ALMA MATER STUDIORUM

UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

PROGETTO E SVILUPPO DI RESTFUL WEB API PER UN SISTEMA AZIENDALE DI MARCATURE DEGLI ACCESSI

Elaborata nel corso di: Sistemi Distribuiti

Tesi di Laurea di:

ANDREA RINALDI

Relatore:

Prof. ANDREA OMICINI

Co-relatore:

Dott. MAURIZIO PENSATO

ANNO ACCADEMICO 2013 2014

SESSIONE III

Alla mia Chiarina

Indice

Glossario **III**

1 Introduzione **3**

2 Il World Wide Web **5**

2.1 Overview 5

2.2 Caratteristiche generali 7

2.3 Lo stile architetturale ReST del Web 10

2.3.1 I vincoli architetturali 10

2.3.2 Gli elementi architetturali 13

2.3.3 ReST: Representational State Transfer 18

2.3.4 ReST applicato ai protocolli del Web 20

2.4 Il Web 2.0 28

3 Web API **31**

3.1 Overview 31

3.1.1 I modelli architetturali più comuni 34

3.1.2 Disambiguazione 35

3.2 I Web Service 35

3.3 ReSTful web API 39

3.3.1 HATEOAS 40

3.3.2 Il processo di progettazione 44
3.4 Conclusioni 48

4 Caso di studio **51**
4.1 Overview sul progetto 52
4.1.1 La struttura 54
4.2 Analisi del dominio 55
4.3 Analisi dei requisiti 57
4.3.1 I requisiti della web application 58
4.3.2 Casi d'uso della web application 59
4.4 Analisi logica 61
4.5 La progettazione delle web API di io@spot 64
4.5.1 Definizione dei descrittori semantici 64
4.5.2 Diagramma degli stati applicativi 67
4.5.3 Definire i nomi 68
4.5.4 Scelta del media-type 69
4.5.5 Scrivere un profilo per le semantiche del dominio 77
4.5.6 Implementazione 78
4.5.7 Pubblicazione 87

5 Conclusioni **89**
5.1 io@spot nel futuro 90

Elenco delle figure **91**

Bibliografia **95**

Ringraziamenti **97**

Glossario

W3C: acronimo di *World Wide Web Consortium*, una comunità internazionale che progetta e sviluppa standard per garantire una solida creascita del Web nel lungo termine.

IEFT: acronimo di *Internet Engineering Task Force*, un organismo internazionale composto da specialisti e ricercatori interessati all'evoluzione tecnica e tecnologica di Internet. Si occupa di sviluppare e promuovere standard Internet e lavora in sretta cooperazione con il W3C.

Architettura software: è l'astrazione degli elementi di un sistema software durante una qualche fase delle sue operazioni. Un sistema potrebbe essere composto da molti livelli di astrazione e molte pasi di operazioni, ciascuna con la propria architettura software. È definita da una configurazione di elementi architetturali - componenti, connettori e dati - tra loro vincolati in maniera da garantire all'architettura un determinato insieme di proprietà.

Stile architetturale: è un insieme coordinato di vincoli che definiscono i ruoli e le caratteristiche degli elementi architetturali; determina anche le modalità di interazione di tali elementi architetturali con una qualsivoglia architettura conforme agli stessi vincoli.

Fiat standard: non è propriamente uno standard, ma un comportamento; è la descrizione di una determinata maniera di fare certe cose. Non rappresenta specifiche riconosciute e condivise anche da altre entità, bensì è spesso legato ad uno specifico contesto applicativo.

JSON-LD: letteralmente *JSON for Linking Data* è un formato per la serializzazione e lo scambio di dati che integra sintassi standard per la descrizione di semantiche applicative, soprattutto in termini di collegamenti tra documenti e risorse.

ALPS: è un formato per la definizione di semplici descrizioni delle semantiche applicative. Un documento in formato ALPS può essere usato come *profilo* per spiegare le semantiche applicative di un dato documento senza dover per forza essere a conoscenza del relativo *media type*. Favorisce la riusabilità di profili tra media-type diversi nell'ottica di definire profili standard per specifici domini applicativi.

DNS: è l'acronimo di *Domain Name System* e rappresenta un sistema software per la risoluzione dei nomi dei nodi della rete nei corrispondenti indirizzi IP e viceversa.

Sistema embedded: termine utilizzato per identificare i sistemi elettronici di elaborazione basati su un microprocessore *special purpose* con piattaforma hardware *ad hoc*, cioè progettato in maniera specifica per una determinata applicazione. Il livello di complessità può variare parecchio se si considera che vengono utilizzati negli scenari più disparati, da semplici dispositivi portatili a grandi in-

stallazioni stazionarie. Questa tipologia di sistema offre minimizzazione dei costi e delle dimensioni, ottimizzazione sia dell'hardware che del software e di conseguenza un'elevata affidabilità.

RFID: acronimo di *Radio-Frequency IDentification*, rappresenta una tecnologia per l'identificazione o la memorizzazione dati automatica di oggetti, animali o persone tramite l'utilizzo di particolari etichette elettroniche (*tag*).

Stack applicativo: indica una piattaforma software per lo sviluppo composta da applicazioni eterogenee, ognuna con caratteristiche e funzionalità diverse, che devono interopare tra loro.

DBMS: nell'informatica un Database Management System (abbr. DBMS) è un sistema software finalizzato a consentire la creazione, la manipolazione e l'interrogazione efficiente di database.

Capitolo 1

Introduzione

Il World Wide Web sta assumendo un ruolo sempre più centrale nella quotidianità degli individui e nelle dinamiche di business delle organizzazioni: il modello di comunicazione sta velocemente cambiando verso scenari distribuiti, dove applicazioni eterogenee riescono a comunicare tra loro condividendo informazioni di ogni genere attraverso la rete internet. Una diretta conseguenza di tale processo evolutivo è la necessità di sviluppare architetture software che garantiscano l'interoperabilità tra sistemi differenti: snegli anni sono stati definiti svariati modelli architetturali, ciascuno con i propri vantaggi e svantaggi, ma con l'obiettivo comune di migliorare la comunicazione. Il più conosciuto e maggiormente utilizzato è quello SOA (Service-Oriented Architecture) che trova una concreta implementazione nella tecnologia dei Web Services. Il grande problema di questo modello è che risulta essere troppo rigoroso e non consente agli applicativi di essere sufficientemente flessibili in risposta a qualsiasi forma di cambiamento; rendono dunque SOA incompatibile con una delle principali proprietà del Web.

L'elaborato si pone come obiettivo quello di presentare un più recente modello architetturale per lo sviluppo di sistemi software basato sugli stessi vincoli e standard del World Wide Web, senza aggiungervi complessità come accade inve-

ce con il modello SOA. A tal proposito, il lavoro svolto si articola in tre macro capitoli:

- il primo è incentrato sul World Wide Web. Vengono proposti rapidi cenni storici sulla sua nascita e sulle sue principali proprietà, per poi ripercorrere la derivazione dello stile architetturale ReST - effettuata da R. T. Fielding - al fine di guidarne i moderni sviluppi e porvi precisi standard tecnologici a supporto;

- il secondo presenta nel dettaglio la tecnologia delle ReSTful web API, basate su modello architetturale ROA (Resource-Oriented architecture) e fedeli ai vincoli dello stile architetturale del Web per garantire semplicità, flessibilità ed estensibilità ai sistemi software che le implementano. Se ne propone una completa metodologia di progettazione da contrapporre ai più comuni processi implementativi che non tengono in considerazioni seri aspetti architetturali e compromettono i vantaggi derivanti dall'adozione dello stile architetturale ReST;

- il terzo capitolo tratta parte di un progetto sviluppato in ambito aziendale e lo adotta come caso di studio: si è infatti applicata la metodologia proposta nel capitolo precedente allo sviluppo delle web API di *io@spot* (un sistema distribuito per il controllo degli accessi e per la gestione delle presenze in contesto aziendale) in maniera da avere riscontri pratici sulle nozioni teoriche affrontate.

Capitolo 2

Il World Wide Web

In questo capitolo viene proposta una rapida descrizione iniziale del World Wide Web, il sistema distribuito più utilizzato al mondo che, negli ultimi tempi, sta subendo una crescita eccezionale sia in termini di utenza che di contenuti e servizi offerti.
Si approfondisce poi l'argomento riprendendo il lavoro svolto dall'informatico Roy Thomas Fielding e proponendone una sintesi sufficientemente dettagliata per enunciare le caratteristiche dello stile architetturale del Web; al termine del capitolo si avrà una chiara comprensione di cosa sia ReST e di come abbia influito sugli sviluppi del WWW e delle tecnologie che oramai ne rappresentano lo standard.
Lo scopo è proprio quello di porre un insieme di basi teoriche per gli argomenti che verranno invece approfonditi nei prossimi capitoli e che rappresentano unità fondamentali di questo elaborato.

2.1 Overview

Il World Wide Web è un grande sistema distribuito che poggia sull'infrastruttura Internet: è nato tra gli anni '80 e '90 tra le mura del CERN di Ginevra in

seguito alla pressante necessità di gestire e condividere le informazioni scientifiche raccolte ed elaborate negli studi e nei centri di ricerca di tutto il mondo. Lo scambio di dati al tempo era già possibile in maniera elettronica - basti pensare alle email - ma l'assenza di standard e strumenti specifici per quel determinato scopo continuava a rappresentare un grosso ostacolo, soprattutto in termini di efficienza.

Fu così che un giovane ambizioso, Tim Berners-Lee, paventò un modello ben strutturato riassumibile in un sistema distribuito avente le seguenti caratteristiche:

- accessibilità tramite la rete per rendere l'informazione raggiungibile da qualunque parte del mondo avente accesso ad Internet, risolvendo il problema delle distanze;
- accessibilità da sistemi eterogenei, poiché la diversità dei terminali utilizzati e i relativi formati supportati rappresentavano un vero e proprio muro (tenendo anche conto del fatto che al tempo gli standard latitavano);
- fornitura di accesso a dati e documenti esistenti, anche in fase di lavori in corso;
- consentire la definizione di aree private per le quali l'accesso fosse limitato.

La definizione più comune con cui è stato - ed è tuttora - definito il modello proposto è quella di *sistema distribuito ipertestuale* per il semplice fatto che ogni documento di testo porta con sé riferimenti diretti ad altre fonti di informazioni correlate. Per avere un esempio concreto basta pensare ad una comune pagina Web: i cosiddetti *link* che propone e con cui l'utente è solito interagire sono una delle forme più ricorrenti di *collegamenti ipertestuali* che indirizzano la navigazione ad altre pagine interconnesse con quella attualmente visualizzata.

Per affermarne la validità, Tim Barners-Lee ne fornì anche una prima implementazione, dando vita ad una embrionale versione di quello che oggi conosciamo

come il World Wide Web: un browser che, per quanto primitivo rispetto a quelli odierni cui siamo abituati, permetteva tramite un'apposita interfaccia visuale di consultare, creare e modificare documenti collegati l'uno con l'altro tramite una fitta rete ipertestuale.
Conseguentemente, visto l'enorme potenziale, il mondo dell'IT ha iniziato ad investire sul progetto sviluppandone le funzionalità e indagandone le criticità fino ad ottenere quanto oggi conosciamo ed utilizziamo: un sistema software che permette di accedere globalmente a qualunque tipo di informazione semplicemente disponendo di una connessione ad Internet e di un computer; uno spazio condiviso di informazioni che macchine e persone possono scambiarsi, consultare ed elaborare.

2.2 Caratteristiche generali

Negli ultimi vent'anni si sono susseguiti innumerevoli eventi di spiccata importantza che hanno segnato la crescita e l'innegabile successo del World Wide Web. Rilevante è l'eccelso lavoro fatto da Roy Thomas Fielding: informatico statunitense, attualmente annoverabile tra i più esperti al mondo per quanto concerne architetture di reti,il quale ha dedicato anni ad identificare i problemi del Web con il chiaro intento di porvi rimedio tramite la definizione di uno stile architetturale ottimale verso cui farne tendere tutti i successivi sviluppi.
Prima di procedere nell'analisi dell'operato di Fielding, è necessario specificare quanto il Web non sia semplicemente un sistema distribuito, bensì un sistema network-based: il primo citato è riconducibile ad un classico sistema centralizzato che viene eseguito su più CPU indipententi le une dalle altre; il secondo invece è un sistema distribuito in grado di svolgere operazioni tramite la rete Internet

(riferimento bibliografico a tanenbaum per la definizione?). Nello specifico, un sistema network-based prevede che:

- la comunicazione tra gli svariati attori coinvolti avvenga tramite scambio di messaggi;
- le operazioni siano richieste ed eseguite tramite la rete in una maniera non per forza trasparente all'utente come invece avviene nella gran parte dei sistemi distribuiti centralizzati (soprattutto in quelli antecedenti al WWW);
- le svariate applicazioni che interagiscono con il Web espongano funzionalità senza la necessità di conoscere il core-business del contesto in cui sono chiamate ad operare.

È sulla base di queste considerazioni che Fielding stesso ha iniziato a derivare lo stile architetturale del World Wide Web, partendo da un'accurata analisi delle principali caratteristiche utilizzate per la classificazione delle architetture dei sistemi network-based:

- performance: rappresentano un problema tutt'ora aperto, nonostante si stiano facendo enormi passi avanti dal punto di vista sia infrastrutturale sia applicativo. La latenza di un sistema network-based è banalmente dovuto al continuo trasferimento in rete di ingenti quantitativi di dati che, dal punto di vista dell'utente finale, significa attesa;

- semplicità: chiunque interagisca con il Web, sia esso un lettore, un autore o perfino uno sviluppatore, riesce ad approcciarvisi senza difficoltà alcuna. Aprendo un browser ci si trova di fronte a contenuti facilmente interpretabili non solo da altri calcolatori, ma in primo luogo dalle persone. Molto sistemi ipertestuali precedenti al WWW hanno fallito, proprio perché non hanno saputo offrire all'utenza un'interfaccia generica, flessibile ed immediata;

- modificabilità: da intendersi come la predisposizione del sistema ad essere configurato, esteso con nuove funzionalità e riusato in altri contesti. Un sistema privo di qualsivoglia grado di estensibilità può essere pubblicato una sola volta: ne consegue una scarsa longevità, poiché l'utenza migrerà ad altri sistemi nel momento in cui vedrà cambiate le proprie esigenze. A conferma di quanto detto, il Web è sopravvissuto in maniera egregia per oltre vent'anni, adattandosi ad ogni tipologia di cambiamento avvenuta;

- scalabilità: è sicuramente uno degli aspetti di maggior spicco per un'architettura come quella del World Wide Web, poiché nulla è sotto il controllo di un'unica entità centrale. Il sistema è una rete informativa che mette in comunicazione un indefinito numero di sotto-sistemi - ben delineati ciascuno dai propri confini organizzativi - che hanno pieno controllo delle proprie facoltà e che possono decidere se e come collaborare con gli altri, senza essere tra loro vincolati da relazioni a lungo termine;

- pubblicazione indipendente: in conseguenza alla scalabilità dell'intero sistema, le parti che lo compongono posso cambiare con ritmi e modi differenti mantenenendosi comunque compatibili con vecchie implementazioni;

- sicurezza: come è già stato detto, il Web mette in comunicazione sotto-sistemi differenti, ciascuno con le proprie regole e politiche interne; questo implica la necessità di poter applicare logiche di riconoscimento/autenticazione sul limitare dei reciproci confini organizzativi. Questa caratteristica porta con sé principalmente due svantaggi: prima di tutto si è costretti ad aumentare il quantitativo di informazioni scambiate in una generica interazione sicura per includere anche dati di autenticazione; inoltre viene limitato il numero di funzionalità che un sistema può esporre apertamente senza imporre vincoli di riconoscimento.

2.3 Lo stile architetturale ReST del Web

Come da definizione, per meglio delineare uno stile architetturale, è importante descrivere tutti quei vincoli che definiscono e delimitano i ruoli e le caratteristiche degli elementi architetturali, soprattutto in termini di relazione tra gli stessi e una qualunque altre architettura conforme a tale stile.
Conscio della mancanza di razionalità architetturale tra i principi della primissima versione del Web, Fielding parte analizzando i più comuni stili di sistemi network-based (riferimento a capitolo tre di tesi di fielding) e deriva uno stile architetturale applicabile ad un sistema ipermediale network-based e adatto a guidare gli sviluppi della moderna architettura del World Wide Web.
Il processo di derivazione si articola prevalentemente in due fasi: la prima prevede che vengano definiti dei vincoli architetturali che, se applicati con criterio, garantiscano scalabilità delle interazioni, indipendenza del deploy delle singole parti del sistema e presenza di entità in grado di mediare le comunicazioni al fine di migliorare efficienza, sicurezza e retrocompatibilità; la seconda fase invece si sviluppa in un'attenta analisi degli elementi architetturali che sono coinvolti nelle interazioni del Web e che devono rispettare i vincoli sopracitati.

2.3.1 I vincoli architetturali

≪REST's client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries proxies, gateways, and firewalls to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communi-

cation translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.≫[5]

Partendo da uno stile senza restrizione alcuna, Fielding prende in esame i vincoli architetturali degli stili più comuni tra i sistemi network-based e li combina in maniera da soddisfare quelle che ritiene essere le necessità di un sistema distribuito ipermediale come World Wide Web:

- client-server: l'interazione tra i componenti del WWW è di natura client-server. I client e i server devono essere separati da interfacce uniformi così che ne possano essere delineati - e separati - i ruoli: i primi non devono essere coinvoliti in attività quali il salvataggio e la persistenza delle informazioni, così da poter avere implementazioni più facilmente portabili tra piattaforme differenti; i secondi invece sono esonerati da questioni come l'interfaccia

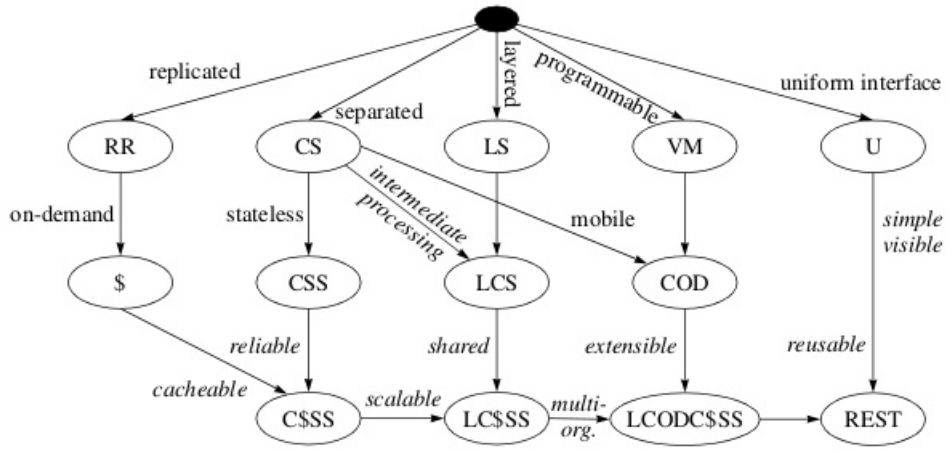

Figura 2.1: Derivazione dello stile ReST dai vincoli

grafica e lo stato dell'utente in modo da risultare semplici e scalabili. Un aspetto altrettanto importante per il contesto del Web è che questo vincolo promuove l'evoluzione indipendente dei componenti del sistema;

- assenza di stato: la comunicazione tra client e server dev'essere tale che nessun contesto client venga storicizzato sul server per far fronte a successive richieste. Il client deve dunque includere di volta in volta tutte le informazioni necessarie al server per capire ed elaborare le richieste, comprese quelle riguardanti un eventuale stato di sessione. I vantaggi di una comunicazione stateless sono una migliore trasparenza delle interazioni, una migliore affidabilità del sistema, una maggiore semplicità soprattutto in termini di implementazione dell'infrastruttura server, mentre gli svantaggi che ne conseguono sono un ridotto controllo del server sui comportamenti applicativi che vengono messi in atto dai client ed un incremento della quantità di dati da trasferire per ogni richiesta, che inficia sulle performance del sistema;

- cache: tutti i dati che vengono inviati dal server all'interno di una risposta devono poter essere, in un modo o nell'altro, contrassegnati come cacheable. Questo consente ad un client di sapere se una data risposta possa essere ritenuta valida anche per successive richieste dello stesso tipo, in maniera da bypassare il server e diminuire la latenza generale del sistema. Un contro di questo vincolo architetturale riguarda l'affidabilità del sistema in termini di validità dei dati: va dunque trattato con estrema attenzione in maniera da evitare spiacevoli scenari di inconsistenza tra i dati in cache e quelli che sarebbero invece ritornati dal server a fronte di una nuova richiesta;

- interfaccia uniforme: le entità coinvolte devono avere un'interfaccia di comunicazione ben definita, in modo da semplificare e disaccoppiare tutte quante le parti architetturali le une dalle altre lasciandole libere di evolversi in ma-

niera autonoma e mantenere un più elevato livello di scalabilità. L'interfaccia del Web è progettata per essere efficiente nel trasferimento di contenuti ipermediali mediamente grossi: di conseguenza tutte le forme di interazione differenti da quella prevista per il Web risultano non ottimizzate;

- sistema stratificato: l'architettura del sistema dovrà impedire ad un componente di vedere al di là dello strato con cui interagisce direttamente. Un client quindi non dovrà mai sapere se sta comunicando con il server di livello più basso oppure con un intermediario (e.g. server che effettuano load-balancing o mettono a disposizione servizi di cache distribuita), questo a favore di una maggiore scalabilità e flessibilità del sistema anche se si rischia di abbassare il livello di performance introducendo overhead per ogni strato che viene aggiunto;

- code-on-demand: opzionalmente, il sistema può offrire la possibilità ai singoli client di chiedere funzionalità a determinati server a run-time (tipicamente sotto forma di script). Il code-on-demand alleggerisce e semplifica i client, aumentando allo stesso tempo il grado di estensibilità del sistema.

2.3.2 Gli elementi architetturali

Quando si parla di elementi architetturali di un sistema si intendono i componenti, i connettori e gli elementi informativi che ne definiscono le basi dell'architettura da un punto di vista dei ruoli dei componenti, dei vincoli di interazione con gli altri componenti tramite i connettori e dell'interpretazione dei dati scambiati, ignorando completamente i dettagli implementativi.
Ciascuno di essi verrà di seguito approfondito in relazione ad uno stile architetturale su misura per il World Wide Web e, dunque, ai vincoli precedentemente descritti.

Nel caso in questione è buona norma partire dagli elementi informativi, poiché è semplice intuire quanto siano fondamentali per la definizione dello stile architetturale di un sistema distribuito ipermediale; vengono considerati tali tutti quanti i dati trasferiti o ricevuti da un componente attraverso un connettore e che contengono un qualsivoglia tipo di informazione, nello specifico:

- le risorse: rappresentano il massimo livello di astrazione dell'informazione che viene condivisa sul World Wide Web. Qualunque tipologia di informazione che possa essere contrassegnata da un nome univoco e di cui valga la pena tenere traccia è una risorsa a tutti gli effetti. Una risorsa è da interpretare come un mapping concettuale ad un certo insieme di entità (siano esse documenti, immagini, servizi...), non come l'entità che corrisponde al suddetto mapping in un determinato momento. Ne consegue il forte vincolo per cui le semantiche di mapping di una risorsa rimangano invariate nel tempo, in maniera che le entità che la referenziano possano cambiare senza problemi. Un'astrazione così spinta della definizione di risorsa implica caratteristiche dell'architettura Web piuttosto importanti (riferimento a Fielding pag 89):
  - generalità nella gestione delle risorse proprio per il fatto che ad una stessa risorsa possono corrispondere più fonti di informazioni senza che debbano essere distinte per tipologia o implementazione;
  - possibilità di referenziare una risorsa in qualità di concetto astratto senza la necessità che questo sia già collegato ad una qualche rappresentazione concreta;
  - flessibilità: in seguito a modifiche delle singole rappresentazioni non c'è bisogno di rimodellare i link che referenziano la data risorsa.

- gli identificatori di risorse: ogni risorsa deve avere un proprio identificatore in forma di URI[3] (DA SISTEMARE IN .BIB) e tale identificatore deve essere univoco. Rappresenta una tecnologia fondamentale per il Web: prima dell'HTML c'erano sistemi ipertestuali, prima dell'HTTP si sono susseguiti svariati protocolli di rete, ma nessuno di questi si è mai parlato direttamente poiché gli URI hanno consentito a tutte queste tecnologie di interconnettersi nella rete del WWW senza la necessità di conoscersi reciprocamente. Il Web ha dunque una capacità che a tutti gli altri stili architetturali manca: identificare ogni singola informazione in maniera univoca e precisa. L'unico aspetto a cui è fondamentale prestare attenzione è che lo stile architetturale ReST lascia agli autori la facoltà di scegliere l'identificatore più adatto alla natura del concetto mappato dalla data risorsa e non sempre si investono il tempo e i soldi necessari per una progettazione ottimale degli URI;

- rappresentazioni delle risorse: sono sequenze di byte che forniscoo una possibile rappresentazione della risorsa cui sono associati. Una rappresentazione può avere la forma di un documento, di un file o di un qualsivoglia messaggio HTTP, l'importante è che contenga tutti i dati utili e necessari a definire lo stato della risorsa.

- metadati di una rappresentazione: solitamente espressi come un insieme di coppie nome-valore in cui il nome corrisponde ad uno standard che definisce la struttura e la semantica del valore associato. Servono sostanzialmente ai client e ai server per interpretare correttamente i messaggi ricevuti: un esempio di metadato è il media-type, il quale fornisce informazioni sul formato dei dati che descrivono la rappresentazione;

- metadati di risorsa: fornisce dati aggiunti relativamente alla risorsa che non sono specifici e necessari per la rappresentazione trattata come, ad esempio,

informazioni riguardati altre risorse correlate;

- dati di controllo: vengono inclusi nelle richieste e nelle risposte in modo che i destinatari possano interpretare correttamente lo scopo del messaggio. Genericamente sono utilizzati per indicare lo stato corrente o lo stato desiderato della risorsa richiesta, piuttosto che non la rappresentazione di una qualche condizione d'errore nella risposta.

Un connettore invece è l'astrazione di un qualunque meccanismo che medi la comunicazione tra i componenti, coordinandoli. Quelli previsti dallo stile architetturale ReST sono:

- client: connettore primario tramite il quale viene avviata una comunicazione in seguito all'invio di una richiesta;
- server: anch'esso connettore primario poiché rimane in ascolto di richieste processandole al momento opportuno, così da rendere accessibili i servizi che è in grado di offrire;
- resolver: a differenza dei primi due è un connettore secondario; ciò nonostante offre un servizio non da poco in quanto traduce identificatori parziali o completi di una risorsa in quello che è l'indirizzo di cui necessitano i componenti per comunicare e scambiarsi dati. Un esempio lampante di connettore resolver è il DNS;
- tunnel: questo connettore rende possibile la comunicazione attraverso un confine di connessione (ad esempio tra un firewall e un gateway di rete più a basso livello). Tra i tanti esempi si annovera il canale SSL, utilizzato infatti da filtro per le richieste prive di autenticazione;
- cache: questo connettore differisce da quelli precedentemente elencati perché localizzato a ridosso dell'interfaccia di un connettore primario (server o

client indistintamente) in maniera da rendere riusabili risposte a richieste ricorrenti in un lasso di tempo più o meno breve.

Ogni connettore dell'architettura ReST si espone al resto del sistema con un'interfaccia assai generica in modo da rendere il più standard possibile le interazioni per l'accesso e la manipolazione delle risorse e per il trasferimento delle rispettive rappresentazioni: ciò contribuisce ad aumentare il grado di componibilità del sistema e a rendere i connettori facilmente sostituibili senza inficiare sul funzionamento dell'intero sistema.

Un componente è un'unità astratta di istruzioni software dotata di un proprio stato interno ed in grado di interpretare e manipolare dati tramite la propria interfaccia. Per quel che concerne lo stile ReST del Web si ritrovano i seguenti componenti:

- Server d'origine: tramite un apposito connettore provvede alla gestione delle risorse e ne fornisce le rappresentazioni entro i limiti delle sue competenze. Deve essere l'ultima tappa di una comunicazione atta a modificare lo stato di una risorsa;

- User Agent: sfrutta un connettore di tipo client per dare inizio ad una comunicazione con l'invio di una richiesta ed attenderne, in qualità di ultimo destinatario, la risposta;

- Proxy e gateway: sono componenti intermedi che provvedono ad implementare il vincolo architetturale di stratificazione del sistema. La particolarità di questi componenti è che si comportano sia da server che da client a seconda delle situazioni: hanno la facoltà di ridirezionare - ed eventualmente trasformare - le richieste e le risposte che vengono scambiate tra il server d'origine e lo user agent. Un client ha piena facoltà di decidere se sfruttare

un proxy o meno, mentre il gateway viene solitamente imposto o dalla struttura di rete sulla quale avviene la comunicazione oppure dal server d'origine stesso.

2.3.3 ReST: Representational State Transfer

Lo stile architetturale del World Wide Web è definito con l'acronimo ReST, che letteralmente significa ***Representational State Transfer*** proprio perché quello che viene scambiato dai componenti altro non è che una rappresentazione dello stato attuale della data risorsa; tale nome rievoca chiaramente l'immagine di come si comportano le applicazioni Web ben progettate: una fitta rete di pagine ipertestuali in cui l'utente si districa selezionando, a partire da una pagina iniziale (stato iniziale dell'applicazione), un link (transizione di stato dell'applicazione) che lo porta ad una seconda pagina Web (stato dell'applicazione variato). Il Representational State Transfer è più semplicemente un'astrazione degli elementi architetturali all'interno di un sistema ipermediale distribuito: ReST ignora i dettagli delle implementazioni dei componenti e delle sintassi dei protocolli utilizzati per la comunicazione, si concentra sui ruoli dei componenti, sui vincoli nelle interazioni con gli altri componenti e sull'interpretazione che essi danno ai dati scambiati.

ReST è uno stile ibrido derivato da diversi stili architetturali network-based e combinato con vincoli ulteriori per la definizione di un'interfaccia comune dei connettori: l'obiettivo pilota è quello di tracciare una strada per i futuri sviluppi dell'architettura Web e degli standard che definiscono i comportamenti dei singoli componenti, in maniera da colmare le lacune della prima versione del Web mantenendo però continuità e retrocompatibilità; tuttavia, come affermato da R. T. Fielding stesso, ReST non è stato studiato solamente con l'intenzione di migliorare il design dell'architettura del Web, bensì anche per riuscire a scovarne gli

effettivi problemi - dovuti principalmente ad ignoranza o disattenzioni - prima che vengano standardizzati da un eccessiva diffusione.

Può essere descrito come un set coordinato di vincoli architetturali che, se applicati nel loro insieme, minimizzano la latenza e la quantità delle comunicazioni sulla rete e allo stesso tempo massimizzano l'indipendenza e la scalabilità delle implementazioni dei componenti: tutto questo lo si ottiene applicando i suddetti vincoli alle semantiche dei connettori, quando invece tutti gli altri stili network-based si sono sempre e solo concentrati sulle semantiche dei componenti. ReST abilita il caching ed il riuso delle precedenti interazioni, garantisce sostituibilità ed estensibilità dinamica dei componenti e permette l'intervento di intermediari nei flussi di comunicazione, mantenendosi sempre in linea con le necessità di un sistema ipermediale distribuito su Internet.

Nello specifico, ReST non restringe la comunicazione ad un protocollo particolare, non forza nemmeno le implementazioni delle risorse secondo modelli predefiniti, si limita a proporre un'interfaccia d'accesso generica ai componenti guidandone poi le modalità di interazione e di implementazione. Se una applicazione necessita di caratteristiche proprie di un'altra architettura e che non è offerta dal Web può implementarle a parte, su un sistema parallelo alla stregua di telnet o mailto. È palese come nella realtà si tenda ad approfittare di questo elevato grado di libertà: in tutte le architetture software ci sono elementi che non si attengono agli standard e ai vincoli dettati dallo stile scelto; a maggior ragione nel Web non tutti i componenti si comportano secondo i principi architetturali definiti dallo stile ReST.

È una dimostrazione di come i principi dell'ingegneria del software possano essere semanticamente applicati alle fasi di valutazione e design di un sistema software reale: se ne può trarre esempio concreto soprattutto da quella che è stata la progettazione degli standard tecnologici che sono stati adottati per il Web e che

vengono di seguito trattati.

2.3.4 ReST applicato ai protocolli del Web

Essendo un sistema distribuito network-based ed essendo strettamente legato ad Internet, gli standard e le implementazioni di riferimento che vennero adottati nel tempo dal W3C per il Web furono gli stessi imposti dal IETF per la rete Internet:

- HTML (Hypertext Markup Language) per la visualizzazione dei dati;
- URI (Uniform Resource Identifiers) per l'indirizzamento delle risorse;
- HTTP (Hypertext Transfer Protocollo) per il trasferimento dei contenuti ipermediali da un componente all'altro

Lo stile architetturale derivato da Fielding per gli sviluppi del World Wide Web venne preso come riferimento anche per l'evoluzione e la formalizzazione degli stessi standard, processi in cui l'informatico statunitense venne direttamente coinvolto.

HTML

Molti sono i formati delle rappresentazioni che possono essere scambiati sul Web in una qualunque interazione client-server e altrettante le tecnologie a disposizione per interpretarli. Per via della sua popolarità, l'HTML è diventato lo standard per rappresentare risorse che debbano essere comprensibili ad un utente umano; in ogni caso è interpretabile anche dai calcolatori in quanto impone che il documento risultante abbia una struttura ben definita.
Gli elementi alla base dell'HTML sono i tag, innestati gli uni dentro gli altri per

definire strutture di contenuti gerarchicamente organizzati, ma ciò che lo avvantaggia rispetto a tecnologie quali XML o JSON è disporre di un insieme predefinito di controlli ipermediali che si sposano alla perfezione con le caratteristiche del World Wide Web [13, p. 193-196]:

- i tag `<link>` e `<a>` rappresentano un collegamento diretto ad una risorsa, in quanto portano il client a fare una richiesta GET all'URL specificato dall'attributo `href` e portano generalmente ad un cambio di stato dell'applicazione (e.g. cambio di vista corrente);

- i tag `<img>` e `<script>` sono sempre dei collegamenti diretti ad una specifica risorsa, ma non causano un cambiamento di stato poiché includono la rappresentazione ottenuta dal server nella vista corrente (il primo in qualità di immagine, il secondo in qualità di codice da eseguire);

- il tag `<form>` viene ritornato dal server provvisto di un URL e di alcuni campi di input. Ha un duplice comportamento che viene definito dal valore dell'attributo `method`: nel caso in cui sia valorizzato con la stringa `POST` utilizza il contenuto dei campi di input per creare il body di una richiesta POST con media type `application/x-www-form-urlencoded`; nel caso in cui invece valga `GET` i campi di input vengono automaticamente riempiti con la rappresentazione della risorsa ritornata dalla chiamata GET specificata, sempre con media type `application/x-www-form-urlencoded`.

Nonostante i controlli ipermediali siano un punto a favore per l'HTML, c'è un aspetto che vale la pena trattare e che limita l'effifacia di questa tecnologia in un contesto di rispetto dei vincoli dello stile architetturale ReST: i tag appena descritti, insieme a tutti gli altri non citati, definiscono una semantica applicativa molto semplice e standardizzata al caso di documenti leggibili da umani; non sono ancora offerte sufficienti alternative per definire semantiche differenti da quelle

previste dall'HTML e questo inficia su importanti proprietà quali flessibilità e estensibilità.
Nuovi sviluppi dello standard HTML [15] stanno pian piano ponendo rimedio a questo limite e alcuni dettagli sono approfonditi nel Capitolo 4.

URI

L'URI è l'elemento più semplice di tutta quanta l'architettura del Web, ma al contempo anche il più importante. La sintassi di riferimento di questo standard non è mai cambiata in maniera significativa, tuttavia si sono radicalmente evolute nel tempo le semantiche di ciò che si intende per risorsa: come diretta conseguenza, sono stati adottati i principi dello stile ReST nella definizione del concetto di risorsa in relazione allo standard URI (riferimento ad rfc URI) e di tutte le semantiche per una generica interfaccia di accesso a manipolazione delle risorse attraverso le relative rappresentazioni.
Per come viene concepita la risorsa dallo stile architetturale ReST, gli identificatori che la referenziano dovrebbero cambiare il meno possibile. Gli stessi autori del Web hanno bisogno di lavorare con identificatori che incarnino le semantiche di riferenziazione ipermediale, che rimangano dunque statici nel tempo nonostante il risultato di un eventuale accesso possa di man in mano variare. ReST pone rimedio a questo requisito semplicemente definendo la risorsa stessa come semantica di ciò che si vuole identificare, piuttosto che non le semantiche del valore che viene associato a tale referenza al momento della sua creazione.
Bisogna entrare nell'ottica per cui una risorsa non è qualcosa salvato fisicamente su disco, non è nemmeno un qualsivoglia meccanismo del server con cui si comunica, bensì un semplice mapping concettuale che viene risolto in una rappresentazione concreta dal server d'origine tramite politiche trasparenti al client che la richiede. Tutto ciò deriva dal vincolo archietturale di mantenere un'inter-

faccia di comunicazione generica, così da potersi astrarre in fase di progettazione dalle singole implementazioni che ogni dominio può dare alle proprie risorse.

```
<scheme>:<scheme-specific-part>
```

Concretamente un identificatore URI altro non è che una stringa composta da:

- uno schema, che indica la metodologia per individuare la risorsa univocamente o, meglio, il protocollo da utilizzare per riuscire ad interpretare in maniera corretta la seconda parte dell'identificatore. I più comuni sono *http*, *ftp*, *mailto* e *file*, ma ne esistono tanti altri tutti che sono ufficialmente annoverati da IANA (Internet Assigned Numbers Authority);
- una parte specifica dello schema che, in formati diversi a seconda del protocollo indicato dallo schema, esplicita parametri utili all'identificazione della risorsa sul server.

Il Web ha quindi una caratteristica che a tutti gli altri sistemi distribuiti network-based manca: la possibilità di referenziare singole informazioni in maniera precisa ed univoca. Tuttavia nella pratica capita spesso di non sfruttare questo vantaggio poiché gli URI vengono progettati senza tenere conto della nozione architetturale di identificatore, né tantomeno della definizione di risorsa come concetto astratto, ma considerando solamente aspetti sintattici: il tutto va a discapito della scalabilità del server e della validità nel tempo degli URI stessi.

HTTP

L'HTTP invece è stato soggetto a diversi cambiamenti e revisioni nel corso degli ultimi vent'anni, essendo il più importante protocollo applicativo di comunicazione tra i componenti del Web e l'unico ad essere stato progettato in maniera

specifica per il trasferimento di rappresentazioni di risorse. L'applicazione dei principi dello stile ReST alla prima versione dell'HTTP ha messo a nudo diversi punti critici su cui poi il W3C e l'IET hanno lavorato duramente:

- la pubblicazione di una nuova versione del protocollo avrebbe comportato problematiche di retrocompatibilità da non sottovalutare;
- si sarebbe dovuta separare l'interpretazione dei messaggi dalle semantiche dell'HTTP e del sottostante layer di trasporto (TCP), in maniera da rendere lo standard il più indipendente possibile dall'effettiva implementazion e maggiormente estensibile;
- le funzionalità di caching dei dati sarebbero dovute essere governate da politiche più precise e attente alle performances e alla consistenza dei dati;
- la semantica dei messaggi non era sufficientemente auto-descrittiva e questo creava prolemi soprattutto in termini di performance piuttosto che di scalabilità.

HTTP è in realtà una famiglia di protocolli, ciascuno avente numeri di major e minor version. Ciascun messaggio scambiato tra due endpoint differenti ha un parametro che indica la versione del protocollo HTTP utilizzata, in maniera da fornire informazioni sul mittente al destinatario sulla base delle quali applicare o meno politiche di retrocompatibilità: consente una sorta di protocol-negotiation [5, p. 118]. Quando un nodo della catena di comunicazione sfrutta differenti implementazioni dello standard HTTP, non serve che tutti gli altri nodi siano compatibili con quella determinata versione, ma basta che lo siano i due più prossimi che trasformeranno poi il messaggio per tutti gli altri così da renderglielo interpretabile. Questo consente di attuare processi di aggiornamento parziali, in modo da estendere gradualmente l'intero sistema senza compromettere la retrocompatibilità con le precedenti versioni.

Lo stile architetturale elaborato da Fielding prevede che una risorsa possa essere un mapping ad un qualsiasi concetto, che possa essere referenziata da più identificatori URI allo stesso tempo e che possa essere trasferita sotto forma dei più svariati formati, ma questo grado di libertà non deve in alcun modo portare un client ad interagire con una data risorsa senza preoccuparsi di rispettare o meno delle regole. Proprio per questo motivo è stata decisiva l'imposizione di semantiche ben precise su quelli che sono i *metodi* - o tipi di messaggi - del protocollo HTTP:

- GET: ritorna la rappresentazione della risorsa identificata dall'URI. Questo metodo è definito *sicuro* perché non ha alcun effetto sullo stato della risorsa;
- DELETE: distrugge la risorsa e non è sicuramente considerabile un metodo sicuro, tuttavia è *idempotente* ovvero, se ripetuta più volte, la chiamata avrà lo stesso effetto sullo stato della risorsa come se fosse effettuata una volta sola;
- POST: crea una nuova risorsa sulla base della rappresentazione ricevuta tramite il corpo della richiesta. Non è un metodo né sicuro né idempotente, poichè eseguendo la stessa chiamata ripetutamente si otterrebbe come risultato la creazione di una nuova risorsa per ogni richiesta inviata;
- PUT: modifica lo stato della risorsa con le informazioni della rappresentazione ricevuta e si dimostra idempotente alla stregua del metodo DELETE;
- PATCH: apporta modifiche a parti dello stato corrente della risorsa: se la rappresentazione ricevuta non contempla determinati aspetti dello stato corrente, questi vengono lasciati invariati. Questo metodo è stato aggiunto successivamente alla definizione dello standard HTTP [14], ma è da considerarsi con una certa importanza poiché permette cambiamenti puntuali allo stato della risorsa;

- HEADER: specifica gli header che verrebbero ritornati in risposta ad una chiamata di tipo GET insieme alla rappresentazione della risorsa. È spesso utilizzato per risparmiare banda, essendo decisamente minore il numero di informazione da ritornare al client;
- OPTIONS: permette di sapere quali sono i metodi HTTP a cui la risorsa specificata risponde.

Bisogna però fare un'attenta distinzione fra quelle che sono le semantiche di protocollo appena elencate e quelle applicative: le prime servono a capire in maniera più o meno approssimativa ciò che il client desidera, semplicemente guardando al tipo di richiesta inviata; le seconde invece definiscono il significato di dati, collegamenti ed interazioni (approfondimento nel Capitolo 3) legati al dominio applicativo.
In questo modo si è riusciti a separare le regole di interpretazione e inoltro dei messaggi HTTP dalle semantiche associate agli elementi del protocollo stesso: gli intermediari sono svincolati dal dover comprendere il significato delle richieste intercettate, rendendo così possibile l'estensione degli elementi architetturali senza la necessità di aggiornare tutta quanta la catena di componenti e connettori.
Per rendere più prevedibile il comportamento dei client in seguito a cambiamenti nel protocollo di comunicazione utilizzato, si utilizza lo *status code*, un parametro presente nelle risposte del server che esplicita il risultato della richiesta in base ad un vasto set di valori predefiniti: questi verranno trattati più in dettaglio nel Capitolo 4 dell'elaborato contestualmente ad un interessante caso di studio.

Come puntualizzato nelle precedenti sezioni, lo stile architetturale ReST impone che i componenti si scambino messaggi auto-descrittivi, in maniera che anche entità intermediarie abbiamo modo di processare i contenuti scambiati. HTTP ha subito diverse modifiche per far fronte a questo vincolo:

- sono state incluse nei metadati della richiesta le informazioni riguardanti l'host di provenienza del messaggio;
- si è delineata una chiara differenza sintattica tra dati di controllo e metadati;
- sono stati aggiunti metadati per descrivere le rappresentazioni con codifiche stratificate, in maniera da non costringere gli intermediari a dover decodificare i contenuti dei messaggi per capirne la semantica.

Oltre al miglioramento delle semantiche di comunicazione tra i componenti, HTTP ha subito negli anni anche un miglioramento dal punto di vista delle performances percepite dall'utente finale. L'intervento più evidente è stato fatto sulla gestione delle connessioni: in un primo stadio HTTP prevedeva una connessione TCP per ogni scambio richiesta/risposta, appesantendo l'interazione con i lenti tempi di setup di ciascuna connessione TCP; con le più recenti versioni dello standard, forti del fatto che i messaggi fossero diventati auto-descrittivi ed indipendenti dal protocollo di trasporto sottostante all'HTTP, si sono rese persistenti le connessioni di default, in maniera da velocizzare le interazioni client/server. Nonostante lo standard HTTP sono stati fatti enormi passi avanti nell'allinearsi ai vincoli dello stile architetturale del Web; ci sono però ancora diversi punti aperti che ne limitano le potenzialità:

- non è ancora supportata dallo standard la possibilità di differenziare le risposte provenienti dal server d'origine da quelle invece ottenute da un componente intermediario o dalla cache, non è quindi possibile avere la certezza che i dati ricevuti siano validi e consistenti con il reale stato della risorsa richiesta;
- non c'è un set di campi obbligatori da includere negli header delle chiamate HTTP che sia sufficientemente esteso per garantire semantiche comuni nell'interpretazione dei messaggi;

- le sintassi dei metadati di una rappresentazione e dei dati di controllo non sono in alcun modo distinte, rendendo difficile la distinzione degli uni dagli altri in uno scenario di controllo stratificato dell'integrità dei messaggi;
- i messaggi HTTP non si possono considerare auto-descrittivi se si considera il fatto che non è tutt'ora possibile ricavare dai contenuti di una qualsiasi risposta la richiesta che l'ha generata. Attualmente è supponibile dall'ordine di ricezione poiché la comunicazione via HTTP è implementata in maniera sincrona, ma dal momento in cui le ultime versioni dello standard (riferimento bibliografico all'RFC di HTTP 1.1) si propongono di essere indipendenti dal protocollo di trasporto sottostante sia necessario porre rimedio a questo limite semantico e architetturale.

Ci si aspetta che nella prossima major dello standard HTTP vengano chiusi questi ed altri punti di minor importanza, anche se si può ben capire che non è un cambiamento facile da apportare su larga scala poiché la risoluzione di alcuni dei suddetti comporterebbe una rottura nella catena di retrocompatibilità che si è cercato di mantenere in tutti questi anni di sviluppi e revisioni del protocollo.

2.4 Il Web 2.0

È già stata sottolineata una delle prime trasformazioni del World Wide Web: Tim Berners-Lee lo aveva progettato in qualità di sistema distribuito ipertestuale per la condivisione di semplici documenti testuali, collegati gli uni con gli altri da una fitta rete di relazioni non per forza lineari; con il passare del tempo si è iniziato ad utilizzare il Web per la condivisione di qualunque tipologia di dato, dai documenti di testo alle immagini, dai contenuti audio a quelli video, trasformandolo in un sistema distribuito ipermediale che raccogliesse informazioni fra loro eterogenee.

Nel 2004 l'editore Tim O'Reilly ha coniato il termine **Web 2.0** [10], sancendo formalmente non una nuova versione del WWW, bensì una nuova tendenza: l'attenzione stava infatti iniziando a spostarsi dal browser e dai comuni contenuti multimediali verso un più ampio insieme di applicazioni software che avrebbe dato una nuova spinta al Web. Si stava iniziando a percepire il Web più come una vera e propria piattaforma piuttosto che non come un semplice contenitore condiviso di informazioni: cose che prima venivano fatte dal sistema operativo iniziavano ad essere rese disponibili in rete sotto forma di servizi (e.g. la posta elettronica non più scaricata su client locale ma consultabile direttamente da browser), i dati stavano cominciando ad essere persistiti sui server di specifici fornitori piuttosto che non sullo storage del proprio computer; mentre gli sviluppatori progettavano applicativi Web per compiti e funzionalità che prima erano offerti da altre piattaforme (e.g. il sistema operativo), gli utenti li utilizzavano sempre più assiduamente portando le tecnologie web nella loro quotidianità. Questo fenomeno di massa di avvicinamento alla rete ha avuto come principale conseguenza la definizione di un profilo sociale, comunitario del Web: le informazioni vengono condivise in maniere sempre più eterogenee con la nascita di strumenti quali i blog, i forum e le chat; si sviluppano piattaforme ideate unicamente per la fruizione di media (Flick, YouTube); il content management trova la sua piena realizzazione nella tecnologia *Wiki* che permette di creare, consultare ed integrare informazioni in un ambiente perfettamente circoscritto; vedono la luce i cosiddetti *social*, servizi e applicazioni per la costruzione di una fitta rete di relazioni sociali con cui condividere il proprio profilo pubblico, tramite cui entrare in contatto per rinsaldare vecchie amicizie e coltivarne di nuove; prendono piede i siti di e-commerce per la vendita di prodotti online, affiancati a sistemi interattivi di valutazione e feedback per gli utenti.

Il Web ha perso quindi le spoglie statiche con cui è nato, diventa dinamico ed

interattivo: gli utilizzatori del Web non si limitano più alla consultazione di documenti, ma sono diventati attori attivi nella creazione e condivisione di contenuti informativi.

È questo lo scenario in cui si vuole introdurre il caldo tema delle Web API, tecnologia che è risultata fondamentale per il passaggio da Web 1.0 a Web 2.0 e che molto probabilmente si confermerà tale anche per le future evoluzioni dello stesso.

Capitolo 3

Web API

Di seguito si analizza come sono andate definendosi le web API nel corso degli anni e come si sono poi evolute divenendo il fulcro dell'intero World Wide Web, poiché sono oramai pochi i sistemi distribuiti che interagiscono con e sul Web a non disporre di un proprio set di API. Approfitteremo di questo capitolo per disquisire anche sulle nomenclature, a volte fuorvianti ed altre equivalenti, con cui si tende solitamente ad appellarle.
La cosa molto interessante - che al momento opportuno verrà con forza sottolineata - è notare come negli ultimi tempi le implementazioni delle web API si stiano conformando sempre di più allo stile e agli standard del World Wide Web per mantenersi semplici, flessibili ed immediate sia nello sviluppo che nelle modalità di utilizzo.

3.1 Overview

Il termine **API** (*Application Programming Interface*) sta ad indicare un'interfaccia per l'agevolazione della comunicazione tra due applicativi software differenti. Sin dall'inizio della storia dei calcolatori si è fatto uso di API; già negli

anni '60 si sentiva l'esigenza di slegarsi dal linguaggio macchina e si cercavano astrazioni sempre più efficaci e funzionali, tant'è che il termine è originariamente nato in riferimento ad un'interfaccia tra l'hardware e il programmatore, in maniera da scaricare quest'ultimo della complessità insita nelle macchine ed evitargli l'onere di replicare ogni volta il codice necessario ad implementare ricorrenti operazioni di routine. Un tema fortemente legato al concetto di API è anche quello dell'*information hiding* [11] che indica la progettazione di un modulo indipendente con il quale sia possibile comunicare tramite la condivisione del minor quantitativo di informazioni possibile per l'esecuzione di una funzionalità software. Ciò porta alla definizione di un'interfaccia minimale sufficientemente stabile da poter essere usata da altri applicativi per accedere ad un predefinito set di risorse e funzionalità. È per questo motivo che solitamente le API vengono paragonate a delle scatole nere, in quanto nascondono al diretto utilizzatore tutto il resto del sistema software che le espone. Uno sviluppatore che fa uso di determinate API non dovrà quindi mai preoccuparsi di eventuali cambiamenti nello strato software sottostante alle stesse API poiché tutto quanto gli continuerà a risultare perfettamente trasparente.

Lo scenario d'utilizzo tipico per le API è stato a lungo quello dei sistemi operativi, basti pensare alle Windows API che Microsoft rese pubbliche per l'implementazione di applicazioni che interagissero con il sistema e ne sfruttassero le funzionalità. Tuttavia, come anticipato nel capitolo precedente, agli inizi degli anni '90 incomincia a prendere piede il World Wide Web, la cui rapida diffusione palesa la necessità di API che astraggano dalle implementazioni dei protocolli utilizzati per facilitare la comunicazione client-server e per gestire lo scambio e la manipolazione delle risorse tra entità non per forza dello stesso dominio organizzativo.

Nascono così le prime *web API*, interfacce di comunicazione per sistemi software distinti che interagiscono con il Web tramite la rete Internet. Si potrebbero defini-

re le web API come un sito Web comprensibile e consultabile dai calcolatori [12]: un umano può accedere tramite il browser al sito web `http://www.weather.com` e avere una rappresentazione grafica delle condizioni meteo attuali; allo stesso modo un'applicazione può accedere alle web API del dominio `weather.com` e reperire le stesse informazioni sulle condizioni meteo in un formato leggibile ed interpretabile dalla macchina. Il Web, infatti, permette alle persone di comunicare e condividere informazioni tramite Internet, allo stesso modo le web API consentono ai siti Web, alle applicazioni mobile ed a qualunque dispositivo di interagire e scambiarsi dati in rete.
Nello specifico, le web API consentono di mettere a disposizione sulla rete informazioni di qualunque genere a sviluppatori, aziende, organizzazioni o addirittura dipartimenti interni allo stesso dominio organizzativo. A seconda degli scopi vengono categorizzate in:

API pubbliche: sono raggiungibili e consultabili da chiunque abbia accesso alla rete Internet, senza bisogno di permessi contrattuali d'alcun genere. Promuovono la condivisione libera dell'informazione, indipendentemente dall'uso che poi ne verrà fatto: attualmente ritroviamo Google tra gli esempi più di spicco;

API private: per quanto il concetto di condivisione incondizionata dell'informazione sia ideologicamente valido, la gran parte delle realtà aziendali punta sulla privatizzazione dei dati, in maniera da sviluppare il proprio business e condividerlo unicamente con partner e clienti.

All'orecchio dei più le API potrebbero ancora sembrare qualcosa di 'geek' ma stanno in realtà diventando il nuovo cuore pulsante del Web e, più in generale, del mondo dell'informazione; poiché permettono di rendere disponibili risorse

e contenuti a chiunque nel mondo, indipendentemente dal livello di competenza tecnica posseduta (riferimento a web semantico, da approfondire nel capitolo 3).

3.1.1 I modelli architetturali più comuni

Gli approcci architetturali più utilizzati per implementare le web API sono sostanzialmente due e ne viene di seguito fornita una breve presentazione:

- **SOA (Service Oriented Architecture):** è stata la tipologia di architettura predominante fino a qualche anno fa ed è tutt'ora la più diffusa, nonostante non sia più la prima scelta dei progettisti. Le caratteristiche dell'approccio SOA sono sicuramente l'estensibilità e la componibilità di un set di servizi autonomi, interoperabili, riusabili e ricercabili tramite appositi provider. Gli standard tecnologici prevedono l'utilizzo di SOAP come protocollo di trasporto di alto livello, WSDL (Web Services Descritpion Language) per la descrizione dei servizi, XML per la formattazione dei messaggi scambiati con i client e WS-* per le specifiche di gestione di funzionalità applicative ad alto livello;

- **ROA (Resource Oriented Architecture):** è lo standard architetturale per l'implementazione di web API che rispettino i principi dello stile ReST e promuovano qualità come disaccoppiamento delle entità in gioco, flessibilità, interoperabilità ed adattabilità [18]. Dipendendo in maniera non indifferente dall'architettura del Web e, dovendo rispettare i vincoli architetturali dello stile ReST, le tecnologie più comunemente utilizzate sono HTTP come protocollo di trasporto, XML o JSON per la formattazione dei messaggi scambiati e WADL (Web Application Description Language) per la descrizione dei servizi esposti.

3.1.2 Disambiguazione

Nelle sezioni seguenti vengono approfonditi temi quali *Web Service* e *ReSTful Web API*. È necessario fare una puntualizzazione sulla terminologia poiché sembra esserci una discreta confusione al rigurado.
Entrambe le tecnologie menzionate sono web API perché offrono un'interfaccia di interazione tra differenti sistemi software che comunicano sulla rete tramite il World Wide Web. Tutti i Web Service sono web API; tutte le ReSTful Web API sono web API; non tutte le web API sono per forza Web Service o ReSTful Web API.
Molto spesso le ReSTful Web API vengono anche chiamate con il nome di *ReSTful Web Service*: non è propriamente corretto poiché sistemi software implementati sulla base dello stile architetturale ReST sono orientati al concetto di risorsa piuttosto che non a quello di servizio.
Infine: gli acronimi SOA e ROA, nonostante rappresentino modelli architetturali, potrebbero essere utilizzati rispettivamente in sostituzione di Web Service e ReSTful Web API in quanto ne sono oramai diventate le implementazioni standard.

3.2 I Web Service

Il World Wide Web Consortium (W3C) fornisce una definizione piuttosto chiara di Web Service in contesto distribuito di rete:

> Un Web Service è un sistema software progettato per supportare l'interoperabilità tra macchine collegate in rete. Ha un'interfaccia descritta in un formato comprensibile alle macchine (WSDL). Altri sistemi possono interagire con il Web Service nella maniera specificata dalla sua stessa descrizione usando messaggi SOAP, tipicamente

> sfruttando HTTP con una serializzazione XML dei dati in congiunzione ad altri standard del Web. [16]

Rappresenta una tecnologia di astrazione per implementare concretamente una web API in ottica service-oriented in un contesto di sistemi distribuiti interoperabili: un Web Service può sfruttare diverse tecnologie per proporsi, con uno stile di comunicazione piuttosto che con un altro, nonostante i più diffusi siano da sempre RCP (Remote Procedure Call) e scambio di messaggi.
Di seguito vengono riportate le caratteristiche principali di un Web Service:

- incapsula logiche relative ad un dato contesto che può essere specifico e dunque legato ad un determinato compito oppure più ampio e relazionato al dominio di una grossa compagnia;
- è facilmente componibile con altri Web Service e riusabile;
- è generalmente autonomo, piuttosto disaccoppiato dal resto del World Wide Web;
- è interoperabile con altri Web Service o sistemi software, non per forza appartenenti agli stessi domini organizzativi;
- il servizio che implementa è generalmente descritto tramite un apposito contratto che può essere più o meno implicito (SOA vs ROA), in modo da favorire l'interoperabilità sopra descritta: ogni interlocutore infatti, consultando la descrizione del Web Service, sa esattamente cosa fare per iniziare una comunicazione.

Ci sono due principali tipologie di classificazione dei Web Service: la prima prevede la suddivisione in base alle responsabilità che hanno in un determinato scenario di interazione con l'esterno, mentre la seconda li classifica in relazione al

ruolo che ricoprono all'interno di un'architettura software.
Nel primo caso spicca la figura del *Service Provider*, un Web Service a cui può essere richiesta l'esecuzione o la descrizione del servizio offerto; il richiedente del servizio, *Service Requestor*, invia una richiesta al provider fornendo tutti quanti i dati necessari per l'esecuzione del servizio e rimane in attesa di una risposta: non sempre richiedente e provider interagiscono in maniera diretta; ci possono essere degli applicativi (*Service Intermediary*) che si frappongono e mediano la comunicazione reindirizzando la richiesta (*mediazione passiva*) ed eventualmente processandola con la possibilità di alterarne i contenuti (*mediazione attiva*).
Tra i tipi di Web Service previsti invece dalla seconda classificazione si annovera il *Service Broker*: è un Web Service a tutti gli effetti e offre un servizio di censimento di tutti quanti Service Provider esponendone i nomi e le descrizioni, così che un qualsiasi Service Requestor abbia la possibilità di ricercare il provider più adatto alle sue esigenze.
SOA è il pattern architetturale più usato per la progettazione dei Web Service e ne definisce principi base quali il disaccoppiamento dal resto del sistema, la componibilità, la riusabilità e l'interoperabilità. Vale la pena approfondirlo perché, visto il collegamento intrinseco con le caratteristiche dei Web Service, è il modello architetturale più adatto per un sistema software orientato ai servizi.
Come già anticipato, i messaggi scambiati in architetture implementate secondo il modello SOA viaggiano su HTTP, ma vengono prima costruiti e formattati secondo le regole del protocollo SOAP (Simple Access Object Protocol): inizialmente ideato per rimpiazzare i protocolli RPC proprietari, poi successivamente trasformato in uno standard per il formato di messaggi generici in seguito adottato dal modello architetturale SOA. Altri protocolli possono essere utilizzati per l'implementazione di Web Services, ma SOAP rappresenta oramai lo standard, anche perché offre estreme flessibilità ed estensibilità per quanto riguarda le strutture dei

messaggi. Un messaggio formattato secondo gli standard del protocollo SOAP è composto principalmente da tre parti, ovvero un contenitore (in gergo *envelope*) che racchiude una parte di header ed una di body: la prima contiene le meta informazioni del contenuto del messaggio, il secondo invece contiene i dati in formato XML necessari al Web Service per l'esecuzione del servizio. I Web Services sono entità logiche autonome, quindi ciascuno di essi possiede un'entità denominata SOAP node (nel linguaggio comune, un programma) usata per trasmettere messaggi in formato SOAP attraverso l'infrastruttura fisica di comunicazione.
Il protocollo che determina come un Web Service debba esporre la propria descrizione è il WSDL che impone di fornire almeno un listato delle funzionalità offerte e dettagli sul comportamento del servizio. La descrizione in formato WSDL è divisa in due parti: una prima, astratta, in cui vengono esplicate le interfacce del Web Service senza alcun riferimento alle tecnologie di implementazione e di trasferimento/ricevimento dei messaggi; una seconda, concreta, che descrive i vincoli da rispettare per una connessione fisica della parte astratta del Web Service ad un protocollo di trasporto (quindi protocollo di rete da usare, indirizzi delle porte tramite cui accedere al servizio, nome del servizio, etc...).
Lo UDDI (Universal Description Discovery and Integration) è uno standard OPEN che prova a risolvere il problema della ricerca e aggregazione di più servizi. Esso prevede un registro a cui sottoscrivere le descrizioni WSDL di un dato Web Service in maniera che possano essere messe in atto politiche di avertising: un Service Requestor consulterà il registro per cercare il Web Service più adatto alle proprie esigenze. Il problema di questo standard è che non vengono imposti vincoli sulle semantiche da rispettare, dunque fa affidamento sulle sintassi della signature della funzionalità offerta, ma non le aggrega semanticamente.

L'argomento potrebbe essere ulteriormente approfondito, andando nello specifico delle tecnologie standard ed analizzando passo per passo le modalità di interazione in uno scenario service-oriented, tuttavia ai fini dell'elaborato è sufficiente una descrizione generale per poter disporre di termini di paragone con l'oggetto di discussione su cui verte il seguito del capitolo.

3.3 ReSTful web API

Riprendendo quanto descritto nel precedente capitolo, il Web si è lentamente focalizzato su tre tecnologie principali: l'HTTP come protocollo di trasporto principale, l'HTML come forma principale di rappresentazione dei dati e gli URI come referenze alle risorse. Questi standard si stanno dimostrando sufficienti per gestire tutti quanti gli scenari applicativi possibili. È dunque lecito porsi la seguente domanda: per quale motivo non vengono sfruttate a pieno le tecnologie su cui si basa lo stesso World Wide Web anche per l'implementazione dei sottosistemi che lo sfruttano per comunicare e, soprattutto, interoperare? Per quanto i Web Services abbiano fortemente segnato il mondo del Web e abbiano contribuito alla definizione di standard e tecnologie tutt'ora indispensabili, sono sfociati in strutture tutt'altro che semplici ed estendibili; oramai vengono implementati sulla base di architetture pesanti e monolitiche che vanno in una direzione del tutto opposta a quella inizialmente tracciata.
Ora sembra essere il momento della rivalsa per il modello architetturale ROA che, sulla base dei vincoli architetturali dello stile ReST e delle astrazioni che ne conseguono, trova una concreta applicazione nelle cosiddette *web API ReSTful* (Richardson and Ruby, 2007): queste possono essere implementate con qualsiasi linguaggio di programmazione, con il protocollo che più si preferisce, ma è chiaro che più ci si attiene agli standard del World Wide Web più tutto risulta semplice.

Il protocollo HTTP e lo standard URI ricoprono tutti i vincoli che Roy Fielding ha esplicitamente elencato nella sua dissertazione: interazione client-server, comunicazione stateless, caching, interfaccia uniforme, stratificazione dell'architettura, code-on-demand.

3.3.1 HATEOAS

C'è un vincolo che però, non essendo stato menzionato nell'elaborato di Fielding, sfugge a molti progettisti e che può invece essere dedotto da una attenta comprensione dei singoli concetti: viene comunemente chiamato **HATEOAS**, abbreviazione di *Hypermedia as the Engine of the Application State*. Parafrasando lo stile ReST, lo stato di un'applicazione è mantenuto dal client, il quale può transitare da uno stato ad un altro unicamente in seguito ad una chiamata HTTP (e ricezione della relativa risposta): come fa dunque un client a sapere quale richiesta deve fare per poter transitare al prossimo stato dell'applicazione? Semplicemente tramite i controlli ipermediali contenuti nella rappresentazione ricevuta in risposta alla richiesta precedente: questo è l'unico modo per garantire il massimo livello di estensibilità in un'architettura progettata a partire dai vincoli del World Wide Web, è l?unico modo per disaccoppiare completamente le entità client da quelle server e renderle indipendenti a tutti gli effetti; così che un server possa cambiare la propria implementazione senza preoccuparsi di compromettere il funzionamento dei client a lui interconnessi. L'ipermedia non è tanto una tecnologia descritta da un qualche set di regole quanto una strategia implementata da innumerevoli standard tecnologici che deve provvedere a:

- istruire il client su come costruire una richiesta HTTP, quindi il metodo da usare, l'URI della risorsa, gli header HTTP o il contenuto da includere nel corpo della richiesta;

- dare garanzie sui contenuti delle risposte HTTP in termini di formato e significato;

- suggerire al client come gestire le varie risposte HTTP nel suo flusso applicativo.

L'ipermedia è quindi descrivibile come il motore del flusso applicativo di un client ed è sempre esistito, fin dalla nascita del Web.
Come mai si inizia ad accorgersene solamente adesso? Semplice: agli inizi il Web è stato un insieme di pagine HTML interconnesse in una perfetta rete ipermediale senza che lo sviluppatore ne fosse pienamente consapevole poiché tutto nascosto dalle logiche trasparenti dei *tag* dello stesso linguaggio. Fino a pochi anni fa i Web Services hanno dominato lo scenario delle web API, quindi è inutile biasimare il fatto di non aver tenuto conto della natura ipermediale del WWW quando i client e i server sono stati portati ad interagire tramite interfacce tutt'altro che generiche.

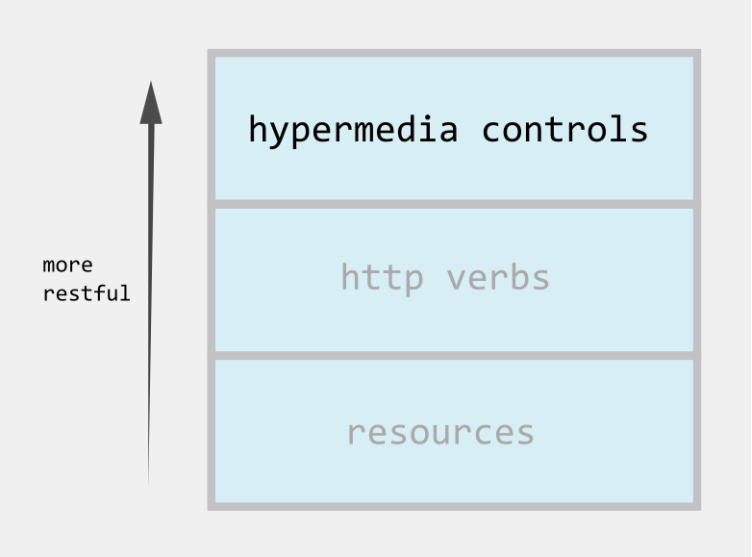

Figura 3.1: Software ReSTful, in sintesi.

L'importanza della semantica applicativa

L'HTML, per quanto presenti ancora dei limiti, è stato descritto nel Capitolo 2 come dotato di innumerevoli controlli ipermediali che favoriscono il conseguimento dell'obiettivo sopra citato. Si facciano invece alcune considerazioni nel contesto di classiche web API che scambino dei contenuti con media type `application/json` su protocollo HTTP: a differenza del tag HTML `<a></a>` che con l'attributo `href` identifica un link ad un'altra risorsa, il campo `"link": "http://www.anotherresource.com"` di un contenuto JSON qualsiasi non rappresenta un collegamento ad una differente risorsa, bensì è semplicemente una stringa da interpretare; il suo significato deve essere spiegato in una qualche documentazione scritta appositamente per esplicitare tutte quante le convenzioni che il provider di API ha stabilito per fornire link ipermediali tramite un formato di dati che non supporta il concetto di ipermedia.
Il problema di fondo è che, a meno di una presenza umana in grado di leggere la documentazione, il client non può sapere in un determinato stato quale richiesta possa essere utile al fine di navigare il flusso applicativo; questo implica che il client non può considerarsi completamente disaccoppiato dal server che espone le API, poiché se vengono cambiati gli URL delle risorse oppure vengono gestiti ulteriori stati applicativi bisogna modificarne il comportamento.
Come però dimostra l'HTML, i controlli ipermediali possono colmare questo divario con le semantiche delle rappresentazioni delle risorse, possono dire al client quale richiesta HTTP fare per navigare ad un certo stato applicativo e soprattutto perché dovrebbe volerla fare. Per guadagnare questa flessibilità servono però formati di dati meno generici dei tag HTML, formati ipermediali definiti sufficientemente nel dettaglio per trasmettere le semantiche applicative dei dati rappresentati nel contesto in esame: a tale scopo si è soliti definire o estendere *media type* specifici che documentino il formato dei dati (le rappresentazioni delle risor-

se) scambiati tra client e server e diano quindi la possibilità ai client di sapere fin da subito i particolari dello scenario applicativo (e.g. il server risponde al client spiegando quali chiamate HTTP avrebbero senso d'essere fatte a partire dallo stato corrente).

Purtroppo però le semantiche dell'applicazione si riferiscono generalmente a cose del mondo reale e i calcolatori non sono intelligenti quanto l'uomo nel comprendere le dinamiche e i concetti del mondo reale: per porre fine al problema bisogna rassegnarsi a scrivere un *profilo*, ovvero una formalizzazione testuale del significato dei dati che integra le semantiche delle rappresentazioni già esplicitate dal media type scelto (si sottolinea il fatto che ad un media type possano corrispondere più profili).

Al riguardo, ci sono diversi enti che si preoccupano di catalogare e render poi disponibili media type e profili di ogni genere, così da renderli riusabili anche in altri contesti che condividono le stesse semantiche applicative. Nel caso in cui queste ultime non coincidano perfettamente, è buona norma estendere un formato con gli elementi mancanti piuttosto che crearne uno nuovo: il vantaggio più lampante è il risparmio di tempo che consegue al non dover studiare nuovamente problematiche di dominio già affrontate da altri; un altro motivo per cui è preferibile evitare la definizione di nuovi media type o profili è evitare l'aumento di entropia nel già disordinato *processo di semantizzazione del Web*.

Tuttavia questo è ciò che succede per la gran parte delle web API progettate e sviluppate al giorno d'oggi: i domini applicativi, a meno di piccole differenze, sono spesso ricorrenti ma ciascuna compagnia utilizza un proprio set di convenzioni rendendosi semanticamente indipendente ed incompatibile dalla concorrenza. Per non rimanere ad un livello d'astrazione eccessivo, si possono prendere come esempio le Facebook API, le Twitter API e le Google API: tutte e tre sono web API con contesti applicativi molto simili tra loro in quanto gestiscono la profi-

lazione di utenti, lo scambio di messaggi, la condivisione di contenuti testuali e multimediali all'interno di date cerchie di contatti ed amicizie, ma ciascuna di loro ha scelto di definire un proprio *standard FIAT* per l'interpretazione semantica delle rappresentazioni scambiate con i client. La conseguenza di questa tendenza è ovvia, i client sono fossilizzati sulle API per cui sono stati inizialmente sviluppati, non sono assolutamente riusabili in contesti applicativi simili e la flessibilità viene del tutto a mancare: realisticamente parlando, in molti casi questo è uno scenario che non preoccupa poiché ciascuna organizzazione ha le proprie API e i propri client senza la necessità o, ancor meglio, la volontà di condividere nulla con la concorrenza; è tuttavia un approccio sbagliato se si vuole progettare API al 100% conformi allo stile ReST per il semplice fatto che non rispecchiano la proprietà di scalabilità nel contesto della rete Internet.

3.3.2 Il processo di progettazione

In seguito alle considerazioni appena fatte, si possono dare per scontate le tecnologie utilizzate per l'implementazione di web API ReSTful; tuttavia non si può fare lo stesso ragionamento per quanto riguarda le procedure di progettazione principalmente per due motivi:

- non ci sono ancora sufficienti standard per le architetture ROA in modo da garantire processi di sviluppo validi e schematici;
- la gran parte del personale IT che si occupa della suddetta fase non ha una preparazione teorica adeguata sullo stile architetturale che si intende adottare.

Uno degli scopi di questo capitolo è proprio quello di proporre metodologie di progettazione e sviluppo tali da soddisfare i requisiti e i vincoli della filosofia ReST.

Vengono ora esposte due possibili soluzioni [13, Chapter 9]: la prima è quella più utilizzata al giorno d'oggi e che porta con sé diverse mancanze, mentre la seconda copre interamente tutti quanti gli aspetti necessari per un'architettura in linea con i principi del Web.

Progettazione breve

L'approccio che si descrive non garantisce di avere delle API ottimali, a volte neppure accettabili. Le tappe in cui si suddivide sono prettamente due:

- scelta di un media type da usare per le rappresentazioni delle risorse esposte dalle API, con conseguenti vincoli sulle semantiche del protocollo di comunicazione e dell'applicazione stessa;

- scrivere un profilo per tutto il resto ed implementare l'architettura sulla base delle conclusioni tratte.

Il motivo per cui questo sia il procedimento più utilizzato risiede fondamentalmente nella sua mal-interpretata semplicità: per smarcare il primo punto è sufficiente scegliere JSON come formato delle rappresentazione, mentre per il secondo non serve preoccuparsi di vincoli sulle semantiche, poiché JSON non ne pone, ma basta spendere un po' di tempo nella definizione di uno *standard Fiat* adatto alle necessità dell'applicazione e corredato con documentazione consultabile.
Gli svantaggi sono evidenti: primo fra tutti il fatto che le API progettate seguendo questo approccio non sono estendibili in alcun modo senza compromettere la stabilità e il corretto funzionamento dei client che vi si interfacciano; bisogna poi tenere conto del fatto che si aumenta l'entropia - già di per sé elevata - nel campo degli standard delle semantiche applicative poiché ognuno definisce il proprio profilo *ad-hoc* senza interessarsi minimamente ad eventuali riusi in altri contesti.

Progettazione completa

Per ottenere un risultato in linea con le proprietà e i requisiti dello stile ReST è necessario approfondire il procedimento sopra illustrato in maniera da coprire meglio alcuni punti essenziali che solitamente si tende ad ignorare. Non è un segreto il fatto che questo secondo metodo risulti essere più costoso ed impegnativo, ma i benefici sono molteplici e verranno analizzati nell'esposizione delle tappe di progettazione:

- individuare quali siano le risorse che le API devono esporre. L'errore più comune che viene fatto è considerare tali risorse sulla base di dettagli implementativi fuorvianti, quali le classi del paradigma Object Oriented su cui si basa il linguaggio che si intende utilizzare oppure le tabelle del database relazionale in cui verranno persistiti i dati. Devono essere definite invece sulla base degli scopi applicativi ed in relazione a come possono essere viste da un ipotetico client: devono essere dei veri e propri descrittori semantici dei possibili stati applicativi. È buona norma raggruppare in maniera gerarchica risorse strettamente legate da un punto di vista semantico, la cui relazione risulti dunque intuitiva e sensata;

- disegnare un diagramma a stati delle API che raggruppi i descrittori semantici e li colleghi in maniera da delineare un flusso di transizioni di stato ragionevole dal punto di vista delle logiche applicative desiderate. Si dovrebbe approfittare di questo passaggio per assegnare alle varie transizioni un metodo HTTP che ne identifichi gli effetti sulla data risorsa. Al termine di questo step devono essere chiari quali sono i descrittori semantici e quali relazioni li colleghino, così da capire le semantiche di protocollo delle API (dunque quali richieste HTTP potrebbero essere fatte da un client) e quel-

le applicative (quali informazioni dovranno essere scambiate tra un client e server);

- il passo successivo prevede di trasformare i descrittori semantici e i relativi collegamenti reciproci in stringhe: queste ultime dovranno essere mappate da un profilo ben definito in maniera da ufficializzare le semantiche previste dalle API e renderle facilmente riusabili. Il profilo è molto importante, soprattutto in termini di estensibilità e flessibilità delle API che vi fanno riferimento, perché è alla base della struttura ipermediale dell'architettura: conviene quindi spenderci il tempo necessario, ricercandone uno già esistente che si adatti al caso specifico - a tal proposito l'ente *IANA* ne fornisce un consistente elenco - o scrivendone interamente uno che documenti le semantiche applicative in questione;

- scegliere un media type che sia compatibile con le semantiche sia del protocollo di comunicazione sia applicative: adottare un media type specifico per il dominio applicativo trattato potrebbe essere la scelta migliore, anche se implicherebbe una rivalutazione generale sul punto precedente; l'alternativa sarebbe creare un nuovo media type in modo da soddisfare completamente i requisiti semantici delle API che si stanno progettando;

- è consigliabile scrivere un ulteriore profilo che documenti le semantiche applicative in un formato standard (e.g. ALPS, JSON-LD): lo scopo è quello di spiegare tutti quanti i descrittori semantici e le tipologie di collegamenti che non sono già state descritte da uno dei due punti precedenti. Se il media type scelto per il dominio applicativo delle API è stato creato ex novo è possibile evitare questo step includendo quante più informazioni possibili nella descrizione del media type stesso;

- è poi necessario sviluppare un server HTTP che implementi il diagramma degli stati del terzo step, un client che invii richieste HTTP dando luogo alle dovute transizioni di stato ed ottenendo rappresentazioni delle risorse (i descrittori semantici) come risposta. Ciascuna di queste rappresentazioni dovrà essere formattata secondo il media type che è stato scelto e collegata al profilo dei collegamenti definito;

- si può infine procedere con la pubblicazione sul Web dell'URL di root delle API: infatti, se tutti i punti precedenti sono stati eseguiti correttamente, l'utente avrà solo bisogno di un punto di inizio per interfacciarsi a dovere con le API sviluppate poiché tutte quante le risorse saranno raggiungibili tramite l'interpretazione dei collegamenti e l'esecuzione delle relative transizioni di stato definite dal media type e dal profilo.

Si termina la descrizione di questa procedura di progettazione con una raccomandazione: nel caso in cui si riveli necessaria la definizione di nuovi formati di dati, conviene rendere pubblici il media type e i profili creati per favorire la riusabilità di semantiche di protocollo ed applicative anche in altri domini. Negli ultimi anni si sta cercando di guidare lo sviluppo web attraverso un processo di semantizzazione che risulta tutto fuorché semplice ed immediato: rendere riusabile il proprio lavoro non può che aiutare nel tentativo.

3.4 Conclusioni

In questo capitolo sono stati descritti i modelli architetturali SOA e ROA per la progettazione di web API, una delle tecnologie di riferimento per lo sviluppo di sistemi distribuiti che necessitano di interfacce per la comunicazione con l'esterno.

Si trova traccia ovunque di un dibattito ancora aperto tra i sostenitori dell'uno e dell'altro modello architetturale: vero è che negli ultimi tempi le implementazioni ROA stanno iniziando a diventare la tendenza principale, nonostante molti sviluppatori si ritrovino ancora perplessi su quale dei due adottare.
Essendo state utilizzate per parecchi anni in maniera piuttosto intensiva, le implementazioni SOA attualmente sono in grado di fornire soluzioni *Business-to-Business* piuttosto stabili e complete, con applicazioni per l'integrazione di Web Services già esistenti e rodate. Questo perché sono corredate da tool maturi che consentono sviluppo e manutenzione rapidi ed agevoli. Le implementazioni SOA possono vantare anche standard ben progettati e più volte revisionati per promuoverne l'interoperabilità, basti pensare al protocollo WSDL per la descrizione delle funzionalità e delle interfacce di interazione o agli standard WS-* per il supporto di funzionalità ad alto livello applicativo.
Nell'ultimo periodo le implementazioni ROA, d'altra parte, possono vantare un notevole incremento dell'indice di diffusione; hanno infatti un grosso vantaggio rispetto alla concorrenza che probabilmente ne definirà, col tempo, il successo: la flessibilità. Nonostante una buona progettazione richieda tempistiche tutt'altro che brevi, sono concettualmente semplici ed intuitive poiché si basano sugli stessi vincoli e standard tecnologici con cui è costruito il Web stesso: l'approccio è sicuramente più diretto ed immediato, flessibile ed agile in ogni sua forma, promotore di estensibilità e retrocompatibilità; motivo per cui grosse compagnie stanno iniziando a mettere in discussione le attuali soluzioni orientate ai servizi in favore di re-implementazioni ReSTful.
L'unica grande pecca del modello architetturale ROA è che non sono ancora stati definiti sufficienti standard per disciplinarne le implementazioni: nel lungo periodo queste mancanze potrebbero portare a problematiche di non poco conto, soprattutto in termini di scalabilità ed interoperabilità con altri sistemi software.

Capitolo 4

Caso di studio

Nella prima parte di questo capitolo viene fatta un'introduzione su un progetto di marcature degli accessi sviluppato per conto dell'azienda *SPOT Software srl*, descrivendone le caratteristiche e le componenti hardware e software che lo costituiscono senza però soffermarcisi eccessivamente in modo da non distogliere l'attenzione da quello che è il tema portante dell'elaborato. Infatti nella seconda parte del capitolo il suddetto progetto viene preso come caso di studio per approfondire e toccare con mano tutta quanta la parte teorica fino ad ora affrontata: nello specifico si trattano le fasi di progettazione ed implementazione delle web API in ottica ReSTful, fondamentali per l'interoperabilità degli elementi costituenti del progetto. Non saranno analizzati nel dettaglio gli aspetti implementativi di ogni singolo modulo delle API, ma si cercherà di dare più importanza alle linee guida che sono state seguite e che possono tornare utili in qualsiasi altro contesto tecnologico.

Per correttezza, si ritiene opportuno specificare che la presentazione del caso di studio fornita nella prossima sezione è basata sull'eccellente lavoro svolto da Andrea Corzani nella sua tesi di laurea magistrale [4], sviluppata a più ampio raggio sullo stesso progetto.

4.1 Overview sul progetto

Il progetto è nato da un'esigenza reale e piuttosto incombente per SPOT Software: il sistema con cui vengono attualmente gestiti gli accessi in ufficio non è sufficientemente affidabile ed è poco pratico nelle modalità di utilizzo, soprattutto dal punto di vista amministrativo. Si è dunque scelto di progettare, assemblare e sviluppare internamente una nuova piattaforma così da agevolare sia il processo di marcatura ed ingresso in azienda sia la fase di consultazione e analisi dei dati raccolti, garantendo anche all'azienda stessa maggior controllo sul software e sull'hardware utilizzati dal sistema. Motivazioni non da meno sono state sicuramente la voglia di toccare con mano tecnologie nuove, di sperimentare casi d'uso differenti da quelli solitamente affrontati, di mettersi alla prova in scenari applicativi mai esplorati, il tutto a vantaggio dei singoli e dell'azienda stessa poiché le competenze richieste da una piattaforma simile toccano ambiti decisamente importanti nel settore dell'IT odierno. Il lavoro è stato portato avanti anche con la speranza di riuscire a pubblicare quanto realizzato, rendendolo così disponibile ad altre aziende aventi la stessa necessità di SPOT Software.

La piattaforma è a tutti gli effetti un sistema distribuito, in quanto costituita da entità computazionali autonome, spazialmente separate, collegate tra loro attraverso una rete e che collaborano per il raggiungimento di un obiettivo condiviso; per quelli che sono la sua struttura e i suoi fini *io@spot* si inserisce perfettamente in un contesto **IoT** (*Internet of Things*), in cui tra i principi fondamentali si ritrovano il monitoraggio degli spazi, l'automatizzazione delle operazioni più frequenti e l'agevolazione dei processi decisionali tramite l'analisi delle informazioni raccolte dalle sempre più diffuse tecnologie embedded - il cui scopo è proprio quello di dotare di intelligenza oggetti altrimenti privi di facoltà d'elaborazione.

Nello specifico, l'obiettivo principale del progetto è quello di faciliare la rou-

tine di timbratura per i dipendenti dell'azienda con l'ausilio delle più recenti tecnologie di comunicazione che i sistemi embedded e i dispositivi mobile mettono a disposizione; a tale scopo è prevista l'installazione di un sistema embedded nei pressi della porta dell'ufficio che avrà il compito di identificare gli utenti, validarne e registrarne l'accesso su un apposito database e far scattare la serratura della porta stessa.
Un aspetto molto interessante è stato il riconoscimento degli utenti, realizzato tramite l'utilizzo di due tecnologie che nell'ambito IoT si stanno oramai affermando in qualità di standard:

- *Near Field Communication* (NFC): è una tecnologia per la fornitura di connettività wireless bidirezionale a corto-raggio con modalità d'utilizzo semplice, intuitiva e sicura. Nasce sulla base delle specifiche RFID (Radio Frequency Identification) con la notevole differenza di consentire comunicazioni bidirezionali, così che due dispositivi possano interagire a distanza di pochi centrimetri l'uno dall'altro. Lo scopo dell'NFC non è il trasferimento di grandi quantità di dati, bensì lo scambio di piccoli contenuti informativi in maniera sicura per agire da ponte a servizi che necessitano di interoperabilità. Gli attori di una qualsiasi comunicazione NFC sono suddivisi in due categorie, attivi e passivi: i primi sono dispositivi elettronici dotati di alimentazione che implementano tutto quanto lo stack protocollare necessario all'interazione peer-to-peer e alle funzioni di lettura/scrittura dati; i secondi invece, anche denominati *tag*, sono piccoli chip elettronici dotati di memoria non volatile e di intelligenza sufficiente per comunicare con un dispositivo attivo, solitamente applicati su oggetti comuni per aumentarne il contenuto informativo;

- *Bluetooth Low Energy* (BLE): è una derivazione dello standard tecnologico Bluetooth conosciuta per i ridotti consumi e costi di mantenimento. Rientra

nello standard di specifiche del Bluetooth 4.0 ed è oramai una tecnologia integrata in tutti i dispositivi mobili; si sposa alla perfezione con il paradigma IoT poiché predisposta allo scambio frequente di dati in scenari applicativi, ad esempio, di monitoraggio o localizzazione.

Di pari passo è richiesta la realizzazione di una applicazione web a supporto delle organizzazioni nella consultazione dei dati d'accesso e delle presenze in ufficio degli utenti: questa applicazione può essere opzionalmente messa a corredo del sistema embedded, ma deve poter funzionare anche da sola consentendo l'imputazione manuale delle timbrature.

4.1.1 La struttura

Il sistema distribuito di *io@spot* può essere suddiviso in tre sistemi software differenti, ma fortemente collegati l'uno all'altro:

- una web application per l'accesso diretto degli utenti e degli amministratori ai dati pertinenti al dominio organizzativo di appartenenza, tramite un'interfaccia visuale semplice ed intuitiva e con il supporto di un solido backend cloud per la validazione degli accessi e la storicizzazione delle timbrature;

- uno *smart device* - il sistema embedded prima introdotto - che deve essere installato all'interno degli spazi aziendali per l'automazione del processo di timbratura e apertura della porta d'ingresso, previa autenticazione dell'utente. Il vincolo più forte che deve essere rispettato riguarda l'associazione univoca tra il singolo device e l'organizzazione presso cui viene installato: si è poi deciso di non dotare il dispositivo di un proprio database, ma di farlo accedere tramite la rete ad una base dati condivisa con gli altri componenti della piattaforma. Espone moderne interfacce di connettività NFC e BLE per la comunicazione diretta con l'utente, supporta tuttavia anche i

classici badge *RFID* per garantire le funzionalità di timbratura ed accesso anche in caso di mancata disponibilità di uno smartphone compatibile con le suddette tecnologie;

- un applicativo mobile, inizialmente solo per sistema operativo Android, che permette all'utente di accedere alle principali funzionalità del sistema con l'ausilio dello smartphone, nel caso in cui questo supporti connettività wireless di prossimità come NFC e BLE: per rispettare i requisiti iniziali, si integra con la piattaforma accedendo alla base dati condivisa e consentendo all'utente l'inserimento delle proprie credenziali in modo da poter essere riconosciuto dallo smart device. È la parte più semplice dell'intero sistema, ma probabilmente quella che dà all'utenza una maggiore percezione della comodità di *io@spot*.

4.2 Analisi del dominio

Sulla base delle considerazioni appena fatte, si procede all'analisi del dominio per definire le entità che sono contemplate dal dominio applicativo di *io@spot* e per evidenziare le relazioni che le legano l'una all'altra. Questa è una fase molto importante, poiché pone le basi per l'analisi dei requisiti, la progettazione e l'implementazione di tutte e tre le componenti del sistema.

User - identifica l'utilizzatore di *io@spot*, dunque un utente registrato al servizio poiché non sono previsti scenari applicativi di dominio pubblico. È identificato dall'indirizzo email utilizzato in fase di registrazione.

Organization - È sostanzialmente l'azienda che usufruisce del servizio, ad essa è associato un insieme di utenti che si distinguono in base al possesso di pri-

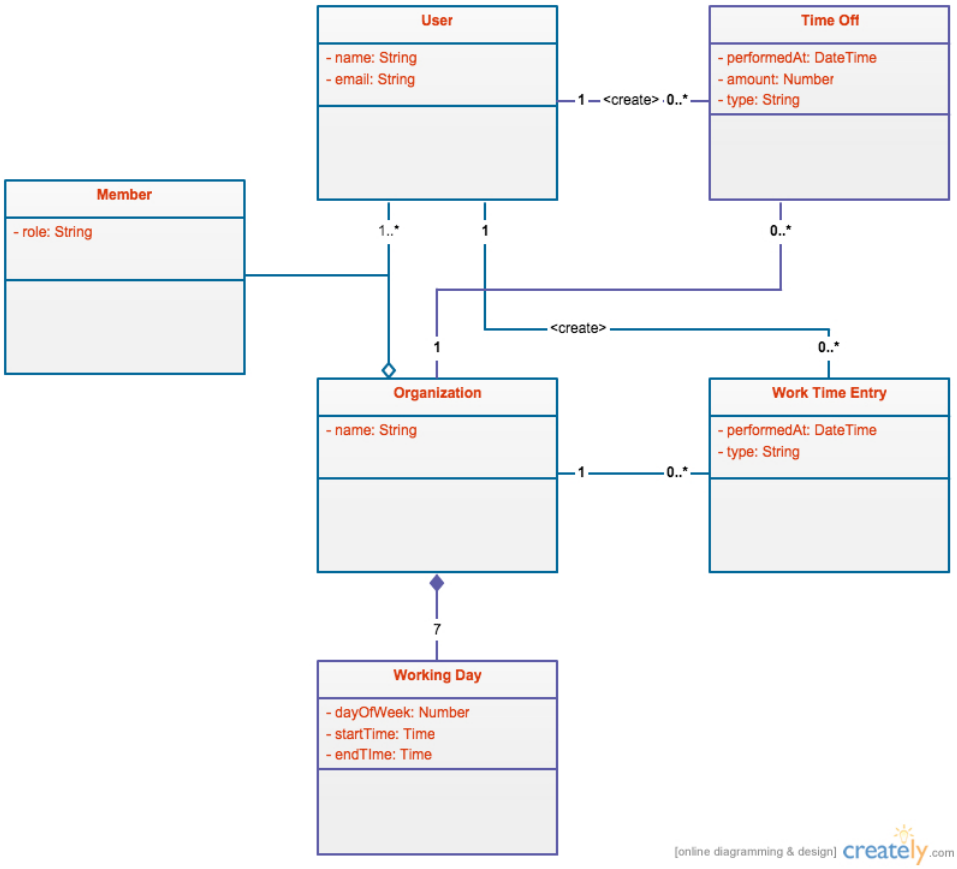

Figura 4.1: Il dominio applicativo.

vilegi d'amministrazione o meno. Ciascuna organizzazione è accompagnata da informazioni riguardanti la settimana lavorativa.

WorkingDay - Rappresenta la configurazione di una giornata lavorativa settimanale per la quale possono essere specificati gli orari di inizio e fine.

Member - Non è tanto un'entità quanto la relazione tra un qualunque utente e un'organizzazione: va letta come *l'utente "user" è membro dell'organizzazione "org"* e prevede un unico descrittore che identifica il ruolo dell'utente nell'organizzazione.

TimeOff - S'intende l'assenza a lavoro che un utente può specificare per un dato giorno: è relativa ad una data specifica e può variare per tipologia e numero di ore di assenza effettiva.

WorkTimeEntry - Rappresenta la timbratura: allo stesso modo dell'assenza è associata sia ad un utente che ad un'organizzazione ed è caratterizzata dal *time-stamp* di data ed ora nelle quali è avvenuta e dalla tipologia (e.g. entrata, uscita, inizio pranzo, fine pranzo...).

4.3 Analisi dei requisiti

Si riportano brevemente i requisiti individuati per il sistema nella sua interezza, senza distinguere i singoli sotto-sistemi che lo compongono:

- il sistema deve garantire un buon livello di sicurezza perché gestisce dati che sono da considerare sensibili per l'azienda che lo utilizza;
- il database utilizzato dal sistema deve essere centralizzato sul cloud e quindi accessibile tramite il supporto della rete Internet, così da poter essere sfruttato anche dalle aziende come servizio cloud (*software as a service*);
- il sistema deve essere scalabile ed estendibile, infatti le necessità e le specifiche di funzionamento possono variare in maniera continua ed imprevista;
- il sistema deve essere semplice ed intuitivo nelle modalità di utilizzo previste, siano esse incentrate sull'interazione hardware diretta o sull'interfacciamento con un applicativo software dedicato.

Nonostante un'analisi come questa possa sembrare minimale e sommaria, se si prendono in esame i componenti della piattaforma i requisiti si moltiplicano e det-

tagliano delineando un sistema complesso e tutt'altro che semplice da progettare e realizzare.

4.3.1 I requisiti della web application

Ai fini dell'elaborato si riporta per intero il risultato dell'analisi dei requisiti della web application [4, p. 42-43]:

- tutti i servizi primari dell'applicazione devono essere disponibili soltanto agli utenti autenticati, quindi non è prevista una parte "pubblica", escluse le canoniche funzionalità per la registrazione e l'autenticazione;
- l'utente può registrarsi direttamente con email e password, oppure può autenticarsi con il proprio profilo Google;
- il sistema deve prevedere la possibilità per gli utenti di recuperare la password dimenticata;
- all'interno dell'applicazione gli utenti devono essere raggruppati in organizzazioni. Gli utenti potranno avere, per ogni organizzazione a cui appartengono, il ruolo di utente oppure amministratore;
- ogni utente deve appartenere ad una o più organizzazioni;
- quando l'utente effettua il login, deve necessariamente scegliere l'organizzazione per la quale vuole visualizzare i propri dati. Può scegliere di cambiare l'organizzazione selezionata in un qualunque momento durante la propria sessione;
- l'utente ha accesso a due principali sezioni: timbrature e assenze;
- per ognuna di queste, potrà visualizzare, inserire, modificare ed eliminare i dati a lui collegati;

- oltre alle proprie timbrature e assenze, un utente amministratore di una determinata organizzazione deve poter vedere, modificare o eliminare anche tutti i dati relativi agli altri utenti dell'organizzazione;
- in aggiunta alle due viste principali, l'utente deve poter accedere ad una sezione secondaria di amministrazione, che dovrà consentirgli di gestire il proprio profilo e le organizzazioni nelle quali ha il ruolo di amministratore;
- un utente amministratore di una determinata organizzazione può modificare gli utenti che vi appartengono e i relativi ruoli;
- l'amministratore può modificare gli orari lavorativi dell'organizzazione (inizio - fine);
- l'amministratore può modificare le tipologie di assenza che gli utenti dell'organizzazione possono inserire;
- l'applicazione deve esssere cross-browser e cross-device.

4.3.2 Casi d'uso della web application

Per completezza vengono riportati in figura 4.2 i casi d'uso della web application: come già detto, è il sotto-sistema di *io@spot* più significativo per il presente elaborato ed è dunque necessario comprenderne a fondo i requisiti, gli scenari di utilizzo e i vincoli che ne hanno segnato la progettazione e lo sviluppo.

Si distinguono immediatamente le due tipologie di attori che possono interagire con la web application: il semplice utente (e.g. il dipendente dell'azienda) può accedere ad ambienti di visualizzazione e modifica di profilo, timbrature ed assenze che gli appartengono all'interno di un determinato contesto organizzativo; l'amministratore che, in aggiunta, può modificare le informazioni dell'organizza-

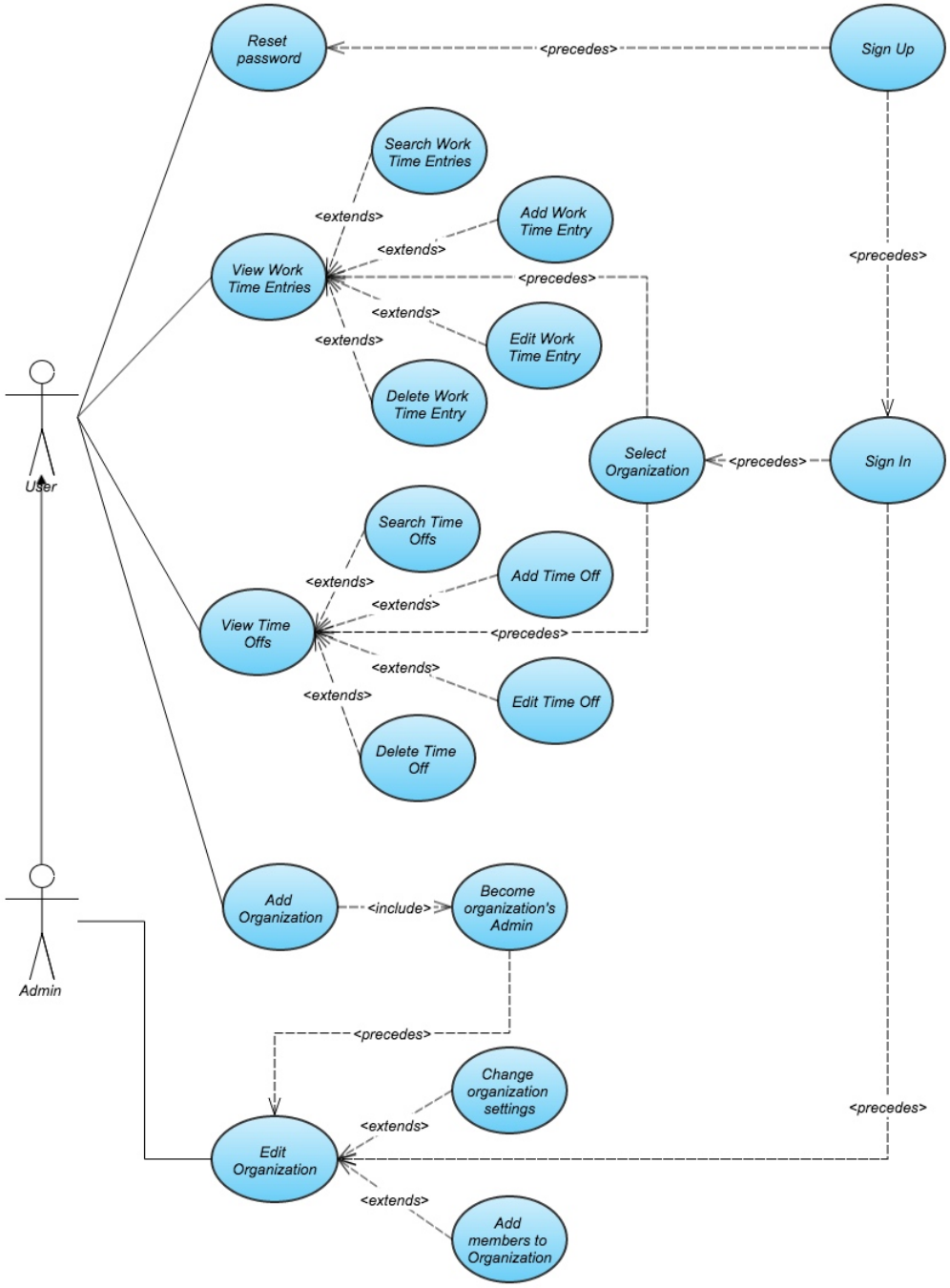

Figura 4.2: Uno schema dei casi d'uso legati alla web application.

zione in cui ricopre tale ruolo, consultare i dati di tutti gli utenti che ne fanno parte ed eventualmente elevarne i privilegi.

E' molto chiaro il concetto di autenticazione: al di là del caso di reset della password, non ci sono scenari in cui non sia prevista la necessità di una login preventiva e, dunque, di essere registrati al servizio.

4.4 Analisi logica

Si cerca ora di enfatizzare e separare logicamente le funzionalità ed i compiti delle parti che compongono il sistema, rappresentandole da un punto di vista strutturale: ciascuna di esse è suddivisa negli strati software relativi alle dimensioni di informazione, controllo e presentazione [7], come mostrato nella figura 4.3.

I requisiti non hanno enfatizzato il server, lo si è quasi dato per scontato, ma strutturalmente parlando risulta essere la componente più importante e vale la pena approfondirlo per poter introdurre l'argomento di principale interesse per l'elaborato. È chiaramente il *core* di tutta quanta la piattaforma poiché implementa le logiche del dominio e si occupa della persistenza dei dati, consentendo una completa interoperabilità dei moduli applicativi precedentemente descritti:

- lo strato di presentazione è composto unicamente dai contenuti HTML necessari alla web application per l'interazione via browser con l'utente;

- lo strato di informazione è costituito da un database su cui sono salvati i dati relativi alle entità del dominio;

- lo strato di controllo è formato da tutti i moduli software necessari all'interfacciamento con i tre applicativi client previsti ed assolve a compiti di fondamentale importanza come l'autenticazione degli utenti e l'accesso ai

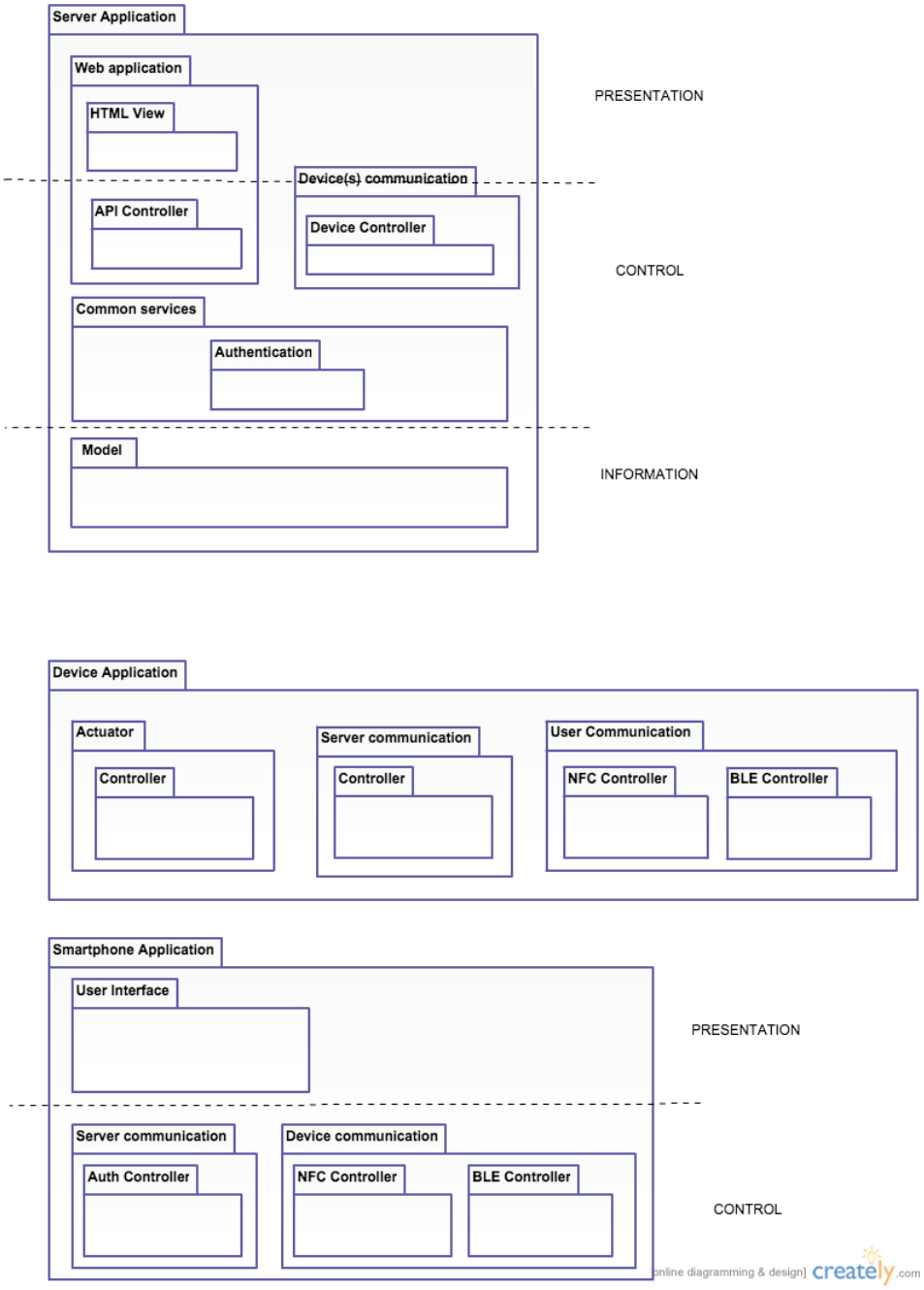

Figura 4.3: Architettura logica del server.

dati. Fra i tre si dimostra essere lo strato più trasversale all'intero sistema *io@spot*.

È proprio in quest'ultimo strato che ritroviamo le web API e, prima di trattarle nello specifico nelle fasi di progettazione ed implementazione, si propone uno scenario tipo di interazione tra la web application e l'API controller con il diagramma di sequenza in figura 4.4.

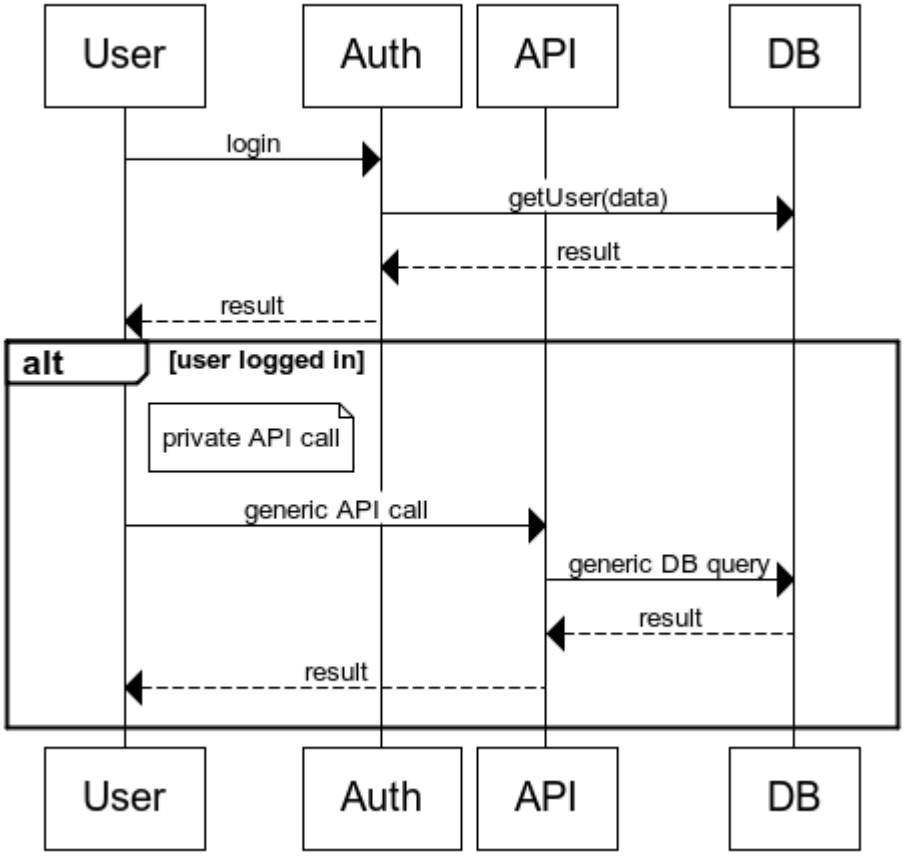

Figura 4.4: Diagramma di sequenza dell'interazione tra gli strati logici del server in uno scenario tipo di utilizzo della web application.

4.5 La progettazione delle web API di io@spot

Per quanto interessante, si mette da parte tutto il resto del sistema - per approfondimenti si rimanda alla tesi di laurea magistrale di Andrea Corzani [4] - così da concentrare l'attenzione sulla progettazione e sull'implementazione delle web API di *io@spot*.

Sono state concepite sulla base dei principi architetturali dello stile ReST ampiamente trattato nei capitoli 2 e 3, all'inizio più per curiosità che per altro. C'è stata infatti in SPOT Software l'esigenza di trovare risposta alla domanda che al giorno d'oggi molti sviluppatori e progettisti software si pongono: come mai il mondo IT migra nella direzione di implementazioni ReSTful quando ci sono soluzioni orientate ai servizi che offrono tool e linee guida temprati ed assodate da più di vent'anni di diffusione?

Tutto questo perché le applicazioni web diventano sempre più corpose e necessitano non solo di strumenti di sviluppo ben fatti, ma anche di solide garanzie in termini di performance, scalabilità ed affidabilità: come descritto nei capitoli precedenti, i vincoli dello stile architetturale ReST consentono, se applicati con criterio, di ottenere ottimi risultati al riguardo.

Il processo di progettazione delle web API della piattaforma *io@spot* è stato affrontato seguendo gli step della metodologia completa presentata nel capitolo 3 sulla base delle considerazioni di Richardson [13].

4.5.1 Definizione dei descrittori semantici

Partendo dai requisiti del sistema e dal modello del dominio già discusso ed esplicitato dalla figura 4.1, si dividono le risorse nelle rappresentazioni che le API devono poi esporre, specificandone le semantiche.

Ciascun utente registrato ha un profilo con tutte quante le informazioni che lo riguardano, incluse le credenziali di accesso.

- Un utente registrato al servizio
  - il nome dell'utente
  - l'email con la quale si è registrato al servizio e che lo identifica univocamente
  - la password che consente all'utente l'accesso al servizio
  - **l'ultima organizzazione ha cui ha effettuato l'accesso**

Ogni utente può appartenere ad una o più organizzazioni, deve quindi essere possibile accedere a tutte le organizzazioni di cui è membro.

- La lista di organizzazioni di cui un utente è membro
  - Un'organizzazione
    - il nome dell'organizzazione
    - il numero di ore lavorative previste giornalmente per l'organizzazione
    - una lista delle tipologie di assenze contemplate dall'azienda
    - la lista dei giorni lavorativi
      - un giorno lavorativo con informazioni sugli orari previsti per l'inizio e la fine delle attività lavorative
    - la lista dei membri dell'organizzazione
      - **un membro dell'organizzazione**

Un utente deve poter accedere alle marcature effettuate.

- Una lista di marcature di accessi

  - una marcatura di acesso
    - **l'utente a cui appartiene la marcatura**
    - **l'organizzazione presso cui è stata effettuata la marcatura**
    - la data in cui è stata effettuata la marcatura
    - la tipologia della marcatura, o entrata o uscita
    - la modalità in cui è stata eseguita la marcatura, se in maniera manuale tramite la web application o in maniera automatica tramite lo smart device

Allo stesso modo devono essere accessibili le assenze inserite da un utente.

- Una lista di assenze
  - Un'assenza
    - la data
    - il totale di ore d'assenza
    - la tipologia di assenza
    - una descrizione che integri la tipologia dell'assenza
    - **l'utente che ha segnato l'assenza**
    - **l'organizzazione presso cui l'utente è risultato assente**

I descrittori semantici delle entità del dominio applicativo sono stati suddivisi per semplicità in quattro macro-blocchi e strutturati gerarchicamente: ogni livello fornisce una rappresentazione specifica. È necessario fare una puntualizzazione al riguardo dell'entità di dominio *WorkingDay*: non è stata trattata come una risorsa a parte perché non è sufficientemente importante per essere soggetta ad una specifica interazione client-server, è piuttosto un semplice attributo della rappresentazione di una organizzazione.

Negli elenci qui sopra sono stati contrassegnati in grassetto alcuni descrittori e al momento la cosa può sembrare priva di significato: è compito del prossimo step chiarire il tutto. Prima di passare oltre però è interessante notare come il dominio di *io@spot* possa essere definito sulla base del pattern collection [13], aspetto piuttosto decisivo nella scelta del media-type delle rappresentazioni.

4.5.2 Diagramma degli stati applicativi

Prima di iniziare a parlare di media-type bisogna tenere in considerazione il fatto che lo stile architetturale ReST è incentrato sul concetto di ipermedia, è dunque necessario definire quali siano i collegamenti tra le rappresentazioni appena definite così da renderle "navigabili" in base alle loro stesse semantiche. È ovviamente un processo iterativo poiché è difficile ottenere subito un diagramma degli stati che soddisfi i requisiti applicativi. Viene da sé che raffinando il diagramma degli stati possono essere messi in discussione alcuni aspetti delle rappresentazioni: alcuni descrittori semantici possano infatti essere trasformati in relazioni, per il semplice fatto che a volte sono meglio descrivibili come una transizione di stato piuttosto che come vere e proprie proprietà delle rappresentazioni. Ecco spiegati i descrittori semantici in grassetto: come mostrato in figura 4.5, sono preferibili sotto forma di collegamenti ad altre risorse.

Per ottenere un diagramma degli stati in linea con i vincoli ReST è necessario definire una rappresentazione che funga da "home page" in modo da poter poi raggiungere tutte quante le altre rappresentazioni semplicemente seguendo le transizioni di stato derivate dalle relazioni reciproche. Come illustra il diagramma della figura 4.5, la rappresentazione di partenza è l'utente loggato: infatti le web API prese come caso di studio non comprendono la parte di autenticazione - poiché difficilmente riconducibile ai principi di ReST ed al concetto di risorsa - e gli scenari contemplati danno per scontato che l'utente abbia effettuato l'accesso.

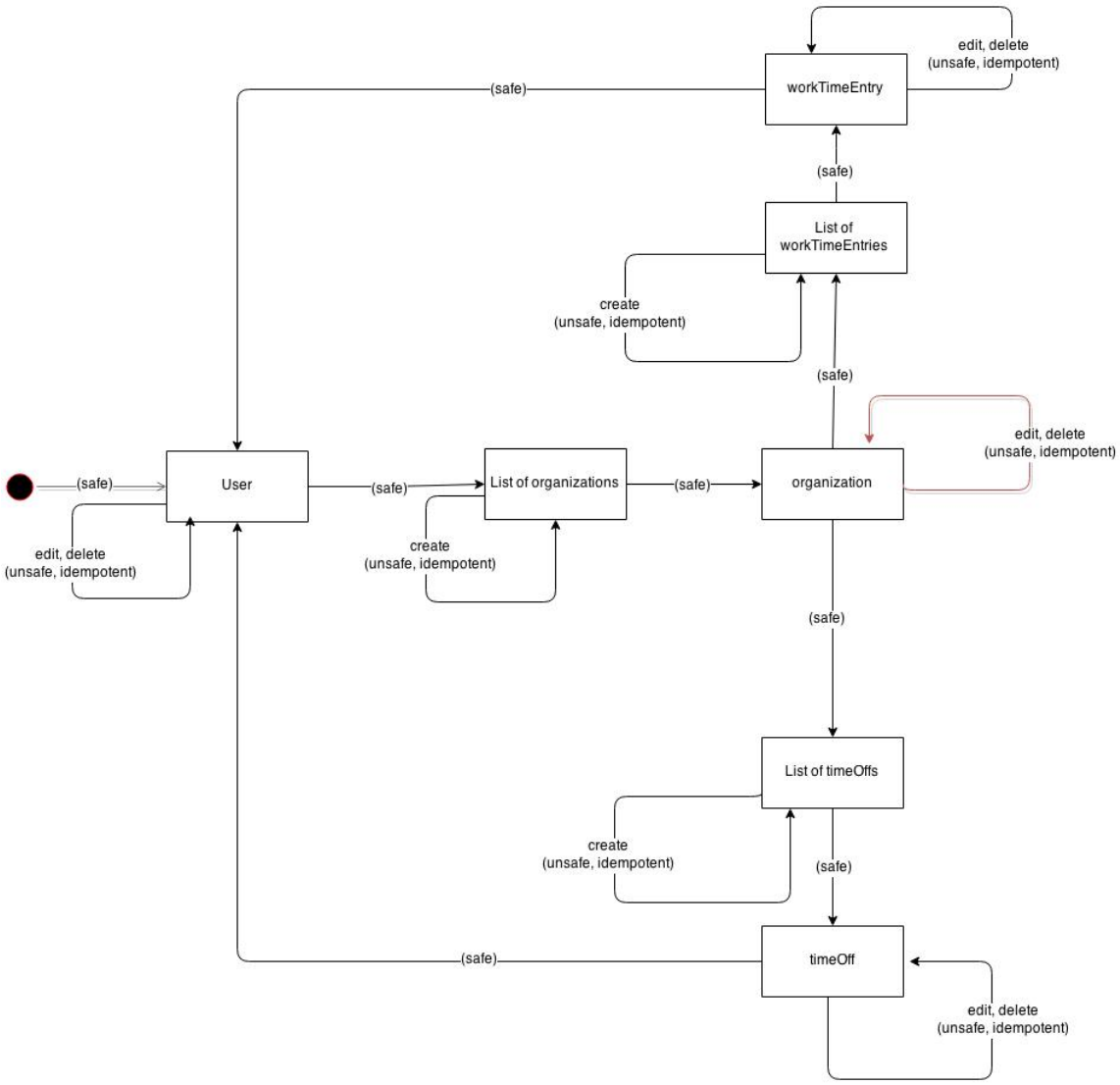

Figura 4.5: Il diagramma degli stati applicativi.

4.5.3 Definire i nomi

Tutti quanti i descrittori semantici e le relazioni definite nei primi due step della progettazione sono da etichettare con nomi appositi: dal punto di vista strutturale è un passaggio quasi superfluo, non incide infatti in alcun modo con la scalabilità e le performance del sistema, tuttavia ha un'importanza non trascurabile

da una prospettiva più orientata alla definizione e standardizzazione delle semantiche applicative. A parte l'adozione di alcune relazioni già registrate dalla IANA [6], non si è fatto uso di alcun profilo già esistente e reso disponibile tramite le apposite *authority*, principalmente dovuto al fatto che il dominio di *io@spot* non è descritto da nessuno dei profili consultati: d'altro canto si è cercato di adottare nomi quanto più comprensibili in maniera da limitare la documentazione necessaria a spiegare le semantiche applicative specifiche del sistema.

Il risultato finale applicato al diagramma degli stati è mostrato in figura 4.5.

4.5.4 Scelta del media-type

Come già anticipato le rappresentazioni delle risorse di dominio di *io@spot* tendono molto al *pattern collection*: una struttura del genere è facilmente descrivibile con il media type applicativo Collection+JSON [2], un'estensione del conosciutissimo standard JSON progettata appositamente per aggiungere semantiche al formato dei dati rappresentati e supportarne le relazioni ipermediali. È fortemente orientato alla gestione di collezioni di dati e a tal scopo supporta appositi template per la ricerca e l'aggiunta di elementi.

Collection+JSON nel dettaglio

Vengono esaminate ora le specifiche del media type Collection+JSON, formalmente definito con la notazione `application/vnd.collection+json`.

Le rappresentazioni delle risorse risultano essere in formato JSON a tutti gli effetti, sono tuttavia costrette ad avere una struttura minimale per essere validate in qualità di documenti Collection+JSON.

```
{
  "collection" : {
```

```
    "version" : "1.0",
    "href": "http://example.org/items"
  }
}
```

La rappresentazione minimale prevede un oggetto `collection` valido che contenga informazioni sulla versione di API corrispondente e l'URL tramite cui è accessibile. Per la trattazione delle specifiche complete del media type si fa riferimento ad una delle rappresentazioni descritte all'inizio della fase di progettazione, in modo da mostrare anche parte del lavoro svolto.

Tipicamente un documento Collection+JSON contiene i seguenti campi:

- un set di `links`: è una proprietà opzionale che contiene oggetti anonimi per collegare la rappresentazione corrente ad altre rappresentazioni e consentire dunque le transizioni di stato;

- una lista di `items`: rappresenta la lista di tutti i quanti record del documento Collection+JSON. Ciascun elemento interno alla lista `items` deve avere un campo `href` che indichi l'URI tramite cui poter ottenere, modificare o eliminare la rappresentazione del dato elemento con l'ausilio degli opportuni metodi HTTP;

- una collezione di `queries`: è anche questa una proprietà opzionale del documento contenente uno o più oggetti anonimi che rappresentano ciascuno un parametro per l'interrogazione dell'URI specificato dal campo `href`.

- un oggetto `template`: è un oggetto opzionale che non deve essere ripetuto più di una volta sola all'interno dello stesso documento. Definisce gli elementi da specificare per aggiungere o modificare una rappresentazione alla lista;

- un oggetto `error`: contiene informazioni aggiuntive su errori riportati dal server. È opzionale e deve essere ripetuto al massimo una volta all'interno dell'oggetto `collection`;
- una collezione `data`: può essere riportata innestata nei campi `items` o `template` e contiene oggetti anonimi che definiscono le proprietà di una rappresentazione.

Nel caso della rappresentazione di una lista di *WorktimeEntry* associata ad un utente autenticato si ha:

```
{
  "collection" : {
    "version" : "1.1",
    "href" : "http://io.spot.it/api/worktimeEntries/",
    "links" : [{
      "rel" : "home",
      "href" : "http://io.spot.it/api/users/550314dce3825d..."
    }, {
      "rel" : "organizations",
      "href" : "http://io.spot.it/api/organizations"
    }],
    "items" : [{
      "href" : "http://io.spot.it/api/worktimeEntries/5503...",
      "data" : [{
        "name" : "manual",
        "value" : "true",
        "prompt" : "Manual"
      }, {
```

```
      "name" : "performedAt",
      "value" : "2015-03-12T16:48:44.454Z",
      "prompt" : "Performed at"
    }, {
      "name" : "workTimeEntryType",
      "value" : "in",
      "prompt" : "Type"
    }],
    "links" : [{
      "rel" : "organization",
      "href" : "http://io.spot.it/api/organizations/550314..."
    }, {
      "rel" : "user",
      "href" : "http://io.spot.it/api/users/550314dce3825..."
    }]
  }, {
    "href" : "http://io.spot.it/api/worktimeEntries/55031...",
    "data" : [{
      "name" : "manual",
      "value" : "true",
      "prompt" : "Manual"
    }, {
      "name" : "performedAt",
      "value" : "2015-03-12T18:20:12.454Z",
      "prompt" : "Performed at"
    }, {
      "name" : "workTimeEntryType",
```

```
      "value" : "out",
      "prompt" : "Type"
    }],
    "links" : [{
      "rel" : "organization",
      "href" : "http://io.spot.it/api/organizations/550314dc..."
    }, {
      "rel" : "user",
      "href" : "http://io.spot.it/api/users/550314dce3825d1c..."
    }]
  }],
  "queries" : [{
    "rel" : "filter",
    "href" : "http://io.spot.it/api/worktimeEntries/",
    "prompt" : "Fill the filters",
    "data" : [{
      "name" : "type",
      "value" : "",
      "prompt" : "Type"
    }, {
      "name" : "from",
      "value" : "",
      "prompt" : "From"
    }, {
      "name" : "to",
      "value" : "",
      "prompt" : "To"
```

```
      }, {
        "name" : "page",
        "value" : "",
        "prompt" : "Page"
      }]
    }],
    "template" : {
      "data" : [{
        "name" : "manual",
        "value" : "",
        "prompt" : "Manual"
      }, {
        "name" : "performedAt",
        "value" : "",
        "prompt" : "Performed at"
      }, {
        "name" : "workTimeEntryType",
        "value" : "",
        "prompt" : "Type"
      }]
    }
  }
}
```

La rappresentazione in risposta alla richiesta di un particolare elemento della collezione è strutturata allo stesso modo, solamente che la lista `items` contiene un unico elemento. È buona norma per le API non ritornare alcun valore per i campi `queries` e `template` nel caso di una rappresentazione di un solo elemento,

specificando comunque il set di `links` che lo collegano alle altre risorse.

```
{
  "collection" :{
    "version" : "1.1",
    "href" : "http://io.spot.it/api/worktimeEntries/",
    "links" : [{
      "rel" : "home",
      "href" : "http://io.spot.it/api/users/550314dce3825d1..."
    }, {
      "rel" : "organizations",
      "href" : "http://io.spot.it/api/organizations"
    }],
    "items" : [{
      "href" : "http://io.spot.it/api/worktimeEntries/55031...",
      "data" : [{
        "name" : "manual",
        "value" : "true",
        "prompt" : "Manual"
      }, {
        "name" : "performedAt",
        "value" : "2015-03-12T16:48:44.454Z",
        "prompt" : "Performed at"
      }, {
        "name" : "workTimeEntryType",
        "value" : "in",
        "prompt" : "Type"
      }],
```

```
      "links" : [{
        "rel" : "organization",
        "href" : "http://io.spot.it/api/organizations/55031..."
      }, {
        "rel" : "user",
        "href" : "http://io.spot.it/api/users/550314dce..."
      }]
    }]
  }
}
```

In caso di errore, il server può ritornare un oggetto `error` come campo interno a `collection` che specifichi i dettagli dell'errore:

```
{
  "collection" :{
    "version" : "1.1",
    "href" : "http://io.spot.it/api/worktimeEntries/",
    "error" : {
      "title" : "Resource not found",
      "code" : "404",
      "message" : "The requested resource was not found."
    }
  }
}
```

Tutte le altre risposte del server, in seguito a richieste di eliminazione, modifica o creazione devono essere prive di un body e limitarsi a specificare lo status code ed eventuali header nei metadati del messaggio HTTP.

Per inviare al server la rappresentazione desiderata per una data risorsa, è necessario preparare un oggetto `template` con i campi valorizzati a dovere ed includerlo nel body della chiamata HTTP (POST in caso di creazione, PUT in caso di aggiornamento).

```
{
  "template" : {
    "data" : [{
      "name" : "manual",
      "value" : "false",
      "prompt" : "Manual"
    },{
      "name" : "performedAt",
      "value" : "2015-03-13T08:31:43.468Z",
      "prompt" : "Performed at"
    }, {
      "name" : "workTimeEntryType",
      "value" : "in",
      "prompt" : "Type"
    }]
  }
}
```

4.5.5 Scrivere un profilo per le semantiche del dominio

Se nel terzo step della progettazione non sono stati trovati sufficienti profili per la descrizione delle semantiche applicative delle rappresentazioni esposte dalle API è opportuno scrivere un nuovo profilo o estenderne uno pre-esistente. Nel caso delle API di *io@spot*, questo è tuttora un punto aperto: al momento

le semantiche applicative sono esplicitate nei commenti del codice poiché non è ancora chiaro l'obiettivo che si vuole raggiungere: si sta valutando se limitarsi alla redazione di documentazione HTML verbosa oppure spingersi alla creazione e pubblicazione di un nuovo profilo così da renderlo consultabile anche dalle macchine piuttosto che non solo da persone. La seconda opzione prevederebbe l'adozione di standard come ALPS o JSON-LD, ancora non troppo diffusi e limitati nel definire universalmente la semantica di un dato dominio applicativo.

4.5.6 Implementazione

Questa fase prevede lo sviluppo di due applicativi software strettamente legati fra loro: prima di tutto un server HTTP che esponga le web API progettate, caratterizzato da un buon livello di performance e scalabilità; in seguito un client HTTP che le consumi e dia luogo alle appropriate transizioni di stato in base alle rappresentazioni ricevute, mantenedosi responsivo ed intuitivo da usare.

È stata fatta un'analisi scrupolosa per la scelta delle tecnologie da adottare nell'implementazione sia dell'uno che dell'altro, senza trascurare tutta quanta la parte di persistenza dei dati. Si è optato per lo stack applicativo MEAN (*MongoDB, ExpressJS, AngularJS, Node.js*), diffusosi ultimamente nel campo dello sviluppo web per via di alcune caratteristiche peculiari:

- al giorno d'oggi è probabilmente l'unica soluzione a prevedere anche un framework client-side ben integrato con la parte server;
- sfrutta tecnologie indipendenti dalla piattaforma, facilmente usufruibili su qualsiasi sistema operativo e con requisiti minimi hardware molto bassi;
- può contare su framework giovani, open-source e con community incredibilmente attive a supporto;

- consente un approccio più immediato, soprattutto nel caso in cui gli sviluppatori siano più orientati alla programmazione front-end: tutti i framework dello stack in questione prevedono l'utilizzo di due sole tecnologie, JavaScript e JSON;

- i framework previsti, se ben utilizzati, promuovono il soddisfacimento di tutti i più rigorosi canoni di qualità previsti per le moderne applicazioni web.

Si analizzano ora le componenti dello stack applicativo MEAN, dettagliando nella dovuta maniera gli aspetti che più influiscono sull'implementazione delle web API.

MongoDB

Rappresenta l'alternativa più gettonata ai classici database MySql o SqlServer, con la differenza di essere un DBMS non relazionale orientato ai documenti: non è strutturato sulle tabelle relazionali dei DBMS tradizionali, ma fa affidamento su un'architettura basata sul salvataggio dei dati in qualità di documenti JSON, ciascuno dei quali contraddistinto da un proprio schema.

I vantaggi rispetto ad un database relazione sono diversi: gestisce in maniera semplice ed efficace grossi quantitativi di dati, strizzando l'occhio ai sistemi di big data sempre più diffusi; si dimostra flessibile grazie ad una struttura dei dati eterogenea e svincolata dalle "tipizzazioni" altrimenti imposte da un modello relazionale, così da evitare inutili occupazioni di spazio su disco; a patto di avere modelli di dati strutturati con criterio, l'esecuzione delle query è in media più basso rispetto ad interrogazioni su database relazionali. Solitamente la gran parte degli sviluppatori si ferma qui, le performance sono un fattore di scelta piuttosto ricorrente, invece prima di confermare MongoDB come base dati per *io@spot* se

ne sono valutati con attenzione anche i punti deboli: non c'è possibilità di effettuare il join tra documenti diversi come invece si è soliti fare con le tabelle dei database relazionali, bisogna ricorrere a strutture di documenti innestati gli uni negli altri con il rischio di complicare le cose e perdere il controllo; MongoDB prevede come operazioni atomiche solamente le operazioni su un documento singolo, preclude quindi la possibilità di effettuare transazioni; ha un consumo di RAM maggiore rispetto ai DBMS tradizionali perché deve mantenere in memoria tutte quante le chiavi che identificano le strutture dati eterogenee all'interno di collezioni; consente letture concorrenti, ma dà accesso esclusivo alle scritture che, tra l'altro, sono prioritarie rispetto a qualunque altra tipologia di operazione.

Quest'ultimo difetto potrebbe destare preoccupazioni, ma l'adozione di Node.js come piattaforma server asincrona e single-thread risolve a priori il problema; per il resto MongoDB risulta idoneo a gestire gli scenari previsti per il progetto in questione.

Per l'interazione con il database verrà usato Mongoose, un ODM (*Object Data Manager*) che incapsula un'istanza di MongoDB stesso: semplifica la modellazione degli schemi delle entità e permette di definire tipologie e vincoli ben precisi per ciascun dato; per di più mette a disposizione diverse API per l'esecuzione di operazioni CRUD e per arricchire di funzionalità le istanze stesse degli schemi.

Node.js

È una piattaforma di sviluppo lato server open-source e cross-platform, basata sull'engine V8 di Google Chrome. Il linguaggio di programmazione è JavaScript, tenute in considerazione le competenze prevalentemente front-end degli sviluppatori web. Node.js si è rivelato essere la soluzione più adatta alle esigenze di un sistema come *io@spot*, poiché altamente flessibile e performante per gestire in maniera ottimale tutti e tre gli applicativi end-user (web application, smart device,

smartphone app): infatti il sopracitato motore V8 è implementato sulla base di un modello event-driven che permette di accodare ed eseguire in modo asincrono un numero piuttosto elevato di operazioni (instaurazione di connessione, operazioni sul database, interazione con il disco) delegandole ad un sofisticato meccanismo di event-loop. Un modello asincrono del genere si dimostra vincente in scenari caratterizzati da un elevato traffico di rete poiché le richieste dei client vengono diligentemente accodate e gestite una per una, evitando gli sprechi di risorse dovute alle più comuni architetture multi-thread.

Come ogni piattaforma server che si rispetti, Node.js ha un sistema preinstallato di gestione delle dipendenze, npm (*node package manager*), che permette un utilizzo agevole e dinamico dei più svariati moduli applicativi messi a disposizione dalla community open-source: è uno strumento che va tuttavia utilizzato con cautela, perché un'eccessiva libertà nella pubblicazione di librerie e framework non sempre garantisce sulla qualità delle stesse.

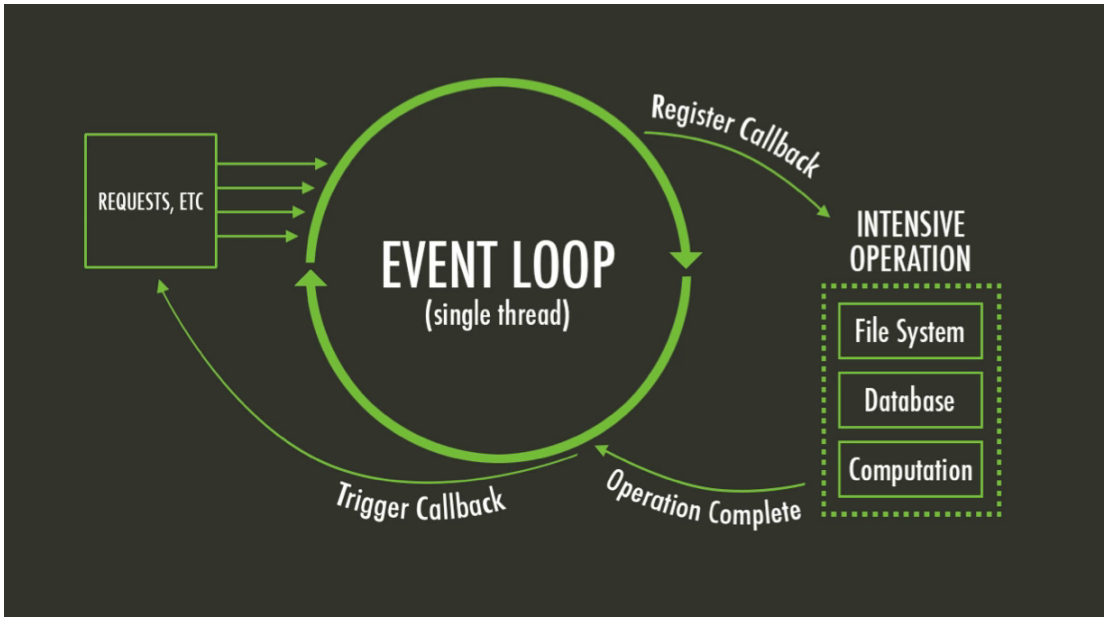

Figura 4.6: Uno schema del funzionamento asincrono event-driven di Node.js

ExpressJS

All'interno dello stack MEAN è il framework che più interessa ai fini dell'elaborato. ExpressJS è finalizzato alla creazione e configurazione di un web server su node.js, vantando caratteristiche che lo hanno reso in poco tempo un must:

- espone un middleware per la gestione del routing delle richieste HTTP, impeccabile dal punto di vista della configurabilità ed estendibilità;
- facilita il reperimento di contenuti statici dal server;
- tutte le sue componenti hanno logiche minimaliste che ne facilitano l'utilizzo e consentono comunque allo sviluppatore di beneficiare di tutti i pro della piattaforma node.js;
- prevede efficaci meccanismi di templating HTML lato server.

Grazie soprattutto al potente meccanismo di routing, Express si dimostra particolarmente adatto all'implementazione di web API sulla base dei vincoli e principi dello stile architetturale ReST: infatti permette allo sviluppatore di astrarre dal mero codice per concentrarsi sulla struttura delle API mappando URL semantici ai metodi dello standard HTTP. Di seguito l'esempio pratico di un modulo node.js atto all'inizializzazione delle API d'accesso all'entità `sample`.

```
var express = require('express');
var sampleController = require('./samplecontroller');

var router = express.Router();

// ritorna tutti i sample
router.get('/api/samples/', sampleController.getSamples);
```

```
// ritorna il sample identificato dal parametro id
router.get('/api/samples/:id', sampleController.getSample);

// crea un nuovo sample
router.post('/api/samples/', sampleController.createSample);

// aggiorna il sample identificato dal parametro id
router.put('/api/samples/:id', sampleController.updateSample);

// aggiorna parzialmente il sample identificato dal parametro id
router.patch('/api/samples/:id', sampleController.updateSample);

// cancella il sample identificato dal parametro id
router.delete('/api/samples/:id', sampleController.deleteSample);

module.exports = router;
```

Sulla linea di questo modulo d'esempio sono stati realizzati i moduli che formalizzazione l'API per ciascuna entità del dominio di *io@spot*. Per brevità non si riportano le implementazioni di ciascun modulo, ma ci si limita ad esporre alcuni tratti generici rilevanti.

L'**autenticazione** è stata definita già all'inizio del capitolo come la chiave d'accesso alle API per la gestione delle timbrature e delle assenze. Siccome la comunicazione HTTP deve essere stateless, si è scelto l'approccio "token based": il server non deve così mantenere il contesto d'interazione con un determinato client, una volta eseguita la login iniziale il client stesso disporrà di un

token che dovrà essere incluso nei metadati di ogni richiesta HTTP sotto forma di `header "Authorization"` e che il server decifrerà per avere informazioni sul contesto d'esecuzione relativo. Per completezza, il token non è altro che una stringa testuale generata secondo un meccanismo di crittografia a chiave simmetrica, chiave che per ovvie ragioni solamente il server conosce. Nella pratica il meccanismo di autenticazione e generazione del token sono gestiti dal modulo *passport.js* che, con l'ausilio di avanzate tecniche di hashing per la cifratura dei dati sensibili, ricerca a database per email l'utente e confronta la password corrispondente con quella inviata dal client. ExpressJS consente di utilizzare funzioni di middleware per la validazione delle richieste HTTP verso le API, evitandone l'accesso ad utenti non autorizzati o addirittura non autenticati:

```
// ritorna tutti i sample solamente se il middleware di
// autenticazione riconosce come valida la chiamata HTTP.
router.get('/api/samples/', authMiddleware.isAuthenticated,
                            sampleController.getSamples);
// crea un nuovo sample solamente se
// riconosce che l'utente è amministratore.
router.post('/api/samples/', authMiddleware.isAdministrator,
                            sampleController.createSample);
```

Gli **URL**, per quanto siano stati definiti non fondamentali nel Capitolo 3 per considerare una API ReSTful, sono un elemento piuttosto importante poiché se ben strutturati possono offrire una descrizione semantica di se stessi che li rende semplici ed intuitivi agli occhi di un qualsiasi utilizzatore.

Gli **status code** delle risposte HTTP vengono spesso mal utilizzati o, addirittura, snobbati. Sono invece tra le informazioni più importanti contenute nei

meta dati di un messaggio HTTP per via del fatto che comunicano al client l'esito dell'elaborazione di una richiesta da parte del server in modo che il client possa reagire correttamente senza bisogno di interpretare ulteriori parametri (che magari sono stati inclusi nel body della risposta). Di seguito una lista degli status code che si è ritenuto necessario gestire per le API di *io@spot*:

- `200 : OK` - indica il semplice successo di una richiesta GET, PUT, PATCH e DELETE;
- `201 : Created` - indica che la richiesta POST è risultata nella creazione di una nuova risorsa. Per correttezza è bene accompagnare questo status code con un header `Location` contenente l'URL della risorsa creata;
- `204 : No Content` - è lo status code relativo ad una richiesta che ha avuto successo ma che non ritorna dati nel body della risposta, come ad esempio potrebbe essere una DELETE;
- `400 : Bad Request` - indica che la richiesta è malformata, dunque non è stata correttamente interpretata dal server;
- `401 : Unauthorized` - rappresenta un errore di autenticazione, come se non fosse stata eseguita la login;
- `403 : Forbidden` - viene ritornato quando l'utente non ha i permessi necessari per accedere alla data risorsa;
- `404 : Not Found` - indica che la risorsa richiesta non esiste;
- `415 : Unsupported Media Type` - sta a significare che nella richiesta è stato specificato un media type non supportato dalle API, nel caso di *io@spot* un qualunque formato differente da `application/vnd.collection+json`.

AngularJS

È il framework client-side più utilizzato al mondo per lo sviluppo di applicazioni web: è stato sviluppato come side-project da Google per fornire ai programmatori tutti gli strumenti necessari alla creazione di una applicazione single page, ovvero costituita da un'unica pagina HTML con caricamento dinamico dei contenuti via JavaScript. Approfondendo i contetti principali di AngularJS si può parlare di:

- **data-binding bidirezionale**: evita la manipolazione diretta del DOM, l'HTML che viene visualizzato è sostanzialmente una proiezione del modello dei dati associato alla vista. Il collegamento che si instaura tra il modello dei dati e il DOM è bidirezionale, ovvero aggiornando uno le modifiche si ripercuotono sullo stato attuale dell'altro;

- **templating**: lo standard HTML viene esteso in maniera da contenere istruzioni su come il modello dei dati debba essere mappato sulla view. A differenza di molti altri framework client, AngularJS non interpreta i template come delle stringhe da iniettare nella pagina, bensì li tratta come veri e propri elementi del DOM;

- **il pattern MVC**: AngularJS per definizione è un framework MVC (*Model-View-Controller*) che, dunque, consente di separare la logica di presentazione dei dati dalla logica di business. Il model rappresenta il semplice contenuto informativo, il controller è il responsabile del caricamento dinamico dei contenuti e della gestione dello stato dell'applicazione, mentre la view è normale HTML che viene esteso dai binding di Angular così da mantenere corrispondenva instaurare un'interazione bidirezionale tra i dati e la loro rappresentazione;

- **dependency injection**: consente di sviluppare applicazioni flessibili in maniera semplice e veloce senza sprecare risorse superflue tramite il caricamento dinamico delle sole dipendenze a moduli utili;
- **direttive**: rappresentano la parte più complicata dell'intero framework ma permettono un elevato grado di estensibilità dell'HTML con componenti customizzati per centralizzare logiche ricorrenti che il DOM da solo non è in grado di applicare;
- **unit testing**: viene riservata una grande importanza alla fase di testing dell'applicazione, almeno quanto a quella di sviluppo, poiché la robustezza di un software contribuisce corposamente al suo successo.

4.5.7 Pubblicazione

Lo stato di sviluppo dell'intero progetto, e non solo delle web API, è fermo a questo punto: è ancora da delineare la strada che si intende intraprendere, ovvero se limitare l'uso del sistema a SPOT Software o provare a trasformarlo in un prodotto da poter vendere o comunque installare in altre aziende. Sicuramente le web API non verrebbero pubblicate prima di aver redatto e reso disponibile un profilo con la descrizione dei significati delle relazioni e dei descrittori semantici delle rappresentazioni del dominio.

Capitolo 5

Conclusioni

L'approccio ReSTful che si è scelto di utilizzare durante il processo di progettazione ha consentito l'approfondimento di concetti e tecnologie molto interessanti, fino a prima dati erroneamente per scontati o del tutto ignorati. Decisiva è stata la consultazione di documenti quali la tesi di dottorato di R. T. Fielding e il libro "ReSTful Web APIs" scritto da Leonard Richardson: hanno portato ad aprire gli occhi su tematiche cruciali per il mondo dello sviluppo web, proponendo talvolta valide soluzioni e altre stimolanti spunti di riflessione per il futuro.

I problemi più impellenti legati alle web API ReSTful sono l'inconsapelezza e talvolta la disinformazione che la gran parte degli sviluppatori tende a dimostrare nei confronti del concetto di ipermedia: sembrano quasi non riuscire a distaccarsi dai dettami della scuola SOA e dalle pesanti tecnologie basate su protocollo SOAP che non fanno altro che precludere la scalabilità e l'estendibilità dei sistemi. Il risultato è ovvio: al giorno d'oggi la quasi totalità delle API una volta pubblicata non può cambiare senza compromettere il funzionamento dei client che le utilizzano. E le soluzioni ReSTful dovrebbero evitare tutto ciò, riconducendo il funzionamento delle API a quello del Web stesso: esso è formato da milioni di siti, pubblicati da migliaia di implementazioni server diverse, soggetti a periodiche

revisioni grafiche e riprogettazioni dei contenuti, acceduti da utenti che utilizzano svariate tipologie di client su piattaforme hardware differenti. Sicuramente progettare secondo il principio HATEOAS e gli altri vincoli architetturali ReST richiede più tempo, soprattutto nel caso di API su scala ridotta, ma garantisce il raggiungimento di alti livelli di estendibilità e flessibilità in un'ottica di continuità e maturazione nel tempo.

L'altra grossa sfida a cui i più audaci hanno già preso parte è quella del divario semantico fra le rappresentazioni scambiate dagli innumerevoli applicativi che popolano il web. Quella di mettere a fattore comune le semantiche di domini uguali o, perlomeno, simili appare come un'impresa tutt'altro che semplice e attualmente sono pochi gli standard a disposizione degli sviluppatori per districarsi nello *zoo ipermediale* [13, Chapter 10] che è diventato il World Wide Web; tant'è che proprio per questo movito nella progettazione delle API di *io@spot* si è evitata la stesura di un profilo: infatti il tempo richiesto da questa attività è stato stimato superiore a quello disponibile, in uno scenario di incerta distribuzione al di fuori del dominio di SPOT Software. In ogni caso, aver progettato e sviluppato a dovere tutti quanti gli altri aspetti risulterà un grande vantaggio nell'integrazione di un profilo semantico e nell'adattarsi ai futuri possibili cambiamenti del sistema.

5.1 io@spot nel futuro

Le web API di *io@spot*, nonostante presentino ancora qualche punto aperto, sono state sviluppate nella loro interezza e soddisfano i requisiti inizialmente stabiliti: rendono attualmente possibile l'interoperabilità dei tre applicativi client nel processo di marcature degli accessi e gestione delle assenze all'interno di una data organizzazione. Come ripetuto più volte si è incerti su quello che sarà il futuro del progetto. L'unica cosa sicura è che a breve verrà fisicamente montato in azienda

così da poter essere utilizzato al posto del vecchio sistema di marcatura e consolidato. In un'ottica di pubblicazione e commercializzazione due sono le cose su cui bisognerebbe lavorare, ossia risolvere alcune problematiche minori di integrazione e sicurezza dovute più che altro alla natura prototipale del progetto ed effettuare un'attenta attività di *load testing* per verificare la scalabilità del sistema, in particolare delle API su scala Internet: il punto più critico potrebbe rivelarsi essere l'accesso al database e potrebbe richiedere una revisione delle scelte progettuali nella gestione delle relazioni tra i diversi *schema* della entità su MongoDB.

Elenco delle figure

2.1 Derivazione dello stile ReST dai vincoli 11

3.1 Software ReSTful, in sintesi. 41

4.1 Il dominio applicativo. 56
4.2 Uno schema dei casi d'uso legati alla web application. 60
4.3 Architettura logica del server. 62
4.4 Diagramma di sequenza dell'interazione tra gli strati logici del server in uno scenario tipo di utilizzo della web application. 63
4.5 Il diagramma degli stati applicativi. 68
4.6 Uno schema del funzionamento asincrono event-driven di Node.js 81

Bibliografia

[1] Web 2.0 conference, 2009.

[2] Mike Amundsen. Collection+json media type.

[3] Tim Berners-Lee. Uri, Maggio 1969. RFC 2396.

[4] Andrea Corzani. *Progettazione e realizzazione di uno smart distributed system per il controllo degli accessi in ambito aziendale*. PhD thesis, Alma Mater Studiorun - Università di Bologna.

[5] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Doctor of Philosophy in Information and Computer Science, 2000.

[6] Internet Engineering Task Force (IETF). Http patch, Ottobre 2010. RFC 5988.

[7] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley Publishing, 1992.

[8] Sam Ruby Leonard Richardson, Mike Amundsen. *RESTful Web APIs*. O'Reilly Media, September 2013.

[9] Lorenzo Mucchi. Corso tecnologie web - world wide web.

[10] Inc O'Reilly Media. Web 2.0 conference, 2004.

[11] Roger S. Pressman. *Software Engineering: a practitioner's approach*. McGraw-Hill, 1982.

[12] Alan Quayle. Web apis are transforming enterprise communications and collaboration.

[13] L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs*. O'Reilly Media, September 2013.

[14] J. Snell and L. Dusseult. Http patch, Marzo 2010. RFC 5789.

[15] W3C. Html5 recommendation.

[16] W3C. Web services glossary.

[17] Wikipedia. Representational state transfer.

[18] Xiwei Xu, Liming Zhu, Yan Liu, and Mark Staples. Resource-oriented architecture for business processes. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 395–402, Dec 2008.

Ringraziamenti

È stata dura, molto dura. Eppure sono qui a scrivere queste ultime righe di tesi, provato dalla stanchezza e dalla tensione degli ultimi mesi. Concludere gli studi lavorando a tempo pieno non è stata una cosa per niente semplice e non ce l'avrei mai fatta da solo. Ringrazio quindi Chiara, per l'affetto e l'incrollabile pazienza dimostrati nonostante i miei sbalzi d'umore e i momenti di crisi, è stata un pilastro fondamentale; con le lacrime agli occhi ringrazio i miei genitori e tutti quanti i parenti che hanno continuato ad incitarmi fino all'ultimo senza mai perdere la speranza (a differenza mia); ringrazio separatamente mio fratello per il semplice fatto che so gli darà fastidio, nonostante l'adolescenza ha saputo dimostrarsi affettuoso e premuroso nei momenti di bisogno; non potrò mai ringraziare abbastanza i miei datori di lavoro, Pietro e Maurizio, per aver creduto in me quando a malapena sapevo cosa fosse Javascript e per avermi riacceso la passione; ringrazio tutta Spot Software perchè, nonostante il periodo, ha trovato la forza di sostenermi ed aiutarmi in quest'ultima tappa universitaria; un grazie particolare ad Andrea per i velati (ma anche no) incitamenti e il costante supporto tecnico, senza cui probabilmente non avrei concluso entro questa sessione di laurea; ringrazio gli amici e tutti coloro i quali mi hanno saputo voler bene e mi sono stati vicini in questi anni. Ultimo, ma non certo per importanza, il professor Andrea Omicini che mi ha saputo indirizzare sulla retta via e mi ha spronato a non mollare nonostante i tempi stretti.