbe generare un token che afferma la richiesta di accesso come amministratore per un client, il client può utilizzare tale token per dimostrare che è connesso ad una applicazione come amministratore. I token sono firmati con la chiave del server, in modo che entrambe le parti siano in grado di verificare che il token sia legittimo. I token possono anche essere autenticati e crittografati, JWT è base comune di altri due standard: JWS (JSON Web Signature) e JWE (JSON Web Encryption). JWT in generale è composto da tre parti: intestazione, payload e firma. L'intestazione contiene informazioni circa la costruzione del token (algoritmo, tipo, ecc.); il payload contiene dati applicativi personalizzabili e dati dello standard, i cosiddetti "claims" (timestamp di emissione, issuer, validità, ecc.); la firma tiene conto di intestazione e payload e il tutto viene codificato in Base64 (reversibile) per essere URL-safe nelle strutture dati delle richieste che lo contengono. Nel presente lavoro è stato utilizzato il JWS, trasferendo al client soltanto la firma, mentre il payload e l'intestazione sono conservati server side, in modo da scongiurare qualsiasi manipolazione.

Questa soluzione per la gestione delle sessioni potrebbe essere abbandonata in favore di una completamente stateless, eliminando l'utilizzo di Redis. Ciò comporta comunque degli svantaggi computazionali e applicativi da valutare. La soluzione stateless fa uso di un token che deve essere interamente rilasciato al client, dunque per payload sensibili sembrerebbe meglio usare JWE. L'unico compito del server sarebbe quello di verificare la sua integrità e la validità applicativa, ad esempio al suo interno potrebbero essere conservati dei dati come l'intervallo temporale entro cui risulta essere valido ed utilizzabile, dati di autorizzazione, dati di statistica (così un token utilizzato troppe volte in un certo intervallo di tempo, se poco realistico per un utilizzo umano, potrebbe essere rifiutato) e così via. Il primo svantaggio è legato al fatto che la verifica circa la validità della sessione (apparentemente) referenziata dal token segue sempre la verifica della firma digitale dello stesso, che dal punto di vista computazionale è oneroso e può essere una base di attacco DoS contro il sistema (Denial of Service). Invece, la precedente strategia usa Redis per verificare se innanzitutto la corrispondente parte del token è presente, in tal caso la sessione risulterebbe non scaduta e non manipolata, a cui seguirebbe la verifica della firma, che è sempre necessaria ma sicuramente non fallimentare: il client infatti riceve la firma del JWT, per cui una sua manipolazione non troverebbe mai una corrispondenza come payload server side; in altre parole, inviare al server un token manipolato vuol dire in automatico non poter attaccare il sistema delle sessioni. Dal punto di vista delle performance si tratterebbe solo di una ricerca in memoria del payload, ma l'architettura di caching in Redis è abbastanza performante da scongiurare qualsiasi problema prestazionale. Il payload sarebbe comunque in chiaro e rilasciarlo interamente al client vuol dire esporre potenziali dati applicativi, assolutamente da evitare. Al più si potrebbe utilizzare il JWE, ma resta un caso da considerare: il logout. In altre parole, come fare a capire che un utente non è più loggato? Al momento del login il server genera un token che fornirà al client e questo dovrà utilizzarlo quando si ripresenterà per usufruire di un servizio, ma nulla vieta il suo riutilizzo (replay), dato che nulla

server side può affermare che è già stato utilizzato. Una soluzione a ciò potrebbe essere considerare il logout come diretta conseguenza della scadenza di una sessione, quindi se un utente richiedesse esplicitamente la terminazione di una sessione il token sarebbe cancellato dalla cache locale del client ma sarebbe comunque valido in caso di utilizzo, per cui un utente malizioso potrebbe continuare ad usarlo per accedere ai servizi fintantoché questo token non scade, ossia la data che trasporta è considerata troppo vecchia rispetto a quella del sistema. Questo intervallo può anche essere molto ampio, dunque rischioso; se troppo breve, comporterebbe disservizi applicativi sul client, per cui bisogna attenzionare le scelte fatte. Per tali motivi si è preferito mantenere lato server un meccanismo di gestione delle sessioni, anche perché Redis supporta la clusterizzazione e, in caso di crescita del sistema, questa strategia non risulterebbe essere un fattore limitante per la scalabilità.

3.4.3 Architettura per la persistenza dei dati

In fin dei conti un'applicazione è uno strumento che consente all'utente finale di interagire in modo controllato con la base dati. Queste interazioni, che avvengono attraverso i meccanismi già descritti, consentono la visualizzazione, modifica, inserimento o cancellazione di un insieme di informazioni ammesse dal sistema, e ogni interazione produce dei risultati che possono essere direttamente o indirettamente utili. La manipolazione delle informazioni può far parte di una o più interazioni e i motivi di una scelta progettuale che distribuisca un gruppo di dati in più interazioni possono essere vari: la creazione di una risorsa può essere molto complessa, per cui si procede per passi successivi in mezzo ai quali si effettuano meccanismi di validazione (sintattici e semantici) server side, opzionalmente client side per una migliore *User eXperience,* non è così user-friendly validare alla fine e scartare tante informazioni già inserite dall'utente. Pur essendo possibile inserire meccanismi di salvataggio di queste, spesso si implementa solo client side per questioni di sicurezza, evitando di appesantire il sistema inutilmente. In fase di esplorazione dei servizi si ricorre spesso a meccanismi di paginazione, ossia ogni richiesta provvederà a recuperare un chunk di informazioni in quantità limitata, sia per motivi prestazionali che di fruizione del servizio: in quest'ultimo caso, la presentazione di migliaia di risorse tutte insieme comporterebbe una confusione per l'utente, meglio mostrarle pian piano e solo se lo desidera, per cui è bene non scaricare, soprattutto in un mondo in cui il consumo di risorse è importante (dati mobili, energia del dispositivo, ecc.). L'insieme di informazioni del social network in questione è abbastanza variegato e i DBMS di tipo SQL non possono dare pieno supporto per la gestione di tali dati. Per tal motivo la scelta ricade anche su DBMS NoSQL, tra cui MongoDB e Redis.

Redis in realtà verrà utilizzato a supporto di vari compiti grazie alle funzionalità che offre: si tratta di un DBMS per la persistenza di strutture dati in memoria e sul disco, utilizzato sia come database che come cache di contenuti e *message broker*, supporta meccanismi di tipo publish/subscribe, clusterizzazione e diversi livelli di persistenza nel file system. È stato introdotto nella sezione precedente come database a supporto dell'autenticazione, ma adesso sarà utilizzato per altri scopi. Postgres invece viene utilizzato come DBMS SQL per due motivi principali: la persistenza delle informazioni di autenticazione dell'utente e i dati geografici con l'estensione PostGIS, non è stato possibile sfruttare MongoDB poiché ha l'estensione per i dati geometrici ma non geografici: la differenza non è banale, soprattutto perché l'applicazione lavorerà su piccole distanze: l'approssimazione dei dati geografici a quelli geometrici comporta dei discostamenti dei risultati rispetto alla realtà non marginali, dunque errati e non proponibili. PostGIS invece offre un buon supporto per i dati geometrici, con un set di metodi molto interessante per operazioni di ricerca entro un certo raggio, distanza di un punto rispetto ad un altro, distanza tra due punti, ecc.

Per quanto detto in precedenza, la scelta di utilizzare MongoDB come DBMS principale ricade soprattutto nella sua natura *schema-less*, anche se le risorse in gioco necessitano tutte del supporto

geografico: ogni risorsa è posizionata geograficamente e tale posizione sarà oggetto di ricerca. Non solo, è stato pensato un meccanismo di "consigli e suggerimenti" tra i vari utenti (che in fin dei conti saranno dei semplici post e commenti ben messi in evidenza) che possono essere geolocalizzati, cosa ancora una volta utile per la ricerca applicativa. La necessità di avere supporto per i dati spaziali, unita alla volontà di avere delle strutture dati *schema-less*, comporta necessariamente l'utilizzo congiunto dei due database: sarà creata una unica entità logica ma i dati vengono fisicamente distribuiti tra i due DBMS, Postgres e Mongo, creando a livello applicativo un meccanismo di associazione forte tra le due parti, in modo che l'una non possa esistere al di fuori o in assenza dell'altra, in modo da costituire appunto un'unica entità logica. Dal punto di vista operativo è necessario agire attraverso gli appositi ORM e ONRM, ActiveRecord per Postgres e MongoID per MongoDB. Per utilizzarli congiuntamente vi sono due alternative: assegnare ad ogni risorsa l'identificativo univoco della controparte, e ad ogni operazione che coinvolge l'una recuperare la controparte per costruire l'oggetto completo, oppure affidarsi ad una libreria che faccia ciò di suo. La seconda sembra essere la scelta più ovvia, con la gemma "Active MongoID" (il nome stesso è l'unione dei nomi dei due ORM) e sostanzialmente crea in modalità *lazy loading* la relazione logica tra le due entità.

Più in dettaglio, dal punto di vista della programmazione si tratta di costruire due modelli separati, a cui corrispondono due classi diverse in Rails, uno per Postgres e uno per Mongo. La gemma consente il collegamento logico tra le entità dei due database attraverso l'utilizzo di due metodi: uno da utilizzare nell'entità di Mongo che faccia riferimento ad un record di Postgres e, per ogni tupla, un metodo che dal modello in Postgres faccia riferimento ad un documento di Mongo: tali metodi sono nella forma **_record(s)* e **_document(s)*, in base al tipo di relazione: *has_one_document, has_one_record, has_many_document, has_many_records, belongs_to_document* e infine *belongs_to_record*. LA differenza tra il *belongs_to_** e il tipo *has_** riguarda il tipo di navigazione tra le entità, se monodirezionale o bidirezionale, e quale tabella contiene la chiave primaria di quale entità: *belongs* specifica che la classe contiene la chiave esterna dell'entità, mentre *has* specifica che nella tabella dell'entità è presente la chiave a cui fa riferimento. La risorsa principale rimane comunque quella di Postgres, nel senso che le altre entità in gioco (recensioni, prenotazioni e altro ancora) eventualmente faranno riferimento all'entità di Postgres, in cui saranno conservati soltanto i dati spaziali, come le coordinate della risorsa, mentre tutto il resto è conservato nell'entità associata in Mongo. Nella figure successive sono riportati rispettivamente un esempio dei modelli in gioco e l'accesso alla risorsa corrispondente, navigabile in entrambe le direzioni.

```
class Resource::Pg::Center < ApplicationRecord
  include ActiveMongoid::Associations

  has_many_documents :resource_review, as: 'category'
  has_one_document :resource_mongo_center
  belongs_to :entity_user, class_name: 'Entity::User'

  validates :entity_user, :resource_mongo_center, presence: true

end
```

Figura 3.7: Modello di risorsa per Postgres.

```ruby
class Resource::Mongo::Center
  include Mongoid::Document
  include ActiveMongoid::Associations
  field :name, type: String
  field :city, type: String
  field :address, type: String
  field :telephone, type: String
  belongs_to_record :resource_pg_center
  validates :name, :city, :address, :telephone, :resource_pg_center, presence: true
  validates :name, length: { minimun: 5, maximum: 50 }
end
```

Figura 3.8: Modello di risorsa per Mongo.

La figura 3.7 mostra la classe che descrive il modello della risorsa "center" in Postgres (identifica il centro sportivo), mentre la figura 3.8 mostra la classe della risorsa "center" ma per Mongo. Per evitare confusione nei nomi e nei file delle classi si può modularizzare il tutto, facendo però corrispondere un analogo sistema di cartelle nel file system, come per quanto detto nella Sez. 3.4.1 per il namespacing in Rails: in tal caso le risorse si chiamano rispettivamente *Resource::Pg::Center* e *Resource::Mongo::Center*. La risorsa in Postgres fa riferimento a vari documenti di Mongo, più recensioni e un solo centro; per Mongo invece vi è il riferimento al record del centro di Postgres. È ovvio che si possono continuare ad utilizzare i metodi per le relazioni tra entità dello stesso database, ad esempio ogni centro ha un utente che lo gestisce (*belongs_to* fa riferimento ad un utente di Postgres), così come in Mongo si usano documenti embedded o con riferimento. Si possono notare delle differenze nella descrizione dei modelli: Postgres non ha alcun riferimento ai campi delle entità, mentre i modelli di Mongo si. Ciò accade perché Mongo non necessita di alcuna generazione di schemi per le collezioni, ma soltanto di un insieme di attributi per poter generare i corrispondenti metodi getter/setter; per Postgres invece è necessario, ma in Rails non si descrive nei modelli, bensì nelle cosiddette *migrations*: una migration è un documento generabile da CLI (*rails generate migration NAME*) che riporta una modifica nel database a livello logico (creazione tabella, aggiunta o rimozione colonne o indici, ecc.) e che sarà interpretata da Rails che ne riporterà le modifiche nel file *schema.rb*. In tale file è presente la descrizione logica del database dopo aver interpretato l'insieme delle modifiche di tutte le migration e che sarà utilizzata nella generazione delle tabelle nel database durante il suo setup; è bene non modificare direttamente il file dello schema poiché le modifiche saranno sovrascritte ad ogni nuova interpretazione delle nuove migrations (*rails db migrate*). Grazie a tale meccanismo Rails consente di effettuare modifiche al database senza doversi occupare dei dettagli della configurazione specifica in ogni modello.

Nelle immagini seguenti viene riportata la migration per il la risorsa *center* e il risultato della sua interpretazione nello schema logico.

```ruby
class CreateResourcePgCenters < ActiveRecord::Migration[5.0]
  def change
    create_table :resource_pg_centers do |t|
      t.st_point :lonlat, geographic: true
      t.references :entity_user, index: true, foreign_key: true
      t.timestamps null: false
    end
    add_index :resource_pg_centers, [:entity_user_id, :created_at]
    add_index :resource_pg_centers, :lonlat, using: :gist
  end
end
```

Figura 3.9: Migration per Postgres (*center*).

```
create_table "resource_pg_centers", id: :serial, force: :cascade do |t|
  t.geography "lonlat", limit: {:srid=>4326, :type=>"st_point", :geographic=>true}
  t.integer "entity_user_id"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index ["entity_user_id", "created_at"], name: "res_pg_centers_user_c_at"
  t.index ["entity_user_id"], name: "index_resource_pg_centers_on_entity_user_id"
  t.index ["lonlat"], name: "index_resource_pg_centers_on_lonlat", using: :gist
end
```

Figura 3.10: Descrizione logica della migration associata.

La figura 3.9 riporta la generazione di una tabella chiamata *resource_pg_center* con campi id (sott'inteso e automaticamente aggiunto), coordinate geografiche nella forma longitudine/latitudine, una chiave referenziata per l'utente che creerà la risorsa e dei timestamp (*created_at, updated_at*) utili per gli indici; il supporto per i dati spaziali in Rails è fornito dalla gemma "Active PostGIS Adapter". L'interpretazione della migration corrisponde alla generazione di una descrizione della tabella come riportato in figura 3.10, in cui è interessante notare il tipo di dato geografico generato: SRID 4326, punto geografico e indice spaziale associato con l'estensione PostGIS (using: gist). In tal modo è possibile sfruttare l'estensione GIS per Postgres per le operazioni su dati geografici. Un esempio di calcolo con dati geografici è quello della della distanza per una risorsa a partire da un punto identificato dalle sue coordinate geografiche, utile da mostrare come dettaglio della risorsa durante la ricerca:

```
def distance_from(from)
  _from_p = RGeo::Geographic.spherical_factory(srid: 4326)
                            .point(from[:lon], from[:lat]).as_text
  _to_p = RGeo::Geographic.spherical_factory(srid: 4326)
                          .point(lonlat.lon, lonlat.lat).as_text
  _select = "ST_Distance(gg1, gg2) as distance"
  _from = "(SELECT ST_GeographyFromText('#{_from_p}') AS gg1,
             ST_GeographyFromText('#{_to_p}') AS gg2) AS foo"
  array = Resource::Pg::Center.select(_select).from(_from).to_a
  return array.first['distance']
end
```

Figura 3.11: Esempio di metodo avanzato per la serializzazione.

In tal caso, invocando il metodo *distance_from* viene restituito il valore della distanza in metri dal punto *from*. In un'ottica più estesa, si può pensare all'invio delle coordinate da parte del client per ottenere la distanza della risorsa in questione rispetto alla posizione dell'utente. L'architettura completa per la persistenza dei dati sarà ripresa nella Sez. 6.5 che tratta il deployment applicativo.

3.4.4 Background processing: Sidekiq

Il normale flusso di interazione è composto dalla sequenza richiesta – elaborazione – risposta. Non tutte le azioni però possono essere soddisfatte in modo sincrono, soprattutto per motivi prestazionali o laddove non è richiesto un risultato immediato. Il processamento in background è alla base delle moderne applicazioni web e stand alone, sia client side che server side, favorendo molto la scalabilità. Tipici task sono il logging, monitoraggio di sistema, invio di email, gestione delle notifiche, processamento digitale dei dati, ecc., ossia tutte quelle azioni che non possono fornire una risposta in tempi ragionevolmente brevi. Non è l'ideale far attendere un utente per tempi lunghi a causa di applicazioni che in fin dei conti sembrano essere non reattive; si tratta anche di un problema prestazionale, i server elaboreranno la richiesta quando possibile, senza essere carichi inutilmente e magari lasciando spazio a quelle richieste con priorità più elevata o dall'elaborazione veloce (magari evitando situazioni di *starvation*). Laddove l'esecuzione di una azione comporta un

utilizzo di risorse computazionali oltre una certa soglia è bene processarla in modalità asincrona, se il tipo di azione lo consente. Di solito non è così evidente, ma uno dei vantaggi principali per cui processare in modo asincrono una richiesta è la possibilità di riprovare ad eseguire un task in modo sicuro, specialmente nell'invio di email o notifiche, processamento immagini e altro ancora.

Per Rails sono disponibili vari framework a supporto del background processing, tra cui Sidekiq, che offre supporto con code di priorità, schedulazione e rischedulazione (in caso di fallimento) automatica o manuale dei task e monitoraggio real-time tramite cruscotto. Sidekiq si appoggia a thread diversi per eseguire ogni job, è necessario creare una classe apposita chiamata "worker" (*EmailWorker*, *NotificationWorker*, ecc.) che contiene un solo metodo (*perform*) all'interno del quale vanno specificate le azioni che il task deve compiere, con eventuali parametri passati, grazie al modulo *Sidekiq::Worker*. In modo dinamico, il modulo di Sidekiq provvederà a generare due metodi statici di classe: *perform_async* e *perform_in*, che servono rispettivamente alla schedulazione asincrona con immediata esecuzione quando possibile e all'esecuzione asincrona non prima di un intervallo specificato come ultimo parametro del metodo. Tali metodi dovranno essere chiamati nel punto del codice in cui si vuole avviare il task nella forma *WorkerClass.perform_**, Sidekiq provvederà in modo sincrono alla serializzazione dell'azione in Redis in una coda gestita, per poi recuperare l'azione ed eseguire quando opportuno. La configurazione di Sidekiq richiede che vengano specificati i parametri per la connessione a Redis e per il thread-pool, ma Sidekiq non dipende da Rails e può essere utilizzato in qualsiasi applicazione Ruby, il che vuol dire che ogni sistema Sidekiq può essere facilmente inserito all'interno di un container e replicato (Sez. 6.5). Sidekiq ammette sia la creazione di code di priorità che lo sharding, ossia la creazione di più client che esistono come entità e code di priorità separate. Tra le *best practices* rientrano la creazione di task idempotenti, transazionali, piccoli e semplici: per semplicità si intende il non serializzare lo stato di una azione se non necessario, ma soltanto i parametri come identificativo di una risorsa che sarà poi recuperata dai database. In tal modo aumenta anche l'efficienza del processamento stesso poiché il meccanismo di schedulazione si appoggia a quello *publish/subscribe* di Redis; nel momento in cui le azioni di un task sono molteplici è utile dividerle in più task, a patto che tra di essi non vi sia alcuna dipendenza temporale: ad esempio una certa azione potrebbe comportare l'invio di una notifica applicativa oltre che l'email, in tal caso è bene separare l'azione logica in due task separati, in diverse code di priorità. Grazie a Sidekiq è stato possibile costruire il meccanismo di invio email e notifiche per il social network in questione, ma si può estendere a tante altre situazioni: ad esempio, il rilascio di una recensione da parte di un utente potrebbe essere processato in background dopo aver superato i controlli di un sistema per la ricerca di parole offensive, spam e altro ancora.

3.4.5 Action Cable: framework real-time per Rails

Nel creare applicazioni web dinamiche i vari framework mettono a disposizione dei meccanismi molto potenti per offrire i servizi all'utente finale, basati principalmente sul paradigma richiesta/risposta veicolate sul protocollo HTTP, ormai alla base delle interazioni web. È possibile ottenere facilmente contenuti vari, in tipo e in dimensioni, con buone prestazioni attraverso ricche interfacce, grazie ai motori di rendering odierni e le funzionalità offerte dai framework client side. Eppure per la maggior parte delle applicazioni il classico paradigma richiesta/risposta su cui si basa il protocollo HTTP non riesce a soddisfare tutte le situazioni, soprattutto quando si ha bisogno di ricevere un contenuto da parte del server, specialmente se in tempo reale, perché di per sé il server non è abilitato ad effettuare alcun trasferimento di dati fintantoché non riceve una richiesta associata. Situazioni di questo tipo non sono così fuori dal normale: aste e giochi online, modifica di documenti condivisi in tempo reale, cruscotti di monitoraggio, chat, notifiche e così via.

Per ovviare alla mancanza di iniziativa da parte del server il meccanismo più semplice (ma anche meno performante) è quello del polling, in cui il client periodicamente richiede al server l'invio di nuovi contenuti, se presenti. Tale modello è però insostenibile al crescere del numero di utenti, si verificherebbe una situazione analoga ad un attacco DoS, poiché i server hanno comunque una capacità limitata e possono gestire fino a qualche migliaio di client contemporaneamente, peggio ancora includendo le normali richieste; comunque si tratterebbe di un meccanismo poco efficiente e in caso di grosse applicazioni non favorisce affatto la scalabilità. Esiste una variante, il long polling, per cercare di diminuire l'effetto dell'elevato numero di richieste, che però va incontro a problemi non banali, tra cui la perdita del concetto di "real time" e l'avere un backend che rischia picchi di traffico in caso di sincronizzazione delle richieste nel tempo. A prescindere, vi sono sprechi cospicui in caso di assenza di nuovi contenuti, perché le richieste (complete di intestazioni) devono essere comunque fatte dai client proprio perché il server non può iniziare una comunicazione, con annessa risposta potenzialmente vuota. La vera soluzione al problema è data dai Websocket, un meccanismo di comunicazione bidirezionale (pensato in casa Google) che nasce da una normale connessione HTTP poi veicolata su un diverso protocollo (websocket, appunto). Il fatto che parta tutto da una normale richiesta HTTP vuol dire essere capace di attraversare senza problemi proxy, firewall e quant'altro, ecco dunque il motivo di una tale scelta progettuale al posto di un nuovo protocollo basato esclusivamente su TCP; tale connessione lato server può essere autenticata attraverso un meccanismo come i cookie, trasportati poiché il client contatterà lo stesso dominio applicativo a cui appartengono, oppure attraverso un meccanismo come HTTP Authentication o altro ancora. Quello dei websocket è un meccanismo di comunicazione bidirezionale, con payload sia testuale che binario, con un sistema di keep-alive leggero tra le parti che non ha sprechi o inefficienze in termini di dati trasportati, come invece accade per una richiesta HTTP con tutti gli header e magari nessun payload utile in risposta. Non è oggetto del documento analizzare nel dettaglio i websocket, però questi sono ormai parte nativa della release 5 di Rails con il framework Action Cable, un sistema per la comunicazione real-time basato sul protocollo dei websocket.

Action Cable fornisce un'interfaccia semplice per scrivere codice sia server-side che client-side (attraverso CoffeScript, compilato poi in Javascript), basa il suo funzionamento sul paradigma publish/subscribe con vari DBMS, di cui Redis è preferito per la sua alta efficienza in tale meccanismo nativo. Il server di Action Cable può essere montato come parte del server di Rails, sfruttando il concetto di "socket hijacking" nella gestione dei websocket, ma in production mode è più efficiente se lanciato in modalità stand alone come mostrato in figura 3.12. È importante sottolineare un concetto alla base del funzionamento di Action Cable, perché tale framework infatti non si comporta in modalità RESTful (come accaduto fino ad ora per le action dei controller) bensì funziona in modalità RPC (Remote Procedure Call), ossia per ogni metodo pubblico dichiarato per un canale vi è un corrispondente metodo lato client che ne consente la sua invocazione remota. Action Cable gestisce una connessione per websocket (istanza della classe *Connection*, parte del modulo *ApplicationCable*) e ogni consumatore (client) può essere connesso a più websocket contemporaneamente. All'interno di ogni connessione vi possono essere più canali, il canale è l'unità logica di lavoro (*NotificationChannel*, *AppearanceChannel*, ecc.) e il consumatore dopo aver stabilito connessione diventa un subscriber del canale. Ogni canale viene identificato da una stringa alfanumerica e in generale può essere broadcaster di zero o più entità, si tratta di un collegamento di tipo publish/subscribe in cui ciò che viene trasmesso al suo interno è inviato a tutti gli ascoltatori, ma se il canale è nominato in base ad un identificativo univoco si stabilisce effettivamente una relazione 1:1 tra server e client, che è utile nel caso del meccanismo di notifica, solitamente inizializzato dopo la corretta autenticazione.

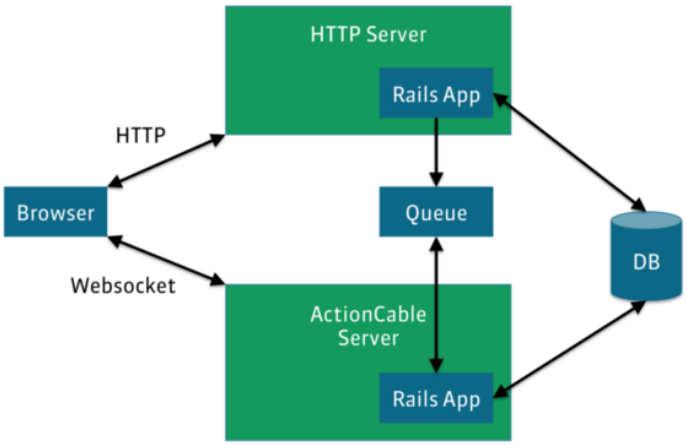


Figura 3.12: Architettura di Action Cable (stand-alone server).

```ruby
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
    end

    private
      def find_verified_user
        if verified_user = User.find_by(id: cookies.encrypted[:user_id])
          verified_user
        else
          reject_unauthorized_connection
        end
      end
  end
end
```

Figura 3.13: Esempio di classe *Connection*.

La struttura di Action Cable per Rails necessita dunque della definizione di una classe *Connection*, in cui si può inserire la logica necessaria condivisibile da ogni connessione, tra cui il meccanismo di autorizzazione delle connessioni in ingresso. Nella figura 3.13 è riportato un esempio di autorizzazione in base al metodo *current_user*, che deve restituire l'utente autenticato (con Devise o altro meccanismo), il metodo *identified_by* serve ad identificare e recuperare la connessione nei momenti successivi, al di fuori della classe connection, per tutte le istanze dei canali. L'assunzione di base che Action Cable fa è che in altro modo sia già stata correttamente effettuata l'autenticazione del client e che l'utente possa essere identificato in modo sicuro (in tal caso con dei cookie firmati), così da poter identificare tutte le connessioni di quell'utente attraverso il suo identificativo ed eventualmente cancellarle se non più autorizzato.

```ruby
# app/channels/web_notifications_channel.rb
class WebNotificationsChannel < ApplicationCable::Channel
  def subscribed
    stream_from "web_notifications_#{current_user.id}"
  end
end
```

Figura 3.14: Esempio di classe *Channel* per le notifiche web.

Action Cable suggerisce l'uso di una classe *Channel*, da cui tutti i canali possono ereditare una logica comune (solitamente assente poiché specifica di ogni canale) a cui seguirà poi la definizione dei singoli canali in base alla logica applicativa (notifiche, chat, ecc.). Nella figura 3.14 è mostrato un esempio di classe per la gestione real-time delle notifiche web, il cui canale è una stringa contenente un identificativo univoco (*user_id*) per cui si tratta a tutti gli effetti di uno stream "a uno". Una nuova istanza di tale classe sarà creata ad ogni connessione da parte di un subscriber e sarà tenuta in vita fin quando il consumer non si disconnetterà, che vuol dire anche giorni, per tal motivo è bene non mantenere al suo interno informazioni che possano causare spreco di risorse. I metodi classici sono *subscribe* e *unsubscribe*, ma altri possono essere aggiunti e richiamati quando necessario.

```ruby
# Somewhere in your app this is called, perhaps from a NewCommentJob
ActionCable.server.broadcast \
  "web_notifications_#{current_user.id}", { body: 'New things!'}
```

Figura 3.15: Esempio di broadcast per le notifiche web.

L'invio di un payload avviene semplicemente chiamando il metodo statico di classe *broadcast*, passando come parametri l'identificativo del canale e il payload desiderato per una certa azione, come mostrato nella figura 3.15. Ciò può essere fatto in un punto opportuno del codice, come all'interno di una callback *after_save*, ossia eseguita dopo il salvataggio di una risorsa nel database, o all'interno di un metodo per il processamento asincrono (come in Sidekiq), Action Cable provvederà poi allo streaming dei contenuti attraverso i websocket, appoggiandosi a Redis.

Pur essendo Action Cable un buon framework, purtroppo la parte client è integrata in Rails con CoffeScript, un linguaggio compilato in Javascript che lega fortemente le viste al rendering server-side. Ciò non fa al caso del presente lavoro, per cui l'utilizzo di Action Cable in uno scenario differente da quello offerto dal suo creatore non è così semplice: non è possibile accedere ai canali così facilmente attraverso una normale libreria client per websocket, è necessario lavorare a basso livello e riprodurre i comandi scambiati dalle componenti interne di Action Cable per poterlo sfruttare come motore API-only. Più in dettaglio, è stata estrapolata una libreria per Javascript (chiamata *actioncable-js*) che può essere inclusa in un progetto Javascript, ma purtroppo non esiste un'analoga libreria per C#, per cui nel caso delle applicazioni mobile scritte con Xamarin è necessario agire a basso livello. Il set di comandi è stato estrapolato analizzando la libreria nel modulo "subscriptions" (*lib/action_cable/connection/subscriptions.rb*). È presente uno switch case con i casi *subscribe*, *unsubscribe*, *message*, ad ognuno dei quali corrisponde l'invocazione di metodi rispettivamente per la connessione, disconnessione e processamento del payload. Si può allora costruire manualmente una struttura richiesta/risposta per la comunicazione tra client e server basata su JSON, la cui struttura dei messaggi di controllo è riportata in figura 3.16, mentre in figura 3.17 vi è il formato di invio e ricezione messaggi.

```
{
  "command":"message",
  "identifier":"{\"channel\":\"MyChannel\"}",
  "data":"{\"to\":\"user1\",\"message\":\"hello \",\"action\":\"chat\"}"
}
```


Figura 3.16: Struttura dei messaggi di controllo (subscription).


```
{
  "command":"subscribe",
  "identifier":"{\"channel\":\"MyChannel\"}"
}
{
  "identifier":"{\"channel\":\"MyChannel\"}",
  "type":"confirm_subscription"
}
```


Figura 3.17: Struttura dei messaggi per la comunicazione.

Nella figura 3.16 si può notare il meccanismo RPC su cui Action Cable basa il suo funzionamento, infatti nella classe che implementa il canale sarà richiamato il metodo identificato dal parametro *action* del payload *data*. Nonostante tale *hardcoding* possa sembrare pesante e difficile, è necessario per poter sfruttare Action Cable come motore API-only senza un client specifico.

3.4.6 Processamento foto e video

L'idea alla base del social network non è soltanto consentire la ricerca geolocalizzata di risorse sportive, altrimenti non sarebbe affatto un social network. Tra i contenuti che gli utenti possono creare rientrano anche la presenza di foto e video. Per Rails esistono varie gemme che offrono supporto alla creazione di contenuti con dati digitali, tra cui *Paperclip* e *Carrierwave*, quest'ultima molto più completa e flessibile della prima, grazie alla quale è stato possibile esporre ogni meccanismo ai client attraverso un set di REST API. La pubblicazione di un contenuto digitale passa attraverso tre fasi: upload del contenuto, processamento e salvataggio. Essendo tali contenuti pesanti rispetto ad un semplice post testuale, azione follow/unfollow, ricerca o altro, sono buoni candidati per il processamento in background.

Carrierwave si basa sul concetto di *uploader*, un agente per il salvataggio e il recupero di un file da una sorgente che può essere basata su file system o rete (come Amazon S3 per lo storage). Si possono creare vari uploader, all'interno di ognuno dei quali è possibile specificare le validazioni in termini di dimensioni, formato dati ammesso, tipo di processamento da effettuare (scaling, rotazione, compressione, ridimensionamento, ecc.) e tipologia di formati in output da conservare. Per ogni caricamento effettuato viene conservata la foto originale con i processamenti di base più una versione del file per ogni comando *version* specificato: ogni metodo *version* racchiude un gruppo di processamenti ulteriori che vengono automaticamente eseguiti. Scenari di applicazione sono vari, basti pensare ad esempio che per ogni immagine di un album può esservi un'anteprima associata da mostrare, facilmente generabile con un comando *version*.

```
version :thumb do
  process resize_to_fit: [200, 200]
end

version :small_thumb, from_version: :thumb do
  process resize_to_fit: [150, 150]
end
```

Figura 3.18: Esempio di *version* con processamento.

La figura riporta un esempio di generazione di due anteprime per le foto, una 200x200 e una 150x150, importante notare che il processamento può essere a sé stante o in cascata, nella figura mostrata si tratta di processamento in cascata, tramite il comando *from_version* (*small_thumb* è il risultato del processamento compiuto dal metodo *thumb*).

È compito del programmatore associare tale file ad un modello, se necessario, poiché Carrierwave si occupa del solo trasferimento. La scelta progettuale è stata quella di creare dei modelli in Mongo (*photo_album* e *video_album*) a cui associare un insieme di elementi multimediali (*photo* e *video*). Grazie a tali modelli è possibile includere altri elementi come descrizione, commenti, hashtag, ecc. Dal punto di vista strutturale i modelli di Mongo sono analoghi a quanto già descritto per la persistenza dei dati (Sez. 3.4.3), bisogna solo richiamare Carrierwave per l'uploader specifico: il metodo è *mount_uploader*, a cui passare il nome della proprietà che richiama il file multimediale e l'uploader stesso.

```
module MediaType
  class Photo
    include Mongoid::Document
    include Mongoid::Timestamps::Short
    include ActiveMongoid::Associations

    mount_uploader :photo, SimplePhotoUploader
    validates :photo, presence: true

    # ActiveRecord Bridging
    belongs_to_record :entity_user, class_name: 'Entity::User'
    validates :entity_user, presence: true

    embedded_in :photo_album, class_name: 'MediaCollection::PhotoAlbum'
  end
end
```

Figura 3.19: Modello per la risorsa *photo*.

Come facilmente intuibile, ritornano tutti i concetti già espressi nelle sezioni precedenti: è presente il mapping tra un record di Postgres e una risorsa di Mongo (*belongs_to_record*) e ogni oggetto *photo* non vive al di fuori di una risorsa *photo_album*, per cui è un documento embedded all'interno di un altro. Lo scenario è analogo per i file video, per brevità non viene riportato.

Dopo aver costruito i modelli per i file multimediali è necessario creare delle API opportune da esporre ai client per l'invio delle informazioni relative al file e il file stesso come dato binario. Ciò viene fatto attraverso la creazione di un controller (uno per il web e uno per il mobile) con le azioni specifiche del caso per ottenere la lista degli album, lista delle immagini o video di un album, aggiunta, modifica o rimozione di una singola entità. Le API per l'aggiunta di un file multimediale sono state necessariamente strutturate in due step successivi: in un primo momento il client dovrà inviare le informazioni associate al post (descrizione dell'immagine, hashtag, posizione geografica, ecc.) e lato server, se tutto valido, sarà creato un insieme di metadati associato al contenuto multimediale; la seconda fase prevede invece il caricamento del file multimediale come dato

binario. Il motivo di una tale scelta progettuale riguarda alcune inefficienze che possono verificarsi server side: la presenza di errori di validazione dei contenuti trasmessi comporterebbe l'annullamento dell'intera azione. Il trasferimento di una foto magari non peserebbe molto, ma quello di un intero album, di un video o un gruppo di questi sarebbe un enorme spreco di risorse in termini di tempo, banda e computazione. L'invio delle informazioni associate al dato multimediale senza alcun trasferimento binario consentirebbe invece la validazione senza sprechi. A supporto di tale meccanismo è stato utilizzato Redis, i dati validati saranno conservati con una scadenza temporale di pochi minuti (nel caso peggiore 10 minuti sono più che sufficienti per il caricamento di una immagine, ne servono almeno 60 per il caricamento di un video) e fino ad un massimo di N risorse temporanee, per "autodifesa"; ad ogni insieme di metadati viene assegnato un identificativo univoco restituito al client, costruito dall'unione dell'id dell'album per il quale va aggiunta la risorsa in gioco e un valore random. Se entro i tempi dati viene completato il caricamento del file, il controller provvederà a recuperare i metadati attraverso l'identificativo che il client dovrà presentare mandando il file binario e creerà un oggetto effettivo (*photo*) con associato il dato multimediale, grazie al lavoro svolto dall'uploader. Ovviamente il tutto sarà autenticato come per le altre API. Con tale meccanismo è stato possibile costruire un set efficiente di API per il caricamento dei file multimediali.

Il resto del set di API comprende vari endpoint per l'esplorazione di queste risorse, la loro modifica e cancellazione. È stato riutilizzato lo stesso meccanismo descritto in Sez. 3.4.1 per l'esplorazione delle risorse, con un insieme di metadati come lista di oggetti JSON che descrivono il singolo oggetto *photo* o *video*, dal quale il client può recuperare l'URL per scaricare il contenuto digitale dal server. L'URL che viene fornito però non è quello diretto della risorsa, proprio per non esporre la struttura del file system a utenti maliziosi. Si tratta di un URL che contiene l'id della risorsa e che un'apposita azione del controller provvederà a cercare in Redis come associazione del file system: ad esempio, se il client volesse scaricare il contenuto della foto *13b421a7* per l'album *7ab2365* effettuerà una richiesta per la URI *www.domain.com/album/7ab2365/photo/13b421a7* e il controller verificherà se in Redis è memorizzata una corrispondenza per la foto *13b421a7* all'interno dell'album *7ab2365*. È vero che c'è dell'overhead aggiuntivo, sia per storage che per computazione, ma si acquisisce un altro vantaggio non trascurabile: la possibilità di disaccoppiare la risorsa logica con il contenuto stesso, i client farebbero richiesta GET per la stessa URI mentre lato server la risorsa può essere spostata, rinominata, e così via, e ciò che consentirebbe anche di aggiungere un driver per lo storage in rete, piuttosto che su file system locale.

3.4.7 Testing e sviluppo BDD

Il testing è una delle principali fasi nel ciclo di vita dello sviluppo software, il cui scopo è quello di verificare se le specifiche di un sistema, stabilite in fase di analisi e design, sono state raggiunte o meno. Serve principalmente ad individuare le carenze di correttezza, completezza e affidabilità delle componenti software durante e dopo lo sviluppo. Testare le funzionalità del sistema è effettivamente qualcosa di molto vago, infatti vi sono varie tipologie di testing mirate a specifiche parti o a tutto il sistema, perché anche se i singoli componenti funzionano secondo quanto, il sistema ottenuto integrandoli potrebbe non esserlo. Pertanto è sempre necessario testate il sistema completo. Quello del testing in realtà è un argomento molto vario e non sarà oggetto di discussione in questo documento, però sarà descritta la sua utilità circa lo sviluppo del backend in Rails. Gli sviluppatori Rails spesso si trovano coinvolti in un ciclo di sviluppo software che segue la metodologia AGILE, in cui la parte di testing non è considerata una fase a parte o da compiere alla fine del ciclo di sviluppo, bensì è fortemente integrata e compiuta in più fasi. Le metodologie di sviluppo software e di testing sono tante, tra queste Rails supporta bene le metodologie BDD e TDD (Behaviour / Test

Driven Development), grazie al suo framework di testing integrato chiamato *minitest*, che consente di verificare se le specifiche del prodotto software sono soddisfatte. Tale framework embedded in realtà non è così potente per l'integration testing, per cui la comunità ha sviluppato due gemme, *Rspec* e *Cucumber*, per il testing completo, compresi Unit, Functional, System, Database, Routes, View, Job e Integration Testing, insomma test per tutti i componenti principali del framework.

Rspec è una gemma che come le altre va inserita nel Gemfile, dopo la sua installazione il framework viene automaticamente configurato per aggiungere dei file a supporto del testing ad ogni generazione di componenti software (controller, modelli, ecc.) dalla CLI di Rails. La struttura di testing del framework rispecchia la gerarchia di cartelle, come riportato nella figura seguente.

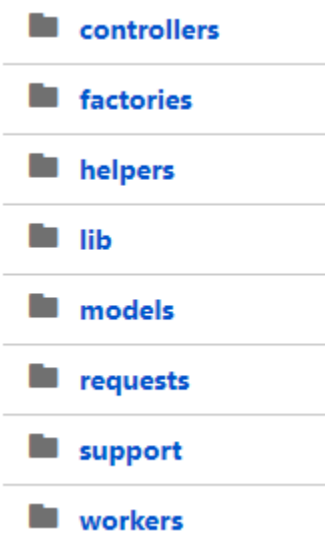


Figura 3.20: Struttura cartelle di Rspec.

Tutte le cartelle (tranne *factories*) contengono una sottostruttura come quella data da eventuali moduli e namespace delle singole risorse, la corrispondenza tra la classe creata e il file di testing è 1:1. All'interno di ogni file va descritto il comportamento che la singola unità deve avere. Tale comportamento differisce in base a cosa si testa, in un controller ad esempio ci si aspetta che contattando un endpoint in assenza di informazioni per l'autenticazione si riceva una risposta HTTP con codice 401, per un modello si possono testare i criteri di validazione relativamente alla presenza di risorse associate, numero massimo di caratteri in un parametro, e così via. L'integration testing viene svolto descrivendo il comportamento di una interazione richiesta/risposta all'interno dei file della cartella "requests", in cui viene lanciata una istanza del server applicativo in modalità *test* (per tenere puliti i database e il file system in automatico sarà fatta la pulizia dell'output di ogni azione). La cartella factories contiene le varie factory: una factory è un file di supporto al testing per la generazione di oggetti mock, solitamente decorati con dati fake. Rspec, ma qualsiasi altra gemma di testing, è spesso accompagnata da un'altra gemma per la creazione di tali oggetti mock (la più usata è *FactoryGirl*) e un'altra per la generazione di dati random (la più usata è *Faker*). Far lavorare insieme le tre gemme è abbastanza intuitivo: con i metodi messi a disposizione da FactoryGirl si creano degli oggetti mock decorati con dei dati falsi, ottenuti dalla gemma Faker; tali oggetti saranno utilizzati per testare le funzionalità dei controller, modelli, librerie custom e altro. Di seguito un esempio di testing completo.

```ruby
FactoryGirl.define do
  factory :entity_user_without_person, class: 'Entity::User' do
    @p =  Faker::Internet.password
    email { Faker::Internet.email }
    password @p
    password_confirmation @p
    confirmed_at Time.now.utc
  end
end
```

Figura 3.21: FactoryGirl per il modello *User*.

```ruby
FactoryGirl.define do
  factory :entity_person_without_user, class: 'Entity::Person' do
    first_name { Faker::Name.first_name }
    last_name { Faker::Name.last_name }
    born_city { Faker::Address.city }
    current_city { Faker::Address.city }
    biography { Faker::Lorem.paragraph 5 }
  end
end
```

Figura 3.22: FactoryGirl per il modello *Person*.

Il codice totale per il testing delle funzionalità è circa quattro volte il codice applicativo del backend, per brevità viene riportato un solo caso circa la validazione di una risorsa "persona". Le due figure precedenti mostrano le factories associate ai modelli *user* e *person* per la creazione degli oggetti mock, mentre la figura di seguito mostra il testing vero e proprio delle risorse.

```ruby
RSpec.describe Entity::Person, type: :model do
  before(:each) do
    @user = FactoryGirl.build(:entity_user_without_person)
    @person = FactoryGirl.build(:entity_person_without_user)
  end

  it 'creates a valid person' do
    @user.entity_person = @person
    @person.entity_user = @user

    expect(@person.valid?).to be true
    expect(@person.save!).to be true
  end

  it 'creates person without user' do
    expect(@person.valid?).to be false
    expect{@person.save!}.to raise_error Mongoid::Errors::Validations
  end
end
```

Figura 3.23: Rspec testing per l'entità User.

Il testing in Rspec riguarda una risorsa identificata dal suo tipo (*type: model*, *type: controller*, ecc.) ed Rspec ammette l'esecuzione di callback prima o dopo certi eventi, come l'inizio (o la fine) di ogni caso di test, utile per l'inizializzazione dello scenario o la creazione di risorse associate. Per facilitare gli scenari agli sviluppatori di test, si raggruppano in blocchi *describe 'main_scenario'* e *it 'scenario'*. Dal punto di vista dell'esecuzione cambia l'invocazione delle callback citate pocanzi: ad ogni blocco *it 'scenario'* corrisponde l'invocazione delle callback nel blocco *before(:each) / after(:each)* mentre per ogni blocco *describe* l'invocazione di *before(:all) / after(:all)*.

Nel primo scenario di test ci si aspetta che la risorsa *@person* sia valida e che vengano correttamente salvate nel database tutte le risorse associate; nel secondo scenario invece ci si aspetta che la risorsa *@person* non sia valida perché non è stata assegnata la risorsa *user*, e che un eventuale salvataggio dia un errore da parte del database, grazie ai vincoli imposti dalla gemma che fa da bridge tra i due DBMS.

Nel lavoro svolto il testing server side è stato compiuto per ogni action di ogni controller, soprattutto in quei meccanismi custom come autenticazione, upload di contenuti multimediali e processamento in background delle risorse (email, notifiche, ecc.), per tutti i modelli, gli endpoint, le librerie e infine è stato eseguito l'integration testing, per simulare le singole richieste dei client, cosa interessante poiché può essere sfruttata come base logica per la costruzione dei meccanismi di interazione con il server lato client, oltre che per rilevare eventuali falle di sicurezza applicative.

4. Xamarin

Quando si parla di Xamarin spesso si pensa alla programmazione cross-platform, ma non è tutto. Xamarin infatti è un'azienda produttrice di software, fondata da ex ingegneri del progetto Mono che hanno dato vita a Mono per Android e MonoTouch. Si tratta di implementazioni basate su C# delle specifiche di Common Language Infrastructure (CLI) e di Common Language Specifications (meglio conosciuto come lo stack .NET - letto "dot net"). È vero che grazie a codice C# i programmatori possono creare applicazioni native per Android, iOS e Windows, ma non si tratta soltanto di programmazione mobile cross-platform, perché Xamarin è stato estero per le piattaforme Desktop macOS e Windows. L'azienda è stata poi acquistata da Microsoft nel 2016, ma Xamarin non può essere definita come una semplice tecnologia, perché offre supporto completo alla produzione del software in termini di sviluppo, testing, distribuzione, monitoraggio e apprendimento delle nuove caratteristiche del framework, per tenere sempre aggiornati gli sviluppatori. Ma perché nasce una soluzione cross-platform di tale tipo? Dopo aver compreso i motivi alla base della nascita di Xamarin, sarà chiaro perché è stata la scelta indiscussa per la creazione delle applicazioni mobile del social network in questione.

4.1 Come e perché nasce Xamarin

La routine di un programmatore consiste nell'analizzare un problema, dividerlo in unità funzionali, per formulare la soluzione più efficiente ed efficace possibile, magari basandosi su opportuni pattern di programmazione e design del codice. A ciò si aggiunge la necessità di rendere il codice funzionante, performante e magari di bell'aspetto, eliminare bug e fornire un comportamento predittivo a quanto creato. A volte è necessario doversi imbattere in riscritture parziali o totali di codice, per renderlo altamente scalabile e manutenibile. Eppure vi sono vari limiti e problemi non banali da risolvere nel mondo della programmazione odierna, ad esempio la scrittura dello stesso codice per una piattaforma differente, in un altro linguaggio di programmazione, basato a sua volta su interfacce applicative (API) differenti. Spesso tale riscrittura per piattaforme completamente diverse, come Android vs iOS, può avere i suoi benefici soprattutto se l'applicazione è già popolare o serve a consolidare un servizio già esistente. Eppure una questione del genere comporta delle conseguenze non trascurabili, sia dal punto di vista dello sviluppatore che aziendale: vuol dire la necessità di dover acquisire le competenze opportune per operare su tutte le piattaforme in questione, mentre per un'azienda significa dover assumere personale appositamente qualificato. Nella riscrittura del codice per un'altra piattaforma possono sorgere vari interrogativi, come la classica questione di rendere il codice nuovo un esatto parallelo del vecchio per facilitarne il mantenimento nel tempo, anche se la tentazione di riorganizzare il tutto in modo migliore è spesso molto forte, soprattutto se nel vecchio erano presenti pattern non ottimali che è stato bene non cambiare per non stravolgere l'esistente. Eppure, tanto più le due versioni divergeranno, tanto più sarà difficile mantenerle coerenti in futuro. A prescindere, ogni riga di codice di una piattaforma sarà da duplicare per l'altra, il che significa doppio del lavoro. Sarebbe ottimo poter scrivere un programma generico portabile su più macchine, che è il motivo alla base della nascita dei linguaggi di programmazione di alto livello e interpretati da virtual machine, ma è lo stesso concetto alla base della programmazione "multi piattaforma", potente e utile ai programmatori.

Nonostante i classici PC Desktop o Laptop continuano ad esistere, perché molte operazioni rimangono fondamentalmente impossibili senza questi dispositivi, gran parte del computing si consuma ormai sui personal device: ricerca rapida di informazioni, chiamate, messaggistica e svago, fruizione di servizi in real-time e on-demand, social networking, condivisione di informazioni e tanto altro, basati su interazione al tocco con tastiera (virtuale) solo se necessaria. Tablet e smartphone sono ormai i "vincitori" sul mercato rispetto ad altri dispositivi ormai scomparsi. Due famiglie principali si dividono gran parte della scena mondiale: Apple, con sistema operativo iOS rigorosamente per iPhone e iPad; Google, con Android, sistema operativo basato sul kernel Linux, funzionante su una varietà di dispositivi. Vi sono altre piattaforme come Blackberry, Windows Phone e Windows Mobile 10, queste ultime due di casa Microsoft, non così popolari sul mercato come le prime citate, eppure tutto parte proprio da qui. Microsoft ha fornito delle API sotto il nome di Windows Runtime (WinRT) basate sul framework .NET, ciò significa che le applicazioni per PC desktop, computer portatili, tablet e telefoni possono condividere gran parte del loro codice. Successivamente nasce la piattaforma universale di Windows (Universal Windows Platform - UWP), una versione del runtime di Windows che costituisce la base per Windows 10 e Windows 10 Mobile.

Come detto, mirare singolarmente alle piattaforme comporta vari ostacoli, tra cui diversi ambienti di sviluppo, interfacce (API) di programmazione e linguaggi di programmazione, ma anche diverse interfacce grafiche: le varie piattaforme usano modi diversi per presentare l'interfaccia grafica (GUI) e la gestione dell'interazione con il dispositivo, così come la gestione della navigazione tra applicazioni e pagine, diverse convenzioni per la presentazione dei dati, ecc. Tutto ciò influenza e non poco le scelte di designers e programmatori, soprattutto perché un'applicazione disinstallata non verrà facilmente scaricata una seconda volta ed è molto facile perdere utenti a causa di applicazioni con pessime UI e UX. I programmatori oggi sono abituati a lavorare in un ambiente di sviluppo integrato (IDE). Tali IDE sofisticati esistono per tutte le piattaforme, ma naturalmente sono diverse: Xcode su Mac per lo sviluppo di iOS, Android Studio per Android, Visual Studio su Windows per lo sviluppo di applicazioni dello stesso target. Ogni piattaforma mobile è basata su un diverso sistema operativo, ognuno dei quali è a loro volta basato su diverse API, per cui in fin dei conti le differenze vanno ben oltre i semplici nomi dei widget. Infine, gli sviluppatori non hanno vasta scelta sul linguaggio di programmazione da utilizzare: Objective-C per iOS, Java per Android, C# per Windows. L'idea di base è avere un'architettura per condividere lo stesso codice, basato sullo stesso linguaggio di programmazione, per tutte e le piattaforme. Magari non si condivide il codice per la creazione delle UIs, ma sarebbe ottimo poter almeno condividere il codice applicativo. La domanda sorge spontanea: quale linguaggio di programmazione?

4.2 La soluzione C# e il Framework .NET

Nonostante le risposte possano essere (apparentemente) molte, la predominante è C# in .NET. Svelato da Microsoft nel 2000, C# è un linguaggio di programmazione abbastanza nuovo, o almeno, se confrontato con Objective-C e Java. In un primo momento C# sembrava essere un linguaggio orientato agli oggetti, piuttosto semplice, fortemente tipizzato e imperativo, certamente influenzato da C++ e Java, ma con una sintassi molto più pulita. Inoltre, la prima versione di C# forniva già supporto per proprietà e eventi, che risultano essere tipi di membri particolarmente adatti per la programmazione di interfacce grafiche. C# ha continuato a crescere e migliorare negli anni, aggiungendo supporto alla programmazione generica, funzioni lambda, LINQ e alle operazioni asincrone che l'hanno trasformato in un linguaggio di programmazione multi paradigma, prestandosi molto bene a pattern di programmazione come quello imperativo, dichiarativo o funzionale. E Xamarin? Poco dopo l'annuncio di Microsoft su .NET nel giugno 2000, la società

Ximian ha avviato un progetto open source (Mono) per creare un'implementazione alternativa del compilatore C# e del framework .NET per Linux; nel 2011, dopo l'acquisto di Ximian da parte di Novell, i fondatori diedero vita a Xamarin, che tutt'ora contribuisce alla versione open source di Mono ma che ha anche adattato Mono come base di soluzioni mobile cross-platform. A Marzo 2016, Microsoft ha acquisito Xamarin con l'obiettivo di condurre lo sviluppo mobile per la più ampia comunità di sviluppatori Microsoft: una sola lingua per più piattaforme. Inizialmente Xamarin trattava le tecnologie di compilazione su tre set di base di librerie .NET: Xamarin.Mac, che si è evoluto dal progetto MonoMac, Xamarin.iOS, che si è evoluto da MonoTouch, Xamarin.Android, che si è evoluto da Mono per Android (MonoDroid). Con "Xamarin" dal punto di vista tecnico si intende anche l'insieme di queste tre librerie, le quali sono costituite da versioni .NET delle API native di Mac, iOS e Android. I programmatori che utilizzano queste librerie possono scrivere applicazioni in C# che saranno mappate sulle API native della singola piattaforma. Tutto ciò ha un grosso vantaggio: è possibile sfruttare qualsiasi libreria di classi del vasto framework .NET per la costruzione di un prodotto software. Mica male! Alla base dell'indirizzamento di più piattaforme con un singolo linguaggio di programmazione vi è la possibilità di condividere il codice tra le applicazioni, ma queste devono essere progettate per tale scopo.

Risulta evidente che il codice applicativo che fa uso di risorse di sistema, di rete o di storage, normali funzionalità considerate parte di API del sistema operativo, può essere ricondotto a librerie di classi .NET: in tal modo questo codice è effettivamente indipendente dalla piattaforma, può quindi essere isolato e addirittura inserito in un progetto separato.

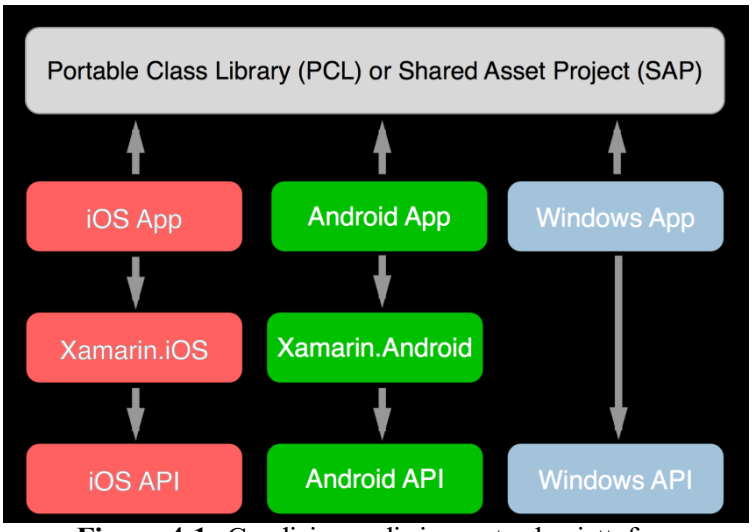


Figura 4.1: Condivisone di risorse tra le piattaforme.

Come mostra la figura 4.1, l'architettura dei progetti Xamarin può essere SAP o PCL: un codice separato come asset condiviso (SAP - Shared Asset Project) costituisce semplicemente codice e altri file di risorse accessibili da altri progetti, mentre una libreria di classi portabili (PCL - Portable Class Library) racchiude tutto il codice comune in una libreria di collegamenti dinamici (DLL), a cui i singoli progetti possono far riferimento, poi compilato in codice nativo. Nel caso dello sviluppo mobile ciò significa che è possibile creare un singola soluzione progettuale formata da quattro sotto progetti, di cui tre rappresentano lo specifico progetto della piattaforma, sia che si tratti di risorse ad accesso comune (SAP) che di progetto condiviso (PCL). Nella soluzione, il progetto contenente codice specifico per la piattaforma contiene il codice che effettua chiamate sia verso il progetto comune (indipendentemente da SAP o PCL) ma anche verso le librerie Xamarin che implementano le API native della piattaforma: ad esempio, l'uso della fotocamera passa attraverso delle API specifiche della piattaforma, per cui è necessario scrivere un codice che non è condivisibile tra le parti, ma può essere racchiuso in una interfaccia per poter essere richiamato dalle varie parti

dell'applicazione, sfruttando i concetti di *dependency injection* e *inversion of control* (Sez. 4.3.2). In realtà le librerie Xamarin.iOS e Xamarin.Android non aggiungono novità sostanziali al complesso, sono semplici wrapper delle API native delle piattaforme citate. Nel creare l'applicazione iOS, il compilatore C# genera l'Intermediate Language (IL) come di consueto, utilizzando poi il compilatore Apple per Mac per generare il codice macchina nativo di iOS come se tutto derivasse direttamente dall'Objective-C. Analogamente per Android, dopo aver generato l'IL, questo viene eseguito su una versione di Mono nel dispositivo, senza interpellare in alcun modo il motore Java, ma comportandosi come se tutto derivasse da Java. È bene precisare che, per applicazioni dalle esigenze specifiche della piattaforma, le sole librerie Xamarin.iOS e Xamarin.Android costituiscono una eccellente soluzione allo sviluppo di applicazioni in linguaggio C#, è infatti garantito l'accesso alle API native con tutto il potere e le responsabilità che ciò implica.

4.2.1 Xamarin Forms

Xamarin.Forms è stata la vera rivoluzione nel campo dello sviluppo cross-platform. Si tratta di un insieme di librerie che consente di scrivere codice per l'interfaccia utente, questo può essere compilato per i dispositivi iOS per i programmi che eseguono su iPhone, iPad e iPod Touch, Android per i programmi che eseguono su telefoni Android e tablet generici, la piattaforma Windows universale (UWP) per applicazioni che si eseguono in Windows 10 o Windows 10 Mobile, ma vi sono anche le API di WinRT per Windows 8.1 e per Windows 8.1 Mobile. I progetti per codice specifico della piattaforma in un'applicazione di tipo Xamarin.Forms sono tipicamente piccoli, spesso costituiti da sole parti con codice di avvio o per l'implementazione di uno specifico servizio strettamente dipendente dalla piattaforma (fotocamera, GPS o altro). Il PCL o SAP contiene la maggior parte del codice dell'applicazione, incluso il codice dell'interfaccia utente. Xamarin.Forms è un insieme di librerie DLL (Dynamic Linked Library) le cui principali sono:

- Xamarin.Forms.Core.dll
- Xamarin.Forms.Xaml.dll
- Xamarin.Forms.Platform.dll
- Xamarin.Forms.Platform.iOS.dll
- Xamarin.Forms.Platform.Android.dll
- Xamarin.Forms.Platform.WinRT.dll
- Xamarin.Forms.Platform.WinRT.Phone.dll
- Xamarin.Forms.Platform.WinRT.Tablet.dll
- Xamarin.Forms.Platform.UAP.dll

In dettaglio, l'implementazione dell'interfaccia grafica è delegata alle librerie Xamarin.Forms.Core e Xamarin.Forms.Xaml; a seconda della piattaforma, Xamarin.Forms.Core utilizzerà una delle librerie Xamarin.Forms.Platform. Queste librerie sono per lo più formate da una collezione di classi (chiamate rendering) che trasformano gli oggetti di interfaccia utente di Xamarin.Forms negli oggetti d'interfaccia utente specifici per la piattaforma. Xamarin.Forms supporta XAML (eXtensible Application Markup Language), un linguaggio XML sviluppato all'interno di Microsoft come linguaggio di markup. XAML non è stato definito per le sole interfacce utente, ma ormai da molto tempo è il motivo per cui lo si usa e tutt'ora viene molto utilizzato per definire le GUI in Xamarin.Forms.

La libreria Xamarin.Forms è vasta, ognuno dei suoi *widget* standard (ad esempio *Label*, *Button*, *Switch*, *Slider*) è implementato da una classe di render nativa, Xamarin.Forms dunque non sostituisce Xamarin.iOS e Xamarin.Android, bensì si integra con loro.

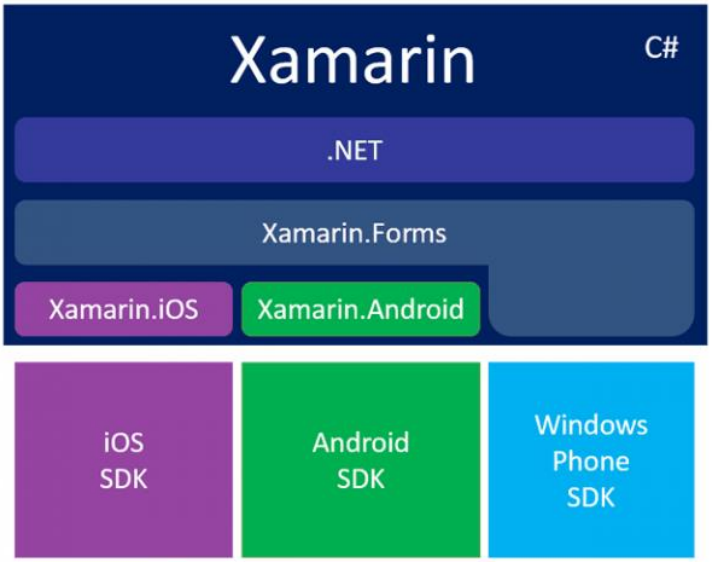


Figura 4.2: Stack di Xamarin per l'ambiente mobile.

Come si può vedere dalla figura sopra, Xamarin si appoggia alle API date dagli SDK nativi. Per il social network in questione la scelta è stata quella di utilizzare Xamarin.Forms per la costruzione delle applicazioni Android e iOS, grazie alla presenza di UIs e funzionalità semplici.

4.3 Xamarin per il mobile

Tutto ciò causa una complessità non irrilevante: un'azienda che vuole scommettere su più piattaforme deve impiegare per principio tre team di programmatori differenti, in cui ciascuna squadra sarà qualificata e specializzata in un particolare linguaggio di programmazione. Trattandosi di una startup con limitate risorse finanziarie, la soluzione trattata vuole evitare proprio questa scelta, creando una buona soluzione software e risolvere gli altri problemi sopra discussi. Come si può dedurre da quanto detto, i vantaggi sono da ricercare nella creazione di due applicazioni per due diverse piattaforme a partire da una unica soluzione software, senza dover sviluppare due prodotti diversi e con la possibilità di avere un team ristretto: a parità di tempistiche e supponendo i tempi di sviluppo uguali per ogni piattaforma, un programmatore Xamarin è N volte più produttivo di un programmatore per le singole piattaforme, vantaggioso per la startup.

In questa sezione sarà descritto l'utilizzo di Xamarin per lo sviluppo delle applicazioni mobile (Android e iOS) che saranno client delle risorse esposte dal backend, descritto nel Capitolo precedente. Si focalizzeranno soltanto gli aspetti principali e le strategie adottate per adempiere determinati obbiettivi di sviluppo, tralasciando tutti i dettagli di configurazione dell'ambiente. La progettazione di un'applicazione mobile è qualcosa di non facile, soprattutto perché, come spiegato in Sez. 2.1, è più un fattore psicologico e di impatto iniziale per l'utente che di performance, che sicuramente giovano. Per entrambe le piattaforme, l'applicazione consiste di un insieme di schermate principali: registrazione, accesso e navigazione dei servizi che a sua volta si divide in una schermata per la home, profilo utente, ricerca delle risorse e notifiche; infine, varie schermate "accessorie" come impostazioni, visualizzazione estesa di un profilo utente o di una entità sportiva, post, recensioni, lista dei commenti, pubblicazione di un post o recensione, ecc.

4.3.1 Il modello MVVM

L'esperienza di uno sviluppatore Xamarin consiste nella creazione di UI in XAML e di codice che opera su tali interfacce, ma al crescere di queste insorgono problemi di mantenimento che riguardano soprattutto l'accoppiamento stretto tra i controlli delle UI e la business logic locale, rendendo difficili mantenimento e testing. Col passare del tempo si è capito che la costruzione di un software scalabile e manutenibile passa attraverso la scrittura di moduli che svolgono un ben preciso compito e la comunicazione tra essi è possibile grazie alle interfacce che ognuno espone. Dal principio di separazione delle responsabilità nascono un insieme di architetture software tra cui il modello MVC (Sez. 2.2) in cui si separa la logica di presentazione delle interfacce dal codice applicativo e dalla logica di gestione dei dati. MVC è nato intorno agli anni '80, ma non è l'unico modello presente e soprattutto non è offerto da tutti i framework. Alcuni infatti lasciano al programmatore piena libertà circa la scelta di un'architettura di organizzazione del codice. L'architettura MVVM (Model-View-ViewModel) nasce da questa voglia di flessibilità, ha modernizzato il pattern MVC per le moderne GUI. MVVM separa il codice nel modello per i dati, la vista come interfaccia utente e gestione dell'input e il ViewModel, collante tra le due parti, che trasmette i dati tra i modelli e la vista, quasi come il "vecchio" controller, solitamente contesto di binding all'interno di cui richiamare i servizi che comunicano con l'esterno.

L'architettura di sviluppo di un'applicazione mobile varia in base al tipo di applicazione stessa, vi sono applicazioni che richiedono l'utilizzo di servizi web, alcune si basano su long-running tasks e altre sulla visualizzazione di contenuti offline. In ogni scenario, proprio per gli stessi principi discussi nella Sez. 2.2, si può ritrovare (perché appunto non è obbligatorio) il modello MVC o MVVM, anche per app stand alone, ma è compito del programmatore creare e separare fisicamente le componenti Model, View e Controller o ViewModel. Il ViewModel non può essere considerato un sostituto del Controller, pur essendo un intermediario della comunicazione tra View e Model, a supporto del binding (monodirezionale o bidirezionale) tra le due parti, attraverso una istanza del ViewModel chiamata BindingContext. L'architettura scelta per la costruzione delle applicazioni in Xamarin è MVVM, con un insieme di modelli che richiamano gli stessi presenti server-side, le viste, come insieme di interfacce navigabili dall'utente e infine i ViewModel, responsabili di fornire i dati alle viste, aggiornarle in caso di cambiamento di questi, attraverso il meccanismo del binding, ed eventualmente comunicare alla base dati server-side le modifiche. A supporto di ciò vi sono un insieme di file che si comportano da gestori di un certo servizio, solitamente per la comunicazione con il backend e l'esecuzione di long running tasks: ad esempio il ViewModel della home page (HomePageViewModel) potrebbe acquisire contenuti di molteplici risorse attraverso l'utilizzo di più servizi (CurrentUserService, NewsFeedService, CenterService, ecc.).

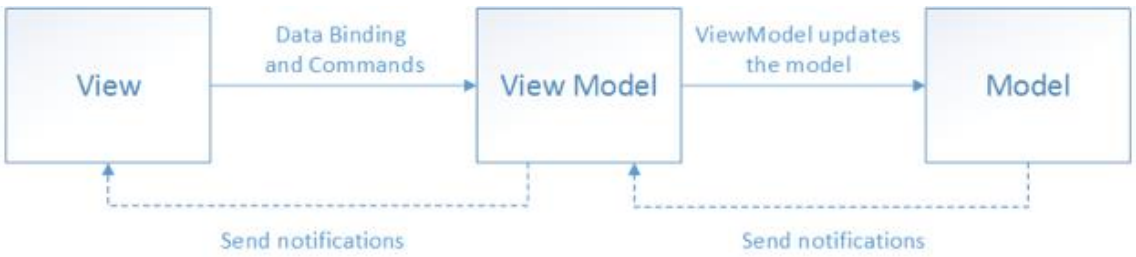


Figura 4.3: Modello MVVM.

Ad alto livello, la vista "conosce" il ViewModel e questo conosce il modello, ma il modello non è a conoscenza del ViewModel, che a sua volta non è consapevole della vista. Come conseguenza di ciò, gli aggiornamenti avvengono attraverso un meccanismo di notifiche (binding). Il ViewModel isola la vista dal modello e consente a questo di evolversi indipendentemente dalla vista, dunque

una modifica dell'interfaccia non comporta una necessaria modifica del modello. Dopo aver creato un progetto Xamarin.Forms di tipo PCL, nella soluzione condivisa si possono aggiungere tutte le componenti necessarie, con interfaccia grafica o meno. Ogni schermata corrisponde ad una coppia di file .xaml e .cs, nel primo è bene impostare l'insieme di *widget* (Button, Label, Layout, ecc.) in modo statico, mentre la dinamicità viene data da quanto scritto in C# e riportato nel secondo file, come insieme di handler che gestiranno l'input utente. L'insieme di servizi, modelli, API e tutto ciò che non ha una corrispondenza grafica va scritto in file C#, mentre le interfacce grafiche possono essere scritte in XAML, utile per i designer e meno verboso per i programmatori. I progettisti e gli sviluppatori possono lavorare autonomamente e contemporaneamente sui loro componenti durante il processo di sviluppo, il testing delle singole unità è semplificato ed è possibile lavorare con dati e oggetti *mock*.

4.3.2 Dependency injection in Xamarin

In ingegneria del software la dependency injection è una tecnica in cui una entità fornisce de dipendenze ad un'altra entità. Per dipendenza si intende un oggetto che sarà utilizzato per certi scopi da parte dell'entità che lo riceve. Concetto alla base di tale tecnica è che l'entità riceve la dipendenza dall'esterno senza doverla procurare da sé con metodi di costruzione come "new" o static factories. L'intento della tecnica è il disaccoppiamento tra le parti, favorendo il mantenimento del software. La dependency injection è una forma concreta della più ampia tecnica di *inversion of control*: piuttosto che essere il codice di basso livello a chiamarne uno di alto livello, il codice di alto livello può invocarne uno di basso livello, l'opposto di quanto si verifica nella programmazione procedurale. L'entità ricevente delega la responsabilità di procurare le dipendenze ad una entità "esterna", l'*injector*, ma non lo invoca direttamente. Piuttosto, è l'injector che risolve le dipendenze e invoca l'entità, la quale è a conoscenza dell'interfaccia della dipendenza, in modo da separare responsabilità non solo di creazione ma anche di utilizzo. I framework che supportano tale pattern sono tantissimi, in Xamarin la dependency injection è fondamentale per richiamare l'implementazione di una interfaccia con codice specifico della piattaforma in una soluzione PCL. L'interfaccia infatti è il punto condivisibile e applicabile tra le varie piattaforme, dunque utilizzato nel codice condiviso, ma la sua effettiva implementazione è da sviluppare nel codice non condiviso. Mentre la soluzione PCL è a conoscenza solo dell'interfaccia, ogni istanza di implementazione del codice non condiviso è recuperata dal container (l'injector) e fornita quando necessario.

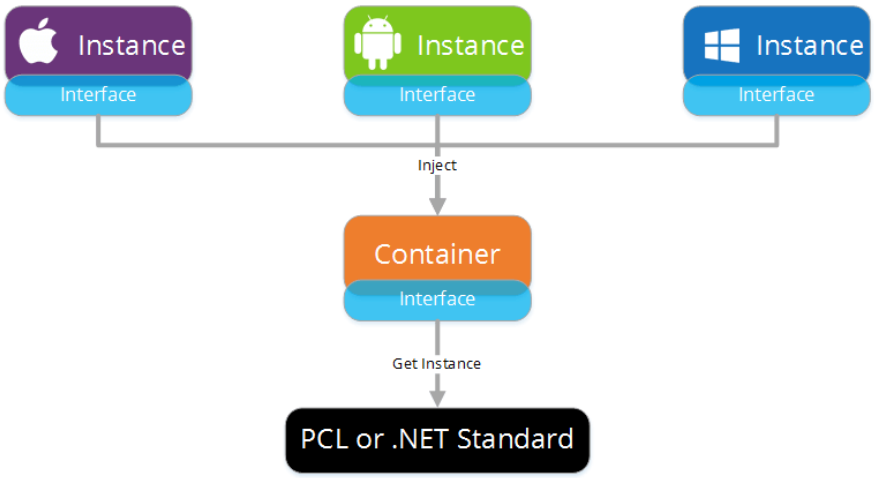


Figura 4.4: Dependency injection in Xamarin Forms.

Per sfruttare la tecnica della dependency injection è necessario definire una interfaccia nella soluzione PCL, e le relative implementazioni in ogni piattaforma con annessa registrazione, mentre la risoluzione avviene quando si utilizza la dipendenza. La registrazione è indispensabile per il container, senza la quale non saprebbe quale deve essere l'istanza di una implementazione per l'entità che la necessita. La risoluzione consiste nella creazione dell'oggetto che implementa l'interfaccia, da fornire poi all'entità ricevente. Mentre la risoluzione, come già spiegato all'inizio della sezione, è qualcosa di non esplicito, la registrazione dell'implementazione deve esserlo.

```
[assembly: Xamarin.Forms.Dependency(typeof(StoreCredential))]
namespace SportAdvisor.Droid.Session
{
    class StoreCredential : IStoreCredential
    {
```

Figura 4.5: Dependency injection in Xamarin Forms.

La figura 4.5 riporta la registrazione del componente *StoreCredential* in ambiente Android, ovviamente lo stesso dovrà essere fatto in tutte le altre implementazioni per le piattaforme utilizzate: si è dichiarata la presenza di un componente StoreCredential che è implementazione dell'interfaccia IStoreCredential, sarà compito del container IoC di Xamarin procurare tale dipendenza quando richiamata. Xamarin mette a disposizione il *DependencyService*, una classe che offre due metodi, *Register<T>* e *Get<T>*, rispettivamente per registrare un componente (alternativa alla registrazione tramite assembly) e per il suo recupero in caso di utilizzo. Si possono utilizzare vari container di terze parti per la dependency injection, ma il DependencyService di Xamarin è sufficiente per la maggior parte dei casi. Come intuibile dal nome dell'interfaccia nell'esempio sopra riportato, la dependency injection è stata utilizzata per la conservazione sicura di informazioni nel dispositivo a supporto di autenticazione e autorizzazione, oltre che per recuperare tutti i servizi di comunicazione con il backend.

4.3.3 Autenticazione e autorizzazione: Xamarin Auth

L'autenticazione è il processo di ottenimento delle credenziali di identificazione di un utente, quali nome e password, e la successiva convalida nei confronti di un'autorità. Se le credenziali sono valide, l'entità che ha inviato le credenziali è considerata un'identità autenticata. Una volta che un'identità è stata autenticata, un processo di autorizzazione determina se l'identità ha accesso a una determinata risorsa. Tale autorizzazione può essere stabilita dal backend con l'assegnazione di un ruolo all'identità dopo l'autenticazione oppure essere implicita nell'identità che si è autenticata. Uno dei pilastri principali di una buona applicazione è la sicurezza che offre ai suoi utenti in base ai dati coinvolti. Sicuramente la sicurezza applicativa dipende tantissimo dalla sicurezza del backend e client-side è bene non gestire dati sensibili, se non strettamente necessario. Come spiegato nella Sez. 3.4.2 di Devise, l'autenticazione viene fatta attraverso email e password di registrazione e, in caso di esito positivo, al client viene restituito un token da utilizzare per ogni richiesta. Da ciò risulta evidente che l'unica informazione sensibile presente nel dispositivo durante l'utilizzo dell'applicazione sia soltanto il token, che di per sé non ha alcuna informazione sensibile e non può essere facilmente manipolato, poiché monouso e strettamente legato all'utente stesso.

La strategia di conservazione può essere qualsiasi, ma è bene conservare il token in modo sicuro, per evitare che una manipolazione possa essere parte di un attacco al backend. La scelta verte infatti sull'utilizzo di una libreria, reperibile attraverso NuGet (gestore dei pacchetti per .NET), chiamata Xamarin Auth, che fornisce supporto per l'OAuth e per lo storage sicuro di dati. Il supporto per OAuth non sarà utilizzato con la suddetta libreria, perché la strategia di autenticazione attraverso l'utilizzo di una entità di terze parti (Facebook, Google+, ecc.) avviene attraverso l'utilizzo di un meccanismo triangolato con il backend per il recupero di informazioni dal provider, per una migliore gestione dell'account server-side. Il meccanismo sfrutta la dependency injection, come spiegato nella sezione precedente. È necessario scrivere una interfaccia comune da utilizzare nella soluzione PCL, mentre l'effettiva implementazione è fatta nella soluzione Android e iOS separatamente, utilizzando i metodi specifici della piattaforma.

```
public interface IStoreCredential
{
    string ID { get; }
    string Token { get; }

    void SaveID(string ID);
    void SaveToken(string Token);
    void SaveIDAndToken(string ID, string Token);
    void DeleteCredentials();
}
```

Figura 4.6: Interfaccia *IStoreCredential* per Xamarin Auth.

```
private bool UserSignedIn()
{
    var AuthService = DependencyService.Get<IStoreCredential>();
    if (AuthService != null)
    {
        if (AuthService.ID != null && AuthService.Token != null)
        {
            // call to backend ... and return true if are valid
        }

    }
    return false;
}
```

Figura 4.7: Interfaccia *IStoreCredential* per Xamarin Auth.

La figura 4.6 mostra l'interfaccia, mentre la figura 4.7 mostra un esempio di utilizzo tramite il DependencyService. Il metodo riportato fa parte di un controllo circa una precedente autenticazione dell'utente all'avvio dell'applicazione: se sono presenti credenziali salvate e valide (verificate con il backend) allora il metodo consentirà la navigazione nell'applicazione, altrimenti mostrerà una interfaccia grafica per l'autenticazione. È ovvio che, in caso di autenticazione positiva, devono essere conservati id e token restituiti dal server per utilizzarli quando necessario. L'implementazione specifica della piattaforma per brevità non viene riportata, consiste nella creazione di classi separate che implementano ognuna l'interfaccia IStoreCredential utilizzando le librerie della piattaforma. È compito dello sviluppatore evitare di esporre dati sensibili o utilizzare un meccanismo debole, la libreria Xamarin Auth offre anche un insieme di metodi per il recupero e il salvataggio di informazioni completamente gestito in ogni piattaforma, creazione di uno spazio sicuro, protetto con un meccanismo di cifratura. Grazie alla libreria Xamarin Auth è stato possibile ideare un meccanismo sicuro per la conservazione client-side di informazioni di autenticazione e autorizzazione.

4.3.4 Servizi, HTTP Client e JSON parsing

Non tutte le applicazioni necessitano di connessione ad Internet per funzionare, basti pensare ai giochi stand-alone, app per la modifica foto e video, composizione documenti e così via. È anche vero che ormai la maggior parte delle applicazioni fornisce dei servizi all'utente appoggiandosi a servizi esterni, che possono essere pubblici o proprietari. Solitamente i servizi riguardano lo storage di dati e la loro manipolazione attraverso operazioni consentite da una logica di business. Laddove l'applicazione lo prevede, è anche possibile usufruire delle funzionalità che essa offre senza alcuna connessione a Internet, con la sincronizzazione dei contenuti che avverrà in un secondo momento. Tra le categorie di applicazioni che necessitano di connessione per funzionare vi sono quelle di messaggistica istantanea, contenuti multimediali, file sharing, ecc. ed è ovvio che il social network del presente lavoro rientra tra queste. Più in dettaglio, l'applicazione in questione è client (consumatore) delle API REST che il backend espone, progettato per la gestione dei dati e le operazioni descritte nel Capitolo 3.

L'utente finale interagirà con le informazioni presenti nelle basi dati in modo controllato, secondo le operazioni che la logica di business espone. L'interazione avviene attraverso le varie interfacce grafiche, ognuna delle quali, oltre a visualizzare i dati in un certo modo, esporrà un insieme di metodi per l'interazione con il backend, appositamente collegati a dei *widget* per l'interazione quali ad esempio Button, TextLabel, Image, vari Layout e così via. Essendo predominante la volontà di costruire un prodotto software facilmente manutenibile nel tempo, e secondo quanto previsto dall'architettura MVVM, questi comandi non sono altro che *handler*, al loro interno richiameranno dei metodi esposti dal ViewModel per l'aggiornamento dei dati (Model) ed eventualmente delle UI (View), grazie al binding che viene stabilito dal programmatore tra le tre parti. I metodi del ViewModel più complessi solitamente si appoggiano a uno o più *Services*, un servizio non è altro che una classe C# deputata a svolgere un insieme di compiti inerenti più o meno lo stesso insieme di dati: si può pensare ad un ProfileService per il recupero di dati circa un utente, NewsFeedService per il recupero di dati da mostrare nella Home Page e i meccanismi per la pubblicazione di un nuovo contenuto, e così via. Un servizio complesso può appoggiarsi ad uno o più altri servizi, non esiste un perimetro ben preciso sull'insieme di informazioni che ognuno deve gestire, dipende dalla complessità del prodotto software e dal grado di separazione dei moduli che si vuole avere.

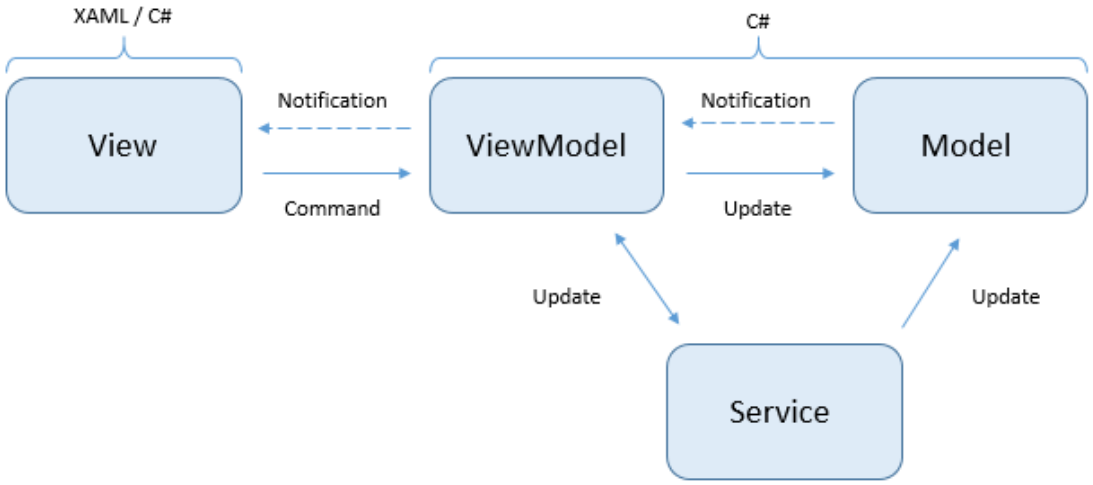


Figura 4.8: Architettura MVVM utilizzata.

Dal punto di vista architetturale i servizi si trovano tra il ViewModel e il Model, evidenziando il principio di separazione delle responsabilità. Il ViewModel può di certo modificare direttamente i modelli ma, trattandosi spesso di operazioni che coinvolgono aggiornamenti nella base dati del backend, è bene farlo attraverso i servizi, magari un aggiornamento remoto corrisponde a molteplici e diversi aggiornamenti in locale, infatti sussiste una comunicazione bidirezionale tra Service e ViewModel, mentre è monodirezionale tra Service e Model.

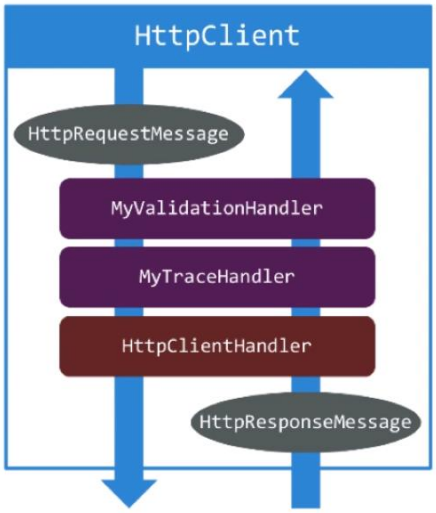
I servizi che consumano le risorse offerte dal backend necessitano di un meccanismo autenticato per le richieste HTTP. Tale meccanismo è fornito da un servizio che gestisce un oggetto singleton di tipo HttpClient a cui aggiunge le informazioni necessarie per l'autorizzazione delle richieste, servizio da richiamare in tutti gli altri servizi che effettivamente comunicano con il backend. Il package *Microsoft.Net.Http* include gli oggetti HttpClient per l'invio di richieste HTTP, ma anche le classi HttpRequestMessage e HttpResponseMessage per il processamento di richieste e risposte. L'oggetto client espone metodi asincroni, sia di alto livello (GET, POST, PUT, DELETE) e altri di basso livello (GetByteArray, GetStream). I metodi di più alto livello consentono una facile manipolazione dei messaggi di richiesta e di risposta, fornendo supporto per l'aggiunta o lettura di header, contenuti, recupero codice di stato, ecc. per cui più utili rispetto agli altri di basso livello in situazioni classiche. La gestione dei task asincroni è possibile con il pattern *async/await*, un meccanismo elegante di scrivere codice parallelo con un formato che richiama quasi in modo naturale la programmazione sincrona: brevemente, un metodo etichettato con la keyword async indica la presenza al suo interno di codice che può essere eseguito in modo asincrono, senza bloccare dunque il thread chiamante; i valori di ritorno dichiarati non possono essere diversi da Task, Task<T> o void. Un metodo di questo tipo deve contenere almeno una keyword await, al raggiungimento della quale il metodo viene sospeso fintantoché il task asincrono non termina; l'assenza di keyword async corrisponde ad una esecuzione sincrona del metodo stesso.

Il formato dati per la comunicazione client server è il JSON (Javascript Object Notation) e la libreria utilizzata per il parsing è la Json.NET di NewtonSoft (di terze parti, open source). Una libreria molto semplice e performante che offre un insieme di metodi statici per la serializzazione di oggetti e deserializzazione di dati. È utile perché il backend risponde soltanto a chiamate con formato dati JSON e dunque il client HTTP deve essere configurato per ciò, oltre all'aggiunta degli header Accept e Content-Type. La libreria offre la possibilità di costruire il client con uno o più filtri, capaci di intercettare la richiesta e la risposta ammettendo un certo insieme di operazioni su queste.

```
public class MyTraceHandler : DelegatingHandler
{
  protected override async Task<HttpResponseMessage> SendAsync
      (HttpRequestMessage request, CancellationToken cancellationToken)
  {
      Debug.WriteLine("request: " + request);

      var response = await base.SendAsync(request, cancellationToken);

      Debug.WriteLine("response: " + response);

      return response;
  }
  ...
}
```

Figura 4.9: Esempio di filtro su richiesta e risposta.

La figura riporta un esempio di filtro che opera su richiesta e risposta, in tal caso stampando su console il contenuto. Il meccanismo dei filtri si appoggia ad NetworkHandler, il più esterno della pipeline e responsabile ad esempio dell'utilizzo del protocollo TLS. Per ogni filtro è necessario avere due costruttori, uno senza argomenti e uno con parametro HttpMessageHandler, rispettivamente da utilizzare nel caso in cui il filtro è ultimo della pipeline (immediatamente prima del NetworkHandler) o meno, così da consentire il passaggio della richiesta/risposta al filtro successivo nella catena di elaborazione. Un esempio di settaggio della pipeline con due filtri è riportato nella figura seguente.




```
var networking = new HttpClientHandler();
var trace      = new MyTraceHandler(networking);
var validation = new MyValidationHandler(trace);
var client     = new HttpClient(validation);
```

Figura 4.10: Esempio di pipeline complessa.

Come si evince dalla figura 4.10, il primo elemento ad essere costruito è il più esterno alla pipeline (verso il backend) dunque l'ultimo ad operare in richiesta ed il primo in risposta: è ovvio che tale elemento deve essere il ClientHandler, necessario per la comunicazione; gli altri elementi sono i filtri che devono operare su richiesta e risposta, in ordine inverso rispetto a quando dichiarati. L'oggetto client prende come parametro il primo dei filtri della pipeline che provvederà a svolgere le sue operazioni e passarlo ai filtri successivi.

Grazie a questo meccanismo dei filtri è stato possibile proteggere le comunicazioni con TLS, abilitandolo nel ClientHandler, oltre che la gestione delle richieste autenticate. Quest'ultima è stata realizzata con un apposito filtro che recupera il token di autenticazione dallo storage sicuro (grazie a Xamarin Auth) e lo introduce nella richiesta, se necessario; inoltre, ogni risposta contiene il nuovo token autorizzativo, che lo stesso filtro si impegnerà a recuperare per aggiornare lo storage. In tal modo si ha un notevole risparmio di codice, le classi sono più snelle e devono occuparsi solo di consumare eventuali dati in risposta.

4.3.5 Caching foto e video

Quello del caching è un argomento che si ritrova in tutte le salse quando si parla di performance di un sistema. Nelle applicazioni mobile che usano servizi web può essere realizzato in varie strategie sia client side che server side, per vari tipi di contenuti: dati multimediali, informazioni calcolate a priori e fornite al client entro un certo intervallo di tempo (la news feed dei social network funziona proprio così), ecc. Per l'applicazione in questione il caching client side viene sfruttato soprattutto per i contenuti multimediali come le immagini, grazie alla libreria FFImageLoading. Tale libreria incorpora un client HTTP configurabile per la scaricare contenuti multimediali dalla rete, salvarli in locale e riutilizzarli quando necessario. Che il client HTTP sia

configurabile è un grosso vantaggio, poiché alcuni contenuti multimediali richiedono l'autenticazione e, in caso di profili privati, una autorizzazione che non può essere concessa senza autenticare il richiedente. La libreria estende la classe ImageView di Xamarin, deputata a mostrare una immagine all'interno del layout in cui viene posizionata. Per adempiere alla funzionalità di caching, la libreria utilizza un insieme di dizionari personalizzabili su cui effettuare una ricerca prima di accedere alla rete per recuperare il contenuto. Ammette un certo intervallo di tempo dopo del quale in automatico il file in cache viene eliminato e include vari meccanismi a supporto delle prestazioni come download paralleli e asincroni, visualizzazione di immagini *dummy* durante il download, pulizia cache, animazioni e trasformazioni.

Al momento non esiste una libreria per il caching video, pur essendo sicuramente utile. Per tal motivo è necessario implementare un meccanismo di caching che ricalchi il lavoro svolto dalla libreria utilizzata per le immagini: si tratta di costruire un dizionario da consultare ogniqualvolta viene richiesto un nuovo video e, in caso di contenuto non trovato, è necessario recuperarlo dal backend, per poi renderlo disponibile. È inoltre necessario un meccanismo di pulizia automatico dei contenuti dopo un certo tempo, per evitare di saturare inutilmente lo storage del dispositivo.

Il caching non aiuta soltanto ad evitare carico inutile al backend, a diminuire i consumi e ad avere una rete meno intasata, ma consente all'applicazione di funzionare anche in assenza di connettività. Si può pensare ad esempio ad un meccanismo di conservazione offline (database locale) delle azioni svolte dall'utente come commenti, mi piace o condivisioni e altro. Tali azioni serializzate dovranno essere recuperate quando è disponibile la connettività, in modo da poter comunicare al backend eventuali cambiamenti nei dati. La presenza di tali meccanismi, uniti ai contenuti multimediali disponibili offline grazie alla cache, rende l'applicazione certamente più appetibile e nel complesso un backend molto performante.

4.3.6 Pull to refresh

L'utilità di un'applicazione non è data soltanto dalle informazioni o servizi che offre, ma anche da come li offre. Si sa, i social network attirano l'attenzione dell'utente man mano che diventano sempre più "real time" in ogni aspetto. È scomodo dover chiudere e riaprire l'app per aggiornarla e il meccanismo dei websocket sicuramente offre un buon contributo a meccanismo come chat, notifiche e altre situazioni, ma non è detto che debba essere massivamente utilizzato per tenere aggiornata qualunque cosa. A tal proposito esiste il meccanismo del pull-to-refresh, ossia all'utente è data la possibilità di aggiornare i contenuti quando desiderato. Tale meccanismo risulta utile in quei contesti come la news feed, in cui l'aggiornamento costante dei contenuti con il meccanismo dei websocket sarebbe troppo invasivo e provocherebbe una cattiva user experience. Le due tecniche si possono ovviamente combinare, ossia con una certa regolarità temporale avviene un aggiornamento automatico; se l'utente lo desidera, può aggiornarli attraverso il pull-to-refresh. Il meccanismo si attiva quando l'utente trascina verso il basso la schermata quando si trova già al punto iniziale, come nel caso di una ListView che mostra il primo elemento della lista.

Il meccanismo si compone di almeno due stati, gestibili con una proprietà booleana isRefreshing. Fortunatamente i widget di tipo ListView e ScrollView implementano già il meccanismo visuale dello spinner, per cui è sufficiente agganciare un metodo da richiamare durante per l'aggiornamento dei contenuti, oltre che la proprietà sentinella isRefreshing per evitare aggiornamenti paralleli. Nel caso trattato con Xamarin e secondo il modello MVVM, è anche necessario che l'ottenimento di un nuovo contenuto aggiorni le interfacce grafiche dell'applicazione, realizzato grazie alle property di C#, richiamando il metodo di notifica (binding) OnPropertyChanged() all'interno del setter.

La figura seguente riporta l'implementazione del meccanismo di aggiornamento della home page.

```
// isRefreshing property for Pull-to-Refresh
public bool IsRefreshing
{
    get { return _isRefreshing; }
    set
    {
        _isRefreshing = value;
        OnPropertyChanged(nameof(IsRefreshing));
    }
}

// refreshCommand property for Pull-to-Refresh
public Command RefreshCommand
{
    get { return _refreshCommand; }
}
```

Figura 4.11: Proprietà e comando per il pull-to-refresh.

Il comando *_refreshCommand* è un elemento privato, inizializzato nel costruttore con il nome del metodo da invocare in caso di refresh (*refreshHome*, figura 4.12) ed esposto in sola lettura all'esterno attraverso la proprietà *RefreshCommand*, similmente per la proprietà IsRefreshing, che però attiva la notifica per il binding associato, che servirà a delimitare e a gestire le operazioni da compiere nel corso del caricamento.

```
// activate the refresh
private void refreshHome()
{
    IsRefreshing = true;
    loadCurrentUserHomeContent();
    IsRefreshing = false;
}

void loadCurrentUserHomeContent()
{
    HomeContent = _currentUserService.getHomeContent();
}
```

Figura 4.12: Proprietà e comando per il pull-to-refresh.

Quando viene attivato il meccanismo di refresh, viene invocato il comando RefreshCommand avente come argomento il metodo refreshHome(), che a sua volta attiverà la sentinella booleana per evitare aggiornamenti paralleli e provvederà ad eseguire una certa logica. La logica dell'esempio mostrato, in accordo con il pattern MVVM, consiste nell'invocare un metodo del servizio che gestisce la home page dell'utente corrente, tramite il metodo loadCurrentUserHomeContent(). Per brevità, il metodo getHomeContent() del servizio non è mostrato, incapsula una comunicazione autenticata con il backend.

Il contenuto è aggiunto alla proprietà HomeContent, che a sua volta invocherà nel setter il metodo di notifica per il binding (figura 4.13), in modo da aggiornare la grafica con il nuovo contenuto, altrimenti i dati resterebbero aggiornati solo nelle variabili e non nella schermata.

```
public List<Post> HomeContent
{
    get { return _homeContent; }
    set
    {
        _homeContent = value;
        OnPropertyChanged(nameof(HomeContent));
    }
}
```

Figura 4.13: Proprietà per il contenuto della Home Page.

```
protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Figura 4.14: Proprietà e comando per il pull-to-refresh.

La figura 4.14 mostra il meccanismo di notifica ad eventi grazie al quale invocare un nuovo rendering visuale degli elementi. Il metodo riceve il nome della proprietà che sarà utilizzato come argomento per la costruzione di un evento riferito al cambiamento della proprietà.

Le tre properties discusse pocanzi devono essere agganciate al meccanismo che la ListView di Xamarin espone, in modo da poter gestire il meccanismo descritto con l'ausilio di queste.

```
homeContentList.ItemsSource = currentUserVM.HomeContent;
homeContentList.IsPullToRefreshEnabled = true;
homeContentList.RefreshCommand = currentUserVM.RefreshCommand;
homeContentList.SetBinding(ListView.IsRefreshingProperty, nameof(currentUserVM.IsRefreshing));
homeContentList.SetBinding(ListView.ItemsSourceProperty, nameof(currentUserVM.HomeContent));
```

Figura 4.15: Configurazione della vista ListView per la Home Page.

L'elemento *homeContentList* è di tipo ListView; è necessario configurare la sorgente dei dati, attivare il meccanismo del pull-to-refresh e associare le tre proprietà RefreshCommand, IsRefreshing e HomeContent. Come si può notare dalla figura 4.15, il binding è necessario affinché l'invocazione del metodo OnPropertyChanged() abbia effetto sulla schermata. In tal modo è stato possibile sfruttare il pull-to-refresh in varie parti dell'applicazione, tra cui notifiche, contenuti della home page, profilo utente, entità sportive e così via.

5. Angular

Angular è un framework open-source basato su Javascript per la costruzione dei frontend nelle applicazioni web, mantenuto oggigiorno principalmente da Google. Ne sono state rilasciate varie versioni, la versione 1 (detta AngularJS) offre un modello MVC completo lato client, mentre dalla versione 2 e successive (note come Angular) offrono un modello MVVM o basato su componenti. La prima versione è incompatibile con le altre, cambiano totalmente i principi e le varie funzionalità, l'ultima rilasciata è la 4 anche se per Ottobre 2017 è previsto il rilascio di Angular 5. La versione 4 consiste in vari miglioramenti della già potente e molto apprezzata versione 2. Angular è uno dei tanti attori in gioco, in forte competizione con React, di casa Facebook. A differenza di Angular, React è una libreria per la generazione delle sole viste, molto più performante grazie al VirtualDOM, ma è necessario progettare una soluzione separatamente per Model e Controller (o ViewModel), se necessario. Da qui l'architettura Flux e la sua variante Redux, sicuramente performanti e altamente scalabili ma nel complesso è tutto più difficile da utilizzare rispetto ad Angular. Per tal motivo la scelta del framework ricade su quest'ultimo, comunque ampiamente utilizzato.

5.1 AngularJS 1.x vs Angular

Come detto, Angular 2 è stata una completa riscrittura del framework, destando non poche polemiche soprattutto per la limitata portabilità di quanto già sviluppato con le versioni precedenti. Angular 2 supporta le specifiche di ECMAScript, TypeScript e Dart. Le specifiche definite da ECMAScript sono delle specifiche standardizzate di un linguaggio meglio noto come Javascript. Ciò è stato fatto per evitare che, a partire da quanto ideato in casa Netscape, ognuno creasse la propria versione incompatibile con le altre (come JScript in casa Microsoft e altri). Il processo di standardizzazione ha avuto alti e bassi, "terminato" ufficialmente nel 2015 con la versione ES6 (ECMAScript 6), nonostante una nuova release sia attualmente in programma per ES7. In Angular vengono eliminati i controller e lo $scope, mentre il two-way data binding non è più la tipologia di binding di default. La completa riscrittura del framework Angular non riguarda soltanto l'eliminazione di tanti concetti ostici e quasi fastidiosi, ma anche voler essere un framework predominante nello sviluppo mobile ibrido (Cordova e Ionic supportano Angular), nonostante i limiti già discussi nella Sez. 2.1 circa la tecnica di sviluppo mobile ibrido. Anche se Angular supporta vari linguaggi, è fortemente consigliato l'uso di Typescript. Si tratta di un linguaggio sviluppato in casa Microsoft, libero e open-source, un super-set di Javascript che segue le regole di ECMAScript. Javascript viene ormai utilizzato per costruire frontend e backend (NodeJS ed ExpressJS sono ormai diffusi), ma il crescente bisogno di robustezza e sicurezza nel codice Javascript costituisce la base di Typescript: un linguaggio fortemente tipizzato e compilato, che in fin dei conti produce del codice Javascript "safe". L'ambiente di sviluppo per Javascript si è consolidato pian piano nel tempo, esistono vari tool a supporto di configurazione e integrazione di pacchetti software, per tal motivo il team Angular ha ideato la Angular CLI (similmente alla CLI di Rails), un ambiente a riga di comando che aiuta lo sviluppatore nella creazione di componenti, import di pacchetti e modifiche varie, basata su NodeJS. Interessante è il supporto che offre nello sviluppo dei contenuti, grazie a vari script che uniscono i vari moduli in un unico bundle e la visualizzazione in un browser attraverso un web server integrato con live reloading.

Le differenze tra le versioni di Angular 2, 4 e 5-beta (Agosto 2017) sono importanti, ma non così marcate come tra la versione 1 e 2. Si tratta soprattutto di miglioramenti circa l'efficienza di un'applicazione Angular, oltre a risoluzioni di bug e varie instabilità. Altri dettagli utili alla costruzione del frontend, sono dettagliati nella Sez. 5.2. Di seguito sono trattati alcuni pilastri di Angular, utili per la costruzione dei frontend.

5.1.1 Observer e ReactiveX

Angular è un framework fortemente orientato agli eventi e integra bene il pattern "observer". In ingegneria del software, si tratta di un pattern in cui un elemento (chiamato soggetto) mantiene una lista di elementi dipendenti (chiamati osservatori), notificando ad essi automaticamente qualsiasi cambiamento di stato pubblicato al suo interno. Il cambiamento di stato spesso è definito "evento" e il pattern è utile quando è necessario comunicare a più entità che è avvenuto un certo cambiamento. Così come è possibile registrarsi per la ricezione di eventi, è anche possibile interrompere tale ricezione. Una possibile strategia per realizzare tale comportamento è data dalla logica ReactiveX, a sua volta implementata in vari linguaggi, tra cui Javascript con la libreria RxJS. Tanti metodi in Angular fanno uso di tale pattern, considerabile anche un sostituito delle Promise dato che, oltre ad essere capace di adempiere le stesse funzionalità, offre molto altro. Entrambi sono un buon supporto per la programmazione asincrona, ma gli Observable sono molto più potenti, robusti e utili. Una Promise è capace di gestire un solo evento, che sia fallito o realizzato, e non supporta la cancellazione; di contro, un Observable è uno stream di zero o più eventi, molto utile nei meccanismi "reactive" in cui l'esecuzione asincrona di vari metodi pian piano darà vita al risultato finale. Pur sembrando una cosa banale, la possibilità di riprovare l'esecuzione di un metodo a causa di un criterio di fallimento fa degli Observable un mezzo potente nelle comunicazioni HTTP.

In tale pattern esistono tre elementi principali: observable, observer e subject. L'oggetto observable rappresenta una collezione basata su un meccanismo di tipo push, dunque fa da provider di eventi o notifiche. Tali eventi o notifiche sono ricevute da un oggetto observer, attraverso un meccanismo di iterazione automatico su una sequenza di riferimenti ad oggetti osservabili. In realtà non si agisce mai direttamente su un oggetto observable, bensì su un soggetto (oggetto subject), che è al contempo sia un observable che un observer. I subject sono molto utili come proxy tra entità, ad esempio un soggetto può comunicare con il backend (observer di esso) e al contempo essere fonte di dati per altri observer (dunque funge da provider o observable), utili per implementare meccanismi come il buffering e il caching delle informazioni. Non si tratta di oggetti thread-safe, per cui si assume che ciò sia eventualmente gestito dalle parti chiamanti, in tal modo si ottiene efficienza nello stream di eventi. Esistono vari tipi di subject: PublishSubject (o Subject), ReplaySubject, BehviourSubject e AsyncSubject.

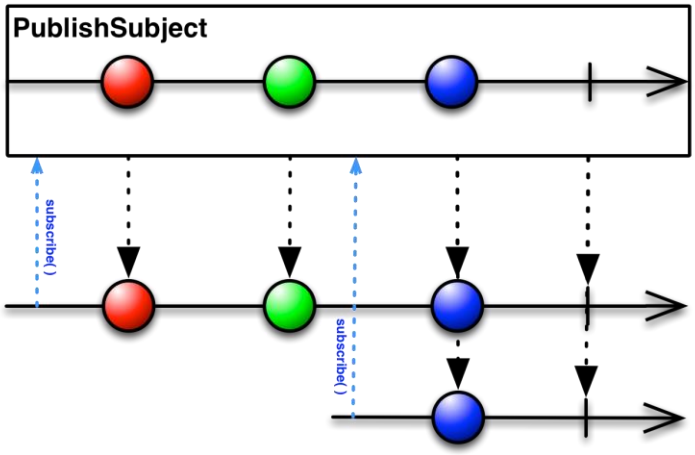


Figura 5.1: Publish Subject.

PublishSubject, noto anche come Subject, ha il comportamento di default, inizia lo stream di eventi da quanto avviene una sottoscrizione, fintantoché non viene chiuso o a causa di un errore o per invocazione del metodo onComplete().

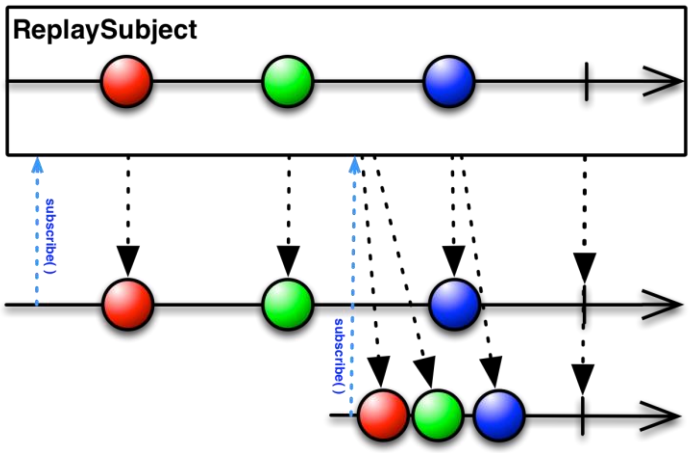


Figura 5.2: Replay Subject.

ReplaySubject conserva tutti i valori pubblicati in esso, che saranno dunque forniti dal primo all'ultimo ad ogni sottoscrizione, oltre agli eventi ricevuti successivamente.

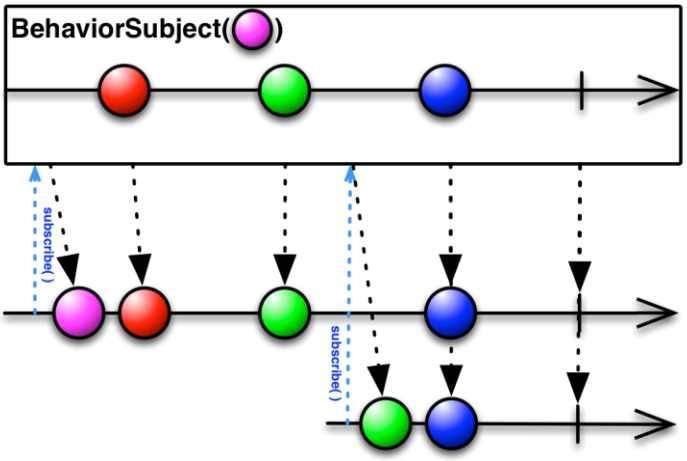


Figura 5.3: Behaviour Subject.

BehviourSubject è simile al precedente, ma vale solo per l'ultimo valore conservato, piuttosto che per tutta la storia di eventi. In altre parole, ogni volta che un observer si iscrive ad esso, riceve l'ultimo valore conservato; al momento di un nuovo stream, il nuovo valore viene aggiornato come ultimo valore.

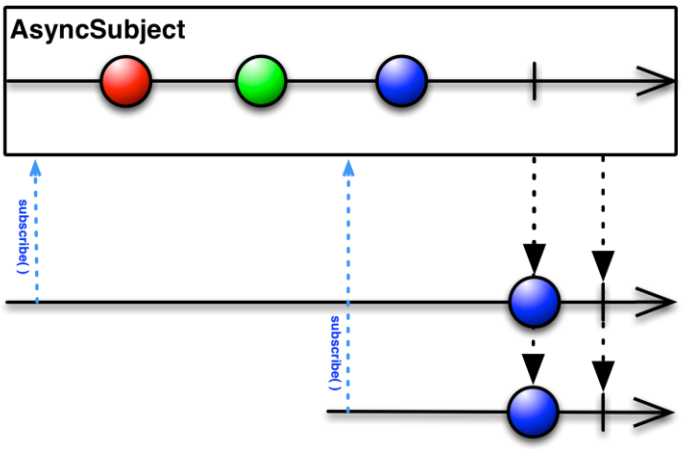


Figura 5.4: Async Subject.

Infine, AsyncSubject funziona quasi come il BehaviourSubject, trasmette solo l'ultimo valore e solo quando il soggetto si trova nello stato "complete", ad esempio perché si tratta di una sequenza che arriva pian piano nel tempo e la si vuole fornire quando è tutta presente, per cui si chiama il metodo onComplete() che lo emette a tutti gli osservatori registrati in quel momento e poi chiude lo stream.

Risultano evidenti le potenzialità offerte dal pattern observer rispetto alle Promise e, grazie all'implementazione offerta dalla libreria RxJS, è stato ampiamente utilizzato per la costruzione di alcuni meccanismi del frontend web (dettagli nelle sezioni successive), specialmente in tutti i servizi tra cui comunicazione con il backend, meccanismo publish/subscribe asincrono per la comunicazione tra componenti non incapsulati tra loro ed infine la gestione dei websocket e delle notifiche.

5.1.2 Routing: Traditional vs Single Page Application

Le applicazioni web tradizionali evolvono con la navigazione utente attraverso un browser, consistono in una richiesta al server per ottenere delle informazioni in risposta, sotto forma di documento web. Ad ogni richiesta per una certa URL corrisponde dunque una elaborazione lato server e la ricezione di un nuovo contenuto, che andrà a sostituire totalmente il precedente. Sul server entra in gioco il routing delle richieste, ossia un meccanismo capace di intercettare la richiesta e capire a quale path o risorsa si riferisce, in modo da poterla delegare alle componenti interne deputate alla sua gestione per l'elaborazione e la costruzione dell'output corrispondente, come accade nel caso di Rails (Sez. 3.3) e Spring per default. Si tratta però di routing gestito interamente dal server, poiché è questo che espone un insieme di risorse raggiungibili attraverso URL e le fornisce nel momento in cui vengono richieste.

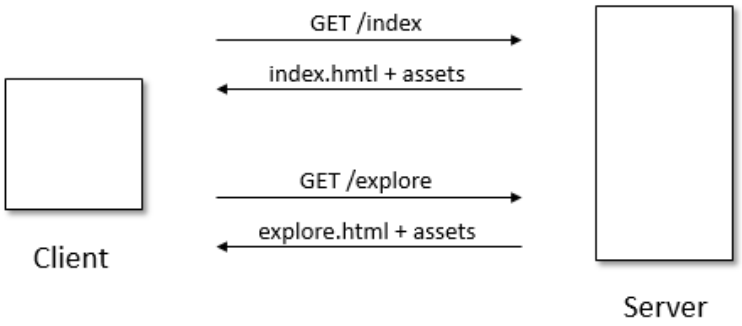


Figura 5.5: Comunicazione tradizionale nelle applicazioni web.

Come mostra la figura, ogni richiesta corrisponde ad una nuova elaborazione e ad un nuovo contenuto: ad esempio, la home page di una applicazione web dinamica tradizionale si trova al path relativo "/index", la sezione esplora utenti in "/explore", e così via; ad ogni risposta corrisponde una sostituzione completa della vecchia pagina, da qui la dicitura di navigazione "con interruzioni" per le applicazioni web tradizionali.

Il vantaggio principale di tale approccio è che ogni richiesta passa attraverso il server e dunque per essere elaborata deve soddisfare la logica presente. Svantaggio principale è la sua insostenibilità al crescere del numero di utenti, perché per ogni richiesta il server deve elaborare un nuovo contenuto, che potenzialmente ha delle variazioni minime o assenti rispetto al precedente. Tale contenuto è comunque trasferito dal server al client, con un payload utile potenzialmente nullo ma un trasferimento di byte anche grande. Come spiegato in Sez. 2, la tendenza odierna è quella di costruire le viste client-side, comunicando con il backend solo per recuperare o aggiornare i dati. Ciò svincola il server dalla costruzione onerosa delle viste, grazie ai potenti framework Javascript adesso responsabili della costruzione di viste eleganti con meccanismi interattivi che non coinvolgono necessariamente il backend, favorendo la scalabilità in vari aspetti. La comunicazione con il server avviene attraverso un meccanismo come REST, il server espone delle risorse (endpoint) attraverso URI che il client contatterà per ottenere le informazioni corrispondenti.

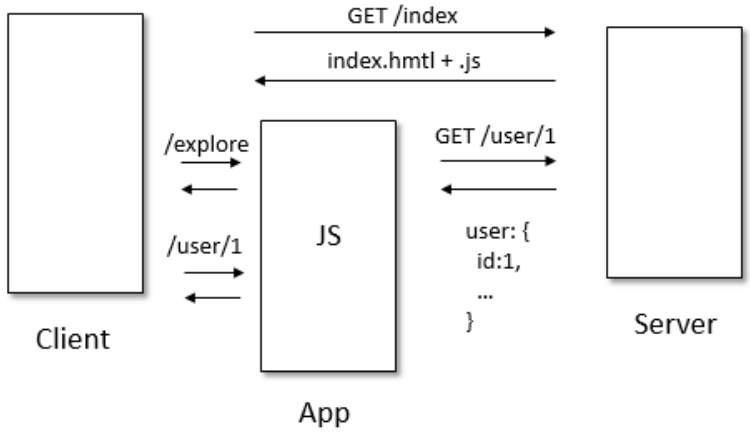


Figura 5.6: Routing nelle SPA.

La figura riporta nette differenze rispetto al meccanismo tradizionale, adesso la navigazione delle viste è interamente gestita dall'applicazione Javascript (per comodità riportata al di fuori del client, ma nella realtà è unica entità). La navigazione è ancora presente, l'utente ha la sensazione di navigare un insieme di pagine, con la differenza che non si avranno realmente delle richieste verso il server per ottenere la nuova pagina, bensì delle richieste per ottenere un nuovo contenuto che andrà a popolare una parte della pagina costruita sul client.

Nell'esempio, l'utente naviga un profilo attraverso la sezione "esplora" dell'applicazione, ma l'unica richiesta fatta dal client al server riguarda i dettagli dell'utente. È ovvio che la prima richiesta assoluta fatta al server consiste nello scaricare la pagina principale (index.html) che avrà i riferimenti a tutti i file Javascript associati per il frontend (*.js), le successive richieste corrispondo al trasferimento dati in JSON o altro formato. Potenzialmente il numero di richieste fatte al server con tale meccanismo può diminuire, ma in caso di grandi applicazioni può anche crescere, ma ciò non è un problema per la scalabilità poiché il payload trasferito è di gran lunga più piccolo rispetto all'intera pagina HTML. Sviluppandosi tutto in locale, ulteriori vantaggi riguardano la velocità di navigazione, migliore user experience e interattività delle parti.

Sul server risiede ancora la logica di accesso ai dati, magari con meccanismi di autenticazione e autorizzazione, ma adesso per ogni URI non corrisponde una vista, poiché la vista in sé viene decisa dall'applicazione sul client, ossia dal complesso di file Javascript deputati "on-the-fly" alla modifica dell'unica pagina HTML man mano che l'utente vi interagisce. Questo meccanismo comporta per lo sviluppatore la costruzione di una macchina a stati per prevenire comportamenti indesiderati e fornire una navigazione corretta, perché si sa, gli utenti non utilizzeranno mai l'applicazione secondo il flusso di navigazione pensato dal programmatore. Tale macchina a stati ovviamente collide con quanto pensato server side, più URI (adesso una risorsa è identificata in modo univoco) possono essere parte di una unica pagina e più pagine possono far riferimento alla stessa risorsa.

I vari framework per i frontend incorporano il routing descritto attraverso varie strategie: la prima utilizzata fu la "hash-bang" notation (#! - attualmente usata in Gmail) perché un hash corrisponde ad una ancora, un riferimento al contenuto della pagina corrente e dunque già presente, in modo da non far partire dal browser (client) una richiesta al server, il complesso Javascript catturerà tale comando di navigazione e si attiverà per modificare la vista; tale strategia è stata sostituita in favore della navigazione (apparentemente) classica con le API di HTML 5 (pushState() e altre) per la corretta visualizzazione dell'URL completo nella barra di navigazione del browser, senza alcun hash. Angular incorpora tale meccanismo e fornisce un supporto alla navigazione con le guardie (Guards) per evitare che un utente navighi in pagine senza autorizzazione, a causa ad esempio di mancata autenticazione, mancata compilazione di pagine precedenti (utile nei form a step successivi), e così via. Il routing client side ha comportato problemi, come l'impossibilità di avere i preferiti (bookmarks) e di consentire ai crawler l'indicizzazione dei contenuti, poiché tali viste sono frutto dell'esecuzione e i crawler ispezionano, non eseguono. Tali problemi sono in parte risolti dai framework, Angular ammette la tecnica di compilazione AoT (Ahead of Time), ma è anche responsabilità del sistemista configurare il server in modo da fornire sempre la pagina principale, qualunque sia l'URL richiesto: in tal modo il framework farà bootstrap dell'applicazione e successivamente elaborerà le viste sul client, rendendole disponibili sia in caso di bookmarking che ai crawler.

Lo scenario del routing appena descritto non si trova nel caso dell'applicazione mobile, ma la creazione della macchina a stati è ben più evidente: l'insieme delle schermate e la gestione degli eventi associati al tocco dei vari widget deve essere parte di una gerarchia di navigazione ben organizzata e progettata dallo sviluppatore, il flusso di navigazione è quello dato dal passaggio da una schermata all'altra.

5.1.3 Compilazione AoT

Un'applicazione Angular consiste di un insieme di componenti con i corrispondenti template HTML, ma prima che il browser possa renderizzare l'applicazione è necessario convertire questi componenti in codice Javascript. Ciò può essere fatto in modalità "just-in-time" subito dopo aver scaricato completamente l'applicazione dal server, approccio standard ma non ottimo.

Si ha bisogno della (pre)compilazione per raggiungere un alto livello di efficienza nell'esecuzione delle applicazioni Angular. Il problema è sempre relativo alle prestazioni, che possono essere fortemente penalizzate per due motivi: la compilazione nel browser richiede il compilatore, da scaricare come parte di una libreria, dunque maggiore lentezza nella fase di bootstrap; inoltre, poiché la compilazione richiede del tempo aggiuntivo, anche l'interattività è penalizzata al crescere della complessità applicativa. Possono essere presenti degli errori logici, come accade per il collegamento dei componenti non dichiarati nel modulo principale, che con la classica tecnica di compilazione sono rilevati solo a runtime e causano l'interruzione dell'esecuzione stessa. Grazie alla compilazione "ahead-of-time" tutto ciò può essere risolto. Il rendering è più veloce, perché il browser scarica una versione precompilata dell'applicazione. Si riduce il numero di richieste asincrone (AJAX) per costruire i partial HTML, poiché gli asset CSS vengono inseriti nel codice Javascript compilato. Si elimina la necessità di scaricare il compilatore Angular, poiché a compilazione non viene più effettuata sul client. Oltre a rilevare gli errori prima di causare una cattiva user experience, si ottiene un vantaggio di sicurezza importantissimo, perché al client è fornito solo l'output della compilazione, evitando possibili attacchi di iniezione; ciò non migliora di certo la sicurezza del backend, che deve essere comunque protetto.

5.1.5 Angular dependency injection

I principi della dependency injection sono stati già ampiamente affrontati e discussi nei Capitoli precedenti, per cui non saranno ripetuti, sarà soltanto analizzato il meccanismo offerto da Angular. Ancora una volta non è necessario costruire le dipendenze direttamente ma è sufficiente averle dichiarate nel modulo principale dell'applicazione. Per utilizzarle nel codice, come nei servizi o nei componenti, è necessario abilitare il meccanismo di iniezione delle dipendenze, attraverso il costruttore della classe: le dipendenze devono essere specificate come parametri e Angular provvederà al loro recupero in automatico, gestendone il ciclo di vita secondo la logica già descritta nelle sezioni precedenti. Le entità che possono essere iniettate dal framework devono essere marcate con il decoratore *@Injectable()*, proprio per metterlo a conoscenza dell'esistenza di tale entità iniettabile. Il decoratore *Injectable* si ritrova quasi ovunque negli elementi dell'applicazione, proprio a sottolineare la natura di Angular fortemente basata su tale meccanismo.

5.2 Angular per il web

Angular è il framework utilizzato per la costruzione del frontend web per il social network in questione. Anch'esso, similmente a quanto fatto dalle applicazioni mobile, è client (consumatore) delle API esposte dal backend. Angular è un argomento molto ampio, per cui saranno discusse soltanto le principali strategie utilizzate per la costruzione dell'applicazione web. La scelta è stata quella di consentire l'utilizzo dell'applicazione solo previa autenticazione, per cui a parte una pagina di welcome e una di login o registrazione, non c'è molto altro. A seguito di una autenticazione positiva, all'utente viene presentata una home page attraverso cui navigare l'applicazione: la home page è a tutti gli effetti una news feed degli utenti seguiti o delle entità sportive preferite, la sezione esplora consente la ricerca delle risorse attraverso una interfaccia master-detail e infine la sezione profilo, in cui l'utente può pubblicare dei contenuti o rivedere quelli passati. I meccanismi per la realizzazione di tutto ciò sono spiegati singolarmente nelle sezioni a seguire.

5.2.1 Component in Angular

Un'applicazione Angular è composta da elementi HTML definiti dal programmatore e classi component che li gestiscono, attraverso una logica applicativa data dall'insieme di servizi potenzialmente organizzati in moduli. Un'applicazione Angular ha un solo modulo di root, chiamato AppModule, all'interno di cui si dichiarano i provider, import, export e il componente principale di cui fare il bootstrap per l'applicazione, solitamente AppComponent. Se si tratta di un'applicazione che non deve essere importata in altre applicazioni la sezione di export nel modulo principale non serve.

Un component controlla una parte di schermata, chiamata vista, a cui è associata un contenuto HTML e il component diventa l'elemento principale del framework, quasi come in React. Il concetto di direttiva presente nel vecchio AngularJS è stato eliminato, in favore di due nuovi elementi, appunto il component e il nuovo elemento direttiva. Un component non è altro che una direttiva Angular con un template HTML, mentre una direttiva Angular è un complesso Javascript che è capace di dare un comportamento all'elemento HTML in cui viene inserito. Ogni componente ha un ciclo di vita gestito interamente da Angular: viene creato, renderizzato insieme ai componenti figli, viene controllato lo stato delle proprietà per notificare eventuali cambiamenti e distrutto quando non più utile. Angular offre delle callback per ogni stato del ciclo di vita del componente, in modo da consentire l'esecuzione di certe operazioni al variare di questi. Anche le direttive, essendo component senza template, hanno gli stessi *hooks*, tranne ovviamente quelli relativi alla visualizzazione.

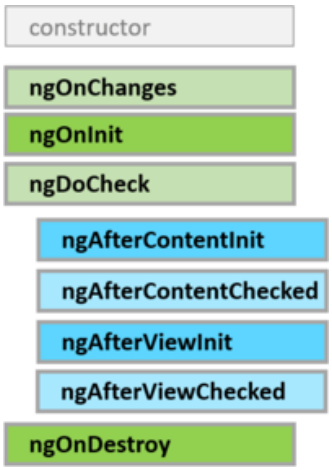


Figura 5.7: Angular lifecycle hooks.

La figura mostra i metodi disponibili, quelli in celeste non sono presenti per le direttive poiché riguardano il contenuto visuale. Il primo passo è chiamare il costruttore del component, in tale fase è ovvio che non sono state inizializzate le proprietà di input (da padre a figlio) né la vista associata al componente stesso, dunque si può operare su quelle proprietà che non riguardano rendering o interazioni. Dal punto di vista pratico questi metodi sono l'implementazione di una interfaccia il cui nome è lo stesso del metodo ma senza il prefisso "ng", se non correttamente importato il framework non chiamerà i metodi e non segnalerà alcun errore. Le callback più utili per il programmatore sono ngOnInit(), ngOnDestroy() ed ngOnChanges(), utili rispettivamente nell'agganciare un component ai meccanismi dei servizi (soprattutto quelli asincroni degli Observable) e a rilasciare eventuali risorse prima della distruzione, mentre l'ultimo citato è utile quando cambia un componente e per qualche motivo si vuole verificare il suo stato oppure operare in qualche modo.

Angular elimina totalmente il concetto di MVC o MVVM nel framework, utilizzando il cosiddetto "component pattern" in cui tutto si basa su componenti (building-blocks) che insieme danno un prodotto finale. Certamente, come discusso nelle sezioni precedenti, i pattern citati sono molto utili per avere un'architettura software elegante e facile da mantenere. Eppure al crescere della complessità del frontend tali strutture possono diventare ingestibili, specialmente per i controller o il complesso di ViewModel e Service. Nello sviluppo software un componente è una unità con logica interna e proprietà, alcune delle quali possono essere esposte all'esterno. Tutto si consuma attraverso delle interfacce, per cui l'interno può potenzialmente evolvere senza inficiare sul resto. L'assenza di un modello MV* sicuramente rende il framework più "libero" nella scelta di un'architettura client-side, ma non deve essere scambiata con l'impossibilità di riprenderne i funzionamenti di base: se strettamente necessario, tutto può essere ricondotto all'utilizzo di Service, classi appositamente pensate per l'espletamento di un insieme di funzioni, come comunicazione con il backend, gestione dello stato client-side, ecc. Infine, più component insieme possono essere parte di un modulo Angular, a formare un raggruppamento di elementi deputati allo stesso compito. Quest'ultima idea è alla base dell'esportazione di moduli da riutilizzare in altre applicazioni Angular, ma è a scelta libera dello sviluppatore sfruttarlo per una migliore organizzazione interna del software. Dato che in Angular tutto è un component, sono stati introdotti diversi meccanismi per la comunicazione tra essi. Il two-way data binding è affiancato dal property binding e dall'event binding, rispettivamente per la comunicazione da un elemento del component verso il DOM e, viceversa, dal DOM ad un elemento del component.

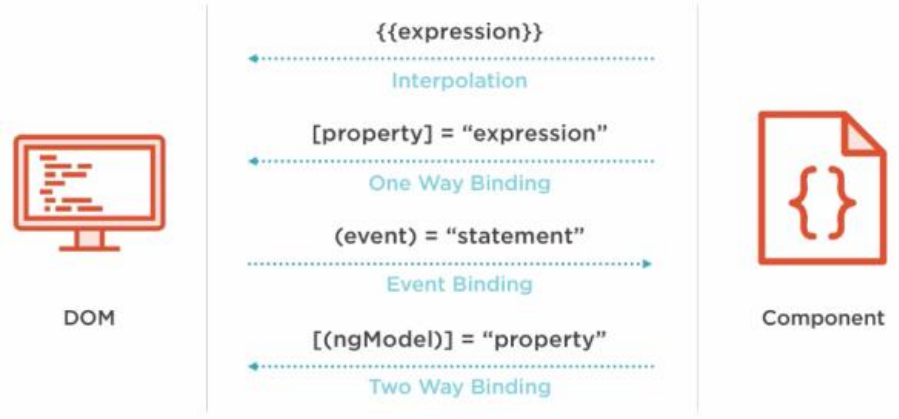


Figura 5.8: Il binding in Angular.

I nuovi meccanismi di binding, evidenziati nella figura 5.8, sono molto utili nella comunicazione tra i componenti, dato che un componente in sé può essere formato da tanti altri.

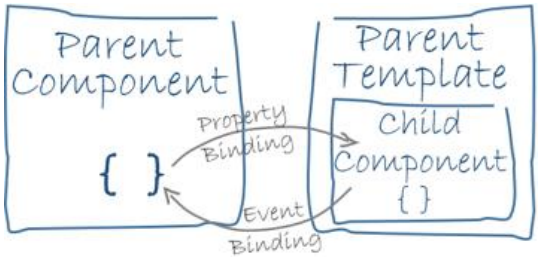


Figura 5.9: Parent-Child binding in Angular.

Un meccanismo di comunicazione come quello della figura 5.9 è molto frequente, ad esempio un'interfaccia di tipo master-detail può essere strutturata con un minimo di tre component: uno generale per la vista che opera come padre e due component figli, la lista e la vista, per i dettagli; tra di essi avviene una comunicazione tramite eventi (figlio-padre) al click di un elemento della lista e una di basata su binding di proprietà (padre-figlio) per mostrare il dettaglio dell'elemento scelto.

L'utilizzo dei component per la costruzione del frontend è stata massiva, poiché la logica alla base di questi è che devono essere più piccoli possibili; se troppo grandi, allora è necessario (e sicuramente possibile) suddividerli in component più semplici. Due delle funzionalità principali del social network sono la ricerca delle risorse sportive e la news feed. La home page è formata da una navbar per la navigazione generale, una lista di post e alcuni contenuti riguardanti l'utente come il widget per la pubblicazione di un contenuto, la lista di alcune risorse preferite o eventi in scadenza, per cui sono necessari: un component generale che faccia da contenitore, uno per la navbar, uno per la lista di post e infine uno per le pubblicazioni. A loro volta, la lista di post è composta da celle, divise in altri 3 component: dettagli utente, dettagli del post e foto (o video) del post, e così via, per cui il numero cresce a dimisura. Ovviamente ha senso separare in component laddove è necessario riutilizzare il componente, ad esempio il widget di pubblicazione, composto al più da un elemento EntryText, caricamento file multimediali e pulsante per la pubblicazione non ha altro, e una loro separazione in diversi component sarebbe inutile. Più complessa risulta essere l'interfaccia di ricerca, formata da una vista master detail per la ricerca basata sul ranking e altri parametri. L'interfaccia è composta da una vista contenitore, la navbar per la navigazione generale (componente riutilizzato), un component per la mappa, il component lista e il component dettagli risorsa. A loro volta, la lista è composta da varie celle, ognuna formata da uno o più component, mentre il dettaglio dell'elemento è formato dai component che lo descrivono (intestazione, informazioni, ranking, recensioni, followers, ecc). Uno schema semplificato della sezione ricerca è mostrato nella figura seguente.

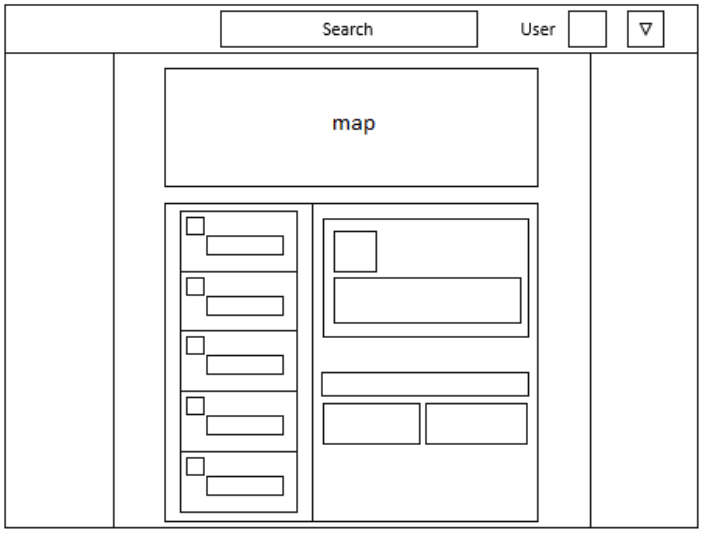


Figura 5.10: Schema semplificato della vista di ricerca.

Nonostante appunto si tratti di uno schema semplificato, si può notare la complessità degli elementi che compongono le single parti, ognuna formata da aggregazione di vari component deputati a rappresentare uno ben preciso contenuto, potenzialmente riutilizzabile per la composizione di altre viste.

5.2.2 Gestione dell'autenticazione

L'autenticazione utente è fondamentale nell'applicazione progettata ed è stata realizzata sfruttando i meccanismi offerti dal pattern Observable, le Guards del modulo di Routing e il modulo HTTP per la comunicazione con il backend. Nonostante il modello MVVM scompare in Angular, l'utilizzo dei Services è utile per avere un unico punto di accesso ai dati e continuare a mantenere separate le responsabilità dei vari componenti, oltre che un facile debug di questi con dei servizi mock, se necessario.

I servizi istanziati sono singleton e rimangono in vita finché non viene totalmente terminata l'esecuzione dell'applicazione, per cui al suo interno è possibile mantenere uno stato, come appunto lo stato dell'autenticazione. La generazione dei servizi può essere fatta attraverso i comandi della ricca Angular CLI, che configura automaticamente i file di progetto con gli import necessari, favorendo il lavoro al programmatore. La gestione locale dello stato dell'autenticazione viene gestita da un BehaviourSubject, che emette l'ultimo valore registrato al nuovo osservatore nel momento in cui si iscrive ad esso. Si può dunque pensare ad un servizio che gestisce una proprietà di tipo *enum* con valori *LoggedIn* e *LoggedOut*, e tale proprietà sarà l'elemento del broadcast. Il servizio di autenticazione deve esporre delle API all'esterno per poter cambiare il valore di tale proprietà. Ma non bastano due metodi generici per settare il nuovo valore, è necessario un ulteriore metodo che scateni il broadcast, ossia un metodo di tipo "emitter" che vada a richiamare il BehaviourSubject ed effettui lo stream agli osservatori. L'utilizzo di BehaviourSubject a volte può risultare scomodo, poiché questo necessita obbligatoriamente di un valore di inizializzazione e non accetta il *null*, in tal caso inizializzarlo a LoggedOut non è un problema, in altri scenari può risultare problematico. La figura seguente riporta un esempio per lo scenario descritto.

```
@Injectable()
export class AuthService {
  private authManager:BehaviorSubject<AuthState>
    = new BehaviorSubject(AuthState.LoggedOut);
  private authState:AuthState;
  authChange:Observable<AuthState>;

  constructor() {
    this.authChange = this.authManager.asObservable();
  }
  login():void {
    this.setAuthState(AuthState.LoggedIn);
  }
  logout():void {
    this.setAuthState(AuthState.LoggedOut);
  }
  emitAuthState():void {
    this.authManager.next(this.authState);
  }
  private setAuthState(newAuthState:AuthState):void {
    this.authState = newAuthState;
    this.emitAuthState();
  }
}
```

Figura 5.11: Servizio per la gestione locale dell'autenticazione.

Si tratta di un servizio chiamato AuthService con due metodi login() e logout(), che rispettivamente settano il nuovo valore nella proprietà e lo emettono grazie al richiamo del metodo emitAuthState(). Le API per la registrazione del nuovo stato circa l'autenticazione saranno richiamate all'interno del servizio di autenticazione con il backend: dopo aver trasmesso le credenziali per la verifica, in caso di esito positivo, verrà richiamata l'API per registrare lo stato LoggedIn, altrimenti quella per registrare lo stato LoggedOut. È ovvio che questo serve esclusivamente per la gestione locale dell'autenticazione, tutte le altre richieste devono essere appositamente autenticate in termini di payload (con il token descritto nella Sez. 3.4.2); non basta lo stato "autenticato" come valore locale dell'applicazione, poiché ogni richiesta non autenticata sarà rifiutata dal backend. A tal punto vi sono due modi differenti per recuperare internamente lo stato dell'autenticazione: sottoscrivere un osservatore al subject in ogni componente, semplice ma poco elegante, oppure avvalersi di una guardia (Guard) e agire attraverso il routing. La seconda scelta, dettagliata nella sezione a seguire, è ovviamente la più complessa da implementare ma anche la più efficiente.

5.2.3 Routing e Guards

Come descritto in Sez. 5.1.2, le applicazioni tradizionali gestiscono interamente il routing delle pagine lato server grazie ad un meccanismo (basato generalmente su controller) che è capace di capire a quale risorsa si riferisce la richiesta e smistarla alle componenti deputate per la sua elaborazione. La migrazione sul client della generazione delle viste comporta la necessità di un modulo apposito per la gestione del routing su questo, poiché il server è adesso deputato a fornire esclusivamente dati che descrivono un modello e che popoleranno le viste. Angular mette a disposizione un insieme di metodi per la navigazione delle pagine attraverso il codice Javascript, ma anche delle callback molto utili che possono essere sfruttate nel ciclo di vita di un evento di navigazione:

- NavigationStart: evento innescato quando si avvia la navigazione.
- RoutesRecognized evento innescato quando il modulo analizza l'URL e riconosce il percorso.
- RouteConfigLoadStart evento innescato prima che la configurazione del percorso sia caricata.
- RouteConfigLoadEnd evento innescato dopo la configurazione del percorso (*lazy mode*).
- NavigationEnd evento innescato quando la navigazione termina correttamente.
- NavigationCancel evento innescato quando la navigazione è cancellata, ad esempio a causa di una Guard che interrompe la navigazione.
- NavigationError evento innescato quando si un errore imprevisto.

Possono esservi delle situazioni in cui l'utente non è autorizzato a visualizzare alcune pagine, a causa ad esempio di mancata autenticazione o perché non sono stati eseguiti degli step precedenti (come nei form complessi per la creazione di risorse), per cui il programmatore deve gestire tale situazione ed eventualmente ridirigere l'utente verso le giuste pagine; non si tratta solo di una questione di sicurezza ma anche di un modo per garantire il giusto flusso di navigazione. Piuttosto che far reinventare la ruota per mantenere uno stato circa le azioni dell'utente nei servizi, Angular mette a disposizione il meccanismo delle Guards, grazie al quale si ispeziona la rotta e viene eseguita una logica prima che la navigazione effettivamente avvenga, eventualmente attivandola e procedendo. Per ogni component da controllare è necessario specificare un array di guards da attivare per quella rotta, la cui logica interna deve soddisfare i requisiti dati. Tale logica è specificata all'interno del metodo canActivate() che ritorna un Observable<boolean> poiché si tratta di un meccanismo asincrono: se la rotta può essere navigata, il valore sarà *true*. Viceversa, se non può essere navigata, vi sono due possibilità: ridirigere l'utente verso una nuova rotta, attraverso la navigazione con i metodi del routing di Angular, oppure restituire *false* per non cambiare pagina, innescando l'evento NavigationCancel.

```
canActivate(): Observable<boolean> {
  return this._authNService.authNChange.map(
    isLogged => {
      if (isLogged === AuthNState.LoggedIn) {
        return true;
      } else {
        this._router.navigate([this.welcomePath]);
        return false;
      }
    }
  ).take(1);
}
```

Figura 5.12: Esempio di guard per il login.

La figura 5.12 riporta l'implementazione di una guard per l'abilitazione del routing solo con autenticazione. Tale guard può essere impiegata ad esempio nella Home Page e tutti gli altri componenti che necessitano dell'autenticazione utente prima di essere navigabili. In caso di mancata autenticazione, l'utente viene immediatamente ridiretto verso la pagina di "welcome", attraverso cui si può autenticare o navigare le risorse pubbliche (AboutPage, HelpPage, ecc.).

Gli scenari di utilizzo delle guards sono molteplici e non limitati soltanto all'esempio visto. Vi sono altri metodi che è possibile utilizzare per costruire un certo flusso di navigazione, tra cui canActivateChild() e canDeactivate(), utili rispettivamente per abilitare la navigazione delle viste figlie e per prevenire la perdita di dati. Ad esempio, il metodo canDeactivate può mostrare un popup se l'utente ha modificato un contenuto senza salvarlo o pubblicarlo, in modo da renderlo cosciente circa eventuali conseguenze come la perdita di dati. Si possono infine pensare a guards complementari, ad esempio la pagina di welcome può essere mostrata solo se l'utente non è autenticato, per cui si può creare una guardia con logica inversa rispetto alla precedente, eventualmente ridirigendo verso la home page se l'utente è autenticato ma naviga manualmente il path "welcome", facendo ovviamente attenzione a non far collidere le due logiche.

Con il meccanismo delle guards è stato possibile costruire la macchina a stati circa la navigazione dell'applicazione da parte dell'utente, in modo da guidarlo correttamente tra le pagine. In tal modo si evitano situazioni in cui la scrittura manuale di URL, mancanza di autenticazione o altro possano compromettere la regolare logica di esecuzione dell'applicazione, pensata dal programmatore e raramente seguita dall'utente.

5.2.4 Observable per il pattern publish/subscribe

Il pattern publish/subscribe (o publisher/subscriber) è uno stile architetturale utilizzato per la comunicazione asincrona basata su messaggi tra entità (oggetti, processi, macchine separate, e così via). I mittenti dei messaggi (detti publisher) non programmano l'invio diretto ai destinatari (detti subscriber). Piuttosto classificano tali messaggi secondo dei criteri, a prescindere da quali ascoltatori possano esservi, mentre i destinatari sottoscrivono un interesse per ricevere messaggi di una o più classi, senza sapere se e quali mittenti esistono. È una tipologia di comunicazione che si basa sul più generico meccanismo con coda di messaggi, offre buona scalabilità nella costruzione software grazie al disaccoppiamento tra le due parti, ma con una limitata flessibilità circa le modifiche sulla struttura dati dei messaggi pubblicati. Il disaccoppiamento avviene attraverso l'utilizzo di un broker, come entità embedded nel framework o come entità a sé stante (RabbitMQ e Apache Kafka sono alcuni esempi). I subscriber ricevono un sottoinsieme dei messaggi pubblicati, attraverso un meccanismo di selezione basato su filtraggio per argomento o contenuto. Le due tecniche comportano responsabilità diverse per gli attori in gioco: la prima riguarda il publisher, che deve obbligatoriamente classificare il messaggio impostando una classe di appartenenza, il subscriber deve solo iscriversi al canale logico. Il filtraggio basato su contenuti prevede che siano filtrati dal ricevente, cioè è questo che decide se trattenere il messaggio per l'elaborazione o scartarlo, in base ad una serie di vincoli imposti, il publisher deve solo emetterlo. In presenza di un broker esterno la prima tecnica di filtraggio è anche la più efficiente, proprio per non caricare i destinatari.

La Sez. 5.1.1 riporta i dettagli teorici del pattern "observer", molto comodo per implementare il meccanismo di comunicazione appena descritto. L'implementazione necessita di un servizio per esporre le funzionalità di publishing e subscribing alle parti che lo utilizzeranno. Il servizio gestisce un Subject basato su canali logici, in modo da favorire il filtraggio per chi si iscrive al canale.

```
@Injectable()
export class PubSubService {
  private publishSubscribeSubject:Subject<any> = new Subject();
  emitter:Observable<any>;

  constructor() {
    this.emitter = this.publishSubscribeSubject.asObservable();
  }

  publish(channel:string, event:any):void {
    this.publishSubscribeSubject.next({
      channel: channel,
      event: event
    });
  }

  subscribe(channel:string, handler:((value:any) => void)):Subscriber {
    return this.emitter
      .filter(emission => emission.channel === channel)
      .map(emission => emission.event)
      .subscribe(handler);
  }
}
```

Figura 5.13: Implementazione del pattern publish/subscribe.

La figura 5.13 riporta il servizio per l'implementazione del meccanismo publish/subscribe con il meccanismo degli observer. L'emitter è a tutti gli effetti un proxy (inizializzato nel costruttore) in cui i vari subscriber si iscriveranno alla parte observable, come dai component o da altri servizi. Più in dettaglio, il metodo *publish* ammette due parametri, una stringa come identificativo del canale e un oggetto di qualsiasi tipo (typescript "any") come elemento del broadcast. Tutto viene in realtà emesso come un singolo oggetto Javascript avente due proprietà, il canale e l'evento stesso, in modo da facilitare il filtraggio in fase di sottoscrizione desiderata. Il metodo *subscribe* è più complesso: dell'emitter, è necessario filtrare le emissioni (elementi del broadcast) il cui canale coincide con quello passato come primo parametro del metodo stesso, estrarre l'evento dall'oggetto (tramite la funzione map()) e sottoscrivere l'handler passato come secondo parametro, attraverso il metodo subscribe dell'observable che ritorna un elemento di tipo Subscriber e che sarà direttamente ritornato dal metodo subscribe del servizio. Il broker di tale pattern è il framework stesso, attraverso la libreria RxJS che implementa in Javascript il meccanismo dell'Observer.

L'utilizzo del servizio è semplice, basta dichiarare un oggetto nel costruttore come dipendenza del tipo PubSubService dell'entità, per poi utilizzare i metodi sopra discussi. Nel lavoro svolto, tale servizio è stato utilizzato in varie parti, tra cui per la comunicazione tra i component per la vista master-detail nella pagina di ricerca, poiché risulta impossibile il binding tra componenti che sono incapsulati con profondità grande, come spiegato nella sezione che segue.

5.2.5 Interfaccia master-detail per la ricerca

Nel mondo web e mobile vi sono dei pattern per la costruzione di interfacce grafiche ormai popolari, tra questi rientra la tipologia master-detail. Si tratta di un pattern molto utile e di facile intuizione, utilizzato nell'esplorazione delle risorse in gioco e composto principalmente da due parti: una lista (master) consente all'utente di scorrere gli elementi, possibilmente raggruppati secondo un certo criterio, ogni cella della lista solitamente mostra pochi dati, per poter dare subito qualche informazione utile all'utente; attraverso tale lista, si può scegliere un elemento da visualizzare in modo dettagliato, infatti la seconda parte (detail) è deputata a rappresentare in forma estesa le informazioni dell'elemento scelto. Le due viste non necessariamente si trovano insieme nella stessa schermata, sia per motivi di progettazione che per motivi di capienza: solitamente si trovano insieme, ma Gmail ad esempio ha una struttura mater-detail basata su due schermate distinte, nonostante il browser web non ridimensionato nel desktop sia abbastanza largo da poter consentire le due viste una di fianco all'altra; se ristretto o utilizzato in display più piccoli, potrebbero entrare in gioco meccanismi di design *responsive*, utili per una diversa sistemazione degli elementi, che mostrerebbero in modo esclusivo una delle due interfacce. Grazie ai meccanismi offerti da Angular, è possibile implementare tale pattern di navigazione in vari modi, tra cui con il binding di eventi e proprietà, come mostrato dalla figura seguente.

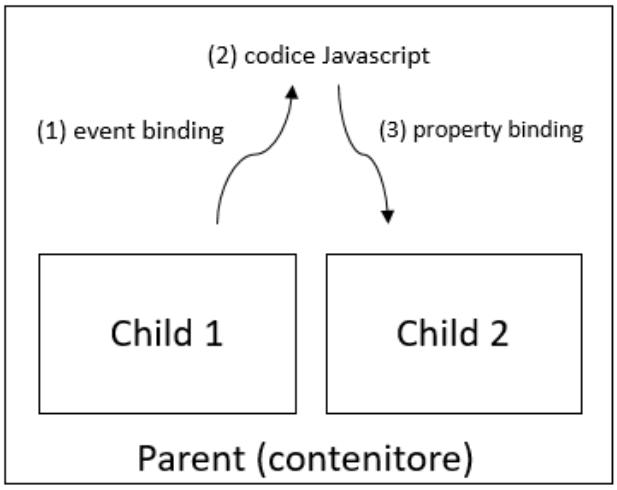


Figura 5.14: Comunicazione tra component (semplice).

Supponendo che il child 1 sia la lista e il child 2 sia il dettaglio, una comunicazione può avvenire attraverso il binding standard offerto da Angular. Ma si tratta di un caso molto semplice e raramente presente, di solito vi sono molti più component in gioco e ciò richiede che possa esservi una comunicazione tra tutti i componenti incapsulati, che può risultare molto scomoda dal punto di vista della programmazione e delle performance, poiché è necessario dover fare comunicare il component padre con quelli figli, passando per tutti gli eventuali component intermedi: ciò vuol dire che tali component intermedi saranno scarsamente riutilizzabili e comunque "sporcati" da codice che serve solo come passacarte.

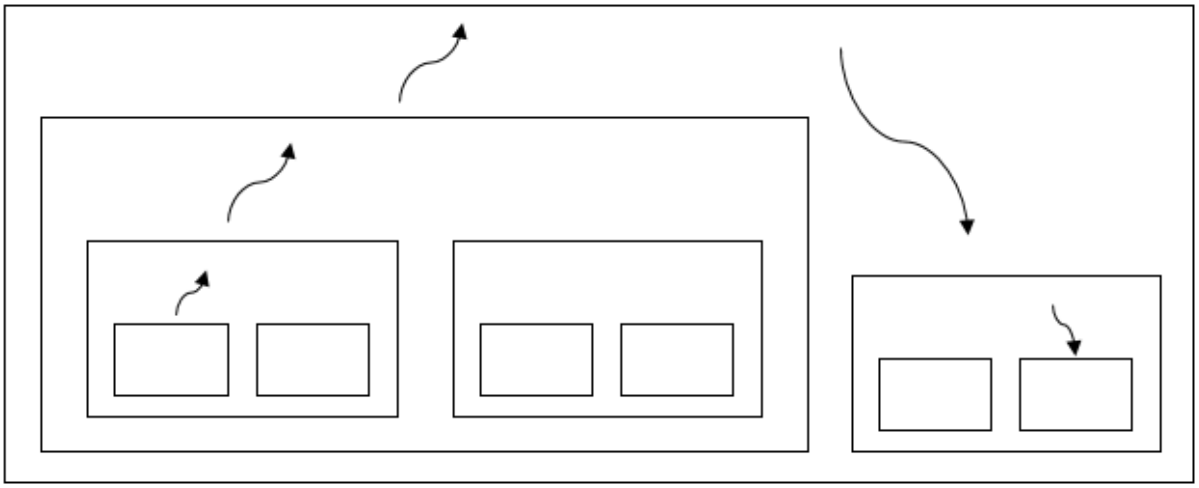


Figura 5.15: Comunicazione tra component (complessa).

Come mostra la figura 5.15, la comunicazione tra un figlio con un livello di incapsulamento arbitrariamente profondo e un altro a qualsiasi livello necessita un binding tra tutti gli elementi intermedi, che a runtime si traduce in una sequenza di eventi rimbalzanti e raramente manipolati man mano che si sale o si scende di livello. I component intermedi sono molto frequenti nello sviluppo dei frontend perché svolgono il ruolo di contenitori, dunque fanno da tramite per la comunicazione tra i component che contengono, ma la comunicazione tra component contenuti a diversi livelli di profondità darebbe vita ad un'applicazione mal progettata e poco manutenibile al crescere della logica di business associata al frontend. Tale situazione può essere evitata con l'utilizzo di un differente meccanismo di comunicazione tra gli elementi.

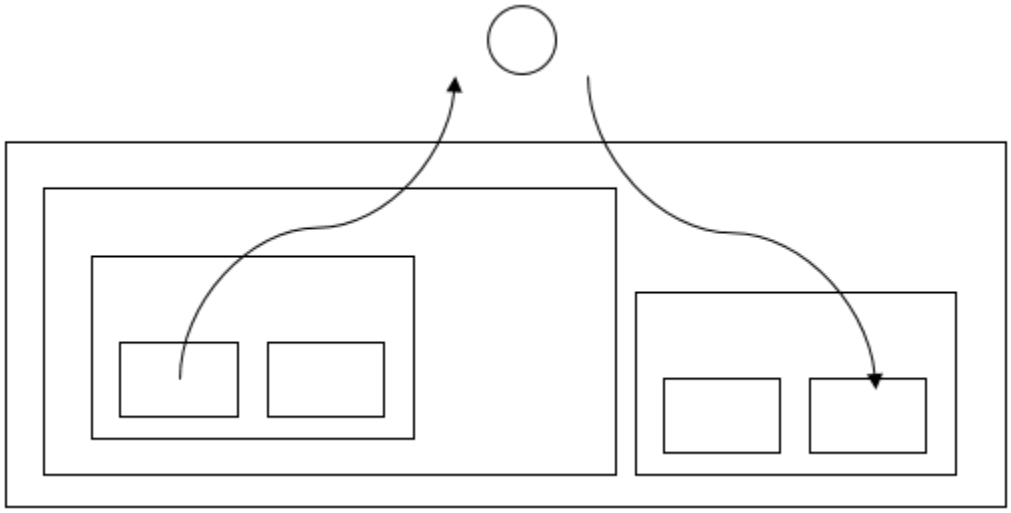


Figura 5.16: Comunicazione tra component (semplice).

La figura 5.16 riporta uno schema di comunicazione tra componenti attraverso il meccanismo publish/subscribe, in tal modo il component figlio effettua uno stream di messaggi che sarà esclusivamente ricevuto dagli altri componenti che eventualmente si iscriveranno al canale logico associato. Ciò aiuta a mantenere le classi intermedie pulite, leggere e a rendere i component riutilizzabili, poiché ognuno ha una certa competenza (separazione delle responsabilità).

Per l'implementazione della vista master-detail, piuttosto che costruire dal nulla una lista visuale di elementi, si è scelto di utilizzare una libreria esterna. La scelta non riguarda solo evitare di commettere errori di implementazione o risparmiare tempo utile, ma cercare una soluzione efficiente per una situazione in cui la quantità di dati da mostrare (a priori sconosciuta) può penalizzare le prestazioni. Si tratta di una delle tante librerie di terze e parti open-source che la

community di Angular mette a disposizione, offre un meccanismo efficiente per la visualizzazione di elementi, con una sostituzione dinamica dei dati (attraverso il binding) man mano che l'utente scorre la lista: più in dettaglio, la libreria popola un insieme fisso di celle con i dati in memoria, dando all'utente la sensazione di scorrere una lista; dal punto di vista visuale la lista esiste davvero, ma dal punto di vista operativo avviene una sostituzione on-the-fly del contenuto delle celle, senza crearne nuove man mano che sopraggiungono altri dati, in modo da non penalizzare le prestazioni. Spesso le interfacce master-detail attingono a dati dal backend, la libreria offre anche un meccanismo per la gestione di eventi come OnScroll, onTopReached, onBottomReached, e altri, per poter eseguire una certa logica al verificarsi di questi: la callback onBottomReached è utile per fare il *fetch* dei dati dal backend, aggiungendoli (virtualmente) in coda alla lista. In realtà la libreria offre un meccanismo per la sola visualizzazione efficiente di una lista di elementi, non è detto che debba far parte di una vista master-detail, è a carico del programmatore costruire tale vista sfruttando il component al suo interno. La realizzazione necessita di un component padre che faccia da contenitore di lista e dettaglio, a loro volta scomponibili in component sempre più semplici. Viene ripreso l'esempio mostrato dalla figura 5.10, dettagliato con codice nella figura 5.18.

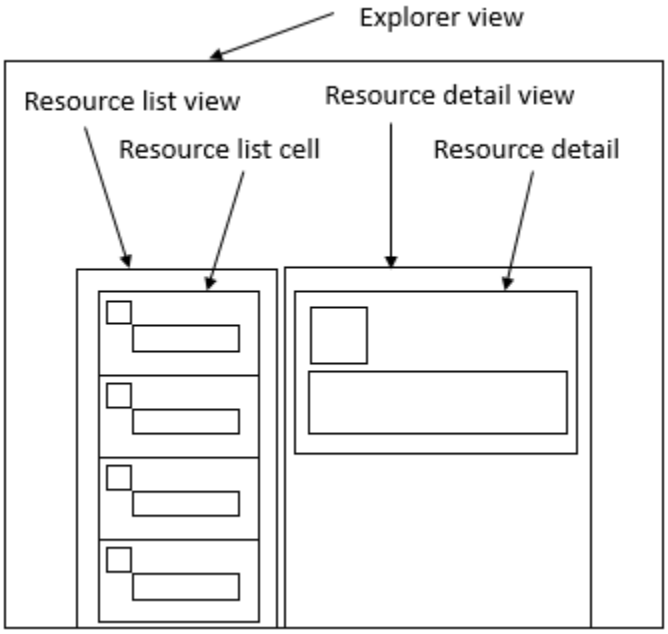


Figura 5.17: Schema della vista master-detail.

Il componente "explorer-view" è contenitore di altri due contenitori: contenitore lista, all'interno di cui vi è il component virtual-scroll della libreria citata, e il contenitore dettaglio, contenente dei component per i dettagli dell'elemento scelto.

```
<div virtualScroll class="center-list virtual-scroll"
     [items]="buffer"
     (update)="centerList = $event"
     (end)="fetchMore($event)">

  <app-center-list-item
    appLinkPointer
    *ngFor="let center of centerList"
    [center]="center"
    (click)="onSelectedCenter(center)">
  </app-center-list-item>

  <div *ngIf="loading" class="loader">Loading...</div>
</div>
```

Figura 5.18: Utilizzo del component virtual-scroll.

Il codice riportato in figura 5.18 mostra l'utilizzo del component virtual-scroll e il binding per la comunicazione. Si tratta di un binding sull'evento *click* del mouse, cioè al click su un elemento della lista si invoca il metodo onSelectedCenter() che attiva la pubblicazione di un evento su un certo canale logico, attraverso il meccanismo publish/subscribe descritto in Sez. 5.2.4. Gli eventi *end* e *update* aiutano a fare il fetch dal backend e a popolare la lista con nuovi elementi in coda; durante questa operazione, una sentinella booleana consente di mostrare in basso alla lista un messaggio di caricamento.

```
onSelectedCenter(center: CenterListItem) {
  this.centerSelectedEvent.emit(center);
}
```

Figura 5.19: Emissione evento di selezione.

```
centerSelected(center: Center): void {
  const CENTER_MD_CHANNEL = this._globalChannel.getCenterMasterDetailChannel();
  this._pubSubService.publish(CENTER_MD_CHANNEL, center);
}
```

Figura 5.20: Invocazione handler dell'evento di selezione.

La figura 5.19 riporta l'invio di un evento al verificarsi del click (selezione risorsa), mentre la figura 5.20 mostra le azioni compiute dall'handler, invocato quando si riceve l'evento (event binding). La sequenza è tale per cui al click viene emesso l'oggetto, corrispondente ad una certa cella della lista, per poi trasmetterlo sul canale. È chiaro che deve esservi qualche ricevente, in tal caso è il contenitore del component che dovrà mostrare i dettagli dell'elemento ricevuto, inizializzato nel costruttore.

```
constructor(private _pubSubService: PublishSubscribeService,
            private _globalChannel: GlobalChannel) {
  const CENTER_MD_CHANNEL = this._globalChannel.getCenterMasterDetailChannel();
  this._centerSubscription = this._pubSubService
    .subscribe(CENTER_MD_CHANNEL,  handler: item => {
      this.centerDetail = item;
    });
}
```

Figura 5.21: Iscrizione al canale logico.

L'iscrizione al canale logico consiste di un handler da invocare alla ricezione di un nuovo contenuto. In tal caso, la sola azione da compiere è aggiornare il contenuto della proprietà *centerDetail*, che attraverso il property binding mostrerà nella vista i dettagli della risorsa ricevuta.

Nonostante il broadcast dell'intero oggetto possa sembrare onerosa, in realtà rimane un meccanismo efficiente, perché l'esecuzione dell'applicazione prevede una sola schermata di tipo master-detail e quindi una sola coppia formata da un publisher e un subscriber. Grazie a questo tipo di implementazione è stato possibile costruire una interfaccia di esplorazione delle risorse molto intuitiva, user-friendly e dalle prestazioni efficienti.

5.2.6 Completamento automatico per la ricerca

La ricerca automatica basata su suggerimenti, ranking e recensioni sicuramente aiuta l'utente finale nell'esplorazione delle risorse in gioco, ma offrire la funzionalità di ricerca testuale di certo non la rende un'applicazione peggiore. I meccanismi di ricerca possono essere vari, ma l'immediatezza nel fornire i risultati è la cosa che rende davvero ottima la funzionalità: il completamento automatico con suggerimenti, la possibilità di filtrare i contenuti con dei parametri o con parole chiave sono alcune caratteristiche che è bene includere per facilitare la ricerca. Il filtraggio si realizza facilmente con il binding di eventi e proprietà, perché i vari widget come slider, entry text, checkbox e altri sono semplici modificatori di uno stato temporaneo del component; tale stato serve a filtrare i dati da mostrare nelle viste rispetto a quelli che non soddisfano le esigenze di ricerca. Ad ogni evento associato ai widget corrisponde una modifica dello stato, a cui segue potenzialmente una modifica delle proprietà interne, che fa scattare l'aggiornamento della vista. La funzionalità di suggerimento parole o completamento automatico è qualcosa di più complesso, che spesso richiede una comunicazione con il backend. Gli observable della libreria RxJS possono essere sfruttati in un contesto di questo tipo, essendo appunto utili alla costruzione di meccanismi "reactive" potenti e robusti.

L'implementazione consiste nell'effettuare una query al backend avente come payload il testo di ricerca e magari altri dati (posizione, lingua, tipo di utente, ecc.). Il completamento delle parole viene fatto man mano che l'utente scrive, per cui l'evento su cui fare il binding è il *keyup* (tastiera) oppure utilizzare le proprietà del modulo FormControl come *valueChanged*, modulo utile per le operazioni sui form come validazione e altro, in tal caso utile per notificare il cambiamento del valore di input. Solo a tal punto potrà essere effettuata la richiesta HTTP e i risultati saranno da elaborare e mostrare a video in una certa forma. Esiste però un problema operazionale che può penalizzare il backend, poiché le parole lunghe fanno scattare un evento ad ogni carattere digitato, è necessario un "debouncing" dell'input. Gli observable di RxJS entrano in gioco proprio a tale scopo, espongono un metodo DebounceTime(delay) che crea un nuovo observable non prima del ritardo specificato e con l'ultimo stato registrato. Tale metodo deve essere chiamato prima di effettuare la query al backend, impostando un tempo ragionevolmente buono per far apparire in fretta i suggerimenti e al contempo non penalizzare le prestazioni, così da debellare il problema. Esiste ancora un altro problema da risolvere, ossia quello dell'input duplicato. Per input duplicato si intende il caso in cui l'insieme delle parole non muta, ma l'input sì: l'aggiunta e l'immediata cancellazione degli spazi sono un esempio di input senza modifiche utili. Viene ancora in aiuto la libreria RxJS, con il metodo distinctUntilChanges() che in automatico provvede a scartare un duplicato se è già stato registrato in precedenza. Ultima ottimizzazione necessaria è evitare che query multiple arrivino in disordine: il caso è molto comune, basti pensare che la ricerca di due parole scatenerebbe già due query. Bisogna quindi evitare che una query più vecchia arrivi dopo una query più recente, sicuramente perché non necessaria. I motivi circa il "sorpasso" di una query o del suo risultato rispetto ad un'altra possono essere vari: dalla latenza di rete al backend distribuito, che in caso di alto carico di lavoro può smistare le richieste alle parti meno cariche e generare un output più in fretta. A prescindere, ad ogni nuova query è necessario dover annullare tutte le precedenti, perché il payload di ricerca utile è l'ultimo digitato dall'utente. Ciò è possibile con gli Observable ma non con le Promise, RxJS consente di annullare le sottoscrizioni di cui non si ha ancora un risultato utile ricevuto attraverso il metodo switchMap(), che provoca il conseguente annullamento delle richieste in sospeso.

```
@Component({
  selector: 'search',
  template: `
    <input [formControl]="queryField">
    <p *ngFor="let result of results">{{result}}</p>
  `
})
export class SearchComponent {
  results:Array<string> = [];
  queryField:FormControl = new FormControl();

  constructor(private apiService_:APIService) {
    this.queryField.valueChanges
      .debounceTime(200)
      .distinctUntilChanged()
      .switchMap(query => this.apiService_.search(query))
      .subscribe(result => this.results.push(result));
  }
}
```

Figura 5.22: Auto completamento di ricerca.

La figura riporta un esempio di codice completo per il meccanismo descritto. Il template, semplice per pura dimostrazione, è composto da un input con event binding su *queryField* e una direttiva *ngFor* che provvederà a mostrare gli elementi nell'array *results*. La sequenza di eventi da chiamare è strutturata esattamente come descritto in precedenza: al cambiamento di un input si ritarda di un certo tempo, si verifica che il nuovo payload sia utile e distinto dai precedenti, si annullano tutte le precedenti query e si sottoscrive una callback per elaborare il risultato. Attraverso il property binding, la modifica della proprietà provocherà l'aggiornamento della vista, in particolare della sequenza costruita dalla direttiva ngFor.

Il meccanismo appena descritto chiude il discorso relativo alla costruzione del frontend e in generale dello sviluppo applicativo del social network. Il Capitolo successivo tratta la strategia di deployment per il backend e per il frontend web, analizzando i moderni principi di virtualizzazione, i problemi comuni che si riscontrano e le soluzioni adottate per fronteggiarli.

6. Deployment

Il ciclo di vita di un prodotto software è un processo molto complesso che si compone di varie fasi, dall'idea, alla realizzazione, messa in campo e il suo mantenimento nel tempo. Ogni fase a sua volta può essere suddivisa in altre fasi, destinate ad essere gestite da uno o più entità, secondo una certa gerarchia. Nel seguente Capitolo saranno discusse le strategie utilizzate per il deployment dell'applicazione. I problemi da fronteggiare non sono pochi, a causa dei limiti imposti soprattutto dalle scarse risorse umane ed economiche di cui la startup dispone. Come detto all'inizio del presente documento, la startup si trova nella fase inziale e il team di sviluppatori è composto da un solo membro, al quale sono state assegnate tutte le mansioni per la realizzazione. Nonostante ciò, lo scopo è quello di gettare le basi per un buon prodotto software facilmente manutenibile nel tempo, in modo da poter ottenere il massimo con il minimo delle risorse, finché possibile.

Dopo aver analizzato le varie strategie utili per il deployment di applicazioni che richiedono un alto grado di scalabilità e le varie strategie di virtualizzazione, sarà discussa la scelta fatta in relazione a quanto possibile.

6.1 Software deployment

Per "software deployment" si intende il processo necessario per la preparazione e messa in campo di un prodotto software in un ambiente specifico. Questo implica installazione, configurazione, testing e varie modifiche per il suo corretto funzionamento, attraverso interventi manuali o con software automatizzati. Queste sono a tutti gli effetti fasi del ciclo di vita della costruzione del software. Le modalità di deployment possono essere varie: installazione diretta, parallela, pilota o scaglionata. L'installazione diretta sostituisce completamente il vecchio sistema, con tutti i rischi del caso, dato che un problema non riscontrato in fase di sviluppo può compromettere il suo funzionamento durante l'utilizzo da parte degli utenti; tale soluzione si adotta quando non vi sono valide alternative per il deployment, oppure quando vi sono cambiamenti urgenti e radicali. L'installazione parallela consiste nella messa in campo del nuovo sistema insieme al vecchio, cosicché quando il nuovo raggiunge un discreto livello di affidabilità, potrà sostituire totalmente il vecchio. L'installazione pilota, similmente alla parallela, consiste nella messa in campo del nuovo prodotto a fianco del vecchio, ma solo per piccole quantità, in modo da sperimentarne il funzionamento. Infine, l'installazione scaglionata prevede che il vecchio sistema venga aggiornato per parti, dunque si ritrovano nella stessa unità di esecuzione sia parti del vecchio che del nuovo. Trattandosi di applicazioni software, la messa in campo e l'esecuzione avviene attraverso server raggiungibili con la rete Internet. Quando si tratta di un'azienda con una certa disponibilità finanziaria, la scelta è quella di costruire internamente il sistema su cui fare il deployment: è il caso di grandi compagnie, banche e altri enti che costruiscono i propri datacenter. Nel caso di entità che non dispongono di grandi risorse finanziarie, la scelta è quella di appoggiarsi ad una infrastruttura esterna che offra servizi su misura di vario genere. In realtà, mantenere un proprio datacenter non è affatto semplice in termini di risorse da impiegare, spesso anche le grandi compagnie si affidano a entità specializzate in tale campo proprio per non doversi più occupare di mantenere l'infrastruttura o perché con una soluzione esternalizzata i costi possono drasticamente diminuire. In quest'ultimo caso i nomi in gioco sono tantissimi: Amazon Web Services, Google Cloud Platform, IBM Cloud e Microsoft Azure sono solo alcuni big che offrono servizi cloud.

6.2 Datacenter: infrastruttura privata vs pubblica

Un data center è un impianto utilizzato per alloggiare sistemi informatici e componenti associati, come i sistemi di computazione, storage e comunicazione. Generalmente è costruito con un certo grado di resilienza grazie alla ridondanza di componenti come alimentatori, connessioni per le comunicazioni, controlli ambientali interni (ad esempio per il condizionamento), vari dispositivi di sicurezza e ovviamente i nodi di calcolo, di rete e di storage. È una infrastruttura hardware a servizio di una o più aziende, le dimensioni possono variare da piccoli centri di elaborazione esclusive delle aziende private, fino a grandi centri sparsi per il mondo a disposizione dei clienti, offrendo varie forme di elaborazione dati e di servizi IT. Un datacenter è composto da "server farm", termine utilizzato per indicare una serie di server collocati in un unico ambiente in modo da poterne centralizzare la gestione, la manutenzione e la sicurezza. Spesso all'interno delle server farm vengono costituiti dei cluster per gestire in maniera migliore carichi di lavoro pesanti o critici (server email, web, database, rendering, GRID computing - infrastruttura di calcolo distribuito), attraverso una tipica architettura distribuita, garantendo al contempo affidabilità e tolleranza ai guasti, tramite appunto ridondanza fisica degli apparati.

I datacenter di qualche decennio fa erano ben diversi dagli attuali, erano composti da un insieme di calcolatori ognuno adibito a fornire specifiche funzionalità o servizi all'interno dell'azienda. Il grosso problema è stato l'eterogeneità delle risorse hardware, ovviamente non condivisibili tra le entità quando si verificavano dei problemi legati alle prestazioni: alcune macchine risultavano cariche in certi momenti della loro vita operativa, altre molto scariche, con un alto grado di variabilità nell'utilizzo delle risorse stesse. Lo spreco si verificava anche a causa della necessità di dover fare provisioning per il caso peggiore, un hardware doveva essere capace di gestire il picco massimo di utilizzo, pur se poi tale picco veniva raggiunto per pochissimo tempo. A ciò si affiancavano problemi che affioravano a lungo termine, come le dipendenze del software da uno specifico hardware e a volte l'impossibilità di sostituirlo con un altro nel caso in cui tale hardware fosse venuto a mancare sul mercato. Nella gestione del datacenter privato vi sono una serie di aspetti critici che devono essere attenzionati, sia dal punto di vista dei servizi interni o dei clienti, sia dal punto di vista legislativo e ambientale. Inoltre, non è affatto banale neppure la sua costruzione, a causa delle elevate spese che ci si ritrova a dover affrontare (*capex* – capital expenditure) oltre che il suo mantenimento (*opex* – operating expenditure). Per cui, se non è strettamente necessario costruire il proprio datacenter, è bene che le aziende migrino verso servizi cloud esterni.

Grazie ai risultati di ricerca ottenuti nel campo della virtualizzazione è stato possibile ottenere pian piano il consolidamento delle risorse. Ciò ha consentito di costruire interi datacenter con un pool omogeneo di risorse hardware, come motore esecutivo di un livello di astrazione che andasse ad emulare quel che prima era un risorsa hardware reale. Tutto ciò ha consentito un grande risparmio economico per le normali aziende, era sufficiente l'acquisto di centinaia di server uguali e utilizzarli con un software di virtualizzazione per a creazione di macchine virtuali su misura, ma ha portato altri big del mercato (o semplici investitori) alla costruzione di datacenter grazie ai quali vendere servizi cloud di tipo:

- IaaS – Infrastructure as a Service
- PaaS – Platform as a Service
- SaaS – Software as a Service
- CaaS – Container as a Service
- BaaS – Backend as a Service
- …
- XaaS – Everything as a Service

Il termine "cloud" oggigiorno è molto utilizzato, forse troppo: per cloud si intende la fornitura di risorse, dati, infrastrutture IT e servizi vari su misura e a vari livelli. Questa soluzione non per forza deve essere esposta al mondo esterno, magari una compagnia si appoggia a servizi esterni per una funzionalità interna. Il cloud consente un outsourcing delle risorse o dei servizi pubblici, raggiungibili tramite una connessione utilizzando la rete WAN o la connettività Internet dai dispositivi che li richiedono. È un modello per consentire l'accesso onnipresente a un pool condiviso di risorse di calcolo e storage configurabili (reti di computer, server di archiviazione, applicazioni e servizi) che possono essere rapidamente provvisti e rilasciati con il minimo sforzo di gestione. Il "cloud computing" indica un paradigma di erogazione di risorse informatiche, come l'archiviazione, l'elaborazione o la trasmissione di dati, chiamata in questo modo se il numero di processi istanziabili è maggiore del numero di nodi hardware, altrimenti si parla di GRID computing (infrastruttura di calcolo distribuito). Le soluzioni di cloud computing e storage forniscono agli utenti e alle imprese diverse capacità per memorizzare e elaborare i propri dati in centri dati privati.

Ciò consente di evitare i costi di creazione e mantenimento, consentendo alle stesse aziende di concentrarsi sulle loro attività fondamentali, piuttosto che impiegare tempo e risorse in gestione di infrastrutture IT. Grazie alle varie strategie messe a disposizione da chi offre questi servizi, il cloud computing consente di ottenere veloce deployment delle applicazioni, con una migliore gestibilità e minore manutenzione, consentendo alle squadre di adattare le risorse in modo più rapido per soddisfare la domanda di mercato, di per sé fluttuante e imprevedibile. Tali servizi esterni in genere sono basati su soluzioni *on demand*, ossia pagamento in base al consumo di risorse, oppure con allocazione statica di istanze, se è possibile fare un *provisioning* minimo di quanto necessario, in modalità flat; la scelta dipende dall'effettivo consumo di risorse, solitamente la flat è più economica della on demand a lungo termine. Un servizio di questo tipo consente alle aziende di scalare quando vi è l'esigenza, per poi diminuire le istanze quando non necessarie. Il cloud computing è ormai diventato un servizio utile e altamente richiesto grazie alle soluzioni offerte, al costo economico dei servizi, scalabilità e accessibilità su scala mondiale.

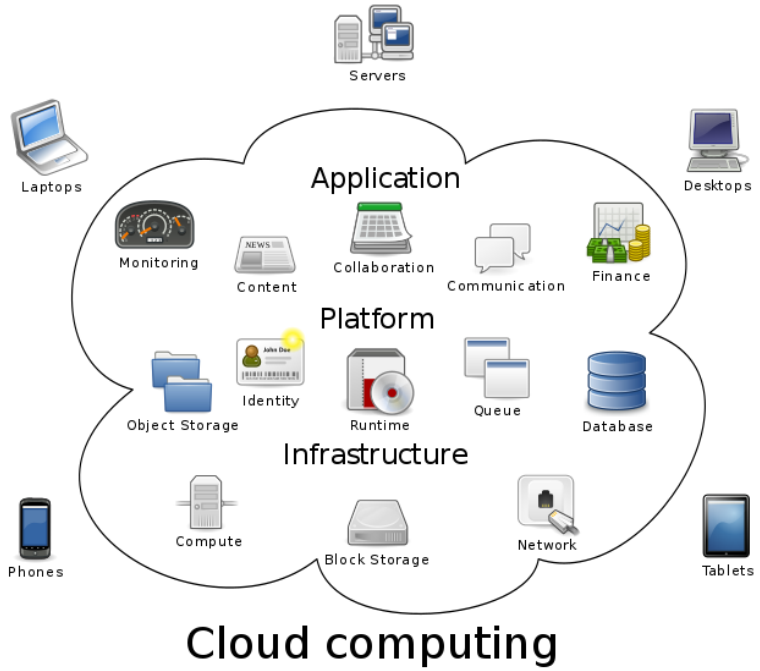


Figura 6.1: Illustrazione del concetto di cloud con i diversi livelli di astrazione.

E dunque no, datacenter e cloud non sono la stessa cosa. Non sono termini intercambiabili, pur se l'enfasi della novità porta spesso a confondere le due cose. Indistintamente da cloud pubblico, privato o ibrido, il "cloud" non fa mai riferimento a risorse fisiche o hardware, o a dove questo effettivamente risiede. Di solito si dice che il cloud risiede in qualsiasi datacenter, che è in esecuzione su qualsiasi hardware della server farm, come uno "strato" al di sopra di questo. Ma vale anche il contrario, nel senso che un cloud può ospitare un intero datacenter virtualizzato: è questo il significato vero di un servizio di tipo IaaS (Infrastructure As a Service). I vari tipi di servizi offerti (riportati nell'elenco precedente) sono molto diversi tra loro, varia il livello di dettaglio che l'acquirente può direttamente manipolare per la creazione di servizi su misura.

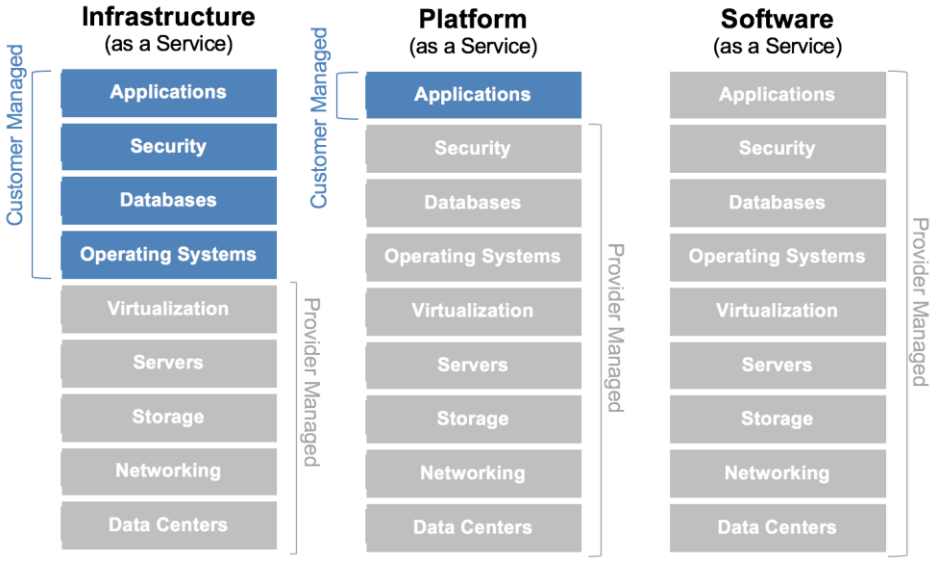


Figura 6.2: Illustrazione grafica: IaaS vs PaaS vs SaaS.

Un servizio IaaS è un outsourcing evoluto di tutte le risorse ICT, consente un livello basso di dettaglio e si ha a disposizione la virtualizzazione delle risorse hardware (CPU, RAM, storage e schede di rete); in altre parole, la flessibilità di un'infrastruttura fisica senza l'onere della gestione della parte hardware a livello di rete, server, storage, e soprattutto senza la preoccupazione di dover garantire elevati livelli di accessibilità dei servizi, cosa di cui si occupa il fornitore cloud attraverso un SLA (Service Level Agreement). In questo caso è possibile installare un certo sistema operativo e creare le applicazioni, gestendo in autonomia la distribuzione del carico computazionale su più istanze. Nella modalità PaaS è il fornitore a gestire l'infrastruttura fino al sistema operativo, sicurezza e storage, il cliente dovrà solo occuparsi di sviluppare la sua applicazione, facendo però attenzione al tipo di servizio fornito e ai linguaggi supportati dalla piattaforma scelta: è importante considerare con attenzione i linguaggi di programmazione, librerie o servizi, strumenti dedicati per l'ambiente scelto (Unix o Windows, enterprise o community, e così via). Gli elementi che costituiscono la PaaS permettono di programmare, sottoporre a test, implementare e gestire le applicazioni senza i costi e la complessità associati all'acquisto, alla configurazione, all'ottimizzazione e alla gestione sia del software che dell'hardware necessari alle attività di sviluppo. SaaS è la tipologia di servizio di più alto livello offerto all'utente finale, l'esempio più comune di servizi è quello delle web mail oppure delle Google Apps. Con il SaaS, chi fruisce del servizio non controlla l'infrastruttura che supporta il software: a livello di rete, server, storage e sistemi operativi la gestione è interamente a carico del provider. Il cliente può solo decidere se limitare le funzionalità del software, stabilendo ad esempio criteri di gestione delle identità e delle priorità degli accessi tramite un set di configurazione dedicate.

Pian piano approdano sulla scena altre tipologie di servizi IT come DaaS, CaaS, BaaS, e XaaS. Brevemente, Desktop come servizio consente a un cliente di ospitare l'intero ambiente di elaborazione desktop tramite un provider di servizi cloud. I container come servizio sono un modello in cui le organizzazioni IT e gli sviluppatori possono lavorare insieme per costruire, migrare e gestire le proprie applicazioni ovunque, grazie ai container; si può intendere come la IaaS, PaaS o SaaS ma con l'utilizzo esclusivo di container piuttosto che software direttamente installato sulla piattaforma, con tecnologie di gestione container come Docker. Backend come servizio è un modello per fornire a sviluppatori di app web o mobile un modo per collegare le loro applicazioni alle API esposte da un backend applicativo, fornendo allo stesso tempo funzioni quali la gestione degli utenti, le notifiche push, e l'integrazione con servizi di social networking, storage e altro, un esempio è Firebase di Google. Grazie a ciò si possono "softwarizzare" server, storage, gli apparati di rete come router o switch, gli apparati di sicurezza come i firewall o gli IDS e via dicendo.

6.3 La virtualizzazione

La realizzazione di datacenter moderni non sarebbe stata possibile senza la virtualizzazione. Per virtualizzazione si intende la capacità di creare qualcosa che emuli un comportamento reale, ci si può riferire ad hardware, software, storage, networking o risorse di altro tipo. Ma il concetto di virtualizzazione per il computing è stato esteso su più fronti e ultimamente se ne sfrutta un particolare tipo per risolvere i problemi delle dipendenze software, configurazione, mantenimento e replicazione dei sistemi. Nel creare servizi distribuiti sono richiesti una varietà di software e versioni di dipendenze che nel tempo possono causare problemi di mantenimento. È possibile che le nuove versioni software rilasciate siano incompatibili con le precedenti e un aggiornamento delle dipendenze può compromettere il funzionamento del sistema stesso, rendendo necessaria la sua modifica per riadattarlo. Ma non tutti i sistemi possono essere ristrutturati e alcuni devono coesistere con altri fino ad una totale ricostruzione. Durante tale fase è necessario mantenere la coerenza tra tante parti e molteplici versioni, spesso incompatibili tra loro, con alta probabilità di conflitti o che il sistema non funzioni come ci si aspetta. A prescindere dalla strategia architetturale del sistema, serve una soluzione per far coesistere le entità. Come se non bastasse, ai problemi operazionali si aggiungono quelli di sviluppo. Spesso l'ambiente di sviluppo locale è ben più semplice in termini di configurazione, ma non è detto che in modalità *production* il tutto continui a funzionare come ci si aspetta e senza configurazioni aggiuntive, che ovviamente andranno fatte ma possono essere in conflitto con quanto sviluppato in locale. Una delle tecnologie possibili per risolvere la maggior parte dei problemi citati è Docker (Sez. 6.3.3), che sfrutta i container per isolare e far coesistere un prodotto software e le sue dipendenze con altri software, anche in varie versioni, presenti nello stesso sistema. Esistono principalmente due tipi di virtualizzazione, che differiscono tra loro nel tipo di astrazione e nei meccanismi usati: virtualizzazione della macchina hardware e virtualizzazione del contesto software, Docker si basa proprio su quest'ultima.

6.3.1 Virtualizzazione classica

Grazie alla virtualizzazione hardware è stato possibile consolidare le risorse un tempo eterogenee e offrire maggior supporto ai vari ambienti di esecuzione, spesso con requisiti hardware e software differenti tra loro, bilanciandole dove necessario per evitare sprechi e sovraccarichi inutili. L'emulazione dell'hardware nasce con i mainframe, elaboratori con grandi risorse di calcolo di cui più client possono beneficiarne mediante la creazione di uno strato hardware virtualizzato per ciascuno, corrispondente ad un pool di risorse ridotto. Si ottiene una macchina virtuale su cui è possibile installare un sistema operativo, detto Guest. Il sistema operativo alla base (Host OS) può essere un "full-fledged" (Unix-Based o Windows) per facilitare alcune funzionalità, oppure si può adottare un software più

leggero per la sola virtualizzazione. Entrambe le soluzioni dispongono comunque di uno strato software chiamato Hypervisor. Questo è l'unico che può compiere realmente le operazioni sulle risorse hardware sottostanti, offrendo al contempo un interfaccia per le parti che si appoggiano ad esso. Ogni macchina virtualizzata esegue il suo Guest OS, all'interno del quale vi sono un gruppo di applicazioni in esecuzione. Dunque si tratta di una vera e propria Virtual Machine (VM) che è possibile fermare, spostare o di cui ripristinarne l'esecuzione.

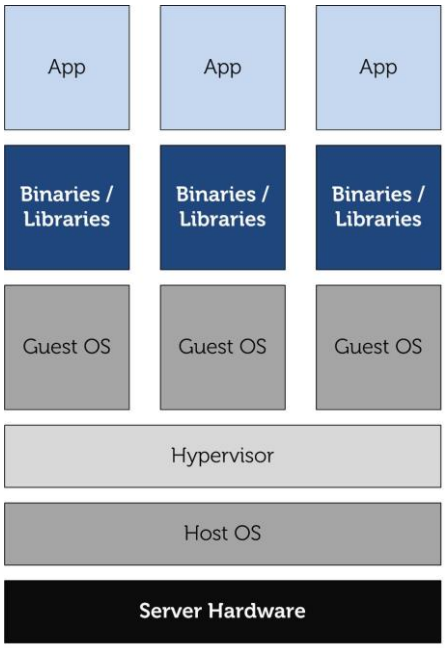


Figura 6.3: Illustrazione della virtualizzazione hardware.

Come mostra la figura 6.3, vi è un doppio strato di sistemi operativi, un Host OS e vari Guest OS. Brevemente, i vantaggi sono legati al fatto che è possibile usare un Guest OS come se alla base vi fosse un hardware esclusivo per questo. Eppure i limiti e gli svantaggi sono non pochi, vi è una doppia barriera user-space e kernel-space da attraversare più volte per le operazioni che richiedono context switch, di cui una per il Guest OS e l'altra per l'Host OS. Inoltre ogni Guest OS richiede uno spazio di setup non trascurabile, considerato come uno spreco di storage.

6.3.2 Virtualizzazione basata su container

L'approccio alternativo è quello basato sulla semplificazione dello stesso contesto, evitando uno spreco di risorse. Si guarda ad un'applicazione in esecuzione nel contesto dello stesso sistema operativo, piuttosto che un sistema operativo al di sopra di un altro. In qualche modo è garantito lo stesso grado di isolamento tra i processi come accade con l'Hypervisor, grazie ad un nuovo processo che farà riferimento a strutture kernel distinte dagli altri. Il *namespace* è l'astrazione che consente di ottenere un gruppo di processi che accedono a delle risorse del kernel in modo separato rispetto alle altre, mentre *cgroups* limita, calcola e isola le risorse di una macchina (CPU, memoria, I/O, rete, ecc.) per gruppi di processi. Un processo può essere avviato con dei parametri per la definizione del namespacing: piuttosto che far riferimento alla struttura di default, farà riferimento ad un differente spazio di nomi kernel, ottenendo una separazione come processi isolati a livello di sistema operativo, non serve più una separazione basata su macchina virtuale. La separazione vale per qualsiasi risorsa: PID, mount point, file system, user e così via: l'utente root o il PID 1 di un container è distinto dall'utente root o PID 1 di un altro container e anche da quello del sistema che li ospita.

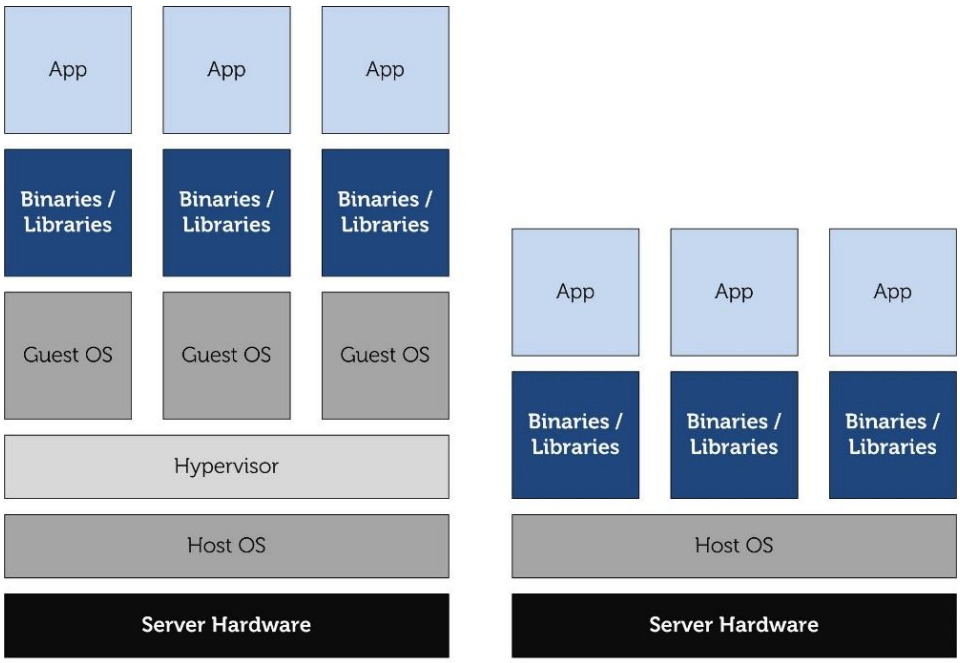


Figura 6.4: Confronto tra le due tecniche di virtualizzazione.

La figura (parte destra) mostra la virtualizzazione basata su container. Come si può notare, vi è un notevole risparmio di risorse, non è necessario dover installare un nuovo Guest OS e si opera direttamente a livello di sistema operativo Host. È sufficiente avere un kernel che possa supportare tale tecnica e Linux ad esempio lo supporta. Un container è il processo che fa riferimento a strutture namespaced del kernel, all'interno del quale possono nascere altri processi figli, poi responsabili dell'esecuzione vera e propria di un certo compito. I vantaggi principali di tale tecnica riguardano la possibilità di impacchettare letteralmente un prodotto software all'interno del container e renderlo indipendente dagli altri. Ciò comporta l'impossibilità di contaminazione tra le varie librerie software, perché risiedono esclusivamente nel container e non nell'Host OS, risolvendo così il problema circa l'impossibilità di far coesistere versioni differenti di librerie all'interno di un'unica architettura. Per quanto detto in precedenza, quest'ultima tecnica è importante per i servizi CaaS.

6.3.3 Docker

Docker è una tecnologia nata da una startup che sfrutta i container Linux per gestire l'esecuzione di più unità di lavoro indipendenti. L'obiettivo primario del progetto Docker è quello di poter letteralmente impacchettare in un unico container un certo prodotto software con le librerie e dipendenze che necessita. Docker dunque non è soltanto un modo diverso di sfruttare la virtualizzazione, è anche una piattaforma per il packaging delle applicazioni. Il container è in esecuzione in user space come un processo isolato, condividendo il kernel dell'Host OS con gli altri container. I container Linux non sono qualcosa che nasce con Docker, furono ampiamente utilizzati per molti anni da compagnie come Oracle, IBM e HP, Docker ne ha solo esteso la popolarità grazie al loro utilizzo come alternativa alla virtualizzazione classica dell'hardware. La piattaforma Docker offre un alto grado di portabilità dei container in differenti ambienti e ne consente l'esecuzione ovunque vi sia un kernel che li supporti, che sia di un Host OS su una macchina reale o a sua volta su una macchina virtuale, come accade nel caso del cloud computing, infatti Docker si sposa bene con il servizio cloud di tipo CaaS (Sez. 6.2). Il programmatore può dar vita ad un prodotto software, piazzarlo in un container e muoverlo laddove necessario per la sua esecuzione, spostando dal locale al remoto: questo è quel che ha reso Docker popolare rispetto ad altri attori come LXC, OpenVZ e Solar Jails, perché questi non offrivano alcune funzionalità che invece Docker offre.

Docker come prodotto si compone di cinque elementi: Docker Engine, Client, Image, Container e Registry. Il Registry ospita varie repository come Ubuntu, Nginx, Alpine, Redis, ecc. con le varie versioni e il Registry ufficiale è Docker Hub. Engine e Client servono rispettivamente all'esecuzione e alla gestione dei container nei vari ambienti, mentre una Image è un insieme di strati read-only del file system stratificato. L'immagine non può mutare e non mantiene uno stato, mentre ogni container è una istanza in esecuzione dell'immagine e ha uno stato con grado di permanenza variabile. Ogni container fa riferimento all'immagine da cui viene lanciato, dunque tutti fanno riferimento agli stessi strati read-only, da qui l'efficienza di Docker e il risparmio di risorse. Lo stato ed eventuali modifiche effettuate dai processi in esecuzione sono esclusive del container e registrate in uno strato R/W che permane fintantoché il container è in esecuzione. Per estendere il ciclo di vita di questo strato è necessario appoggiarsi ad un meccanismo di memorizzazione come un volume o il bind mounts: un volume è una directory o un file del file system host che è agganciato direttamente ad un container, mentre il bind mounts specifica un path assoluto o relativo dell'host. Quando si utilizza un volume, viene creata una nuova directory all'interno dello spazio di archiviazione di Docker sulla macchina host e Docker ne gestirà il contenuto. Con il bind mounts ci si basa sul file system della macchina host, che ha una struttura di directory specifica e dunque si è vincolati ad essa e al tipo di file system, per cui è consigliato l'uso dei volumi. Docker apre ancor di più la strada ai cosiddetti microservizi, pattern architetturale utilizzato il deployment applicativo del backend e del frontend web, discussi nella Sez. 6.5.

6.4 Microservizi

Il bisogno crescente di complessità nelle applicazioni Internet spinge verso l'abbandono di soluzioni monolitiche, in cui in un'unica entità sono comprese la logica di presentazione, elaborazione e memorizzazione dei dati. Si va verso un meccanismo capace di mettere insieme fonti di dati differenti, gestiti con cicli di vita differenti, adeguate ad un'alta flessibilità e con caratteristiche specifiche in base alla tecnologia coinvolta. Il vecchio approccio monolitico tradizionale è ancora una buona scelta per molte applicazioni semplici: il deployment è immediato, unico vantaggio effettivo, ma non è sostenibile nel tempo o al crescere della complessità del software, tale approccio porta con sé limiti non trascurabili, relativi alla scalabilità, alle prestazioni e al consumo di risorse. Spesso le applicazioni si trovano a dover soddisfare un picco di richieste che eccede la capacità per cui sono state progettate, e ovviamente non c'è alcun tipo di preavviso in tali situazioni, magari in un certo momento un gran numero di utenti necessita di una particolare funzionalità del servizio messa a disposizione. Un'architettura monolitica richiede un impegno a lungo termine per una tecnologia: obbliga un matrimonio con lo stack tecnologico scelto all'inizio dello sviluppo, e in alcuni casi con una particolare versione di quella tecnologia. Per tal motivo la scelta dello stack tecnologico ha un'importanza non banale nella costruzione di un prodotto software. Date le numerose limitazioni, una scelta migliore per applicazioni grandi e complesse è il modello a microservizi.

L'architettura a microservizi è un pattern che consiste nello strutturare un'applicazione come una collezione di servizi "leggermente accoppiati" (*loosely coupled*). L'architettura a microservizi consente il deployment continuo di applicazioni grandi e complesse, consentendo ad un'azienda di evolvere le tecnologie utilizzate in modo più semplice. Non si tratta più di una infrastruttura centralizzata bensì fortemente distribuita, certamente più difficile da mantenere. Il voler migliorare l'insieme di servizi comporta necessariamente una loro espansione, ma la necessità di far scalare un prodotto software implementato con un'architettura monolitica comporta la sua intera replicazione, comprese le parti che effettivamente non lo richiedono. Ciò causa un elevato spreco di risorse e un tempo di deployment molto ampio. Il punto fondamentale è poter dissezionare il

problema in parti tra di loro distinte, in modo da poter essere allocate non solo in processi differenti, ma anche in macchine differenti, appunto per evidenziare la loro separazione logica. Macchine diverse non vuol dire solo separazione fisica, ma anche strutturale: nulla impedisce intrinsecamente la possibilità di utilizzare tecnologie differenti per le varie parti, in modo da poter sfruttare le caratteristiche di ognuna. Alcuni algoritmi complessi possono essere implementati in C++, il login utente può essere fatto in Rails o JavaEE, le basi dati possono essere sia relazionali che non relazionali, purché poi ogni parte esporti un insieme di API standard per essere raggiungibile dal mondo esterno e/o dalle parti interne, magari attraverso comunicazione diretta o mediata da broker.

I benefici dei microservizi sono molteplici, subentra una forte compartimentazione tra i moduli, ognuno è relativamente piccolo e la comprensione di un problema è più semplice per lo sviluppatore, aiutando nel complesso ad aumentare la loro produttività. Si può procedere con lo sviluppo in parallelo delle parti, ogni team non ha bisogno di aspettare che l'altro completi una parte. Si può procedere senza conflitti di sviluppo, con l'ausilio di entità mock che si interfacciano allo stesso modo. Ogni servizio può essere distribuito indipendentemente da altri servizi, facilitando non solo la distribuzione di nuove versioni ma anche l'avvio di nuove repliche. La messa in campo può essere fatta in modo indipendente, ogni modulo ha il suo ciclo di vita ed è facile cambiare o riscrivere le singole parti, anche con differenti stack tecnologici soprattutto se a costi diversi. Un altro vantaggio importantissimo che si ottiene è legato ad una maggiore facilità di manutenzione, ogni modifica su una parte non necessita del coinvolgimento delle altre. Ovviamente, se nel modello monolitico un danno accadeva su una unica macchina ora può potenzialmente accadere su N macchine distinte. Tale scenario però si presta bene a meccanismi di *load balancing* tra le entità, aggiungendo un grado di resilienza complessiva che potenzialmente può essere maggiore della precedente architettura. Si ottiene un vantaggio in termini di isolamento nei guasti: ad esempio, un *memory leak* in un servizio non influenzerà gli altri, salvo il fatto che il suo malfunzionamento applicativo può inibirli, mentre in un'architettura monolitica comporta il non funzionamento dell'intero sistema. Infine, forse cosa più importante insieme alla scalabilità, elimina tutti gli impegni a lungo termine per aver abbracciato uno stack tecnologico. È chiaro che il ruolo del sistemista assume una rilevanza non banale, oltre che un sacco di nuove responsabilità, basti pensare che in casa Google nasce il ruolo di "Site Reliability Engineer".

È proprio in questo scenario che si colloca Docker come tecnologia a supporto dei microservizi, in cui in ogni container viene eseguita una istanza di una parte software con una singola responsabilità (backend, data storage, broker, background workers, search engine, cache, ecc.). Un contenitore Docker fornisce un pacchetto immutabile, portatile e stateless per i microservizi e le immagini possono essere spostate, condivise e utilizzate come base per la creazione di nuovi container. Diventa facile replicare le singole istanze, basta lanciare un nuovo container e configurarlo opportunamente; in casi più complessi ci si può appoggiare a vari strumenti di gestione come Docker Swarm, Docker Compose, Kubernetes e così via.

Il modello a microservizi porta con se anche dei limiti operativi. Il debug e l'*integration testing* sono più difficili, perché tutto è disaccoppiato e non è facile capire quale parte delle tante non funziona come dovrebbe. Gli sviluppatori devono implementare un meccanismo di comunicazione tra le parti, basato su protocolli come REST o proprietari. Il sistema nella sua interezza è capace di funzionare solo se tutte le sue parti sono *"up & running"*. Se una istanza venisse a mancare, non è così scontato eseguirne un ripristino, perché è necessario capire la semantica dei vecchi backup in relazione al momento in cui si vuole lanciare la nuova stanza. Non è detto che siano backup ancora validi, possono essere logicamente "scaduti" rispetto all'evoluzione del sistema dopo il fault. Le singole parti tendono a morire con tempistiche differenti e la consistenza dei dati diventa qualcosa da non trascurare, perché ogni servizio può richiedere un database separato e tra questi possono esistere varie informazioni che devono essere sincronizzate (pur non essendo replicate).

La logica dei microservizi è una estremizzazione nel modello distribuito, è necessario valutare il trade-off tra vantaggi e svantaggi nell'abbracciare un tale modello. Per un sistema è fondamentale la possibilità di evolvere, si parte sempre da un progetto legato alla raccolta dati e una qualche forma di elaborazione, offerta agli utenti come funzionalità applicativa, che nel tempo andrà a mutare, perché il mondo ha bisogno di altro e bisogna adattarsi alle esigenze se si ha voglia di sopravvivere, come è lecito pensar di fare. Se è stato fatto male sin dal principio, è alta la probabilità di dover riprogettare l'intero modulo, se non tutto il sistema, nei casi peggiori. Un approccio a microservizi aiuta ad evitare una situazione di completa ristrutturazione dell'architettura applicativa. I requisiti di scalabilità e il tipo di applicazione certamente fanno del presente lavoro un buon candidato per essere basato sui microservizi.

6.5 Strategia di deployment

Il deployment per l'applicazione creata riguarda principalmente il backend e il frontend web, per il mobile è sufficiente rilasciare l'applicazione firmata nei vari store. La strategia di deployment utilizzata deve favorire la scalabilità e le prestazioni con il minimo consumo di risorse. La strategia di deployment è indipendente dal tipo di infrastruttura utilizzata, può trattarsi di una macchina hardware fisicamente isolata (server dedicato) o di una macchina virtualizzata, come nel caso del cloud computing. A causa delle limitate risorse disponibili, già discusse nelle precedenti, sezioni ci si appoggia però ad una infrastruttura di cloud computing esterna di tipo IaaS. Per quanto discusso, la scelta ricade sull'utilizzo dei container gestiti con Docker, sfruttando l'approccio dei microservizi. L'esempio di deployment più semplice può essere quello riportato nella figura seguente.

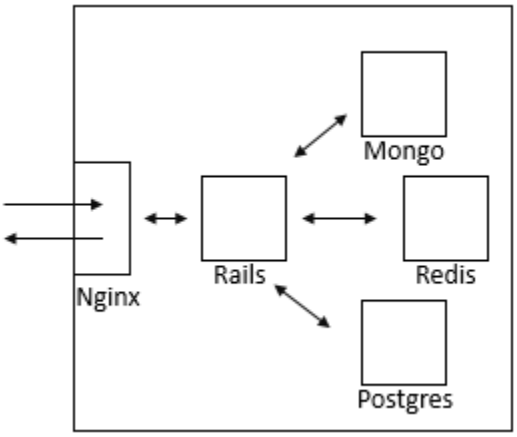


Figura 6.5: Deployment minimale del backend.

La figura 6.5 mostra un esempio minimale di deployment, in cui sono presenti almeno quattro container: Rails, MongoDB, Postgres e Redis. Nginx fa da HTTP server e reverse proxy[4], può essere installato direttamente sulla macchina fisica, essendo utile una sola istanza, ma nulla vieta un suo utilizzo attraverso container. Questo esempio di deployment è il più semplice possibile, il backend applicativo (Rails) comunica con una istanza dei database per ricavare le informazioni necessarie. Il crescente bisogno di prestazioni però non può essere soddisfatto da una simile strategia, possono esservi varie soluzioni in favore della scalabilità. Ogni container ha un ruolo ben preciso, per cui è sufficiente replicare l'unità di lavoro dove necessario.

[4] Anche come server mail o TCP/UPD generico.

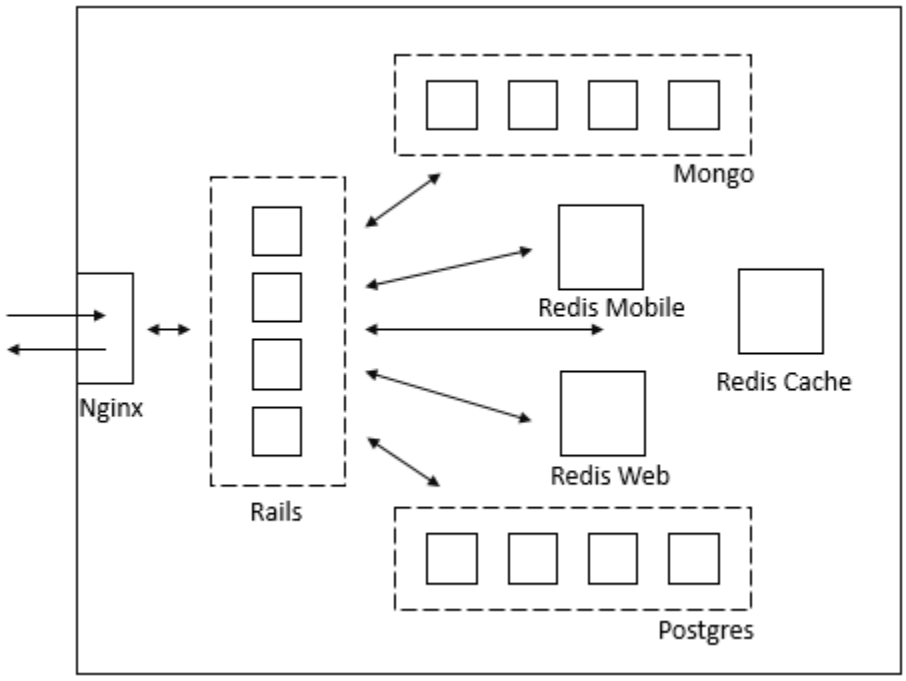


Figura 6.6: Deployment avanzato del backend.

La figura 6.6 riporta un esempio di deployment avanzato per un backend che possa sostenere un alto carico di lavoro. La replicazione delle istanze Rails deve essere accompagnata dalla corretta configurazione del reverse proxy per la funzionalità di load balancing, mentre i database Postgres e Mongo necessitano rispettivamente di clustering e sharding, questi possono essere a loro volta fatti con altre repliche di container. Essendo Redis utilizzato per compiti abbastanza leggeri, può essere creata una istanza con un compito ben preciso: ad esempio, una per la gestione del caching, una per la gestione circa lo stato dell'autenticazione mobile e una per il web, e così via. Tutte le istanze di Rails avranno come configurazione per il caching l'unica istanza di Redis, così come per l'autenticazione mobile o web, gestione immagini e video, ecc. Se il carico di lavoro per le singole istanze Redis diventa insostenibile, può essere clusterizzato, mantenendo però separati i vari cluster per compiti. La gestione dei container avviene attraverso i comandi e la CLI del Docker Client, grazie ai quali è possibile replicare e configurare le parti dove necessario. Si possono già notare i vantaggi dell'approccio a microservizi: un alto carico di lavoro può essere soddisfatto semplicemente replicando le istanze di Rails, senza dover necessariamente replicare i database, come invece accade nell'approccio monolitico. Inoltre, si riducono tantissimo i tempi per il deployment, basta lanciare un nuovo container e configurare il reverse proxy.

Più semplice è il caso del deployment per l'applicazione web. Il deployment minimale consiste di un unico container raggiungibile con apposita configurazione Nginx. Grazie al meccanismo utilizzato per la costruzione dell'applicazione web, tale container dovrà fornire per qualsiasi URL il file index.html e il complesso di file Javascript associati. Il carico di lavoro server-side per il frontend web è praticamente nullo, ma se il numero di richieste diventasse insostenibile, è possibile replicare le istanze dell'applicazione Angular, usando come load balancer o lo stesso nginx utilizzato per il backend, oppure utilizzare nginx in due container separati, in ascolto su porte distinte, uno per servire il backend e uno il frontend. La figura mostra sia l'esempio minimale che avanzato per il deployment del frontend web con i container, senza la replicazione di nginx.

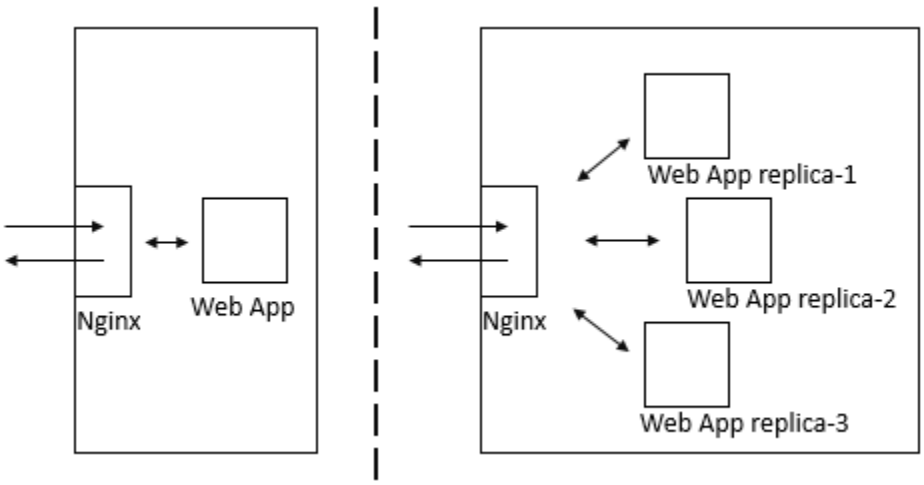


Figura 6.7: Deployment web minimale (parte sinistra) e avanzato (parte destra).

Anche in tal caso la scalabilità è data dalla replicazione dei soli container per la web app, senza coinvolgere in alcun modo i container per il backend. Ovviamente i due scenari fanno parte di un'unica macchina fisica o virtualizzata, ma si può aggiungere un grado di isolamento tra i container a livello di rete, aumentando il grado di sicurezza complessivo del backend, separandoli attraverso le User Defined Networks: non è necessario che tutti debbano parlare con tutti, si può pensare ad una configurazione di reti isolate per le coppie Rails-Redis, Rails-Mongo, Rails-Postgres e una per i gruppi di container deputati a servire il frontend.

```
http {
    ssl_session_cache       shared:SSL:10m;
    ssl_session_timeout         10m;

    upstream api_backend {
        ip_hash; // sessions strategy
        server              172.1.1.2:3000;
        server              172.1.1.3:3000;
        server              172.1.1.4:3000;
    }

    server {
        listen              443 ssl;
        server_name         www.domain.com;
        ssl_certificate     path_to_file/www.example.com.crt;
        ssl_certificate_key path_to_file/www.example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers         HIGH:!aNULL:!MD5;

        location / {
            //Path to the folder to serve static content
            try_files $uri / index.html;
        }
    }

    server {
        listen              8080 ssl;
        server_name         api.domain.com;
        ssl_certificate     path_to_file/api.domain.com.crt;
        ssl_certificate_key path_to_file/api.domain.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers         HIGH:!aNULL:!MD5;

        location / {
            // Set PROXY Headers
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass http: api_backend;
        }
    }
}
```

Figura 6.8: Configurazione di nginx per il backend e il web server.

La figura 6.8 riporta un esempio di configurazione per nginx come reverse proxy e load balancer per i due casi appena discussi. Sono necessari due blocchi server di configurazione, uno per il backend e uno per il frontend, in cui specificare il dominio e la porta per cui tale blocco viene invocato. Il frontend è esposto all'indirizzo classico (www.domain.com) e alla porta di default (443) con TLS, mentre le API al sottodominio "api" (api.domain.com) sulla porta 8080, sempre con TLS, facendo uso della funzionalità di load balancing offerta da nginx. Il frontend deve servire sempre lo stesso file index.html, radice dell'applicazione Javascript, a prescindere da quale file venga richiesto al server web: ciò è necessario perché la navigazione delle pagine avviene con il routing client-side fornito da Angular (Sez. 5.1.2) altrimenti, nel contattare un path diverso da quello di base "/", si otterrebbe una risposta HTTP con codice di stato 404. Le immagini invece saranno servite dal backend Rails, dunque reperibili attraverso un dominio diverso da quello del frontend. Ogni richiesta ad API (api.domain.com) viene passata al blocco di configurazione (api_backend - nome a piacere) in cui sono elencati i server che effettivamente eseguiranno la richiesta, dichiarati nel formato IP:PORT, e la strategia di bilanciamento del carico. Nginx offre tre strategie: least-connected, ip-hash e round-robin. Le modalità round robin e least connected sicuramente offrono una distribuzione migliore del carico, ma per mantenere eventuali sessioni logiche è meglio usare la ip-hash, che ne tiene traccia per poter demandare alla stessa istanza l'esecuzione di una richiesta riconducibile alla stessa sessione. Nel caso del frontend web replicato, è sufficiente ripetere una configurazione simile a quella del backend: all'interno del blocco "location" si devono specificare le istanze dei frontend, magari con strategia di bilanciamento round robin dato che non vi sarà alcuna sessione.

Questa modalità di deployment consente all'architettura di backend di essere molto flessibile, la presenza di più mail server, search engines server per i websocket non avranno alcuna limitazione da parte dell'architettura già presente: è sufficiente lanciare delle istanze, magari separate tra loro con altre User Defined Networks, da configurare opportunamente con i vari load balancer o con le istanze esistenti. Ad esempio, i websocket potrebbero avere un unico endpoint comune, come nel caso di Action Cable in modalità stand alone, poi smistato tra le varie istanze di Rails; una istanza di nginx come mail proxy può essere anteposta ad una serie di container che fanno da mail server per l'invio di email dell'applicazione; meccanismi di ricerca avanzati e potenti, come elasticsearch, possono essere inseriti a supporto della business logic con modifiche minime e senza dover necessariamente interrompere l'esecuzione dell'intero servizio applicativo.

La configurazione appena discussa, che ha permesso il deployment di un backend altamente scalabile e facilmente manutenibile, chiude l'ultimo Capitolo e il lavoro di tesi svolto.

7. Conclusioni

Il presente documento ha trattato il lavoro di tesi svolto presso il Politecnico di Torino, che riguarda la realizzazione di un social network, idea di business di una startup sportiva che attraverso questo vuole facilitare l'accesso alla pratica sportiva agli utenti e promuovere le categorie professionali come istruttori e nutrizionisti o entità sportive come centri, organizzazioni, società e così via, attraverso un sistema di recensioni e ricerca geolocalizzata di risorse. La realizzazione comprende lo sviluppo completo del prodotto software, dalla progettazione all'implementazione ed infine il suo deployment. Il prodotto software consiste di varie parti: un'applicazione web, due applicazioni per smartphone (Android e iOS) e l'intera architettura server di backend. La startup si trova attualmente nella fase di *discovery*, con limitate risorse finanziarie a disposizione, traducendosi inequivocabilmente in un team ristretto, oltre che nella scelta di servizi informatici economici, come compromesso costo-funzionalità. Ciò ha vincolato in parte la realizzazione del prodotto nelle tecnologie scelte, ma di certo non può togliere importanza allo stack utilizzato. La trattazione riguarda soltanto la messa a punto di un MVP (Minimum Viable Product), la volontà però è stata quella di creare un buon prodotto sin dal principio, infatti i problemi principali alla base di queste nuove forme di imprenditoria riguardano proprio il non riuscire a realizzare un buon prodotto da inserire sul mercato in modo da acquisire visibilità sin da subito. Eppure le scelte fatte sono state frutto di una progettazione accurata, che vuole soddisfare la volontà della startup per creare un prodotto degno di un mercato ancora senza competitors. È lecito chiedersi allora quale impatto possano avere tali scelte sul futuro del prodotto e ovviamente sul futuro della startup stessa, capire quali sono i vantaggi e gli svantaggi dello stack tecnologico scelto, se vi possono essere problemi circa la sua sostenibilità nel tempo, se ammette margine di evoluzione o meno e soprattutto se è possibile far evolvere l'applicazione senza stravolgerne l'architettura.

Si è abbandonato il modello monolitico di sviluppo software in favore del modello a microservizi, grazie alla separazione delle responsabilità delle parti in gioco è stato possibile aumentare il grado di scalabilità e di resilienza complessivo, oltre che ridurre i tempi di deployment a lungo termine. Abbandonare l'approccio monolitico in favore del modello a microservizi non è stata l'unica strategia utile per una buona architettura software, vi sono state altre due scelte rivoluzionarie in ambito di sviluppo applicazioni Internet: sfruttare la capacità computazionale dei client per la generazione delle viste e l'utilizzo di Xamarin per lo sviluppo mobile cross-platform, che ha consentito un riutilizzo del codice fino al 90% e un risparmio di tempi e risorse molto grande. Sono proprio queste scelte a rendere lo stack tecnologico utilizzato come uno dei migliori possibili: Ruby on Rails per il backend, Angular per il frontend web e Xamarin per la realizzazione delle applicazioni Android e iOS.

Rails è un framework per la costruzione full stack di applicazioni web con una curva di apprendimento praticamente piatta. Si tratta di un framework MVC basato su Ruby, un linguaggio interpretato che consente di utilizzare contemporaneamente diversi paradigmi di programmazione, che Java e C++ ad esempio non consentono, seppur le prestazioni non sono paragonabili. Rails è molto potente grazie alle varie gemme curate dalla vivace community, la maggior parte open-source, tra cui spicca la ricca presenza di ORM e librerie varie per definire in modo molto semplice modelli di dati e relazioni associate. Però Rails con le View generate server-side non scala affatto bene, formate da un misto HTML e Ruby, come le pagine con script PHP o JSP, che al crescere della complessità applicativa rischiano di sovraccaricare il backend in computazioni non sempre necessarie.

Per questo motivo la scelta è stata quella di separare la generazione delle viste lato server da modelli e controller, per spostarle interamente sui client web e mobile, in modo da sfruttare solo le parti ottimali del framework stesso. La gestione della navigazione applicativa tramite routing sul client consente di utilizzare il backend solo per recuperare le informazioni da mostrare nelle viste opportune, grazie al sistema di API costruite con la filosofia REST. La comunicazione tra le parti è più efficiente rispetto all'architettura tradizionale delle applicazioni web: per ogni URL che il client contattava, corrispondeva un ciclo richiesta-elaborazione-risposta con una renderizzazione totale della vista sul client, da qui il nome di "navigazione con interruzioni"; con il backend deputato a servire informazioni piuttosto che contenuti HTML completi, le viste sono costruite sui client, lo scambio dati è ridotto in payload e dunque si ha una navigazione sensibilmente più veloce e più fluida. Le parti comunicano basandosi sulla filosofia REST e usando il formato dati JSON, molto leggero e più flessibile rispetto all'XML, che spesso necessita di librerie aggiuntive molto onerose.

Tutto ciò rende il backend estremamente performante e scalabile, ma la sua scalabilità è anche favorita da altre scelte, come quella del deployment basato su container: grazie ad nginx configurato come reverse proxy, è stato possibile ottenere un grado di scalabilità molto alto, nel senso che si possono replicare con l'ausilio di Docker un numero di istanze "illimitato", sia per i server applicativi che per la clusterizzazione dei database e così via. Il vero limite alla replicazione di istanze è dato dalla capacità della macchina hardware o del servizio di cloud computing che li ospita. In questo scenario Ruby on Rails diventa dunque uno dei framework di maggior interesse per la costruzione del backend, accanto ad altri come ExpressJS o Phoenix Elixir.

Oggigiorno la scelta per il framework da utilizzare per la costruzione del frontend web sembra essere quasi scontata: Angular o React? Di certo non sono gli unici, ma sono i più discussi. E di certo non si tratta "solo" di abbracciare la filosofia di Google o Facebook, rispettivi ideatori, bensì di strutturare in modo completamente diverso un prodotto software. React infatti non è un framework, bensì una libreria per la costruzione di componenti web riutilizzabili altamente performante, grazie all'utilizzo di un Virtual DOM per la manipolazione delle viste. Nonostante sia più performante rispetto ad Angular, soprattutto in caso di enorme mole di dati da mostrare nelle viste, gli sviluppatori che lo usano attualmente non sono tanti ma neppure pochissimi, e i motivi sono legati appunto alla tipologia di compiti che questa libreria è in grado di risolvere. Non trattandosi di un framework completo che possa aiutare in toto a realizzare un frontend, necessita di ulteriori componenti che nel complesso non creano dipendenze reciproche e che possono avere vita indipendente, come Flux e Redux, che non hanno incuriosito più di tanto gli amanti di Angular come alternativa alla costruzione di frontend. Lo svantaggio più grande di React è proprio questa mancanza di completezza per lo sviluppo web, che causa una curva di apprendimento ben più ripida: è necessario avere padronanza con altri componenti software e progettare un'architettura per la gestione dei dati, dello stato locale e della business logic. Un po' come reinventare qualcosa di simile ad un modello MVC, MVVM o altro ancora, che certamente richiede tempi di sviluppo maggiori. Per tal motivo la scelta ricade spesso su Angular, grazie alla sua completezza come framework MVC (in Angular JS – 1.x) e più in generale come framework basato su component, che rende la curva di apprendimento meno ripida. A dirla tutta, anche React è basato su component, come a testimoniare che secondo Google la filosofia di casa Facebook non era così trascendentale. Ciò non vuol dire che Angular è un framework "banale": Angular fa numeri non solo perché un offre strumenti e modelli completi di sviluppo web, ma perché nel tempo è diventato robusto (soprattutto con l'aggiunta del supporto per Typescript), potente ed efficiente, molto usato per la costruzione di portali web tra cui Netflix, Paypal, tutti i frontend di casa Google e quello del presente lavoro. La potenza di Angular e i tempi di sviluppo brevi (in riferimento ad altri framework) sono stati i due fattori che hanno inciso principalmente sulla sua scelta.

Per il tipo di applicazione trattata e per le potenzialità del suddetto framework, ben supportato da una vivace community open source, non vi sono motivi che al momento possano far pensare ad un suo abbandono in favore di altre strategie. Ma se Rails e Angular sono buoni candidati "a vita" per l'applicazione realizzata, forse non si può dire lo stesso per Xamarin, che di certo ha contribuito a ridurre notevolmente i tempi di sviluppo delle applicazioni mobile e che in futuro potrebbe essere sostituito del tutto in favore di altre strategie. Xamarin è davvero uno strumento potente per la creazione di applicazioni sia mobile che non, grazie allo sviluppo cross-platform che è in grado di offrire. È stato ampiamente dettagliato nel Capitolo 4, principalmente risolve tanti dei problemi che esistono nella realizzazione di un'applicazione per varie piattaforme. Dover riscrivere le stesse funzionalità per una piattaforma differente, in un altro linguaggio di programmazione, basato a sua volta su interfacce di programmazione (API) differenti, comporta conseguenze da non sottovalutare nell'abbracciare lo sviluppo nativo, sia dal punto di vista personale che aziendale: vuol dire la necessità di dover acquisire le competenze opportune per operare sulla piattaforma in questione, che per un'azienda significa dover assumere personale appositamente qualificato. E per una startup come quella coinvolta nello sviluppo trattato non è certamente il caso ideale. Ciò non vuol dire che le aziende grandi non faranno uso di Xamarin o altri framework di sviluppo come Ionic, Apache Cordova, ecc. La necessità di risparmiare su più fronti non è l'unico vincolo, quello caratterizzante per la scelta dello sviluppo cross-platform è legato a quel che l'applicazione deve offrire. Xamarin consente il la scrittura di codice riutilizzabile, è una buona soluzione "code-once-deploy-all", vuol dire che potenzialmente si può scrivere una sola volta del codice C# che sarà mappato su API native di ogni piattaforma per ottenere le rispettive applicazioni. In questo senso si ha un notevole risparmio di risorse: considerando lo sviluppo nativo e supponendo i tempi di sviluppo uguali, ogni team specifico è adibito allo sviluppo per la singola piattaforma e ognuno può produrre una sola applicazione, dunque servono N team per realizzare la stessa applicazione per le N piattaforme (Android, Windows, iOS, macOS, …); di contro, con Xamarin si può ridurre il personale totale di N volte e l'unico team può produrre N applicazioni con le stesse tempistiche, generandone poi una per piattaforma. Oppure a parità di personale si ottiene un risultato finale fino a N*N volte maggiore, se si ha necessità di lavorare su N applicazioni diverse.

Xamarin nel tempo si è evoluto e pian piano aumenta la percentuale di codice condivisibile tra le piattaforme, arrivando anche al 90% del totale. Al giorno d'oggi però vi sono certe funzionalità che Xamarin non è ancora in grado di fornire, basti pensare a quelle applicazioni con UI molto complesse o che necessitano di computazioni che si appoggiano a funzionalità specifiche della piattaforma. Il limite intrinseco di Xamarin è che si tratta di un wrapper di funzionalità native, non è detto che tutte le piattaforme mobile o desktop offrano API per le stesse funzionalità, per cui Xamarin non può inventare ciò che non è offerto dal nativo. In altre parole, l'unico motivo che può portare all'abbandono completo di Xamarin in favore dello sviluppo nativo è la necessità di utilizzare caratteristiche uniche della piattaforma o realizzare funzionalità che con Xamarin sono scomode o impossibili da ottenere. È praticamente impossibile continuare lo sviluppo sull'applicazione finale generata a partire dalla soluzione condivisa, che a prescindere dalla piattaforma target ha un certo overhead aggiuntivo sia per dimensioni (codice assembly linkato staticamente) che come tempi di bootstrap applicativo, a causa dell'architettura Mono o .NET che poi eseguirà effettivamente l'applicazione sul device. Con Xamarin si è vincolati ad un minimo comune denominatore per il design di UI ed infine, elemento forse un po' trascurato, è comunque necessario conoscere il framework stesso, padroneggiarne le funzionalità e i pattern offerti.

In fin dei conti Xamarin è una tecnologia che nel tempo sta compiendo passi da gigante, di certo è impensabile che possa eliminare tutti i vincoli discussi pocanzi, al più può ridurli, però è anche vero che si presta bene alla costruzione di applicazioni cross-platform per tanti scenari non troppo complessi. La scelta di utilizzare Xamarin per contribuire alla realizzazione del prodotto non

riguarda solo i limitati tempi di sviluppo (secondo la volontà della startup), in modo da potersi affacciare in fretta sul mercato, ma perché si tratta di un'applicazione dalle funzionalità e UI che possono essere realizzate con quanto Xamarin offre. Il futuro non è prevedibile, ma stando a quanto attualmente pensato dalla startup, non vi sono motivi urgenti a breve termine per cui si debba abbandonare tale strategia in favore dello sviluppo nativo, dunque Xamarin può continuare ad essere principale attore per il mobile nello stack tecnologico.

È degna di discussione anche la scelta circa l'infrastruttura per il deployment. Grazie al cloud il mondo dei servizi è cambiato, siamo ormai nell'era del "…as a Service", ma per scegliere bisogna capire bene la chiave dell'offerta e i vantaggi che ne derivano. Come spiegato nella seconda Sezione del Capitolo 6, vi sono vari modelli di servizi offerti con il cloud (IaaS, PaaS, SaaS, …). Utilizzare dei servizi cloud a pagamento è stata una scelta praticamente forzata, perché la startup non ha risorse a sufficienza per poter costruire un proprio datacenter. Il vantaggio è ben evidente, un'azienda che gestisce il proprio datacenter dovrebbe possedere risorse hardware a sufficienza per il massimo picco di utilizzo (facendo *provisioning*) e che spesso non viene sfruttato, risultando come utilizzo scarso di risorse praticamente sprecate. La soluzione basata su cloud computing aumenta la flessibilità con l'aggiunta o l'espansione delle risorse richieste in modalità on-demand, dunque si ottengono un sacco di vantaggi soprattutto a lungo termine. Anche grandi società migrano parte o tutti i loro servizi dai propri datacenter verso infrastrutture cloud create e mantenute da altri: i motivi non riguardano soltanto il costo di mantenimento dell'infrastruttura privata o del personale, che sicuramente è una spesa aggiuntiva, ma il fatto che ormai i servizi offerti sono così avanzati rispetto a quelli che ognuno dovrebbe costruirsi da preferire una soluzione già esistente, funzionante e affermata sul mercato. Di certo i costi di costruzione e mantenimento sono alti e chi opta per il proprio datacenter evidentemente ha i suoi buoni motivi per farlo: è il caso di Google, Facebook, IBM, Microsoft e tanti altri grandi nomi, ma non la startup in questione. Nel caso in esame, la scelta non pregiudicherà una eventuale evoluzione dell'architettura software né tantomeno sarà pregiudicata la possibilità di scalare, grazie al deployment basato su container. E neppure passare ad un datacenter di proprietà potrà essere un problema: a parte la configurazione hardware dell'infrastruttura per la computazione, il networking, la sicurezza, ecc., rispetto al vecchio cloud sarà necessario soltanto trasferire i contenuti nella nuova sede e mandare in esecuzione il backend. L'applicazione certamente evolverà nel tempo, sia nei servizi offerti all'utente finale, sia come miglioramento del prodotto software. Evoluzioni a breve termine dal punto di vista applicativo potrebbero essere ad esempio l'inclusione dei social network più popolari come meccanismo per l'accesso (Facebook, Google+ e altri), un sistema di messaggistica tra gli utenti, oltre ovviamente al perfezionamento delle idee già messe in campo. Per il prodotto software in sé, sarà migliorato il meccanismo di ricerca, in modo da rendere più performante e più naturale possibile l'interazione dell'utente con i servizi. Certamente, che si tratti di modificare parte del prodotto software o costruire un datacenter privato, saranno necessarie risorse economiche ed umane aggiuntive, per cui non si prospettano cambiamenti radicali in tempi brevi. In altre parole, con buona probabilità il prodotto creato resterà tale fin quando la società in questione non avrà raggiunto una evoluzione da permettergli le dovute modifiche.

Concludendo, lo stack tecnologico scelto per l'architettura software trattata consentirà al prodotto di essere facilmente mantenuto nel tempo, è stata progettata e realizzata una soluzione che potrà fronteggiare senza grosse difficoltà un numero di utenti molto ampio, evolvere nel tempo senza causare sprechi di alcun tipo nelle risorse utilizzate. La startup può così presentarsi in un mercato senza competitors e proseguire il cammino di crescita, avendo in mano un buon prodotto da poter lanciare come importante elemento di innovazione, magari conquistandosi l'interesse degli investitori necessari che ne consentiranno il salto di qualità e la stabilizzazione sul mercato.

Riferimenti bibliografici

[1] Yukihiro Matsumoto - *The Ruby Programming Language*

[2] Michael Hartl - *Ruby on Rails Tutorial (Rails 5)*

[3] Sam Ruby - *Agile Web Development with Rails 5*

[4] Official Ruby on Rails Documentation - *https://rubyonrails.org/*

[5] Charles Petzold - *Creating Mobile Apps with Xamarin.Forms Book (1st Edition)*

[6] Steven F. Daniel - *Mastering Xamarin UI Development*

[7] Official Xamarin Documentation - *https://www.xamarin.com/*

[8] Matt Frisbie - *Angular 2 Cookbook*

[9] Nathan Murray - *ng-book: The Complete Guide to Angular 4*

[10] Official Angular Documentation - *https://angular.io/*

[11] Remo H. Jansen - *Learning TypeScript*

[12] James Turnbull - *The Docker Book: Containerization Is the New Virtualization*

[13] Sébastien Goasguen - *Docker Cookbook: Solutions and Examples for Building Distributed Applications*

[14] Official Docker Documentation - *https://www.docker.com/*

Ringraziamenti

Credo che scrivere i ringraziamenti sia sempre stata la cosa più difficile da fare, tante parole non rendono sempre interessanti i discorsi, per cui cercherò di essere breve e diretto.

Innanzitutto vorrei vivamente ringraziare il mio relatore, Prof. Giovanni Malnati, per avermi guidato in questo arduo compito di creazione del prodotto e nella stesura della tesi stessa, di mostrarmi aspetti relativi allo stesso che senza di lui probabilmente avrei ignorato, accompagnandomi passo dopo passo nella risoluzione critica e completa di qualsiasi problema. Ho trovato in lui un saldo punto di riferimento per ogni dubbio, consiglio strategico o approfondimento di qualsiasi tipo. Non dimenticherò mai i suoi insegnamenti, né i profondi discorsi umani di fine corso, un esempio che è stato e che sarà, per cui mi sento in dovere di ringraziare.

Similmente ed indistintamente, vorrei ringraziare tutti gli altri professori e professoresse da cui ho avuto il piacere di ricevere insegnamenti in questo percorso accademico presso il Politecnico di Torino, per le grandi conoscenze donate, la pazienza e l'entusiasmo trasmesso, per avermi fatto capire come si possa amare e far amare lo studio e la conoscenza di argomenti non sempre facili, per la formazione e la precisione che mi hanno formato in questo percorso che tanto desideravo intraprendere. Senza il loro sapere non avrei mai acquisito la forma mentis che caratterizza il mio essere e le competenze oggi in mio possesso.

E ovviamente Mamma e Papà, grazie. Le mie parole non saranno mai abbastanza per poter esprimere la gratitudine che meritate. Perché ne meritate tanta. Come tanto è stato il sostegno a distanza in questi due anni, lontano da casa, lontano dai vostri abbracci, dai vostri sorrisi, dal vostro essere tutto per me. Il nostro legame ha avuto momenti di alti e bassi, ma mi avete sempre dato la forza di andare avanti anche a migliaia di chilometri di distanza da casa, da quell'ambiente in cui sono cresciuto e di cui avrò sempre nostalgia. Sono andato via dalla mia terra per realizzarmi, è vero, ma a malincuore. E ancora non capisco la maggior parte di coloro che non vedono l'ora di abbandonare le mura di casa per intraprendere una vita senza mantenere il legame con i propri genitori, coloro che ci hanno dato la vita. Ogni telefonata o videochiamata mi ha sempre dato la forza e il coraggio per andare avanti e a rialzarmi ad ogni caduta. L'amore che mi avete sempre riservato è stato il motivo principale che oggi mi porta a dedicarvi queste poche righe per ringraziarvi e dimostrarvi quanto vi amo. Ci sono cose che non si possono esprimere a parole, in quella casa non è sempre andato tutto per il verso giusto, ma gli abbracci in aeroporto, quelli veri, hanno sempre colmato ogni vuoto.

Antonio, Agata, Chiara, forse sono più le litigate che i bei momenti trascorsi, ma grazie anche a voi per il sostegno dato e per aver creduto in me, sempre. E scusate se a volte non sono stato all'altezza, ma ci sto ancora lavorando, spero di farcela entro questa vita, che spesso non basta mai per nulla.

Sapete bene che amo la musica, da quando avevo 14 anni ogni primo pomeriggio del fine settimana sparivo da casa per tornare il giorno dopo. Classico: dove vai? "A lavoro, suono…". La più grande passione che mi ha accompagnato in tutta l'adolescenza e che certamente ha contribuito alla mia personalità, ma che purtroppo a Torino ho dovuto abbandonare quasi del tutto. Ciò inizialmente mi ha demoralizzato e non di poco, le lacrime che solcavano il mio viso non sono state così lontane dalla storia raccontata da un grande artista della musica. Sono nato in un paese che non offre grandi cose, per cui mi tocca riprendere il mio essere andato via di casa. Cambiare è difficile, non cambiare è fatale.

"*Vai finalmente a stare in città. Là troverai le cose che non hai avuto qui…*".

Si, è vero, ma non ci trovi tutto in città. Laddove si respira cemento non ci trovi l'affetto e l'accoglienza che casa tua può dare. Perché in quella terra ci sono nato, e in quelle strade ci lascio il mio cuore. Ma mia cara Sicilia, ne hai lasciato scappare un altro, a far così presto ti ritroverai sola. Tornerò, ma chissà cosa troverò…

Eppure ad ogni mio ritorno ho sempre trovato un gruppo di persone pronte a farmi festa, sia i parenti che le stesse persone con cui ho avuto il piacere di lavorare durante la mia adolescenza, non abbiamo mai perso il legame che si è instaurato nel tempo. Grazie anche a voi per l'affetto e il sostegno dato.

Un ringraziamento particolare va a Maria Teresa e Valentina, che hanno affidato a me l'arduo compito di dare vita a questo prodotto per rivoluzionare il mondo dello sport. Il mio augurio è che possa al più presto diventare una realtà indispensabile per gli utenti e che questa startup possa realizzarsi al meglio. Stephen Hawking disse: "*Per quanto la vita possa sembrare difficile, c'è sempre qualcosa che si può fare per avere successo*.", ed è proprio il nostro obbiettivo.

Ringrazio anche tutti coloro che mi hanno sostenuto in questo percorso, dai nuovi amici conosciuti ai nuovi colleghi, compagni d'avventure e di cori da stadio, in una città fantastica che di certo mi mancherà e che merita un posto nel mio cuore.

Fin qui non è stato nulla facile, affatto. Però mi sento anch'io orgoglioso vincitore, spesso vedo gli altri come predestinati a qualcosa, mentre io ho dovuto sudare per ottenerlo. Finalmente ho completato il percorso di alta formazione ingegneristica che tanto desideravo, sono felice di aver raggiunto un altro obbiettivo che considero importantissimo nella mia vita, consapevole che si tratta di un punto di partenza e non di arrivo. Adesso mi tocca entrare nel mondo del lavoro e spero di poter sfruttare a pieno le conoscenze acquisite. Ma si sa, chi fa della passione una professione, passerà tutta la vita senza lavorare. La vita spesso ci riserva situazioni in cui bisogna tenere duro ed impegnarsi fino in fondo per uscirne vivi e l'aver lottato senza sosta mi rende orgoglioso.

Gli aquiloni si alzano con il vento contrario, mai a favore.

Marco.