POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Progetto e implementazione di Stardust: un editor online per la configurazione di un sistema di navigazione indoor

Relatore
Prof. Giovanni Malnati

Candidati
Federica Ferro
Seyedshoya Moosavikhorshidi

Ottobre 2017

Indice

Sommario

Nel presente elaborato verranno argomentate le scelte progettuali, lo studio delle tecnologie e l'implementazione per la realizzazione di Stardust, un editor online per lo sviluppo di un sistema di navigazione indoor.

Lo studio verrà presentato seguendo l'evoluzione naturale delle varie fasi realizzative del lavoro.

Presenteremo prima di tutto lo stadio di preparazione al progetto nel quale verranno identificati i bisogni e formulate le soluzioni.

In particolare, nel *primo capitolo*, partendo dal concetto di navigazione indoor e sulla base delle tecnologie ad oggi esistenti per la sua progettazione, parleremo delle innovazioni introdotte dalla piattaforma Stardust per la soddisfazione dei bisogni degli utenti.

Il *secondo capitolo* avrà invece lo scopo di esporre le principali funzionalità di Stardust, sviluppate ed affinate partendo dall'analisi dei requisiti e del benchmarking, e costruite sulla base di specifici principi fondamentali per la progettazione.

Nel corso del *terzo capitolo*, a partire dai requisiti tecnologici, arriveremo alla costruzione di uno schema di massima dell'architettura del sistema.

Il *quarto*, *quinto* e *sesto capitolo* costituiscono la seconda parte del presente lavoro di tesi, nella quale verranno approfonditi i dettagli teorici rispettivamente delle tecnologie utilizzate per la gestione dei dati, il back end ed il front end della piattaforma.

La terza parte dell'elaborato consta del *settimo capitolo*, nel quale saranno presentati alcuni dettagli relativi all'implementazione della piattaforma.

Infine con l'*ottavo capitolo* chiudiamo il presente lavoro, riportando le nostre considerazioni conclusive ed accennando agli sviluppi futuri in serbo per Stardust.

Parte I

La progettazione

Capitolo 1

Introduzione

Nel presente capitolo parleremo della navigazione indoor. Partendo dalla sua definizione ed evoluzione negli ultimi anni, la nostra attenzione sarà poi volta alla necessità di questa tecnologia ed alle varie metodologie ad oggi esistenti per la sua realizzazione. L'obiettivo è quello di presentare una panoramica il più chiara ed esaustiva possibile, quindi un buon punto di partenza per la successiva argomentazione delle scelte progettuali ed implementative alla base di Stardust. Questa piattaforma infatti, come vedremo nel corso della successiva esposizione, è stata pensata per la progettazione di percorsi e punti di interesse, atti all'ottenimento di TAG orientati per l'introduzione del servizio di navigazione indoor in un qualsiasi edificio.

1.1 Navigazione indoor: cos'è

Fino a pochi anni fa il termine navigazione era utilizzato dai più nella sua accezione di viaggio in mare. Oggigiorno, grazie all'avvento di piattaforme rivoluzionarie, quali Google Maps, e all'utilizzo di massa di dispositivi mobili capaci di supportare software complesso, con navigazione si intende anche un insieme di servizi capaci di: determinare posizionamento e orientamento delle persone come delle cose, siano queste ferme o in movimento; indicare, dato un punto di arrivo, percorsi stradali e tratti secondari, in accordo alle regole della viabilità ed alle variabili preferenziali a disposizione dell'utente

Ad oggi possiamo affermare che questi servizi di navigazione vantano un utilizzo di massa e sono un supporto ormai insostituibile per le persone e per le aziende, quindi tanto per fini personali quanto per quelli lavorativi.

Probabilmente non basterebbero tutte le pagine della presente tesi per elencare gli utilizzi ed i vantaggi introdotti dalla navigazione digitale, tuttavia dobbiamo sottolineare che questo servizio ha interessato principalmente gli spostamenti in luoghi esterni.

Esiste un ulteriore ambito nel quale la navigazione sta riscontrando, soprattutto negli ultimi tempi, molte richieste da parte degli enti pubblici e privati: gli spostamenti in luoghi chiusi. Ed è qui che introduciamo un termine cruciale per il presente lavoro di tesi: indoor. L'etimologia ci dice che questa parola, proveniente dalla lingua inglese, significa "interno". Questo termine in generale viene utilizzato con l'accezione semantica di azioni svolte in spazi circoscritti in una specifica e limitata area. Di conseguenza con navigazione indoor si indica un servizio che si pone come obiettivo quello di supportare le persone nei loro spostamenti all'interno di strutture nelle quali l'orientamento risulta un aspetto critico.

Questo tipo di localizzazione, a differenza di quanto visto per il suo corrispettivo in ambienti esterni, non ha ancora raggiunto un punto di compimento nelle tecnologie e nelle modalità di funzionamento, principalmente a causa della sua recente introduzione. Inoltre, come avremo modo di analizzare in seguito, più nel dettaglio, la navigazione indoor, dato il suo utilizzo in edifici delimitati e strutturati per loro definizione, permette l'introduzione di più tecniche di posizionamento. Questa è la prima macro differenza rispetto alla navigazione in spazi aperti, che ha invece trovato nel GPS (Global Positioning System)[1] la soluzione ottima. Essendo gli edifici degli spazi chiusi, spesso irraggiungibili dal segnale satellitare e/o composti da più piani, il GPS non può essere considerato un'opzione papabile per la realizzazione della navigazione indoor.

Come vedremo Stardust è un editor il quale, sulla base di una preesistente tecnologia di posizionamento, che prevede l'utilizzo di TAG orientati, ha come obiettivo quello di rendere la creazione ed il mantenimento della navigazione indoor un processo semplice e veloce. Far diventare questo servizio più utilizzabile, rispetto alle piattaforme ad oggi esistenti, per la sua costruzione, rappresenta per noi un importante passo in avanti per la sua scelta da parte di enti pubblici e privati, e quindi uno sprono per una sua maggiore diffusione.

[1] **Global Positioning System**: Accurata struttura di navigazione e rilevamento in tutto il mondo basata sulla ricezione di segnali provenienti da una serie di satelliti orbitanti [1].

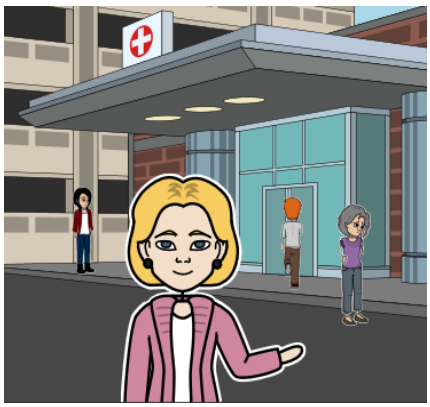
1.2 L'orientamento all'interno degli edifici

Esistono svariate strutture pubbliche o private nelle quali l'orientamento può risultare piuttosto complicato. Anche installando numerose mappe per le indicazioni dei punti di interesse in una struttura, spesso è necessario un sistema di posizionamento più capillare e dinamico. Pensiamo ad esempio alle centinaia di aule ed uffici dislocati fra i vari edifici all'interno delle Università, oppure agli ambulatori ed alle divisioni nelle Strutture Ospedaliere. Questi contesti sono caratterizzati da alcuni aspetti per noi di enorme rilievo:

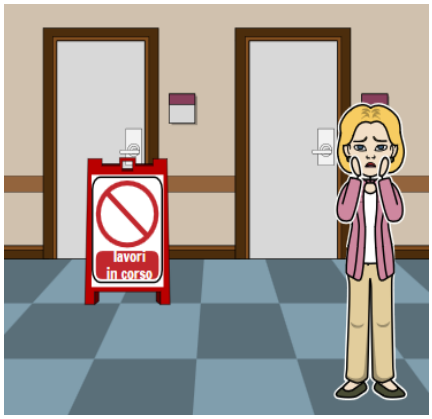
- Frequenti riorganizzazioni, con spostamento di personale e servizi da una postazione ad un'altra, all'interno dello stesso edificio o in edifici adiacenti;
- Stringenti richieste da parte del pubblico sui tempi di raggiungimento di una specifica meta;
- Presenza di un'elevata mole di persone, delle quali la maggior parte ha una conoscenza limitata o del tutto assente del luogo in cui si trova;

Per capire come la soddisfazione di tali requisiti rende le segnalazioni grafiche o verbali spesso incoerenti e/o poco esaustive, consideriamo un caso reale all'interno di una Struttura Ospedaliera:

Laura è una casalinga di 42 anni. Oggi ha una visita ambulatoriale con il Dott. Rossi alle ore 16:00. È la prima volta che si reca in quella struttura ospedaliera, poiché ha prenotato la visita telefonicamente. Laura ha viaggiato in auto ed ha utilizzando Google Maps per raggiungere la sua destinazione.



Una volta giunta all'interno della Struttura Ospedaliera, Laura vorrebbe chiedere indicazioni su come raggiungere l'ambulatorio del Dott. Rossi. Ben presto si rende conto che in accettazione vi è una fila interminabile. Laura non ha molto tempo a disposizione, decide così di cercare da sola.



Finalmente, dopo aver chiesto indicazioni in giro, spesso contraddittorie, Laura trova l'ambulatorio del Dott. Rossi. Sfortunatamente però in quella stanza ci sono dei lavori in corso, e di sicuro il medico che cerca sarà stato spostato in un altro studio. Ma dove? Mancano solo 10 minuti alle 16:00 e Laura non sa come fare per arrivare in tempo.

Un sistema di navigazione indoor, sempre presente ed aggiornato, avrebbe di sicuro risolto le problematiche appena mostrate nello storyboard.

Crediamo infatti che l'immediatezza, la chiarezza e la precisione, renderebbero il servizio digitale di localizzazione il luoghi chiusi la scelta preferita da parte del pubblico, anche in presenza di altre modalità di orientamento, come le mappe o la richiesta a persone sul posto. Una tale affermazione può essere supportata dal fatto che, per quanto la rete stradale sia piena di indicazioni, la maggior parte delle persone, quando si trova o deve raggiungere un luogo che non conosce, non alza lo sguardo ai segnali stradali ma apre Google Maps sul proprio cellulare.

Alla luce di tali osservazioni riteniamo corretto considerare la navigazione indoor un servizio che sarà sempre più richiesto, e quindi un ottimo scenario nel quale poter inserire nuove idee e realizzazioni.

1.3 Le piattaforme disponibili

Ad oggi esistono svariate aziende le quali, a partire da differenti tecnologie di posizionamento, offrono il servizio di navigazione indoor.

Citiamo come primo esempio il sistema Project Tango, realizzato da Google e Lenovo, il quale utilizza particolari rappresentazioni degli spazi chiusi per la navigazione di interni. Questa piattaforma prevede la mappatura 3D dell'ambiente, tramite sensori e camere del dispositivo mobile, così da catturare i movimenti degli utenti nello spazio.

Un ulteriore esempio ci viene dato da Smart Track, un sistema che ha dato il nome alla rispettiva startup, spinoff dell'Università degli Studi di Genova, che consente la navigazione grazie all'installazione di BLE (Bluetooth Low Energy) nei punti di interesse della struttura, così da intercettare i dispositivi degli utenti durante i loro spostamenti.

Queste sono solo due delle tante possibilità attualmente esistenti per la micro-localizzazione. Tutte hanno come obiettivo comune quello di limitare lo smarrimento in luoghi chiusi di grandi dimensioni. Ed in effetti lo fanno, mettendo a disposizione degli utenti finali dei servizi davvero potenti e facili da utilizzare. Tuttavia vi è un aspetto non banale da considerare: coloro i quali vogliono introdurre il servizio di navigazione indoor in un edificio, necessitano dell'intervento di personale qualificato e/o dell'installazione di dispositivi elettronici. Quindi il collo di bottiglia non è rappresentato dalle applicazioni fornite agli utenti finali, bensì dai processi di costruzione delle mappe di orientamento.

La nostra piattaforma, Stardust, prevedendo l'utilizzo di TAG ARRU[2], non richiede ai dispositivi mobile di mappare gli spazi 3D, né i costi di installazione di apparecchiature di puntamento radio, a differenza di quanto visto per le altre tecnologie ad oggi esistenti. Risulta chiaro quindi che i nostri utenti sono coloro i quali vogliono offrire il servizio di navigazione indoor.

1.4 Progettare la navigazione indoor

Il progetto Stardust è stato pensato con l'obiettivo di rendere la progettazione e l'installazione, richieste per la navigazione indoor in un qualsiasi edificio, un processo intuitivo, veloce ed economico. Questo proposito si traduce nella creazione non di un servizio che richiede l'intervento di esperti, ma bensì di un prodotto completo ed utilizzabile

[2] ***Augmented Reality for the Rest of Us***: Sistema brevettato da TonicMinds consistente in tag dotati di direzionalità.

direttamente dagli utenti. Il nostro proposito è stato quello di realizzare, per chiunque volesse rendere navigabile un particolare ambiente, una piattaforma costruita sui principi di immediatezza, accuratezza e di auto manutenibilità.

Per immediatezza intendiamo la possibilità di poter comprendere la logica di interazione con il minimo sforzo, nella prospettiva di utilizzo anche da parte di utenti non specializzati in materia architetturale.

L'aspetto dell'accuratezza consiste nella previsione del maggior numero di variabili previste all'interno di un edificio, la cui presenza influenza più o meno direttamente la determinazione dei percorsi di navigazione. Pensiamo a tal proposito a scale, ascensori o percorsi riservati al personale, solo per fare qualche esempio.

In fine vi è l'auto manutenibilità, vista come la possibilità di riflettere immediatamente i cambiamenti dell'ambiente, nella struttura quanto nell'organizzazione, senza l'intervento di personale specializzato.

1.5 Considerazioni

Non sarebbe improprio affermare che per la maggior parte delle persone i servizi di geo localizzazione costituiscono ormai un aiuto costante e prezioso. L'enorme successo registrato per le tecnologie di supporto agli spostamenti in ambienti esterni ci spinge verso la ricerca e lo sviluppo nell'ambito della navigazione indoor. Un esempio ci è stato dato negli ultimi anni dalla proliferazione di nuove tecnologie ed investimenti fatti nel settore. Un aspetto critico è tuttavia costituito dalla fase di costruzione ed installazione dell'infrastruttura necessaria al rilevamento della posizione e direzione di spostamento del singolo nello spazio. Le tecnologie attualmente adoperate, infatti, richiedono l'intervento di specialisti nel settore e/o l'installazione di dispositivi wireless per la localizzazione. È a questi limiti realizzativi che noi vorremmo dare una soluzione. Creare un ambiente di progettazione semplice ed efficace, articolato secondo i crismi di un editor di disegno, che possa coprire tutte le funzionalità necessarie per una accurata navigazione indoor, e che risponda nel tempo ai cambiamenti dell'ambiente, è stato sin dall'inizio lo scopo del nostro lavoro di tesi.

Capitolo 2

Stardust

In questo capitolo entreremo nel cuore della piattaforma Stardust. Partendo dall'analisi dei requisiti e del benchmarking, arriveremo alla presentazione delle funzionalità principali previste dall'editor.

Come avremo modo di capire nel corso del presente capitolo, gli strumenti messi a disposizione dalla piattaforma creano una sorta di puzzle, nel quale ogni nuova entità da inserire prende senso in relazione agli elementi già presenti e quelli che dovranno essere inseriti in una fase successiva.

La nostra analisi seguirà un approccio bottom-up. Partendo quindi da una rappresentazione semplificata del sistema, aggiungeremo "un pezzo alla volta", fino ad arrivare alla presentazione della piattaforma nella sua interezza.

Per ottimizzare la delicata e complessa fase di progettazione dell'editor, nel quale ogni possibile flusso delle azioni dell'utente deve essere previsto e reso funzionale, abbiamo utilizzato specifici principi di progettazione, che hanno rappresentato per noi una risorsa insostituibile. Un editor infatti, sia questi per il disegno o per la scrittura di testi, dovrà prevedere un parco di opzioni e sotto-funzionalità per coprire i più svariati aspetti decisionali dell'utente, e le complesse variabili relative ad ogni specifico progetto, mantenendo sempre alta l'attenzione su di una user experience ottimale.

2.1 Cos'è Stardust

Stardust è una piattaforma online provvista di un editor per la progettazione ed il mantenimento di percorsi atti alla navigazione indoor all'interno di un edificio.

L'obiettivo principale di questo sistema è quello di rendere il servizio di navigazione indoor un prodotto, ovvero uno strumento progettabile, installabile e manutenibile dai soli utenti richiedenti, senza l'intervento di personale esterno qualificato.

Stardust si basa su menù di funzionalità user friendly, che possiamo suddividere in tre gruppi principali:

1) Funzionalità preliminari;
2) Funzionalità progettuali;
3) Funzionalità pratiche;

Le funzionalità preliminari sono quelle che entrano in gioco prima della fase di disegno dei percorsi vera e propria, e sono:

- Inserimento e manipolazione di una planimetria come file immagine;
- Definizione della planimetria come specifico piano;
- Inserimento degli edifici appartenenti allo stesso complesso;
- Organizzazione dei piani negli edifici di una struttura;

Seguono le funzionalità progettuali, ovvero quelle che permettono la costruzione dei percorsi di navigazione. Queste sono:

- Inserimento di nodi rappresentati punti di interesse sulla planimetria del piano, quali porte, scale, ascensori, toilette o TAG isolati nei punti di snodo dei percorsi;
- Creazione di archi fra i nodi, rappresentati tutti i tratti percorribili;
- Definizione, tramite etichette, di tutti gli elementi inseriti sul foglio di lavoro;
- Aggiunta di percorsi privati, come ad esempio quelli accessibili dal solo personale della struttura;

- Organizzazione dei piani in sottogruppi semanticamente appartenenti a reparti differenti;

- Dislocazione dei vari edifici, secondo la loro reale ubicazione nello spazio, su di una superficie;

Infine le funzionalità pratiche sono quelle relative agli output necessari per l'installazione dei TAG e all'aggiornamento informativo degli stessi. Queste sono:

- Geolocalizzazione del progetto;
- Generazione di file output contenenti i TAG necesari e le istruzioni per l'installazione degli stessi all'interno degli edifici;
- Impostazione delle variabili di navigazione, come data ed ora dalle quali dare il via agli aggiornamenti;

I tre gruppi di funzionalità appena esposti non devono essere intesi come parti separate della piattaforma, bensì costituiscono degli insiemi di strumenti legati semanticamente, la cui aggregazione edifica un flusso logico, che accompagna le azioni dell'utente dall'inserimento della prima planimetria fino alla conclusione del suo progetto. Questo è reso possibile grazie dal fatto che tutte le funzionalità appena elencate sono state corredate da specifici menù che, oltre a definire i vari aspetti caratterizzanti, rendono il susseguirsi delle azioni un processo intuitivo.

2.2 Analisi dei requisiti

Un editor di progettazione richiede, prima di tutto, la contestualizzazione del suo utilizzo e l'individuazione di tutte le funzionalità necessarie per il perseguimento del suo scopo.

Poiché Stardust si basa sul disegno di percorsi e punti di interesse su planimetrie, abbiamo ritenuto necessario, prima di tutto, analizzare i bisogni degli utenti. Questi infatti costituiscono la maggior fonte informativa ed un ottimo punto di partenza sul quale costruire un sistema concepito per i loro bisogni. Una progettazione solipsistica rischia infatti di sbilanciare la logica verso aspetti che, in fase di utilizzo della piattaforma, potrebbero risultare non necessari o addirittura del tutto inutili.

Successivamente, una volta raccolte le esigenze degli utenti, abbiamo analizzato alcuni fra i più comuni editor, la maggior parte dei

quali appartenenti alla suite Microsoft Office[3], così da tradurre le richieste del pubblico in funzionalità, dotate di una specifica individualità, ma la cui unione potesse dar vita ad un flusso organizzato, logico ed uniforme.

2.2.1 Interviste agli utenti

Partendo dall'identificazione degli aspetti principali ricoperti da Stardust, e tramite l'analisi di tool di progettazione CAD[4], abbiamo identificato le modalità di interazione, le competenze richieste ed i vincoli imposti agli utenti. Questa fase è stata utile alla determinazione di una serie di domande, più o meno tecniche, così da rendere le interviste previste il più possibile fruttuose. Tale scelta di modalità di analisi del target è stata dettata dall'esigenza di raccogliere il maggior numero di informazioni su alcuni aspetti per noi più ostici, e per far affiorare esigenze, e quindi funzionalità, nuove. Infatti, un'analisi tramite sondaggio, per fare un esempio, avrebbe imposto dei vincoli nelle risposte e quindi un limite non accettabile nella prima fase di sviluppo della piattaforma. Queste interviste sono state svolte tramite un numero variabile di domande a risposta aperta, rivolte ad alcuni utenti facenti parte della direzione di alcune strutture pubbliche, interessati all'introduzione di tecnologie per la navigazione indoor.

In particolare gli edifici suddetti sono l'Ospedale Regina Margherita (TO) e lo stesso Politecnico di Torino. Tali strutture hanno rappresentato per noi una risorsa insostituibile, date le loro architetture complesse e la diversità di pubblico presente nelle stesse, facendo affiorare così la maggior parte degli aspetti esistenti in una realtà architetturale dove l'orientamento spesso rappresenta un aspetto critico.

Le nuove conoscenze acquisite ci hanno dato la possibilità di identificare, oltre alle funzionalità da aggiungere, anche quelle da modificare o da scartare rispetto a quelle previste nelle ipotesi preliminari.

Il primo bisogno, chiaramente emerso dalle interviste, è stato quello di poter beneficiare di un prodotto gestibile in modo completamente autonomo. Per gestione autonoma si intende la creazione, l'installazione e la manutenzione del servizio di navigazione indoor da parte dello stesso personale dell'edificio ospitante. In pratica, dopo una breve fase di studio all'utilizzo della piattaforma, una qualunque

[3] **Microsoft Office**: suite di applicazioni desktop. Fra le più popolari ricordiamo Microsoft Word, Microsoft Excel e Microsoft PowerPoint.

[4] **Computer-Aided Drafting**: Indica il disegno tecnico assistito dall'elaboratore.

persona, con i dei requisiti di informatica di base, deve poter essere in grado di gestire il sistema e capirne la logica in tutte le sue parti. Da qui sono nate delle parole chiave, da tenere ben presenti nella successiva fase di progettazione, ovvero:

- praticità;
- semplicità;
- produttività;

Per praticità intendiamo la presenza di un flusso interattivo unico ed intuitivo, che non preveda l'esistenza di "più modi per fare la stessa cosa", evitando quindi funzionalità dispersive, a favore di una più ricercata maneggevolezza.

Il concetto di semplicità, che in assenza di termini di paragone potrebbe risultare poco chiaro in un contesto come questo, lo intendiamo qui nell'accezione di similarità fra Stardust e altri tool simili, ad oggi di maggior impiego. Per fare qualche esempio, la maggior parte degli editor, se non tutti, prevedono un menù principale, costituito da pulsanti, dal quale accedere agli strumenti principali. Ancora, il click del mouse su uno di questi pulsanti permette di accedere a tutte le sotto-funzionalità correlate. Trovare sempre questo pattern semplifica di molto la vita dell'utilizzatore, che dovrà quindi imparare solo un sottoinsieme, spesso molto limitato, di funzionalità dedicate ad una specifica piattaforma.

La produttività risulta quindi una diretta conseguenza della praticità e della semplicità. Anche strutture composte da più edifici multipiano devono poter essere progettate in tutte le loro entità navigabili in un tempo ragionevole.

2.2.2 Analisi del benchmarking

Nel paragrafo 1.3 abbiamo parlato di due particolari tecnologie, le più comuni, utilizzate per la navigazione indoor, e di altrettante aziende che le adoperano per offrire questo servizio. Continuiamo qui questo discorso, andando ad analizzare con maggior dettaglio il benchmarking, quindi studiando i competitor, per introdurre le principali novità offerte da Stardust.

Le piattaforme da noi scelte sono state: Project Tango, di Google e Lenovo; Smart Track, start up spin off dell'Università degli studi di Genova.

Sottolineiamo sin da subito che, indicare Project Tango come un servizio per il rilevamento degli utenti e dei loro spostamenti all'interno di luoghi chiusi, sarebbe piuttosto limitativo. Stiamo parlando infatti di un sistema estremamente vasto che permette di ottenere molto più che informazioni di navigazione. Tuttavia, non essendo noi interessati in questo conteso alle sue innumerevoli applicazioni, consideriamo questa piattaforma solo in relazione al suo utilizzo come strumento di navigazione indoor.

In pratica Project Tango, previa fase di mappatura 3D di un edificio, utilizza camera e sensori del dispositivo mobile dell'utente per riconoscere la posizione dello stesso nello spazio, rilevando ostacoli e cammini. Le varie indicazioni, in questo caso i percorsi da intraprendere, vengono mostrate sul dispositivo in realtà aumentata, sovrapposte alle immagini restituite dalla camera.

Smart Track invece fornisce il sistema di navigazione indoor tramite l'installazione di particolari dispositivi elettronici per il posizionamento. In particolare occorre distribuire ad ogni punto di interesse (ambulatorio/ufficio) dell'edificio un dispositivo radio Bluetooth Low Energy (BLE) alimentato a batteria, interoperabile con gli smartphone. L'utente quindi può raggiungere il punto di interesse selezionato utilizzando il proprio smartphone come un navigatore. L'applicazione sviluppata da Smart Track visualizza la posizione dell'utente e traccia il percorso più breve che questi deve seguire per raggiungere la destinazione desiderata [2].

Queste due grandi piattaforme, che forniscono agli utenti finali sistemi indubbiamente ottimi, hanno tuttavia un limite, per il quale Stardust può essere la soluzione. Sia Project Tango che Smart Track, infatti, forniscono dei servizi e non un prodotto. Ciò significa che l'utente richiedente deve necessariamente avvalersi dell'intervento di queste aziende in ogni fase del lavoro, dal progetto al mantenimento dello stesso nel tempo. Inoltre, a causa delle tecnologie utilizzate, sono necessari, a valle, dispositivi con caratteristiche tecniche, e quindi costi, non alla portata di tutti.

Stardust invece non richiede l'intervento di aziende esterne, in nessuna fase del lavoro, né di dispositivi mobile dalle caratteristiche stringenti. L'utente richiedente può quindi, una volta aperta la piattaforma, progettare i percorsi del suo edificio come su di un normale tool di disegno, stampare i TAG, installarli e mantenerli aggiornati nel tempo, tutto con il minimo sforzo. Il lavoro viene svolto tramite l'inserimento di nodi ed archi in un ambiente 2D, notoriamente meno complicato rispetto alla gestione di modelli in 3D. Inoltre i TAG non hanno neanche lontanamente i costosi e la fragilità dei BLE, e non

richiedono al dispositivo finale costose caratteristiche hardware e software per il riconoscimento.

2.3 Prototipo e testing

La fase successiva alle interviste e all'analisi del benchmarking è stata quella relativa alla creazione di un prototipo di disegno, contenente alcune delle funzionalità previste dalla piattaforma Stardust finale, quali l'inserimento di nodi ed archi su di una planimetria, che ci ha permesso di avviare il processo di testing.

Gli utenti partecipanti sono stati selezionati con differenti età e qualifiche, e a tutti era richiesta una conoscenza di base di alcuni software appartenenti alla suite Microsoft Office. Questa è stata una fase cruciale del nostro lavoro poiché, per un editor che si rispetti, è necessario sin dal principio perfezionare l'interazione fra utenti e strumenti.

Ogni tool di disegno, infatti, possiede un proprio insieme di comandi, siano questi shortcut da tastiera o selezioni da mouse, che devono agevolare quanto più possibile lo svolgimento del lavoro. Questi aspetti sono tutt'altro che scontati poiché, senza un'attenta osservazione delle azioni istintive dell'utente intento ad una determinata azione, si rischia di creare un'interazione innaturale ed infruttuosa. Per fare un esempio esplicativo del concetto appena esposto, proviamo a definire tutte le possibili alternative progettuali di un tool il cui unico compito è quello di disegnare nodi collegati da archi. Come vedremo, queste due entità sono le colonne portanti della piattaforma Stardust, tuttavia per il momento le estrapoleremo in parte dal contesto, senza quindi associarle a zone o percorsi specifici su di una planimetria, per concentrarci sulle sole azioni degli utenti. Consideriamo quindi un editor "giocattolo" per l'inserimento e l'unione tramite archi di due nodi:

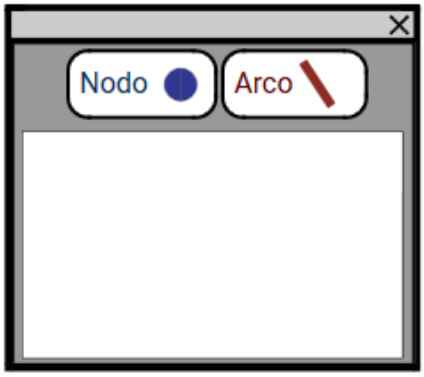


Figura 2.1: Semplice editor per il disegno di nodi ed archi

Notiamo dalla figura sopra riportata la presenza di due pulsanti, rispettivamente per l'inserimento di un nodo e di un arco, e di un foglio di lavoro. Precisiamo che l'immagine mostra una versione estremamente semplificata del prototipo reale, così da convogliare le attenzioni solo su di un sottoinsieme delle funzionalità realmente costruite.

Una parte di utenti intenti ad inserire un nodo ha considerato lo strumento come un pulsante e quindi, dopo il click del mouse sullo stesso, ha spostato il cursore sul foglio di lavoro nel punto dove, tramite un altro click, ha inserito poi il nodo.

Un altro gruppo di utenti ha invece interpretato lo strumento come un'entità dalla logica drag and drop[5]. Di conseguenza la loro interazione ha previsto un click del mouse prolungato durante il trascinamento del nodo sul foglio di lavoro, ed il rilascio nel punto di inserimento desiderato.

La fase di testing del prototipo è servita proprio per determinare se la maggior parte degli utenti tendesse ad utilizzare istintivamente l'uno o l'altro approccio di interazione.

Una volta ottenute le statistiche, e scelto il gruppo a percentuale maggiore, la fase successiva ha richiesto di "forzare", tramite scelte grafiche ad hoc riportate sul prototipo, anche la minoranza degli utenti a comprendere immediatamente come utilizzare uno strumento prima ancora di dare inizio all'interazione. La validità delle modifiche apportate è stata quindi verificata in fasi di testing successive.

Poiché la maggior parte degli utenti ha considerato lo strumento come un pulsante, in fase di progettazione post testing sono stati

[5] **drag and drop**: susseguirsi delle azioni di click, trascinamento e posizionamento di un oggetto all'interno dell'interfaccia grafica di un computer.

riversati tutti gli aggiornamenti apportati al prototipo, che hanno indotto anche gli altri utenti, quelli che durante la prima fase di testing avevano considerato gli elementi come rispondenti alla logica del drag and drop, a comprendere sin da subito, al secondo utilizzo, che quella funzionalità doveva essere attivata tramite il click sul relativo pulsante. Nel caso appena considerato però le diverse interpretazioni di utilizzo degli utenti, in relazione ad uno stesso strumento, sono state estremamente limitate ed in buona parte anche predicibili. In altri casi invece lo scenario è risultato più articolato, e la fase decisionale ha quindi richiesto l'intervento di più considerazioni rispetto ai soli dati statistici.

Considerando sempre lo stesso esempio, sul nostro editor giocattolo, partiamo ora dalla situazione in cui l'utente ha già inserito due nodi e deve unirli tramite un arco. A tutti è stato indicato il fatto che un arco poteva essere tracciato solo a partire da un nodo, e che doveva terminare in un altro nodo. Anche in questo caso la totalità degli utenti si è divisa principalmente in due gruppi: quelli che utilizzavano un click del mouse per avviare il disegno, ed un altro click per terminarlo; quelli che mantenevano attivo il click durante tutta la fase di tracciamento del segmento, per poi rilasciarlo nel nodo di arrivo. La scelta di quale di queste due opzioni mantenere non è stata banale in quanto, a differenza dell'inserimento di un nodo, la creazione di un arco non è un'operazione circoscritta nel tempo, e ha richiesto quindi l'ulteriore osservazione di tutte le azioni svolte dall'utente, volontariamente o involontariamente, lungo tutta la durata di tracciamento di una linea. Stiamo parlando quindi di una situazione di questo tipo:

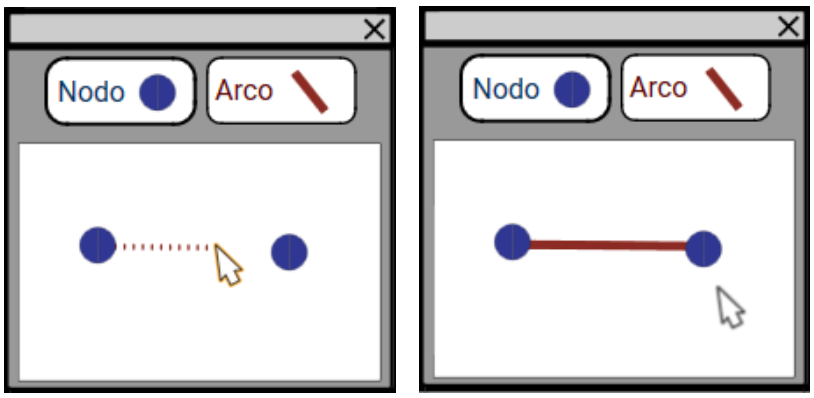


Figura 2.2: Processo di tracciamento di un arco fra due nodi.

Inoltre, essendo un arco per definizione nascente e terminante in un nodo, nelle opzioni di tracciamento abbiamo dovuto tener conto anche della risposta del sistema nei casi in cui l'utente tentava di concludere il tracciamento dell'arco in punti del foglio di lavoro non consentiti, ovvero dove non era già presente un nodo. In casi come questi il sistema può fare due cose: o l'undo dell'arco oppure la creazione di un nodo ex novo nel punto di terminazione dello stesso. Poiché la generazione automatica di un nodo avrebbe potuto confondere l'utente nella sua definizione, abbiamo optato per la prima soluzione.

Ancora, prima di arrivare al nodo di arrivo, oltre ad input in zone non consentite, sono state osservate altre pratiche comuni da dover gestire, quali click del tasto destro del mouse o posizionamento del cursore al di fuori dell'area di lavoro, ad arco non ancora terminato. Avallare uno di questi comportamenti avrebbe significato riconsiderare la risposta del sistema in altre fasi, una pratica che può ripercuotersi su di un numero non determinabile a priori di altri strumenti del tool. D'altra parte forzare l'utente a muoversi in una sola direzione richiede una certa sicurezza che l'imposizione data non possa essere in qualche modo controproducente, e che non dia all'utente una visione di funzionamento troppo rigida.

Quindi ogni interazione è stata concepita non solo in relazione alla naturalezza delle azioni svolte dagli utenti per l'attivazione della funzionalità, ma anche rispetto a tutti i possibili input non previsti durante la fase di disegno vera e propria.

Lo scenario decisionale come abbiamo visto è andato gradualmente a complicarsi. Ribadiamo che l'esempio appena esposto considera un sottoinsieme molto ristretto delle azioni che possono essere svolte da un utente durante l'utilizzo di uno strumento, e che di questi ne sono stati esposti solo due: disegno di un nodo; disegno di un arco.

Stardust attualmente prevede la coesistenza nel tool di disegno di all'incirca trenta strumenti differenti, corredati da specifici sotto-menù, ottimizzati rispetto alle varie azioni degli utenti ed incastrati in un unico flusso logico di progettazione.

Negli esempi considerati abbiamo avuto modo di mostrare come la fase di test ci abbia permesso di osservare le azioni svolte dagli utenti nell'iterazione con un prototipo della piattaforma Stardust. Tuttavia abbiamo anche sottolineato come una scelta progettuale possa ripercuotersi a macchia d'olio anche su altre affini, rendendo quindi la sola osservazione degli utenti non sufficiente. A tal fine è risultata per noi di estrema utilità l'adozione di alcuni principi di progettazione, ben noti in letteratura, che saranno discussi nel prossimo paragrafo.

2.3 Principi fondamentali per la progettazione

Grazie alle interviste agli utenti, all'analisi del benchmarking ed al testing del prototipo, abbiamo ricavato i requisiti necessari che la piattaforma Stardust deve soddisfare. In particolare l'utente dovrà:

1. Lavorare con un editor simile a quelli già largamente utilizzati, così da avere un numero limitato di funzionalità da imparare da zero;
2. Poter lavorare su planimetrie senza avere specifiche competenze di strumenti CAD;
3. Poter mantenere il proprio progetto aggiornato nel tempo, apportando modifiche con il minimo sforzo, e con la massima chiarezza delle operazioni da svolgere in caso di aggiunta, spostamento o rimozione di un TAG;
4. Essere in grado di interagire con la piattaforma nel minor tempo possibile;
5. Poter disporre di un parco di opzioni tale da rendere il proprio progetto il più ricco possibile di informazioni utili alla successiva navigazione indoor;
6. Avere una rappresentazione architetturale di un edificio estremamente organizzata e navigabile;
7. Disporre di strumenti di disegno pensati e funzionanti appositamente per le sue necessità;
8. Avere sempre a disposizione una guida veloce all'utilizzo;
9. Essere supportato anche nella fase finale di installazione dei TAG;
10. Definire i percorsi di navigazione sui quali lavora, sia nel tempo che nello spazio;
11. Non doversi preoccupare di come i migliori percorsi di navigazione saranno poi calcolati e resi disponibili;
12. Poter costruire progetti di grandi dimensioni, come strutture multi edificio, e lavorare con più versioni degli stessi;

Costruiti questi requisiti, siamo poi passati alla fase di progettazione degli stessi. Considerando il fatto che ognuna di queste funzionalità prevede una serie di variabili, che devono incastrarsi l'un l'altra in un flusso unico, abbiamo ritenuto necessaria l'adozione di alcune regole che ci potessero guidare e supportare durante questa fase così importante del nostro lavoro. In particolare abbiamo seguito i quattro principi fondamentali della progettazione [3], per noi delle vere e proprie colonne portanti per la creazione di software efficace ed efficiente. Partiamo quindi dal primo di questo principi:

principio di progettazione n. 1*: La semplicità favorisce la regolarità.*

Su questo principio di progettazione si basa la costruzione dei primi tre requisiti prima elencati:

1. Lavorare con un editor simile a quelli già largamente utilizzati, così da avere un numero limitato di funzionalità da imparare da zero;
2. Poter lavorare su planimetrie senza avere specifiche competenze di strumenti CAD;
3. Poter mantenere il proprio progetto aggiornato nel tempo, apportando modifiche con il minimo sforzo e con la massima chiarezza delle operazioni da svolgere in caso di aggiunta, spostamento o rimozione di un TAG;

Il concetto chiave per la soddisfazione di questi requisiti è stato quindi quello della semplicità. Per quanto riguarda la soddisfazione del primo, quindi, sono stati analizzati alcuni layout dei più comuni tool di progettazione. Questo ci ha permesso di costruire lo scheletro dell'editor di Stardust:

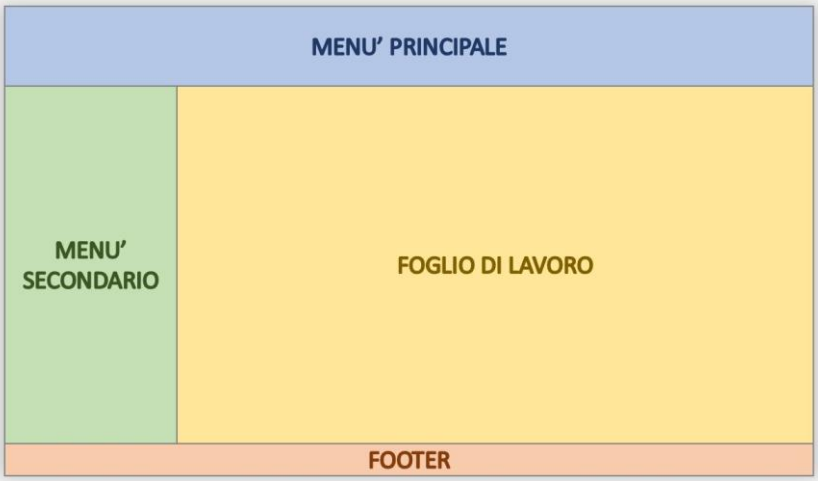


Figura 2.3: Divisione dello schermo in quattro aree distinte.

Come mostrato dalla figura sopra riportata, l'intero editor è suddiviso in quattro aree distinte: un menù principale, statico, dal quale accedere alle funzionalità principali; un menù secondario, dinamico, contenente varie opzioni, che dipendono dalla specifica funzionalità scelta dal menù principale; un foglio di lavoro, ovvero l'area preposta all'inserimento delle planimetrie sulle quali l'utente disegnerà percorsi e punti di interesse; un footer che prevede l'accesso a comandi diretti, come quelli di zoom-in e zoom-out della planimetria. La semplicità di questa strutturazione risiede nel fatto che le aree in cui il layout è suddiviso sono solo quattro e sono immutabili, ovvero restano le stesse in ogni fase del progetto.

Passiamo ora al secondo requisito, ovvero quello relativo alla possibilità di lavorare su planimetrie senza la necessità di dover imparare ad utilizzare i tipici strumenti CAD che le manipolano. In questo caso abbiamo applicato il principio di semplicità andando a trasformare la planimetria in un blueprint[6]. Pertanto l'utente può lavorare con un file immagine della planimetria, impostato come background del foglio di lavoro, che ha come unico scopo quello di rendere il disegno sovraimpresso facilmente tracciabile e dimensionalmente corretto. Quindi, non solo l'utente non necessita delle competenze richieste dagli strumenti CAD, ma è anche facilitato durante la fase di tracciamento del disegno, con il blueprint che gli

[6] **Blueprint**: termine utilizzato in computer grafica per indicare un'immagine, rappresentate ciò che deve essere disegnato, impostata come background del foglio di lavoro, che ha come unico scopo quello di guidare l'utente nel tacciamento.

indica in ogni momento in quale area dell'edificio sta lavorando. L'immagine seguente mostra l'evoluzione dell'editor con l'aggiunta della planimetria, come blueprint, per l'inserimento nelle posizioni corrette di nodi ed archi:

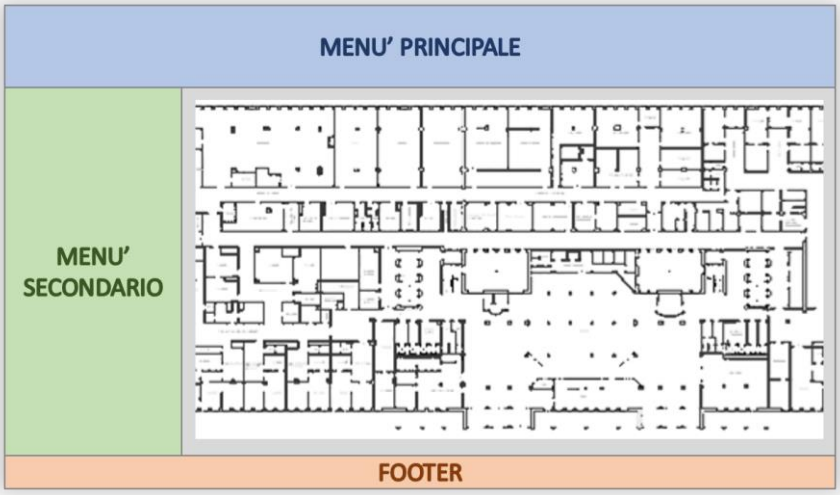


Figura 2.4: Inserimento planimetria nell'editor di Stardust.

Infine il principio di semplicità è stato applicato alla fase di aggiornamento di un progetto, ovvero a tutti quegli aspetti che entrano in gioco nel momento in cui l'utente deve modificare le informazioni associate a percorsi o punti specifici dell'edificio. Poiché questo processo può doversi tradurre o in un aggiornamento delle informazioni associate ad un TAG, oppure all'aggiunta o rimozione di uno di questi, questo stadio, se lasciato anche solo in parte al buon senso dell'utente, potrebbe complicare di molto il suo lavoro e comunque non garantire l'assoluta correttezza delle informazioni finali. Abbiamo quindi deciso di sgravare l'utilizzatore da ogni onere aggiungendo due importanti funzionalità. La prima è stata quella di dotare l'azione di "stampa TAG", che produce in output un file con i TAG da stampare, di un automatismo per il quale, se il progetto non risulta corretto in ogni suo aspetto, non può essere avviata. L'aspetto appena esposto, relativo alla correttezza di un progetto e alla sua determinazione da parte del sistema, sarà chiarito in seguito.

Sempre per il primo principio fondamentale della progettazione, applicato alla fase di generazione dei TAG, a ognuno di questi sono state associate specifiche istruzioni per il punto di inserimento, o spostamento e rimozione in caso di aggiornamento del progetto.

Infine è stato previsto un vincolo di sistema relativo all'immissione da parte dell'utente di ora e data dalla quale le modifiche inserite dovranno entrare in vigore. In questo modo le fasi di manutenzione, con relativa installazione, non creeranno disallineamenti informativi rispetto alle versioni precedenti.

Passiamo adesso al secondo dei quattro principi fondamentali per la progettazione:

principio di progettazione n. 2*: Minori sono le dimensioni, maggiore è la velocità.*

L'applicazione di questa regola, a differenza di quanto visto per quella precedente, non ha interessato il solo ambito della progettazione logica della piattaforma, ma si trova piuttosto sulla linea di confine fra quest'ultima e la fase relativa all'analisi dei requisiti tecnologici, affrontati al capitolo 3. Senza anticipare i contenuti relativi all'architettura scelta, possiamo dire che, poiché Stardust è stata concepita come una piattaforma online, la questione delle dimensioni, relativamente a tutte le informazioni ed i processi da dover gestire contemporaneamente, ha rappresentato un aspetto critico da non sottovalutare. Fra i dodici requisiti richiesti infatti, il quarto ci dice che l'utente deve:

4. Essere in grado di interagire con la piattaforma nel minor tempo possibile;

In effetti questa è una prerogativa necessaria, richiesta da qualunque prodotto informatico che possa essere considerato tale. Tuttavia creare un editor online richiede un'attenzione particolare relativamente alla velocità di risposta del sistema, che per applicativi simili, ma stand alone, entro un certo limite può essere data per scontata. Nel caso di Stardust tutto questo non si traduce solo in una serie di ottimizzazioni attuate in fase implementativa, ma anche in un'attenta analisi in fase progettuale. È stato infatti indispensabile determinare il numero di funzionalità necessario all'ottenimento del risultato, abbastanza dettagliato da rendere il prodotto efficace, ma senza mai sconfinare nel superfluo. Questa decisione ha interessato in particolare le richieste di:

5. Poter disporre di un parco di opzioni tale da rendere il proprio progetto il più ricco possibile di informazioni utili alla successiva navigazione indoor;

6. Avere una rappresentazione architetturale di un edificio estremamente organizzata e navigabile;

Questi, infatti, sono due aspetti che, per loro natura, richiedono la presenza di una mole elevata di dati e di funzionalità. A tal proposito, per ridurre i tempi di risposta del sistema, Stardust è stato strutturato modularmente. In particolare, pur essendo previsto un elevato numero di funzionalità, ogni azione dell'utente prevede l'entrata in gioco soltanto di un numero limitato di queste. Quindi è possibile disporre di tutti gli strumenti necessari, con tempi di risposta performanti. Ad esempio, in fase di disegno, l'utente può scegliere o la modalità nodo oppure quella arco. In presenza dell'una l'altra non è attiva, evitando così di sovraccaricare le risorse.

Anche per quanto riguarda la navigazione fra le planimetrie degli edifici, appartenenti alla stessa struttura, è possibile specificare in modo diretto a quale puntare, lasciando il resto dei dati a riposo. Facciamo quindi un passo in avanti verso l'evoluzione del sistema:

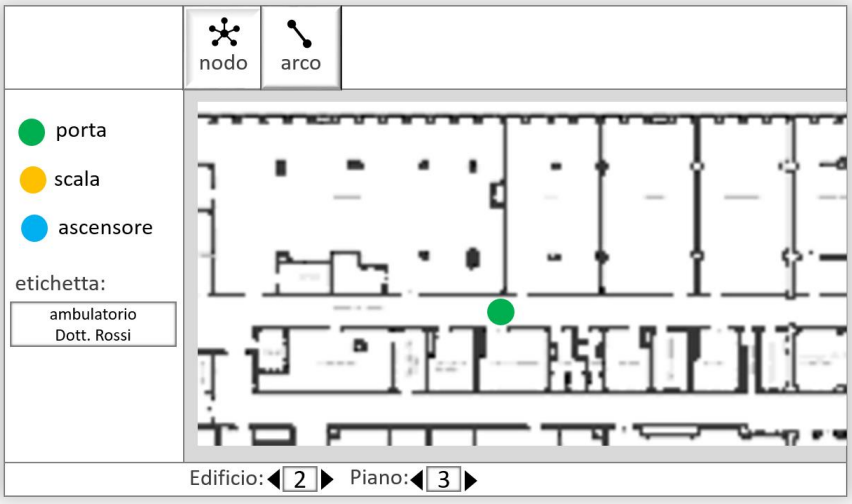


Figura 2.5: Prototipo in modalità di aggiunta nodo.

Possiamo notare come, in modalità disegno nodo, nel menù secondario sono disponibili solo le operazioni necessarie all'inserimento di questo elemento. Inoltre due semplici campi di input nel footer permettono lo spostamento immediato fra i vari piani ed i vari edifici dell'intera struttura. Tuttavia in ogni istante l'utente può lavorare su di una sola planimetria. Passiamo adesso al terzo principio fondamentale di progettazione:

principio di progettazione n. 3*: Rendi più veloce possibile le operazioni comuni.*

Su questo principio si basa la suddivisione fra: funzionalità primarie, che devono essere inserite nel menù principale; funzionalità complementari, che devono essere inserite nel menù secondario; funzionalità dirette, ovvero quelle che non prevedono particolari opzioni di funzionamento, che devono essere poste nel footer. I requisiti che ci hanno guidati in questa fase di scelta sono stati:

7. Disporre di strumenti di disegno pensati e funzionanti appositamente per le sue necessità;
8. Avere sempre a disposizione una guida veloce all'utilizzo;
9. Essere supportato anche nella fase finale di installazione dei TAG;

Il layout dell'editor è stato quindi costruito ponendo in primo piano tutti gli strumenti che sarebbero stati presumibilmente utilizzati più di frequente, come mostrato in figura:

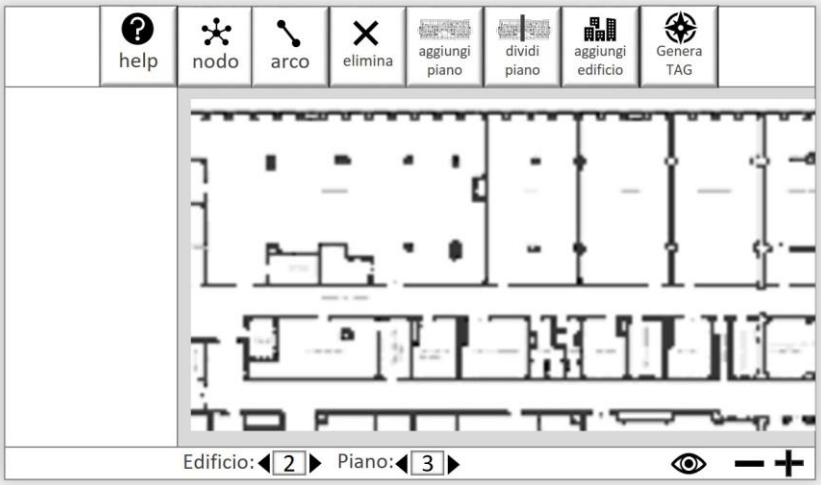


Figura 2.6: Aggiunta all'editor di footer e menù principale.

Nel menù principale è stato quindi previsto l'accesso diretto alle funzionalità fondamentali per il disegno e la progettazione.

Oltre all'inserimento di nodi ed archi, già introdotti precedentemente, troviamo adesso: help online; eliminazione nodo o

arco; aggiunta nuovo piano; divisione piano in sotto-aree; aggiunta nuovo edificio.

Nel footer sono invece presenti le operazioni dirette di zoom-in, zoom-out e nascondi/mostra planimetria. È stata inoltre inserita la funzionalità relativa alla produzione in output dei TAG. Anche questa infatti è un'operazione comune che deve essere accessibile nel minor tempo possibile.

Ricordiamo che dal pulsante *genera TAG* del menù principale è possibile ottenere un file stampabile contenente i marker previsti dal progetto in corso, ognuno dei quali corredato da specifiche istruzioni per l'installazione. Siamo giunti a questo punto all'ultimo dei quattro principi fondamentali per la progettazione:

principio di progettazione n. 4*: Un buon progetto richiede buoni compromessi.*

Questa regola è stata tenuta in considerazione soprattutto per il soddisfacimento degli ultimi tre requisiti, ovvero:

10. Definire i percorsi di navigazione sui quali lavora, sia nel tempo che nello spazio;
11. Non doversi preoccupare di come i migliori percorsi di navigazione saranno poi calcolati e resi disponibili;
12. Poter costruire progetti di grandi dimensioni, come strutture a multi edificio, e lavorare con più versioni degli stessi;

Il compromesso in questi casi si trova fra la semplicità richiesta e la necessità di evitare ogni possibile incoerenza fra i dati del progetto Stardust e l'applicazione a valle che li usa per la navigazione indoor. Questo si traduce nella necessità di dover suddividere in modo furbo il carico di lavoro fra le azioni richieste dall'utente e gli automatismi del sistema. In particolare la definizione nel tempo e nello spazio di un progetto è stata lasciata a carico dell'utente, mentre tutta la logica relativa al calcolo dei percorsi di navigazione più brevi è stata affidata completamente al sistema.

La costruzione di progetti di grandi dimensioni e la possibilità di poter lavorare con più versioni degli stessi è stata gestita invece tramite una struttura modulare della piattaforma. Un progetto quindi è stampabile, ovvero è possibile generare i TAG richiesti, solo se questi è stato geolocalizzato. In questo modo l'applicazione mobile può segnalare all'utente quando si trova nei pressi di un edificio nel quale è presente il servizio di navigazione indoor.

L'altro compromesso si trova fra il bisogno di rendere il sistema online il meno complesso possibile, e la necessità degli utenti di non preoccuparsi di come i percorsi di navigazione migliori vengano calcolati e resi disponibili. L'approccio utilizzato in questo caso è stato quello di richiedere all'utilizzatore alcune informazioni indispensabili sull'architettura della struttura, utili al calcolo automatico dei percorsi più brevi.

Ad esempio è necessario inserire sulla planimetria i nodi che rappresentano i punti in cui sono presenti delle scale e quelli in cui invece vi sono degli ascensori. Ognuno di questi elementi deve essere dotato poi di un'etichetta esclusiva, utile al suo riconoscimento in planimetrie appartenenti a piani diversi dello stesso edificio. In questo modo viene creato automaticamente il collegamento fra i piani e l'orientamento delle aree all'interno delle strutture.

Ancora, ogni arco possiede una proprietà che ne indica i limiti di percorribilità in relazione al fatto che il rispettivo percorso venga intrapreso da persone esterne oppure dal personale della struttura stessa. Se quindi un arco viene contrassegnato dall'accesso riservato, allora in automatico questi non sarà utilizzabile, e quindi non considerato nel calcolo del percorso migliore, dagli utenti che non lavorano in quella struttura.

Per quanto riguarda poi la necessità di poter costruire progetti di grandi dimensioni, e lavorare con più versioni degli stessi, il compromesso che è stato trovato, sempre per agevolare l'utente e non complicare eccessivamente il sistema, è stato quello di creare una dashboard separata con l'elenco ed i link a tutti i progetti personali, più varie indicazioni quali data di creazione e stato di avanzamento. Queste nuove funzionalità saranno mostrate graficamente nel prossimo paragrafo, che prsenterà struttura e navigazione reali della piattaforma.

Prima di concludere questo paragrafo sottolineiamo che, anche se i quattro principi fondamentali della progettazione sono stati esposti in relazione a specifici requisiti, gli stessi sono stati applicati, anche se in misura minore, anche nella costruzione di tutte le altre caratteristiche funzionali della piattaforma Stardust.

2.4 Stardust: struttura e navigazione

In questa sezione mostreremo la struttura reale della piattaforma Stardust e descriveremo, seguendo un flusso di navigazione standard, le varie funzionalità offerte e le scelte grafiche adottate nella costruzione del sistema.

2.4.1 Home Page

La piattaforma si apre con la pagina contenente le sezioni per effettuare il login e la registrazione a Stardust.

Questa pagina funge anche da vetrina, così da mostrare sin dal primo ingresso le funzionalità offerte e lo stile adottato. La figura seguente ne mostra alcune schermate:

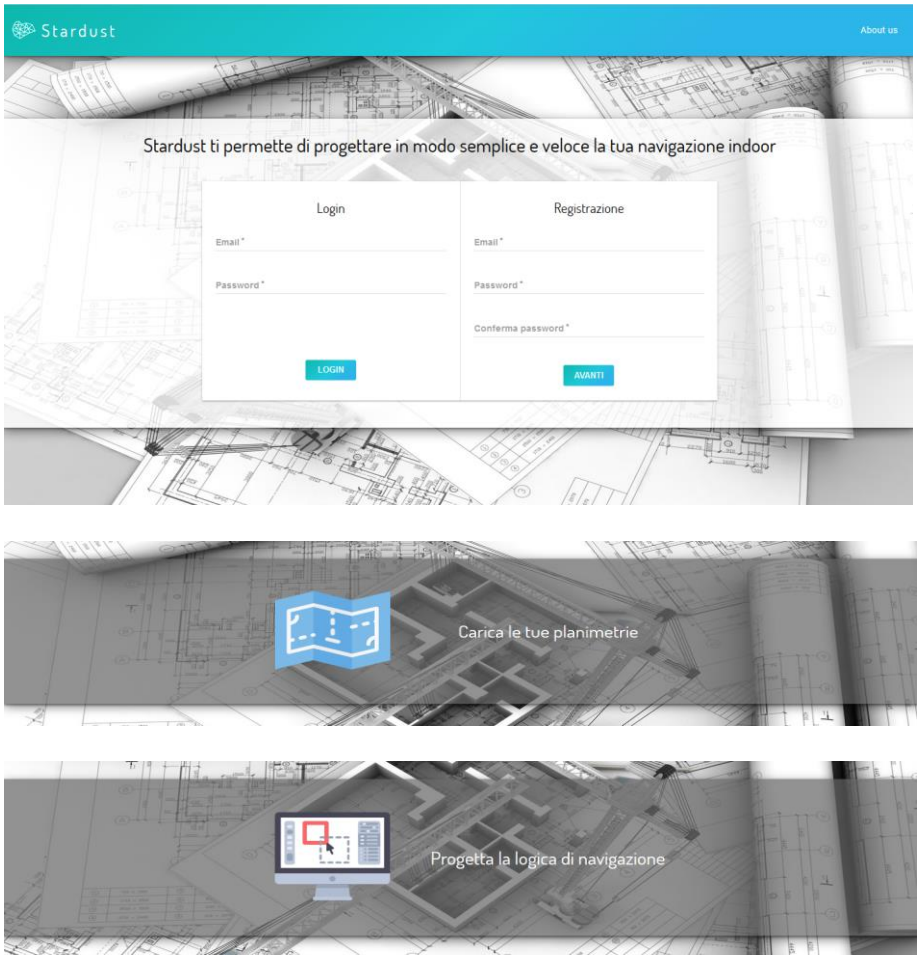


Figura 2.7: Home page di Stardust

Lo stile scelto per questa pagina si basa in parte sulle regole del material design ed in parte sui principi del design scheumorfico. Ricordiamo che il material design prevede la costruzione di elementi dalle forme regolari e dai colori uniformi. È un tipo di stile essenziale e moderno, ottimo per la realizzazione di applicazioni web.

Il design scheumorfico invece cerca di dare tridimensionalità agli elementi basandosi sulle regole reali di riflessione della luce, creando particolari effetti di luminosità ed ombre. Questa grafica viene utilizzata maggiormente per applicazioni desktop, quali tool per il disegno o la scrittura.

Le due figure seguenti mostrano rispettivamente un esempio di applicazione di material design ed uno di design scheumorfico per la costruzione di pulsanti:

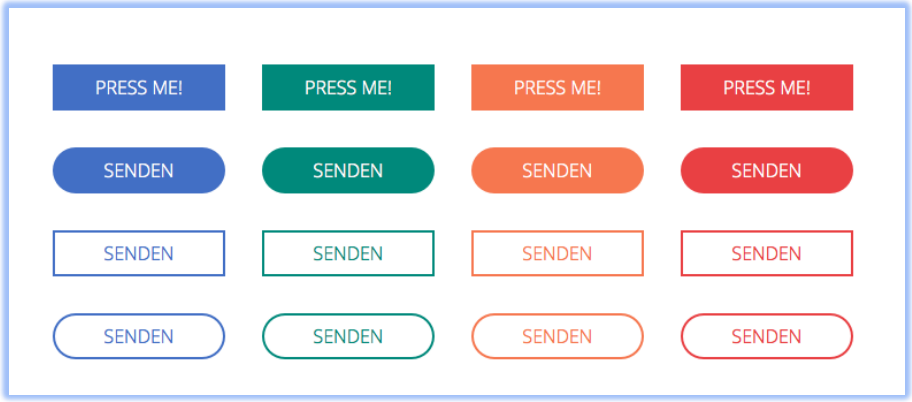


Figura 2.8: Esempio di applicazione material design.

Figura 2.9: Esempio di applicazione design scheumorfico.

La nostra scelta di creare un design ibrido è dettata dal fatto che Stardust è sia un'applicazione web che un editor di disegno. Ritornando alla home page, il background, che mostra alcuni fogli con disegni di planimetrie posti su di una scrivania, ha il compito di trasmettere all'utente l'anima della piattaforma sin dal primo ingresso, ed aumentare così l'immersività. Il titolo iniziale "Progetta la tua navigazione indoor" ed i box posti in basso, con alcune salienti descrizioni delle possibilità offerte da Stardust, permettono di far scoprire le caratteristiche della piattaforma anche ad un utente esterno.

2.4.2 Dashboard

Una volta effettuato il login l'utente viene indirizzato alla propria Dashboard, mostrata nella seguente figura:

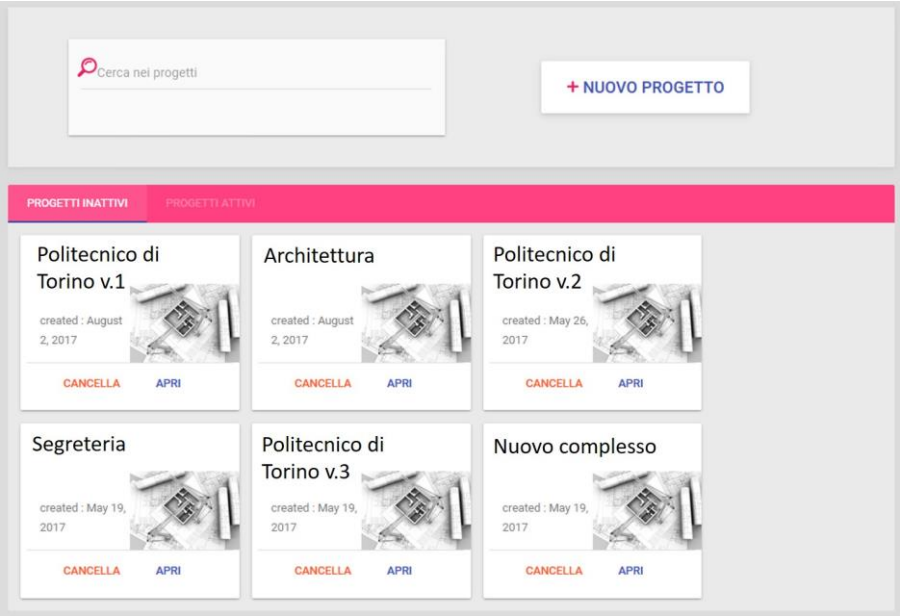


Figura 2.10: Dashboard di Stardust

Qui il design è stato mantenuto minimal poiché, essendo questa la pagina di accesso all'editor, è stata scelta la funzionalità a favore della ricchezza di contenuti. Mentre la Home, infatti, ha anche uno scopo promozionale, la Dashboard è invece una pagina di passaggio, una sorta di router che deve semplicemente indicare in modo chiaro che progetto aprire nell'editor. In particolare la schermata è stata suddivisa in due sole aree logiche.

Nella prima vi è un campo di input dedicato alla ricerca di un lavoro, ed un pulsante, tramite il quale è possibile creare un nuovo progetto. Questo si traduce nella navigazione verso un editor vuoto, ovvero sprovvisto di planimetrie, pronto per l'inserimento della prima immagine che farà da blueprint per un piano della struttura.

La seconda area è invece dedicata ai progetti già creati in precedenza. Questi sono organizzati secondo elenchi di card raggiungibili tramite la selezione di una delle due voci nella barra di navigazione. In particolare quest'ultima permette di spostarsi fra i progetti attivi e quelli non attivi. Con attivi si intendono tutti quei progetti che, all'atto del login, risultano in uso dal servizio di

navigazione indoor. Al contrario i progetti non attivi sono quelli ancora in fase di costruzione, non ancora resi disponibili all'esterno dell'editor. Qui l'utente può mantenere più versioni dello stesso progetto. Ognuno di questi poi, come prima anticipato, è rappresentato da una card che ne mostra il nome, la data di creazione ed il link all'editor per la modifica. È inoltre presente il pulsante *Cancella* per effettuarne l'eliminazione, previa conferma dell'azione desiderata, da parte dell'utente, da finestra di dialogo.

2.4.3 Editor

Poniamoci nella situazione in cui l'utente, una volta avuto l'accesso alla Dashboard personale, abbia selezionato *Nuovo Progetto*. A questo punto viene aperto l'editor online di Stardust, mostrato nella seguente figura:

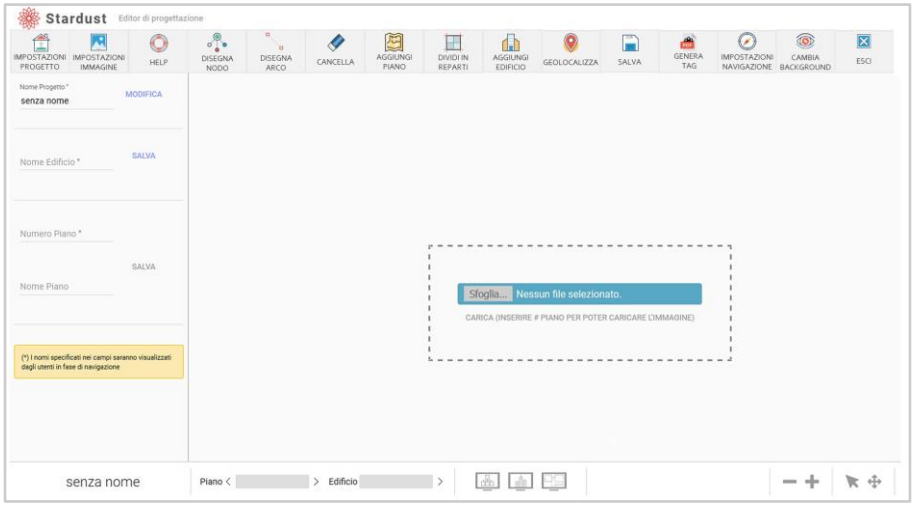


Figura 2.11: Editor di Stardust.

Come possiamo notare dall'immagine, e come già spiegato nel paragrafo 2.3 relativo alla progettazione, la schermata è composta da quattro aree:

1) Menù principale;
2) Menù secondario;
3) Foglio di lavoro;
4) Footer;

Proseguiamo quindi con l'analisi dettagliata di ognuna di queste parti. Nella zona superiore della schermata, al di sotto dell'header, troviamo il menù principale. Questi è composto da una pulsantiera, mostrata nella seguente figura:

Figura 2.12: Ingrandimento menù principale dell'editor di Stardust.

Ogni pulsante rappresenta una specifica funzionalità della piattaforma, descritta da un'icona e dal nome.

Le funzionalità semanticamente collegate sono state poste l'una accanto all'altra. Partendo da sinistra troviamo:

- **Impostazioni progetto:** Questa è la prima funzionalità che l'utente utilizzerà una volta inserita la prima planimetria. Da qui infatti è possibile immettere le informazioni base del progetto, modificabili nel corso della creazione dello stesso;

- **Impostazioni immagine:** Da questa sezione è possibile cambiare alcuni parametri relativi all'immagine della planimetria corrente. L'utente potrebbe infatti avere la necessità di sostituire l'immagine oppure di nasconderla così da mostrare i soli archi e nodi inseriti sulla stessa. Ricordiamo che le immagini caricate funzionano da blueprint, quindi da riferimento per la fase di disegno;

- **Help:** Questa funzionalità permette di accedere ad una pagina statica dalla quale reperire l'help online di Stardust, ovvero una sorta di corso alla progettazione suddiviso in sezioni. Queste ultime sono state pensate per coprire ognuna il procedimento richiesto per compiere un determinato task. Il contenuto delle sezioni è stato sviluppato a partire dalle domande più frequenti ottenute dagli utenti in fase di testing;

- **Disegna nodo:** Questa funzionalità permette di inserire un nodo in un punto della planimetria. Un nodo possiede una propria identità e può indicare un elemento architetturale ben preciso, una zona di interesse per il pubblico oppure un semplice punto

di snodo necessario per la corretta navigazione all'interno dell'edificio;

- **Disegna arco:** Questo pulsante, una volta attivato, permette di inserire archi di collegamento fra nodi. Ogni arco può avere origine e può concludersi solo ed esclusivamente in un nodo. Queste entità rappresentano i percorsi percorribili reali all'interno della struttura. Ognuno di questi cammini possiede determinate caratteristiche, utili alla scelta del percorso più breve per arrivare da un punto A ad un punto B all'interno dell'edificio. Quando due archi vengono disegnati l'uno sovrapposto all'altro, nel punto di intersezione viene automaticamente inserito un nuovo nodo dal sistema;

- **Cancella:** Questa funzionalità permette di eliminare un elemento inserito sul foglio di lavoro, sia questi un arco oppure un nodo. La cancellazione di un nodo prevede anche quella di tutti gli archi ad esso collegati. Al contrario la cancellazione di un arco non prevede l'eliminazione di altri elementi inseriti sulla planimetria;

- **Aggiungi piano:** Come il nome stesso suggerisce, l'aggiungi piano permette di inserire nel progetto una nuova planimetria. Quindi il foglio di lavoro mostrerà la sezione per l'inserimento di una nuova immagine, come visto in precedenza all'atto di apertura di un nuovo progetto. Il menù secondario, come vedremo fra poco, conterrà tutti i campi relativi alle informazioni del nuovo piano inserito così, da posizionarlo correttamente all'interno della struttura;

- **Dividi piano in reparti:** Questa funzionalità è utile nel caso in cui su di uno stesso piano vi siano sotto-aree semanticamente scorrelate. Pensiamo proprio a reparti diversi posti sullo stesso piano in una struttura ospedaliera. In casi come questi, una volta inseriti tutti i nodi e gli archi su di una planimetria, è possibile attivare questo strumento per tracciare aree diverse sull'immagine, ognuna dotata di una propria identità. Tutti gli elementi ed i percorsi appartenenti ad una determinata area saranno in automatico accomunati da una stessa proprietà;

- **Aggiungi edificio:** Questo più che uno strumento è l'ingresso verso un'estensione dell'editor pensata appositamente per la dislocazione su di una superficie dei vari edifici appartenenti ad

una stessa struttura. Stiamo quindi parlando di una nuova modalità di disegno. In pratica il foglio di lavoro si trasforma andando a mostrare, invece che una planimetria come background, uno sfondo bianco rappresentante l'area che comprende un complesso multi-edificio. Su questo nuovo foglio di lavoro sono presenti nuovi elementi, che sono appunto gli edifici, che seguono la logica del drag and drop. In pratica l'utente può spostare questi oggetti posizionandoli così come previsto nella realtà. Anche in questo caso entrano in gioco i nodi e gli archi, dove i primi rappresentano gli ingressi agli edifici, mentre i secondi i percorsi esterni utili al loro raggiungimento. È quindi una funzionalità di livello superiore rispetto alle altre la quale, data l'immissione di nuove regole e scenari di progettazione, sarà analizzata separatamente in seguito;

- **Geolocalizza:** La funzione di geolocalizzazione permette di accedere alle mappe di Google per impostare le coordinate geografiche delle strutture. Un progetto può essere utilizzato, ovvero attivato per la navigazione indoor, solo se sono state inserite le sue informazioni posizionali. Come vedremo questo vincolo è stato imposto per ragioni che si trovano a valle della progettazione, ovvero per il soddisfacimento di alcuni requisiti richiesti dall'applicazione mobile che si avvale delle mappe prodotte da Stardust per la navigazione;

- **Salva:** All'interno di un menù principale deve sempre essere previsto un pulsante per il salvataggio immediato del progetto in corso. In Stardust da qui è possibile anche accedere alle impostazioni per il salvataggio automatico;

- **Genera TAG:** Questa funzionalità permette di generare il file contenente tutti i TAG da stampare ed installare, ognuno dei quali corredato da opportune istruzioni per il posizionamento. Questo pulsante resta disattivato per la maggior parte del tempo di utilizzazione della piattaforma in quanto, per evitare errori di incoerenza dei dati, questa funzionalità può essere utilizzata solo se il sistema riconosce che tutte le informazioni obbligatorie sono state inserite e se non sono stati commessi errori grossolani in fase di disegno. Per fare qualche esempio pratico il pulsante *Genera TAG* è attivo solo se: la struttura è stata geolocalizzata; è stata inserita la data di partenza del servizio di navigazione indoor; tutti gli edifici sono stati

collegati da percorsi esterni; tutti i nodi di una planimetria sono connessi ad almeno un arco;

- **Impostazioni navigazione:** Questa sezione è dedicata all'inserimento dei dati utili al corretto funzionamento della navigazione indoor prevista a valle. In caso di aggiornamento di una parte di un progetto attivo, l'utente dovrà inserire qui i dati relativi all'entrata in vigore delle modifiche apportate. Se le impostazioni di navigazione non vengono inserite non possono essere generati i TAG;

- **Cambia background:** Questa funzionalità è puramente volta alla user experience, poiché permette di mutare al volo la palette di colori utilizzata per tutti gli elementi dell'editor. Poiché il completamento di un progetto richiede un certo tempo, è importante dare la possibilità all'utente di scegliere le tonalità che personalmente preferisce o ritiene affatichino meno la vista;

- **Esci:** Infine il tasto *Esci* permette di abbandonare il progetto in corso, chiudendo l'editor e reindirizzando l'utente alla sua Dashboard;

Passiamo ora all'analisi del menù secondario, ovvero il box posto alla sinistra del foglio di lavoro. Quest'area è dedicata all'impostazione delle varie funzionalità e variabili relative ad ogni singolo strumento del menù principale.

Vediamo quindi il suo contenuto in relazione al particolare pulsante selezionato dall'utente:

Impostazioni progetto: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Impostazioni progetto*:

Figura 2.13: contenuto menù secondario *Impostazioni progetto*.

Il menù secondario relativo a *Impostazioni progetto* contiene:

- Un campo obbligatorio per l'inserimento del nome della struttura alla quale il progetto in corso si riferisce;
- Un campo opzionale per l'inserimento del nome dell'edificio alla quale la planimetria appartiene. Il sistema genera comunque automaticamente un id per ogni nuovo edificio;
- Un campo obbligatorio per l'inserimento del numero del piano corrente. Tale valore, unito all'id dell'edificio, genera d'id del piano;
- Un campo opzionale per l'inserimento del nome del piano corrente;

Tutti i nomi inseriti, come indicato all'interno del box giallo posto al di sotto del form, saranno utilizzati per le indicazioni durante la navigazione indoor.

Impostazioni immagine: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Impostazioni immagine*:

Figura 2.14: contenuto menù secondario *Impostazioni immagine*.

Il menù secondario relativo a *Impostazioni immagine* contiene:

- un check box per la visualizzazione o la scomparsa della planimetria corrente, che fa da blueprint;
- un pulsante per l'eliminazione della planimetria corrente, con successiva apparizione del box per l'inserimento di una nuova immagine;

Help: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Help*:

Figura 2.15: contenuto menù secondario *Help*.

L'*Help* è suddiviso in sei sezioni. La prima ha lo scopo di fornire all'utente una panoramica generale dell'editor e del suo funzionamento, offrendo inoltre le informazioni necessarie relative alle impostazioni iniziali per l'avvio di un nuovo progetto.

La seconda sezione è dedicata ai nodi e agli archi, alle loro caratteristiche e proprietà.

La terza sezione mostra come inserire correttamente i piani all'interno di un edificio e gli step da seguire per suddividere un piano in reparti.

La quarta sezione è dedicata alla modalità di inserimento di un nuovo edificio. Ricordiamo che questa prevede un cambiamento nel paradigma di interazione con il foglio di lavoro. Tale sezione è quindi una sorta di tutorial che guida l'utente nello sviluppo in questo diverso settore progettuale.

Infine le ultime due sezioni sono dedicate all'installazione dei TAG all'interno degli edifici, con l'aggiunta di utili nozioni relative ai principi di funzionamento della navigazione indoor prevista a valle.

Disegna nodo: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Disegna nodo*:

Figura 2.16: contenuto menù secondario *Disegna nodo*.

Il menù secondario relativo a *Disegna nodo* contiene:

- La lista delle tipologie di nodo, che sono: generico; porta; scale; ascensore; TAG isolato; toilette; ristoro. Questi rappresentano punti di interesse all'interno di un edificio, fatta eccezione per i nodi generici ed i TAG isolati. I nodi generici sono punti di snodo dei percorsi, non associati quindi ad alcun elemento strutturale di interesse. I TAG isolati invece sono utili all'inserimento di destinazioni diverse da quelle già presenti nella lista, quali uffici, aule o ambulatori. È possibile cambiare la tipologia di un nodo tramite la selezione dello stesso sul foglio di lavoro e la scelta di una nuova voce dalla suddetta lista;
- Uno slider che permette di regolare la dimensione dei nodi, così da renderli coerenti rispetto alle dimensioni indicate dalla planimetria sottostante;
- Un campo per l'inserimento delle etichette da associare ai singoli nodi. Queste saranno utilizzate e visualizzate durante la navigazione indoor. Ad esempio, se un TAG isolato viene inserito per indicare l'ufficio di un determinato professore all'interno di un'Università, in questo campo va inserito il nome dell'insegnante.
- Un check box che permette di visualizzare o meno le etichette dei vari nodi sul foglio di lavoro, in una piccola area colorata posta al fianco degli stessi.

Disegna arco: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Disegna arco*:

Figura 2.17: contenuto menù secondario *Disegna arco*.

Il menù secondario relativo a *Disegna arco* contiene:

- Un check box che permette di impostare al volo l'arco come percorso riservato. Ricordiamo che questi sono i cammini percorribili dal solo personale di un edificio.
- Uno slider che permette di regolare lo spessore degli archi, così da renderli coerenti rispetto alle dimensioni indicate dalla planimetria sottostante;

Aggiungi piano: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Aggiungi piano*:

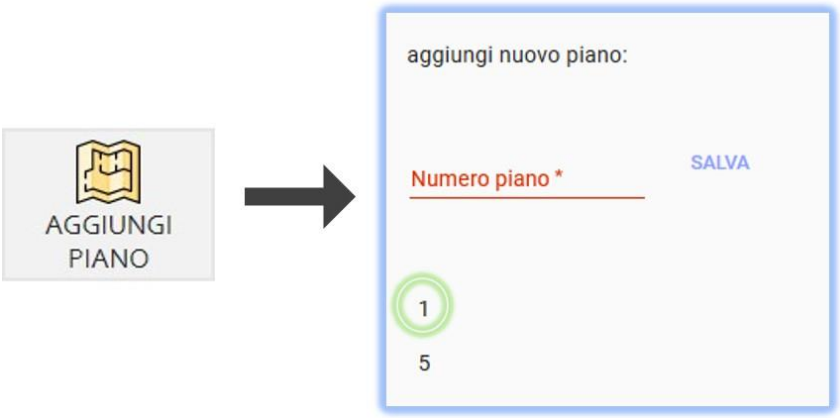


Figura 2.18: contenuto menù secondario *Aggiungi piano.*

Il menù secondario relativo a *Aggiungi piano* contiene un campo per l'inserimento del numero dello stesso.

È inoltre presente una lista di tutti i piani già inseriti all'interno dello stesso edificio, con specifica indicazione di quello correntemente visualizzato sul foglio di lavoro.

Ricordiamo che l'immagine relativa alla planimetria del nuovo piano verrà inserita accedendo all'apposito link posto sul canvas, e che un eventuale etichetta può essere aggiunta nel menù secondario di *Impostazioni progetto*.

Dividi piano in reparti: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Dividi piano in reparti*:

Figura 2.19: contenuto menù secondario *Dividi in reparti*.

Nel menù secondario relativo a *Dividi piano in reparti* è presente il tool che permette di tracciare un'area rettangolare sul foglio di lavoro ed un campo per l'inserimento del nome del reparto. Tutti i nodi già inseriti, o che verranno inseriti, all'interno di questa area saranno uniti fra di loro come facenti parte di un unico reparto. Quest'ultimo da un punto di vista logico costituisce una proprietà aggiuntiva per i nodi, mentre da un punto di vista pratico determina l'introduzione di una nuova destinazione per la navigazione. L'esempio calzante è quello relativo alla presenza di più reparti posti sullo stesso piano in una struttura ospedaliera.

Aggiungi edificio: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Aggiungi edificio*:

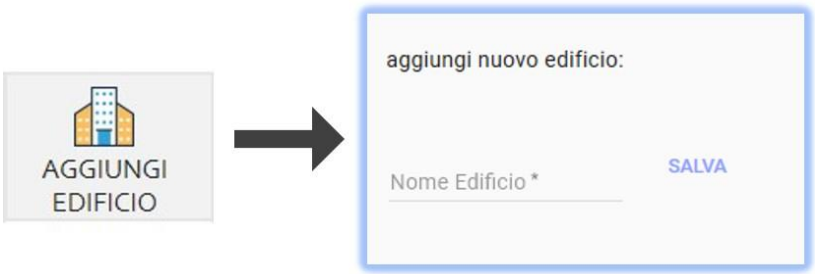


Figura 2.20: contenuto menù secondario *Aggiungi edificio*.

Il menù secondario relativo ad *Aggiungi edificio* contiene un solo campo per l'inserimento del nome dello stesso.

Come vedremo gran parte delle funzionalità e dei cambiamenti introdotti da questo strumento saranno presenti sul foglio di lavoro.

Geolocalizza: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Geolocalizza*:

Figura 2.21: contenuto menù secondario *Geolocalizza.*

Il menù secondario relativo a *Geolocalizza* contiene le note mappe Google per la navigazione e l'inserimento di punti di interesse. In questo caso le coordinate intercettate dall'utente saranno associate al progetto in corso.

Salva: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Salva*:

Figura 2.22: contenuto menù secondario *Salva*

Da questa funzionalità quindi si accede anche al menù per le impostazioni relative al salvataggio automatico. Poiché questo è un editor online, abbiamo ritenuto necessaria la presenza di più slot temporali di scelta, con una granularità che va dal salvataggio automatico ogni 5 minuti fino ad ogni ora.

Impostazioni navigazione: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Impostazioni navigazione*:

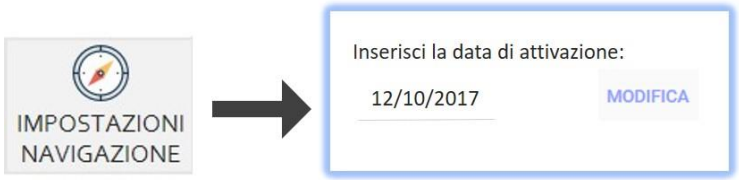


Figura 2.23: contenuto menù secondario *Impostazioni navigazione*.

Il menù secondario relativo a *Impostazioni navigazione* contiene un unico capo per l'inserimento della data a partire dalla quale i dati correnti saranno forniti al servizio di navigazione indoor. Questa informazione è utile non solo in fase di avvio del servizio, ma anche in caso di aggiornamento dello stesso, quando è previsto un aggiornamento oppure uno spostamento dei TAG già installati. In questo modo possono essere evitati eventuali disallineamenti informativi fra il progetto e la sua applicazione.

Cambia background: L'immagine seguente mostra il contenuto del menù secondario relativo alla funzionalità *Cambia background*:

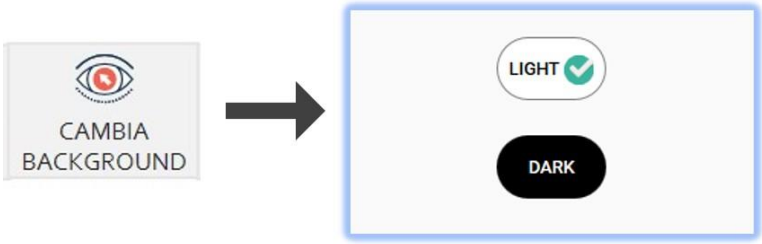


Figura 2.24: contenuto menù secondario *Cambia background*

Nel menù secondario relativo a *Cambia background* sono presenti i pulsanti per l'azionamento di uno specifico stile per l'editor. Attualmente Stardust prevede due diverse palette di colori: light, per tonalità che vanno dal bianco al grigio chiaro; dark, per tonalità che vanno dal grigio scuro al nero.

Passiamo adesso al contenuto del foglio di lavoro. Quando viene creato un nuovo progetto, oppure all'aggiunta di un nuovo piano, il canvas è pronto per l'inserimento di una nuova immagine:

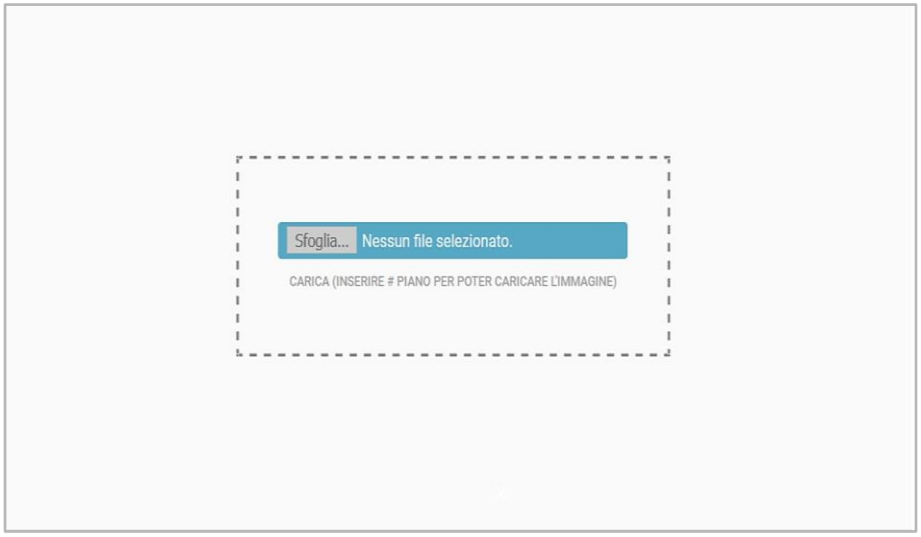


Figura 2.25: contenuto foglio di lavoro all'avvio dell'editor.

L'immagine può essere inserita o tramite accesso al disco personale, con successiva selezione del file, oppure attraverso il trascinamento del file immagine sul foglio di lavoro. È previsto il supporto a tutti i formati immagine.

Una volta che il progetto è stato avviato, il foglio di lavoro può assumere quattro diverse modalità:

1. modalità piano;
2. modalità edificio;
3. modalità grafo;
4. modalità collegamento piani;

La prima di queste è quella principale, ed è mostrata nella seguente figura:

Figura 2.26: contenuto foglio di lavoro in *modalità piano*.

In questo caso l'utente può inserire nodi ed archi su una planimetria posta in background.

Poiché una struttura può essere composta anche da più edifici, è necessario mantenere la continuità del servizio di navigazione indoor nel passaggio fra gli stessi. Per tale ragione si è resa necessaria l'introduzione di una nuova modalità di progettazione che è quella relativa al collegamento degli edifici. In questo caso il foglio di lavoro mostrerà una sorta di superficie sulla quale disporre gli edifici, che saranno trattati come entità trascinabili e rilasciabili, secondo la loro reale ubicazione.

La seguente figura ne mostra un esempio:

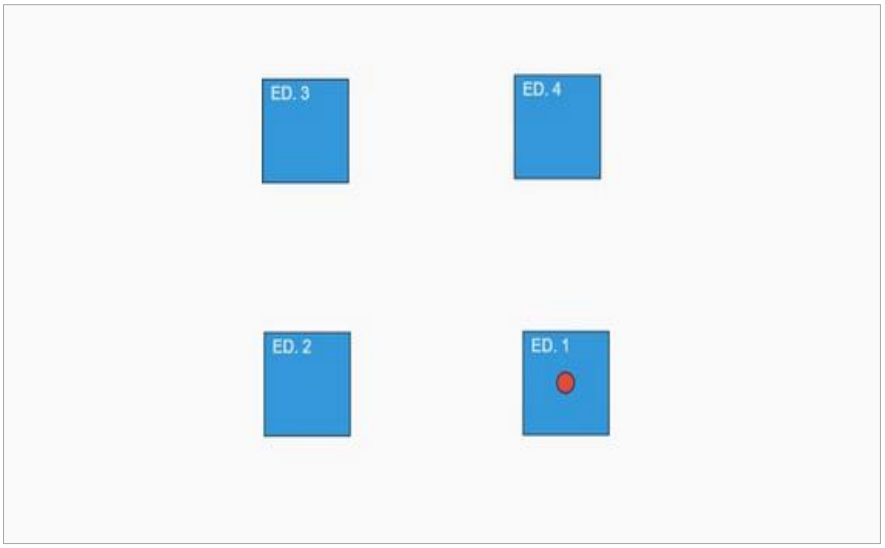


Figura 2.27: contenuto foglio di lavoro in *modalità edificio*.

Anche in questo caso è necessario inserire nodi ed archi. Tuttavia, mentre questi ultimi continuano a mantenere le stesse funzionalità viste in modalità piano, i nodi in questo caso hanno invece la restrizione di poter assumere soltanto la funzione di porte di ingresso. In pratica ogni edificio deve essere trattato come una black box: potrebbe contenere qualunque cosa ma questo non deve interessare l'utente, che deve solo etichettare l'ingresso all'edificio con la stessa label utilizzata per la porta d'entrata del piano di ingresso della struttura in questione. A questo punto sarà il sistema a fare tutto il resto.

Abbiamo poi la modalità grafo, accessibile solo dal footer, che permette di visualizzare l'albero architetturale dell'intero progetto. La seguente figura ne mostra un esempio:

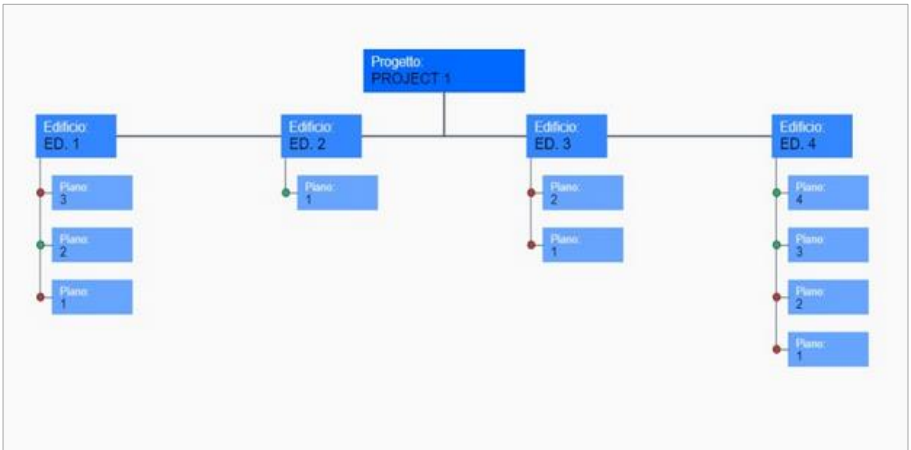


Figura 2.28: contenuto foglio di lavoro in *modalità grafo*.

Possiamo in questo caso notare che il nodo radice è costituito dal progetto stesso. I nodi figli sono invece gli edifici, a loro volta genitori dei nodi piani. Questa modalità è in sola lettura, ovvero l'utente non può apportare nessuna modifica alla gerarchia, che ha quindi il solo scopo di mostrare in modo immediato l'organizzazione del progetto per verificarne la correttezza.

L'ultima modalità prevista per il foglio di lavoro è quella relativa al collegamento fra piani all'interno di un edificio. Abbiamo precedentemente spiegato che questo può essere ottenuto tramite l'inserimento di uno stesso identificativo alle entità architetturali di collegamento, quali scale o ascensori. In questo modo sarà il sistema a creare automaticamente un percorso di congiunzione fra i grafi relativi a due piani adiacenti.

Tuttavia spesso i collegamenti strutturali non corrispondono a quelli logici per la navigazione, oppure particolari architetture possono prevedere unioni di planimetrie in modo non naturale. Alla luce di queste considerazioni è stata quindi inserita una sezione avanzata per il collegamento fra i piani.

In questo nuovo scenario il foglio di lavoro si presenta nel seguente modo:

Figura 2.29: contenuto foglio di lavoro in *modalità collegamento piani*.

Il canvas viene quindi suddiviso in due aree distinte, preposte al contenimento di due diverse planimetrie da collegare. All'interno di ogni area è prevista una sezione che rimanda al menù secondario. Questi, una volta attivata la modalità di collegamento fra piani, assumerà il contenuto mostrato nella seguente figura:

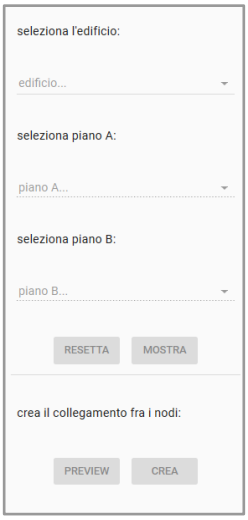


Figura 2.30: contenuto menù secondario in *modalità collegamento piani*.

I campi di input previsti, partendo dall'alto, sono relativi alla specifica dell'edificio e dei due piani da collegare.

Terminiamo questa parte mostrando l'ultima sezione prevista nell'editor: il footer. Qui possiamo trovare alcuni semplici strumenti, che pur non prevedendo particolari opzioni di funzionamento, sono comunque di grande utilità. L'immagine seguente mostra il contenuto del footer:

Figura 2.31: contenuto footer dell'editor.

Partendo da sinistra, subito dopo il nome del progetto, troviamo:

- **Piano:** Questo campo, oltre a mostrare il piano corrente sul quale l'utente sta lavorando, permette anche di spostarsi in modo immediato fra i vari piani inseriti nell'edificio corrente;
- **Edificio:** Così come la precedente funzionalità, questo campo, oltre a mostrare l'edificio corrente sul quale l'utente sta lavorando, permette anche di spostarsi in modo immediato fra i vari edifici previsti all'interno dell'intera struttura;
- **Vai modalità grafo:** Questo rappresenta l'unico punto di accesso alla modalità di visualizzazione grafo;
- **Vai modalità edificio:** Permette di accedere alla modalità di progettazione degli edifici;
- **Vai modalità piano:** Permette di accedere alla modalità di collegamento fra piani;
- **Nascondi/Mostra:** Permette di nascondere/mostrare l'immagine della planimetria in background, nella modalità di visualizzazione piano;
- **Cursore:** Da qui è possibile passare dalla modalità drag, utile nel caso di planimetrie di grandi dimensioni che superano la grandezza standard del foglio di lavoro, a quella di cursore per utilizzare tutti i tool della piattaforma;
- **Zoom:** Per effettuare le note operazioni di zoom-in e zoom-out sul contenuto del foglio di lavoro;

2.5 Conclusioni

Nel corso del presente capitolo sono state esposte tutte le fasi che hanno portato alla realizzazione delle funzionalità e del flusso logico della piattaforma Stardust.

Partendo dall'analisi dei requisiti abbiamo, sulla base delle interviste fatte agli utenti e sullo studio dei competitor, ricavato le caratteristiche necessarie al perseguimento degli obiettivi iniziali. A questo punto sono stati applicati specifici principi fondamentali per la progettazione, che ci hanno portati all'evoluzione dall'insieme di requisiti teorici alla costruzione del sistema reale.

Alla luce di tutti gli aspetti emersi nel corso della presente sezione possiamo giungere ad una serie di importanti considerazioni. La trasformazione del servizio di navigazione indoor a prodotto, apre un nuovo scenario nell'ambito socio-tecnico nel mondo della navigazione digitale. È possibile quindi creare tool pensati appositamente per il compimento di specifici task, in modo tale da rendere autonomi tutti coloro i quali desiderano agevolare il pubblico negli spostamenti all'interno di edifici, nei quali l'orientamento risulta essere un aspetto critico.

Inoltre, l'interazione naturale con un editor pensato, anche nel caso di progetti più complessi, così da semplificare al massimo il lavoro dell'utilizzatore, minimizzando le competenze necessarie, elimina buona parte degli ostacoli iniziali che possono impedirne l'adozione. Tutto ciò è proprio quello che Stardust si propone di fare. Le energie dell'utente possono essere concentrate solo sugli aspetti di suo interesse: egli potrà costruire da zero e mantenere nel tempo i suoi progetti.

Un ulteriore vantaggio è quello economico. Ricordiamo infatti che Stardust si basa sulla generazione di TAG, da installare opportunamente negli edifici. Oltre al fatto che questi marker non richiedono l'intervento di personale esterno né in fase di progettazione, né in quelle di installazione e mantenimento, c'è da aggiungere che i TAG non sono dispositivi elettronici costosi e soggetti a malfunzionamenti.

Concludiamo dicendo che Stardust per molti aspetti rappresenta un'innovazione nel mondo della navigazione indoor ed anche, a nostro parere, un ottimo punto di partenza per rendere questa tecnologia nascente alla portata di tutti.

Capitolo 3

L’architettura

Il presente capitolo ha lo scopo di fornire una visione ad alto livello dell’architettura di Stardust. A partire dai requisiti tecnici richiesti, e considerando i limiti imposti dai vari sistemi considerati, collegheremo le varie componenti tecnologiche scelte nella costruzione di un’architettura il più possibile ottimizzata. Precisiamo che la parte teorica relativa alle varie tecnologie introdotte nelle prossime sezioni, sarà trattata nel dettaglio a partire dalla parte II del presente lavoro di tesi. L’obiettivo che ci prefiggiamo in questa parte è quello di rendere note al lettore le ragioni alla base della scelta di determinate tecnologie rispetto ad altre, e all’ottenimento di una conoscenza sufficientemente approfondita dello scheletro della piattaforma Stardust.

3.1 Requisiti tecnologici

Una volta determinate le varie funzionalità che il sistema deve essere in grado di svolgere, si è passati allo studio delle tecnologie esistenti per il soddisfacimento dei requisiti ottenuti durante la prima fase della progettazione di Stardust.

A questo punto lo stadio più complesso è stato quello di determinare, dato un insieme di tecnologie appartenenti alla stessa area applicativa, quale potesse essere la migliore scelta per l’implementazione della nostra piattaforma. Per prendere questa delicata decisione abbiamo focalizzato la nostra attenzione su alcune caratteristiche fondamentali richieste dalle tecnologie da adottare:

Multipiattaforma: Con multipiattaforma si intende il requisito di portabilità del prodotto su più piattaforme. Come tutti sappiamo i dispositivi ed i browser tramite i quali gli utenti accedono ad Internet sono numerosi. Un sistema deve sempre essere in grado di funzionare correttamente, al di là del supporto che lo ospita. Non sarebbe infatti accettabile porre dei vincoli di usabilità di un prodotto, poiché è quest'ultimo che deve adattarsi all'utente, non il contrario. Le scelte tecnologiche quindi dipendono fortemente dalla soddisfazione di questo requisito fondamentale. È altresì vero che l'editor di Stardust, per sua costruzione, difficilmente può essere adattato a dispositivi mobile. Il nostro intento di portabilità della piattaforma è quindi da intendersi in relazione ai diversi sistemi operativi e browser presenti ad oggi sui dispositivi desktop.

Performance: Da questo requisito dipendono fortemente le scelte tecnologiche adottate per la costruzione dell'editor di Stardust. Questi infatti deve essere in grado di dare risposte immediate ad ogni input dell'utente, un'impresa tutt'altro che banale considerando il fatto che tutto il processo avviene online.

User experience: Poiché Stardust è una piattaforma online, che contiene al suo interno un editor di lavoro, è necessario prestare grande attenzione a quelle che sono le scelte di design nelle varie parti che compongono il sistema. In particolare risulta indispensabile adottare le regole tecniche suggerite in letteratura per la costruzione di pagine web, come il material design, e quelle invece da adottare per i tool di lavoro, come lo scheumorfismo. La doppia convivenza di due schemi progettuali diametralmente opposti deve essere tale da aiutare l'utente nella comprensione delle parti, senza creare confusione o incoerenza di stile. La necessità di dover adottare due diversi insiemi di regole per la costruzione grafica dell'intera piattaforma condiziona non poco la scelta delle tecnologie da adottare;

Flessibilità e modularità: Il sistema deve essere in grado, all'occorrenza, di poter essere aggiornato e/o modificato in alcune delle sue parti, nel minor tempo possibile e senza andare ad intaccare altre funzionalità. Questa eventualità richiede una scelta oculata delle tecnologie da utilizzare, capaci di poter operare cambiamenti con il minimo sforzo, senza la necessità di re-ingegnerizzare l'intero sistema, ma dando la possibilità di agire solo su alcune parti dello stesso come se fossero isolate dalle altre;

Open Source: La scelta di software Open Source è stata, più che un requisito, la prima regola applicata per ottenere un importante scrematura dalla vastità di soluzioni prospettate all'inizio di questa fase;

Butterfly effect: Con effetto farfalla vogliamo intendere tutti quegli eventi indesiderati che accadono quando, data una specifica scelta, questa va a ripercuotersi negativamente su altre già presenti nel sistema. Questo si traduce nella necessità di scegliere le tecnologie non solo rispetto alle proprie caratteristiche, perse singolarmente, ma anche rispetto ai possibili effetti che la sua integrazione può sollevare nell'intero sistema;

3.2 Panoramica dell'architettura

In questa sezione costruiremo passo dopo passo l'architettura di Stardust. Partendo dai tre macro-blocchi che costituiscono l'applicazione, ovvero il database, il back-end ed il front-end, andremo ad analizzare le tecnologie adottate in ognuna di queste parti per poi costruire lo schema completo dell'architettura.

Anche durante l'analisi di ogni singolo componente, cercheremo sempre di non perdere mai di vista la visione d'insieme dell'intero scheletro architetturale, così da avere sempre chiara la logica di interazione che avviene nell'intero sistema. Partiamo quindi da uno schema a blocchi di base:

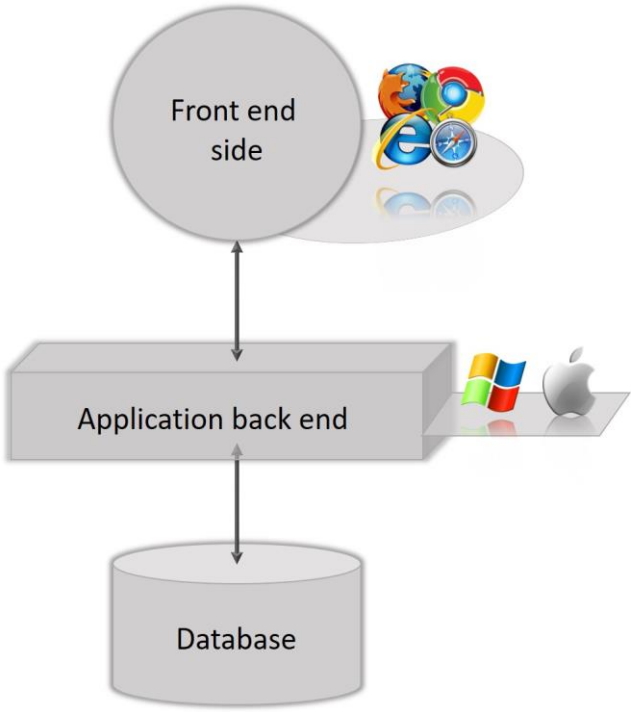


Figura 3.1: Architettura di base della piattaforma Stardust.

Procediamo quindi con la scomposizione dei tre blocchi principali mostrati nello schema precedente. Per ognuno di questi verranno considerati i requisiti tecnologici richiesti e le relative scelte adottate. Inoltre, per rendere più comprensibile questa sezione, dallo stampo puramente tecnico, abbiamo deciso di adottare un paradigma per l'esposizione dei contenuti ben noto in letteratura e a nostro parere, dati i propositi di Stardust, anche piuttosto azzeccato. Questo paradigma infatti prevede di paragonare la costruzione di un'applicazione web a quella di un edificio.

Per prima cosa quindi dobbiamo pensare alle fondamenta. Queste, per loro definizione, devono essere abbastanza robuste da poter reggere il peso dell'intera struttura. Inoltre è da qui che partono le colonne portanti, che vanno a dipartirsi lungo tutti i piani dell'edificio. In questo caso le fondamenta sono rappresentate dai dati.

Le colonne portanti rappresentano la presenza di questi dati lungo l'intera architettura di sistema:

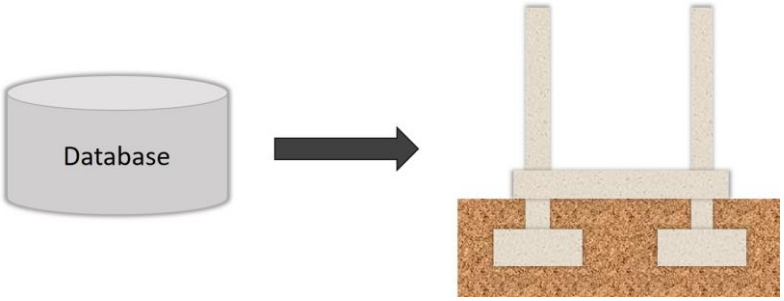


Figura 3.2: Primo step costruzione di un'architettura.

Per la scelta relativa alle tecnologie da adottare nell'ambito dei dati abbiamo considerato sia la natura di Stardust in quanto applicazione web, che quella in quanto editor. In particolare alcuni dati sono relativi alle sole sessioni web, altri invece riguardano gli elementi e le entità inserite nel tool di disegno. È stata quindi scelta la coesistenza di due diverse tecnologie per la costruzione e la manipolazione dei dati.

Considerando che ogni progetto costruito nell'editor di Stardust è composto da nodi ed archi, e che ognuno di questi è corredato da un numero variabile di proprietà e regole, la nostra decisione è caduta immediatamente sull'adozione di Neo4j, un sistema per la gestione di database dove i dati vengono rappresentati tramite grafi, quindi con nodi, archi di collegamento e proprietà associabili a queste due entità. Il fatto che i dati vengono rappresentati così come si presentano nella realtà è stato per noi un grande vantaggio durante la fase di sviluppo del sistema, e non solo.

Neo4j prevede già al suo interno l'implementazione dell'algoritmo di Dijkstra per il calcolo del percorso minimo fra due nodi. Questo è proprio ciò che serve all'applicazione mobile per indicare all'utente finale il cammino da intraprendere all'interno di un edificio.

Senza entrare nel dettaglio di come funziona Neo4j, cosa che faremo invece nel corso del capitolo 4, elenchiamo le caratteristiche di questo software, che ne fanno la scelta perfetta per i dati prodotti dall'editor:

- È open source;
- Prevede una strutturazione dei dati a grafo, quindi i dati sono ripartiti fra nodi ed archi;
- Rispetto ad altri database transazionali, ma basati su tabelle, la struttura a grafo permette dei tempi di attraversamento della stessa molto minori;

- Al suo interno prevede già l'implementazione dell'algoritmo di Dijkstra per il calcolo del percorso minimo dati due nodi di un grafo;

Per quanto riguarda invece la parte relativa ai dati delle sessioni web la nostra scelta si è orientata, vista la natura non transazionale dell'applicazione, su tecnologie NoSQL.

Fra i tanti DBMS non relazionali ad oggi esistenti abbiamo prediletto l'adozione di MongoDB, principalmente per la sua elevata affidabilità e per il grande successo che sta ad oggi riscuotendo in ambito aziendale.

Una volta scelte le tecnologie siamo poi passati alla modellazione dei dati. Durante questo processo l'aspetto più delicato è stato quello relativo all'identificazione e all'organizzazione di tutte le regole, le proprietà ed i vincoli degli elementi inseriti sul foglio di lavoro dell'editor di Stardust. I nodi e gli archi su di una planimetria, infatti, oltre ad essere adeguatamente arricchiti da tutti i dati necessari ad una corretta navigazione, necessitano anche di una relazione con i punti d'accesso di altre planimetrie, dello stesso edificio o di altri edifici della stessa struttura, per garantire la continuità di percorrenza fra gli stabili appartenenti allo stesso complesso.

Una volta gettate le basi sulla rappresentazione dei dati, il passo successivo è stato quello relativo alla scelta delle tecnologie da adottare per la parte del back end applicativo.

Continuando il nostro paragone fra la costruzione di un'applicazione web con quella di un edificio, possiamo dire che il back end corrisponde al piano terra della struttura, dove è quindi posta la porta d'ingresso alla stessa:

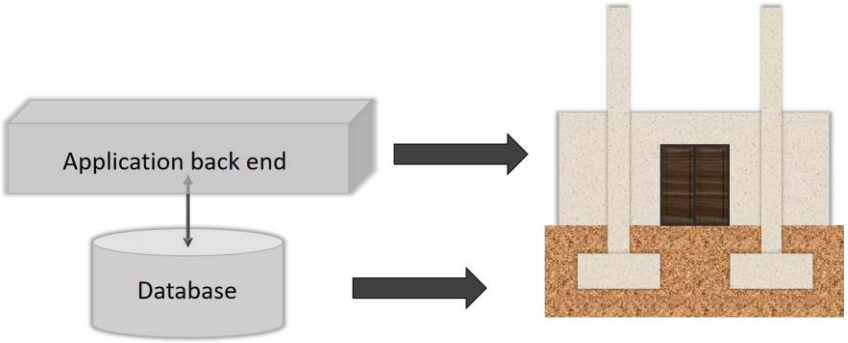


Figura 3.3: secondo step costruzione architettura.

Nei contesti di Single Page Application, concetto che avremo modo di definire in seguito, vi è una netta separazione fra front end e back end. Quest'ultimo ha il compito di gestire la business logic, elaborando i dati

forniti dal client per restituire i risultati richiesti. I concetti chiave alla base della nostra scelta per la tecnologia di back end sono stati performance e conoscenza pregressa del linguaggio. Questo ci ha portati verso l'adozione di NodeJs, un runtime Javascript. NodeJS è un framework leggero ed efficiente, che usa un modello I/O non bloccante e ad eventi. Inoltre l'ecosistema dei pacchetti di Node.js, npm, è il più grande agglomerato di librerie open source al mondo [4]. Passiamo ora alla costruzione del secondo ed ultimo piano del nostro edificio, che rappresenta il client side o front end dell'architettura:

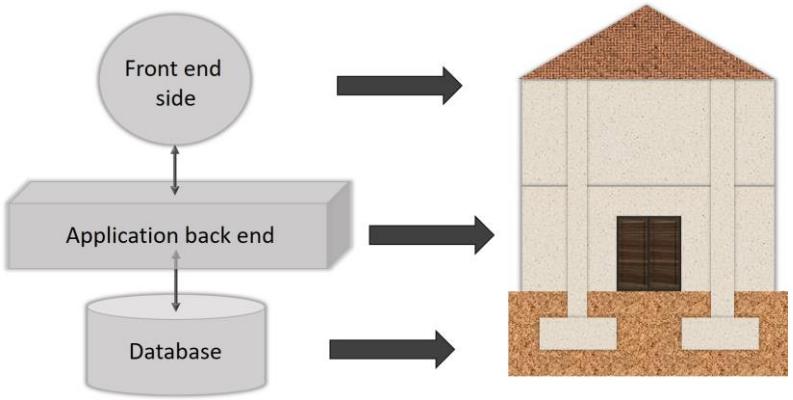


Figura 3.4: ultimo step costruzione architettura.

Sulla base dello studio delle architetture ad oggi più utilizzate per lo sviluppo di applicazioni web, abbiamo deciso di realizzare una *Single Page Application*. Questa prevede che il carico computazionale venga in parte spostato sul client. Nelle SPA in fase di inizializzazione i browser caricano tutto il codice necessario ad una migliore navigazione fra le pagine del sito. Data questa caratteristica, e il desiderio di riutilizzare le conoscenze relative a JavaScript, già presente nella parte di back end, la tecnologia scelta per il front end è stata AngularJS, un framework open source, sviluppato da Google, basato sull'architettura MVC (Model View Controller).

Gli elementi principali di AngularJs sono: i service, che si occupano della comunicazione con il back end; i controller e le view, dove i primi si occupano di gestire il modello così che questi si adatti alle esigenze dettate dai secondi; il router, che permette la navigazione fra le view dell'applicazione. La combinazione di AngularJs, HTML5[7], Angular

[7] **HTML5**: ultima versione dell'HTML, il linguaggio di markup alla base della strutturazione di pagine web.

Material[8] e CSS3[9] ci hanno permesso di costruire delle interfacce grafiche multipiattaforma strutturate per ottimizzare al massimo la user experience.

Per quanto riguarda invece la costruzione dell'editor, questi ha richiesto la presenza di una tecnologia ottimizzata per il disegno su canvas[10]. La nostra scelta è ricaduta su P5.js, una libreria JavaScript, implementata ed ottimizzata proprio per il disegno e le animazioni su browser.

Fortemente basata sul linguaggio Processing, p5 ha l'obiettivo di rispecchiare la semplicità e la potenza di quest'ultimo, però su piattaforme web. Questa libreria è stata wrappata all'interno dell'editor di Stardust, nell'area dedicata al foglio di lavoro. L'architettura, vista nel suo insieme, segue il paradigma MVC: MongoDB e Neo4j rappresentano il Model; AngularJs e P5.js rappresentano la View; Node.js rappresenta il Controller.

Prima di concludere la trattazione dell'architettura di Stardust, aggiungiamo gli ultimi elementi all'edificio costruito un passo alla volta nel corso della seguente sezione.

Poiché è stata adottata un'architettura RESTful[11], le scale interne che conducono dal primo al secondo piano sono delle API ReST. Ci sono poi gli endpoint, le porte di accesso alla nostra struttura. In particolare ne abbiamo due: la prima è quella relativa all'ottenimento dell'index.html da parte del back end, con tutti i riferimenti utili alla costruzione della Single Page Application; la seconda è relativa all'autenticazione, fornisce cioè le chiavi di accesso a tutte le pagine della piattaforma web.

[8] **Angular Material**: libreria per la creazione di interfacce grafiche, creata appositamente per client realizzati tramite AngularJS.
[9] **CSS3**: ultima versione del CSS, il linguaggio per la formattazione di pagine web.
[10] **Canvas**: estensione dell'HTML standard, che permette il rendering dinamico di immagini bitmap gestibili attraverso un linguaggio di scripting.
[11] **RESTful**: il REpresentational State Transfer è un tipo di architettura software per i sistemi di ipertesto distribuiti come il World Wide Web.

Vediamo quindi il risultato finale ottenuto:

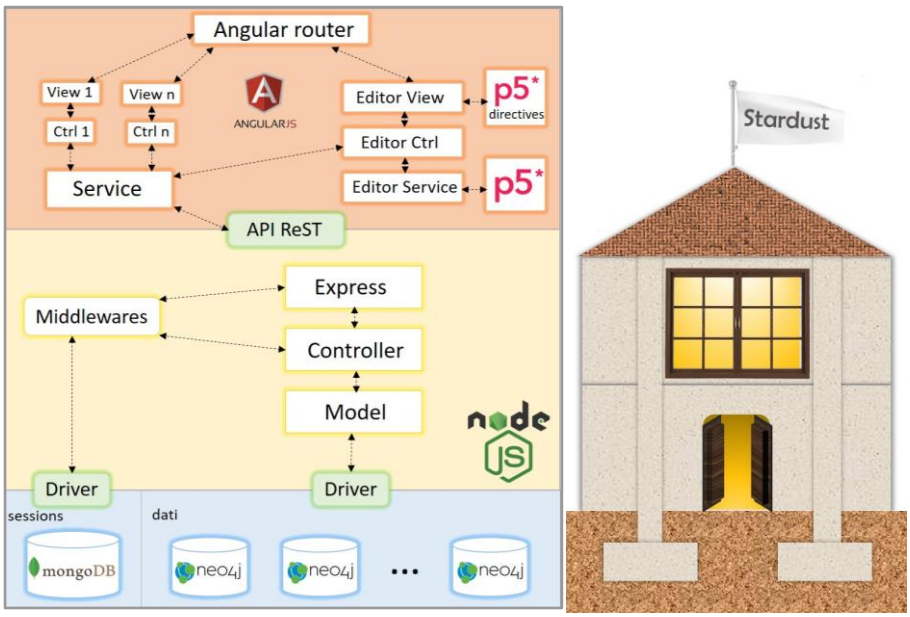


Figura 3.5: architettura completa della piattaforma Stardust

3.3 Conclusioni

Nel corso del presente capitolo abbiamo presentato le varie tecnologie adottate per la realizzazione dell'architettura di Stardust. Queste scelte, come abbiamo visto, sono state una diretta conseguenza di specifici requisiti tecnologici che la piattaforma avrebbe dovuto soddisfare.

Durante la trattazione abbiamo utilizzato degli schemi esemplificativi, che hanno aiutato il lettore nella costruzione del sistema tramite un approccio bottom-up.

Nelle varie sezioni sono state introdotte tutte le tecnologie ed i paradigmi utilizzati. La loro descrizione è stata volontariamente mantenuta ad un livello di dettaglio basso in quanto il loro studio approfondito è stato posto nella parte II del presente lavoro di tesi, che ha inizio dal prossimo capitolo. In particolare i capitoli 4, 5 e 6 sono dedicati rispettivamente allo studio teorico delle tecnologie adottate nei tre livelli logici di cui l'architettura di Stardust si compone:

- Dati (capitolo 4);
- Back end applicativo (capitolo 5);

- Front end (capitolo 6);

Qui si chiude quindi la parte I, dedicata a tutti gli aspetti progettuali che hanno avuto luogo prima della fase di implementazione vera e propria.

Parte II

Le tecnologie utilizzate

Capitolo 4

Gestione dei dati

In questo capitolo analizzeremo i sistemi adottati per la gestione dei dati di Stardust. Il capitolo è suddiviso in due parti principali. La prima è quella relativa a MongoDB, ovvero il DBMS utilizzato per la memorizzazione e la gestione delle sessioni di rete. La seconda invece si occupa di Neo4j, il database dedicato al cuore della piattaforma, ovvero ai dati relativi all'editor di progettazione. Ognuna di queste tecnologie sarà argomentata rispetto alla teoria alla base del funzionamento ed alle caratteristiche che ne hanno determinato la scelta migliore per il nostro sistema. In ogni sezione sarà inoltre fornita la panoramica degli scenari nei quali MongoDB e Neo4j vengono adottati e alle soluzioni alternative esistenti, così da contestualizzare i vantaggi e gli svantaggi di ognuna delle due tecnologie considerate.

4.1 MongoDB

Sui libri di teoria, o sui siti web dedicati a MongoDB, questi viene presentato come un sistema per la gestione di basi di dati, non relazionale, orientato ai documenti. Questa descrizione, seppur breve, riassume perfettamente i concetti chiave e le novità introdotte dai DBMS come MongoDB. Per comprendere le ragioni di questa affermazione dobbiamo fare un piccolo salto indietro nel tempo.

Negli ultimi decenni infatti lo standard de facto per lo storage e la gestione dei dati è stato quello dei database relazionali. Questo nome proviene dal modello sul quale si basano, il modello relazionale appunto, introdotto da Edgar F. Codd.

Lo schema logico-concettuale, per questa tipologia di database, si basa sulla teoria degli insiemi e si avvale dell'algebra relazionale. Per quanto negli anni siano state proposte le più svariate alternative al modello relazionale, questi è riuscito a mantenere il titolo di miglior prodotto per la progettazione ed implementazione di sistemi reali per decenni.

Tuttavia, con il passare del tempo, ed in particolare con l'evoluzione del web, avutasi agli albori degli anni 2000, i famosi e consolidati vantaggi attribuiti ai database relazionali, data la natura e l'incredibile mole di dati provenienti dalla rete, sono venuti meno. Le aziende leader nel settore sono state così costrette a sviluppare nuove tecnologie, che meglio si adattassero alle stringenti richieste di scalabilità orizzontali dei sistemi e con cui fosse relativamente semplice effettuare cluster di elaborazione [5]. Due esempi ci vengono dati da *Dynamo* e *Big Table*, nate nel 2004 rispettivamente dai due colossi Amazon e Google. Da qui nasce lo slancio verso nuove organizzazioni dei dati che, abbandonando i principi del modello relazionale, vengono indicate come tecnologie NoSQL:

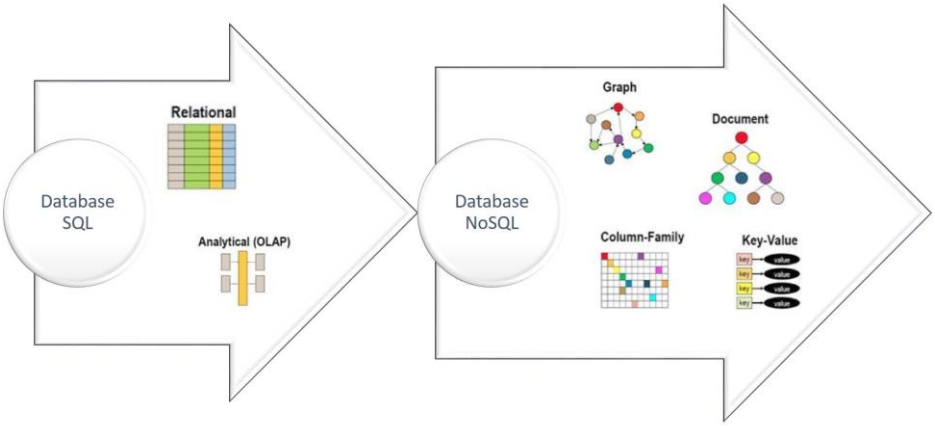


Figura 4.1: passaggio da database SQL a database NoSQL.

Per cogliere le reali differenze introdotte da queste nuove tecnologie, e quindi comprendere la natura di MongoDB, dedichiamo i prossimi paragrafi alla descrizione dei modelli introdotti fino a questo punto.

4.1.1 Database relazionali

I database relazionali nascono nel 1970 dall'idea di un ricercatore della IBM, Edgar F. Codd. All'epoca le tecnologie principali per le basi di dati erano rappresentate dal network database e dallo hierarchical model. Non ci volle molto prima che il modello relazionale fosse riconosciuto come il migliore, soppiantando così in modo definitivo i vecchi sistemi. Il punto di forza di questo nuovo modello, che ne ha determinato anche il successo e la durata, è rappresentato dallo svincolamento fra rappresentazione interna dei dati e come questi possono invece essere gestiti dai programmatori. Inoltre, la presenza di regole chiare e rigorose, e lo sviluppo del linguaggio SQL, hanno sancito la vittoria dei database relazionali sul mercato. Gli anni successivi all'introduzione del modello relazionale hanno visto la nascita di svariati RDBMS (Relational Data Base Management System) e la migrazione delle aziende verso questa nuova tecnologia.

Passiamo ora, dopo questa breve introduzione storica sulla nascita e la diffusione del modello relazionale, all'esposizione di quelle che sono le sue caratteristiche funzionali fondamentali.

L'unità logica di lavoro di un RDBMS prende il nome di transazione. Una transazione, anche se composta da più istruzioni SQL, è vista come un'operazione computazionale atomica, ovvero non separabile. Quindi se all'interno di una transazione è prevista un'operazione di lettura e successivamente una di scrittura sul database, e malauguratamente quest'ultima operazione non dovesse andare a buon fine, l'intera transazione risulterebbe non conclusa correttamente, andando così a ripristinare il contenuto del database al suo stato di partenza. I dati inoltre sono vincolati a schemi rigidi e a vincoli di integrità referenziale.

Il contenuto di un database relazionale prevede la normalizzazione dei dati ed un'organizzazione a tabelle, le cui righe vengono riempite dai dati stessi.

Esistono in aggiunta tutta una serie di processi relativi alla gestione dell'accesso concorrente e al recovery dei dati che garantiscono il soddisfacimento delle cosiddette proprietà ACID [6]:

- **Atomicity:** una transazione non può essere divisa in unità più piccole. Non è possibile quindi lasciare il database in uno stato intermedio durante l'esecuzione;
- **Consistency:** l'esecuzione di una transazione non deve violare i vincoli di integrità imposti al database;

- **Isolation:** l'esecuzione di una transazione è indipendente dall'esecuzione simultanea di altre transazioni;
- **Durability:** l'effetto di una transazione completata non viene perso in caso di guasto. Così viene garantita l'affidabilità del sistema;

Precisiamo che il quadro generale appena esposto è una semplificazione del reale processo di funzionamento di un database relazionale, ed ha il solo obiettivo di far emergere le caratteristiche che, come vedremo a breve, sono differenti rispetto a quelle dei DBMS non relazionali quali MongoDB.

4.1.2 Database non relazionali

Nella sezione 4.1.2 abbiamo presentato le caratteristiche principali di un database relazionale. Fra queste ricordiamo l'utilizzo del linguaggio SQL per le interrogazioni e le modifiche sui dati, la presenza di uno schema rigido e la normalizzazione.

I database non relazionali, invece, sono caratterizzati dai seguenti aspetti:

- Non utilizzano il linguaggio SQL, per tale ragione vengono indicati anche come database NoSQL;
- Lo schema non è rigido, nel senso che sono possibili modifiche dello stesso in corso d'opera;
- È prevista la replicazione dei dati;

Mentre i database relazionali, con le proprietà ACID, sono orientati fortemente alla consistenza dei dati, i database non relazionali, puntando sulle performance del sistema, rendono i dati più disponibili, inficiando quindi sulla loro consistenza.

Molte delle proprietà dei database NoSQL provengono direttamente dal teorema CAP (Consistency Availability Partition), esposto per la prima volta da Eric Brewer nel luglio del 2000. Questo teorema in realtà non è rivolto strettamente al solo ambito dei database, ma si riferisce allo scenario più vasto delle applicazioni distribuite. Il teorema CAP afferma che, dati i requisiti di:

Consistency: abilità del sistema di eseguire un'operazione per intero, altrimenti di non eseguirla affatto;

Availability: il sistema è sempre in grado di dare una risposta, sia questa positiva o negativa;

Partition tolerance: capacità del sistema distribuito di proseguire nel suo funzionamento anche nel caso in cui uno dei suoi nodi vada fuori uso;

In un sistema di dati condivisi distribuito è possibile avere solo due delle tre citate proprietà [7].

La figura sotto riportata mostra graficamente il concetto esposto dal teorema CAP. In particolare l'area contrassegnata in rosso indica la "zona proibita", ovvero quella nella quale sussistono tutte le proprietà:

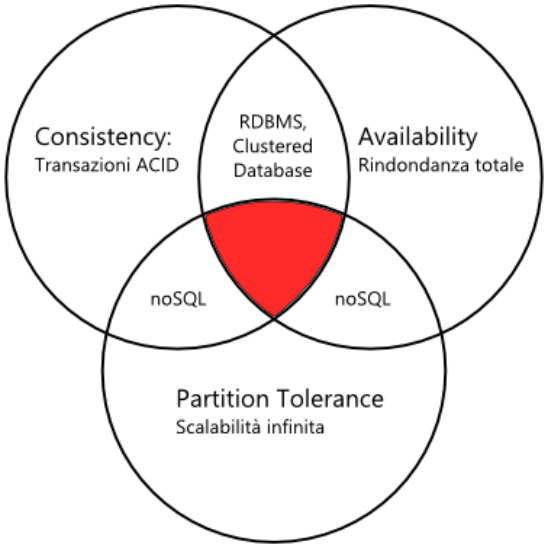


Figura 4.2: teorema CAP

Da questo enunciato Brewer ha poi delineato le cosiddette proprietà BASE, che sono:

- **Basically Available**: il sistema garantisce una risposta ad ogni richiesta;
- **Soft state**: lo stato del sistema può cambiare nel tempo anche in assenza di input. Questa è una conseguenza dell'eventually consistency;
- **Eventually consistent**: il sistema diventerà eventualmente consistente quando in qualche momento non ci saranno più

degli input. Quindi continuerà a dare risposte, senza controllare la consistenza di ogni transazione prima di passare alla successiva;

I database NoSQL sono stati costruiti partendo da questi principi teorici. In particolare la proprietà che predomina è quella di *partition tolerance*.

Tuttavia le caratteristiche generali dipendono dalla specifica tipologia di database non relazionale. Esistono infatti diverse famiglie di database NoSQL. Fra queste citiamo:

Column-oriented: come il nome stesso suggerisce, questi database prevedono il salvataggio dei dati in colonne. Ricordiamo che i database relazionali, invece, sono raw-oriented. La differenza fra questi due approcci è mostrata nella seguente figura:

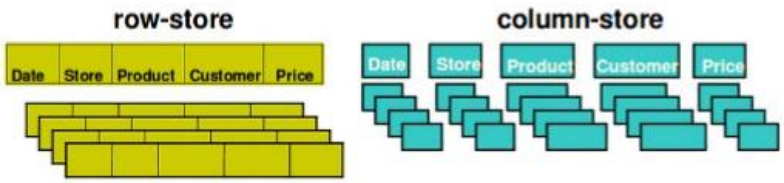


Figura 4.3: Strutture dati row-store e column-store.

Le principali implementazioni commerciali NoSql facenti parte della famiglia column-oriented sono Google Datastore, HBase e Apache Cassandra;

Document store: Questi sono database orientati ai documenti che permettono l'inserimento e la manipolazione di dati organizzati in modo semi-strutturato. I documenti fungono da record del database, i quali quindi non rispondono a nessuno schema univoco, essendo gli uni diversi dagli altri. E' comunque possibile sottoporre query complesse e creare indici sui dati [8]. La flessibilità di questo modello lo rende una delle scelte migliori quando si parla di dati provenienti dal web, per loro natura eterogenei, destrutturati e mutabili. La figura seguente mostra la differenza esistente nella rappresentazione interna dei dati fra un database relazionale ed uno orientato ai documenti:

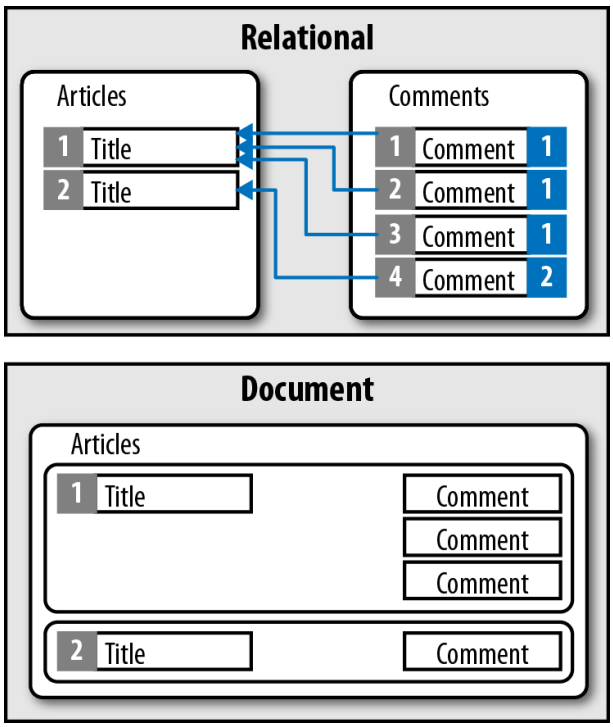


Figura 4.4: modello dei dati relazionale vs. orientato ai documenti.

Le principali implementazioni commerciali NoSql facenti parte della famiglia document oriented sono lo stesso MongoDB, CouchDB, BaseX, Jackrabbit, Terrastore, Lotus Notes e Redis;

Key-value store: questi database prevedono la memorizzazione dei dati tramite coppie chiave-valore. Poiché non è prevista un'organizzazione in strutture complesse, questo modello pone delle limitazioni relativamente alla complessità delle query e all'indicizzazione dei dati. Questi infatti possono essere manipolati solo tramite la chiave, mentre i valori restano oscuri al sistema. Dati tali caratteristiche, e poiché è prevista l'implementazione degli array associativi, questi database vengono principalmente sfruttati in alcune tipologie di memorie cache. La seguente figura mostra uno schema esemplificativo della differenza strutturale dei dati rispettivamente fra un database relazionale, uno chiave-valore ed uno orientato ai documenti:

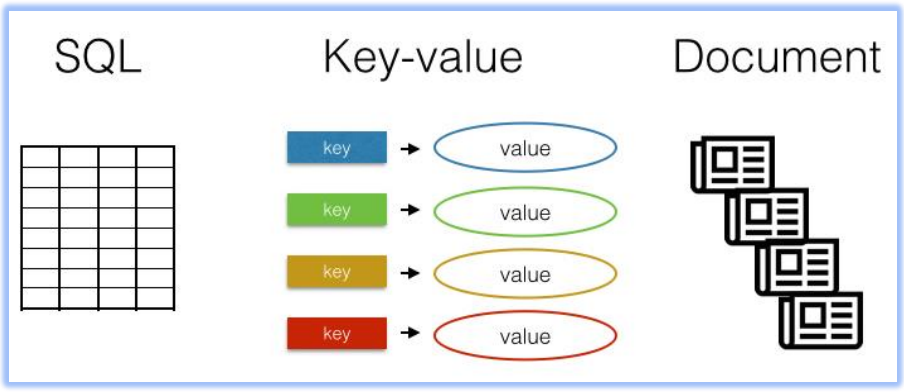


Figura 4.5: tabelle, coppie chiave-valore e documenti.

Le principali implementazioni commerciali NoSql facenti parte della famiglia key-value store sono MemcacheDB, Redis, Memcached, Voldemort e Berkley DB;

Esiste infine la famiglia dei database orientati ai grafi tuttavia, poiché la seconda parte del presente capitolo è interamente dedicata a questa tipologia, rimandiamo la loro trattazione al paragrafo 4.2.

4.1.3 Caratteristiche di MongoDB

Come anticipato nelle sezioni precedenti, MongoDB è un DBMS facente parte della famiglia dei database orientati ai documenti. I componenti principali dell'architettura di MongoDB sono [9]:

- **Database**: così come previsto dai database relazionali, questo è il livello più alto del sistema. Tuttavia, mentre i database relazionali contengono principalmente tabelle e view, il database MongoDB raccoglie strutture dati che prendono il nome di collezioni. Di solito un'istanza di MongoDB server contiene più database;

- **Collezioni**: le collezioni di MongoDB sono insiemi di documenti. Questi equivalgono alle tabelle dei database relazionali. Anche se le collezioni sono collegate fra di loro, non è previsto uno specifico schema di connessione come fra le tabelle dei database relazionali; ˆ

- **Documenti**: un documento è l'unità dei dati elementare di MongoDB. Ognuno di questi è costruito da coppie di attributi

chiave-valore. Documenti appartenenti alla stessa collezione possono inoltre avere, a differenza di quanto visto per i database relazionali, campi che differiscono.

La seguente figura mostra uno schema di massima dell'architettura di MongoDB:

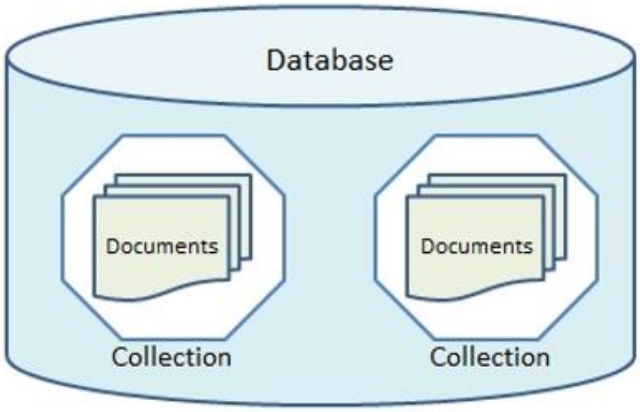


Figura 4.6: architettura di MongoDB.

Il formato di rappresentazione dei documenti di MongoDB prende il nome di BSON[12], progettato per essere efficiente sia dal punto di vista dello spazio di memorizzazione dei dati, che dal punto di vista della velocità di ricerca.

All'interno dei documenti di MongoDB il valore di un campo può essere uno qualsiasi dei possibili tipi di dati che BSON permette di rappresentare, inclusi altri documenti, array oppure array di documenti.

Per quel che riguarda l'accesso ai dati, MongoDB non offre delle API ReST basate su HTTP, ma adotta uno schema proprietario che si appoggia su TCP/IP, il quale mette a disposizione anch'esso operazioni di tipo CRUD[13] [10].

MongoDB offre inoltre due scelte per creare il collegamento fra i documenti, che sono: le relazioni, ovvero dei link per rappresentare i collegamenti fra documenti; i documenti innestati, ovvero un documento che ha come unico compito quello di contenere dati che sono fra di loro correlati [11].

È inoltre prevista la presenza degli indici, implementati tramite BTree, atti alla memorizzazione di solo una porzione di una collezione.

[12] **BSON**: Il Binary JSON è estensione del formato JSON (JavaScript Object Notation) per lo storing di oggetti binari.
[13] **CRUD**: Create, Read, Update, Delete.

Grazie a queste particolari strutture dati è quindi possibile accedere velocemente ad una sotto parte di interesse dell'intero database. Le tipologie di indice previste da MongoDB sono:

- **default_id**: indice creato automaticamente sul campo id di ogni collezione;
- **single field**: indice creato su di un qualunque campo di un documento;
- **computed index**: indice composto su più campi di un documento, dei quali il primo detta i criteri di ordinamento;
- **multikey index**: particolare struttura dati utilizzata da MongoDB per indicizzare i valori dei campi di un array;
- **geospatial index**: indice nato per aumentare l'efficienza delle query di ricerca su coordinate geospaziali;
- **text indexes**: indice nato per aumentare l'efficienza delle query di ricerca su testo libero in una collezione;

Prima di concludere questa parte relativa a MongoDB, alla luce delle conoscenze acquisite fino ad ora, riprendiamo il teorema CAP esposto precedentemente in relazione alle proprietà considerate per la costruzione di database NoSQL.

MongoDB si pone nella zona di intersezione CP, quindi risponde alle proprietà di *consistency* e *partition tolerance*. Sono tuttavia previsti dei particolari meccanismi per aumentare sia la disponibilità spaziale che quella temporale dei dati. In particolare il *Replica Set* è un meccanismo di replicazione dei dati di MongoDB. Questi si basa sulla copia degli stessi su server diversi così da salvaguardare l'intero sistema in caso di guasto di una singola macchina. Fra tutte queste istanze di MongoDB soltanto una, chiamata primary, è preposta alle operazioni di lettura e scrittura dei dati. Le altre istanze vengono aggiornate in modo asincrono.

Infine il meccanismo di MongoDB utilizzato per la partizione dei dati tra macchine diverse prende il nome di *Sharding*. I server del cluster prendono il nome di shard, ed ognuno di questi rappresenta un database indipendente dagli altri. L'instradamento delle varie richieste fra il client e lo shard opportuno viene gestito dai *query routers*. I metadati necessari al corretto funzionamento di un cluster sono invece contenuti nei *config server*. La divisione dei dati fra i diversi shard può essere o di tipo range based, dove i dati vengono divisi a blocchi, oppure di tipo hash based, applicando un'opportuna funzione hash.

4.2 Neo4j

Neo4j è un database open source orientato ai grafi, prodotto dalla Neo Technology. Il notevole vantaggio proveniente dall'adozione di questa tecnologia risiede nel fatto che la teoria dei grafi, applicata al calcolo di shortest path problem, come previsto in Stardust, porta una serie notevole di vantaggi, come Neubauer suggerisce [12].

In effetti poter disporre di un'organizzazione dei dati del tutto sovrapponibile al loro reale contenuto informativo è un raro privilegio per i programmatori.

Per comprendere il funzionamento di Neo4j è necessario fare innanzitutto un passo indietro per comprendere la natura dei grafi e come questa viene sfruttata dai DBMS.

4.2.1 DBMS a grafo

Un grafo è una struttura costituita da due entità base: i nodi e gli archi. I nodi, anche detti vertici, sono connessi fra di loro tramite gli archi, anche detti relazioni. La figura seguente mostra un esempio di grafo:

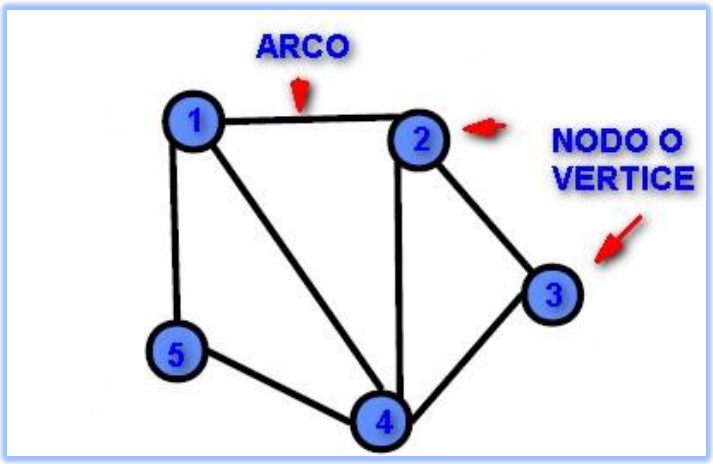


Figura 4.7: esempio di grafo.

Grazie ai grafi è possibile modellare un numero elevatissimo di scenari, dove i nodi rappresentano le entità e gli archi le relazioni che esistono fra questi.
Ogni arco inoltre può possedere un orientamento, e deve sempre avere origine e fine in un nodo.

Fra i numerosi tipi di grafo ad oggi esistenti quello di maggior interesse per il presente studio di tesi è il *property graph model*. Questo modello prevede l'aggiunta di un'ulteriore entità: le proprietà. Queste in realtà non costituiscono degli oggetti a se stanti, ma sono piuttosto un'importante aggiunta applicata ai nodi e agli archi di un grafo [13].
I DBMS a grafo rappresentano quindi i dati tramite nodi ed archi. Rispetto al modello relazionale e a quello orientato ai documenti, discussi nel paragrafo precedente, i *graph database* forniscono un modello sicuramente più originale ed in molti casi anche più potente e flessibile. Ad esempio, nel caso in cui i dati risultassero fortemente connessi, la scelta di utilizzare relazioni esistenti fra nodi, rispetto alle costose operazioni di join fra tabelle in un database relazionale, aumenterebbe di molto le performance.

Inoltre i grafi per loro natura non prevedono uno schema rigido, di conseguenza riescono ad adattarsi bene all'aumentare della mole dei dati. Questa flessibilità si traduce nella semplice aggiunta di vertici e relazioni fra gli stessi, senza inficiare sulle precedenti configurazioni del sistema.

Oltre ai nodi, agli archi e alle proprietà esistono altre entità che permettono ai grafi di modellare un elevato numero di situazioni. Fra queste vi sono le *Labels*, ovvero delle etichette applicabili a sottogruppi di nodi. Ritornando all'editor di Stardust, riconsideriamo la funzionalità per dividere un piano in più reparti. Questi sono proprio insiemi di nodi contrassegnati dalla stessa label. In generale è comunque possibile associare allo stesso nodo più etichette. Queste, oltre che a modellare contesti reali, possono essere utilizzate anche per velocizzare l'esecuzione di query che devono agire solo su di una sotto parte del grafo. Il loro utilizzo viene previsto anche in caso di aggiunta di indici sulle proprietà, per definire delle constraint, oppure a runtime per seguire lo stato dei nodi.

In un DBMS a grafo ci sono anche i *Path*, ovvero i percorsi lungo i nodi e le relazioni del grafo, ottenibili mediante query. La proprietà più significativa di un percorso è la sua lunghezza, in termini di archi intrapresi dallo stesso. Quindi un percorso di lunghezza zero parte e arriva nello stesso nodo. Un percorso di lunghezza uno comprende invece o due nodi connessi direttamente da un arco, oppure un singolo nodo auto-connesso tramite un arco.

Sui path è possibile applicare potenti algoritmi, i Graph Alorithms, che permettono, specificati i nodi di partenza e di arrivo, di determinare uno o più percorsi rispondenti a precise caratteristiche. Fra questi ricordiamo gli algoritmi di shortest paths, all paths, all simple paths e Dijkstra [15].

4.2.2 Caratteristiche e architettura di Neo4j

Neo4j fa parte della categoria dei graph DBMS. I suoi punti di forza, che ne fanno una scelta vincente per molte applicazioni, sono le prestazioni, la scalabilità e la robustezza ai guasti del sistema. Le principali caratteristiche di Neo4j sono:

- **Transazioni ACID:** ovvero rispondenti alle proprietà di atomicità, consistenza, isolamento e durabilità (vedi sotto-paragrafo 4.1.1);
- **Availability**: ovvero il sistema è sempre in grado di dare una risposta, sia questa positiva o negativa;
- **Cypher**: linguaggio di interrogazione dichiarativo;
- **Index-free adjancency**: tabella che contiene tutti i nodi di un grafo, ognuno con relativi puntatori ai nodi connessi;

Neo4j può memorizzare miliardi di nodi ed archi: la index-free adjancency e la divisione di un unico grafo in più *store file,* assicurano comunque alte performance per le query che prevedono l'attraversamento dell'intera struttura.

La seguente figura mostra l'architettura di massima di un server Neo4j:

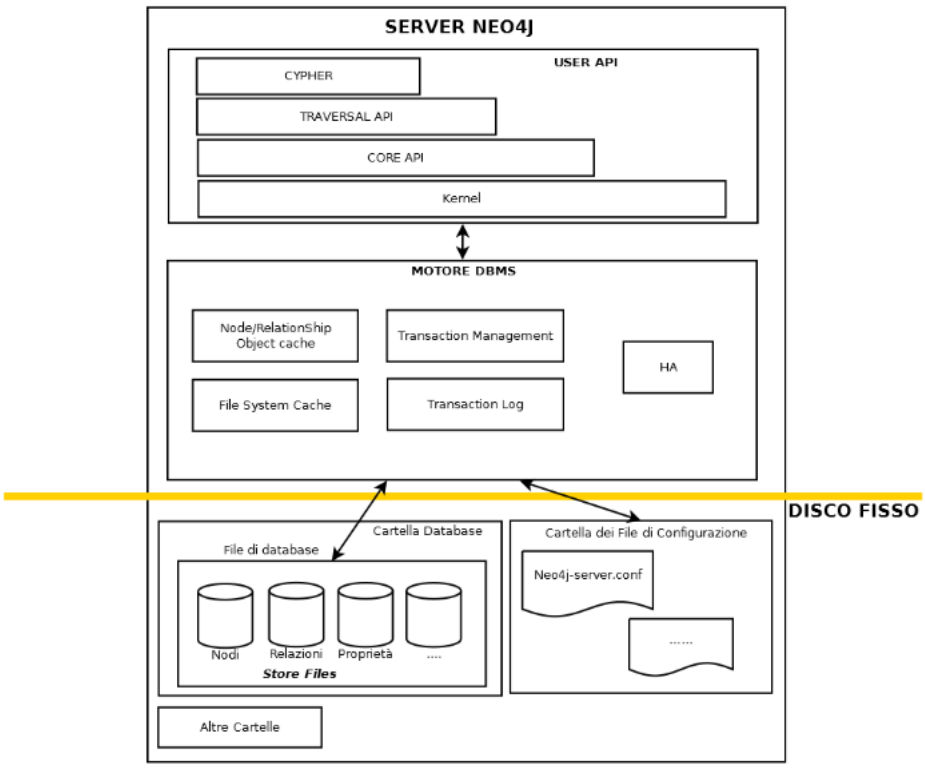


Figura 4.8: architettura di Neo4j [14].

Analizziamo l'immagine sopra riportata partendo dal basso. Prima di tutto troviamo la cartella unica del database che contiene i vari *store file*. Neo4j prevede un numero elevato di differenti store file, tuttavia qui ci concentreremo solo su quelli relativi alle tre entità principali prima introdotte: i nodi, le relazioni e le proprietà.

In particolare ognuno di questi elementi prevede una propria struttura di memorizzazione che prende il nome di *record*. Di seguito è riportata la struttura del record previsto per i nodi:

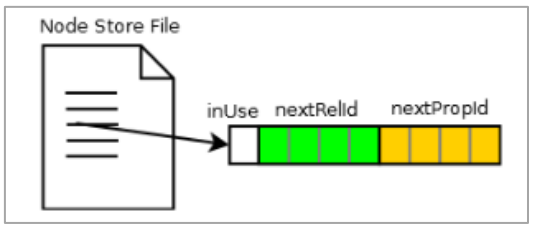


Figura 4.9: struttura dei record per i nodi di un grafo.

I record di uno stesso elemento hanno una lunghezza fissa. Questo fa sì che le ricerche possano essere svolte molto velocemente. Quella prevista per i nodi è di 9 byte. Questi sono suddivisi nel seguente modo:

- **inUse (1 byte)**: flag che segnala se il record corrente contiene oppure no i dati relativi ad un nodo del grafo;
- **nextRelId (4 byte)**: id del primo arco connesso al nodo;
- **nextPropId (4 byte)**: id della prima proprietà relativa al nodo;

Di seguito è riportata la struttura del record previsto per gli archi:

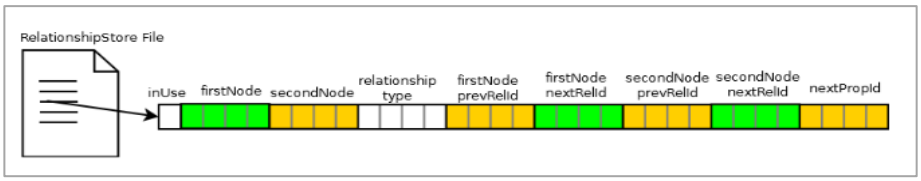


Figura 4.10: struttura dei record per gli archi di un grafo.

La lunghezza prevista per i record delle relazioni è di 33 byte, molto maggiore quindi rispetto a quella dei nodi, data la natura degli archi.

I campi seguono la stessa logica vista per i record relativi ai nodi, ma in questo caso sono previste delle informazioni aggiuntive di:

- **firstNode (4 byte)**: id del nodo di partenza dell'arco;
- **secondNode (4 byte)**: id del nodo di arrivo dell'arco;
- **relationship type (4 byte)**: spazio dedicato al codice identificativo del tipo di arco;
- **firstNodeprevRelId (4 byte)**: id del nodo di partenza dell'arco precedente;
- **firstNodenextRelId (4 byte)**: id del nodo di partenza dell'arco successivo;
- **secondNodeprevRelId (4 byte)**: id del nodo di arrivo dell'arco precedente;
- **secondNodenextRelId (4 byte)**: id del nodo di arrivo dell'arco successivo;

Infine ci sono i record delle proprietà, mostrati nella seguente figura:

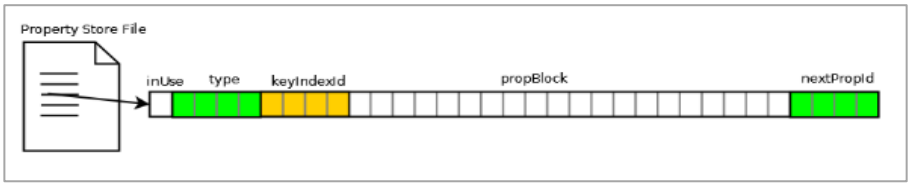


Figura 4.10: struttura dei record per le proprietà di un grafo.

Possiamo notare dall'immagine sopra riportata che anche questi record hanno una lunghezza pari a 33 byte, tuttavia i campi sono strutturati in modo diverso, fatta eccezione per il primo e l'ultimo. Abbiamo infatti:

- **type (4 byte)**: tipo di proprietà;
- **keyIndexId (4 byte)**: id dell'indice;
- **propBlock (20 byte)**: blocco riservato al contenuto della proprietà;

Risalendo lungo l'architettura di Neo4j troviamo il motore del DBMS, composto da cinque elementi fondamentali:

1. *Node/Relationship Object Cache*: preposta al mantenimento di parti di grafo caricate di recente dal File System;
2. *File System Cache*: preposta al caricamento di porzioni di store file;
3. *Transaction Management*: gestore delle transazioni;
4. *Transacrion Log*: file di log delle transazioni;
5. *HA*: parte preposta all'ottenimento della proprietà di High Availability;

Infine il livello più alto dell'architettura è riservato alla *User API*. Neo4j mette a disposizione diverse librerie il cui utilizzo dipende fortemente allo specifico scenario di applicazione.

Partendo dal livello più basso troviamo: *Kernel API*, per accedere al ciclo di vita delle transazioni; *Core API*, per la manipolazione del grafo mediante codice Java; *Traversal API*, per molti versi simile alla *Core API*, ma è una Java API dichiarativa; Cypher, il linguaggio nativo di Neo4j per le interrogazioni.

Fra le altre API messe a disposizione dal server, quella maggiormente utilizzata per il presente lavoro di tesi è stata l'API

REST. Questa è preposta all'invio delle richieste in formato JSON lato client.

Una delle caratteristiche di Neo4j, che lo rendono differente da molte altre architetture, è la presenza di due differenti tipologie di *indice*. Ricordiamo che gli indici sono strutture dati accessorie costruite per rendere l'accesso ad un sottoinsieme di dati molto più veloce. I due indici suddetti sono lo *schema index* ed il *non-schema index*. La differenza sostanziale fra queste due tipologie risiede nel fatto che gli indici *non-schema* vengono implementati tramite un componente chiamato *Lucerne*, capace di definire e costruire diverse tipi di indice. La comunicazione con Lucerne avviene tramite chiamate POST all'API REST. Un'ulteriore caratteristica di questi indici è che, a differenza degli *schema index*, il loro aggiornamento è a carico del programmatore e non del DBMS.

4.3 Conclusioni

Nella prima parte di questo capitolo abbiamo passato in rassegna le caratteristiche di base delle due principali tipologie di database ad oggi esistenti: database SQL e database NoSQL.

Successivamente siamo passati alla trattazione delle due tecnologie da noi scelte per l'implementazione di Stardust: MongoDB e Neo4j. La necessità di utilizzare due diversi sistemi di storage dei dati deriva dalla doppia natura Stardust, ovvero piattaforma web ed editor. Quindi, alla luce delle varie considerazioni fatte in fase di progettazione dell'architettura del sistema, come già discusso nel capitolo 3, la nostra attenzione si è focalizzata verso dei DBMS che potessero adattarsi a queste due esigenze.

MongoDB è stato utilizzato per i dati relativi alla pura parte web, in particolare per la gestione delle sessioni.

Neo4j, invece, è stato utilizzato per modellare i dati relativi ai progetti costruiti nell'editor.

Entrambe le tecnologie rispondono ai requisiti, individuati in fase di progettazione, di flessibilità, scalabilità e performance elevate. In particolare MongoDB ci ha permesso di memorizzare dati caratterizzati da una struttura variabile, senza inficiare sulla reperibilità degli stessi. Anche Neo4j, grazie alla sua rappresentazione a grafo perfettamente sovrapponibile alla reale natura dei dati dell'editor, si è rivelata la scelta vincente.

In entrambi i casi, inoltre, l'assenza di uno schema rigido ci ha permesso di avanzare, in fase di sviluppo, secondo la metodologia del trial and error, senza perturbare i dati preesistenti nel sistema.

Puntualizziamo infine che entrambi i DBMS sono stati descritti nelle loro parti architetturali e nelle loro funzionalità dando particolare risalto agli aspetti che sono stati maggiormente utilizzati nella fase implementativa di Stardust.

Capitolo 5

Application back end

Il presente capitolo è dedicato alla descrizione della tecnologia utilizzata per l'application back end di Stardust: Node.js. Questi è un framework sviluppato per la realizzazione di applicativi server side. In Node.js l'application back end viene implementato mediante Javascript, il noto linguaggio di scripting, orientato agli eventi ed agli oggetti, che normalmente viene utilizzato per la realizzazione di applicazioni web lato client. L'intero sistema si basa sul Javascript Engine V8, il runtime di Google, disponibile per le principali piattaforme. Come avremo modo di approfondire nel corso del presente capitolo, Node.js sfrutta il modello di programmazione *event-driven*, in modo asincrono così da assicurare un'efficienza di elaborazione maggiore.

5.1 Node.js

Una delle principali caratteristiche di Node.js, che lo differenziano dalla maggior parte dei prodotti utilizzati per lo sviluppo dei server, è quella utilizzare il linguaggio JavaScript, tipicamente utilizzato lato client.

Node.js sfrutta la programmazione ad eventi, abbandonando quindi i modelli che prevedono la gestione di thread concorrenti oppure quelli basati sui processi, utilizzati dai web server tradizionali. Il tutto avviene in modalità asincrona.

Tutte queste peculiarità fanno di Node.js un prodotto davvero innovativo per la programmazione lato server.

Gli ambiti nei quali questa tecnologia sta riscuotendo un enorme successo sono: applicazioni real-time; giochi on-line, chat, social network, sistemi di messaggistica, sistemi di notifica.

Di seguito è riportata l'immagine relativa all'architettura di Node.js, i cui elementi saranno discussi nei sotto paragrafi successivi:

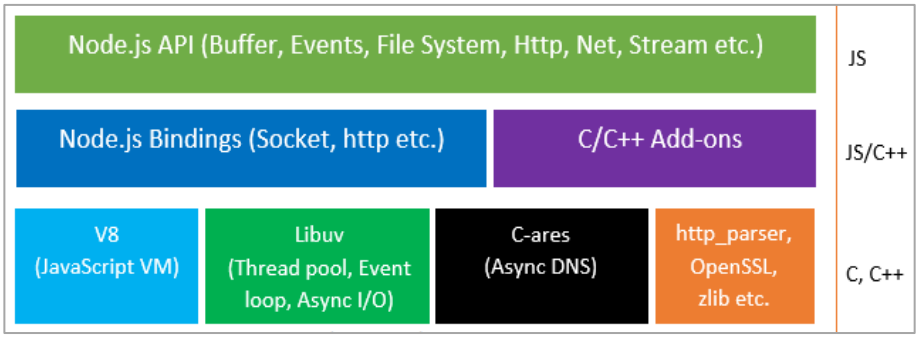


Figura 5.1: architettura di Node.js

5.1.1 JavaScript

JavaScript è il linguaggio di scripting più famoso nel mondo della programmazione web lato client.

È orientato agli oggetti e agli eventi, e trova largo utilizzo nella creazione di elementi dinamici ed interattivi.

La potenza di questo linguaggio come protagonista della programmazione lato client è indiscussa. Più complessa, invece, è la determinazione del ruolo che JavaScript può giocare all'interno di un ambiente completamente diverso, come è quello della programmazione lato back end. Il primo dubbio potrebbe nascere proprio dalle funzionalità messe a disposizione da questo linguaggio, concepito sin dall'inizio per realizzare animazioni. Per la creazione di un back end applicativo, infatti, vengono normalmente adoperati linguaggi che presentano caratteristiche che JavaScript oggettivamente non ha, quali [16]:

- comunicazione con il sistema operativo;
- gestione dei thread;
- strutturazione in classi;
- presenza di moduli e librerie ad hoc;

Tuttavia, come avremo modo di approfondire nel paragrafo 4.1.2, la possibilità di poter adoperare un approccio event driven, rende JavaScript adatto anche alla programmazione lato server, e non solo. Il fatto che questo linguaggio non preveda la gestione del livello più basso del sistema, lo rende uno strumento più semplice per i programmatori.

5.1.2 Programmazione ad eventi

Il modello event-drive prevede, come il nome stesso suggerisce, che un programma rimanga in ascolto per il verificarsi di determinati eventi, così da reagire a questi in modo opportuno. Quindi tutte le azioni avvengono a seguito del verificarsi di un certo fatto.

Le operazioni avvengono in modo asincrono (vedi paragrafo 4.1.3). Spieghiamo ora la logica implementata in V8, che permette la programmazione ad eventi. Nel momento in cui devono essere eseguite operazioni di I/O lente, come quelle relative alle comunicazioni di rete o all'accesso al disco fisso, V8 passa la chiamata su di un thread. Questo thread è non bloccante e viene scelto all'interno della *thread-pool*. Tale processo permette al thread principale di proseguire nella sua esecuzione. Quando poi il thread non bloccante avrà terminato il suo compito, sarà V8 stesso ad intercettare la sua disponibilità per una nuova esecuzione, ed a riporre il thread nella thread-pool. La seguente immagine mostra un esempio pratico del processo appena descritto, relativo al modulo *http* (vedi paragrafo 4.1.4):

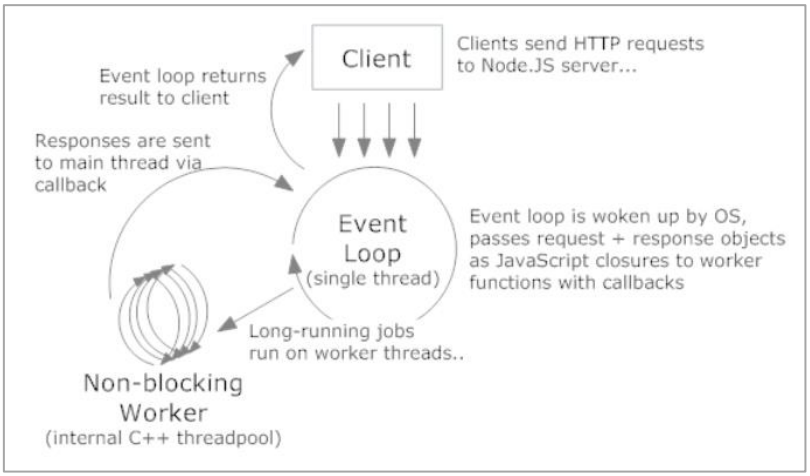


Figura 5.2: funzionamento modulo http di Node.js.

5.1.3 Esecuzione asincrona

L'esecuzione asincrona, che si ha quando "il mittente spedisce il messaggio e continua ad effettuare le proprie operazioni", è una caratteristica di Node.js che gli permette di vantare ottime performance. Infatti le operazioni più lente, quali l'accesso al disco, non vanno ad intaccare l'efficienza dell'intero sistema.

Quindi, mentre il paradigma di esecuzione sincrona prevede che un'azione venga eseguita solo a seguito del completamento di quella precedente, nell'esecuzione asincrona, invece, l'azione viene lanciata ogni volta che si verifica un evento, indipendentemente dal punto di esecuzione nella quale di trova l'azione precedente.

La seguente figura mostra la differenza del susseguirsi delle varie fasi esecutive fra un'architettura sincrona ed una asincrona [18]:

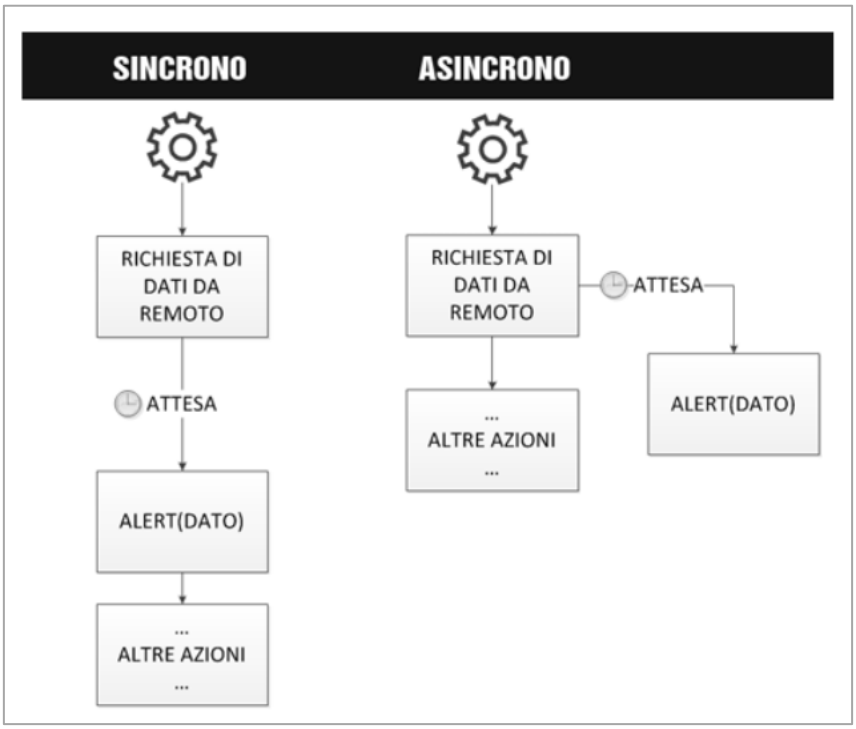


Figura 5.2: esecuzione sincrona vs. esecuzione asincrona.

5.1.4 I moduli di Node.js

Node.js offre alcuni utili moduli che il programmatore può utilizzare direttamente all'interno del suo codice. Una parte di questi moduli vengono rilasciati automaticamente all'installazione di Node.js, altri invece possono essere inseriti in corso d'opera tramite il comando *npm*, che richiama l'omonimo package manager. I moduli principali sono [17]:

- **Http**: questo modulo mette a disposizione la classe *Server*, che serve ad avviare un web server;
- **Globals**: permette di interagire con alcuni utili oggetti e variabili, quali il nome di file e cartelle correnti o lo standard I/O;
- **Url**: grazie a questo modulo è possibile creare e modificare gli url. Questi possono essere trattati sia come oggetti che come stringhe;
- **File system**: questo modulo lavora in comunione con *path* (vedi sotto) per la gestione di varie operazioni da svolgere sui file del file system, quali: scrittura; lettura; copia; rinomina.
- **Path**: utile alla gestione dei reali percorsi previsti dal sistema operativo;
- **Querystring**: utile alla creazione di stringhe di valori. Ogni elemento della stringa ha una struttura uguale del tipo *nome* = *valore*;
- **Util**: modulo preposto all'ottimizzazione dell'uso delle variabili;
- **Net**: per la realizzazione di applicativi server-client basati sulle socket;
- **Soket.io**: permette la creazione di connessioni attive fra server e client. Questi può quindi ricevere gli aggiornamenti da parte del server senza la necessità di dover formalizzare delle richieste;

5.1.5 Node.js e Express

Costruire un server da zero, con la definizione di tutte le API REST necessarie, utilizzando Node.js puro potrebbe portare il programmatore verso la stesura di codice confusionario, poco modulare e riusabile. Per tale ragione sono stati sviluppati framework dedicati, pensati proprio per facilitare la vita dello sviluppatore alle prese con Node.js.

Fra i vari prodotti esistenti quello da noi utilizzato per l'implementazione di Stardust è *Express*. Questi viene definito come un framework per applicazioni web Node.js flessibile e leggero che fornisce una serie di funzioni avanzate per le applicazioni web. Express prevede una miriade di metodi di utilità HTTP e middleware, così la creazione di un'API affidabile risulta un processo facile e veloce. Inoltre fornisce uno strato sottile di funzionalità di base per le applicazioni web, senza nascondere quelle di Node.js [19].

Uno degli aspetti più vantaggiosi di questo framework è rappresentato dai *Routers*, sui quali possono essere definiti gli endpoint dei servizi previsti. Quindi Express permette di organizzare tutte le route pertinenti tra loro in file separati, in modo estremamente semplice ed immediato. Inoltre le route possono essere definite sia a livello di *router* che a livello di *app*. La struttura dei *Routers* prevede tre campi [20]:

1. **Method**: metodo http al quale rispondere;
2. **Path**: campo che definisce la rotta. Qui possono essere presenti espressioni regolari e/o parametri;
3. **Handler**: funzione che deve essere eseguita al match della rotta. A questa funzione in ingresso vengono passati gli oggetti *request* e *response*;

5.1.6 Middlewares

La definizione dei *Middlewares* è quella di funzioni che hanno accesso a tre oggetti: oggetto *request*, oggetto *response* e oggetto *middleware successivo*.

Queste funzioni hanno un comportamento simile a quello degli interceptors sulle richieste web. Ogni middleware può quindi decidere una delle seguenti azioni:

- Eseguire qualsiasi codice;
- Apportare modifiche agli oggetti richiesta e risposta;

- Terminare il ciclo richiesta-risposta;
- Chiamare la successiva funzione middleware nello stack;

Express (vedi sottoparagrafo 4.1.5) procede nella sua esecuzione proprio tramite chiamate al middleware che possono essere:

- A livello applicativo;
- A livello del router;
- Per la gestione di errori;
- Nativo o di terze parti;

5.2 ReST

A oggi la realizzazione di un application back end richiede la strutturazione di un ecosistema capace di trasformare, aggregare, replicare, cercare ed archiviare una mole significativa di dati.

Tali sistemi devono inoltre rispondere ai requisiti di robustezza, interoperabilità, scalabilità, efficienza ed alte performance. E così negli ultimi anni è nata la necessità di poter disporre di sistemi dinamici ed incentrati sui dati.

Da questi bisogni nasce l'architettura software ReST (Representational State Transfer), pensata appositamente per il web. Questo nome è stato utilizzato per la prima volta da Roy Fielding, negli anni 2000, per definire questa architettura ibrida, poiché derivante da diversi sistemi distribuiti network-based ed integrata di vincoli addizionali che definiscono un'interfaccia di comunicazione standard.

In poco tempo ReST ha rimpiazzato i suoi predecessori. Gli ingredienti principali del suo successo sono stati la sua semplicità, la sua conformità al protocollo HTTP e alla sua caratteristica di essere resource oriented. In particolare lo scambio di risorse fra client e server avviene tramite un'interfaccia predefinita: il client è allo scuro di come queste risorse vengano organizzate sul server.

È quindi previsto un "accordo" iniziale fra client e server sul protocollo da utilizzare per scambiarsi le risorse.

L'architettura ReST prevede un'implementazione libera dei componenti, tuttavia impone i seguenti sei vincoli [21]:

1. **Cacheable**: è previsto che il client possa fare caching. Per tale ragione lato server devono essere specificate quali risorse possono essere cachate e quali invece no;

2. **Client-Server**: Il server è allo scuro dello stato del client, così come il client è allo scuro di come il server memorizza i dati. I due quindi possono essere sviluppati, o addirittura sostituiti, indipendentemente l'uno dall'altro. Questo è possibile grazie alla presenza di interfacce che li separano, e che rappresentano una costante anche in caso di sostituzione del client o del server;
3. **Code on demand**: questo più che un vincolo rappresenta un'opzione architetturale. In pratica è previsto che il server possa trasferire al client del codice eseguibile. In questo modo i client si fanno carico di una parte dell'application logic;
4. **Layered system**: il server può essere strutturato gerarchicamente su più livelli;
5. **Stateless**: lo stato di ogni sessione deve essere memorizzato solo sul client;
6. **Uniform interface**: questo vincolo è una conseguenza di quello *Client-Server*. Infatti ci dice che, per mantenere separati client e server, le interfacce di comunicazione devono essere costruite in modo omogeneo ed invariabile;

L'elemento chiave dell'architettura ReSTful è la risorsa:

- Si può accedere alle risorse tramite il loro URI, che è un identificatore globale;
- Il client ed il server scambiano fra loro rappresentazioni delle risorse attraverso un'interfaccia standard (nel nostro caso HTTP);
- I campi di una risorsa visibili durante l'esecuzione di una richiesta sono: l'identificatore; il formato di rappresentazione, come l'HTML, XML o JSON; l'azione richiesta.

Ad oggi l'architettura ReST è adoperata da aziende del calibro di Google, Twitter, Amazon, Skype, eBay e Paypal.

5.3 Conclusioni

In questo capitolo abbiamo descritto le caratteristiche di Node.js, la tecnologia adottata per lo sviluppo dell'application back end di Stardust.

Le motivazioni alla base di questa scelta risiedono nelle caratteristiche di Node.js, che lo differenziano dalla maggior parte dei

prodotti utilizzati per lo sviluppo server side. Fra queste la più vantaggiosa è stata quella di poter utilizzare lo stesso linguaggio adoperato lato client: Javascript.

Inoltre la possibilità di poter gestire il sistema ad un livello più alto, supportati dal modello event-driven e dall'esecuzione asincrona, ha semplificato molti aspetti concettuali in fase di implementazione. La gestione dei thread e le operazioni di I/O sono infatti invisibili allo sviluppatore. L'impossibilità di eseguire codice parallelo, poiché in ogni istante vi è sempre un solo thread in esecuzione, sgrava completamente il programmatore dagli oneri legati ai problemi di sincronizzazione.

L'ultima considerazione, ma non per questo meno importante, è quella che Node.js prevede la coesistenza con altri framework, come Express, creati proprio per facilitare lo sviluppo dell'intero sistema tramite una serie di funzioni avanzate ed utilità per la creazione di applicazioni web.

Capitolo 6

Tecnologie per il front end

In questo capitolo presenteremo le due tecnologie utilizzate per lo sviluppo del front end di Stardust: AngularJS e P5.js. In particolare AngularJS è stato il framework impiegato per la costruzione dell'intera piattaforma web, mentre P5.js è stata la libreria utilizzata per la realizzazione dei disegni e delle animazioni sul canvas dell'editor. Quindi considerando, la struttura dell'intera applicazione web (vedi paragrafo 2.4), la home page, la dashboard e l'editor, fatta eccezione dell'area dedicata al disegno, sono state sviluppate in AngularJS. Il foglio di lavoro invece è stato implementato utilizzando P5.js, wrappato all'interno del framework AngularJS.

6.1 AngularJS

AngularJS è un framework open source per lo sviluppo client-side di applicazioni web, interamente scritto in JavaScript.

La sua idea nasce nel 2009 da parte di Misko Hevery, un dipendente Google, e la prima versione è stata rilasciata tre anni dopo, nel 2012. In poco tempo questa tecnologia ha riscosso un grandissimo successo, non solo grazie al grande nome dell'azienda dalla quale è nata, ma anche per le grandi potenzialità che ha dimostrato sul campo. Ad oggi Angular è alla versione 4, che ha abbandonato JavaScript a favore del suo super-set TypeScript, e che prevede numerosi cambiamenti rispetto al modello di programmazione previsto nella prima versione. Tuttavia, poiché questo passaggio è avvenuto contestualmente alla realizzazione del presente lavoro di tesi, la nostra scelta per l'implementazione di Stardust è ricaduta sulla versione 1.5.6, data la sua comprovata stabilità.

Per comprendere appieno il ruolo di un framework come AngularJS, dobbiamo considerare i limiti esistenti nell'utilizzo del solo HTML. Questi infatti ci permette di costruire pagine statiche. Tutte le dinamicità e funzionalità non previste da HTML devono quindi essere create tramite l'utilizzo di script esterni.

AngularJS permette di creare applicazioni dinamiche estendendo l'HTML. Come vedremo nel corso del presente capitolo, questo è possibile grazie alle *direttive,* che insegnano al browser una sintassi nuova, tramite la quale il programmatore potrà creare in modo semplice e veloce contenuti complessi. Di seguito è riportato lo schema a blocchi relativo all'architettura di AngularJS. I singoli componenti saranno discussi nel corso dei seguenti sotto-paragrafi.

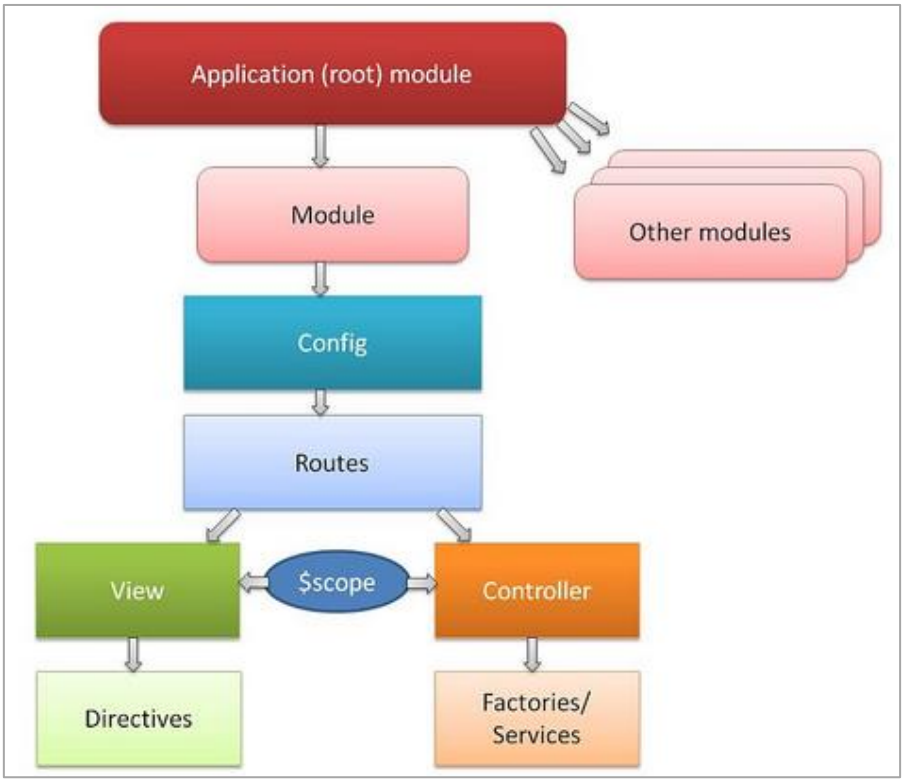


Figura 6.1: Architettura di AngularJS.

6.1.1 Le Single Page Application

Nelle applicazioni web la comunicazione fra server e client ha inizio con la richiesta di quest'ultimo di una pagina. Questa richiesta arriva al server, il quale la processa così da restituire al client la pagina html voluta. Da questo punto in poi il processo di navigazione dell'utente può avvenire in due modi diversi: quello tradizionale e quello basato sulle *Single Page Application*.

Il metodo tradizionale prevede la navigazione tramite l'accesso a link oppure a seguito dell'inserimento di dati in un form. In particolare una di queste azioni innesca l'invio di una nuova richiesta del client al server. Così come visto per la prima pagina, il server processerà la nuova richiesta e restituirà la pagina html dovuta. Questi step si ripetono quindi allo stesso modo per ogni reindirizzamento ad una nuova pagina. Il punto debole del metodo tradizionale è quello di generare una grande quantità di dati sulla rete.

La logica delle Single Page Application prevista in AngularJS, invece, prevede che il client richieda solo la pagina iniziale. Durante la successiva navigazione le interazioni con il server avvengono tramite richieste ajax[14]. Questo si traduce in aggiornamenti parziali della pagina iniziale. Di conseguenza i tempi di risposta dell'applicazione, alle azioni dell'utente, di riducono notevolmente.

[14] **ajax**: L' Asynchronous JavaScript and XML, è una tecnica di sviluppo software che si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente.

La seguente figura mostra le differenze appena esposte fra il ciclo di vita delle pagine web tradizionali e le Single Page Application [22]:

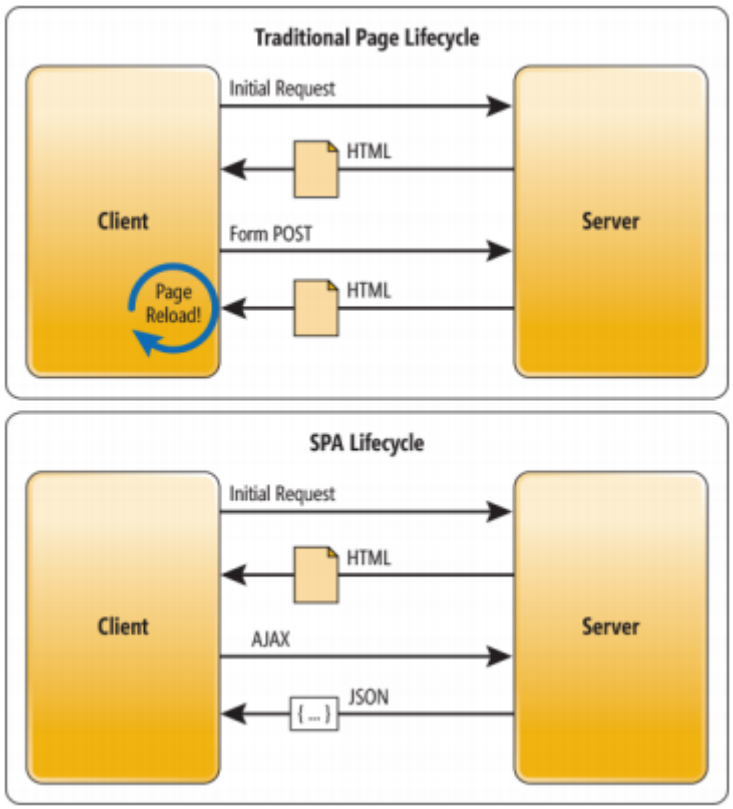


Figura 6.2: applicazioni tradizionali vs. Single Page Application.

6.1.2 Module

I moduli sono dei veri e propri contenitori che racchiudono i vari componenti previsti in un'applicazione AngularJS. In pratica l'intero codice di un'applicazione può essere suddiviso in blocchi semantici differenti, appartenenti ognuno ad uno specifico modulo. In questo modo tutto il codice è ben organizzato e, oltre ad essere più comprensibile, può anche essere testato modularmente e riutilizzato.

I moduli inoltre gestiscono il bootstrap dell'applicazione. Infatti, a differenza di altri framework che prevedono l'esecuzione di un metodo main per l'inizializzazione dei componenti, AngularJS prevede dei moduli dichiarativi che si occupano di questo compito. Ogni modulo ha un nome che lo identifica e che deve essere inserito all'interno dell'html per creare l'associazione con le view.

Oltre ad un identificativo ogni modulo possiede anche un array, atto al contenimento di altri moduli dai quali il modulo corrente ha delle dipendenze. Il corpo di un modulo può essere considerato in due parti distinte:

- **Configuration blocks**: i blocchi di configurazione vengono eseguiti durante le fasi di registrazione dei provider e di configurazione. All'interno di questi blocchi possono essere iniettati solo provider e costanti. Questo vincolo evita che l'istanziamento dei service (vedi sotto-paragrafo 6.1.4) avvenga prima della corretta configurazione dell'applicazione;
- **Run blocks**: i blocchi di esecuzione servono a far partire l'applicazione. La loro esecuzione avviene dopo la configurazione di tutti i service (vedi sotto-paragrafo 6.1.4). All'interno di questi blocchi possono essere iniettate solo istanze e costanti. Questo vincolo evita che in fase di esecuzione avvengano configurazioni del sistema;

Abbiamo detto che i moduli organizzano l'intero codice in blocchi accomunati per tipologia o per funzionalità. I moduli sono inoltre riutilizzabili, ecco perché AngularJS ne prevede molti già implementati e pronti per l'utilizzo. Esistono ad esempio moduli per: effettuare la localizzazione; gestire le animazioni; gestire la fase di test; interagire con un web server RESTful; gestire i cookie. Fra i più importanti troviamo il modulo per la gestione del *routing* di una applicazione.

Con routing ci si riferisce alla funzionalità grazie alla quale è possibile suddividere la Single Page Application i più view, ognuna delle quali può essere poi caricata al momento giusto, ed essere associata ad uno o più controller (vedi sotto-paragrafi 6.1.3 e 6.1.5). Questo meccanismo di gestione delle rotte avviene tramite specifici metodi e servizi.

6.1.3 Controller

I controller di AngularJS sono dei particolari oggetti che, associati a gruppi di elementi all'interno del DOM, controllano i dati in essi contenuti.

Ad ogni istanza di un oggetto controller viene associato un child scope (vedi sotto-paragrafo 6.1.7), poi iniettato nel costruttore del controller.

Poiché un controller crea le variabili richieste nella parte dell'html che questi gestisce, il numero e l'organizzazione degli stessi all'interno

del codice di dell'intera applicazione è in parte dettata dai requisiti della stessa ed in parte è a discrezione dell'utente. Non esistono infatti vincoli sul come e quanti controller distribuire all'interno di una pagina html. È possibile infatti creare uno solo di questi oggetti, oppure tanti, che possono essere anche disposti in modo gerarchico. È tuttavia sconsigliato creare controller troppo ampi o contenenti la gestione di feature differenti, che renderebbero il codice poco leggibile e manutenibile.

Un controller ben scritto dovrebbe infatti contenere la business logic di una sola view. Ci sono poi tutta una serie di operazioni sul DOM che sarebbe bene non gestire all'interno di un controller, come ad esempio: la presentation logic; la condivisione di codice fra controller; i format input; il filtraggio dell'output; la gestione del ciclo di vita dei componenti [23]. Questi compiti infatti devono essere ripartiti fra direttive e service, come vedremo a breve.

La seguente figura mostra un estratto di codice nel quale viene definito ed utilizzato un controller AngularJS:

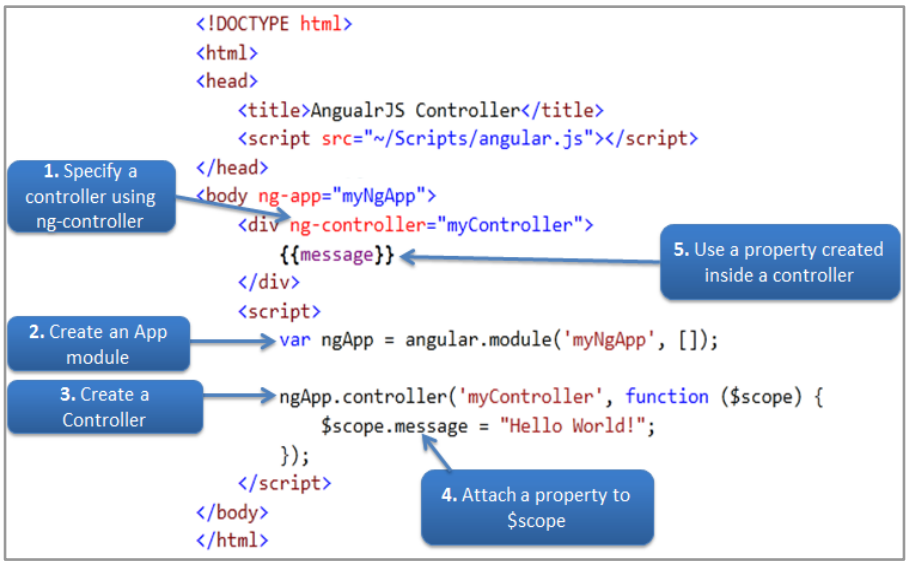


Figura 6.3: Creazione ed utilizzo di un controller AngularJS.

La figura mostra come:

- Associare un controller ad elementi del DOM tramite la direttiva *ng-controller*;
- Creare il modulo principale dell'applicazione, ovvero l'*App module*;
- Creare un controller;

- Associare la proprietà *message* allo scope;
- Usare la proprietà *message,* creata nel controller, all'interno del DOM utilizzando le doppie parentesi graffe.

Molti degli elementi appena elencati saranno descritti nel dettaglio nei seguenti sotto-paragrafi.

6.1.4 Service

I service di AngularJS sono degli oggetti utili alla condivisione e all'organizzazione del codice di un'applicazione. Questi vengono legati fra loro tramite un meccanismo chiamato *Dependency Injection*. I service godono di due particolari caratteristiche:

1. **Singleton**: un service factory è preposto alla generazione delle singole istanze di un service. Queste rappresentano un riferimento in ingresso ai componenti che utilizzano quel determinato service;
2. **Lazily instantiated**: come il nome stesso suggerisce, i service vengono istanziati in maniera pigra, ovvero solo nel momento in cui un componente ne necessita l'utilizzo;

Esistono svariati service che AngularJS mette già a disposizione per l'adempimento di alcuni servizi. Questi sono ben riconoscibili all'interno del codice poiché il loro nome è accompagnato sempre dal carattere $ posto all'inizio.

Una delle prime cose che lo sviluppatore deve decidere, prima di creare un service, è a quale tipologia fare affidamento per l'ottenimento del task desiderato. Esistono infatti differenti metodologie di creazione di un service [24], che sono:

1. **Constant**: utilizzato per iniettare valori costanti nell'applicazione a run time. La differenza rispetto ai value (vedi punto 5) è che i constant non necessitano di un provider (vedi punto 3);
2. **Factory**: permette di creare un service senza la necessità di definire esplicitamente un provider, che viene però creato automaticamente da AngularJS;

3. **Provider**: per istanziare le altre tipologie di service a run time. In particolare questi sono oggetti che prevedono un metodo *get()*, chiamato dall'oggetto *injector* all'atto di creazione di un service. Tutti i service vengono costruiti da un provider, fatta eccezione per i constant;
4. **Service**: permette di creare un service senza la necessità di definire esplicitamente un provider, che viene però creato automaticamente da AngularJS. Il suo comportamento sembra identico a quello dei *factory*, ma esiste una differenza relativa al modo in cui gli oggetti vengono istanziati. Service infatti prevede la possibilità di creare una gerarchia di oggetti, e quindi di sfruttare il principio di ereditarietà;
5. **Value**: utilizzato per iniettare valori costanti nell'applicazione a run time. La differenza rispetto ai constant è che i value vengono creati dal provider;

6.1.5 View

Abbiamo accennato precedentemente al fatto che AngularJS permette di creare applicazioni dinamiche estendendo l'HTML tramite direttive. Questo è possibile grazie alla presenza di un componente, *l'HTML Compiler*, che consente di arricchire l'HTML di nuovi elementi e attributi, ed anche di modificare il comportamento di quelli già previsti dal linguaggio. Sono le direttive, compilate dall'HTML Compiler, a svolgere queste nuove funzioni.

Per capire come avviene questo processo è necessario analizzare nel dettaglio tutte le entità appena citate. Cominciamo quindi dalla compilazione, che in questo contesto assume un significato differente rispetto a quello usato quando si parla di compilazione di un linguaggio di programmazione. In questo caso si intende il processo tramite il quale AngularJS lega un template HTML con le direttive in esso contenute. Il termine compilazione viene utilizzato perché questa operazione avviene in maniera ricorsiva, analogamente a quanto accade nella compilazione di codice sorgente [25]. Il *Compiler*, che è il componente chiave in questo processo, è un service di AngularJS. Per svolgere il suo compito il *Compiler* opera due task:

1. Attraversa l'intero DOM alla ricerca di direttive. Il risultato di questa operazione prende il nome di *linking function*;

2. Prende tutte le direttive collezionate al passo 1 e le lega allo scope giusto. Vengono così a crearsi le live-view, ovvero le view riflettono i cambiamenti dello scope model e viceversa;

Per comprendere appieno la potenza delle direttive facciamo qualche esempio pratico. Supponiamo di ricevere dal server una lista di elementi, di dimensione variabile, ognuno dei quali deve essere mostrato a video in un box separato. L'html non ci permette di svolgere questa operazione, in quanto statico. Sarebbe quindi necessaria la creazione di uno script. Esiste una direttiva AngularJS, la quale permette istantaneamente di clonare gli elementi del DOM nei quali è inserita, tante volte quanti sono gli item di una collezione. Questa direttiva prende il nome di *ng-repeat*.

Le direttive sono quindi dei marker che specificano al compiler come e quali elementi del DOM modificare.

Così come visto per i controller ed i service, anche le direttive di AngularJS possono essere o quelle già previste dal framework, come *ng-repeat*, oppure create dallo sviluppatore per specifiche esigenze. In quest'ultimo caso AngularJS mette a disposizione quattro differenti categorie di direttive custom. Queste sono:

1. **Attribute**: direttive che si comportano come gli attributi dei tag html;
2. **Comment**: direttive innestate nei commenti;
3. **CSS**: direttive passate come classi CSS;
4. **Element:** direttive che costituiscono nuovi elementi html, ovvero nuovi tag;

6.1.6 Data binding

In AngularJS con *data binding* si indica la sincronizzazione automatica dei dati, prevista dal framework, fra *model* e *view*. Da un punto di vista pratico il data binding si traduce nel fatto che quando cambia il model, automaticamente questo cambiamento si riflette anche nella view, e viceversa.

Il data binding può avvenire ad una via o a due vie: in questi casi si parla rispettivamente di *One-Way Data Binding* e *Two-Way Data Binding*.

Per comprendere le differenze che distinguono questi due approcci, consideriamo le due figure seguenti, che ne mostrano il rispettivo flusso logico:

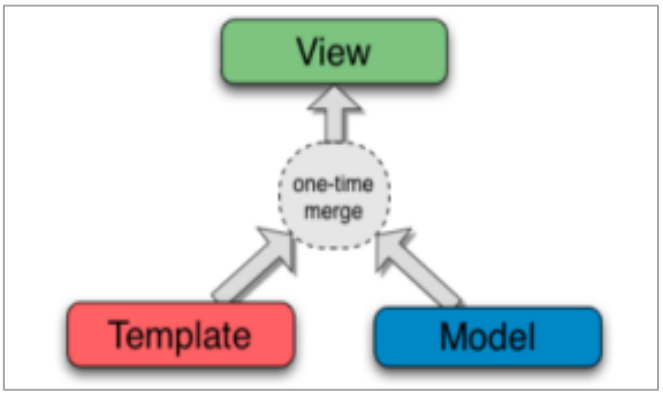


Figura 6.4: One-Way Data Binding

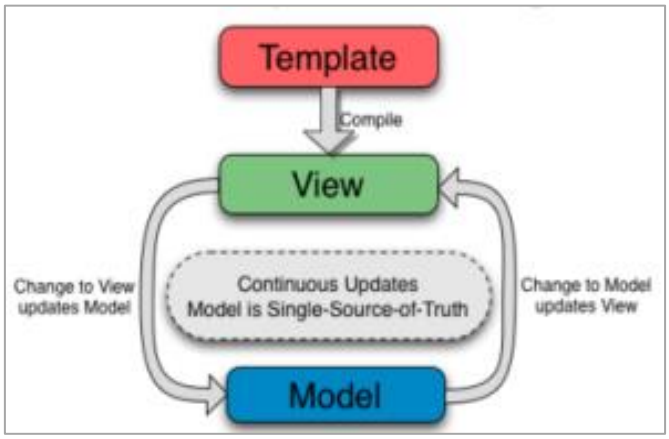


Figura 6.5: Two-Way Data Binding

Come possiamo notare dalla Figura 6.2, il *one-way data binding* prevede che la sincronizzazione tra modello e view venga esplicitamente gestita dal programmatore.

Nel *two-way data binding*, presente in AngularJS e mostrato in Figura 6.3, la situazione è completamente differente. Prima di tutto troviamo il *template*, ovvero il codice html non compilato, con al suo interno le direttive di Angular. Il browser compila il template, fornendo in uscita la view, una sorta di visualizzazione real-time dello stato corrente del modello associato. Quindi ogni variazione che avviene su

questa view viene automaticamente riportata sul modello corrispondente, e viceversa.

L'importanza del data binding è difficilmente comprensibile tramite la descrizione verbale del suo funzionamento. Questa proprietà infatti può essere capita appieno solo in fase di utilizzo. Tuttavia è importante sapere che rappresenta uno degli aspetti che più contribuiscono alla potenza del framework AngularJS. Ad esempio, il fatto di poter fare delle verifiche su un controller in modo isolato, poiché questi risulta essere completamente separato dalla view, permette alla fase di testing dell'applicazione di essere estremamente più semplice e produttiva. La connessione fra view e relativo model viene svolta usando uno dei due seguenti modi:

1. inserimento nella view della direttiva *ng-model*;
2. utilizzo delle espressioni Angular, che sono frammenti di codice delimitati da due parentesi graffe;

6.1.7 Lo scope

Lo scope in Angular rappresenta l'unico punto di contatto fra view e controller. In particolare ognuno di questi oggetti, la cui notazione nel codice è $scope, possiede le variabili e le funzioni definite all'interno dell'applicazione. Lo scope quindi rappresenta il contesto nel quale vengono valutate le espressioni e salvati i dati.

Un'applicazione AngularJS possiede un solo oggetto scope che fa da root, tuttavia possono esserci altri scope, che prendono il nome di child o isolate, e possono essere innestati. In tal caso i child scope erediteranno le proprietà dei parent scope, gli isolate scope invece non ereditano proprietà in quanto sono, appunto, isolati. Questi oggetti sono quindi organizzati in una struttura ad albero.

Per fare un esempio pratico poniamoci nella situazione in cui all'interno del DOM ci sia un'espressione. Quando AngularJS deve valutarla, per prima cosa cerca nello scope al livello più basso della gerarchia in cui è presente l'espressione. Se nello scope corrente non è possibile risolverla allora si risale lungo l'albero verso il parent scope. Questo processo continua fino alla risoluzione dell'espressione e può arrivare fino al root scope. Inoltre gli scope possono:

- propagare eventi;
- controllare le espressioni;
- propagare le modifiche a model e view;

- osservare le mutazioni del model;

6.2 P5.js

P5.js è una libreria JavaScript open source, nata per offrire le stesse funzionalità di Pocessing, come la creazione di animazioni e opere d'arte digitali, utilizzando tuttavia un linguaggio ed un paradigma di implementazione differenti.

Ricordiamo che Processing è un linguaggio di programmazione orientato agli oggetti, che eredita tutta la sua sintassi da Java ma che prevede l'aggiunta di molte funzioni ad alto livello specifiche per la creazione di elementi grafici e multimediali.

P5.js si propone come una libreria per il web, potente ma semplice da utilizzare, che rende le sue funzionalità accessibili anche ai programmatori alle prime armi. È quindi pensato per i designer, per gli artisti digitali e per tutte le moderne applicazioni.

P5.js ha un set vastissimo di funzionalità per il disegno, si può infatti affermare che è possibile, grazie a questa libreria, realizzare qualsiasi tipo di animazione o disegno supportabili dal browser.

Anche se il nostro utilizzo di p5.js per Stardust è stato su canvas, dobbiamo sottolineare che è possibile disporre di tutte le funzionalità di questa libreria anche al di fuori di questo specifico elemento HTML. Stiamo dicendo quindi che è possibile creare disegni ed animazioni di ogni tipo su tutta la pagina del browser. Inoltre è possibile sovrapporre e far interagire le proprie creazioni con tutti gli elementi previsti dall'HTML, quali testi, figure, campi di input, video e suoni [26].

6.2.1 Come iniziare

La creazione di un nuovo progetto p5 è un processo estremamente semplice, e può essere utilizzato qualunque editor per la scrittura di codice. Infatti, una volta impostato l'index.html, basta aggiungere a questi un tag *script* con la specifica del link al file p5.js, disponibile anche in versione minificata. È inoltre possibile reperire tutte le versioni della libreria tramite CDN (Content Delivery Network).
Successivamente deve essere aggiunto lo script sketch.js, ovvero il file nel quale l'utente scriverà il suo codice. In particolare il file sketch.js contiene due funzioni da implementare.

Queste sono:

1. **Setup()**: funzione nella quale inserire il codice relativo al set up del disegno. Questa funzione sarà eseguita una sola volta, prima della funzione draw();

2. **Draw()**: funzione nella quale inserire il codice per disegnare. Questa viene eseguita subito dopo quella di setup() e ad intervalli temporali configurabili. Di default i disegni vengono aggiornati con una frequenza di 60 volte al secondo;

Nel prossimo sottoparagrafo sarà mostrato un esempio di utilizzo della libreria p5.js per la realizzazione di un semplice disegno su canvas.

6.2.2 Un esempio di utilizzo

In questa sezione mostreremo un esempio di utilizzo di p5.js per la realizzazione un semplice disegno, utile a dimostrare le potenzialità è l'immediatezza di utilizzo di questa libreria.

Supponiamo di voler disegnare un cerchio. Nel sotto-paragrafo precedente abbiamo mostrato le due funzioni del file sketch.js. Per disegnare un cerchio basta inserire una sola linea di codice all'interno della funzione draw().

La figura seguente mostra il codice suddetto ed il relativo risultato:

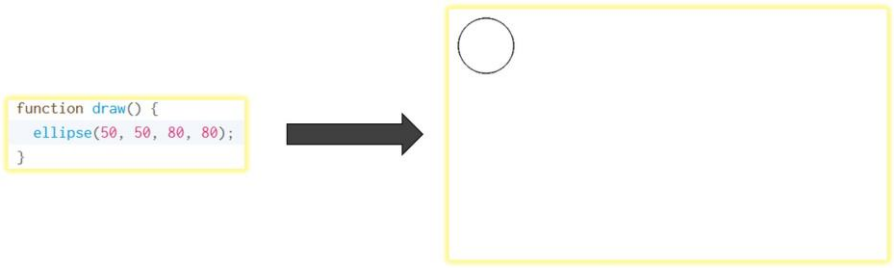


Figura 6.6: codice e risultato per il disegno di un cerchio in P5.js

Nella parte sinistra dell'immagine è mostrato il codice necessario al disegno di un cerchio. Questi consta di una sola istruzione, viene infatti chiamata la sola funzione *ellipse* di p5 per il disegno di una ellisse. Questa riceve quattro parametri: la distanza in pixel dal centro dell'ellisse al lato sinistro della schermata; la distanza in pixel dal centro dell'ellisse al lato superiore della schermata; larghezza in pixel dell'ellisse; altezza in pixel dell'ellisse. Poiché questi due ultimi valori

sono stati posti uguali, il risultato sarà un cerchio, mostrato a destra della figura.

Facciamo ora un passo in avanti, aggiungendo al nostro disegno un po' di dinamicità. Supponiamo infatti di voler tracciare lo stesso cerchio di prima ma su di un canvas, e non solo. Vogliamo anche che venga disegnato un cerchio in ogni punto in cui si trova il mouse, e che il click del tasto sinistro di quest'ultimo trasformi l'area interna del cerchio da bianca a nera. La seguente figura mostra il codice ed il risultato relativi agli output appena descritti:

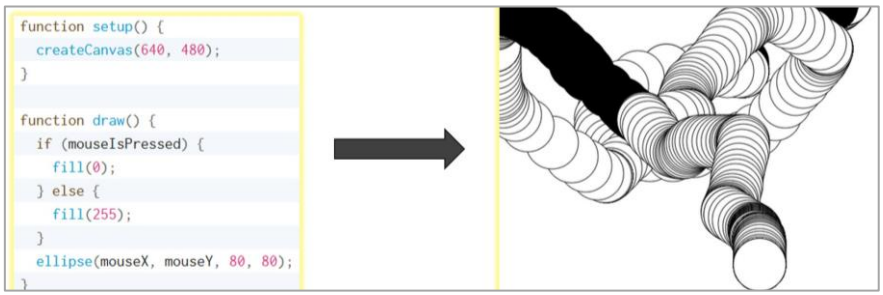


Figura 6.7: codice e risultato dell'animazione di un cerchio in P5.js

Possiamo notare che il codice relativo alla creazione del canvas e alle animazioni è estremamente conciso e comprensibile.

Troviamo prima di tutto, in setup(), la creazione del canvas con la specifica di larghezza ed altezza in pixel.

Nella funzione draw() è stato aggiunto un costrutto condizionale per la verifica del click del mouse, con relativa impostazione del cambiamento di colore dell'area del cerchio.

Il risultato, a destra della figura, mostra chiaramente la traiettoria percorsa dal mouse sul canvas ed in quali punti è avvenuto il click. Questo esempio ci rivela chiaramente come la realizzazione di disegni ed animazioni con p5.js richieda più sforzo per la descrizione verbale di ciò che si vuole fare che per l'implementazione.

6.2.3 La community p5.js

P5.js può vantare una community di sviluppatori estremamente vasta ed attiva. È infatti possibile trovare non solo migliaia di esempi riutilizzabili, ma anche centinaia di moduli per arricchire p5 di specifiche funzionalità.

Fra queste menzioniamo:

- **p5.dom**: consente di interagire con oggetti HTML5 quali video, audio, webcam, input e testo;
- **p5.sound**: consente di accedere alle funzionalità Web Audio, incluso l'ingresso audio, la riproduzione, l'analisi e la sintesi;
- **p5.collide2D**: creato da Ben Moren, fornisce strumenti per il calcolo della rilevazione di collisioni per la geometria 2D;
- **p5.geolation**, creato da Ben Moren, fornisce tecniche per acquisizione, osservazione, calcolo e geofencing delle posizioni degli utenti;
- **p5.dimensions**: estende le funzioni vettoriali di p5 per lavorare con qualsiasi numero di dimensioni. Creato da Smilebags e Max Segal;
- **p5.gui**: per la generazione di interfacce grafiche. Creato da Martin Schneider;
- **p5.particle**: per creare effetti particle e fountain basati su strutture dati che possono provenire da input JSON o definite dall'utente. Creato da Robert Cook;

Possiamo notare che la maggior parte di queste librerie aggiuntive è stata sviluppata dalla community.

È inoltre previsto a breve il rilascio di un editing dedicato allo sviluppo con p5.js.

6.3 Conclusioni

In questo capitolo abbiamo analizzato le due tecnologie utilizzate per lo sviluppo del front end della piattaforma Stardust: AngularJS e p5.js. Abbiamo visto che AngularJS è un framework molto potente, le cui caratteristiche permettono di creare applicazioni web in modo estremamente modulare.

Con AngularJS è possibile aderire al modello ReST, discusso nel capitolo 5, relativo all'application back end. Inoltre il server è sgravato dalla generazione delle view, un vantaggio questo che riduce notevolmente il transito di dati sulla rete, così da poter beneficiare di risposte più veloci che nell'ambito di un editor online risultano di importanza cruciale.

Successivamente, all'interno di AngularJS, è stata wrappata la libreria p5.js per la realizzazione dei disegni e delle animazioni previste sul foglio di lavoro si Stardust.

Nel corso di questo capitolo abbiamo avuto modo di mostrare come p5.js metta a disposizione tutte le funzionalità necessarie per la creazione di elementi grafici su pagine web.

La combinazione di queste due tecnologie è stata per noi la scelta vincente per la realizzazione di una piattaforma eterogenea come Stardust. Sia AngularJS che p5.js sono dei prodotti che possono vantare la caratteristica importantissima di essere completi, permettendo quindi lo sviluppo da zero di applicazioni web complesse.

Con questo capitolo si chiude la parte II del presente lavoro di tesi, relativa agli aspetti teorici di base delle tecnologie utilizzate per lo sviluppo di Stardust. Sottolineiamo che in questa parte sono stati volutamente omessi esempi di codice, per concentrarci maggiormente sugli aspetti concettuali. Rimandiamo al capitolo 7, relativo alla fase di implementazione, esempi di codice relativi ai richiami teorici dati in questa parte.

Parte III

L’implementazione

Capitolo 7

Lo sviluppo di Stardust

In questo capitolo presenteremo alcuni estratti di codice relativi all'implementazione di Stardust. Tali esempi sono stati scelti con lo scopo di mostrare le caratteristiche principali delle tecnologie affrontate nella parte II. La nostra descrizione quindi non coprirà l'intero sviluppo, ma cercherà invece di riportare alcune parti salienti a puro titolo esemplificativo. Consideriamo inoltre che, anche se già completamente utilizzabile, prevediamo per la piattaforma Stardust l'inserimento di nuove funzionalità ed ottimizzazioni, come spiegheremo meglio nel capitolo 8. Per tale ragione puntualizziamo che la fase di sviluppo non può dirsi ancora conclusa.

7.1 La modellazione dei dati

Nel corso del capitolo 3 abbiamo avuto modo di mostrare alcuni aspetti teorici relativi alle tecnologie utilizzate per la gestione dei dati: MongoDB e Neo4j. In relazione a quest'ultimo abbiamo detto che il notevole vantaggio proveniente dall'adozione di questa tecnologia risiede nel fatto che la teoria dei grafi, applicata al calcolo di shortest path problem, come previsto in Stardust, porta una serie notevole di vantaggi. In particolare il più grande è rappresentato dal fatto che con Neo4j abbiamo potuto disporre di un'organizzazione dei dati del tutto sovrapponibile al loro reale contenuto informativo previsto nei progetti Stardust.

Vediamo dunque come queste affermazioni sono state realmente modellate nella realtà:

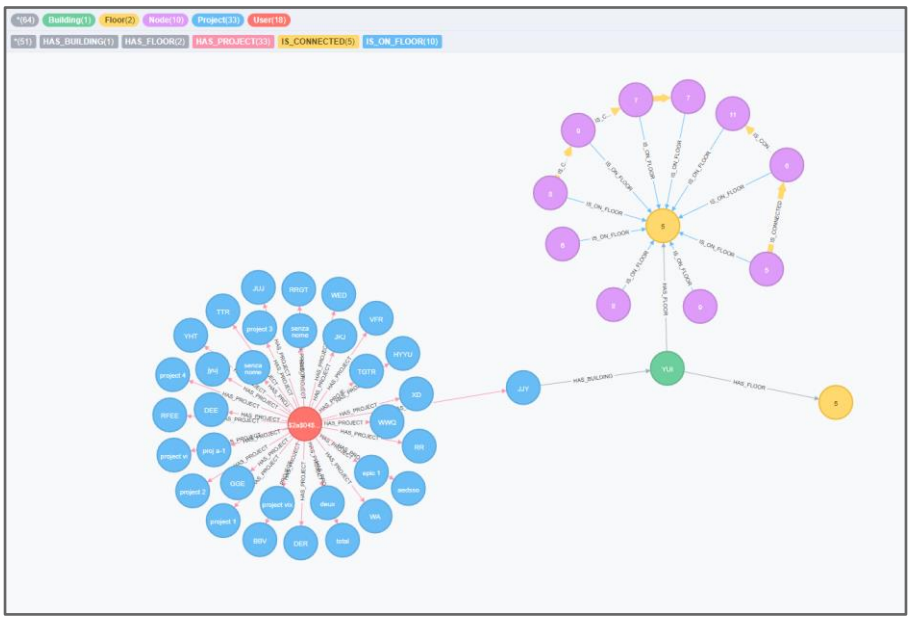


Figura 7.1: Nodi e loro relazioni in Neo4j

L'immagine sopra riportata mostra la console di Neo4j con all'interno la rappresentazione grafica di un tipico progetto Stardust. Questa consta di un grafico nella quale i nodi sono gerarchicamente collegati tramite archi, e tutti posseggono delle specifiche proprietà. Partendo dal livello più alto della gerarchia troviamo i nodi:

- Utente;
- Progetti dell'utente;
- Edifici di un progetto;
- Piani di un edificio;
- Nodi di un piano;

Per quanto concerne invece gli archi, questi sono caratterizzati da una proprietà che li distingue in:

- Archi di collegamento fra nodi;
- Archi di collegamento fra piani;

Neo4j è quindi uno strumento potentissimo nei casi in cui è prevista una forte relazione fra i dati. Infatti il modello mostrato è fatto in modo tale che da un utente si può arrivare a tutti i nodi di un grafo, e viceversa. Vediamo ora nel dettaglio le entità mostrate in figura ed alcune proprietà che, associate agli elementi del grafo, ne delineano il ruolo ed il comportamento:

- USER: nodo creato all'atto di iscrizione di un nuovo utente (nodo rosso in figura);
- PROJECT: nodo creato quando un utente aggiunge un nuovo progetto (nodi blu in figura);
- HAS_PROJECT: relazioni che legano il nodo USER ai suoi nodi PROJECT (frecce rosse in figura);
- BUILDING: nodo creato quando al progetto viene aggiunto un nuovo edificio (nodo verde in figura);
- HAS_BUILDING: relazioni che legano il nodo PROJECT ai suoi nodi BUILDING;
- FLOOR: nodo creato quando ad un edificio viene aggiunto un nuovo piano (nodo giallo in figura);
- HAS_FLOOR: relazioni che legano il nodo BUILDING ai suoi nodi FLOOR;

Infine ogni piano prevede al suo interno un ulteriore grafo, questa volta non gerarchico, che contiene proprio i nodi e gli archi inseriti dall'utente in fase di progettazione sul foglio di lavoro di Stardust.

Ogni piano può quindi avere solo uno di questi grafi, ed i suoi nodi e relazioni rappresentano rispettivamente i punti di interesse ed i percorsi del piano.

Nei grafi relativi ai piani troviamo:

- NODE: nodi creati quando sulla planimetria viene aggiunto un nuovo punto di interesse (nodi viola in figura);
- IS_CONNECTED: associata ad una relazione collega i due nodi ai suoi estremi (frecce gialle in figura);

- IS_ON_FLOOR: relazione che collega tutti i nodi di un piano allo stesso (frecce blu in figura);

7.1.1 Nodi e relazioni

Mostriamo ora i primi esempi di codice, relativi alla definizione di nodi di tipo USER, PROJECT, BUILDING, FLOOR e NODE:

Listato 7.1 Definizione di un nodo di tipo USER

```
{
 "records": [
   {
     "keys": [
       "n"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 1232,
           "high": 0
         },
         "labels": [
           "User"
         ],
         "properties": {
           "password":
"$2a$04$o9Ysge37o2EKAu9kSI8uFu9Gi4IcI4Akw8R4g4Gyj
0AFxsLN4wOOy",
           "id": "4ff3141b-1fc3-48a5-8c4b-
963ec2b2afc2",
           "email": "asd@asd.it"
         },
         "id": "1232"
       }
     ],
     "_fieldLookup": {
       "n": 0
     }
```

```
    },
...
```

Questo codice rappresenta un estratto del risultato di una query che restituisce un array di oggetti rappresentanti gli utenti di Stardust. I capi previsti sono:

- *records*: contiene tutti i record che rappresentano l'oggetto USER;
- *length*: numero di campi previsti nella return della query;
- *_fields*: contiene tutti i dati di un utente, quali la sua etichetta, la password, l'email e l'id;
- *_fieldLookup*: mappa i campi della query a numeri, partendo da zero;

Listato 7.2 Definizione di un nodo di tipo PROJECT

```
{
 "records": [
   {
     "keys": [
       "n"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 0,
           "high": 0
         },
         "labels": [
           "Project"
         ],
         "properties": {
           "deleted": false,
           "name": "project 1",
```

```
            "created_at": 1495463859311,
            "id": "954dac67-7f08-4803-9eb1-
d10e394e953c"
          },
          "id": "0"
        }
      ],
      "_fieldLookup": {
        "n": 0
      }
    },
…
```

Nel caso di nodo PROJECT è prevista anche l'etichetta di deleted, che indica se il nodo è stato cancellato (alla creazione deleted sarà uguale a false), quella del nome del progetto, la data di creazione e l'id univoco.

Listato 7.3 Definizione di un nodo di tipo BUILDING

```
{
 "records": [
   {
     "keys": [
       "n"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 1,
           "high": 0
         },
         "labels": [
           "Building"
         ],
         "properties": {
           "name": "building 1",
           "created_at": 1495463888329,
```

```
            "id": "d062a7a9-fd70-4220-9436-
afa0f8fd76ea"
          },
          "id": "1"
        }
      ],
      "_fieldLookup": {
        "n": 0
      }
    },
...
```

Quindi per un nodo rappresentante un edificio i campi principali sono quelli relativi al suo nome, alla data di creazione e all'id.

Listato 7.4 Definizione di un nodo di tipo FLOOR

```
{
 "records": [
   {
     "keys": [
       "n"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 2,
           "high": 0
         },
         "labels": [
           "Floor"
         ],
         "properties": {
           "number": "1",
           "id": "c32e3be8-9f58-4638-baa8-
068dbaaf566e"
         },
         "id": "2"
       }
```

```
      ],
      "_fieldLookup": {
        "n": 0
      }
    },
…
```

I campi principali di un nodo di tipo piano sono il suo numero ed il suo id.

Listato 7.5 Definizione di un NODE del grafo di un piano

```
{
 "records": [
   {
     "keys": [
       "n"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 3,
           "high": 0
         },
         "labels": [
           "Node"
         ],
         "properties": {
           "border": 153,
           "yOffset": 0,
           "deleted": false,
           "xOffset": 0,
           "Etikett_lenght" : 4
           "size": 15,
           "etikett": "AULA",
           "x": 132.78801843317973,
           "name": 0,
           "y": 226,
```

```
            "id": "ee955a41-e3b2-4def-8728-
70859b019398",
            "locked": false,
            "type": "generico",
            "status": false
          },
          "id": "3"
        }
      ],
      "_fieldLookup": {
        "n": 0
      }
    },
...
```

Come già anticipato nel capitolo 2, in relazione alla presentazione delle funzionalità dell'editor di Stardust, i nodi del grafo di un piano rappresentano dei punti di interesse della struttura, e vengono inseriti direttamente dall'utente sul foglio di lavoro. Quindi in questi elementi devono essere salvate tutte le informazioni utili sia al corretto funzionamento della navigazione indoor, sia al loro ritracciamento nella posizione corretta della planimetria quando un progetto viene salvato, chiuso e riaperto. Questi dati sono:

- *border*: colore del bordo espresso in scala di grigi;
- *xOffset* e *yOffset*: offset relativo allo spostamento di un nodo (tramite drag and drop) dal suo punto di creazione;
- *deleted*: flag che indica se il nodo è stato cancellato;
- *size*: dimensione del nodo in pixel, impostabile dall'editor di Stardust;
- *etikett*: etichetta associata al nodo dall'utente;
- *etikett_lengh*: lunghezza (in caratteri) dell'etichetta di un nodo, utile al dimensionamento dinamico del suo container sul canvas dell'editor di Stardust;
- *x* e *y*: coordinate del nodo sul piano del canvas, dove l'origine (0,0) e posta nel punto in alto a sinista;
- *status*: flag che indica se un nodo è selezionato oppure no;

- *type*: indica il tipo di nodo che, come indicato nel capitolo 2, può essere generico, porta, scale, ascensore, TAG, toilet, ristoro;
- *name*: numero che viene automaticamente associato ai nodi di un grafo all'atto della loro creazione. Questo valore parte da zero e cresce al crescere degli inserimenti di nuovi nodi di uno stesso piano;

Passiamo adesso ad un esempio di codice relativo alla definizione di un arco. L'relazione in questione collega due nodi di un piano.

Listato 7.6 Definizione di un arco IS_CONNECTED


```
{
 "records": [
   {
     "keys": [
       "r"
     ],
     "length": 1,
     "_fields": [
       {
         "identity": {
           "low": 4,
           "high": 0
         },
         "start": {
           "low": 6,
           "high": 0
         },
         "end": {
           "low": 3,
           "high": 0
         },
         "type": "IS_CONNECTED",
         "properties": {
           "xEndRect1": 125.29385260867981,
           "xStartRect1": 119.29495859946323,
           "xEndRect2": 140.28218425767966,
           "yStartRect2": 378.29576780571495,
```

```
            "yEndRect2": 226.29576780571492,
            "yStartRect1": 377.70423219428505,
            "isDrawing": false,
            "color": 20,
            "yEndRect1": 225.70423219428508,
            "xStartRect2": 134.28329024846306,
            "shift": false,
            "end_node_id": "ee955a41-e3b2-4def-
8728-70859b019398",
            "length": 152.11833133890806,
            "started": true,
            "finished": true,
            "start_node_id": "9aab35af-6e78-4a70-
bb27-d9a8b1756d05",
            "deleted": false,
            "angle": -1.5313503906350778,
            "id": "875eb7e0-6abe-48d5-af38-
14637ced6b06",
            "locked": false,
            "status": false
          },
          "id": "4",
          "startNode": "6",
          "endNode": "3"
        }
      ],
      "_fieldLookup": {
        "r": 0
      }
    },
```


Poiché una relazione nasce e termina fra due nodi, questi devono essere specificati. Troviamo quindi uno *startNode* ed un *endNode*, contenenti rispettivamente gli identificativi del nodo di partenza e di quello di arrivo dell'arco in questione.

Sono inoltre presenti otto coordinate che specificano il posizionamento delle due linee con le quali viene rappresentato un arco.

7.1.2 Le sessioni

Presentiamo adesso un documento MongoDB, che ricordiamo viene utilizzato per il salvataggio delle session della piattaforma web.

Listato 7.7 Session con MongoDB

```
{
        "_id" : "YsAJv5xCGcKSQmGA5EUpeaWJOFec8tcN",
        "session" :
"{\"cookie\":{\"originalMaxAge\":null,\"expires\":null
,\"httpOnly\":true,\"path\":\"/\"},\"passport\":{\"use
r\":\"5e7ebfed-3488-46f1-9e4e-bffb24ef0002\"}}",
        "expires" : ISODate("2017-10-05T15:30:31.502Z")
}
```

Troviamo quindi:

- *_id*: id dell'utente, lo stesso utilizzato da Neo4j;
- *expires*: data di scadenza della session;

7.2 La configurazione del back end

Una delle principali caratteristiche di Node.js è la sua flessibilità. Esistono infatti svariati framework che possono essere affiancati a Node.js per creare più facilmente un'application back end. Fra tali framework quello da noi utilizzato è stato Express che, come già spiegato nel capitolo 5, ci ha permesso di usufruire del suo servizio di server-side routing e di sfruttare il middleware per l'intercettazione delle chiamate.

Abbiamo inoltre utilizzato una serie di npm-package, ad esempio per i driver di Neo4j e MongoDB, per l'autenticazione degli utenti e per il logging.

7.2.1 Express e punto di accesso alla piattaforma

Ogni progetto Node.js si compone di almeno due file. Il primo è package.json, il secondo è quello che fa da punto di ingresso dell'applicazione, che nel nostro caso prende il nome di app.js.

Di seguito sono mostrati i listati relativi a questi due file implementati nell'application back end di Stardust.

Listato 7.8 File package.json di Node.js

```
{
 "name": "stardust",
 "version": "1.0.0",
 "description": "online editor for indoor navigation",
 "main": "app.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit
1",
   "start" : "node app.js"
 },
 "author": "seyedshoya moosavikhorshidi - federica
ferro",
 "license": "ISC",
 "dependencies": {
   "bcrypt": "^1.0.2",
   "body-parser": "*",
   "connect-mongo": "^1.3.2",
   "cookie-parser": "^1.4.3",
   "express": "*",
   "express-session": "^1.15.2",
   "lodash": "^4.17.4",
   "mongoose": "^4.9.10",
   "morgan": "*",
   "multer": "^1.3.0",
   "neo4j-driver": "*",
   "passport": "^0.3.2",
   "passport-local": "^1.0.0",
   "sessionstore": "^1.2.18",
   "uuid": "^3.0.1"
 }
}
```

Il file package.json contiene quindi una sorta di descrizione dell'intero progetto. Inoltre nel campo *dependencies* vi è l'elenco di tutti gli npm-package che devono essere scaricati e installati nella cartella *npm_modules,* per il corretto funzionamento dell'intera applicazione.

Listato 7.9 File app.js di Node.js

```
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');
var neo4j = require('neo4j-driver').v2;
var mongoose = require('mongoose');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var passport = require('passport');
var apiController =
require('./controllers/apiController');
var userController =
require('./controllers/userController');
var port = require('./config/config').port;
var mongo_url = require('./config/config').mongo;
var MongoStore = require('connect-mongo')(session);
mongoose.connect(mongo_url);

var app = express();

//MIDDLEWARE
app.use(logger('dev'));
app.use(bodyParser.json({limit: '50mb'}));
app.use(bodyParser.urlencoded({ limit: '50mb',
extended : true}));
app.use(express.static('uploads'));
app.use(express.static(path.join(__dirname,
'public')));

//AUTENTICAZIONE
app.use(cookieParser());
app.use(session(
   {
       secret: 'anystringoftext',
       saveUninitialized: true,
       resave: true,
       //store user sessions in mongo
```

```
        store : new MongoStore(
            {
                mongooseConnection :
mongoose.connection,
                ttl : 10 * 24 * 60 * 60 // session is
valid for 10 days
            }
        )
    }));

app.use(passport.initialize());
app.use(passport.session()); // persistent login
sessions

//ROUTER
app.use('/api', apiController);
app.use('/user', userController);

//PORT LISTENING
app.listen(port, function(){
 console.log('server is running on port ' + port +
'...');
});

module.exports = app;
```

In app.js vengono:

- istanziate tutte le librerie esterne necessarie;
- introdotta l'istanza di express;
- inseriti i middleware necessari al logging delle attività;
- fornite le risorse statiche;
- gestito il caricamento di file;
- gestiti il salvataggio delle sessioni ed il routing;
- specificata la porta sulla quale il server è in esecuzione;

7.2.2 I controller

L'application back end di Stardust si compone di tre controller:

1. userController.js;
2. authController.js;
3. apiController.js;

Il primo, lo *userController*, importa prima di tutto la classe *Router* di Express e mette poi a disposizione quattro endpoint, che sono:

1. */user/getme*: metodo GET per recuperare lo stato corrente di un utente (se è loggato o meno);
2. */user/login*: metodo POST per effettuare il login di un utente;
3. */user/register*: metodo POST per la registrazione alla piattaforma web;
4. */user/logout*: metodo POST per effettuare il sign out;

Riportiamo di seguito, il codice relativo all'endpoint /user/login.

Listato 7.10 Endpoint /user/login di userController.js

```
router.post('/login',
   passport.authenticate('local-login'),
       function (req, res) {
           console.log('authentication succedded!');
           return res.json(req.user.properties);
       }
);
```

Specifichiamo che l'oggetto *passport* viene esportato dal controller authController, il quale gestisce l'autenticazione utilizzando il package Passport. Nello specifico *passport.authenticate* funziona da middleware. Questi, prima dell'esecuzione di *function (req, res)*, interviene per controllare se l'utente in questione esiste o meno nel database. Solo se l'utente è già presente nel database il middleware

passa il controllo alla funzione successiva, e quindi il login viene effettuato.

Passiamo ora al secondo controller, l'*authController*. Come il nome stesso suggerisce questo controller viene utilizzato per l'autenticazione degli utenti e per verificare se gli stessi hanno i permessi necessari per accedere alla pagina richiesta. Come già accennato prima, in questo controller viene utilizzata la modalità *passport-local* della libreria Passport. Riportiamo di seguito il codice relativo ad un middleware dell'authController.

Listato 7.11 Un middleware dell'authController

```
//AUTHENTICATE access
passport.ensureAuthentication = function(req, res,
next) {
   if(req.isAuthenticated()){
       return next();
   }
   res.status(403).json('you are not logged in!');
}
```

Il codice sopra riportato si riferisce alla funzione che interviene ad ogni chiamata fatta all'apiController, e che verifica se un utente può ottenere l'accesso richiesto.

Vediamo infine l'ultimo controller previsto nell'application back end di Stardust, ovvero l'*apiController*. Questi è responsabile di tutti gli endpoint relativi ai progetti Stardust ed ai loro grafi. È stato necessario implementare un numero elevato di questi endpoint, e di seguito ne sono stati riportati alcuni.

Nei seguenti stralci di codice sono stati tralasciati i dettagli implementativi relativi alle funzioni che eseguono le azioni vere e proprie, in quanto gli esempi riportati hanno il solo obiettivo di mostrare le varie API messe a disposizione dal server di Stardust.

Listato 7.12 POST per l'inserimento di una planimetria

```
//POST : api/add/image
router.post('/add/image/:floor_id',passport.ensureAuthentica
tion,
function (req, res){...}
```

Listato 7.13 GET per il recupero di una planimetria

```
//GET: get floor image
router.get('/get/image/:floor_id',
passport.ensureAuthentication,
function (req, res){...}
```

Listato 7.14 POST per l'inserimento di un piano

```
//POST: api/add/floor
router.post('/add/floor/:building_id',passport.ensureAuthent
ication,
function (req, res){...}
```

Listato 7.15 PUT per l'aggiornamento degli attributi di un piano

```
//PUT: api/update/floor
router.put('/update/floor/:floor_id',
passport.ensureAuthentication,
function (req, res){...}
```

Listato 7.16 POST per il collegamento fra piani

```
//POST: api/floor/connect
router.post('/floor/connect/:building_id',passport.ensureAut
hentication,
function (req, res){...}
```

Listato 7.17 POST per il salvataggio su db di un grafo

```
// POST: api/add/graph
router.post('/add/graph', passport.ensureAuthentication,
function(req, res){...}
```

Listato 7.18 PUT per la geolocalizzazione del progetto

```
//PUT : api/project/geoloc
router.put('/project/geoloc/:project_id',passport.ensureAuth
entication,
function (req, res){...}
```

Listato 7.19 GET per recuperare la posizione del progetto

```
//GET: api/project/geoloc
router.get('/project/geoloc/:project_id',passport.ensureAuth
entication,
function (req, res){...}
```

Listato 7.20 GET per recuperare i progetti di un utente

```
// GET: api/projects
router.get('/projects', passport.ensureAuthentication,
function (req, res, next){...}
```

Listato 7.21 POST per creare un nuovo progetto

```
// POST: api/add/project
router.post('/add/project', passport.ensureAuthentication,
function(req, res){...}
```

Listato 7.22 PUT per aggiornare un progetto

```
// PUT: api/update/project
router.put('/update/project', passport.ensureAuthentication,
function(req, res){...}
```

Listato 7.23 PUT per eliminare un progetto

```
// PUT: api/delete/project/:id
router.put('/delete/project/:project_id',passport.ensureAuth
entication, function(req, res){...}
```

Listato 7.24 POST per creare un edificio

```
// PUT: api/delete/project/:id
router.put('/delete/project/:project_id',passport.ensureAuth
entication, function(req, res){...}
```

Listato 7.25 POST per creare un edificio

```
// POST: api/add/building
router.post('/add/building', passport.ensureAuthentication,
function(req, res){...}
```

Listato 7.26 PUT per aggiornare un edificio

```
// PUT: api/update/building
router.put('/update/building',
passport.ensureAuthentication,
function(req, res){...}
```

Listato 7.27 GET per recuperare gli edifici di un progetto

```
//GET: api/building
router.get('/building/:project_id',
passport.ensureAuthentication,
function (req, res){...}
```

Listato 7.28 GET per recuperare i piani di un edificio

```
//GET: api/floor
router.get('/floor/:building_id',
passport.ensureAuthentication,
function (req, res){...}
```

Listato 7.29 GET per recuperare un progetto subito dopo la sua creazione

```
//GET: api/project/name
router.get('/project/name/:project_id',
passport.ensureAuthentication, function (req, res){...}
```

Listato 7.30 GET per recuperare il grafo di un piano

```
//GET: api/graph
router.get('/graph/:floor_id',
passport.ensureAuthentication,
function (req, res){...}
```

Listato 7.31 GET per recuperare tutti i dati di un edificio in una sola chiamata

```
//GET: api/building/data
router.get('/building/data/:id',
passport.ensureAuthentication, function (req, res){...}
```

7.2.3 I models

I models di Node.js sono dei servizi che, tramite il driver, possono accedere al database in lettura ed in scrittura. L'application back end di Stardust ha due modelli, che sono:

1. Graph.js: servizio responsabile di tutti gli accessi in lettura o in scrittura al database, relativi ai dati dei progetti e alle azioni svolte nell'editor;
2. Users.js: servizio responsabile di tutti gli accessi in lettura o in scrittura al database, relativi ai dati degli utenti;

Di seguito sono riportate due funzioni dei models appena descritti.

Listato 7.32 funzione di Graph.js per la creazione di un nuovo progetto

```
//CREATE NEW PROJECT
graph.createProject = function(email, project_name,
callback) {
   return session
       .run('CREATE (p:Project {created_at:
{created_at}, updated_at: {updated_at}, name :
```

```
{project_name}, id : {p_id}, deleted : false}) RETURN
p',
            {
                created_at: new Date().getTime(),
                updated_at: new Date().getTime(),
                project_name: project_name ,
                p_id : uuid.v4()
            })
        .then(function (result) {
            if(!_.isEmpty(result.records)){
                return session
                    .run("MATCH (u:User {email:
{email}}), (p:Project {id: {p_id}}) CREATE (u)-
[:HAS_PROJECT]->(p) RETURN p",
                        {
                            p_id :
result.records[0]._fields[0].properties.id,
                            email : email
                        }
                    )
                    .then(function (result) {
                        return callback(null,
result.records[0]._fields[0].properties);
                    })
                    .catch(function (err) {
                        if(err) throw err;
                    })

            }else{
                callback({msg : 'could not create the
project', status : 400}, null);
            }
        })
        .catch(function (err) {
            if(err) throw err;
        })
}
```

La funzione appena riportata per prima cosa crea un nodo di tipo PROJECT. Successivamente crea una relazione di tipo HAS_PROJECT fra il nodo appena creato ed il nodo relativo all'utente del progetto. Infine le due entità generate vengono salvate sul database.

Listato 7.33 funzione di Users.js per la restituzione di un utente

```
//GET USER BY EMAIL
users.getUserByEmail = function (email, callback) {
   return session
        .run('MATCH (user:User {email: {email}}) RETURN
user', {email: email})
        .then(function(result){
            if(!_.isEmpty(result.records)){
                callback(null,
result.records[0].get('user'));
            }else{
                callback({msg : 'email does not
exist!', status : 400}, null);
            }
        })
        .catch(function (err) {
            if(err) throw err;
        })
}
```

La funzione appena riportata controlla se un utente, dato il suo id, è presente oppure no nel database.

7.3 Il front-end

Come visto nel capitolo 6, il front-end di Stardust è stato costruito tramite l'adozione di due tecnologie, entrambe librerie JavaScript, che sono AngularJS e p5.js.

Inoltre, data la natura di *Single Page Application* di Stardust, il routing dell'intera applicazione web è a carico del client. A tal scopo è stata utilizzata la libreria *ui-router* per AngularJS. Di seguito è riportato un estratto del file *router.js*.

Listato 7.34 Documento di router.js

```
angular.module('stardust.application')
   .config(['$stateProvider', '$urlRouterProvider',
'$httpProvider' ,'$locationProvider', function
```

```
($stateProvider, $urlRouterProvider, $httpProvider,
$locationProvider) {

        $httpProvider.interceptors.push(interceptor);
        $urlRouterProvider.otherwise('/Dashboard');

        $stateProvider
            .state('login',{
                url: '/login',
                templateUrl: '/partials/home.html',
                controller: 'homeController',
                controllerAs: 'hmc',
            })
            .state('Starcode',{
                url: '/Starcode',
                templateUrl: '/partials/starcode.html',
                controller: 'starcodeController',
                controllerAs: 'scc',
            })
            .state('Dashboard',{
                url: '/Dashboard',
                templateUrl:
'/partials/dashboard.html',
                controller: 'dashboardController',
                controllerAs: 'dbc',
            })
            .state('Editor',{
                url: '/Editor/:project_id',
                templateUrl: '/partials/editor.html',
            })

    }])
```

Notiamo la presenza dell'*interceptor*, che appunto intercetta le chiamate per l'autenticazione degli utenti.

7.3.1 I controller

Il front-end di Stardust ha richiesto l'implementazione di vari controller, fra questi citiamo:

- dashboardController;
- editorController;
- homeController;
- mapController;

Tutti i controller svolgono delle micro-funzionalità per gestire le diverse parti della web application. Il loro nome indica a quali view sono stati associati.

Fra tutti possiamo affermare che il controller più rilevante di Stardust è l'editorController, poiché responsabile di tutte le interazioni che avvengono nell'editor e sul foglio di lavoro, quest'ultimo costruito utilizzando un framework diverso, p5.js. È qui quindi che viene gestito il cuore della piattaforma, e per tale ragione questo controller oltre ad essere il più importante è anche il più articolato.

Ad esempio, quando viene caricata la pagina dell'editor, parte l'editorController con l'esecuzione della funzione *onPageLoad()*. Questa, a sua volta, chiama la funzione *getProject()*, la quale recupera l'id ed il nome del progetto. A questo punto vengono reperiti tutti i dati per la ricostruzione esatta dei grafi creati dall'utente nel progetto. In particolare vengono eseguite le funzioni *getProjectPosition()*, *getProjectBuildings(projectID)* e *getBuildingData(buildingID)*, che si occupano rispettivamente di recuperare la posizione, la lista degli edifici, i piani e i grafi di ogni struttura del progetto.

Una volta ottenuti tutti questi dati viene chiamata la funzione *broadcastNodePath()*, che è un evento. In particolare tutte le comunicazioni fra controller e service avvengono tramite eventi. Di seguito sono riportati due esempi, rispettivamente di lancio e ricezione di un evento, nell'editorController per la comunicazione con l'editorService.

Listato 7.35 Lancio di un evento in editorController.js

```
$rootScope.$broadcast('load-graph',
   {
       nodes : event_nodes,
       paths : event_paths
   }
);
```

Il codice sopra riportato mostra un evento scatenato dall' editorController. L'editorService reagisce a questo evento andando a wrappare p5 nel foglio di lavoro e mostrando i dati ottenuti relativi al progetto.

Listato 7.36 Ricezione di un evento in editorController.js

```
$scope.$on('newNode', function (event, data) {
   if(self.project.buildings[self.selectedBuilding])
{
self.project.buildings[self.selectedBuilding].floors[s
elf.floorIndices[self.selectedBuilding]].nodes =
data.nodes;
}
})
```

Questo secondo esempio rappresenta un evento che parte dall'editorService e che l'editorController è pronto a ricevere. L'evento in questione è l'aggiunta di un nuovo nodo sul canvas (in p5), che aggiorna la lista totale dei nodi del progetto.

7.3.2 I service

Poiché i service di AngularJS rappresentano lo strato preposto alle chiamate e alla ricezione dei dati verso e dal server, per poi fornirli ai controller, in Stardust è stato necessario implementare i seguenti servizi:

- p5WrapperService;

- editorService;
- editorServiceA;
- editorServiceB;
- editorApiService;
- authenticationService;

Il servizio p5WrapperService è quello che si occupa di wrappare la libreria p5.js in AngularJS, che la utilizza nel foglio di lavoro dell'editor di Stardust.

Per svolgere questo compito il service si avvale dell'elemento del DOM nel quale inserire le funzionalità di p5 e di un ulteriore service, ovvero l'editorService.

Il wrapper crea una nuova istanza di p5 e la mette a disposizione dell'editorService. In questo modo tutte le funzioni di p5 saranno disponibili in *instance mode*, ovvero utilizzando la sintassi *p5.setup()* o *p5.draw()*, e così via.

Listato 7.37 Wrapper per p5 in p5WrapperSerice

```
factory('p5WrapperFactory', ['$injector', 'p5',
function($injector, p5) {
   var p5Wrapper = {
       init: function(sketch, node) {
           this.destroy();

           if(sketch) {
               if($injector.has(sketch)) {
                   sketch = $injector.get(sketch);
               }
               this.instance = new p5(sketch.editor, node);
           }
       },

       destroy: function() {
           if(this.instance) {
               this.instance.remove();
               this.instance = null;
           }
       }
   };
```

```
    return function() {
        return Object.create(p5Wrapper);
    };
}]);
```

L'editorService è quello che contiene tutta la logica necessaria all'interazione con il canvas, pilotato appunto da p5.js. Ad esempio grazie a questo service è possibile disegnare un nodo, o comunicare con l'editorController.

Data la natura ad eventi prima discussa, l'editorService contiene anche tutte le funzioni responsabili della ricezione degli eventi emessi dall'editorController.

Come sappiamo le entità principali dei grafi sono *Node* e *Path*. Nel service questi due componenti sono rappresentati da classi, ed hanno un loro prototype. L'inserimento su canvas di un nuovo nodo o arco si traduce quindi nell'istanza di un nuovo oggetto di una di queste due classi:

- **new Node(x, y, name, type)**: per istanziare un nuovo oggetto node.X e y rappresentano le coordinate del suo centro sul piano, name il suo nome e type il tipo di punto di interesse fra i sette messi a disposizione dall'editor;
- **new Path(start, end)**: per istanziare un nuovo oggetto path. Start ed end sono due oggetti della classe Node, e rappresentano il nodo di inizio e di arrivo dell'arco;

Listato 7.38 Funzione show della classe Node

```
Node.prototype.show = function () {
    p.noStroke();
    this.setType(this.type);
    if (this.locked) {
        p.stroke(0);
        this.size = nodeSize * 2;
        p.noStroke();
    }
    else {
        this.size = nodeSize;
    }
```

```
    //draw the node
    p.fill(this.getFill());
    p.stroke(this.getFill());
    p.ellipse(this.x, this.y, this.size, this.size);

    //draw etikett
    if(this.etikett && this.type != 'tag' && !hide_et){
        //setup the node label
        p.noStroke();
        p.fill(this.getFill());

        if(!this.locked){
            //triangle
            p.triangle(this.x + this.size/2, this.y,
this.x + this.size + 1, this.y - this.size / 2, this.x
+ this.size + 1, this.y + this.size/2)
        }

        //type
        p.rect(this.x + this.size, this.y - 8,
this.etikett.length * 12, 30, 2);
        p.textSize(this.nodeSize * 2 / 3);
        p.fill(236, 240, 241);
        p.text(this.type, this.x + this.size +
this.etikett.length * 2.5, this.y + 4);

        //etikett
        p.fill(p.color(52, 152, 219));
        p.rect(this.x + this.size, this.y + 10,
this.etikett_length * 12, 30, 2);
        p.textSize(this.nodeSize * 2 / 3);
        p.fill(236, 240, 241);
        p.text(this.etikett, this.x + this.size +
this.etikett_length * 2.5 , this.y + 30);
    }

    p.fill(this.getFill());
    p.stroke(125);

}
```

La funzione sopra riportata disegna un nodo sul canvas tramite il metodo *ellipse(x, y, dim1, dim2)*, già discusso nel capitolo 6.

Ricordiamo che questo metodo riceve le coordinate del centro dell'ellisse e le dimensioni in pixel di larghezza ed altezza, che nel caso di un cerchio saranno uguali.

P5.js mette a disposizione tutta una serie di funzioni legate agli eventi da mouse e tastiera, come ad esempio:

- *p.mousePressed = function(){...};*
- *p.mouseMoved = function(){...};*
- *p.mouseDragged = function(){...};*
- *p.mouseReleased = function(){...};*

Queste funzioni vengono utilizzate ad esempio per: aggiungere un nuovo nodo al click del mouse; spostare un nodo trascinandolo con il mouse; collegare due nodi tracciando un arco.

Un aspetto critico, che bisogna considerare quando si creano animazioni e disegni con p5, è rappresentato dalle prestazioni. Il problema nasce dal fatto che p5 di default chiama la funzione *draw()* 60 volte al secondo. Di conseguenza se le dimensioni di un grafo diventano piuttosto grandi, la generazione dello stesso diventa un compito complesso per il browser. Alla luce di tale problematicità abbiamo ritenuto necessario ottimizzare il codice per ottenere buone performance dall'editor. In particolare la decisione che abbiamo preso nei confronti di questo problema è stata quella di bloccare il loop di p5 quando non si verificano eventi. Ad esempio, quando il mouse è fermo di sicuro non può accadere nulla sul canvas. In questa circostanza viene chiamato il metodo *noLoop()*, il quale ferma il ciclo draw di p5. Non appena si verifica un evento al quale è associata una risposta del sistema viene chiamato il metodo *redraw()* che riprende il disegno sul canvas. Passiamo adesso alla descrizione di altri due servizi, collegati semanticamente e sintatticamente all'editorService, e che prendono il nome di editorServiceA ed editorServiceB. Questi entrano in azione quando l'utente accede allo strumento per il collegamento fra piani. In questa circostanza, a causa di svariate difficoltà tecniche, non è stato possibile mantenere su di un unico canvas due grafi differenti appartenenti a due piani distinti. L'implementazione ha invece previsto la creazione di due istanze del canvas, delle quali una è riservata ad un

piano, indicato con la lettera A, e l'altra al piano successivo indicato con la lettera B. L'utente potrà quindi interagire con due canvas posti l'uno accanto all'altro, ognuno dei quali rispondente al proprio service, editorServiceA o editorServiceB.

L'editorAPIService è invece un servizio dal funzionamento standard previsto in AngularJS, poiché si occupa delle chiamate al server e di fornire i dati ai vari controller. Le *dependency* di cui necessita sono quindi solo *$http* e *$q*, che si occupano appunto delle chiamate http e di fornire le risposte come *promise*.

Di seguito è mostrato un estratto di codice dell'editorAPIService, il quale si occupa di inviare i dati necessari al server per la creazione di un nuovo progetto, e di fornire la risposta al controller.

Listato 7.39 POST per la creazione di un nuovo progetto in editorAPIService

```
var createNewProject = function (title) {
   var log = $q.defer();
   $http.post("/api/add/project",
        {
            project_name : title
        }).then(
        function(data){
            log.resolve(data);
        },
        function(reason){
            log.reject(reason);
        }
   );
   return log.promise;
}
```

Infine abbiamo l'authenticationService, il servizio che si occupa dell'autenticazione degli utenti sulla piataforma web. Questi è quindi provvisto delle funzioni per il login, la registrazione ed il logout. Di seguito è riportato il codice relativo alla funzione di logout.

Listato 7.40 Funzione per il logout

```
var logout = function(){
   var log = $q.defer()
   $http({
       url: '/user/logout',
       method: 'POST',
       headers: {
           'Content-Type' : 'application/x-www-form-
urlencoded; charset=UTF-8'
       }
   }).then(
       function(data){
           log.resolve(data);
       },
       function(reason){
           console.log("Logout failed: "+reason);
           log.reject(reason);
       }
   );
   return log.promise;
}
```

7.3.3 Le direttive

Nel corso del capitolo 6 abbiamo descritto le direttive di AngularJS, ovvero quegli elementi che estendono l'html in quanto rappresentano DOM personalizzati oppure creati da zero, e costituiscono uno degli aspetti più potenti del framework. In Stardust sono state utilizzate varie direttive, quali:

- *passwordCheckDirective* e *passwordLenghtDirective*: queste direttive sono state costruite per il check della password. Controllano rispettivamente che il contenuto del campo password e quello di conferma password siano uguali, e che la password sia almeno di otto caratteri;

- *fileDirective*: responsabile del controllo dei file relativi alle immagini caricate sull'editor;

- *footerToolDirective*: serve per evidenziare graficamente gli strumenti del footer quando vengono selezionati;
- *mapDirective*: direttiva che wrappa il servizio Google Places in AngularJS. Questo permette di poter utilizzare la funzionalità di autocomplete sulle ricerche svolte sulle mappe di Google;

7.4 Conclusioni

In questo capitolo abbiamo parlato dell'implementazione di Stardust nelle sue parti fondamentali: la modellazione dei dati; la realizzazione della Single Page Application; la congurazione di Node.js.

Questo capitolo è stato costruito estrapolando dall'intero codice alcuni esempi per noi importanti ai fini di una buona comprensione dei meccanismi e delle tecnologie che sono entrati in gioco nella fase di sviluppo della piattaforma.

Alla luce delle descrizioni teoriche mostrate nella parte II, questo capitolo ha avuto quindi l'obiettivo di rafforzare i concetti cardine delle tecnologie utilizzate.

Capitolo 8

Conclusioni e sviluppi futuri

Nel presente lavoro di tesi è stata presentata la piattaforma Stardust. Dopo una prima fase di approfondimento nell'ambito della navigazione indoor, nella quale abbiamo analizzato i vantaggi e gli sviluppi attuali di questa tecnologia, siamo passati all'introduzione del nostro progetto. In particolare, partendo dall'analisi dei requisiti, degli utenti e del benchmarking, abbiamo determinato le caratteristiche necessarie della piattaforma, andando così a delineare le modalità di utilizzo e le innovazioni introdotte rispetto allo stato dell'arte.

Abbiamo visto che Stardust, sulla base di una preesistente tecnologia di posizionamento, che prevede l'utilizzo di TAG orientati, ha come obiettivo quello di rendere la creazione ed il mantenimento della navigazione indoor un processo semplice e veloce.

Il cuore di Stardust è rappresentato dal suo editor, un tool di progettazione che permette all'utente di specificare i punti di interesse ed i percorsi navigabili in un edificio, lavorando sulle planimetrie dello stesso. Il progetto Stardust è stato pensato con l'obiettivo di rendere la progettazione e l'installazione, richieste per la navigazione indoor in un qualsiasi edificio, un processo intuitivo, veloce ed economico. Questo proposito si traduce nella creazione non di un servizio che richiede l'intervento di esperti, ma bensì di un prodotto completo ed utilizzabile dagli utenti.

Nella fase successiva abbiamo delineato i requisiti tecnologici ed architetturali, arrivando così a costruire uno schema di massima. Di qui siamo passati alla descrizione delle motivazioni alla base di determinate scelte, arricchendo il tutto di richiami teorici utili alla comprensione della successiva fase relativa all'implementazione della piattaforma.

Possiamo a questo punto avanzare verso alcune considerazioni personali e nuove idee per l'evoluzione futura della piattaforma.

L'idea iniziale di introdurre un prodotto nuovo ha rappresentato per noi una sfida, che abbiamo affrontato con grande entusiasmo. Stiamo stati infatti, sin dall'inizio, spinti nel credere che le innovazioni introdotte da Stardust avrebbero portato grandi benefici nel mondo della navigazione indoor, a favore della sua scelta da parte di enti pubblici e privati, e quindi uno sprono per una sua maggiore diffusione. Il timore iniziale di realizzare un prodotto che non fosse totalmente funzionale per le reali esigenze degli utenti è svanito con la crescita della piattaforma, del suo test e con i numerosi feedback positivi ricevuti.

Ovviamente sono ancora tante le funzionalità e le novità che vogliamo introdurre, per rendere questo prodotto il più efficiente e completo possibile. Fra queste le più rilevanti sono:

Estensione funzionalità dell'editor: stiamo attualmente valutando l'aggiunta di nuove entità architetturali e funzionalità da rendere disponibili nell'editor di Stardust, come ad esempio nuovi punti di interesse, che possono trovarsi strutture differenti come quelle delle navi, e di un maggior numero di opzioni di percorrenza sugli archi. Stiamo inoltre valutando la possibilità di inserire segmenti curvi, ora ottenibili solo tramite una serie di nodi connessi da archi brevi;

Versione stand alone: è in cantiere l'estensione della piattaforma web anche in versione stand alone. Per tale scopo il nostro interesse è rivolto verso Electron, una libreria open source, sviluppata dal team di GitHub, che permette di trasformare applicazioni web in applicazioni desktop, o di creare da zero applicazioni desktop utilizzando le stesse tecnologie del mondo web. Il tutto si basa sulla compilazione di codice JavaScript, che come visto nel corso dei precedenti capitoli, è stato utilizzato sia per la realizzazione del front end che per quella del back end di Stardust. Electron prevede inoltre di pacchettizzare l'applicazione per Mac, Windows, e Linux con lo stesso risultato in termini di grafica e funzionalità;

Dashboard arricchita da statistiche: attualmente la dashboard di Stardust prevede l'organizzazione di tutti i progetti relativi ad un utente, oltre ad essere il punto di accesso all'editor. Una sua evoluzione, ancora in fase embrionale, è quella relativa alla sua trasformazione in pannello di amministrazione, dal quale quindi l'utente può accedere alle statistiche di attraversamento dei TAG, e ricevere o inviare notifiche agli utenti finali;

CAPITOLO 8. CONCLUSIONI E SVILUPPI FUTURI

Dopo sette mesi di studio, analisi, dibattiti, ricerche, progettazione ed implementazione possiamo con grande orgoglio osservare la nostra piattaforma in funzione. Questo ci spinge verso un nostro ulteriore investimento in questo progetto, convinti che un giorno Stardust venga utilizzato ampiamente dal pubblico come prodotto completo per la configurazione di un sistema di navigazione indoor.

Ringraziamenti

Bibliografia

[1] *English Oxford Dictionary*, [WEB]: http://en.oxforddictionaries.com/ (ultima consultazione 15 settembre 2017)

[2] Saverino Pagano, Simone Peirani, Maurizio Valle, Alessandro Cortese, *Smart Track*, [WEB]: http://www.smartrackitaly.com/ (ultima consultazione 15 settembre 2017)

[3] David A. Patterson, John L. Hennessy, *Struttura e progetto dei calcolatori*, Zanichelli, 2010

[4] *Node.js guide*, [WEB]: https://nodejs.org/it/ (ultima consultazione 15 settembre 2017)

[5]Rod Johnson, *Expert One-on-One: J2EE Design and Development*, Wrox, 2002

[6] Elena Baralis, Silvia Chiusano, *Database Management Systems - Introduction to DBMS, DataBase and Data Mining Group* - Politecnico di Torino

[7]Eric Brewer, Towards Robust Distributed Systems, *ACM Symposium on the Principles of Distributed Computing*, luglio 2000, [WEB]: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf (ultima consultazione 19 settembre 2017)

[8] Gaurav Vaish, *Getting Started with NoSQL*, Packt Publishing, marzo 2016

[9] Francesco Marchioni, *MongoDB for Java Developers*, Packt Publishing, gennaio 2017

[10] *MongoDB - Index Introduction*, [WEB]: http://docs.mongodb.org/manual/core/indexes-introduction/ (ultima consultazione 21 settembre 2017)

[11] *MongoDB - Data Modeling Introduction* [WEB]: http://docs.mongodb.org/manual/core/data-modeling-introduction/ (ultima consultazione 21 settembre 2017)

[12] Peter Neubauer, Graph Databases, *NOSQL and Neo4j*, 12 maggio 2015, [WEB]: http://www.infoq.com/articles/graph-nosql-neo4j (ultima consultazione 29 settembre 2017)

[13] Ian Robison, Jim Wabber, *The Graph Database,* O'Reilly Media, 2013.

[14] Matteo Torta, *DBMS basati sui grafi: analisi e prototipazione di Neo4j*. Università degli studi di Bologna, 2014

[15] *Manuale Neo4j*. [WEB]: http://docs.neo4j.org (ultima consultazione 23 settembre 2017)

[16] Stefano Gombi, *Aggregazione di dati con gestione ad eventi su Node.Js*, Packt Publishing, 2013.

[17] Ivan Ferrazzi, *Node.js - Introduzione alla programmazione*, [WEB]: http://node.js/articles/develop (ultima consultazione 23 settembre 2017)

[18] Daniele Pezzatini - *Introduzione a Node.js*, [WEB]: http://neo4j-introduction.org (ultima consultazione 23 settembre 2017)

[19] *Express for Node.js*, [WEB]: http://expressjs.com/it/ (ultima consultazione 25 settembre 2017)

[20] *Primi passi con Node.js,* [WEB]: http://codingjam.it/ (ultima consultazione 25 settembre 2017)

[21] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based*. Software Architectures, University of California, Irvine, 2000

[22] *AngularJS*, [WEB]: https://AngularJS-architecture.com (ultima consultazione: 28 settembre 2017)

[23] G*uide to AngularJS Documentation*, [WEB]: https://docs.angularjs.org/guide/ (ultima consultazione: 29 settembre 2017)

[24] Ari Lerner, *ng-book - The Complete Book on AngularJS*, Fullstack.io, 2013

[25] *AngularJS Documentation*, [WEB]: https://docs.angularjs.org (ultima consultazione: 30 settembre 2017)

[26] *P5.js*, [WEB]: https://p5js.org/ (ultima consultazione: 30 settembre 2017)