Politecnico di Milano
Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Dipartimento di Ingegneria Industriale e dell'Informazione

Social Mashlight

Evoluzione di un Framework per Mashup

Relatore: Prof. Luciano BARESI

Tesi di Laurea di:
Luca CHIEPPA Matr. 739809

Anno Accademico 2013-2014

Dedica

Come ogni casa è più solida se sono forti le sue fondamenta, così anche la mia vita si regge su tre colonne, che mi hanno permesso di diventare quello che sono: tre caratteri così diversi che oramai sono parte del mio essere.

***Alla nonna Angelica**, che mi ha insegnato a vivere con determinazione e con forza di spirito, ma sempre nel rispetto delle regole, e a cui devo gran parte della mia educazione e disciplina.*

***Alla nonna Lucia**, che mi ha insegnato il vero significato della generosità e dell'accoglienza, e che nella vita si è disposti a fare anche grandi sacrifici quando si vuole bene a qualcuno.*

***A mio nonno Giuseppe**... un uomo fiero e austero, che amava ed andava orgoglioso del suo lavoro di Parrucchiere, della sua nobile arte. Lui mi ha trasmesso con le azioni il vero significato del primo articolo della nostra costituzione: realmente nel proprio lavoro si realizza se stessi.*

Di lui ricordo anche le camminate per Milano, soprattutto i viaggi in autobus e in tram: ricordo come se fosse ieri quando mi insegnava a prendere 'la piccola' e 'la lunga' per raggiungere infine la sua casa.

Ora mi rendo conto di essere proprio in quel punto del lungo viaggio che è la vita: nel momento in cui scendo dal primo autobus in viale Molise, per salire sul prossimo.

A queste tre colonne dedico questo lavoro di tesi.

Siamo come nani sulle spalle di giganti, così che possiamo vedere più cose di loro e più lontane, non certo per l'altezza del nostro corpo, ma perché siamo sollevati e portati in alto

Bernardo di Chartres

Abstract

Social Mashlight è un framework per la creazione di Mashup, che tiene in particolare considerazione le nuove realtà *Social*.

Si è partiti da Mashlight, uno strumento realizzato nel 2008, da cui sono state prese le basi teoriche e su cui è stato fatto un lavoro di ri-progettazione completa, al fine di aggiornarlo, introdurre nuove funzionalità ed aumentarne l'efficienza e la manutenibilità.

È stato sviluppato con l'utilizzo di linguaggi per il Web, come HTML, Javascript, XML e PHP, ed appoggiandosi a librerie come Symfony e jQuery.

Il suo punto di forza è l'approccio modulare con cui sono state sviluppate tutte le sue parti.

Durante la progettazione, è nata una collaborazione con il lavoro di tesi di un altro studente nell'ambito degli APR (Aeromobili a Pilotaggio Remoto), e sono state programmate nuove componenti e nuovi casi di studio che hanno dimostrato la flessibilità di Social Mashlight in differenti ambiti d'uso.

Ringraziamenti

> Se amate la vita non sprecate tempo, perché è ciò di cui sono fatte tutte le nostre vite.
>
> Benjamin Franklin

Al termine di questa lunga fase di studio, vorrei ringraziare tutti quelli che mi hanno aiutato a raggiungere questo obiettivo e che hanno creduto in me.

Un grazie al **prof. Luciano Baresi**, che con grande pazienza mi ha seguito in questo lavoro di tesi, e **Mattia Moretta**, con cui ho realizzato la parte conclusiva di questa impresa.

Primi fra tutti ci sono però i miei **genitori Nicola e Carmela**: senza il loro sostegno e la loro fiducia nelle mie capacità non avrei raggiunto nessuno degli obiettivi della mia vita.

Ringrazio i miei fratelli, **Federico** e **Valerio**, che oltre ad essere uniti per legame di sangue, sono anche i miei migliori amici.

E tra gli amici che sento vicini come fratelli, sento di dover ringraziare anche **Andrea Gusmaroli**, **Paolo Rescalli**, **Mario Maggi** e **Marco Ranghetti**: grazie per essere cresciuti con me e di esservi divertiti al mio fianco.

Ringrazio la mia zietta **Antonella**, che mi sopporta ogni volta che le invado la casa e che posso dire sia la mia migliore allieva di informatica.

Un grazie a mio zio **Guido**, che è sempre disponibile ad aiutarmi, e grazie a lui ho potuto stampare questo libro così importante.

Ringrazio anche la zia **Doris**, che ogni volta mi rallegra con i suoi biglietti di auguri e la sua inesauribile allegria.

Ringrazio infinitamente **Margherita**, a cui voglio bene come ad una zia.

Ringrazio tutti gli amici de **I Cavalieri del Dado**, con cui ho passato momenti di divertimento indimenticabili e sfide all'ultimo tiro di dado.

Ringrazio anche tutti gli amici de **The Game's Rebels**, sempre pronti a esplorare nuovi giochi, una fucina inesauribile di iniziative.

Grazie a tutti i miei amici di Caravaggio, in particolare **Diana Castagna**, **Elisa Passalacqua**, **Nicola Saulle** ed un amico con cui ho condiviso momenti indimenticabili e che la fragilità di queste nostre vite ha portato via troppo presto:

grazie **Davide Esposto Andrilli** per essere stato un amico buono e sincero come ce ne sono pochi.

Grazie anche alle mie amiche **Anna** e **Gloria**, con cui ho passato tranquille serate tra film e sfide agguerrite sulla plancia di gioco.

Una ringraziamento speciale anche ai miei **colleghi del Politecnico**, con cui ho condiviso le gioie e i dolori di questo lungo cammino.

Meritano un sentito ringraziamento i **condomini del mio palazzo**: uno dei migliori posti dove crescere e in cui ho trovato persone gentilissime e generosissime.

Un grazie anche a tutti i **professori del Politecnico di Milano**, che non ho mai capito perché non vengono ringraziati nelle tesi di laurea, ma che nel bene o nel male sono coloro che forgiano gli ingegneri del domani.

Grazie a tutti gli amici e parenti che qui non ho lo spazio per elencare: se ho dimenticato qualcuno (come è probabile), prometto di fare ammenda.

Space: the final frontier.
These are the voyages of the starship Enterprise.
Its continuing mission: to explore strange new worlds,
to seek out new life and new civilizations,
to boldly go where no one has gone before...

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

> The best things in life aren't things.
>
> Art Buchwald

Il Web si evolve velocemente, ed è difficile oggigiorno rimanere al passo con tutte le novità e le nuove possibilità offerte. La nascita del *Web 2.0* ha dato un nuovo ruolo attivo agli utenti, che prima erano solamente fruitori passivi.

Dall'esplosione di nuove funzionalità sono nati metodi di programmazione in ambito Web. Uno di questi è il paradigma SaaS (Software as a Service): Internet ha iniziato a fornire non solo pagine, ma veri e propri servizi riutilizzabili dai programmatori.

Collegato a questo nuovo modo di intendere la rete, da lì a poco è nato il concetto di Mashup: integrando tutte queste funzionalità, è possibile a loro volta creare nuovi servizi da rendere disponibili agli altri.

Nel frattempo, mentre era studente ad Harvard, Mark Zuckerberg progettò il primo sito di social networking che avrebbe rivoluzionato il modo di comunicare per milioni di persone: Facebook.

I Social Network si sono affacciati sempre più numerosi su Internet, ma si sono sempre presentati con un'ottica a servizi, e quindi possono essere utilizzati nella realizzazione di software più complessi.

Come se non bastasse, da qui a pochi anni è previsto che il Web faccia un'ulteriore e decisivo passo avanti verso un nuova realtà di vita. È l'Internet delle Cose (IoT), in cui a comunicare non saranno solo gli essere umani, ma anche gli oggetti: un'immensa e strutturata rete globale che scandirà anche i momenti privati della nostra vita.

È da questo nuovo sentiero futuristico che provengono gli Aeromobili a Pilotaggio Remoto: i droni prima utilizzati per scopi militari entrano nel mondo civile con la promessa di rivoluzionarlo.

Amazon, Facebook, Samsung e altri colossi del mercato stanno investendo ingenti risorse nello sviluppo di questi dispositi, per aumentare il livello di connessione dell'umanità.

È in questo scenario in continuo rinnovamento che Social Mashlight fa la sua comparsa. L'idea di creare un framework che possa in modo semplice e flessibile creare un'interazione fra componenti diversi, al fine di concorrere ad un determinato scopo definito dall'utente, apre nuovi ed interessanti scenari di sviluppo.

1.1 Struttura della Tesi

Il Capitolo 2 a fronte, si occupa di analizzare la realtà in cui nasce il presente progetto di tesi, mostrando i cambiamenti del modo di sfruttare Internet e il framework di cui Social Mashlight vuole essere l'evoluzione. Viene anche approfondito il concetto di Internet of Things e dell'affermarsi delle nuove tecologie degli APR.

Il Capitolo 3 a pagina 27 tratta dello sviluppo del nuovo framework, soffermandosi sugli strumenti utilizzati per realizzarlo, come le librerie Symfony e jQuery (Capitolo 3.2.1 a pagina 31).

Nel Capitolo 3.3 a pagina 36 vengono invece illustrati i componenti fondamentali: Stickers, Griglie e Processo.

Viene mostrata la struttura fortemente modulare dell'intero progetto, come la sua divisione in Bundle:

- il GridMeBundle, trattato nel Capitolo 3.4.1 a pagina 40, che si occupa dell'interfaccia grafica;
- lo StickersBundle, trattato nel Capitolo 3.4.2 a pagina 48, che è una vera e propria libreria di componenti per poter costruire le proprie funzionalità ad hoc;
- il MashlightBundle, trattato nel Capitolo 3.4.3 a pagina 53, che è il vero cuore del framework e si occupa della creazione dei mashup;

Il Capitolo 4 a pagina 61 raccoglie una serie di Casi di Studio, in cui si vedono le capacità di Social Mashlight messe all'opera.

Infine il Capitolo 5 a pagina 85 vuole illustrare gli obiettivi raggiunti e i margini di miglioramento e sviluppo che il framework aspira a raggiungere.

Il tutto termina con la lista degli acronimi e la bibliografia utilizzata per la redazione di questo documento.

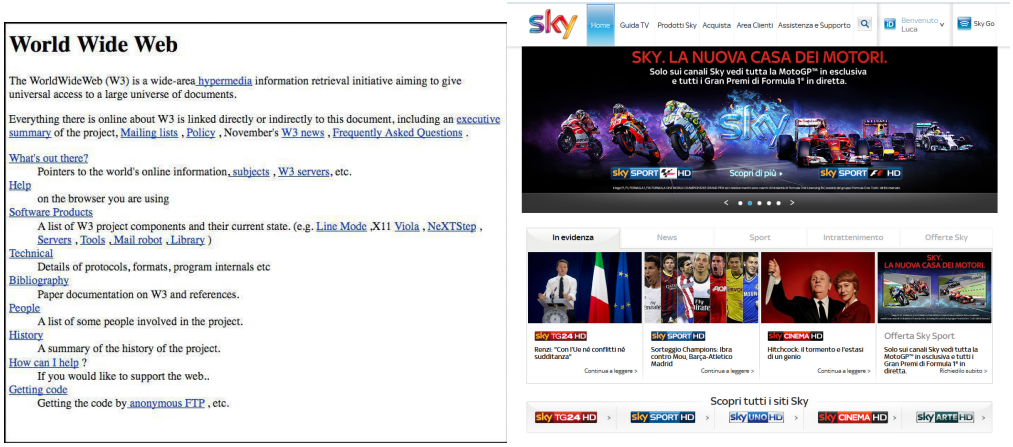
Capitolo 2

Stato dell'Arte

> Quanto più ci innalziamo, tanto più piccoli sembriamo a quelli che non possono volare.
>
> Friedrich Nietzsche

2.1 Web 2.0

Il World Wide Web (Web) si è da sempre contraddistinto per la sua capacità di evolversi ad una velocità impressionante. Se si mette a confronto la prima pagina Web pubblicata da Tim Berners-Lee nel 1991 ed una qualsiasi pagina attuale, si noterà che in poco più di venti anni moltissime cose sono cambiate, sia nella complessità di realizzazione che per la quantità di strumenti ed elementi innovativi che vengono proposti agli utenti.



(a) Pagina di Berners-Lee (b) Pagina web recente

Figura 2.1: Differenza tra la prima pagina comparsa sul Web e una pagina attuale

Una caratteristica di questa evoluzione repentina però è la gradualità dell'intero processo. Dal caos iniziale sino all'affermazione degli standard di programmazione

ancora attualmente utilizzati, in rare occasioni si è assistito a salti netti; soprattutto non ci sono mai state imposizioni forzose, le quali avrebbero danneggiato gli utenti finali noti per la loro *resistenza al cambiamento*. Tale processo è così pervasivo che gradualmente ha cambiato le abitudini dell'individuo, ed essendo un mezzo di comunicazione globale, ha condizionato nel tempo anche i comportamenti di intere società.

Quindi è molto difficile definire dei veri e propri *step*: è il caso del *WEB 2.0*, per cui non esiste una definizione univoca e precisa, e non a caso il termine è stato coniato a posteriori.

È possibile però individuare la componente innovativa di questa nuova fase del web, caratterizzata da un livello maggiore di interazione tra il sito Internet e l'utente comune. Se si prendono in considerazione *blog*, *forum* o *wiki*, dovrebbe essere abbastanza chiaro il nuovo ruolo che rivestono gli utenti: non più solamente fruitori passivi di pagine, ma ora coinvolti nella produzione di contenuti.

Si è vista così la nascita di una nuova logica di programmazione: dal cosiddetto *Web Statico* ribattezzato *WEB 1.0*, si è passati al *Web Dinamico*. Le pagine mostrate all'utente vengono modificate in base alle informazioni ricevute dall'utente stesso e viene introdotta una separazione tra il ruolo di amministratore del sito web e quello di programmatore del codice, prima coincidenti.

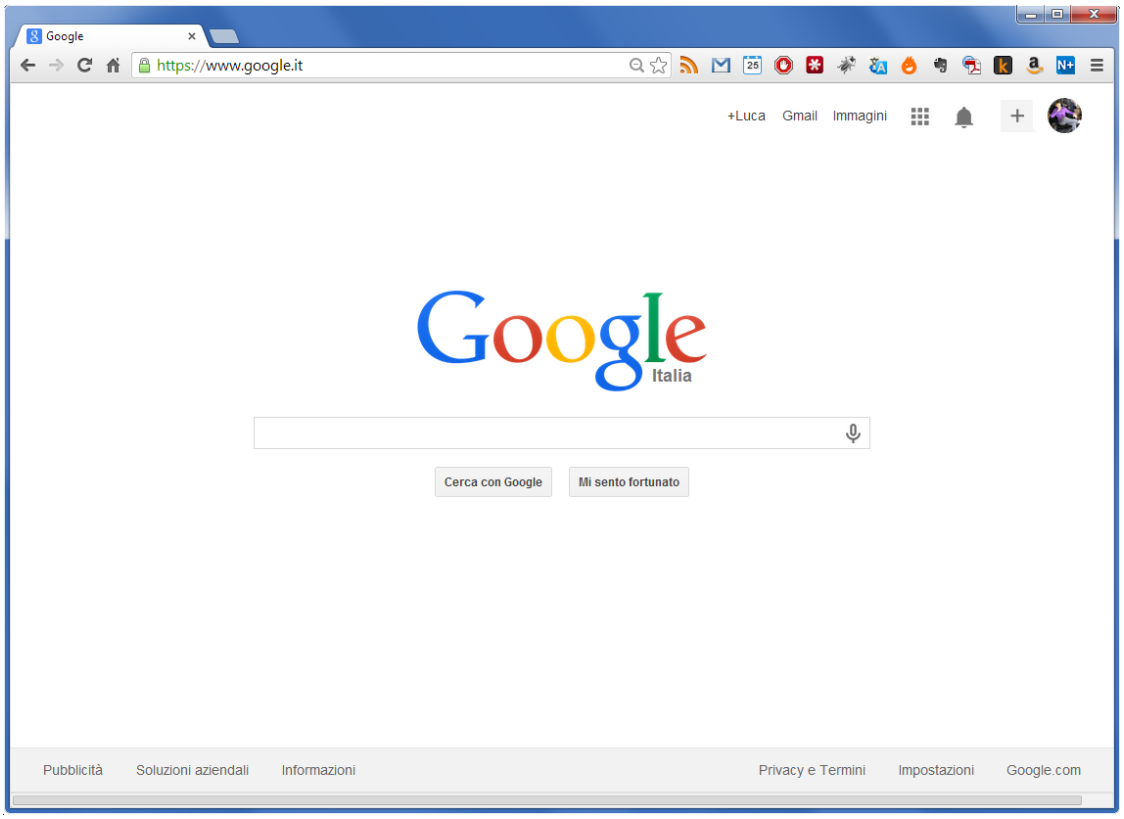


Figura 2.2: Esempio di un browser attuale con numerose estensioni

In questo processo è bene tenere in considerazione anche i browser, ovvero i principali applicativi per la fruizione del web, che da sempre devono stare al passo

con questi cambiamenti ed evolversi molto velocemente. Spesso loro stessi si sono fatti promotori e veicolo di nuove funzionalità o nuovi paradigmi. La capacità di cavalcare o addirittura anticipare le innovazioni in questo campo ha portato numerose aziende software al successo.

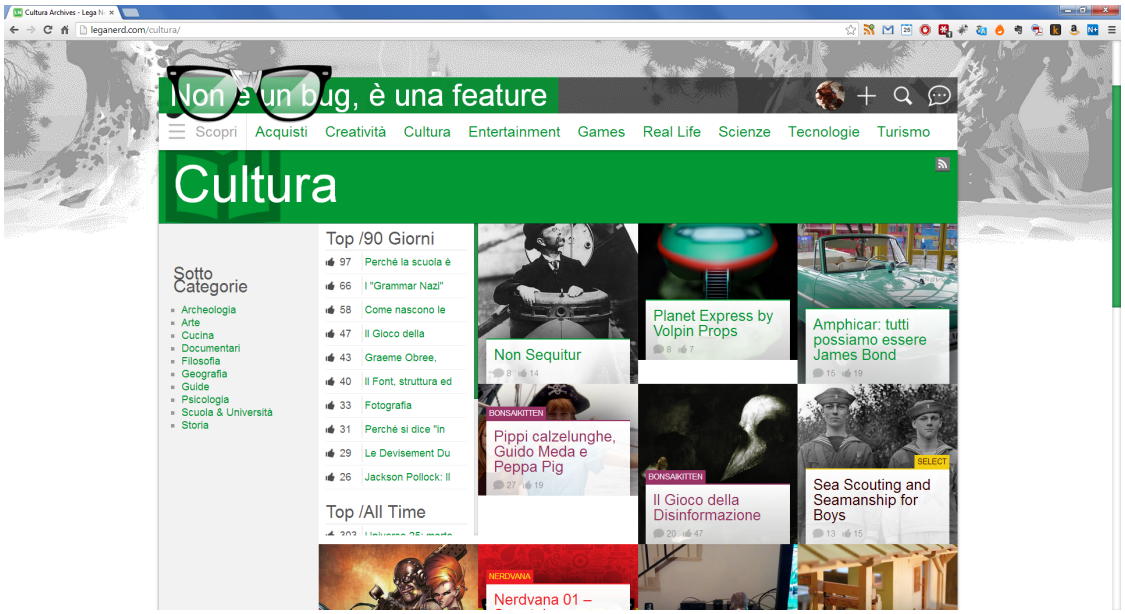


Figura 2.3: Su Lega Nerd chiunque può diventare un autore e pubblicare il proprio articolo

Con l'affermarsi del Web 2.0 i siti hanno perso sempre più la connotazione di pagine statiche e si sono trasformati in veri e propri servizi. Si è assistito così alla nascita del paradigma *Software as a Service*: applicativi con diversi gradi di complessità, prima fruibili solo attraverso un sito web dedicato, hanno ora la possibilità di essere sfruttati in remoto ed utilizzati in software di parti terze.

Questo ha portato notevoli benefici al mondo dei programmatori: sono stati messi a disposizione servizi riusabili e ben fatti (in quanto specifici e molto spesso realizzati da professionisti), utilizzabili dai programmatori nelle proprie pagine web o nei propri programmi tramite poche linee di codice o librerie.

2.2 Mashup

Dal paradigma Software as a Service (SaaS) è nato il concetto di *Mashup*: dati eterogenei e provenienti da diverse fonti remote interagiscono tra di loro per raggiungere uno scopo comune, dando vita ad applicativi web che diventano a loro volta nuovi servizi.

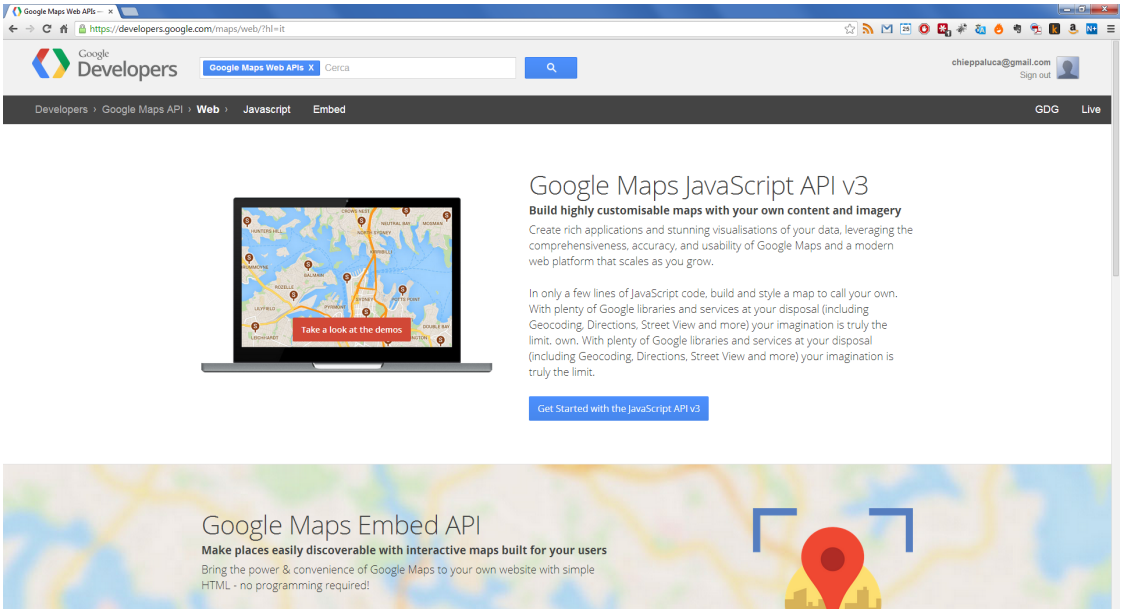


Figura 2.4: Google mette a disposizione delle librerie per poter integrare il proprio servizio di mappe nei siti web

Per quanto nell'ambito della programmazione web questo sistema funzionasse bene, era presente una forte barriera all'ingresso per l'utente finale, molto spesso privo di nozioni di programmazione. Si è quindi cercato di mettere a disposizione questa mole di servizi e di possibilità anche all'utente meno esperto, creando portali che permettessero con pochi semplici passaggi di personalizzare i servizi secondo l'esigenza dell'utente.

Un approccio di questo tipo comporta notevoli vantaggi, sia in termini economici che per i tempi di sviluppo. Per uno sviluppatore è estremamente comodo ed efficiente poter attingere a software collaudato e sicuro senza un utilizzo eccessivo delle risorse interne.

D'altra parte questo meccanismo può presentare dei problemi. Per esempio nel caso in cui un servizio su cui si fa affidamento risultasse inaccessibile per problemi tecnici, esso comprometterebbe l'intera rete ad esso collegato, causando problemi all'utente della nostra applicazione e rendendo inaffidabile il *mashup*. In fase di progettazione va quindi valutata attentamente la qualità dei servizi a cui ci si appoggia.

È possibile dividere i mashup in categorie in base alla loro natura. Una prima divisione è tra *Data Mashup*, *Logic Mashup* e *Presentation Mashup*.

Presentation Mashup Sono la versione di *mashup* più semplice e diffusa, in quanto consentono di combinare informazioni di diverse fonti senza alcuna interazione tra loro.

La pagina iGoogle (figura), recentemente dismessa, ha rappresentato per otto anni un esempio perfetto di questa tipologia. Numerosi widget erano disposti a piacimento dall'utente nella pagina: si poteva così tenere d'occhio la classifica del campionato di calcio mentre si controllava la propria posta elettronica, con ogni componente circoscritto in una specifica porzione della pagina web.

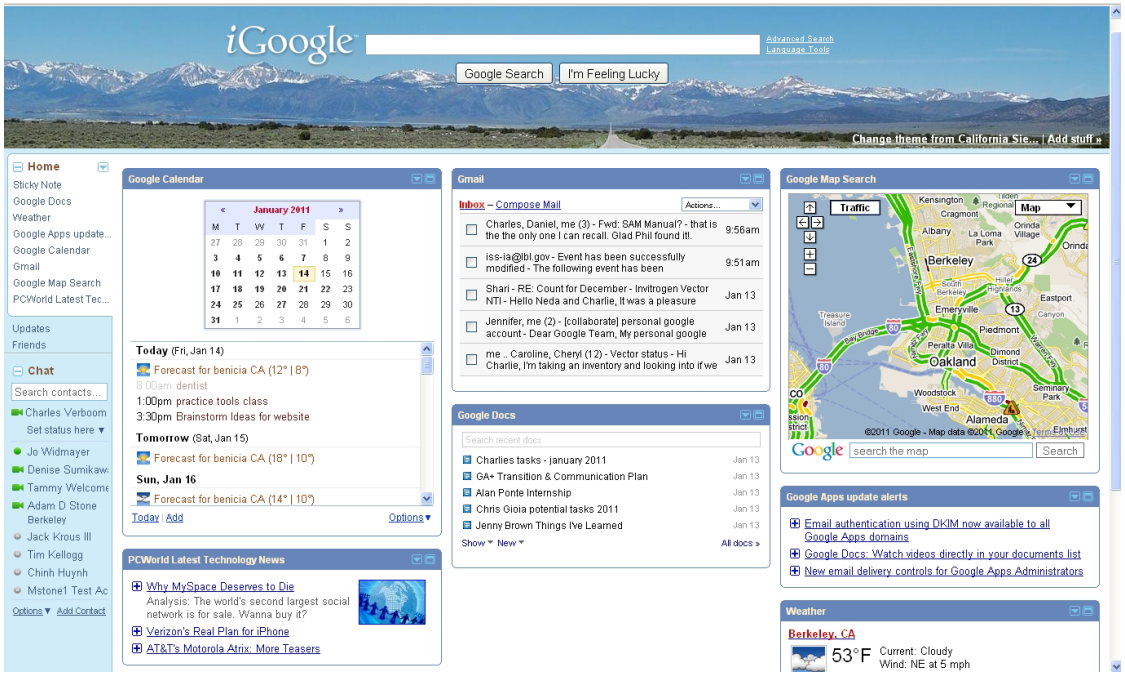


Figura 2.5: Immagine nostalgica di iGoogle

In questa tipologia di *mashup* il codice è spesso composto da poche righe in HyperText Markup Language (HTML) e/o JavaScript, che viene inserito nella pagina tramite incorporamento.

Logic Mashup Questa tipologia di *mashup* permette di aggregare la logica applicativa di diversi servizi esterni. Il risultato può essere la creazione di una nuova funzionalità o la realizzazione di un'applicazione composita.

Figura 2.6: Microsoft Popfly è stato un esempio di Logic Mashup

Data Mashup Lo scopo di questa categoria di *mashup* è l'aggregazione di dati provenienti da diverse fonti in un'unica presentazione.

Per esempio se si posizionano su una mappa di *Google Maps* le informazioni meteo delle diverse città, vengono attinti dati da diversi servizi e vengono presentati in una singola interfaccia: abbiamo ottenuto un semplice *Data Mashup*.

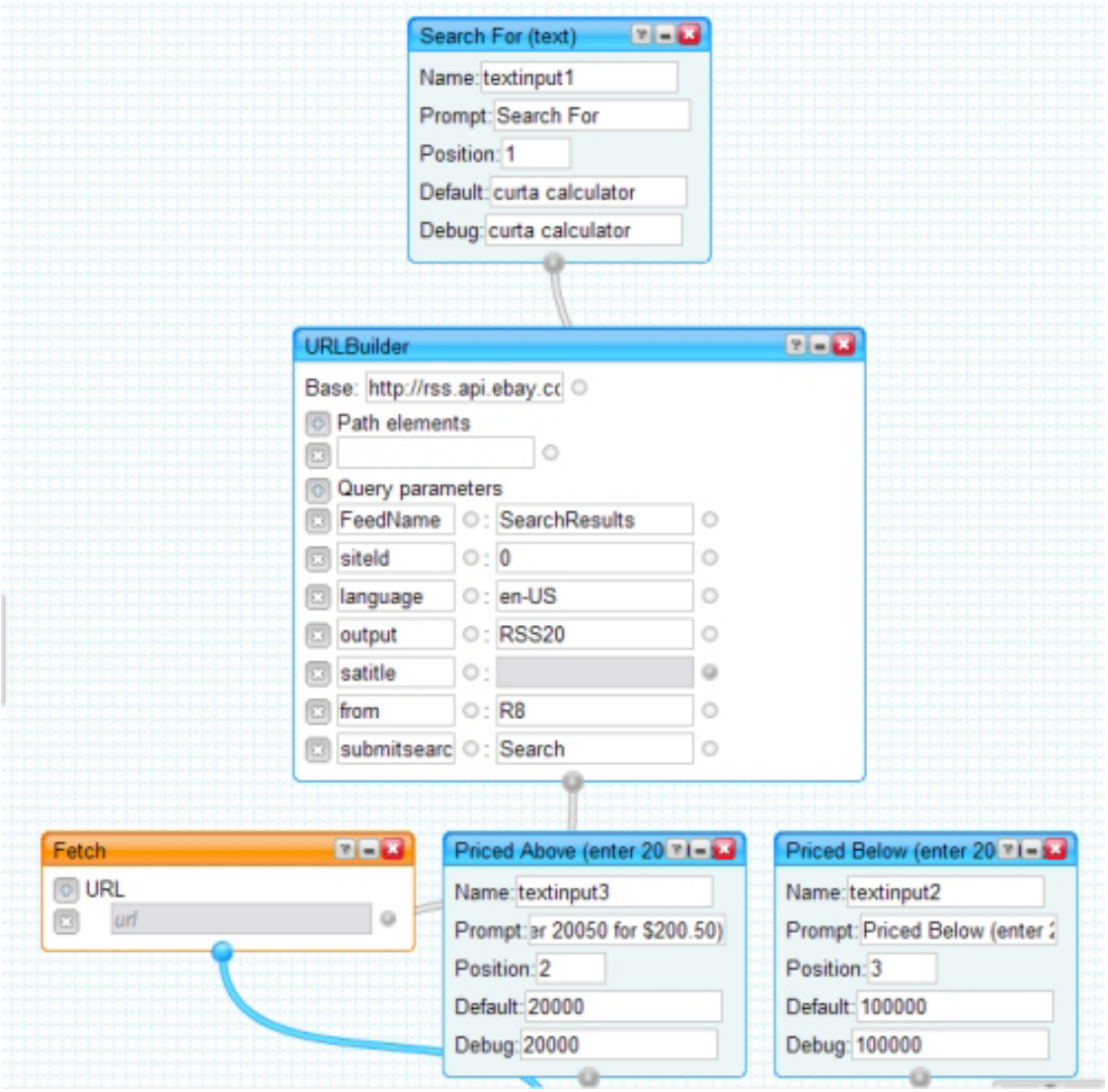


Figura 2.7: Yahoo Pipes è un feed aggregator, esempio di Data Mashup.

In base alla localizzazione del mashup ci si può avvalere di un ulteriore distinzione tra *Client-Side* e *Server-Side* Mashup.

Client-Side Questa categoria di mashup viene utilizzata solitamente per visualizzare contenuti dinamici all'utente. Si avvale tecnologie come Javascript, Flash o Applet per la presentazione dei contenuti, ed XML, RSS o JSON per la trasmissione dati con il web server che fornisce il servizio.

Server-Side Questa tipologia non si affida al solo *client* per la ricezione dei dati: è infatti presente un web server che colloquia con i server dei vari servizi utilizzati. Solitamente opera un'elaborazione prima di inviarli in risposta al client, che si occuperà quindi della visualizzazione all'utente.

Anche se sono perlopiù scomparsi i portali con il solo scopo di raccogliere widget, assorbiti e rinnovati dall'affermarsi dei plugin/estensioni dei browser e dai social network, la pratica del mashup sopravvive ancora e viene utilizzata spesso in blog, forum e portali di informazione.

2.3 Mashlight

2.3.1 Introduzione al framework

Mashlight è un framework nato nel 2008, con l'obiettivo di fornire uno strumento versatile e leggero per lo sviluppo di mashup.

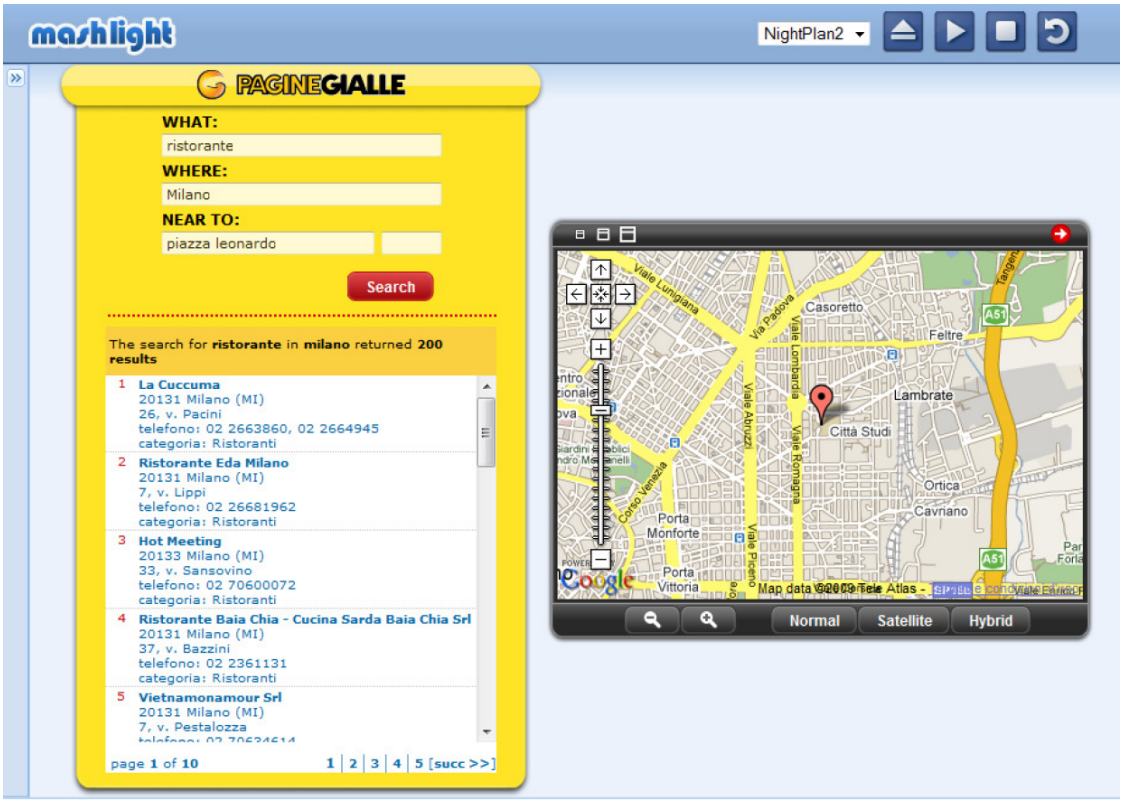


Figura 2.8: Mashlight nella sua versione 2.0, che introduce il concetto di macroblocco.

La maggior parte degli strumenti messi a disposizione per la realizzazione dei Mashup fornisce un'astrazione di medio-basso livello, per un target utente esperto. Ne esistono alcuni che tentano di essere più semplici per coloro che non possiedono approfondite conoscenze tecniche, perdendo la flessibilità di utilizzo delle versioni più complesse.

Mashlight ha l'obiettivo di porsi trasversalmente rispetto alle possibilità offerte, ovvero permettere mashup complessi ma al contempo appoggiarsi ad un framework *lightweight* orientato ai processi, tenendo in considerazione anche un utilizzo *mobile*.

I requisiti seguiti dagli sviluppatori sono stati:

Flessibilità per supportare tutte le tipologie di mashup in scenari differenti;

Usabilità fornire ad ogni tipologia di utente un adeguato livello di astrazione;

Leggerezza e Portabilità mettere a disposizione un ambiente leggero e compatibile con ogni dispositivo fisso o mobile;

Client Side non dipendente da un server;

Semplicità per realizzare mashup in modo rapido ed intuitivo.

Il vero elemento di novità è il concetto di *processo*, che permette di definire durante l'esecuzione quando i blocchi debbano essere eseguiti e quali dati devono utilizzare. Il processo viene definito mediante grafi orientati nei quali i nodi rappresentano le componenti funzionali, i blocchi appunto, mentre gli archi indicano le relazioni fra essi. Sono presenti nel grafo due tipi di archi: uno rappresenta il flusso di esecuzione, l'altro le dipendenze fra i dati.

Il *framework* è realizzato mediante tecnologie web, HTML JavaScript e XML, per garantire la massima portabilità in ogni scenario di utilizzo. I blocchi funzionali sfruttano il concetto di widget, quindi ogni componente è una mini-applicazione Web 2.0. La libreria di blocchi è quindi espandibile a piacimento a seconda delle necessità dell'utilizzatore.

2.3.2 I componenti fondamentali

Si può suddividere il *framework* in componenti fondamentali, così da capire meglio come viene realizzato ed eseguito il mashup.

I Blocchi

I blocchi rappresentano le unità fondamentali del *framework*, le quali scambiano dati tra loro mediante dei parametri di ingresso e uscita. Tali parametri possono essere semplici o strutturati, obbligatori o opzionali. L'unico vincolo della piattaforma è che nel caso un blocco abbia più di un parametro in uscita, detti *outlink*, può venire utilizzato solo uno di questi, evitando l'esecuzione di percorsi in parallelo.

I blocchi vengono riconosciuti dalla piattaforma tramite una rappresentazione formale utilizzando il formato XML. Esso specifica:

- le informazioni generali sul blocco;
- le preferenze di configurazione definibili nell'esecuzione del processo;
- i tipi di dati personalizzati;
- le porte di uscita dette outlink per il collegamento di altri blocchi;
- i parametri in ingresso;
- i parametri in uscita.

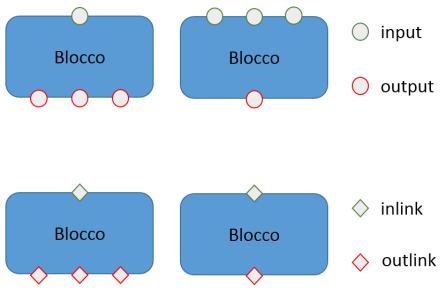


Figura 2.9: Blocchi di Mashlight

Il Processo

I blocchi vengono collegati tra loro per formare un flusso detto *flusso di esecuzione*, rappresentabile mediante un grafo orientato. Tale grafo è la somma di due grafi specifici: *flusso di processo* e il *flusso dati*.

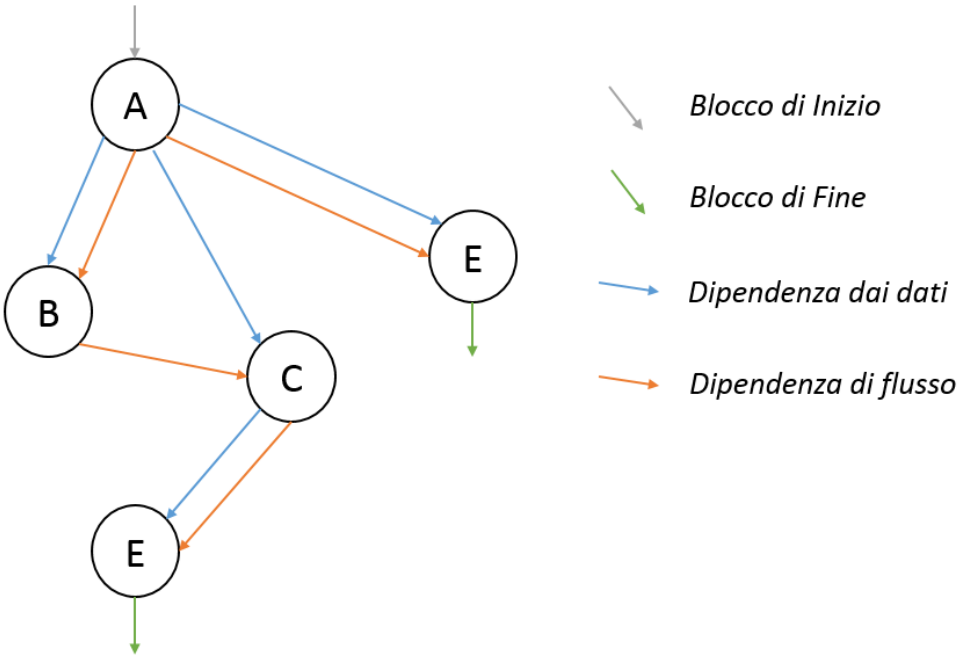


Figura 2.10: I flussi di un processo.

Il *flusso di processo* è il cammino percorso dall'utente nella sua interazione con i blocchi. Esso è dotato di un unico nodo iniziale e finale. Sono ammessi cicli, biforcazioni ed unioni.

Il *flusso dati* invece mostra lo scambio di dati tra i blocchi nell'esecuzione del processo. Il passaggio di parametri può avvenire per assegnamento di un valore in uscita da un blocco precedente o di un valore costante.

L'unico vincolo è che l'assegnamento dei parametri sia consistente con il flusso del processo, in quanto la disponibilità del valore del parametro di uscita del blocco è disponibile dopo la sua esecuzione.

Per evitare inconsistenze nei dati, viene seguita la seguente politica di assegnamento:

- priorità massima all'assegnamento di costanti, l'ultima costante assegnata è prioritaria;
- priorità all'assegnamento del parametro, è prioritario quello proveniente dal nodo eseguito più recentemente;
- assegnamento costanti ordinati con priorità inversa;

- assegnamento parametri ordinati con priorità inversa;

Il processo viene quindi elaborato a partire da un file XML, denominato *manifesto di processo*, che il *framework* traduce nell'esecuzione del mashup.

Funzioni avanzate

Il *framework* fornisce un set di funzionalità aggiuntive, oltre all'esecuzione statica del mashup. È infatti possibile fermare l'esecuzione o tornare ad un passo precedente mediante il meccanismo di *undo*. Questo non è da intendersi come quello implementato dalle basi di dati, ma consiste semplicemente ad un annullamento dell'esecuzione degli ultimi blocchi e non ad un ripristino dello stato originale dei parametri.

Nel caso di acquisti online, per esempio, ad operazione compiuta non è più possibile ripetere o annullare un pagamento. Per questo motivo è possibile inibire l'*undo* tramite un apposito attributo.

Un altro caso è la presenza di clicli nell'esecuzione di un blocco. Per evitare dispendio di risorse per salvare ogni stato, viene salvato solo il parametro aggiornato e quindi viene inibito l'*undo* dal framework stesso.

2.3.3 L'architettura

Il framework si suddivide in due componenti principali: il *Runtime Engine* e il *Mashup Builder*. Il tutto è corredato da una libreria di blocchi chiamata *Block Builder*.

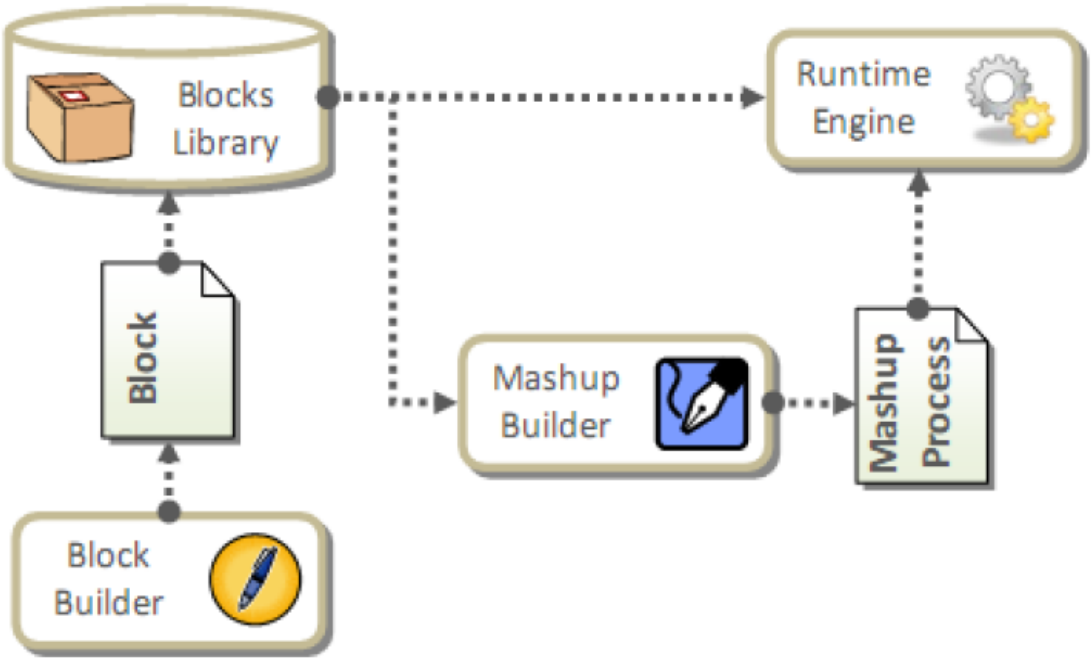


Figura 2.11: L'architettura interna di Mashlight.

Il *Runtime Engine* si occupa dell'esecuzione del processo. È scritto completamente in codice JavaScript ed è eseguibile mediante un web browser. Riassumendo, l'ambiente di esecuzione fornisce le seguenti funzionalità:

- caricare il processo di Mashup;
- avviare e fermare l'esecuzione;
- spostarsi all'interno dell'esecuzione (undo);
- visualizzare la mappa del mashup;

Il *Mashup Builder* invece si occupa della costruzione dei mashup, tramite un'interfaccia semplice e intuitiva.

Infine il *Block Builder* permette di aiutare il programmatore a creare nuovi blocchi per la piattaforma, in modo che rispetti tutti i vincoli per poter comunicare con gli altri componenti del framework.

2.4 Social Network

Quando il giovane Mark Zuckerberg progettò e realizzo il sito web *The Facebook* sicuramente non aveva in mente di creare un nuovo fenomeno mediatico che avrebbe scatenato un nuovo passo evolutivo per il web.

Negli ultimi anni infatti si è assistito alla nascita dei *Social Network*: questi servizi, nati inizialmente come piattaforme di interazione tra utenti, hanno trasformato il modo in cui milioni di utenti si relazionano con il web, fino a condizionare anche le relazioni sociali.

Si assiste tutt'ora ad una vera e propria transizione del web verso una fase *social*, ove l'esperienza dell'utente viene personalizzata in base alle sue informazioni personali ed i sui pregressi storici. Tutto ciò di personale che viene messo liberamente online sul profilo *Facebook, Twitter* o *Myspace*, viene utilizzato dalle aziende e sfruttato per campagne pubblicitarie o iniziative di vendita proattiva, tramite meccanismi di profilazione del cliente.

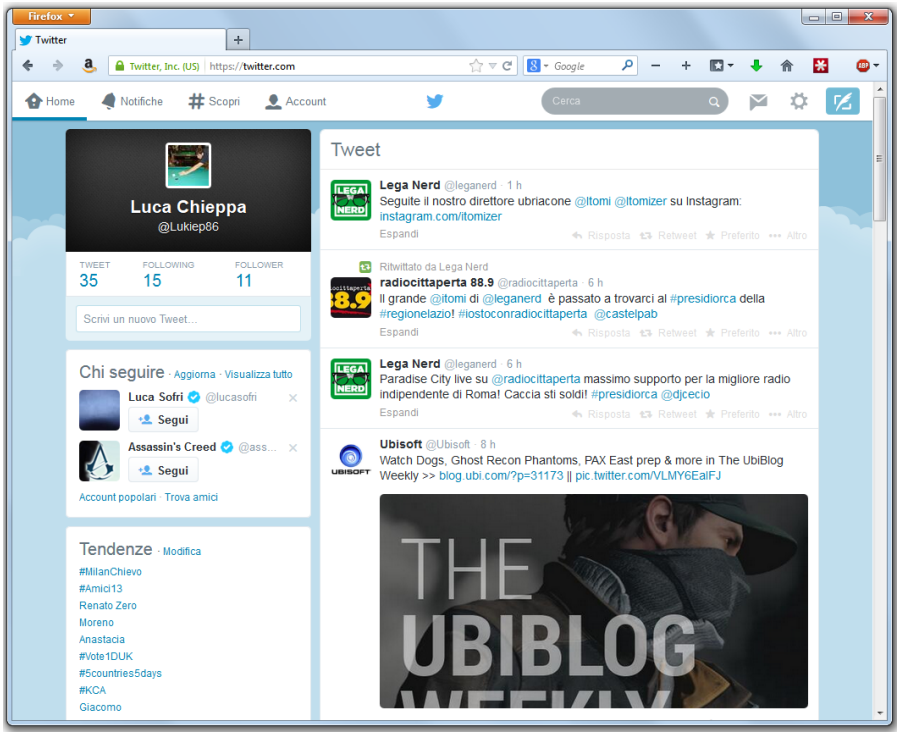


Figura 2.12: Twitter, il social network famoso per il limite dei 140 caratteri per post

Si potrebbe anche sostenere che il *Social Web* sia in realtà la piena realizzazione del concetto di Web 2.0: il valore della piattaforma infatti non è intrinseco, ma è dato dalla conservazione delle informazioni inserite da milioni di utenti, utili alle imprese per profilare il target di clientela. Non a caso *iGoogle* è stato sospeso definitivamente da novembre del 2013 da Google in favore del proprio social network *Google+*.

I Social Network prolificano nella rete, spesso focalizzandosi su determinati aspetti specifici della vita dell'utente e fornendo strumenti avanzati per quel tipo

di focus, siano essi intesi come diario pubblico, galleria di foto, esperienze di viaggi o di vita sentimentale.

I siti web classici hanno ovviamente iniziato a sfruttare il fenomeno, inglobando nei propri blog e portali sempre più componenti social: esempi diffusi sono i nuovi meccanismi di autenticazione o i social widget.

Se da un lato il numero di informazioni presenti sulla rete sono aumentate vertiginosamente, dall'altro la frammentazione di tali risorse è il vero grande limite, per l'incapacità di poterle sfruttarle efficacemente.

Le grandi realtà *social* come Facebook, Twitter o Google+, che hanno a disposizione ingenti capitali, molto spesso preferiscono acquisire le funzionalità di piccole realtà comprando le aziende che li hanno sviluppati, piuttosto che creare meccanismi di interazione tra diversi servizi. Ovviamente non possono comprare tutto poiché la rivalità tra queste grandi compagnie crea una spietata competizione nell'acquisto degli *asset* più interessanti.

Non è ancora ben chiaro quando questa fase avrà termine o molto più probabilmente quando si assisterà all'arrivo di una nuova; questi nuovi strumenti però hanno avvicinato al web milioni di persone, creando identità digitali più o meno integrate con quelle reali, ed hanno introdotto nuovi meccanismi di comunicazione e condivisione delle esperienze su scala globale.

2.5 Servizi Trasversali

Il paradigma a Mashup e di SaaS, affiancato dall'affermazione della filosofia *Social* del web, ha portato allo sviluppo di realtà trasversali, che si pongono a cavallo di diversi ambiti di utilizzo.

Durante lo sviluppo di Social Mashlight sono stati presi in considerazione diversi applicativi e strumenti attuali, per creare qualcosa di originale ma al contempo integrare soluzioni efficaci dal parco software esistente.

La vera spinta evolutiva arriva dal campo *mobile*, che negli ultimi anni si è rivoluzionato per l'offerta di nuovi sistemi operativi e nello sviluppo di applicazioni (soprannominate più sinteticamente *Apps*).

Un primo esempio del concetto di mashup in ambiente Android è rappresentato da **Atooma**. Sviluppata da un team italiano, è stata premiata al Barcellona Word Congress dopo una selezione che ha coinvolto oltre 1.400 concorrenti.

Figura 2.13: Screeshot di Atooma.

Questa applicazione permette di combinare in modo creativo le *Apps* installate con l'interazione dell'utente e le *features* del telefono. Un vero e proprio mashup di funzioni volto a semplificare la vita dell'utilizzatore automatizzando le operazioni più semplici.

Il concetto alla base è semplice: ad una determinata condizione corrisponde un'azione: in termini informatici, una struttura IF-DO.

Alcuni esempi: se si riceve una mail da un collega, vengono salvati gli allegati nella specifica cartella di Dropbox; se si giunge a casa, avviene la connessione alla rete preferita e si attivano tutte le notifiche di Facebook; se piove e se c'è traffico viene anticipata la sveglia di mezz'ora; se è il compleanno di alcuni amici vengono mandati automaticamente gli auguri, etc.

Per quanto viene esposto nel Capitolo 2.6 a pagina 20, questo è un primo modo di dare ai componenti interni di un singolo *device* una maggiore reattività ai contesti.

L'approccio utilizzato per Atooma è quello di creare uno strumento tecnologicamente molto avanzato ma alla portata di tutti.

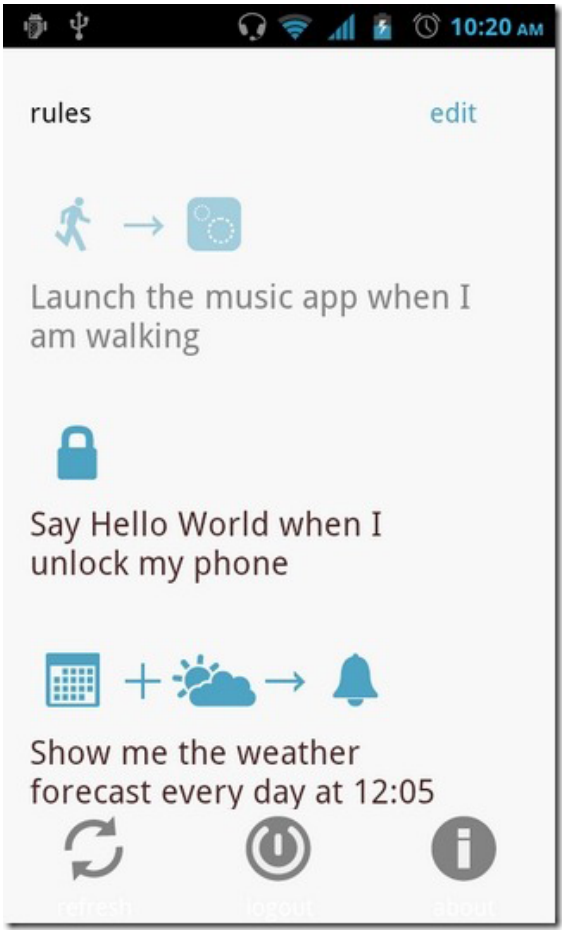


Figura 2.14: Azioni su on{X}.

Meno immediato ma più ambizioso è il progetto **on{X}**. Sviluppato da Microsoft per Android, questa *App* permette di istruire lo smartphone a reagire a determinate situazioni con delle azioni specifiche.

Se, ad esempio, si arriva sul luogo di lavoro, si può programmare il *device* affinché mandi un SMS ad uno specifico contatto od un messaggio su Facebook; oppure quando si lascia l'automobile e si inizia a camminare, fare in modo che la suoneria aumenti di volume o disattivi il vivavoce.

La differenza fondamentale con Atooma consiste nella modalità di configurare le azioni: on{X} si appoggia ad una libreria remota in codice JS, gestita ed aggiornata da una comunità di utenti. I programmatori possono quindi configurare le proprie azioni personalizzate; se non si conosce questo linguaggio, bisogna utilizzare le azioni messe a disposizione dagli altri.

In sostanza l'*App* installata sullo smartphone fa da ponte con le librerie remote: interpreta il codice JavaScript remoto e in base all'azione agisce sulla configurazione del *device*.

Sul versante più propriamente *Social*, ci sono molte nuove realtà che cercano il loro spazio. Una di queste è **Yammer**, un servizio che mette a disposizione nell'ambito lavorativo un vero e proprio *Social Network* aziendale.

Come si vede dalla Figura 2.15 a fronte, l'interfaccia è molto simile a quella di *Facebook*, ma l'enorme vantaggio è l'assenza di distrazioni 'mondane', sfruttando questi nuovi canali di comunicazione per raggiungere obiettivi aziendali.

Non a caso si stanno introducendo nuovi standard, tipo **Open Social**, per permettere lo sviluppo di applicativi in *Cloud* che si integrino in modo più efficace tra loro.

Anche i meccanismi di autenticazione stanno subendo diverse evoluzioni: effettuare un accesso o una registrazione sfruttando i propri dati *Google+* o *Twitter* è ormai un'operazione comune, come l'associazione di più *Social Network* nei propri account di acquisto online.

Non per nulla si è introdotto anche in questo campo un nuovo standard di autenticazione degli utenti: **OAuth 2.0**.

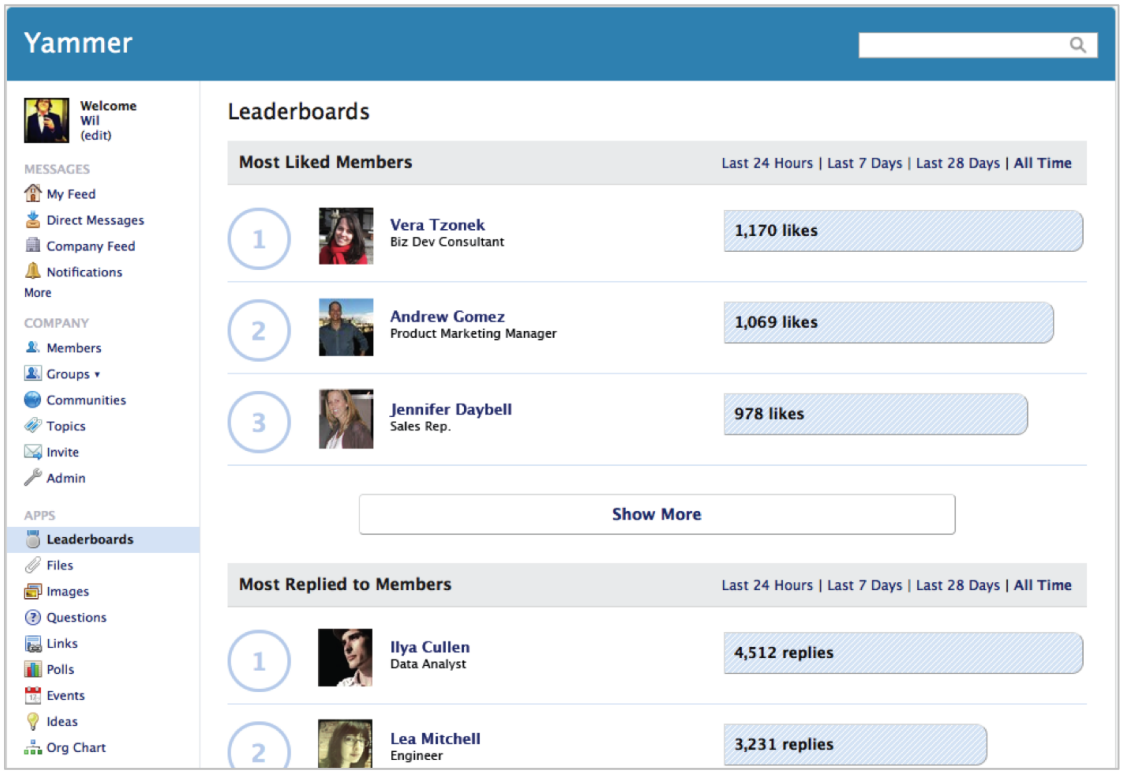


Figura 2.15: Esempio di un Social Network aziendale realizzato con Yammer

Il dinamismo che contraddistingue questo settore è un grande punto di forza per l'avanzamento tecnologico e intellettuale, ma finché non si assisterà all'affermazione di standard più condivisi bisognerà tenere in considerazione qualsiasi nuova realtà, aumentando i costi di ricerca e sviluppo.

Per questi motivi si è deciso di interessarsi ad ambiti così diversi per il presente lavoro di tesi, ma al contempo permeabili ad una piattaforma così eterogenea come Social Mashlight.

2.6 Nuove frontiere: Internet of Things

Possiamo individuare nell Internet of Things (IoT) o **Internet delle Cose** una possibile e concreta evoluzione del Web verso una nuova fase.

Quando nel 1999 Kevin Ashton, fondatore degli Auto-ID Labs del MIT, tenne un meeting con i vertici della Procter & Gamble per esporre le sue ricerche, il concetto era ancora puramente teorico. Venne messo in evidenza come su Internet circolasse una quantità mostruosa di dati: a quel tempo ammontavano a circa **50 petabyte**, frutto di una costante opera di creazione di contenuti da parte degli essere umani.

Figura 2.16: Nell'*Internet delle Cose* ogni oggetto è collegato alla rete.

Queste informazioni però erano prive di un qualche tipo di struttura o 'intelligenza', quindi impossibili da far elaborare automaticamente ad un computer, per quanto potente fosse.

> Se avessimo computer che già conoscono tutto ciò che c'è da conoscere sulle cose utilizzando dati acquisiti senza il bisogno di intervento umano, saremmo in grado di tenere traccia e classificare tutto ciò che ci circonda riducendo la quantità di rifiuti prodotti, perdite e costi. [...] L'Internet delle cose ha il potenziale di cambiare il mondo esattamente come lo ha cambiato Internet. Forse anche di più.
>
> [Kevin Ashton]

Lo scopo di questa nuova frontiera, per quanto dovesse risultare ambizioso al tempo, era di dare intelligenza agli oggetti di tutti i giorni: far sì che il mondo elettronico tracciasse una mappa di quello reale, permettendo agli oggetti, alle persone, ai dispositivi informatici di interconnettersi in un'unica, immensa, rete mondiale.

Gli oggetti possono così comunicare con altri oggetti o persone, scambiarsi dati ed informazioni in maniera diretta ed immediata, al punto di poter modificare il proprio comportamento in base al contesto in cui vengono utilizzati.

Figura 2.17: Smart Object.

Si capisce da subito che per un passaggio ad una realtà così complessa e avveniristica, il primo passo era necessariamente quello di dotare ogni oggetto della capacità di comunicare e interconnettersi. Negli ultimi anni si è assistito alla proliferazione di codice a barre, sensori senza fili e tag Radio Frequency IDentification (RFID): se si da un'occhiata agli oggetti che sono giunti nelle nostre case, si noterà che sono sempre più dotati di una di queste tecnologie al proprio interno.

Mediante queste tecnologie di riconoscimento e tracciamento, gli oggetti sono ormai diventati a tutti gli effetti nodi della rete e sono in grado di comunicare informazioni agli altri oggetti/nodi sfruttando connettività senza fili (Wi-Fi o Bluetooth o la rete GSM).

Per fare chiarezza è giusto citare degli esempi pratici.

Una sveglia, sfruttando informazioni condivise da altri oggetti/nodi della Rete (ad esempio i sensori stradali), sarebbe in grado di prevedere il volume di traffico e quindi di attivarsi nel momento più opportuno (prima o dopo rispetto a quando l'avremmo programmata) in modo da evitare che l'utente resti bloccato nel traffico mattutino.

In palestra, ogni attrezzo connesso alla Rete saprebbe in anticipo quale esercizio si deve svolgere senza che ci sia bisogno ogni volta di programmarlo. Basterà avvicinarsi alla cyclette affinché i sensori di quest'ultima vi riconoscano e impostino automaticamente tutti i parametri in base al vostro programma di allenamento.

Figura 2.18: Smart House

Il campo da basket (o quello da tennis) sarebbero in grado di tenere traccia delle statistiche sulla vostra gara grazie ai sensori installati nel pavimento cosicché a fine partita potreste consultare i dati nello spogliatoio in tutta tranquillità.

Un ruolo fondamentale in questo tipo di evoluzione sarà determinato dall'affermazione della domotica: oggetti di uso casalingo con la capacità di comunicare tra loro, vengono gestiti da una centralina domestica capace di monitorare ogni cosa e effettuare regolazioni e operazioni in modo autonomo. Per esempio saremo avvertiti se un prodotto

nel frigorifero è in scadenza, i termostati si autoregoleranno in base agli ambienti e le temperature esterne, etc.

Se questi casi sembrano ancora lontani da un'applicazione quotidiana, basterà ricordare i nuovi sensori che vengono messi nelle scarpe da corsa per tenere traccia della propria attività, oppure dei nuovissimi spazzolini elettrici che tengono traccia delle proprie abluzioni mattutine per capire che questo futuro non è molto distante. Nel caso della domotica è già realtà.

Anche i nuovi dispositivi indossabili, come gli *Smart-Watch* messi da poco in produzione da aziende come Google e Samsung, rientrano in questa nuova visione, e avranno un ruolo dominante nella vita di tutti i giorni.

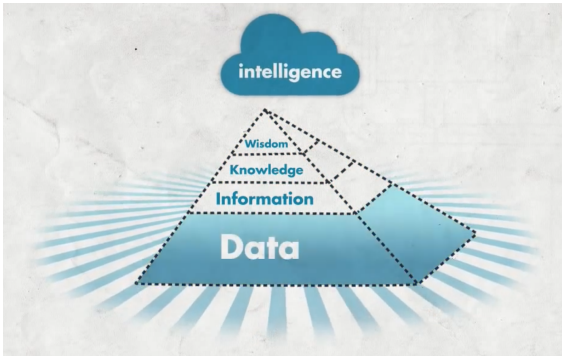


Figura 2.19: Piramide dell'intelligenza

Per capire più in dettaglio come un oggetto acquisisca una sorta di intelligenza intrinseca, è utile elaborare un modello a piramide: dalla semplice e più cospicua mole caotica di dati, salendo verso il vertice della piramide vengono create sovrastrutture che garantiscono una maggior complessità di comportamento. Obiettivo finale del processo è quello di poter conferire agli oggetti caratteristiche di reattività agli input e alle informazioni ricevute, interagendo attivamente nella rete in cui si vengono a trovare.

Nel contesto specifico del lavoro di tesi in questione, poter sfruttare diversi oggetti con diversi gradi di 'intelligenza' può essere un ottimo terreno di sviluppo. Infatti è possibile costruire interfacce e componenti singoli che si relazionano con oggetti specifici, al fine di creare mashup che li pongano in relazione tra loro per svolgere comportamenti più o meno complessi.

Quindi con strumenti come *Mashlight* è possibile aggiungere una sovrastruttura ulteriore nella piramide della Figura 2.19, un'ulteriore rete sopra la rete volta a introdurre nuovi strumenti risultato dell'iterazione di oggetti più semplici.

L'abbattimento dei costi dei componenti di trasmissione senza fili permetteranno nei prossimi anni di dotare qualsiasi oggetto di queste nuove opportunità, dando inizio ad una nuova era ed influenzando significativamente il modo di vivere delle future generazioni.

2.7 Nuove tecnologie: i Droni

Dopo l'annuncio di Amazon di puntare sull'utilizzo di droni civili nella consegna dei propri prodotti, il mercato di questi dispositivi è stato posto sotto una lente di ingrandimento mediatica.

Anche Facebook, in collaborazione con Ericsson, Nokia, Qualcomm e Samsung, ultimamente ha manifestato l'idea di realizzare una rete di droni e satelliti con lo scopo di portare Internet in tutto il mondo (progetto dall'originalissimo nome *Internet.org*).

Figura 2.20: USA Reaper Drone

Questo settore è stato oggetto negli ultimi anni di molti sovvenzionamenti: prima in campo bellico e successivamente in quello civile. La fase di sviluppo e messa in opera in quest'ultimo campo però è ancora ai primi passi, e sebbene esistano velivoli molto complessi e dalle molteplici possibilità, mancano ancora servizi e interfacce per poterli sfruttare efficacemente ed integrarli con altre realtà.

Se si ripensa al concetto di *Internet of Things* del precedente capitolo, diventano chiare le molteplici applicazioni di un dispositivo del genere, soprattutto se dotato di una certa dose di autonomia operativa.

La progettazione di questi velivoli, resa possibile ai giorni nostri in dimensioni molto ridotte, un basso costo di vendita e dispositivi di controllo estremamente portatili, rendono queste attrezzature idonee ad un'ampia varietà di impieghi che differiscono soltanto dalla loro dotazione di sensori.

Social Mashlight si presta bene allo sviluppo di interfacce di controllo per dispositivi APR: data la versatilità di utilizzo e la possibilità di diverse dotazioni, è comodo poter creare configurazioni specifiche attraverso la composizione dinamica di moduli, ognuno con un set di funzioni dedicate.

L'APR è il componente principale di ciò che viene definito Unmanned Aircraft System (UAS), ovvero sistema di pilotaggio remoto. Per poter operare correttamente però è necessaria la presenza di un *sistema di controllo*, del *collegamento di controllo* e di un *equipaggiamento di supporto*.

Il sistema di controllo può essere una piccola attrezzatura portatile che ne consente il pilotaggio a vista, oppure una complessa stazione di terra che può presentarsi come la cabina di pilotaggio di un aeromobile di linea. La scelta dipende dalle dimensioni del velivolo e dal suo raggio operativo.

Il collegamento di controllo dipende dell'area che il velivolo può coprire e dei dati che esso deve trasmettere alla stazione ricevente. La tecnologia più

avanzata consente addirittura un pilotaggio via satellite, permettendo al dispositivo una libertà d'azione planetaria.

L'equipaggiamento di supporto è strettamente legato al compito che l'Unmanned Aerial Vehicle (UAV) deve assolvere. Può essere solitamente modificato a piacimento tramite l'istallazione o rimozione di specifici componenti hardware.

Figura 2.21: Rappresentazione dei componenti di un APR

Da queste premesse nasce il progetto di collaborazione che ha portato alla creazione di due mashup e quattro casi d'uso per il presente elaborato.

Social Mashlight si propone come interfaccia verso il sistema di controllo e l'equipaggiamento di supporto, inoltre può fornire un'elaborazione dei dati restituiti per analisi più approfondite.

Per la necessità di un'architettura a servizi come fonte di dati e per le conoscenze specifiche dei blocchi funzionali interni per il controllo di un APR, si è deciso di collaborare con il dott. Moretta Mattia, che per il suo lavoro di tesi presso il Politecnico di Milano si sta occupando di queste tematiche.

Attraverso l'utilizzo della tecnologia REST, il velivolo Parrot AR Drone 2.0 e la relativa libreria YaDrone, il sistema mette a disposizione i seguenti servizi web:

Campionamento Date le coordinate spaziali il servizio restituisce le immagini campionate nei relativi punti.

Monitoraggio Date le coordinate spaziali il servizio restituisce le immagini campionate nei relativi punti un numero arbitrario di volte.

Ispezione Data un'area di visita e dei servizi terzi di supporto il servizio restituisce i riscontri positivi nei quali è stato individuato un pattern di ricerca.

Ricerca Data un'area di visita e dei servizi terzi di supporto il servizio restituisce il singolo possibile riscontro positivo nel quali è stato individuato l'oggetto ricercato.

Inseguimento Data un'area, un timeout e dei servizi terzi di supporto il servizio restituisce il percorso effettuato dal velivolo per poter inseguire l'obiettivo specificato.

L'architettura considerata è composta da due livelli di deployment, indispensabili per il corretto funzionamento del sistema: il sistema di controllo e analisi dei dati svolto da Social Mashlight ed il velivolo stesso.

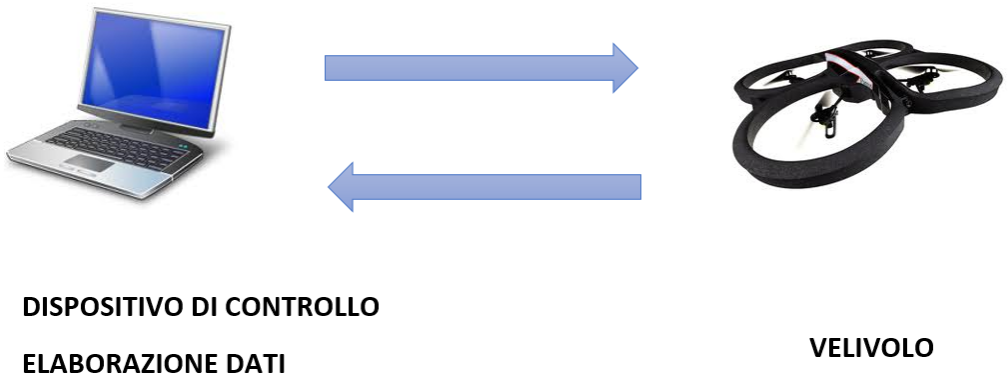


Figura 2.22: Architettura a due livelli di deployment

La richiesta di un servizio da parte dell'utente seguirà i seguenti semplici passi:

1. viene invocato il servizio nel dispositivo di controllo mediante l'inserimento di tutti parametri di configurazione necessari;
2. avviene un'elaborazione dei dati e viene invocato il relativo servizio REST;
3. l'APR decolla ed effettua le operazioni necessarie alla raccolta delle informazioni;
4. vengono elaborati i dati acquisiti dalla sensoristica equipaggiata a bordo del velivolo;
5. i risultati vengono visualizzati a schermo mediante i componenti di Social Mashlight.

Capitolo 3

Social Mashlight

> Impara tutto. Vedrai che in seguito nulla sará superfluo.
>
> Ugo di San Vittore

3.1 Da Mashlight 2.0 a Social Mashlight

Le funzionalità dei Social Network, accompagnate da ricchissime Application Programming Interface (API), si sposano facilmente con il concetto di mashup e di blocchi funzionali. Quindi si è pensato di introdurre nuovi blocchetti *Social*, che permettano l'interazione di Mashlight 2.0 con queste nuove realtà.

Ci si è subito resi conto che la piattaforma originale, per quanto potente e versatile, non è più utilizzabile efficacemente, principalmente per i seguenti motivi:

- molte delle librerie utilizzate dal framework non sono aggiornate, e sui browser attuali vengono generati numerosi errori durante l'esecuzione, compromettendo il funzionamento;
- quasi tutti i blocchi non sono più funzionanti, in quanto i fornitori dei servizi terzi li hanno dismessi o non li hanno più aggiornati;
- l'attuale codice sorgente, è difficilmente leggibile e modificabile data la sua complessità.

Dopo alcuni tentativi di aggiornamento della piattaforma tramite un processo di *reverse enginering* dal codice originale, ci si è resi conto che fosse meglio ripensarla interamente, mantenendo però la struttura dei processi e la modalità di costruzione dei mashup del vecchio framework, frutto di scelte efficaci e a suo tempo collaudate.

L'ultimo aspetto dell'elenco è quello che ha reso la scelta della riprogrammazione la via migliore: infatti il framework è difficilmente aggiornabile, in quanto scritto completamente in codice JavaScript *ad hoc* e non è facilmente leggibile, anche se

abbondantemente commentato. Inoltre l'evoluzione del web ha creato strumenti e standard nuovi e più efficaci per scrivere codice web più ordinato e manutenibile.

Infine, dato l'aumento di velocità delle connessioni internet e le modalità d'accesso alla rete, non sussistono più le premesse per ospitare il framework solamente lato client, cosa che pone dei limiti alle possibilità di evoluzione della piattaforma, ed anche nell'ambito mobile l'idea di un framework in javascript puro è desueta e sconsigliata.

3.1.1 Smart Server

Come si è già accennato, Mashlight 2.0 è strutturato in classi JavaScript. L'intero framework viene scaricato sulla macchina dell'utente che, tramite il proprio browser, costruisce ed esegue mashup. È presente anche qualche riga di codice PHP che serve per quei blocchi che richiedono connessioni a database o usano librerie realizzate con questo linguaggio di programmazione.

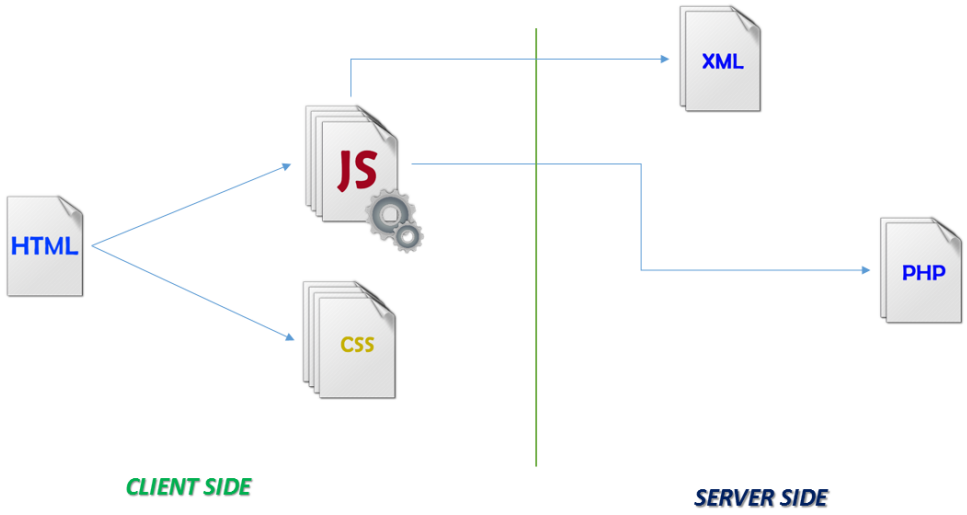


Figura 3.1: Mashlight è stato progettato per essere client-side.

Durante la progettazione della nuova versione, si è scelto di strutturare il framework come servizio ospitato lato server, a cui gli utenti accedono con il proprio browser attraverso un pannello di autenticazione.

Mediante il meccanismo di autenticazione, è possibile salvare su un database o come strutture eXtensible Markup Language (XML), i mashup creati dall'utente e personalizzarne maggiormente l'esperienza. Infatti uno dei vantaggi di questo approccio, è quello di poter collegare gli account *Social* al proprio profilo, e poter sfruttare appieno le API messe a disposizione.

Spostando la logica di programmazione sul server, utilizzando *framework* di terze parti come Symfony, si è potuto organizzare in modo strutturato il codice in

classi specifiche, alcune delle quali sono oggetti e altre veri e propri servizi interni che collaborano tra loro per l'esecuzione del mashup.

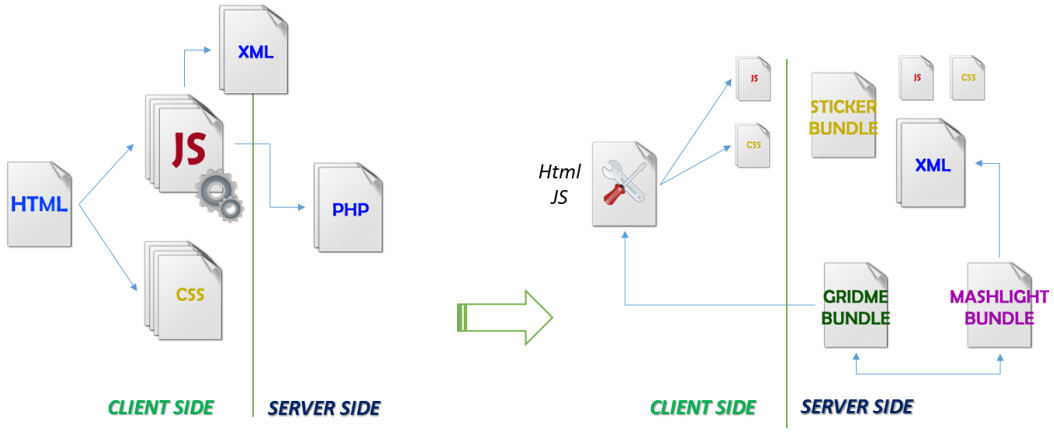


Figura 3.2: Social Mashlight genera il set di risorse lato server da inviare al client.

Invece di inviare l'intero framework come codice JavaScript, Social Mashlight passa al client solo le risorse necessarie per l'esecuzione del mashup corrente. Viene costruita dinamicamente una pagina HTML a cui vengono collegate le pagine JavaScript e CSS dei blocchi apparenti al mashup, oltre alle pagine di base necessarie per far funzionare il tutto.

La comunicazione dei componenti con il server avviene in due modalità, in base alla natura della richiesta.

Nel caso vengano richiesti dati da parte di un blocco specifico, viene inoltrata una richiesta Asynchronous JavaScript and XML (AJAX) che si occuperà di aggiornare il componente o recuperare/salvare dei dati mediante scambio di file XML/ JavaScript Object Notation (JSON). Questo tipo di richiesta è meno onerosa per il client in quanto non forza la pagina al ricaricamento, ma aggiorna solo porzioni di HTML mediante modifiche Document Object Model (DOM).

Se invece viene richiesto l'aggiornamento dell'intera pagina o il passaggio ad un'altra schermata del mashup corrente, allora tramite il meccanismo delle *Rotte* (vedere capitolo 3.2.1 a pagina 31) si inoltra la richiesta al server di costruire la nuova pagina necessaria e di inviarla al browser dell'utente.

3.1.2 Criticità Risolte

Durante lo sviluppo del nuovo framework si è cercato di porre rimedio ad alcune criticità presenti nella versione precedente.

Dal punto di vista della sicurezza, inviare il codice sorgente in chiaro al client è sconsigliabile, in quanto può essere manomesso dall'utente finale per compiere operazioni dannose al servizio principale, o può essere alterato da un utente intermedio portando al noto attacco denominato *Man in the middle*.

Per questo si è utilizzato PHP che interpreta il codice sorgente, crea le pagine HTML, e successivamente le invia al browser dell'utente per la loro visualizzazione.

Come si è già detto, scaricare lato client tutto il framework è oneroso nel precaricamento della pagina, inviando spesso risorse che non verranno utilizzate. Social Mashlight invia pagine leggere e solo le risorse necessarie, ottimizzando tempi di caricamento della pagina e la sua esecuzione.

Ci si è accorti che nel precedente framework non è stata considerata la possibilità di poter utilizzare più volte nello stesso mashup un blocco specifico. Social Mashlight si è occupato quindi di creare identificativi univoci per i singoli blocchi, permettendo l'inclusione duplicata (se il manifesto del blocco lo permette).

Infine si è reso il codice ordinato secondo gli standard di programmazione contenuti nella documentazione di Symfony, permettendo così di poter render l'aggiornamento e la manutenzione del framework più facile, anche con un'opportuna documentazione del codice con *PHP Doc*.

3.2 Strumenti e Tecnologie

3.2.1 PHP con Symfony

Gran parte della struttura di Social Mashlight è stata realizzata mediante l'utilizzo del linguaggio di programmazione interpretato *PHP 5* (acronimo ricorsivo di *'PHP: Hypertext Preprocessor'*, originariamente acronimo di *'Personal Home Page'*).

Questo linguaggio è fortemente orientato alla programmazione web lato server: supporta il paradigma object-oriented (orientato agli oggetti) e fornisce una serie di funzionalità utili come la connessione con i database più diffusi, gestione delle sessioni e protocolli di sicurezza (OpenSSL).

Figura 3.3: Logo Symfony

Se inizialmente PHP veniva utilizzato per la scrittura di poche righe di codice, con il tempo le applicazioni sono diventate sempre più complesse e su Internet hanno iniziato ad essere disponibili appositi set di librerie per facilitare il programmatore.

Sono nati quindi moltissimi framework diversi, come *Yii*, *Zend Framework*, *DooPhp* e *Symfony*, ognuno con i propri punti di forza e teorie di *coding* specifiche.

La scelta di quale utilizzare va attentamente valutata: non vanno confrontate solamente le funzionalità e le caratteristiche messe a disposizione, ma bisogna scegliere soprattutto quello che si ritiene migliore per il tipo di sito o servizio che si intende progettare.

Nel caso di Social Mashlight, si è deciso di utilizzare Symfony.

Sviluppato dall'azienda francese SensioLabs e rilasciato nel 2005 sotto la licenza *MIT Open Source*, ad oggi è nella lista dei framework PHP più utilizzati nel web e può appoggiarsi su un'ampia comunità di utenti che collaborano al miglioramento e ampliamento delle sue librerie.

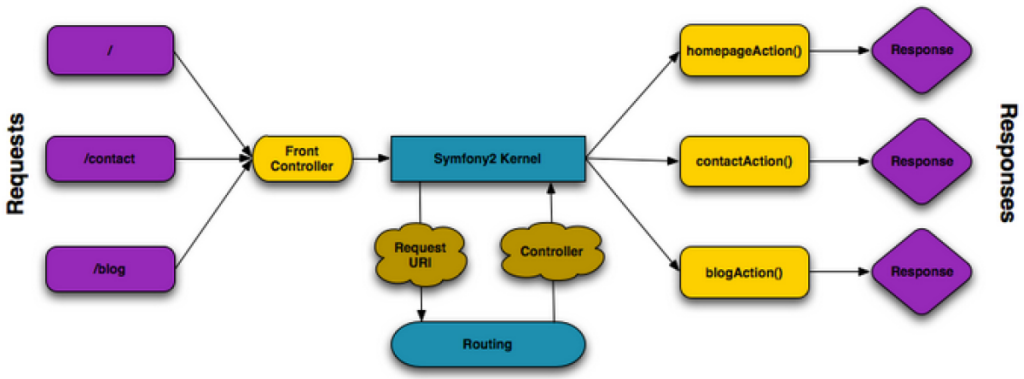


Figura 3.4: Struttura delle Rotte in Symfony.

Questo framework si basa sul concetto di *Rotta*. Con questo termine si indica un particolare tipo di Uniform Resource Locator (URL) che non è associato necessariamente ad una pagina esistente, ma richiama una determinata *azione*.

Qualsiasi sia la rotta richiesta, il framework invoca un particolare tipo di file PHP chiamato *Front Controller* (di default è il file app.php posizionato nella cartella web): questo file ha lo scopo di cercare nella configurazione di Symfony quale azione corrisponde alla rotta richiesta.

L'azione non è altro che una particolare funzione PHP, posizionata in un altro tipo di file estensione della classe *Controller*, che si occuperà di restituire al chiamante un oggetto *Response*: esso può contenere una normale pagina HTML, una pagina PHP, un documento XML o qualsiasi altro tipo di risorsa che il browser dell'utente può interpretare.

Ricapitolando l'intero flusso:

1. si digita l'URL, che deve appartenere ad una rotta registrata per non lanciare eccezioni, per esempio 'www.miosito.it/hello';
2. Symfony riconosce la rotta e trova a quale classe la funzione corrisponde, per esempio la funzione helloAction() del controller HelloController.php;
3. viene lanciata la funzione che genera la risorsa richiesta, per esempio il codice statico di una pagina HTML;
4. la risorsa viene mandata al browser dell'utente per il suo utilizzo.

Per ulteriori approfondimenti sulle rotte e la loro versatilità si rimanda alla documentazione ufficiale, molto ben fatta e disponibile anche in italiano.

Per Social Mashlight, la scelta è ricaduta su questo prodotto in quanto è fortemente orientato alla modularità dei suoi componenti. Perfino il framework base è configurato come modulo, modificabile in ogni sua parte.

Gli altri componenti che si occupano di arricchire Symfony di funzionalità aggiuntive, come Twig (vedere 3.2.2 nella pagina successiva) o Doctrine, sono anch'essi moduli posizionati nella cartella denominata 'vendor', e possono essere utilizzati liberamente dal programmatore o modificati secondo il proprio bisogno.

In Symfony i moduli hanno il nome parcolare di *Bundle*.

Se si decide di seguire le *best-practice* consigliate nella documentazione, il codice dell'intera applicazione viene suddiviso in file riposti in una ben definita gerarchia di cartelle, ognuna con un ruolo ben preciso.

Una serie di file di configurazione scritti in codice YAML o XML (anche in PHP, ma sconsigliabile) si occupano di indicizzare correttamente le risorse all'interno del *bundle*. Quindi per registrare il proprio bundle basterà modificare il file di configurazione principale del bundle 'Symfony'.

Quindi avremo una struttura simile di cartelle in ognuno dei propri bundle:

Controller/ contiene tutti i controllori (p.e. HelloController.php);

Resources/config/ ospita la configurazione del bundle, compresa l'assegnamento delle rotte (solitamente denominato *routing.yml*);

Resources/views/ contiene i template TWIG, organizzati in sottocartelle con il nome del controllore di riferimento (p.e. Hello/index.html.twig);

Resources/public/ contiene le risorse per il web (immagini, fogli di stile, ecc.) ed è copiata o collegata simbolicamente alla cartella web/ del progetto;

Tests/ contiene tutti i file utili per i test del bundle.

Un'altra funzionalità che è utile per Social Mashlight è la possibilità di creare *servizi*. Ci vogliono pochi passi di configurazione per trasformare una qualsiasi classe PHP in un servizio Symfony, ed è possibile soprattutto definire dipendenze tra loro, così da creare una vera e propria architettura interna modulare.

Per quanto sia inizialmente un po' oneroso padroneggiare la piattaforma, sono molti i vantaggi a medio e lungo termine, sia in termini di manutenzione che durante lo sviluppo e il testing della propria applicazione.

Inoltre in rete sono disponibili numerosi *bundle* aggiuntivi che possono essere integrati nel proprio codice, rendendo più veloce e affidabile il proprio prodotto.

3.2.2 HTML e TWIG

In quanto applicazione web, le pagine prodotte da Social Mashlight sono composte di codice *HTML*.

Non ci si soffermerà sulle caratteristiche di questo linguaggio, attualmente alla sua quinta versione, ma si vuole porre l'attenzione sulla struttura della pagina e su come viene costruita dal framework.

Social Mashlight è formato da pagine HTML costruite dinamicamente tramite la collaborazione di due linguaggi: PHP e Twig.

Twig è definibile come un *template engine* per PHP. È stato sviluppato dall'azienda SensioLabs e rilasciato nel 2010 sotto licenza BSD, ed è per questo che si trova nativamente come bundle proprietario in Symfony.

Figura 3.5: Logo Twig

Se è pur vero che è possibile produrre pagine HTML tramite PHP puro, è vero anche che in progetti complessi il codice che ne risulta è difficilmente leggibile, e molto spesso non rende in maniera chiara la struttura della pagina che verrà inoltrata al browser dell'utente. Infatti anche se PHP è nato come un *template engine*, negli anni ha perso questa caratteristica in favore di una maggior versatilità di utilizzo.

Twig invece è completamente dedicato alla realizzazione di *templates*, e si noterà subito la facilità di scrittura del codice e la sua leggibilità.

Su Twig possiamo dire che è:

Conciso il codice da scrivere equivalente è decisamente conciso, per esempio si possono evitare le chiamate a funzioni di *escape*.

Completo supporta tutti i costrutti tipici di un linguaggio di programmazione: ereditarietà, blocchi, cicli, e molto altro.

Sicuro permette il controllo del codice prodotto all'interno di una *sandbox*, con l'utilizzo di funzioni di *escaping*.

Veloce è efficiente nell'interpretazione delle pagine e fa uso di meccanismi di *caching* per la loro memorizzazione.

Reattivo agli errori in caso di errori infatti viene postato un apposito messaggio che mostra la linea di codice incriminata e il tipo di problema.

Un *template* può generare qualsiasi tipo linguaggio. Quando si da un nome al file, solitamente si usa la convenzione seguente: `nomefile.estensione.twig`. Sono file validi per esempio `hello.html.twig` per generare una pagina `hello.html`, oppure `hello.xml.twig` per generare invece un file XML.

All'interno del template, si possono definire dei blocchi con un nome personalizzato.

Questa struttura ha lo scopo di poter ridefinire il contenuto di ciascun blocco mediante relazioni di ereditarietà. È possibile creare un *template* base con la struttura di una pagina web vuota, e ogni *template* figlio si occuperà solamente di riempire i blocchi con il contenuto specifico.

L'ereditarietà e la possibilità di costruire pagine HTML in modo più efficace, è stato fondamentale nello sviluppo di Social Mashlight.

Quando viene chiamato un *mashup*, vengono popolati un serie di template appositi con i link alle sole risorse JavaScript e CSS indispensabili, e la pagina viene arricchita di codice JavaScript inline generato dal *Core Engine* di Social Mashlight per far collaborare i vari componenti tra di loro.

Il risultato che viene ricevuto dall'utente è quindi una comune pagina WEB 2.0 dedicata al mashup richiesto, mascherando completamente la complessità delle funzioni dell'*engine* retrostante.

```
{% extends 'GridMeBundle:Page:base.html.twig' %}


{% block title %}GridMe - Social as you wish{% endblock %}
{% block body %}
<div class="ui-layout-center">
    {% block center%}
        {{ render( controller ('GridMeBundle:Page:welcomeText') ) }}
    {% endblock %}
</div>

<div class="ui-layout-north">
    {{ render( controller ('GridMeBundle:Page:nordPannel') ) }}
</div>
<div class="ui-layout-south">
    {{ render( controller ('GridMeBundle:Page:sudPannel') ) }}
</div>
<div class="ui-layout-east">
    {% block est %}
        {{ render( controller ('GridMeBundle:Page:estPannel') ) }}
    {% endblock %}
</div>
<div class="ui-layout-west">
    {% block west%}
        {{ render( controller ('GridMeBundle:Page:westPannel') ) }}
    {% endblock %}
</div>
{% endblock %}

{% block javascripts %}
    {% block jsImport %}
        {{ page.js('http://code.jquery.com/jquery-1.10.1.min.js') }}
        {{ page.js('http://code.jquery.com/ui/1.10.3/jquery-ui.js') }}
```

Figura 3.6: Esempio della struttura di un template realizzato con Twig.

3.3 I Componenti

3.3.1 Stickers

Il mashup è composto da blocchi mobili che cooperano tra loro chiamati Sticker.

In Social Mashlight sono avvolti da una cornice blu e identificati da un proprio nome specifico. Possono essere trascinati liberamente all'interno della parte centrale della pagina e si può agire anche sulle loro dimensioni.

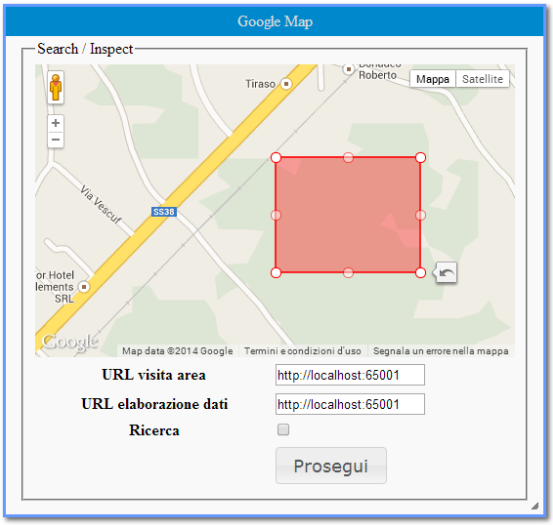


Figura 3.7: Esempio di Sticker.

Ogni Sticker ha una propria funzionalità specifica: si possono considerare come delle mini applicazioni web 2.0 indipendenti tra loro, con una serie di parametri di ingresso e di uscita e funzioni di *update*.

Esempi di Sticker possono essere blocchi che visualizzano mappe, calendari o strutture più complicate come dei form di inserimento dati o iframe. Quasi tutto può diventare uno Sticker con opportuni accorgimenti nella loro programmazione.

Dal punto di vista del codice sorgente, in Social Mashlight tutti gli Stickers sono posizioni all'interno del bundle StickersBundle (per ulteriori dettagli sul suo funzionamento, vedere il relativo capitolo 3.4.2 a pagina 48).

Ognuno ha un proprio set di risorse: pagine e template HTML, CSS e JS, immagini e icone che devono essere poste all'interno di una cartella specifica che deve seguire una nomenclatura formata da `[Produttore][NomeSticker]`. Per far funzionare correttamente lo Sticker e per una maggior chiarezza, si consiglia di creare sottocartelle col nome del tipo di risorsa contenuta: per esempio tutti i file CSS possono essere posizionati nella cartella `DanteMioSticker/css/`.

```
lukiep-imageviewer ..... Cartella Principale
├── css ..... Cartella per le risorse CSS
├── js ..... Cartella per le risorse JS
├── images ..... Cartella per le immagini
├── icons ..... Cartella per le icone
└── xml ..... Cartella per le risorse XML
```

È inoltre presente un file XML di manifesto, che ha lo scopo di indicizzare tutte le risorse interne e assegnare dei parametri formali, come per esempio nome e dimensioni. In questo file sono anche elencate le funzioni di ingresso e uscita dell'applicazione, e altre funzioni utili al caricamento iniziale o al suo aggiornamento.

È la lettura di questo file che permette a Social Mashlight di inizializzare il mashup: ha un ruolo fondamentale e verrà illustrato dettagliatamente nel capitolo 3.4.2 a pagina 48.

3.3.2 Griglie

In Social Mashlight una griglia è l'equivalente di una singola schermata, in cui gli Stickers collaborano tra di loro senza dover aggiornare la pagina. Possiamo definirla come un *macroblocco*, composizione di blocchetti più piccoli, ma anch'essa con parametri di ingresso e di uscita.

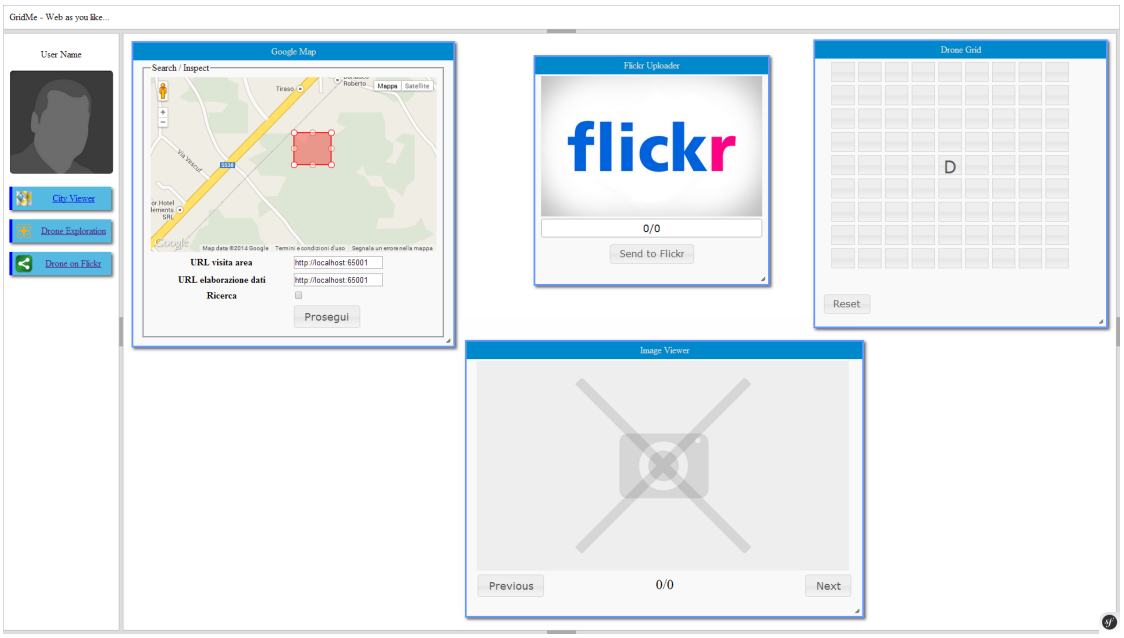


Figura 3.8: Esempio di un insieme di Stickers che compongono una Griglia.

Quando una griglia termina la sua esecuzione, Social Mashlight salva tutte le variabili di uscita e le passa come parametri in ingresso alla griglia successiva. Questo avviene mediante l'utilizzo delle sessioni PHP.

Ricapitolando, ad ogni griglia quindi deve corrispondere almeno uno Sticker; l'insieme di tutte le griglie compone il processo completo del *mashup*.

All'interno dell'applicazione, una classe apposita si occupa della costruzione della griglia, salvando al suo interno tutti gli Stickers che deve utilizzare e le interazioni tra di loro.

3.3.3 Il Processo

La costruzione del processo in Social Mashlight avviene in modo simile alla versione precedente, solo che ora il tutto viene gestito dal server: al browser dell'utente viene inviata la pagina già costruita.

Le variabili utilizzate dagli Stickers vengono inviate al server e memorizzate con l'utilizzo delle sessioni PHP.

Un processo si suddivide in schermate, ovvero le griglie precedentemente illustrate, e ogni griglia è composta da uno o più Stickers cooperanti tra loro.

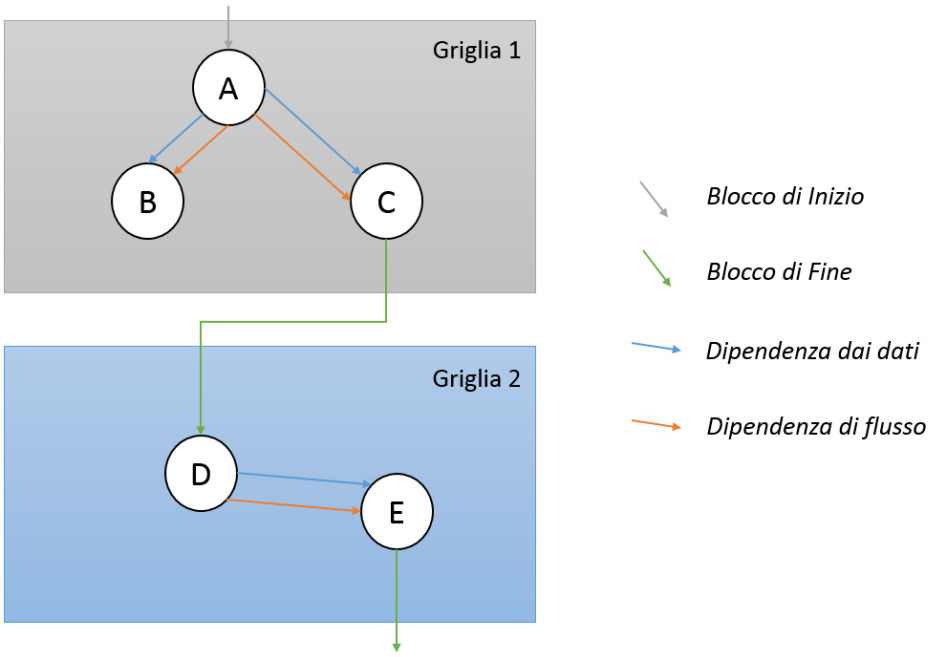


Figura 3.9: Esempio di un processo su più griglie.

Ad ogni pulsante presente, è associato un evento che può (a seconda delle specifiche del mashup) aggiornare uno o più Sticker della medesima griglia o inviare dati alla griglia successiva. Se tutti blocchi permettono alle griglie di tornare a fasi precedenti del processo, allora sono presenti opportuni metodi per tornare alla griglia antecedente, altrimenti il flusso è unidirezionale.

L'intero processo viene gestito dalla classe `CoreEngine` del MashlightBundle, il quale è strutturato come un servizio interno. Questo si occupa di gestire il flusso delle griglie e viene interrogato dall'interfaccia grafica (realizzata dal GridMeBundle) per la costruzione della pagina corrente da visualizzare.

`CoreEngine` interroga il file XML (o il database, nel caso delle versione definitiva del framwork) per recuperare la configurazione dei vari mashup salvati dall'utente e prepara i links per richiamarli.

Quando l'utente clicca su un mashup specifico, questo componente si occupa di richiamare la classe `Grid` e utilizza i servizi interni di Social Mashlight per costruire la pagina da restituire all'interfaccia grafica.

Listing 3.1: process.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<process>
    <about>
        <name>CityViewer</name>
        <icons>
            <icon size="128">images/map-icon.png</icon>
        </icons>
        <description>A little description for the mashup</description>
    </about>
    <grids>
        <grid num="1">
            <connection>
                <outlink stid="inputbox" funname="sendInputText">
                    <variable type="string" name="city" isArray="false"></
                        variable>
                </outlink>

                <inlink sid="googlemaps" funname="codeAddress">
                    <variable type="string" name="address" isArray="false"></
                        variable>
                </inlink>

                <inlink sid="wikicity" funname="searchCity">
                    <variable type="string" name="iframe" isArray="false"></
                        variable>
                </inlink>

                <varbridge>
                    <from>
                        <variable type="string" name="city" isArray="false"></
    variable>
                    </from>
                    <to>
                        <variable type="string" name="address" isArray="false"
    ></variable>
                        <variable type="string" name="iframe" isArray="false">
                            </variable>
                    </to>
                </varbridge>
            </connection>
        </grid>
    </grids>
</process>
```

3.4 I Bundles

3.4.1 Grafica: GridMeBundle

GridMeBundle si occupa di costruire il sito web che ospita al proprio interno i mashup.

Si è deciso di creare un *bundle* a parte per rendere indipendente la presentazione dall'*engine* interno. In questo modo si può in futuro modificare la piattaforma con cui si utilizza Social Mashlight, oppure affiancare a quella attuale una grafica ottimizzata per tablet e dispositivi mobili.

Dopo una pagina di ingresso in cui inserire le credenziali d'accesso, si entra nel pannello principale di Social Mashlight.

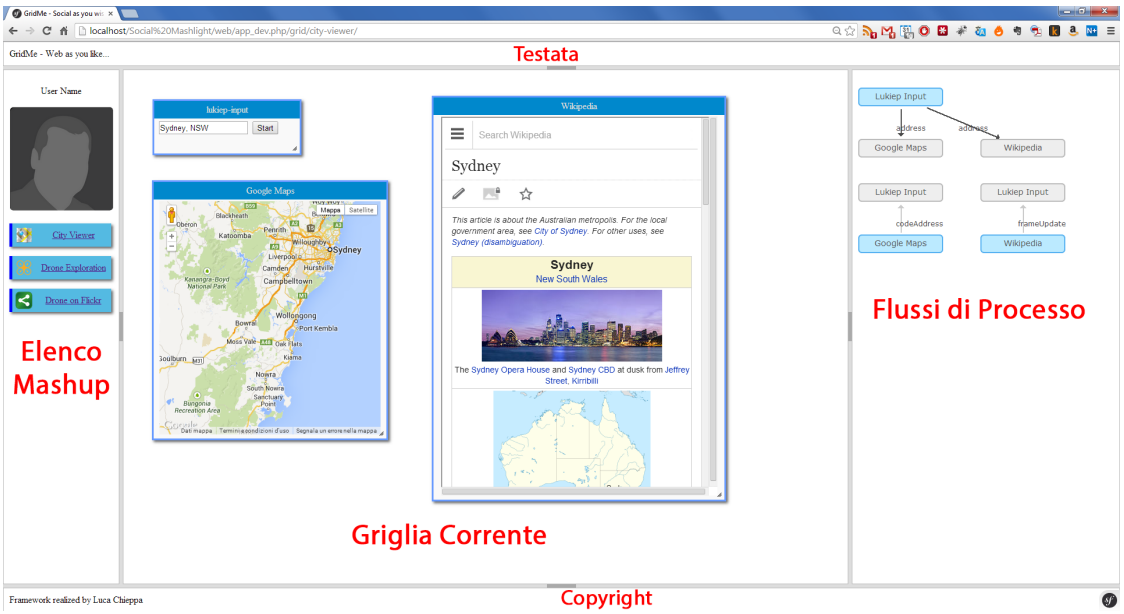


Figura 3.10: Struttura della pagina di Social Mashlight.

Nella Figura 3.10 si possono notare cinque diverse zone che possono essere ridimensionate o nascoste a piacimento. Ognuna svolge un proprio compito ben preciso, e in questa trattazione useremo riferimenti cardinali per identificarli in modo univoco.

Il pannello nord e il pannello sud si occupano di mostrare le informazioni base sulla piattaforma come il logo e il copyright.

Il pannello ovest ha la funzione di mostrare le informazioni dell'utente, come il suo nome e la foto. Ha inoltre lo scopo di visualizzare tutti i mashup realizzati dall'utente come un elenco ordinato.

Ogni voce viene contraddistinta da un nome e da un'icona, assegnati durante la sua realizzazione per permettere una miglior distinzione. Cliccando su una delle voci si comanda al framework di richiamare il relativo mashup.

Questo pannello, con quello nord e sud, non cambierà il suo contenuto durante l'utilizzo della piattaforma, a meno di effettuare l'accesso con un utente differente.

La vera parte vitale del framework è il pannello centrale. Esso ha lo scopo di visualizzare una schermata di ingresso e, al caricamento dei mashup, di visualizzare la griglia corrente. L'utente potrà interagire con ogni componente posto al suo interno, procedendo nell'esecuzione del processo oppure modificare posizione e dimensione degli Sticker a proprio piacimento.

Quando tramite un evento della pagina corrente, si procede alla richiesta di una nuova griglia, questa porzione della schermata viene aggiornata.

L'ultimo pannello è quello posizionato a est. Tipicamente risulterà 'collassato' all'avvio di un mashup, in quanto ha una funzione meramente tecnica. Il suo scopo è quello di mostrare all'utente la struttura del processo, ovvero come gli Sticker interagiscono tra di loro tramite una struttura a nodi.

Si potranno vedere due flussi. Il primo rappresenta il flusso di esecuzione, il secondo il flusso dati (vedere Capitolo 2.3.2 a pagina 12). Da notare è che questa rappresentazione serve a mostrare solo la porzione di processo che interessa la griglia corrente.

Per realizzare pannelli e componenti con una maggior flessibilità di utilizzo e effetti grafici, si è fatto uso di diverse librerie JavaScript.

Figura 3.11: jQuery e jQueryUI

Prima fra tutte è `jQuery`, nota libreria che ha lo scopo di velocizzare la scrittura del codice JavaScript e che permette alcuni effetti di aggiornamento e modifica del codice senza il ricaricamento della pagina.

Su `jQuery` si appoggia la libreria grafica `jQueryUI`: questa libreria fornisce componenti nuovi per la pagina HTML, come *ProgressBar* e *Spinner*, e funzionalità aggiuntive molto utili come rendere blocchi della pagina *draggable* o *resizeble*.

Per ottenere il *layout* a pagina piena, con la suddivisione in pannelli elastici, si è invece utilizzata un'altra libreria grafica `UI Layout` (ovviamente appoggiata sempre a jQuery). È previsto però che tale libreria verrà sostituita da una versione meno obsoleta e con la possibilità di una maggior personalizzazione, `jQuery EasyUI`.

Ogni pannello è realizzato mediante l'utilizzo di *templates Twig*. Ogni file di template si occupa di una porzione di pagina specifica, e grazie alle proprietà di ereditarietà (vedere capitolo 3.2.2 a pagina 33) si può ricreare una struttura nella costruzione della pagina, mostrata nell'immagine seguente.

I contenuti e il *rendering* dei templates avviene mediante le classi `Controller` del presente bundle. I template infatti sono strutture prive di contenuti (o con i

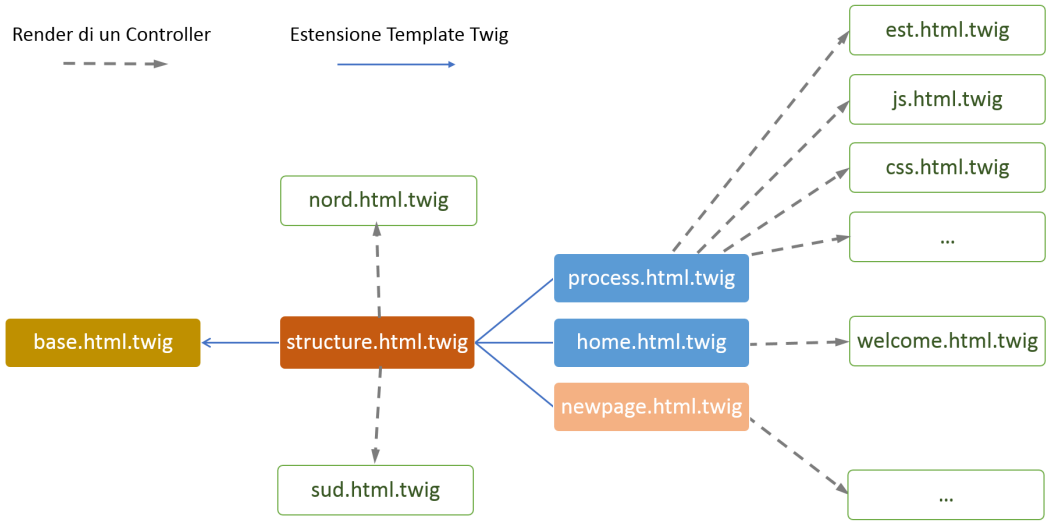


Figura 3.12: Struttura gerarchica dei template Twig del MashlightBundle. Alcune parti della pagina vengono rimepite attraverso il rendering di altri template.

soli contenuti comuni a più pagine): è compito dei *controller* trasformare la rotta e iniettare il codice desiderato nei template per la loro visualizzazione (vedere capitolo 3.2.1 a pagina 31).

In Figura 3.12 è mostrata in modo sintetico la struttura base di ereditarietà dei templates in Social Mashlight, inoltre sono evidenziate le chiamate interne a controller per la costruzione di porzioni interne di pagina.

Il codice seguente invece mostra come Twig costruisce il corpo principale: il file `structure.html.twig` estende `base.html.twig` e costruisce alcune parti della pagina con altri template tramite la chiamata ad opportune *Action*, per esempio `GridMeBundle:Page:welcomeText`.

Listing 3.2: structure.html.twig

```
1  {% extends 'GridMeBundle:Page:base.html.twig' %}
2
3
4  {% block title %}GridMe - Social as you wish{% endblock %}
5  {% block body %}
6  <div class="ui-layout-center">
7      {% block center%}
8          {{ render( controller ('GridMeBundle:Page:welcomeText') ) }}
9      {% endblock %}
10 </div>
11
12 <div class="ui-layout-north">
13     {{ render( controller ('GridMeBundle:Page:nordPannel') ) }}
14 </div>
15 <div class="ui-layout-south">
16     {{ render( controller ('GridMeBundle:Page:sudPannel') ) }}
```

```
17 </div>
18 <div class="ui-layout-east">
19     {% block est %}
20         {{ render( controller ('GridMeBundle:Page:estPannel') ) }}
21     {% endblock %}
22 </div>
23 <div class="ui-layout-west">
24     {% block west%}
25         {{ render( controller ('GridMeBundle:Page:westPannel') ) }}
26     {% endblock %}
27 </div>
28 {% endblock %}
29
30 {% block javascripts %}
31     {% block jsImport %}
32         {{ page.js('http://code.jquery.com/jquery-1.10.1.min.js') }}
33         {{ page.js('http://code.jquery.com/ui/1.10.3/jquery-ui.js') }}
34
35         {{ page.js(asset('bundles/gridme/js/jquery.layout.js')) }}
36
37         <!-- jQuery Block Javascript Code -->
38         {{ page.js(asset('bundles/gridme/js/jquery.wz_jsgraphics.js'))
               }}
39         {{ page.js(asset('bundles/gridme/js/arrowsandboxes.js')) }}
40     {% endblock %}
41  {% endblock %}
42
43 {% block jscode %}
44     <script>
45         $(document).ready(function () {
46             $('body').layout({ applyDefaultStyles: true });
47             $('body').layout('collapse','east');
48             $('.sticker').draggable({
49                 cancel: 'div.box'
50             });
51             $( ".sticker" ).resizable({
52                 animate: true
53             });
54
55
56             readyFunction();
57
58
59         });
60     </script>
61
62     {% block jsFunction %}{% endblock %}
63 {% endblock %}
64
65 {% import "GridMeBundle:Macro:page.html.twig" as page %}
```

```
66 {% block stylesheets %}
67     <link rel="stylesheet" href="http://code.jquery.com/ui/1.10.3/
           themes/smoothness/jquery-ui.css">
68     {#{% stylesheets 'bundles/gridme/css/base/*' filter='cssrewrite'
           %}
69         <link rel="stylesheet" href="{{ asset_url }}" />
70     {% endstylesheets %}#}
71     {{ page.css( asset('bundles/gridme/css/base/page.css') ) }}
72     {{ page.css( asset('bundles/gridme/css/base/west.css') ) }}
73     {{ page.css( asset('bundles/gridme/style/arrowsandboxes.css') )
           }}
74 {% endblock %}
75
76 {#
77
78 Local version of jQuery and jQuery UI
79 {{ page.js( asset('bundles/gridme/js/jquery-1.10.2.min.js') ) }}
80 {{ page.js( asset('bundles/gridme/js/ui/jquery.ui.js') ) }}
81 #}
```

Lasciando perdere i controller che si occupano di operazioni minori, sono due le classi che nel GridMeBundle sono fondamentali per il corretto funzionamento dell'interfaccia: `PageController` e `GridController`.

Il `PageController` si occupa della costruzione della struttura della pagina statica del framework nelle sue varie parti e della visualizzazione dei messaggi standard come quello di benvenuto.

Listing 3.3: GridController.php

```
1  <?php
2
3  namespace Lukiep\GridMeBundle\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6
7  class PageController extends Controller
8  {
9
10     // Routing action for /home/
11     public function homeAction($name = 'Luca')
12     {
13         return $this->render('GridMeBundle:Page:home.html.twig', array
               ('name' => $name));
14     }
15
16     // Display the Home Welcome Text - home.html.twig
17     public function welcomeTextAction() {
18         return $this->render('GridMeBundle:Page:welcome.html.twig');
19     }
20
```

```
21     // Display the West Pannel Informations - structure.html.twig
22     public function westPannelAction() {
23         return $this->render('GridMeBundle:Page:west.html.twig');
24     }
25
26     // Display the Est Pannel Informations - structure.html.twig
27     public function estPannelAction() {
28         return $this->render('GridMeBundle:Page:est.html.twig');
29     }
30
31     // Display the Nord Pannel Informations - structure.html.twig
32     public function nordPannelAction() {
33         return $this->render('GridMeBundle:Page:nord.html.twig');
34     }
35
36     // Display the Sud Pannel Informations - structure.html.twig
37     public function sudPannelAction() {
38         return $this->render('GridMeBundle:Page:sud.html.twig');
39     }
40
41 }
```

Il `GridController` invece si occupa della costruzione della schermata centrale, e le sue *action* (ovvero i metodi interni mappati con delle rotte specifiche), hanno lo scopo di richiamare la classe `CoreEngine` del MashlightBundle per la visualizzazione della griglia corrente del mashup.

Listing 3.4: GridController.php

```
1  <?php
2
3  namespace Lukiep\GridMeBundle\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6
7  class GridController extends Controller
8  {
9
10     private $coreEngine;
11
12
13     public function builderAction($gridname) {
14         $this->coreEngine = $this->get("core_engine");
15
16         $session = $this->getRequest()->getSession();
17         $session->invalidate();
18
19         return $this->render('GridMeBundle:Grid:processbody.html.twig'
               , $this->coreEngine->renderGrid($gridname));
20     }
```

```
21
22      public function prevgridAction($gridname){
23
24          $this->coreEngine = $this->get("core_engine");
25          $this->coreEngine->prevGrid();
26
27          return $this->render('GridMeBundle:Grid:processbody.html.twig'
                , $this->coreEngine->renderGrid($gridname));
28      }
29
30      public function nextgridAction($gridname){
31          $this->coreEngine = $this->get("core_engine");
32          $this->coreEngine->nextGrid();
33
34          return $this->render('GridMeBundle:Grid:processbody.html.twig'
                , $this->coreEngine->renderGrid($gridname));
35      }
36
37      public function savevarAction(){
38          $request = $this->getRequest();
39          $session = $this->getRequest()->getSession();
40          $session->set('param', $request->query->get('param'));
41          return $this->render('GridMeBundle:Grid:successo.html.twig');
42      }
43
44
45      public function welcomeTextAction() {
46          return $this->render('GridMeBundle:Grid:welcome.html.twig');
47      }
48
49      public function estPannelAction() {
50          return $this->render('GridMeBundle:Grid:est.html.twig');
51      }
52
53  }
```

Come si vede dalla struttura della classe, oltre a visualizzare la griglia corrente, ci sono opportune rotte mappate con le azioni *prevgridAction* e *nexgridAction*, che richiamano da `CoreEngine` le griglie precedenti e successive del processo. Sarà compito del MashlightBundle stabilire a quali eventi assegnare queste rotte speciali per permettere la corretta prosecuzione del flusso del mashup (vedere capitolo 3.4.3 a pagina 53).

Da sottolineare è che GridMeBundle non si occupa della gestione del mashup nella sua esecuzione o nei suoi meccanismi interni ma fa da ponte tra l'interazione dell'utente e il MashlightBundle. GridMeBundle riceve da MashlightBundle tutte le risorse da visualizzare già pronte, e si occupa solo dell'impaginazione; oppure registra un'azione dell'utente su uno degli sticker e la inoltra alla classe `CoreEngine` che modifica lo stato del processo e risponde `GridController` con le opportune

risorse da mandare a schermo.

Se invece si tratta di funzioni non legate ai flussi del mashup, ma a semplici dinamiche interne allo sticker (per esempio la modifica di un campo di testo) e quindi codificate nelle risorse specifiche (per esempio una funzione JavaScript), allora la comunicazione avviene direttamente con lo StickersBundle, senza mediazione del MashlightBundle.

Tutto questo è ovviamente volto a separare logica di presentazione da logica di processo, e permettere la sostituzione dell'interfaccia grafica senza modificare la struttura interna del framework.

3.4.2 Componenti: StickersBundle

StickersBundle contiene tutti gli Sticker che Social Mashlight mette a disposizione per la creazione dei mashup.

Differentemente dagli altri due bundle del framework, non ha nessuna classe controller, quindi non si occupa della loro visualizzazione.

Possiamo considerarla come una libreria di componenti, che gli altri bundle usano per costruire i mashup, oppure come un 'magazzino' di risorse.

Gli Sticker presenti in questa sezione sono 'inerti': finché non vengono elaborati dal framework, sono un mero insieme di file e immagini, senza la possibilità di portare a termine alcun compito.

Social Mashlight si occuperà di contestualizzarli e dargli una 'vita': verrà assegnato un identificativo univoco allo Sticker, verranno importate le risorse disponibili, e si metteranno in collegamento le sue funzioni con la griglia che lo ospita. Da quel momento il blocco potrà essere parte del processo.

La flessibilità del framework sta proprio in questo: dallo stesso asset di risorse, è possibile creare diversi Stickers, che a seconda del processo in cui sono inseriti possono avere comportamenti anche molto diversi tra loro.

Rispettando le *best-practice* della programmazione in Symfony, viene mantenuta una struttura a cartelle classica, anche se verrà usata principalmente la cartella `Resources`.

Per ogni Sticker, viene creata una cartella `[Produttore][NomeSticker]` per contenere tutti i file del blocco: un esempio è `LukiepInput` che contiene le risorse del blocco *Input* sviluppato da *Lukiep*.

```
Resources
├── public ..... Cartella per tutte le risorse
│   ├── google-maps ..... Sticker
│   ├── lukiep-imageviewer ..... Sticker
│   │   ├── css ..... Cartella per le risorse CSS
│   │   ├── js ..... Cartella per le risorse JS
│   │   ├── images ..... Cartella per le immagini
│   │   ├── icons ..... Cartella per le icone
│   │   └── xml ..... Cartella per le risorse XML
│   ├── lukiep-input ..... Sticker
│   ├── lukiep-morettadroneform ..... Sticker
│   └── lukiep-morettadronegrid ..... Sticker
└── view
    ├── Lukiep ..... Cartella per i templates
    └── Moretta ..... Cartella per i templates
```

La disposizione delle sottocartelle è a completa discrezione del programmatore dello Sticker, ma si consiglia di raggruppare file simili nella stessa cartella: quindi avremo una cartella `Resources/LukiepInput/css` per tutti i fogli di stile, un'altra `Resources/LukiepInput/js` per gli script JS, un'altra `Resources/LukiepInput/images` e così via.

Nel caso si vogliano creare template con Twig, essi vanno invece posizionati nella cartella apposita del bundle Symfony destinata a questo tipo di risorse. Tutti i file di questo tipo verranno salvati nella cartella `Resources/view/produttore`: per esempio `Resources/views/Lukiep/index.html.twig`. Si è deciso di non differenziare i template per Sticker, per dare la possibilità di utilizzare lo stesso template a più componenti di uno stesso programmatore.

Nella realizzazione dello Sticker bisogna far attenzione ad alcune linee guida, per non aver comportamenti anomali nell'esecuzione del mashup.

Quando si scrive il codice HTML, non è possibile utilizzare tag che non siano contenuti nel blocco *body* del documento: dato che tale codice verrà inglobato in una pagina preesistente, non è possibile per esempio utilizzare i tag *<html>*, *<header>* o simili, a meno di non far uso di *iframe*.

Si consiglia caldamente di usare riferimenti di classe, piuttosto che identificativi (attributo 'class' e non 'id'), al fine di rendere il componente il più versatile possibile.

Per quanto riguarda il codice JS, è opportuno suddividerlo in funzioni specifiche per ogni compito da eseguire, e posizionarle in file separati. Per quanto sia possibile inserire codice direttamente nei tag appositi della pagina HTML, tale modalità è difficilmente aggiornabile e non tiene in considerazione che uno Sticker possa essere utilizzato più volta nella stessa griglia.

```
6
7   var geocoder;
8   var map;
9   function initialize(divID) {
10      geocoder = new google.maps.Geocoder();
11      var latlng = new google.maps.LatLng(-34.397, 150.644);
12      var mapOptions = {
13          zoom: 8,
14          center: latlng
15      }
16      map = new google.maps.Map(document.getElementById(divID), mapOptions);
17  }
18
19  function codeAddress(address) {
20
21      geocoder.geocode({'address': address}, function(results, status) {
22          if (status == google.maps.GeocoderStatus.OK) {
23              map.setCenter(results[0].geometry.location);
24              var marker = new google.maps.Marker({
25                  map: map,
26                  position: results[0].geometry.location
27              });
28          } else {
29              alert("Geocode was not successful for the following reason: " + status);
30          }
31      });
32  }
33
```

Figura 3.13: Esempio di divisione in funzioni di un file JavaScript.

Ogni volta che uno Sticker viene inizializzato in Social Mashlight, gli viene assegnato un identificativo univoco. Questo identificativo è reso disponibile dalla piattaforma per rendere univoche risorse e chiamate a funzioni, quindi è buona

norma sfruttarlo specialmente se il componente può comparire più volte in una griglia.

Nel caso dei fogli di stile, è buona norma assegnarli a componenti contrassegnati da classi, piuttosto che da identificativi univoci.

Inoltre è bene tenere in considerazione che gli Sticker possono essere ridimensionanti nell'interfaccia grafica, e quindi è meglio creare *layout* adattivi: per esempio al posto di usare un dimensionamento in pixel, si dovrebbero usare unità di misura relative (tipo *width: 100%*).

```
<script>
    var node0 = 'node_5339fb971520a';
    var node1 = 'node_5339fb972443d';
    var urlparam = 'nothing';
```

Figura 3.14: ID degli Sticker

L'unica risorsa obbligatoria è la presenza di un particolare file XML che è il manifesto dello Sticker.

Listing 3.5: config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <block>
4      <about>
5          <name>Image Viewer</name>
6          <icons>
7              <icon size="128">images/map-icon.png</icon>
8          </icons>
9          <folder>lukiep-imageviewer</folder>
10         <style>
11             <size width="680" height="450" />
12         </style>
13     </about>
14
15     <components>
16         <javascript-urls>
17             <javascript-url type="relative">js/show-image.js</javascript-url>
18         </javascript-urls>
19
20         <css-urls>
21            <css-url>css/viewerstyle.css</css-url>
22         </css-urls>
23
24         <div-ids></div-ids>
25
26         <html-contents></html-contents>
27
28         <twig-files>
29             <twig-file>Lukiep:image-viewer.html.twig</twig-file>
30         </twig-files>
31     </components>
32
33     <functions>
34         <inlink>
35             <function name="loadImages">
```

```
36                <variable type="string" name="stringimage" isArray="false"></
                      variable>
37            </function>
38        </inlink>
39
40        <outlink></outlink>
41
42        <onload>
43            <function name="showPreloadFun"></function>
44        </onload>
45
46        <variables></variables>
47
48        <update>
49            <function name="loadImages">
50                <variable type="string" name="stringimage" isArray="false"></
                      variable>
51            </function>
52        </update>
53    </functions>
54 </block>
```

Come si vede dalla figura, esso è suddiviso in diversi tipi di descrittori:

- una serie di campi che descrivono il blocco, come il nome e l'autore;
- alcuni paramentri come *<folder>* che vengono utilizzati per localizzare le risorse;
- una serie di parametri che istruiscono il framework su come il blocco debba essere visualizzato;
- l'elenco delle risorse dello Sticker;
- l'indicizzazione di tutte le funzioni JavaScript, e il loro ruolo nell'esecuzione del blocco.

All'interno del tag *<components>* le risorse vengono raggruppate in base alla loro tipologia.

Javascript URLs Vengono indicizzate tutte le pagine JS utilizzate dallo Sticker. Ogni risorsa può essere un percorso *relativo o assoluto*, in base al valore espresso dall'attributo 'type'.

CSS URLs Vengono indicizzati tutti i fogli di stile CSS utilizzati dallo Sticker. Ogni risorsa può essere un percorso *relativo o assoluto*, in base al valore espresso dall'attributo 'type'.

DIV IDs Al posto del codice HTML, il framework permette di creare un blocco DIV con un determinato ID espresso nel file di configurazione; alcuni Sticker infatti vengono costruiti da codice JS che si occupa di riempire il

contenuto di un determinato blocco vuoto. Quando viene aggiunto alla pagina del mashup, Social Mashlightsi occupa di renderlo univoco aggiungendo come prefisso l'uid dello Stickeral valore espresso: per esempio *map-canvas* diventerà *map-canvas_node_532b2a50e1f29*.

HTML Contents Contiene il riferimento a file HTML o è possibile inserire direttamente il codice sorgente (con gli opportuni escape per i caratteri speciali). Tale codice verrà aggiunto alla pagina senza modifiche da parte del framework.

TWIG File Contiene l'elenco dei files di template TWIG da utilizzare. Il percorso viene espresso mediante la codifica a namespace utilizzata da Symfony.

Per vedere come questi componenti vengono aggiunti alla pagina, si rimanda alla discussione del Capitolo 3.4.3 a fronte.

3.4.3 Motore: MashlightBundle

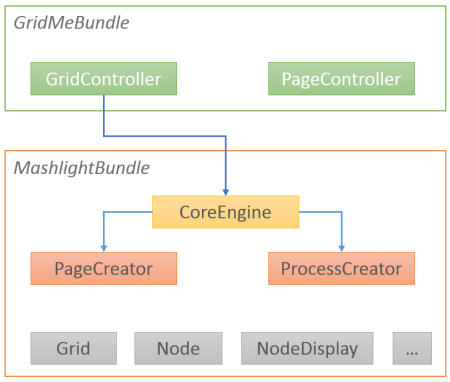


Figura 3.15: Struttura interna a servizi.

Il vero cuore del framework è composto dalle componenti di questo *bundle*.

Tutta la logica applicativa volta alla creazione e gestione dei flussi di processo di un mashup viene infatti gestita dalla classe `CoreEngine`.

Grazie agli strumenti messi a disposizione da Symfony, questa classe in realtà si comporta come un servizio interno: quando GridMeBundle richiede il servizio, viene istanziato un singolo oggetto PHP, a cui d'ora in poi è possibile accedere da tutti i controller che successivamente ne avessero bisogno (per approfondimenti si rimanda alla documentazione ufficiale di Symfony).

Visto che avremo attivo nel sistema un unico servizio che fa uso dei metodi della classe `CoreEngine`, in esso viene salvato lo stato e i componenti del processo in esecuzione, nonché le variabili nelle sessioni.

`CoreEngine` si appoggia a due altri servizi: `PageCreator` e `ProcessCreator`.

`PageCreator` si occupa di costruire le diverse parti di pagina che verranno inoltrate a GridMeBundleper la loro visualizzazzione.

Dal servizio vengono quindi costruiti:

- un blocco di collegamenti a risorse CSS (link relativi o assoluti), tipicamente posizionando nel header del GridMeBundle;

```
<!-- START Imported CSS -->

<link href="/Social Mashlight/web/bundles/stickers/moretta-droneform/css/formstyle.css" rel="stylesheet" type="text/css" />
<link href="/Social Mashlight/web/bundles/stickers/lukiep-imageviewer/css/viewerstyle.css" rel="stylesheet" type="text/css"

<!-- END Imported CSS -->
```

Figura 3.16: CSS Links

- un blocco di collegamenti a risorse JS (link relativi o assoluti), anch'esso posizionato nel header del GridMeBundle;
- un blocco di codice HTML composto dall'interpretazione dei template e l'incorporamento dal codice proveniente dagli Stickers, posizionato nel corpo centrale della pagina.

Tutti i blocchi precedentemente descritti sono creati a partire da template appositi, che creano codice ben formattato per un interprete HTML.

```
<!-- Start Imported JS -->

<script src="/Social Mashlight/web/bundles/stickers/moretta-droneform/js/form-functions.js"></script>
<script src="/Social Mashlight/web/bundles/stickers/moretta-droneform/js/grid-struct.js"></script>
<script src="/Social Mashlight/web/bundles/stickers/lukiep-imageviewer/js/show-image.js"></script>

<!-- END Imported JS -->
```

Figura 3.17: JS Links

La pagina così creata però non è ancora interattiva: i vari Stickersono semplicemente importati come lo sono le risorse indipendenti.

Per creare l'interazione lato client dei componenti bisogna affidarsi al `ProcessCreator`, che produce:

```
<style type="text/css">


    #sticker_node_5339fb971520a{
        position: absolute;
        width: 535px;
        height: 650px;
        top: 50px;
        left: 50px;
    }

    #sticker_node_5339fb972443d{
        position: absolute;
        width: 680px;
        height: 450px;
        top: 50px;
        left: 635px;
    }

</style>
```

Figura 3.18: CSS generato dinamicamente per il processo.

- un blocco di regole CSS, contenuto nel tag *<style>*, che hanno il compito di posizionare correttamente gli Stickers nella pagina (Figura 3.18);
- un blocco di funzioni JS contenute nel tag *<script>*, che hanno il compito di formare le relazioni tra i bottoni e le funzioni degli Stickers per ottenere il corretto funzionamento del mashup (Figura 3.19 a fronte).

Tutti questi servizi si appoggiano su un sistema di oggetti PHP che rappresentano le varie componenti della pagina.

```
<script>

    var node0 = 'node_5339fb971520a';

    var node1 = 'node_5339fb972443d';

    var urlparam = 'nothing';

    function readyFunction() {
        preloadFun();
        initGrid();
        showPreloadFun();
    }

    function macroFunction0() {
        var input1 = getInputText();

        codeAddress(input1);

        frameUpdate(input1);
    }
    function showSampleImage(sImage)
    {
        stringimage = sImage;
        loadImages(stringimage);
    }


</script>
```

Figura 3.19: JS generato dinamicamente per il processo.

È presente l'oggetto `Process`, che ha lo scopo iniziale di leggere i file XML dei processi (o interrogare il database) e creare la rappresentazione interna dei nodi e delle griglie.

Successivamente, fornisce un set di metodi che servono a mantenere traccia del processo e delle variabili utilizzate.

L'oggetto `Grid`, invece, è la rappresentazione interna della griglia di un processo. Ha lo scopo di:

- salvare al suo interno tutti gli Stickers di cui è composta;
- fornire metodi per il recupero delle risorse interne, sia per quanto riguarda gli Stickers che la griglia stessa;
- fornire un set di funzioni per costruire componenti per la costruzione della pagina;
- fornire funzioni per creare collegamenti tra gli Stickers (per esempio associare bottoni a determinate funzioni JS dei nodi interni).

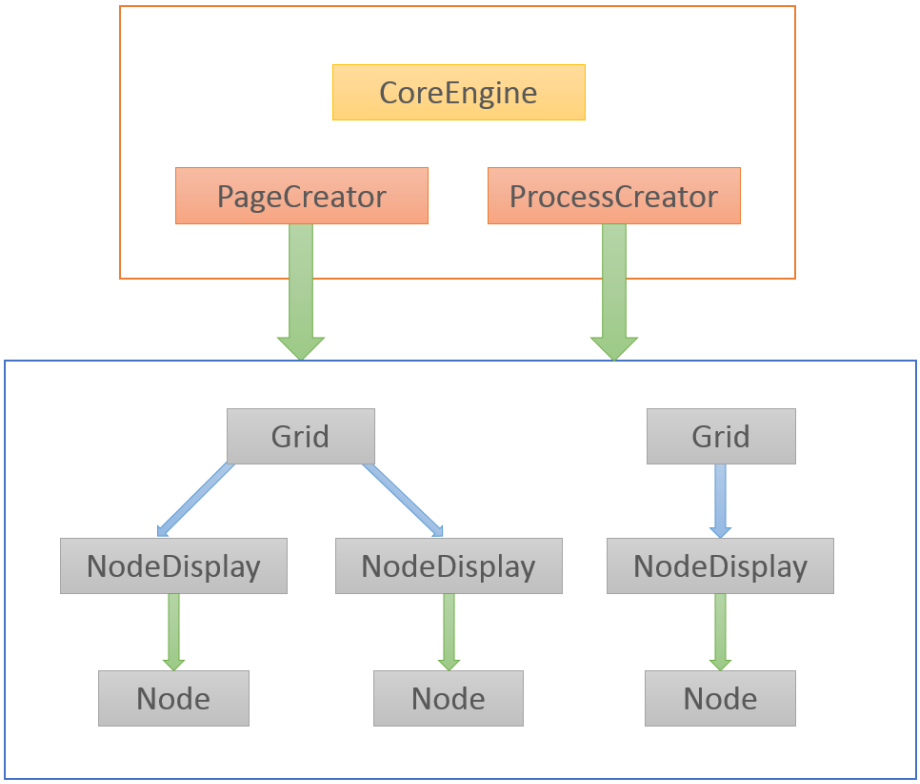


Figura 3.20: Collegamenti interni tra le classi PHP

Gli Stickers sono rappresentati all'interno di `Grid` tramite oggetti della classe `NodeDisplay`. Ha lo scopo di definire la rappresentazione del componente nella griglia, ed è composta da:

- una coppia di coordinate relative per il posizionamento;
- un oggetto `Node` che crea lo Sticker a partire dal file XML;
- il numero del nodo all'interno della griglia.

Il set di risorse lette tramite l'analisi del manifesto XML dello Sticker, viene salvato all'interno della classe `Node`.

Per completezza si mostra la struttura della classe nel file PHP.

Listing 3.6: Node.php

```php
<?php

namespace Lukiep\MashLightBundle\Core\Classes;

use Symfony\Component\Finder\Finder;
use Symfony\Component\DomCrawler\Crawler;
use Lukiep\MashLightBundle\Core\Utils;

class Node {

    ## Node Attribute ##
    private $isValid;
    private $nodeName;
    private $nodeFolder;
    private $xmlConfig;
    private $uniqueID; // Node ID
    private $width = 400;
    private $height = 400;


    ## Node Utility ##
    // Navigate the XML Documents
    private $crawler;
    private $stickersRoot;
    private $stickersRelative;
    private $utils; // Class for utility functions

    ## Node Resources ##
    public $divIDs = array();
    public $jsURLs = array();
    public $cssURLs = array();
    public $HTMLContentsCode = "";
    public $onloadFunctions = array();
    public $onclickFunctions = array();
    public $updateFunctions = array();
    public $twigFiles = array();

    /**
     * Class Constructor
```

```
40      */
41     public function __construct(Utils $utils) {
42         [...]
43     }
44
45     /**
46      *
47      * @param type $folderURL The Sticker folder name
48      */
49     public function loadXMLConfig($folderURL) {
50
51         [...]
52     }
53
54
55     ######### PRIVATE FUNCTIONS ##########
56
57     // Populate the Node object
58     private function loadAbouts(){
59       [...]
60     }
61
62     private function loadComponents(){
63         [...]
64     }
65
66     private function loadFunctions(){
67         [...]
68     }
69
70     ######### Additional Functions ############
71
72
73     ######## GETTERS E SETTERS ##########
74
75     public function getWidth() {
76         return $this->width;
77     }
78
79     public function getHeight() {
80         return $this->height;
81     }
82
83     public function getNodeName() {
84         return $this->nodeName;
85     }
86
87     public function getUniqueID() {
88         return $this->uniqueID;
89     }
```

```
90
91      public function haveDivID(){
92          if(empty($this->divIDs)){
93              return false;
94          }
95          else
96              return true;
97      }
98
99      public function getDivIDs() {
100         return $this->divIDs;
101     }
102
103     public function getJsURLs() {
104         return $this->jsURLs;
105     }
106
107     public function getCssURLs() {
108         return $this->cssURLs;
109     }
110
111     public function haveHTMLBlock(){
112         return (!empty($this->HTMLContentsCode)) ? true : false;
113     }
114
115     public function getHTMLContentsCode() {
116         return $this->HTMLContentsCode;
117     }
118     public function getOnloadFunctions() {
119         return $this->onloadFunctions;
120     }
121
122     public function haveTwigFiles(){
123         return (!empty($this->twigFiles)) ? true : false;
124     }
125
126     public function getTwigFiles() {
127         return $this->twigFiles;
128     }
129 }
```

Capitolo 4

Casi di Studio

> Se la conoscenza può creare dei problemi, non è con l'ignoranza che possiamo risolverli.
>
> Isaac Asimov

I seguenti casi d'uso hanno ambiti di utilizzo differenti, per poter meglio comprendere le potenzialità del framework.

Si inizia con un caso molto semplice, per prendere famigliarità con le meccaniche interne (Capitolo 4.1 a pagina 63).

Seguono poi una serie di casi di studio frutto della collaborazione portata avanti in questi mesi con Mattia Moretta:

- Caso 4.2 a pagina 69;
- Caso 4.3 a pagina 73;
- Caso 4.4 a pagina 77;
- Caso 4.5 a pagina 78;

Infine un caso d'uso tipicamente Social e tuttora in fase di completamento, e di cui verrà analizzato solo il mockup (Capitolo 4.6 a pagina 83).

Tramite le funzionalità messe a disposizione da Social Mashlight, si sono potute creare elaborate interfacce grafiche per l'utilizzo dei servizi per il controllo di un APR. Nei casi d'uso seguenti sono stati adoperati due Mashup: *Drone Sample* e *Drone Exploration*.

Sono stati sviluppati ad hoc degli Stickers che si occupano di interfacciarsi con i servizi REpresentational State Transfer (REST) dell'APR, ed altri di uso più generico come l'*Image Viewer* o il blocco social *Flickr*.

Si è deciso di assegnare un nome caratteristico ad ogni caso di studio, in questo modo viene associato facilmente alle operazioni svolte dal velivolo o l'obiettivo che si prefigge il mashup.

Per il primo caso verranno riportati i file di manifesto del blocco. Essendo uguale la struttura del file per tutti gli Stickers, non verrà riportata per concisione per gli altri casi.

4.1 City Viewer

Il primo mashup creato con Social Mashlight è un form di ricerca per le informazioni su una città del mondo, a cui è stato dato il nome di **City Viewer**.

I componenti sono i seguenti. Verranno riportati per questo caso d'uso i manifesti XML dei blocchi.

Input Box

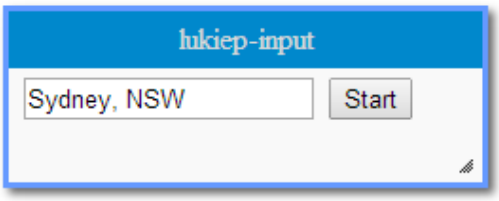


Figura 4.1: Input box per la scelta della città.

Questo componente è stato realizzato semplicemente come un form HTML: l'utente inserisce il nome della città e scatena un'azione cliccando sul pulsante 'Start'.

L'azione viene catturata poi dal framework, che esegue i successivi passi del mashup.

Viene riportato il file di manifesto del blocco.

Listing 4.1: config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3
4  <block>
5      <about>
6          <name>lukiep-input</name>
7          <icons>
8              <icon size="128">images/map-icon.png</icon>
9          </icons>
10         <folder>lukiep-input</folder>
11         <style>
12             <size width="250" height="50" />
13         </style>
14     </about>
15
16     <components>
17         <javascript-urls>
18             <javascript-url type="relative">js/input.js</javascript-url>
19         </javascript-urls>
20
21         <css-urls></css-urls>
22
23         <div-ids></div-ids>
```

```
24
25          <html-contents></html-contents>
26
27          <twig-files>
28              <twig-file>Lukiep:input.html.twig</twig-file>
29          </twig-files>
30      </components>
31
32      <functions>
33          <onload></onload>
34
35          <update-functions></update-functions>
36
37          <outlink>
38              <function name='sendInputText'>
39              <variable type="string" name="address" isArray="false"></variable
                    >
40           </function>
41          </outlink>
42
43      </functions>
44  </block>
```

Google Maps

Un semplice componente che è stato sviluppato con l'utilizzo delle API di Google.

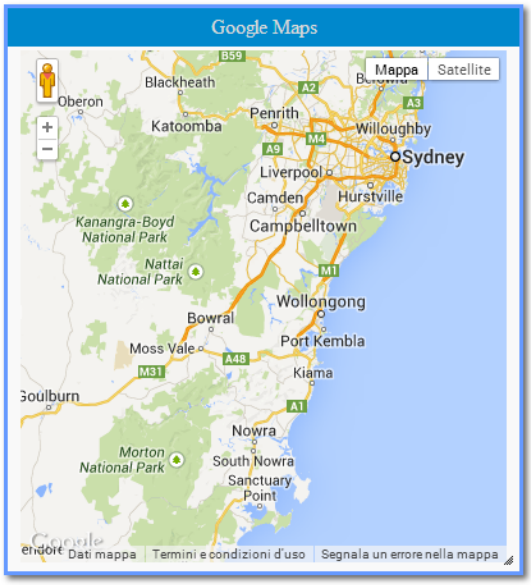


Figura 4.2: Componente che visualizza le città sulla mappa.

Tramite la funzione di inizializzazione, caricata da Social Mashlight all'avvio della griglia, viene configurato il componente. Mediante la funzione `codeAddress`, viene ricevuta una stringa con il nome della città e viene mostrata la relativa mappa.

Listing 4.2: config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<block>
    <about>
        <name>Google Maps</name>
        <icons>
            <icon size="128">images/map-icon.png</icon>
        </icons>
        <folder>google-maps</folder>
    </about>

    <components>
        <javascript-urls>
            <javascript-url type="relative">js/mapFunctions.js</javascript-
    url>
            <javascript-url type="absolute">https://maps.googleapis.com/maps/
                api/js?key=AIzaSyDW-TFwAT_lSKqHNUf1HW6hMIMsbEEHiFg&#38;sensor
                =false</javascript-url>
        </javascript-urls>

        <css-urls>
            <css-url>css/mapsCSS.css</css-url>
        </css-urls>

        <div-ids>
            <div-id name="map-canvas" prefix="map-canvas" />
        </div-ids>

        <html-contents></html-contents>

         <twig-files></twig-files>
    </components>

    <functions>
        <onload>
            <function name="initialize">
                <variable name="map-canvas" type="DivId" isArray="false"></
                    variable>
            </function>
        </onload>

        <variables></variables>

        <update>
            <function name="codeAddress">
                <variable type="string"></variable>
            </function>
        </update>
    </functions>
</block>
```

Wikipedia

Questo componente è realizzato utilizzando codice HTML direttamente inserito nel documento XML. È un semplice *iframe* in cui è visualizzata la versione mobile della nota enciclopedia informatica.

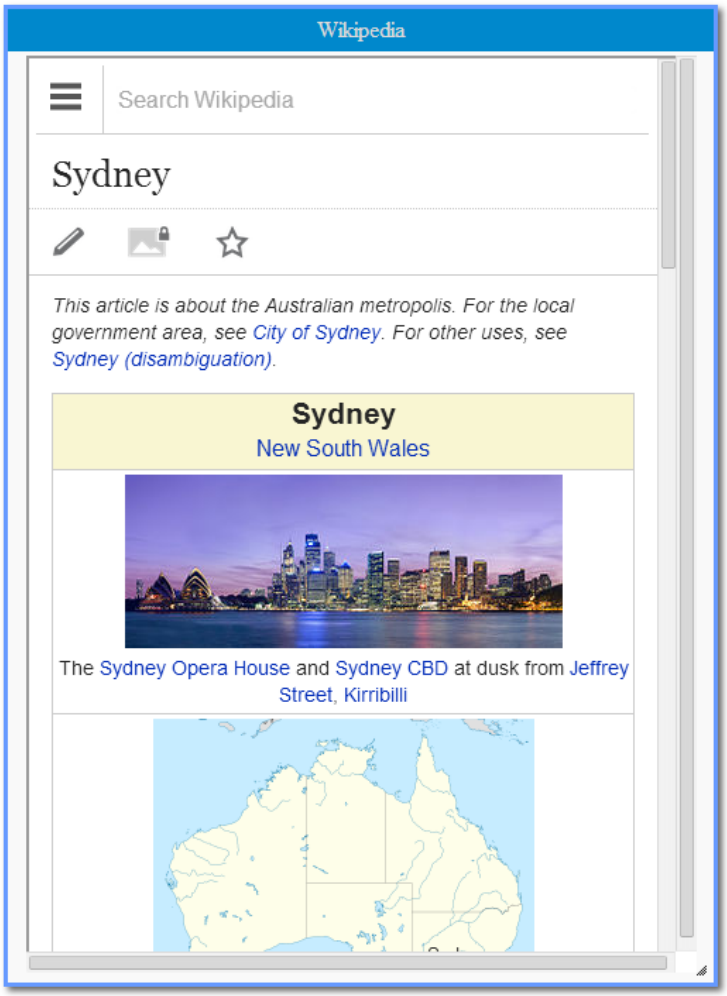


Figura 4.3: Componente che mostra le informazioni su Wikipedia.

A parte le risorse CSS e le immagini, lo Sticker viene creato dalla lettura del seguente file.

Listing 4.3: config.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--
3 This file is part of the Symfony package.
4
5 (c) Luca Chieppa <gridme@chieppaluca.it>
6
7 For the full copyright and license information, please view the LICENSE
8 file that was distributed with this source code.
9 -->
```

```
10
11
12 <block>
13     <about>
14         <name>Wikipedia</name>
15         <icons>
16             <icon size="128">images/map-icon.png</icon>
17         </icons>
18         <folder>wikipedia</folder>
19         <style>
20             <size width='500' height='650' />
21         </style>
22     </about>
23
24     <components>
25         <javascript-urls>
26             <javascript-url type="relative">js/wikipedia.js</javascript-url>
27         </javascript-urls>
28
29         <css-urls>
30             <css-url>css/wikipedia.css</css-url>
31         </css-urls>
32
33         <div-ids>
34         </div-ids>
35
36         <html-contents>&lt;div id=&quot;wiki-canvas&quot;&gt;&lt;iframe src
               =&quot;http://en.m.wikipedia.org/wiki/Sydney&quot; &gt;&lt;/
               iframe&gt;&lt;/div&gt;</html-contents>
37
38         <twig-files></twig-files>
39     </components>
40
41     <functions>
42         <onload>
43         </onload>
44         <variables></variables>
45         <update-functions>
46             <function name="codeAddress">
47                 <variable type="string"></variable>
48             </function>
49         </update-functions>
50     </functions>
51 </block>
```

Esecuzione della Simulazione

1. Viene inserito nel campo dello Sticker 'lukiep-input' il nome della città da ricercare.

2. Si clicca sul pulsante 'Start'.

3. I componenti 'Google-Maps' e 'Wikipedia' si aggiornando automaticamente mostrando le informazioni geografiche e storiche della città.

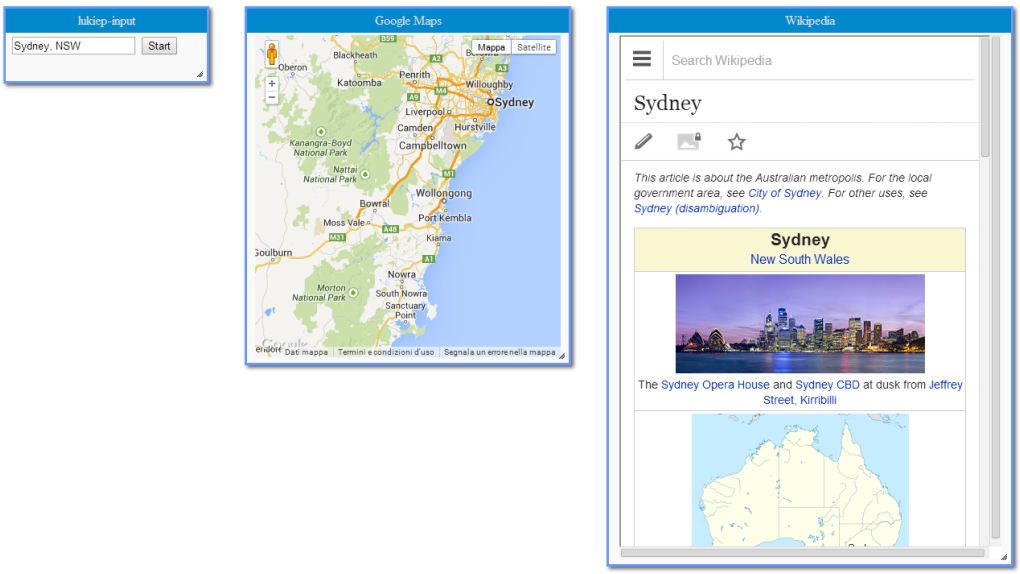


Figura 4.4: Esempio di come utilizzare il mashup per visualizzare le informazioni su Sydney.

4.2 Drone Sample

In questo caso d'uso è stato realizzato un mashup che implementa il servizio di **Sample** del APR.

Il mashup è composto da due Stickers:

Drone Form

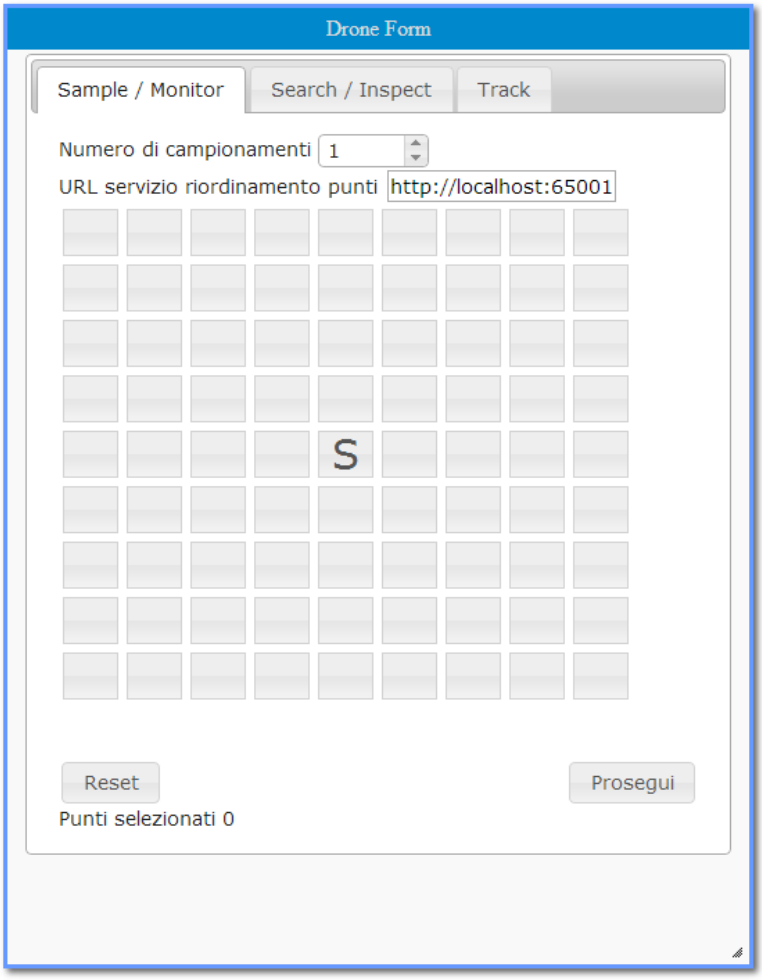


Figura 4.5: Sticker dedicato al controllo del APR per le sue funzionalità di Sample/Monitor.

In Figura 4.5 è mostrato il componente con cui è possibile fornire al drone le coordinate su cui effettuare le proprie rilevazioni.

Ogni punto sulla griglia rappresenta una coordinata in un'area di ricerca definita a priori dal servizio terzo. Selezionando un punto si comanda al drone di effettuare una o più rilevazioni in quella determinata coordinata, producendo una foto per ogni operazione.

Il punto centrale, marcato con l'etichetta 'S' (per 'Start'), è la coordinata di origine da cui parte l'APR. Tutti gli altri punti rappresentano le coordinate relativa a partire da quella centrale.

Tramite lo spinner 'Numero di Campionamenti' si comanda all'APR quanti rilevamenti fare su ogni coordinata.

Il form 'URL servizio riordinamento punti' invece è un indirizzo che viene fornito al servizio REST per riordinare le coordinate selezionate in modo da poter creare un percorso ottimizzato: infatti non viene tenuto conto dell'ordine in cui si selezionano i punti sulla griglia, ma dai punti si crea il percorso minimo.

Image Viewer

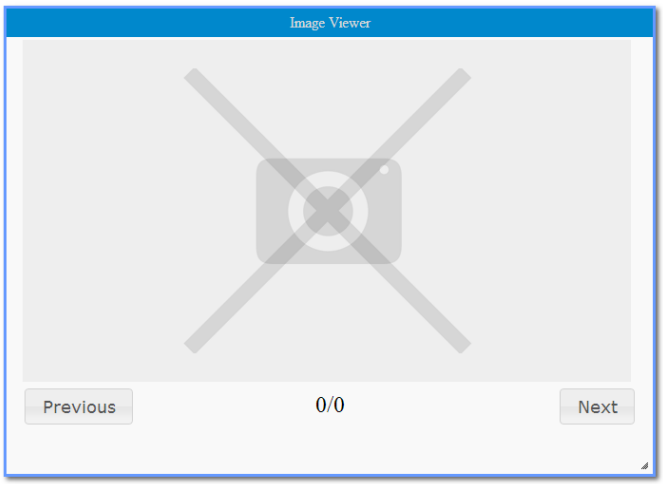


Figura 4.6: Sticker dedicato alla gestione di una galleria di immagini.

Il blocco in Figura 4.6 è uno Sticker interno a Social Mashlight che si occupa di ricevere un insieme di immagini e di visualizzare una galleria.

Questo componente non è legato all'utilizzo del APR ma viene impiegato al fine di mostrare all'utente le immagini rilevate.

Esecuzione della Simulazione

1. Sulla griglia, l'utente seleziona un gruppo di punti. I pulsanti selezionati si coloreranno di arancione e nella parte bassa dello sticker verranno visualizzate le coordinate relative a partire dal punto di partenza centrale (contrassegnata dalla coppia $x = 0$ e $y = 0$);

2. Cliccando sul pulsante 'Prosegui', l'APR parte per l'attività di 'Sample'. Sulla coordinata dove è stato possibile effettuare la rilevazione, apparirà un'icona che notifica la disponibilità di una o più foto.

3. Cliccando sulla coordinata contrassegnata dall'icona di una immagine, il sistema visualizzerà il set di immagini nel blocco *Image Viewer*. Il nu-

mero di foto dipende dal parametro impostato nello spinner ‘Numero di campionamenti’.

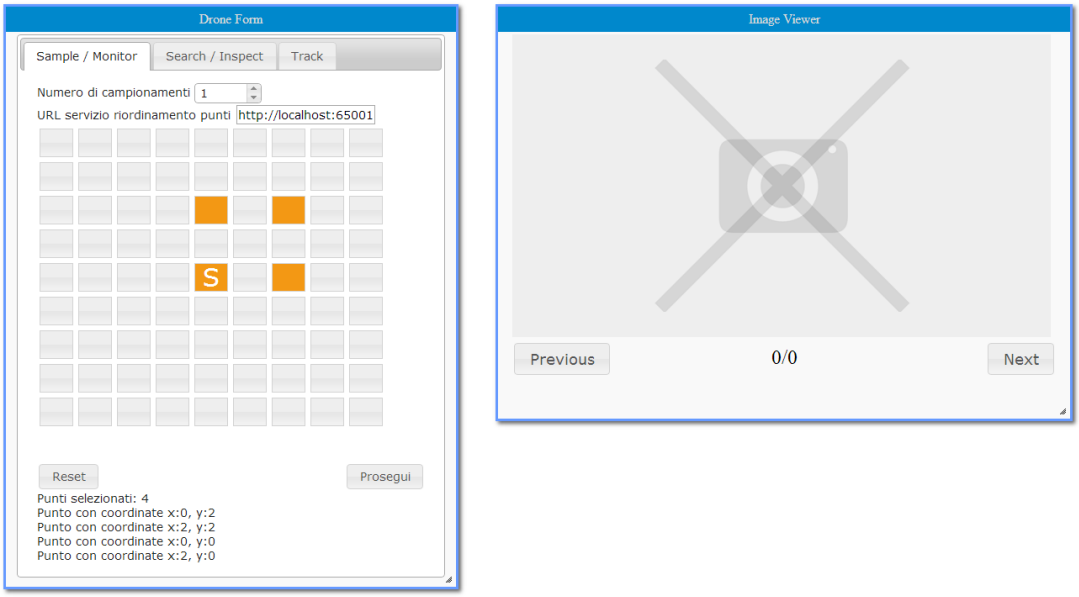


Figura 4.7: Selezione delle coordinate sulla griglia.

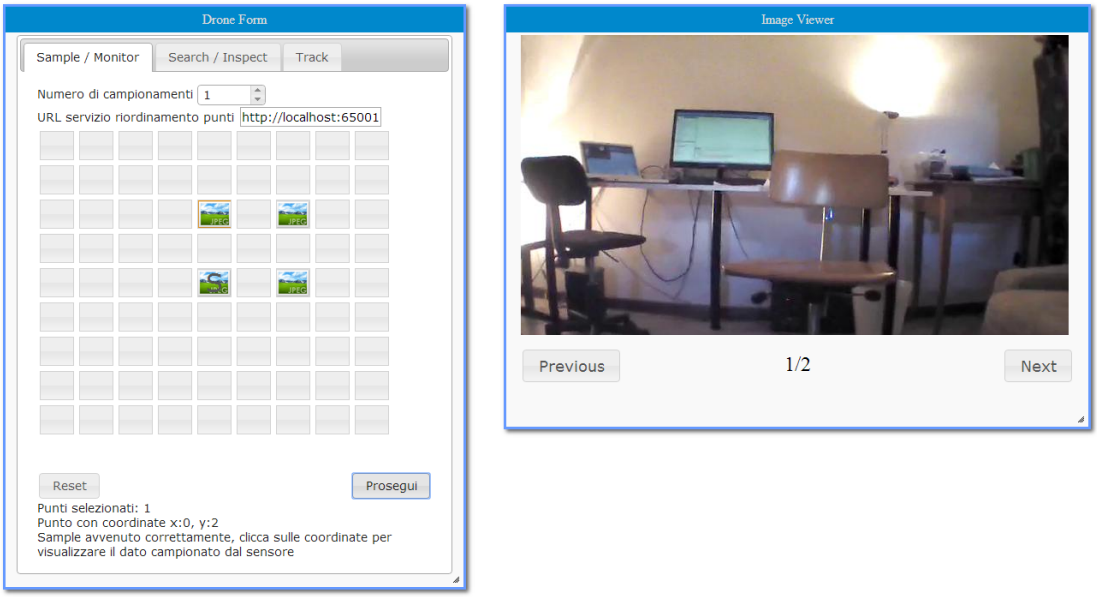


Figura 4.8: Visualizzazione delle immagini ottenute dal campionamento.

4.3 Drone Search

Per il secondo caso d'uso dedicato agli APR, si utilizza lo stesso mashup utilizzato per il precedente, ma viene implementato il servizio di **Search/Inspect**.

Questo mashup utilizza i seguenti tre Stickers:

Drone Form

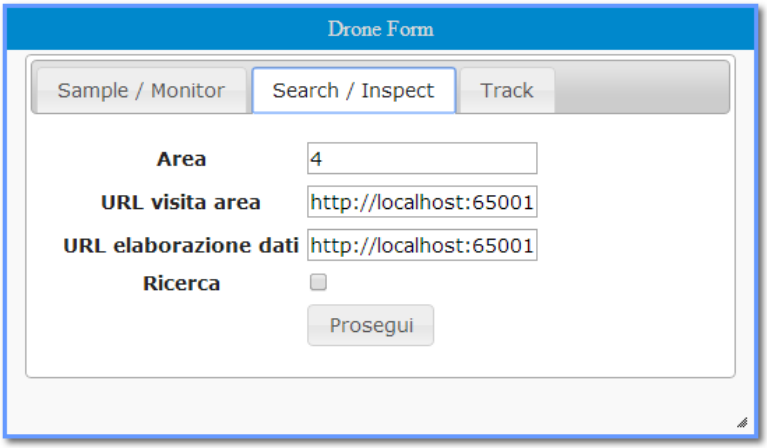


Figura 4.9: Sticker dedicato al controllo del APR per le sue funzionalità di Search/Inspect, disponibili mediante l'utilizzo della seconda tab del componente.

In Figura 4.9 è mostrato il componente *Drone Form*, ma è stata selezionata la seconda tab, etichettata con 'Search/Inspect'.

Al posto della griglia è presente un form in cui possono essere definiti i collegamenti ai servizi accessori per il velivolo.

- il parametro **Area** definisce i metri quadrati in cui il velivolo si muove;
- **URL visita area** definisce il servizio che data un'area genera i punti da visitare al suo interno sotto forma di griglia;
- **URL elaborazione dati** rinvia al servizio che elabora i risultati e risponde se un determinato punto è di interesse o meno.
- il checkbox **Search**, se spuntato, indica al servizio che si vuole effettuare una *Ricerca*, se non selezionato viene invece effettuata una *Ispezione*.

Drone Grid

Il blocco in Figura 4.10 nella pagina seguente ha la stessa disposizione di pulsanti come nel *Drone Form*, disposti in una griglia $9x9$. Su questa griglia non è possibile

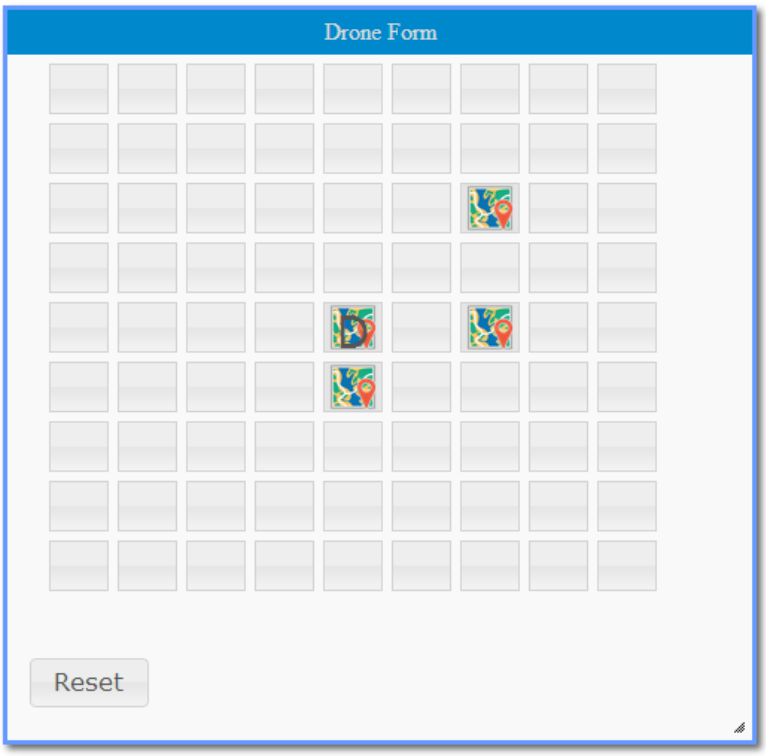


Figura 4.10: Sticker dedicato alla visualizzazione dei risultati di una Search/Inspect su una griglia

impartire istruzioni al velivolo, ma è utilizzata per la presentazione dei risultati. I punti di interesse rilevati dall'APR vengono contraddistinti da un'icona che rappresenta una mappa.

Image Viewer

Questo Sticker è lo stesso utilizzato nel primo caso d'uso (vedere il capitolo 4.2 a pagina 69).

Esecuzione della Simulazione

1. Nel form, l'utente definisce tutti i parametri per configurare il servizio REST;

2. Cliccando sul pulsante 'Prosegui', l'APR parte per l'attività di *Search* (o *Inspect*).

   La griglia del mashup verrà aggiornata mostrando una nuova pagina con i due Stickers precedentemente descritti. Sulla coordinata dove è stato possibile effettuare la rilevazione apparirà un'icona con una mappa, per informare l'utente che in quel punto sono disponibili dei risultati (nel nostro caso delle immagini).

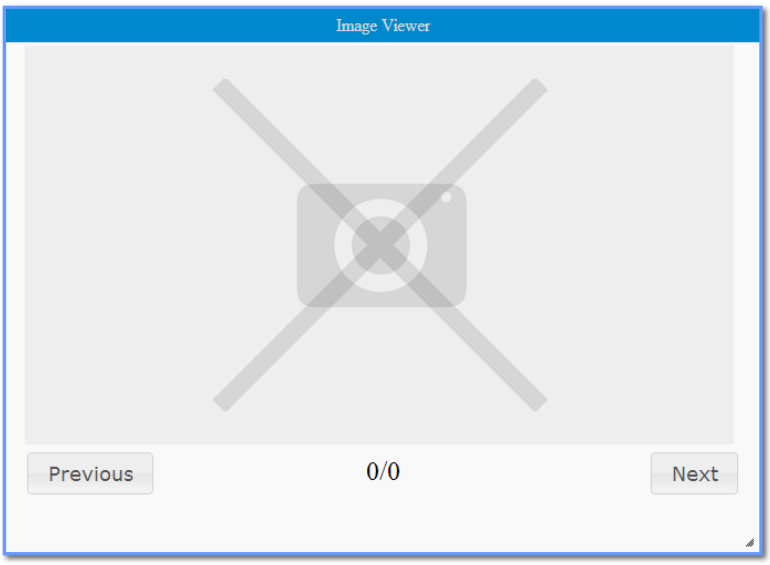


Figura 4.11: Sticker dedicato alla gestione di una galleria di immagini

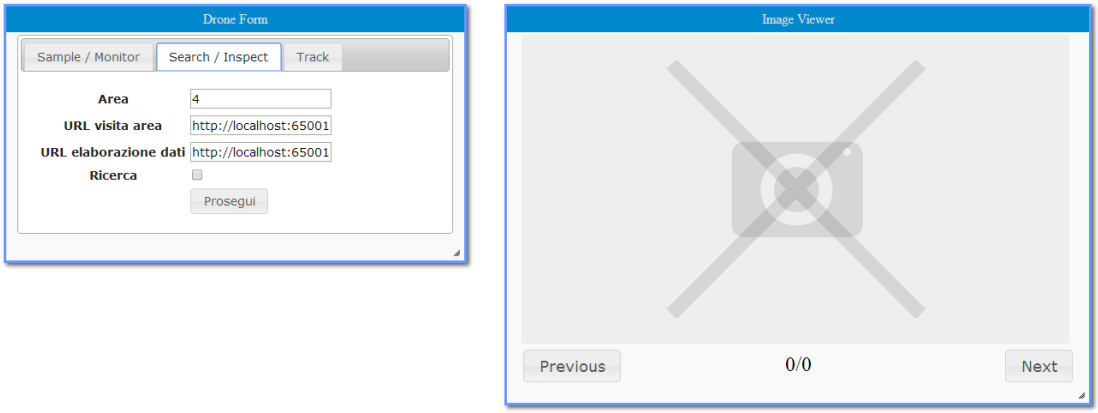


Figura 4.12: Definizione di tutti i parametri per avviare il servizio; l'*Image Viewer* presente nella schermata non viene utilizzato per questo tipo di operazione.

3. Cliccando sulla coordinata contrassegnata dall'icona, il sistema visualizzerà il set di immagini nel blocco *Image Viewer*. Nel caso d'uso in questione è lecito aspettarsi una sola fotografia.

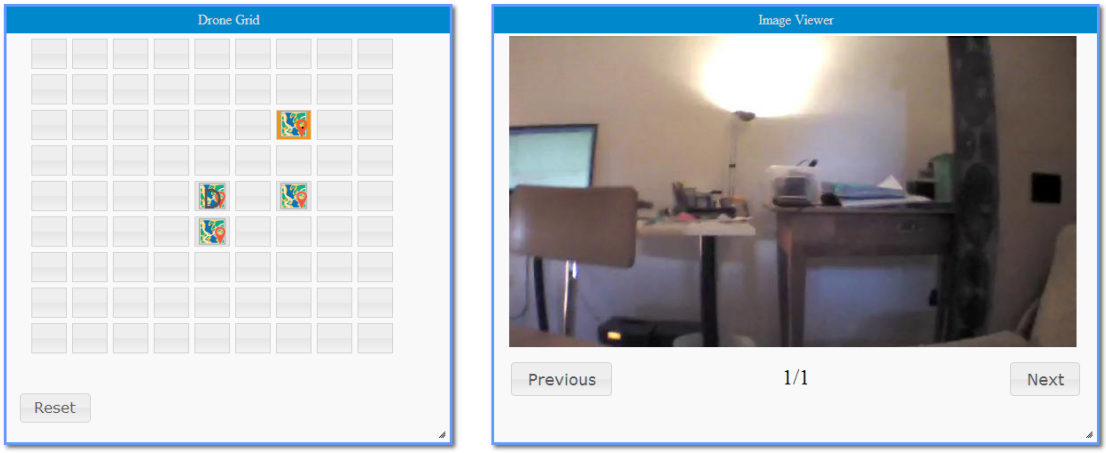


Figura 4.13: Visualizzazione delle immagini nella seconda pagina del mashup.

4.4 Drone Track

La terza scheda presente all'interno dello Sticker *Drone Form* permette all'utente di interfacciarsi con il servizio **Track**.

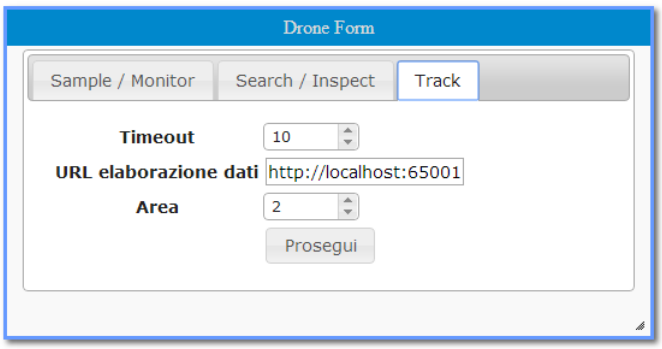


Figura 4.14: Selezione di un'area di mappa e parametri di ingresso per il servizio.

Prevede l'inserimento dei seguenti parametri:

- il **timeout** espresso in secondi, tempo massimo dopo il quale il servizio deve interrompere l'inseguimento dell'obiettivo;
- il **servizio web** relativo all'elaborazione delle immagini campionate dal velivolo;
- **l'estensione massima dell'area**, espressa in metri quadrati, entro la quale il velivolo può rimanere durante l'inseguimento; nel caso in cui l'oggetto inseguito fuoriesca dalla superficie indicata, il processo viene interrotto.

La chiamata al servizio comporta l'invocazione dello Sticker che mostra la griglia di punti (descritto in precedenza nel Capitolo 4.3 a pagina 73). Sono mostrati in tempo reale i punti nei quali il velivolo si è spostato; ad ogni coordinata viene associata l'immagine precedentemente campionata.

Il flusso delle azioni eseguite dall'utente per poter utilizzare il servizio è il seguente:

1. inserimento dei parametri necessari;
2. invocazione del servizio a seguito della pressione del pulsante 'Prosegui';
3. visualizzazione dei punti che costituiscono il percorso intrapreso dal velivolo per l'inseguimento dell'oggetto.

4.5 Drone Exploration

Questo mashup rappresenta un'evoluzione del caso d'uso del Capitolo 4.3 a pagina 73, dove vengono utilizzati i servizi di **Search/Inspect**. Nella sua composizione, sono stati utilizzati i seguenti Stickers:

Google Map

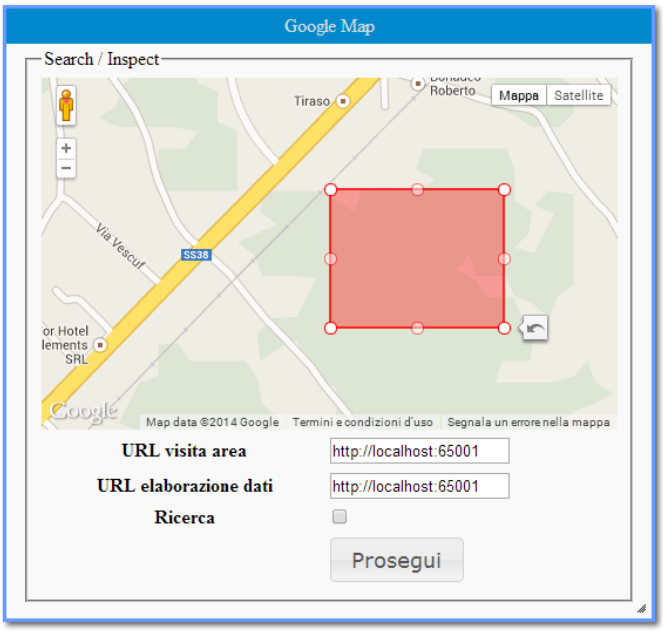


Figura 4.15: Selezione di un'area di mappa e parametri di ingresso per il servizio.

Questo form di inserimento fornisce al servizio del APR le informazioni necessarie all'avvio della ricerca/ispezione. Andranno inserite le seguenti informazioni:

- l'individuazione dell'**area con coordinate geografiche** nella quale ricercare il pattern, selezionabile attraverso un'interfaccia grafica realizzata con l'utilizzo delle API relative al servizio Google Maps;
- il servizio web che dovrà generare un **insieme di punti da visitare**;
- il servizio web che **analizzerà i rilevamenti** ottenuti nei punti interessati;
- la **distinzione** tra il servizio di ricerca ed ispezione.

Lo stesso Sticker sarà utilizzato nel corso dell'esecuzione del processo per la visualizzazione delle coordinate geografiche relative al punto in questione, che verranno mostrate all'utente all'interno della mappa di Google Maps con appositi *Marker* interattivi (Figura 4.16 a fronte).

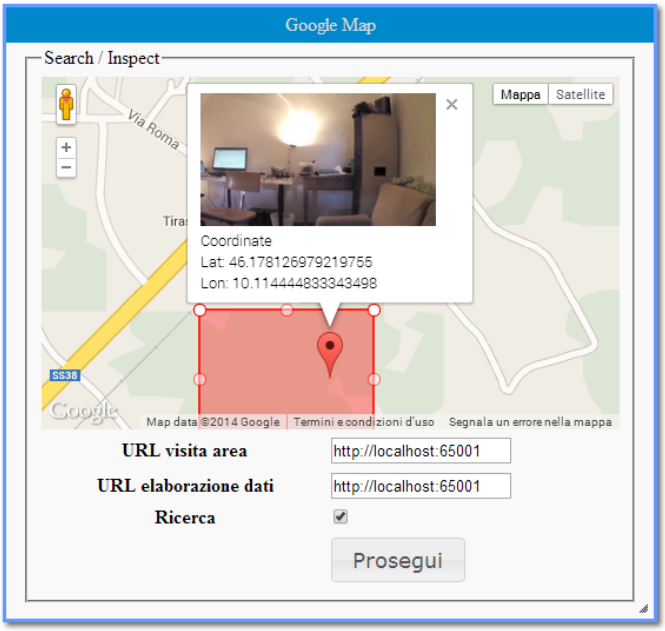


Figura 4.16: Dopo aver trovato i punti di interesse, è possibile visualizzarli come Marker su questo sticker.

Drone Grid

È lo stesso componente utilizzato nel caso d'uso del Capitolo 4.3 a pagina 73: una griglia di punti, centrata nella posizione di decollo del velivolo, nella quale sono evidenziate le coordinate dove sono stati ottenuti riscontri positivi.

Essa è composta da un numero fisso di celle (pari a 81), le cui coordinate sono rapportate all'area d'interesse selezionata in precedenza nella mappa di Google Maps. Essa risulta utile per avere un'idea approssimata della posizione dei riscontri all'interno dello spazio.

Flickr

Lo Sticker mostrato in Figura 4.17 nella pagina seguente consente, attraverso la pressione sul pulsante 'Send to Flickr', una semplice ed immediata condivisione delle immagini, caricate dal sistema sul famoso portale web dedicato alla gestione delle immagini.

Image Viewer

Stesso componente utilizzato nei casi d'uso precedenti: un semplice visualizzatore di immagini, indispensabile per poter visionare i campionamenti identificati nei punti evidenziati sulla griglia.

Figura 4.17: Sticker dedicato al caricamento remoto delle immagini sul portale web Flickr

Esecuzione della Simulazione

Il processo durante l'interazione con l'utente, procede secondo le seguenti fasi:

1. viene selezionata un'area rettangolare mediante l'utilizzo del componente Google Maps;
2. vengono forniti gli URL relativi ai servizi esterni, eventualmente selezionando la checkbox 'Ricerca' se interessati al relativo servizio;
3. tramite la pressione sul pulsante 'Prosegui' si avvia l'attività dell'APR e l'elaborazione dei dati correlati;
4. vengono visualizzati a schermo i riscontri positivi all'interno della griglia dei punti;
5. selezionando i punti contrassegnati da icone colorate, si ottiene un'immediata visualizzazione della coordinata all'interno della mappa di Google Maps e viene mostrata l'immagine campionata all'interno dell'*Image Viewer*; se non è presente un'immagine, viene visualizzata la sola coordinata geografica.
6. l'utente a questo punto può eventualmente effettuare il salvataggio remoto dei punti mediante l'utilizzo dello Sticker dedicato a Flickr attraverso la pressione del pulsante 'Send to Flickr'.

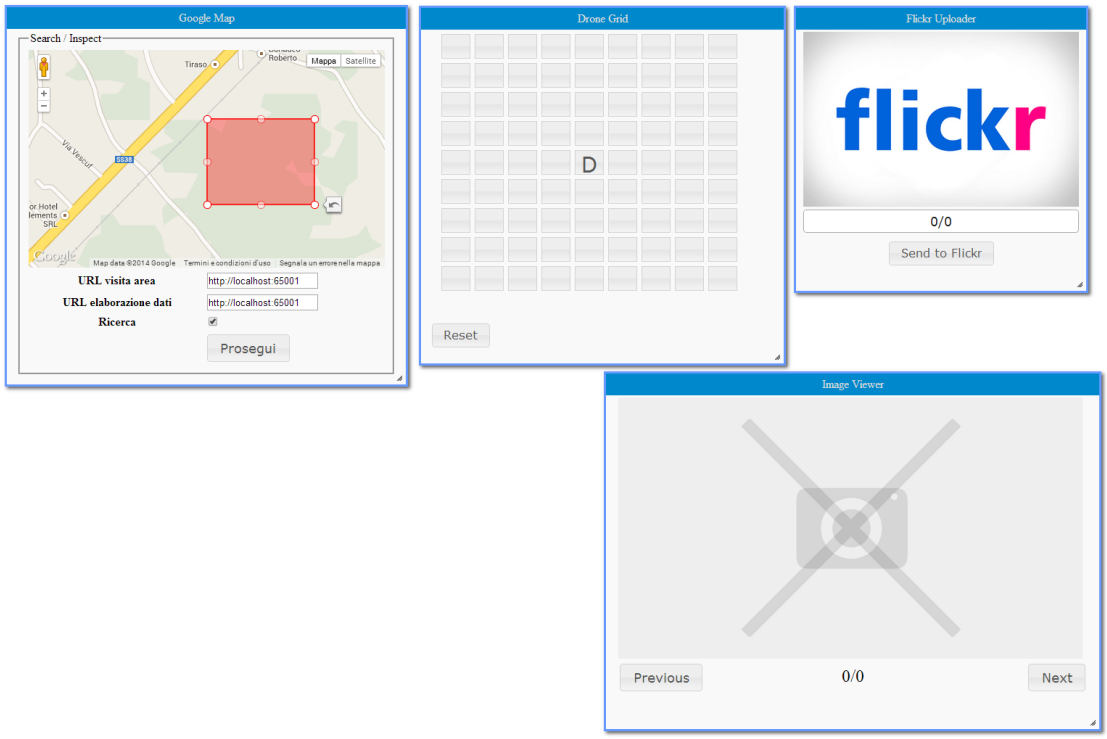


Figura 4.18: Definizione dell'area di esplorazione su Google Maps

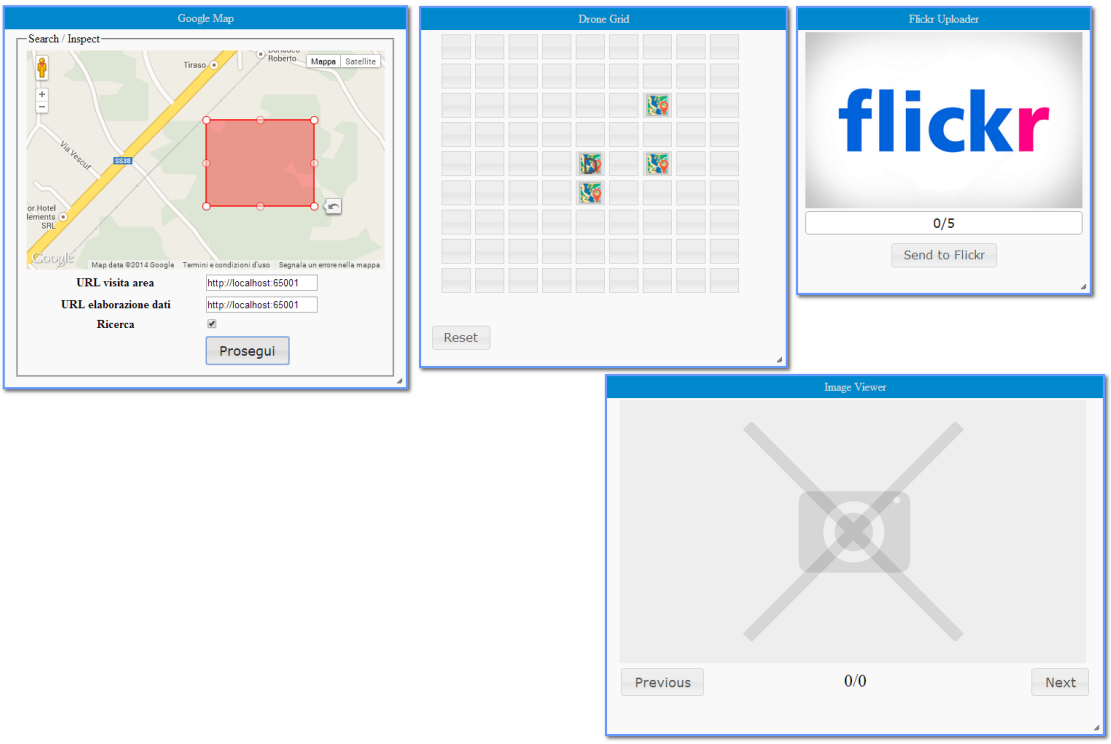


Figura 4.19: Vengono evidenziati i punti di interesse

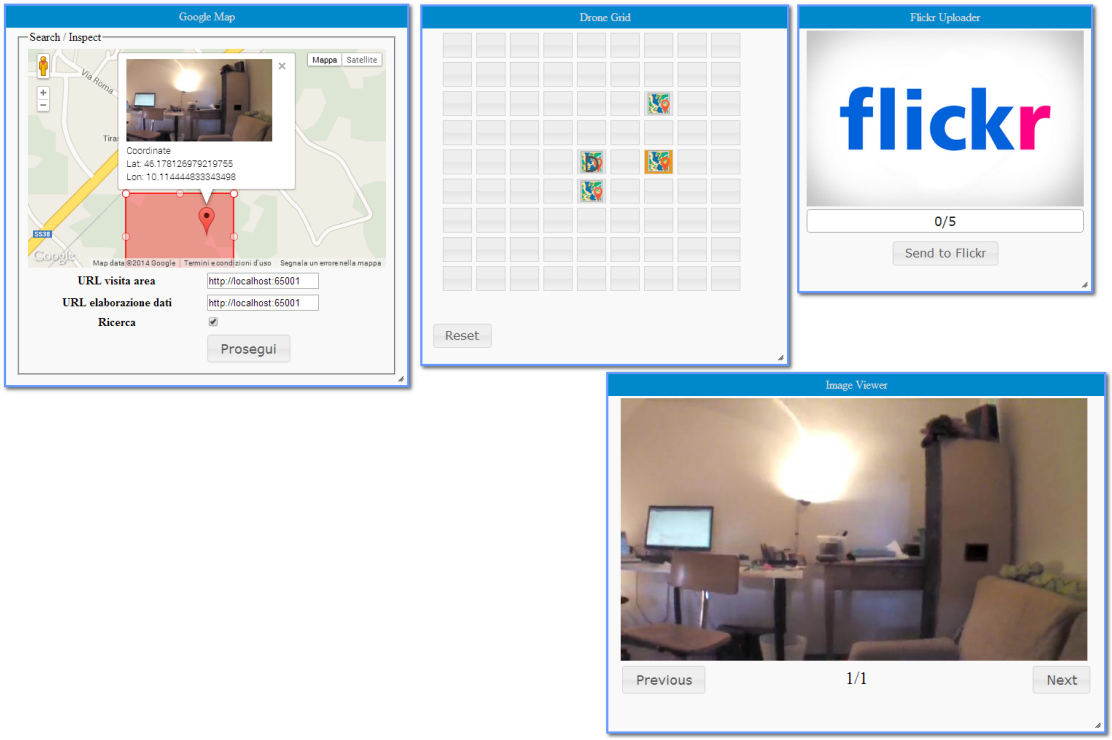


Figura 4.20: Visualizzazione d'insieme dei risultati dell'esplorazione

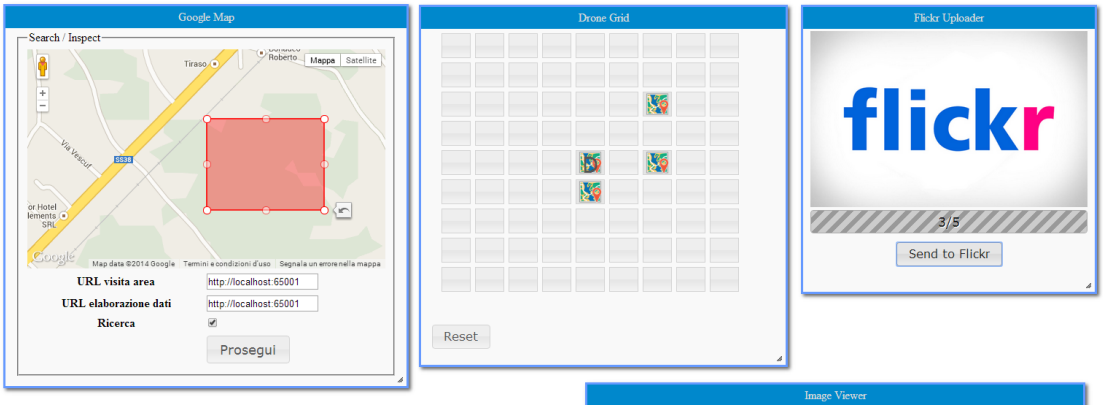


Figura 4.21: Caricamento in remoto delle rilevazioni

4.6 Social Viewer

Nonostante sia già stata evidenziata la possibilità di sfruttare componenti *Social* come nello Sticker Flickr (Capitolo 4.5 a pagina 79), si vuole creare un caso d'uso dedicato all'interazione con più *Social Network*.

L'idea è quella di visualizzare una serie di bacheche in cui visualizzare le attività di un particolare utente sui tre *Social Network* più diffusi in Italia: Facebook, Google+ e Twitter.

Non è possibile al momento mostrare il codice o i componenti utilizzati, ma si mostra di seguito il mockup ideato per il suo futuro sviluppo.

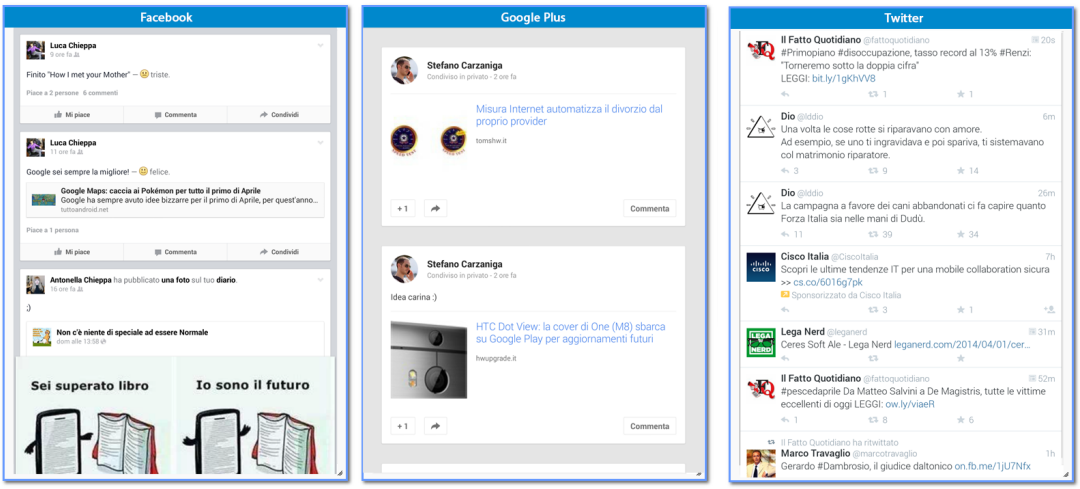


Figura 4.22: Visualizzazione dei miei profili su una singola schermata di Social Mashlight.

Capitolo 5

Conclusioni

> Il mondo non l'abbiamo avuto
> in eredità dai nostri genitori, ma
> in prestito dai nostri figli.
>
> Proverbio Indiano

Con questo lavoro di tesi è stato avviato un processo di profondo rinnovamento della piattaforma Mashlight. Dalla precedente versione è stata mantenuta la filosofia di *blocco* e di *processo*: l'idea che per gestire informazioni e funzionalità così diverse bisognasse focalizzarsi solo sul 'quando' attivare i componenti e su 'quali' dati trasferire.

Anche se l'idea originale era quella di aggiornare il framework attuale, è stato necessario un lavoro di riprogettazione completa, cha ha portato a:

- migliorare sensibilmente l'efficienza generale con cui si creano e si visualizzano i mashup;
- creare un software facilmente manutenibile in quanto sviluppati su piattaforme che stanno entrando a fare parte degli standard del Web.

Va messo in evidenza che Social Mashlight è ancora immaturo per un rilascio imminente, in quanto non sono ancora state sviluppate quelle caratteristiche che permettano a più utenti un utilizzo in remoto.

Non è stata ancora introdotta un'interfaccia grafica semplificata per la costruzione dei mashup, che avviene solo mediante la lettura di un documento XML. Un progetto futuro può essere quello di realizzare una schermata di *building*, che tramite il trascinamento dei blocchi e il collegamento tra le loro connessioni, permetta il salvataggio in una base di dati consistente.

Nonostante la suddetta limitazione nella versione attuale, si è visto nei casi d'uso come questo framework sia maturo per l'impiego in ambiti di ricerca più disparati: dalla semplice composizione di componenti social, al controllo remoto di un APR.

Un primo obiettivo consiste nella creazione di un database con tecnologie come Propel o Doctrine, che permettano di mappare oggetti PHP direttamente sulla base di dati, con driver interni per gestire diversi tipi di DBMS (Mysql, Oracle etc).

Nonostante sia stata presa in considerazione l'idea di rendere ogni componente unico, ci sono ancora dei meccanismi di gestione degli identificativi che richiedono il passaggio materiale degli ID come parametri alle funzioni. Nell'ottica di un miglioramento nelle meccaniche interne, sarebbe buona cosa rendere trasparente allo sviluppatore di Sticker questa funzionalità, in modo che sia la piattaforma stessa a gestire le chiamate a funzioni e la creazione dei blocchi in modo indipendente.

Infine, sarebbe utile ottimizzare e documentare approfonditamente lo sviluppo degli Stickers, in modo da creare delle librerie che permettano la programmazione di questi componenti in modo veloce ed intuitivo.

Acronimi

> Il problema dell'umanità è che gli stupidi sono strasicuri, mentre gli intelligenti sono pieni di dubbi.
>
> B. Russell

A-D

API Application Programming Interface

APR Aeromobili a Pilotaggio Remoto

AJAX Asynchronous JavaScript and XML

CSS Cascading Style Sheets

DOM Document Object Model

C-N

HTML HyperText Markup Language

IoT Internet of Things

JS JavaScript

JSON JavaScript Object Notation

O-U

PHP PHP: Hypertext Preprocessor

REST REpresentational State Transfer

RFID Radio Frequency IDentification

SaaS Software as a Service

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

URL Uniform Resource Locator

X-Z

XML eXtensible Markup Language

W3C World Wide Web Consortium

Web World Wide Web

Bibliografia

[1] Annalisa. *Cos'é Atooma*. A cura di Cos'é. URL: http://www.xn--cos-8la.com/cose-atooma/.

[2] Facebook. *Developer Center*. URL: https://developers.facebook.com/.

[3] Fastweb, cur. *Internet delle cose*. URL: http://www.fastweb.it/web-e-digital/internet-delle-cose-cos-e-e-come-funziona/.

[4] Flickr. *App Garden*. URL: https://www.flickr.com/services/api/.

[5] Google. *Google Developers*. URL: https://developers.google.com/?hl=it.

[6] Mattia Moretta. «A service-oriented architecture for sensing and actuating UAVs». Tesi Magistrale in Ingegneria Informatica. Politecnico di Milano, 2014.

[7] *OAuth 2.0*. URL: http://oauth.net/.

[8] *OpenSocial*. URL: http://opensocial.org.

[9] Lorenzo Pantieri. *L'arte di scrivere in LaTeX 2ε*. Aracne editrice S.r.l, 2008.

[10] Giuseppe Rossitto e Danilo Sabato. «Mashlight 2.0: un framework lightweight per mashup». Tesi triennale in Ingegneria Informatica. Politecnico di Milano, 2008.

[11] Twitter. *Developers*. URL: https://dev.twitter.com/.

[12] Hardware Upgrade, cur. *Notizie*. URL: http://www.hwupgrade.it/.

[13] Wikimedia, cur. *Wikipedia*. URL: http://it.wikipedia.org/wiki/.