



Università degli Studi di Camerino

SCUOLA DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica (Classe L-31)

**Analisi e sviluppo di un sistema software
per la domotica d'ufficio: il backend di
“Proximity System”**

Laureando
Saverio Tosi

Matricola 090311

Relatore
Dott. Francesco De Angelis

Correlatore
Dott. Francesco Strazzullo

A.A. 2015/2016

Indice

1	Introduzione	11
1.1	Analisi e progettazione	12
1.1.1	Analisi dei requisiti	13
1.2	Miglioramento dell'esperienza utente	14
2	IoT	15
2.1	Il futuro delle smart home	15
2.2	Raspberry	16
2.2.1	Perchè Raspberry	17
3	Beacon	19
3.1	iBeacon	19
3.1.1	Privacy and Location	20
3.1.2	Precisione degli iBeacon	21
3.1.3	Monitoraggio regioni	21
3.1.4	Ranging	22
3.1.5	Physical limitations	22
3.1.6	Calibrazione	22
3.1.7	UUID	23
3.2	BlueUp	24
3.2.1	BlueBeacon	24
3.2.2	Caratteristiche	24
3.2.3	Trasmissione	25
3.2.4	Batteria	25
3.3	Utilizzo dei Beacon in Proximity System	26
4	Scelta tecnologica	27
4.1	JSON	27
4.1.1	Gli array literal	27
4.1.2	Gli object literal	28
4.1.3	La sintassi JSON	28
4.2	REST	29
4.3	Documentazione	31
4.4	NodeJS	33

4.5	Express	37
4.6	Mongoose	38
5	Websocket	41
5.1	Panoramica della applicazione real-time e HTTP	41
5.2	Introduzione alle WebSocket	42
5.2.1	L’handshake WebSocket	43
5.2.2	L’interfaccia	44
5.2.3	Aumento delle prestazioni	45
5.2.4	Supporto dei browser	48
5.3	Socket.io	48
5.3.1	Backend	49
5.3.2	Frontend	49
6	Conclusioni	51
A	Il codice	53

Elenco dei codici

4.1	User collection	30
4.2	Beacon collection	30
4.3	Device collection	31
4.4	GPIO collection	31
4.5	package.json	33
4.6	operazioni asincrone	33
4.7	operazioni sincrone	34
4.8	test/config/test_config.js	34
4.9	test/config/setup_tests.js - connectDB	34
4.10	test/config/setup_tests.js - dropUserCollection	35
4.11	test/config/setup_tests.js - closeDB	35
4.12	test/config/setup_tests.js - async	35
4.13	test/user/create_accounts_error_spec.js	35
4.14	test/user/create_accounts_error_spec.js	35
4.15	test/user/create_accounts_spec.js	36
4.16	test/user/create_accounts_spec.js	36
4.17	test/user/create_accounts_spec.js	36
4.18	config/db.js	36
4.19	server.js	37
4.20	server.js - middleware	37
4.21	server.js - listen	38
4.22	userSchema	38
4.23	get user	39
5.1	client handshake	43
5.2	server handshake	44
5.3	interfaccia WebSocket	44
5.4	JSON dati macchina	45
5.5	header client	45
5.6	header server	45
5.7	backend chat socket.io	49
5.8	service mySocket	49
5.9	controller mySocket	50

Elenco delle figure

1.1	Il nostro Trello, nella fase iniziale del progetto	13
2.1	Raspberry PI 1 Model B+	17
3.1	Autorizzazioni iOS	21
3.2	Calibrazione	23
4.1	Swagger editor	32
4.2	Swagger ui	32
5.1	Esempio di long-polling	42
5.2	Frame WebSocket	46
5.3	Traffico dati non necessario	47
5.4	Latenza	47

Elenco delle tabelle

3.1	iBeacon advertisement	20
3.2	Esempio utilizzo UUID, major e minor	20
3.3	Proximity state	22
3.4	Probabilità collisione UUID	24
3.5	Distanza massima di trasmissione	26
3.6	Durata batteria BlueBeacon Mini in mesi	26
5.1	Supporto dei browser	48
5.2	Supporto dei browser Mobile	48

1. Introduzione

Questa tesi rappresenta la terza ed ultima parte dello stage++¹, realizzato in collaborazione con extrategy, il Prof. Francesco De Angelis e con il collega e amico Marco D'Argenio. In questo stage ci è stato chiesto di realizzare un'applicazione per la domotica pensata per l'ufficio che prevedesse l'uso dei beacon, ovvero dei piccoli trasmettitori bluetooth che permettono la geolocalizzazione interna negli edifici. Proprio per questo motivo, il nome che abbiamo deciso di dare al nostro prodotto è: "Proximity System".

Gli obiettivi che dovevamo raggiungere con questo progetto erano numerosi, ragion per cui, io e Marco, ci siamo divisi il carico in maniera bilanciata e logica. Io ho trattato tutti gli argomenti inerenti al backend e ai beacon. Marco, invece, ha trattato gli argomenti relativi al frontend dell'applicazione e ha studiato il BMC² del prodotto per vedere la fattibilità dello stesso da un punto di vista economico. Questi argomenti non verranno trattati qui, ma nella tesi di Marco: "Analisi e sviluppo di un sistema software per la domotica d'ufficio: il frontend di Proximity System".

In sintesi, il progetto prevede tre parti:

1. Il backend in esecuzione su Raspberry PI
2. Una web app per l'amministrazione del sistema
3. Un'app mobile per gestire i dispositivi

Tutte e tre le componenti hanno una cosa in comune, sono tutte scritte in Javascript.

In questa tesi vedremo in dettaglio il primo punto e in generale come utilizzare lo stack MEAN per lo sviluppo di applicazioni web rispetto al tradizionale stack LAMP e come le WebSocket possano aumentare drasticamente le performance di un'applicazione real-time.

Il codice completo del progetto è disponibile nei due repository

1. www.github.com/e-xtrategy/unicam-beacon-server
2. www.github.com/e-xtrategy/unicam-ionic-beacon-app

Il primo contiene sia il backend con le API REST che la web app in Angular. Il secondo contiene l'App Ionic compatibile con Android, iOS e, parzialmente (almeno per ora), Ubuntu Touch.

¹Lo stage++ è un progetto unico, formato dall'unione di tre esami: project work, stage e tesi.

²Business Model Canvas

1.1 Analisi e progettazione

La fase di analisi e di progettazione non si è sviluppata in maniera tradizionale, in quanto extrategy basa lo sviluppo di software sulle metodologie agili, più adatte ad ambienti dinamici come lo è il web. Fra le pratiche promosse da questi metodi ci sono la formazione di team di sviluppo piccoli, cross-funzionali e auto-organizzati, lo sviluppo iterativo e incrementale, la pianificazione adattiva, e il coinvolgimento diretto e continuo del cliente nel processo di sviluppo.

Per essere precisi, abbiamo utilizzato un mix tra Scrum e Kanban. Con Scrum[[Danb](#)], il prodotto viene realizzato tramite una serie di iterazioni a intervalli regolari chiamate sprint che danno al team un framework per consegnare al cliente del codice con una cadenza costante. L'idea è quella che degli sprint frequenti rinforzano l'importanza di fare delle giuste stime e permettono di ricevere feedback in tempo reale sul lavoro svolto. Ogni sprint è caratterizzato da questi quattro riti:

1. **Pianificazione:** un incontro iniziale in cui definire cosa fare con lo sprint corrente.
2. **Scrum giornaliero:** un mini incontro giornaliero della durata di una decina di minuti per sincronizzare il team.
3. **Demo:** un incontro per illustrare il lavoro svolto dal team.
4. **Retrospettiva:** una revisione di ciò che si è stato fatto per farlo meglio la volta successiva.

Kanban[[Dana](#)] ci è d'aiuto durante gli sprint per tenere sott'occhio tramite una board l'avanzamento del nostro progetto. Con questa metodologia il lavoro dell'intero team gira intorno alla kanban board, uno strumento utilizzato per visualizzare e ottimizzare il flusso del lavoro per tutto il team. Mentre delle lavagne o intere pareti sono popolari per molti team, le board virtuali sono una caratteristica cruciale per lo sviluppo di software con le metodologie agile per la loro tranciabilità, facilità di collaborazione e accessibilità da diverse località.

Al di là che la board sia fisica o digitale, la sua funzione è quella di assicurare che il lavoro del team sia visualizzato, che il flusso sia standardizzato e che ogni imprevisto sia immediatamente visualizzato e risolto. una kanban board di base ha tre step: To Do, In Progress e Done. È ovvio che non c'è nessun vincolo sulla gestione del flusso e esso dipende da diversi fattori. La metodologia kanban è basata sulla trasparenza del lavoro svolto e sulla comunicazione, perciò la kanban board dovrebbe essere vista come unica fonte di verità per il team.

Questa impostazione crea l'esigenza di una piattaforma in grado di tenere traccia dello sviluppo del software e delle user-stories da implementare, capace anche di coordinare gli sviluppatori dedicati a quel caso. Nonostante kanbanize probabilmente è una delle soluzioni online più adatte ad implementare la metodologia Kanban, abbiamo scelto Trello perché il nostro piccolo team non necessita di applicare alla lettera Kanban con tutte le procedure consigliate per la gestione del flusso, e una semplice todo list era più che sufficiente.

Trello[[Tre](#)] è un gestore di progetti basato sul web, originariamente realizzato da Fog Creek Software nel 2011. In questo caso il progetto è rappresentato da schede, che contengono liste corrispondenti ad elenchi di attività. Le liste sono formate da card, che rappresentano le singole attività. Le card sono tenute a passare da una lista all'altra,

tramite la funzione drag-and-drop; per via del generico flusso di una card, essa dovrebbe passare dalla lista delle cose da fare a quelle fatte, al momento del passaggio dall'idea alla realizzazione dell'attività in questione. Nello specifico le card rappresentano le user-stories, funzionalità utile al raggiungimento di un obiettivo di business, una descrizione appositamente non dettagliata che il software deve avere e che alimenti la discussione tra il cliente/utente e lo sviluppatore. Più programmatori possono essere assegnati alle card, ed insieme alle relative schede, possono essere raggruppati in organizzazioni differenti. Ogni card può accettare commenti, allegati, voti, date di scadenza e liste di controllo. La suddivisione di un progetto in schede/liste, a loro volta suddivise in card formano una gerarchia di dati su misura che facilita la gestione efficace dei progetti stessi, nonché delle attività dell'intera organizzazione.

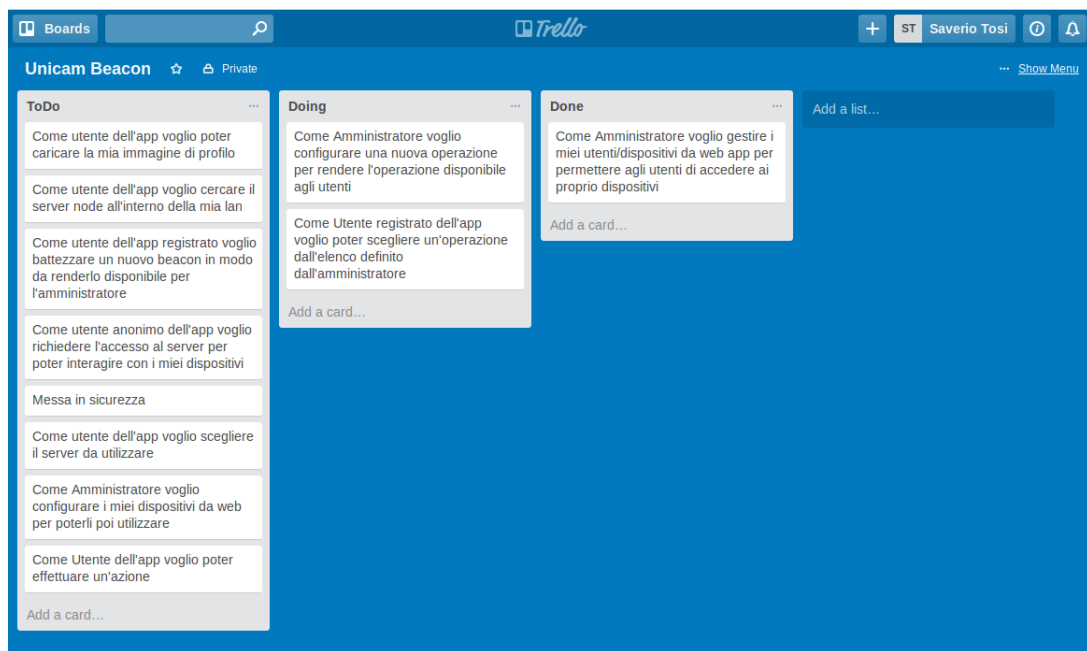


Figura 1.1: Il nostro Trello, nella fase iniziale del progetto

1.1.1 Analisi dei requisiti

Si vuole realizzare un sistema per la domotica per d'ufficio, che oltre ad essere comandato con un'app, utilizzi i beacon per eseguire dei comandi automatizzati.

Il backend deve essere caricato su un RaspberryPI 2 e deve fornire i servizi REST sia per la webapp che per l'app mobile. Il backend, oltre a fornire le API REST, deve essere in grado di comunicare con il mondo esterno: sia con input che con degli output. Per il prototipo, gli input saranno simulati con l'utilizzo di bottoni, nel prodotto finito essi saranno sostituiti con degli interruttori e dei sensori. Gli output devono pilotare dei relè monostabili che permetteranno di comandare l'accensione/spegnimenti di luci, l'apertura di porte e di cancelli.

Il pannello di amministrazione è rappresentato dalla webapp a cui potrà accedere soltanto l'admin. Essa permette all'admin di gestire gli utenti dell'app dandogli e togliendogli privilegi, creare nuovi dispositivi da associare ai vari GPIO del RaspberryPI e di configurare il modo in cui gli utenti possono interagire con i dispositivi: in ma-

niera manuale, automatica(tramite beacon) o se per poter controllare il dispositivo è necessario trovarsi nelle vicinanze di esso.

L'ultima componente è l'app mobile. Chi scaricherà l'app dovrà per prima cosa impostare l'indirizzo del server, se non si conosce l'indirizzo deve essere possibile fare una scansione della rete locale per trovare i backend attivi nella propria LAN. Una volta che si è connessi al server si può proseguire con il login/registrazione. La registrazione non abilita l'utente all'utilizzo di Proximity System, l'utente non potrà far nulla finché l'admin non gli dà i permessi per utilizzare l'app. Con l'app un utente ha due funzionalità: la prima è poter cercare nuovi beacon da segnalare all'amministratore; la seconda è interagire con i propri dispositivi.

1.2 Miglioramento dell'esperienza utente

Dopo una prima stesura del codice e la realizzazione di un primo prototipo, abbiamo deciso migliorare l'esperienza utente dando la possibilità al server di inviare notifiche push ai client. Per fare ciò, abbiamo utilizzato le WebSocket, una delle funzionalità più interessanti introdotta con l'HTML5.

Nel Capitolo 5 illustreremo come sia possibile migliorare le prestazioni di una generica WebApp che fornisce dei dati in tempo reale, per poi dimostrare come sia semplice utilizzare tale specifica, soprattutto se si utilizzano delle librerie ad hoc come Socket.io.

2. IoT

In questo capitolo andremo a vedere per prima cosa cos'è l'internet delle cose, per poi descrivere la board che abbiamo utilizzato per realizzare il nostro progetto.

2.1 Il futuro delle smart home

Tutti noi abbiamo sentito parlare, almeno una volta, dell'internet delle cose, ma in pochi sanno dare una definizione precisa di che cosa sia l'IoT. Questo perché una definizione rigorosa ad oggi non esiste. Quello che si intende per Internet of Things è l'estensione della rete ad un numero sempre crescente di dispositivi, o meglio oggetti. L'insieme di questi: termostati, robot, droni, smart tv, lavastoviglie e molti altri elettrodomestici; rappresenta una solida base per la costruzione di vere e proprie case intelligenti.

L'IoT sta prendendo, giorno dopo giorno, una fetta di mercato sempre più ampia, evolvendosi in fretta e senza seguire standard e protocolli ben specifici. Marteen Ector¹ ha scritto un articolo[Ect16] in cui evidenzia dei possibili problemi relativi al futuro di questi oggetti intelligenti. In particolare si chiede se ci troviamo in una di queste ere:

- Internet of Useless Things era
- Internet of Isolated Things era
- Internet of Insecure Things era
- Internet of Smart Cheap Things era
- Internet of App-Enabled Things era

Andiamo ad analizzare ogni singola era singolarmente. Molti di questi oggetti sono *inutili*, o quasi. Si pensi ad una semplice lampadina, è bello poterla pilotare senza doversi muovere dal divano, ma il problema che risolve è banale.

Non tutti questi smart device parlano la stessa lingua. Prodotti di costruttori diversi usano protocolli diversi *isolandosi* l'un l'altro.

Ci capita spesso di vedere falle di sicurezza che permettano a dei semplici "smanettoni" di accedere senza troppi problemi al microfono e alla webcam della nostra smart tv, anche delle marche più famose. Siamo sicuri che vogliamo una casa *insicura*?

Il quarto punto rappresenta in realtà un vantaggio dato dal costo delle componenti elettroniche che si sta abbassando drasticamente. Grazie a ciò è possibile acquistare dei mini-pc per pochi euro. Una generica casa con l'ausilio di questi dispositivi e di qualche sensore può raccogliere più informazioni di quanti ne raccoglieva google 10 anni fa.

¹Marteen Ectors è uno dei massimi esponenti dell'IoT di Canonical

Infine, per ogni oggetto che compriamo, troviamo la relativa app sullo store che permette di utilizzare il dispositivo che altrimenti sarebbe inutilizzabile.

Prendendo atto di queste considerazioni e del fatto che il nostro sistema deve adattarsi più all'uso in ufficio che in una casa, abbiamo deciso che il nostro sistema deve avere le seguenti caratteristiche:

1. **Open source:** per dare la possibilità a chiunque di migliorarlo e di integrarlo con i propri dispositivi;
2. **Centralizzato:** non vogliamo avere tanti oggetti intelligenti, ma uno centrale che ne controlli tanti "stupidi". Questa soluzione non è adatta per spazi ampi come una casa in cui comporterebbe un alto costo per il cablaggio, ma è adatta a singole stanze come un ufficio.

2.2 Raspberry

Raspberry è un piccolo calcolatore elettronico dalle dimensioni di una carta di credito sviluppato nel Regno Unito dalla Raspberry Pi Foundation. Lo scopo di questa fondazione è quello di sviluppare un dispositivo economico, che ha la capacità di stimolare l'insegnamento dell'informatica e della programmazione nelle scuole. La scheda è stata progettata per ospitare sistemi operativi basati su un kernel linux, tra i più usati troviamo:

- Raspbian
- Ubuntu mate
- Snappy Ubuntu Core
- Windows 10 IoT Core

Per quanto riguarda i componenti hardware, esso ha la stessa struttura dei moderni Personal Computer. Raspberry Pi rispetta perfettamente il modello di Von Neumann, nella quale si individuano quattro componenti fondamentali: l'unità di elaborazione e controllo, la memoria, il sistema di ingresso/uscita, il sistema di interconnessione. Inizialmente esistevano tre modelli di Raspberry Pi:

1. Model A, consigliato per i progetti embedded;
2. Model B, consigliato per l'utilizzo nelle scuole;
3. Model B+, potenziamento del Model B.

Successivamente sono uscite nuove versioni del model B, caratterizzate da più RAM e una CPU migliore, che prendono il nome di Raspberry PI 2 e 3. Per il nostro progetto abbiamo scelto di utilizzare il Raspberry PI 2 in quanto è il dispositivo che apre a maggiori possibilità. Il Raspberry Pi 2, infatti, è dotato di 4 porte USB, un connettore femmina RJ45 10/100BaseT, lettore MicroSD, connettore display DSI, presa HDMI, connettore fotocamera CSI, un jack da 3,5mm e il cip Broadcom BCM2836 con processore quad-core Cortex-A7 affiancato da 1024 MB di memoria RAM.

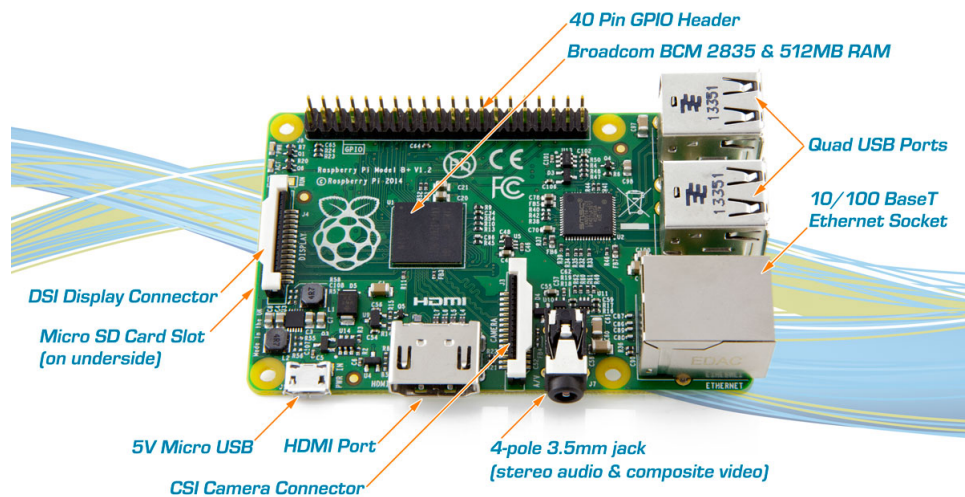


Figura 2.1: Raspberry PI 1 Model B+

2.2.1 Perchè Raspberry

Come già descritto nel capitolo 1, il prodotto che dobbiamo realizzare non solo deve essere in grado di far girare un webserver, ma deve anche essere in grado di comunicare con il mondo esterno. Raspberry Pi si è rivelato perfetto per questo scopo. Essendo un dispositivo low cost che però ha sia un processore discreto che la possibilità di comunicare con relè e sensori, in quanto è dotato di ben 40 GPIO.

3. Beacon

Un beacon è una piccola antenna trasmettitore bluetooth, che lavora alla frequenza standard di 2,4 ghz (la stessa del Wi-fi, per intendersi). Un beacon può inviare soltanto una piccola stringa di codice, grande qualche manciata di byte, che funge da identificatore del beacon stesso.

Nel corso degli anni le tipologie di beacon sono aumentate col trascorrere del tempo. Ad oggi, le tre principali tipologie di beacon sono:

1. iBeacon - Apple
2. EddyStone - Google
3. AltBeacon - Open

Il prodotto della Apple è stato il primo ad entrare nel mercato e per questo è il più diffuso in commercio. Degli iBeacon verrà fatta una panoramica più ampia nel prossimo paragrafo, per il momento diciamo soltanto che prevede un solo tipo di frame. EddyStone è un prodotto Google che funziona sia con Android che con iOS. La principale differenza rispetto al prodotto Apple, è la possibilità di scegliere tra due diversi tipi di frame da inviare in broadcast. Per finire, AltBeacon nasce con l'esigenza di creare un protocollo per i beacon che sia aperto e che faccia dell'interoperabilità il suo punto di forza.

3.1 iBeacon

iBeacon technology[App14], introdotta con iOS7, è una tecnologia che permette di migliorare la localizzazione per app. Utilizzando il Bluetooth Low Energy (BLE), un device che supporta gli iBeacon può essere utilizzato per stabilire se esso è dentro la regione di un determinato oggetto. Quindi, Permette ad un dispositivo iOS di determinare quando quest'ultimo entra o esce da una regione, con una stima della prossimità dal beacon.

Se si è interessati ad utilizzare gli ibeacon, si rientra in almeno una delle seguenti categorie di persone:

1. **App Developers**

Se si vuole migliorare la geo-localizzazione di un'app, si può utilizzare il Core Location APIs in iOS per ricevere notifiche quando un dispositivo entra ed esce dalla regione di un beacon e per stimare la prossimità dal beacon stesso. Tutto quello di cui si ha bisogno è incluso in iOS SDK.

2. **Persone che vogliono utilizzare un dispositivo con la tecnologia iBeacon**

Se si è interessati ad utilizzare il logo di iBeacon, ma non si è un produttore

di iBeacon, si deve ottenere una licenza del logo prima di utilizzarlo. <https://developer.apple.com/ibeacon/>

3. Persone che vogliono creare dispositivi con la tecnologia iBeacon

Se si è interessati a costruire dispositivi con iBeacon technology, si deve ottenere una licenza da Apple prima di realizzarli. Con la licenza si ha l'accesso a specifiche tecniche ed il permesso ad utilizzare il logo iBeacon.

Passiamo ora a descrivere il frame. Un iBeacon advertisement invia le seguenti informazioni via BLE:

Campo	Dimensione	Descrizione
UUID	16 bytes	Lo sviluppatore dovrebbe definirne uno per la propria app e utilizzo
Major	2 bytes	Indica uno specifico caso d'uso dell'iBeacon
Minor	2 bytes	Suddivide ulteriormente la regione

Tabella 3.1: iBeacon advertisement

UUID, major e minor identificano unicamente il beacon. Un UUID può essere generato utilizzando il comando `uuidgen` da terminale in OS X, o tramite la classe `NSUUID`. La seguente tabella mostra come questi valori possano essere utilizzati da una catena di negozi. Utilizzando queste informazioni, un dispositivo iOS può identificare quando

Negozi	Ancona	Jesi	Camerino
UUID	D9B9EC1F-3925-43D0-80A9-1E39D4CEA95C		
Major	1	2	3
Minor - Vestiti	10	10	10
Minor - Scarpe	20	20	20
Minor - Intimo	30	30	30

Tabella 3.2: Esempio utilizzo UUID, major e minor

entra ed esce da un negozio e in che tipologia di negozio si trova.

iBeacon utilizzano il BLE, perciò richiede un iPhone 4s(o superiori), iPod touch(quinta generazione), iPad(terza generazione o iPad mini).

3.1.1 Privacy and Location

Dato che la tecnologia iBeacon è parte del Core Location, le stesse autorizzazioni sono richieste per poter essere utilizzata(vedi Figura 3.1). L'utente visualizzerà lo stesso alert quando un'applicazione tenterà di utilizzare le API di iBeacon¹.

Come quando è attivo il GPS, quando utilizziamo la tecnologia iBeacon sarà presente una freccia nella status bar.

¹NOTA: Ciascun pacchetto Bluetooth che è associato con gli iBeacon sarà escluso dalle API del CoreBluetooth, nonostante ciò, senza attivare il Bluetooth le API di iBeacon non funzioneranno.

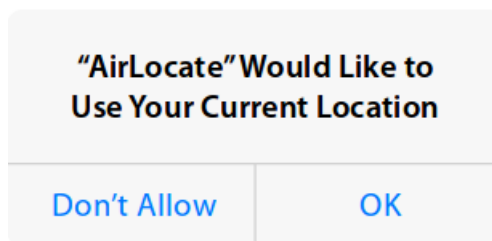


Figura 3.1: Autorizzazioni iOS

3.1.2 Precisione degli iBeacon

Per migliorare la user experience, è importante prendere in considerazione come il segnale dei beacon viene letto e utilizzato per determinare la distanza da esso. Quando un dispositivo iOS rileva un beacon, utilizza la potenza del segnale (RSSI - Received Signal Strength Indication) per determinare la proximity dal beacon. Più potente è il segnale e migliore sarà la stima fatta dal device. La potenza del segnale è generalmente correlata dalla distanza fra il beacon e il dispositivo. In una condizione ideale la vicinanza del dispositivo dal beacon migliora la precisione. Quando un dispositivo è lontano dal beacon, la potenza del segnale sarà minore rispetto a quando è vicino. Minore è la potenza del segnale, minore sarà la precisione nel calcolo della prossimità. Lo stesso accade con il GPS. Come il dispositivo si avvicina al beacon, il segnale diventa più forte e di conseguenza migliora la precisione della stima della distanza. Ad esempio, se ci troviamo a un metro di distanza il segnale sarà forte e la distanza dal dispositivo sarà calcolata con un errore di qualche cm. Altrimenti, se ci troviamo a dieci metri di distanza il segnale sarà debole e la distanza dal dispositivo sarà calcolata con l'errore di qualche metro. Come per il segnale GPS, anche quello del beacon può essere ostacolato da un oggetto fisico. Il corpo umano, per esempio, può attenuare il segnale Bluetooth. Basta dare le spalle al beacon per diminuire la precisione del dispositivo nel calcolare la distanza dal beacon.

Quando si costruisce un'applicazione che utilizza GPS o beacon, è importante prendere in considerazione la precisione. Il valore riportato dall'oggetto Core Location, indica il livello di incertezza, o margine di errore misurato in metri.

3.1.3 Monitoraggio regioni

In maniera simile al monitoraggio di regioni con geofence, un'applicazione può richiedere di ricevere una notifica quando un dispositivo entra o lascia una regione definita da un beacon. Quando si crea la richiesta si deve specificare l'UUID dell'iBeacon advertisement. Mentre un'applicazione può monitorare un numero limitato di 20 regioni, usando un singolo UUID in più zone, un dispositivo può monitorare svariate zone contemporaneamente.

In aggiunta all'UUID, un'applicazione può selezionare il major ed il minor per rendere più specifica la regione da monitorare.

Come per il monitoraggio di regioni con il GPS, quando un utente entra o esce da una regione l'applicazione riceverà una notifica. Se l'applicazione non è in esecuzione (per esempio se è stata terminata per necessità di ram da parte del sistema), l'applicazione verrà lanciata in background e gli verrà inoltrata la notifica. Un'importante considerazione sta nel fatto che se l'utente disabilita "Background App Refresh" allora l'app

non verrà riavviata e quindi non riceverà le notifiche.

3.1.4 Ranging

iOS 7 ha introdotto un nuovo set di API per determinare approssimativamente la prossimità utilizzando iBeacon technology, processo conosciuto come "Ranging". Basandosi sui più comuni scenari, iOS applica dei filtri alla precisione calcolata per determinare una stima della prossimità dal beacon. Questa stima è indicata utilizzando uno dei seguenti stati di prossimità:

Proximity State	Descrizione
Immediate	Indica l'immediata vicinanza del dispositivo al beacon
Near	Se non ci sono ostacoli tra il beacon ed il dispositivo, indica una distanza approssimativa che va da 1 a 3 metri.
Far	Questo stato indica che un beacon è stato rilevato ma non si è in grado di determinare se si è Near o Immediate. Un'importante considerazione è che questo stato non implica che si è lontani dal beacon.
Unknow	La prossimità dal beacon non può essere determinato. Questo può indicare che il ranging è appena iniziato.

Tabella 3.3: Proximity state

3.1.5 Physical limitations

I dispositivi iBeacon utilizzano il Bluetooth Low Energy per inviare i messaggi. BLE lavora ad una frequenza di 2.4GHz e può essere soggetto ad attenuazioni da qualsiasi oggetto fisico come pareti, porte o qualsiasi altro oggetto. La frequenza 2,4 GHz può essere ostacolata anche dall'acqua, ciò significa che anche le persone attenuano il segnale. Come detto precedentemente, quando il segnale ricevuto è debole, diminuisce la capacità del dispositivo di calcolare una stima della prossimità.

3.1.6 Calibrazione

Per migliorare la user experience, un aspetto fondamentale è la calibrazione dei beacon nel vostro sistema. È consigliato eseguirla per ogni singolo dispositivo. Il calcolo della distanza da parte del Core Location è effettuato utilizzando il valore Measured Power ottenuto, appunto, con la calibrazione. Per eseguirla dobbiamo:

1. Installare il beacon;
2. Utilizzando un dispositivo con iOS 7 o superiore con il Bluetooth 4.0, campionare la potenza del segnale ad un metro di distanza per un minimo di 10 secondi. In questa fase bisogna ricordarsi di tenere il dispositivo in verticale e di lasciare libera la metà superiore del device.
3. Muovere il dispositivo avanti e indietro lentamente per 30cm mantenendo l'orientamento e la distanza dal beacon(vedi Figura 3.2).

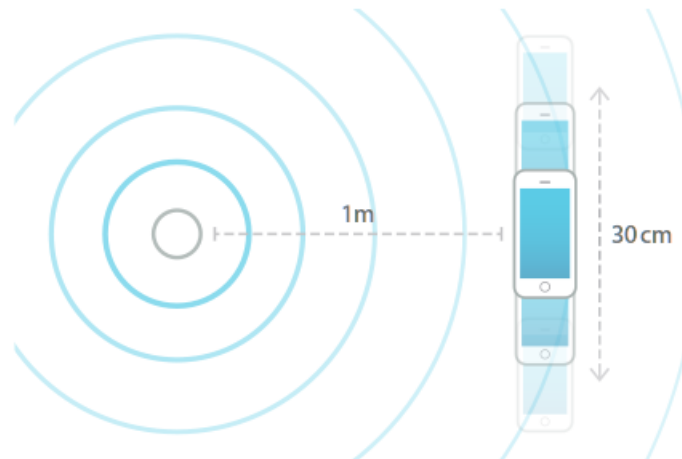


Figura 3.2: Calibrazione

4. Calcolare la media dei valori *rss* registrati per ottenere la **Measured Power**;
5. Applicare il valore di **Measured Power** al beacon;

Dato che il sistema circostante può influire sulla frequenza del bluetooth, è importante ripetere la procedura per ogni beacon installato.

3.1.7 UUID

Un **Universally Unique Identifier**(UUID) definito in RFC 4122[[IETF05](#)] è un identificatore standard usato per la costruzione di applicazioni. Esso è semplicemente un numero rappresentato da 128 bit. Il significato di ogni bit è definito nella specifica di ciascuna delle sue varianti.

Per essere letti anche dagli esseri umani, molti sistemi lo rappresentano in esadecimale inserendo dei caratteri dash al suo interno. Un esempio: **de305d54-75b4-431b-adb2-eb6b9e546014**

L'obiettivo degli UUID è quello di distribuire sistemi con un id univoco senza un'unità centrale di controllo. In questo contesto la parola univoco prende il significato di "praticamente univoco" e non garantisce niente. Chiunque può creare un UUID ed utilizzarlo per identificare qualcosa sapendo che le probabilità che qualcun altro generi lo stesso UUID sono veramente basse, se non nulle.

Un UUID è formattato secondo le specifiche delle varianti. Esistono 5 varianti di UUID:

1. Version 1 (MAC address & data-time)
2. Version 2 (DCE Security);
3. Version 3 (MD5 hash & namespace);
4. Version 4 (random);
5. Version 5 (SHA-1 hash & namespace).

Se ai 128 bit togliamo i due bit che identificano RFC 4122 e i 4 che identificano la versione, abbiamo che gli UUID generati hanno 122 bit random. La possibilità che due UUID abbiano lo stesso identico valore può essere calcolata utilizzando il calcolo della probabilità (il problema del compleanno).

$$p(n) \cong 1 - e^{-\frac{n^2}{2 \cdot 2^x}} \quad (3.1)$$

Questa è la probabilità di una collisione calcolando n UUID, con $x = 122$:

n	probabilità
$65.719.476.736 = 2^{36}$	$0,0000000000000004 = 4(10^{-16})$
$2.199.023.255.552 = 2^{41}$	$0,00000000000005 = 5(10^{-13})$
$70.368.744.177.664 = 2^{46}$	$0,0000000005 = 5(10^{-10})$

Tabella 3.4: Probabilità collisione UUID

3.2 BlueUp

Tra le diverse aziende presenti nel mercato, abbiamo scelto un'azienda italiana per l'acquisto dei nostri primi beacon: la BlueUp.

3.2.1 BlueBeacon

BlueUp realizza beacon e sensori con tecnologia Bluetooth Low Energy: la serie BlueBeacon² offre una delle più ampie selezioni del mercato.

I dispositivi BlueBeacon sono proposti in due versioni firmware, che supportano i principali standard presenti sul mercato: la tecnologia iBeacon di Apple oppure la specifica Eddystone di Google.

3.2.2 Caratteristiche

I BlueBeacon hanno prestazioni radio allo stato dell'arte, superiori alla maggior parte dei dispositivi beacon in commercio. Questo risultato è garantito da una serie di scelte tecnologiche che sono state adottate sui BlueBeacon:

- utilizzo del pluri-premiato chip radio Nordic nRF51822, che garantisce ottime prestazioni a livello radio e di consumi;
- antenna stampata su PCB di tipo meandered PIFA (Planar Inverted F Antenna) che offre prestazioni superiori in termini di guadagno rispetto alle antenne a chip comunemente adottate;
- antenna progettata per essere adattata in presenza dell'involucro plastico, riducendo quindi le perdite per riflessione;
- rete di adattamento RF basata su singolo chip (balun), che riduce le perdite di inserzione rispetto alle convenzionali reti ad elementi discreti;

²Per maggiori informazioni riguardo le caratteristiche tecniche dei prodotti BlueUp si rimanda alla bibliografia[[Blu](#)]

Grazie a queste soluzioni, i BlueBeacon, a parità di potenza emessa dal chip radio, trasmettono un livello di potenza superiore di un fattore 2 - 4 rispetto a dispositivi concorrenti. Questo permette di ridurre la potenza emessa dal chip, con un conseguente riduzione dei consumi e, quindi, aumento della vita operativa delle batterie. Inoltre, un segnale con un livello di potenza maggiore garantisce una superiore stabilità in ricezione e un aumento della distanza di comunicazione del beacon.

3.2.3 Trasmissione

La misura della distanza con i beacon è basata sul valore dell'RSSI (Received Signal Strength Indicator), ovvero della potenza del segnale RF ricevuto dallo smartphone. Il segnale ricevuto risente fortemente di una serie di elementi, fra cui, l'orientazione e la posizione relative dello smartphone rispetto al beacon, la presenza del corpo umano, fenomeni di riflessione e diffrazione dall'ambiente (soprattutto strutture metalliche). Di conseguenza, il valore di RSSI non può fornire una misura esatta della distanza in un ambiente complesso (questa condizione sarebbe vera solo nel caso ideale di spazio libero), ma solo una stima approssimata della vicinanza dello smartphone dal beacon. Da tenere in conto, inoltre, che l'accuratezza è fortemente dipendente dall'intervallo di advertising.

La massima distanza di trasmissione (ovvero la massima distanza a cui può essere ricevuto il segnale trasmesso dal beacon) dipende da una serie di parametri. Il più importante è la potenza trasmessa: maggiore è la potenza trasmessa è maggiore è la distanza, anche se non c'è una relazione di linearità fra le due grandezze, a causa dei meccanismi fisici di propagazione. Altri aspetti che hanno un impatto sulla distanza di trasmissione sono:

- il punto di installazione del beacon: più il beacon è posto in alto, maggiore è la distanza di comunicazione
- l'ambiente operativo: la presenza di ostacoli, oggetti (soprattutto metalli e liquidi), persone, frapposti (o in prossimità) fra il beacon ed il ricevitore ha un impatto non trascurabile sulla distanza
- le prestazioni del ricevitore: la sensibilità e il guadagno del ricevitore variano da modello a modello di smartphone, e inoltre la sensibilità può essere anche affetta da fattori esterni, come la presenza di altri dispositivi radio
- altri fattori, fra cui l'orientazione reciproca fra beacon e smartphone, il modo come viene impugnato lo smartphone da parte dell'utente.

La tabella 3.5, riporta la distanza massima teorica, nel caso di diversi valori di potenza trasmessa, valutata nell'ipotesi di un ricevitore (smartphone) con sensibilità (minimo valore di RSSI ricevibile) pari a -90dBm. I risultati della tabella sono stimati nel caso di beacon e smartphone posti a 1.5 metri di altezza, in linea di vista, in spazio libero.

3.2.4 Batteria

La vita operativa della batteria del BlueBeacon dipende da una serie di fattori: intervallo di advertising (fattore principale), potenza trasmessa, valore e variazioni della temperatura operativa, modalità operativa ("non-connectable mode" o "connectable mode"). La tabella 3.6 riporta la vita operativa attesa (in mesi, espressa in funzione

	Mini	Maxi	Forte	USB
+4 dBm	100m	100m	110m	-
+0 dBm	60m	60m	65m	30m
-8 dBm	25m	25m	30m	12m
-20 dBm	6m	6m	7m	3m

Tabella 3.5: Distanza massima di trasmissione

della potenza trasmessa e dell'intervallo di advertising), per ognuno dei beacon a batteria, BlueBeacon Mini, Maxi e Forte. I valori sono stimati sulla base delle misure di laboratorio di assorbimento di corrente e dei valori nominali di capacità e autoscarica forniti dal produttore della batteria. La temperatura operativa è ipotizzata di circa 20C: nel caso di valori di temperatura estremi (in particolare molto bassi), la vita operativa può essere limitata. Le misure sono effettuate con i beacon configurati in "non-connectable mode". Se i beacon sono in "connectable mode", la vita operativa della batteria si può ridurre fino al 30% in meno.

	0,1 sec	0,3 sec	0,5 sec	0,7 sec	1,0 sec
+0 dBm	4	11	16	20	25
-8 dBm	4,5	11,5	17	22	27
-20 dBm	5	12,5	19	24	29

Tabella 3.6: Durata batteria BlueBeacon Mini in mesi

3.3 Utilizzo dei Beacon in Proximity System

Inizialmente si era pensato di realizzare un prodotto che eseguisse operazioni sull'ufficio basandosi unicamente sulla posizione dell'utente all'interno di esso, da qui il nome Proximity System. In seguito, abbiamo stabilito che, nonostante l'esecuzione automatica di determinate azioni a seconda della posizione doveva rimanere la caratteristica principale, il tutto deve poter funzionare anche senza beacon. Per questo motivo, abbiamo previsto tre modalità di funzionamento per i singoli dispositivi:

1. **Manuale:** l'utente può comandare il dispositivo da qualsiasi posizione;
2. **Semi-automatica:** l'utente può comandare il dispositivo solo se si trova all'interno della regione del beacon associato al dispositivo;
3. **Automatica:** il dispositivo si accende e spegne automaticamente rispettivamente se un utente entra o esce dalla regione del beacon associato.

4. Scelta tecnologica

In questo capitolo illustreremo i meccanismi di base della nostra applicazione e modelleremo i nostri dati, per poi continuare scrivendo i test che valideranno il comportamento della nostra applicazione e descriveremo come è organizzato il nostro codice.

Per sviluppare la nostra applicazione abbiamo utilizzato lo stack MEAN che può essere sintetizzato come segue

- M = MongoDB/Mongoose.js: il popolare database, e l'elegante ODM per Node.js
- E = Express.js: un framework web leggero
- A = Angular.js: un framework per la creazione di single page application
- N = Node.js: un interprete javascript lato server

Lo stack MEAN è un sostituto dello stack LAMP (Linux, Apache, MySQL, PHP/-Perl/Python/P...) molto popolare per la costruzione di applicazione web negli anni 90.

In questa tesi non descriveremo Angular perché verrà fatto nell'elaborato di Marco in cui descriverà il frontend. Andiamo quindi a costruire le REST API che non hanno una user interface, ma sono la nostra base per qualsiasi tipo di applicazione, come un sito web, un'app Android, iOS o Ubuntu Touch.

4.1 JSON

JSON è un formato di dati molto leggibile basato su un sottoinsieme della sintassi JavaScript, cioè *array literal* e *object literal*, proposto da Douglas Crockford come formato per i servizi web in sostituzione al verboso XML. Dato l'utilizzo della sintassi JavaScript, le definizioni JSON possono essere incluse all'interno dei file JavaScript ed è possibile accedervi senza l'analisi aggiuntiva necessaria con i linguaggi basati su XML.

4.1.1 Gli array literal

Gli *array literal* rappresentano il metodo più semplice per creare un Array in JavaScript. Infatti, basta elencare una lista di valori separati da una virgola all'interno delle parentesi quadre, per esempio:

```
1 var users = ["Saverio", "Marco", "Francesco"];
```

Funzionalmente parlando è equivalente alla forma tradizionale:

```
1 var users = new Array("Saverio", "Marco", "Francesco");
```

Entrambe le definizioni generano lo stesso risultato ed è possibile accedere ad ogni elemento dell'array utilizzando il relativo indice

```
1 console.log(users[0]); \\Saverio
2 console.log(users[1]); \\Marco
3 console.log(users[2]); \\Francesco
```

Quando si parla di array in JavaScript è bene tenere in mente due cose. La prima è che JavaScript non è tipizzato e quindi l'array può contenere tipi di dati completamente diversi. La seconda è che nonostante entrambi i metodi per la creazione di un array siano leciti in JavaScript, soltanto gli array literal sono validi in JSON.

4.1.2 Gli object literal

Un object literal è definito tra parentesi graffe. Al loro interno troviamo un numero qualsiasi di coppie chiave-valore, definite con una stringa, i due punti ed il valore. Ogni coppia deve essere seguita da una virgola, eccetto l'ultima. Questo insieme di coppie chiave-valore rappresenta alla fine un oggetto. Un esempio:

```
1 var user = {
2     "firstname": "Saverio",
3     "permission": 0,
4     "block": false
5 };
```

Il codice qui sopra genera un oggetto user con le proprietà firstname, permission e block. Per accedere alle proprietà dell'oggetto basta usare la notazione con il punto

```
1 console.log(user.firstname); \\ "Saverio"
2 console.log(user.permission); \\ "0"
3 console.log(user.block); \\ "false"
```

Lo stesso oggetto potrebbe essere creato utilizzando il costruttore Object di JavaScript

```
1 var user = new Object();
2 user.firstname = "Saverio";
3 user.permission = 0;
4 user.block = false;
```

Anche in questo caso, sebbene entrambi gli approcci siano validi in JavaScript, in JSON è valida solo la notazione object literal.

4.1.3 La sintassi JSON

La sintassi di JSON non è altro che il miscuglio di object literal ed array literal per memorizzare dati. Quello che bisogna tener bene a mente è che JSON rappresenta solo dati. Un esempio:

```
1 [
2     {
3         "firstname": "Saverio",
4         "permission": 0,
5         "block": false
6     },
7     {
8         "firstname": "Marco",
9         "permission": 10,
10        "block": false
11    }
12 ]
```

La prima cosa che salta all'occhio è che in questo documento non sono presenti variabili così come punti e virgola. In questo modo quando si trasmette dei dati tramite HTTP

ad un browser, il tutto avviene abbastanza velocemente grazie al numero ridotto di caratteri.

Oltre a questo, ci sono altri ovvi benefici dell'utilizzare JSON come formato dati per la comunicazione JavaScript: esso consente di non preoccuparsi della valutazione dei dati, e quindi, garantisce un accesso più veloce alle informazioni che essi contengono. Per un confronto dettagliato tra JSON e XML si rimanda alla bibliografia [JJN07, pag. 251].

4.2 REST

REST sta per Representational State Transfer[Lei15]. È una alternativa più semplice di SOAP e WSDL che sono protocolli basati su XML.

REST utilizza un modello client-server, dove il server è un server HTTP e il client invia richieste HTTP (GET, POST, PUT, DELETE), con un URL che al suo interno contiene i parametri codificati. L'URL describe l'oggetto e il server risponde, almeno nel nostro caso, con un codice e del JSON (JavaScript Object Notation). In genere ci si aspetta che un server REST restituisca un documento XML, sebbene nulla dell'architettura REST definisca il tipo di dati restituiti. Un generico URL potrebbe restituire un'immagine, un documento HTML, un file CSV o qualunque altro tipo di dato.

Dato che utilizziamo lo stack MEAN, abbiamo scelto di utilizzare documenti JSON che risultano particolarmente adatti per la nostra applicazione, in cui tutti i nostri componenti sono in javascript e MongoDB interagisce perfettamente con questo formato. Faremo degli esempi più dettagliati più avanti quando definiremo i nostri Data Model.

L'acronimo CRUD è spesso utilizzato per descrivere le operazioni in un database. CRUD sta per CREATE, READ, UPDATE and DELETE. Queste operazioni possono essere facilmente mappate con i metodi HTTP, come segue:

- POST: Un client vuole inserire o creare un nuovo oggetto
- GET: Un client vuole leggere un oggetto
- PUT: Un client vuole aggiornare un oggetto
- DELETE: Un client vuole eliminare un oggetto

Alcuni dei codici di risposta del protocollo HTTP che spesso vengono utilizzati insieme alle API REST sono i seguenti:

- 200 - "OK"
- 201 - "Created" (Utilizzato con POST)
- 400 - "Bad Request" (Ad esempio per l'assenza di parametri)
- 401 - "Unauthorized" (Non ci poso i parametri per l'autenticazione)
- 403 - "Forbidden" (L'utente è autenticato ma non ha i permessi)
- 404 - "Not Found"

Una descrizione completa la si può trovare nel rispettivo RFC 2616[IET99]

Lo sviluppo di API REST abilita lo sviluppo della fondamenta con il quale poi svilupperemo altre applicazioni. Come già detto precedentemente, queste applicazioni possono essere pensate per il web oppure specifiche di una determinata piattaforma.

Oggi giorno, ci sono molte aziende che sviluppano applicazione che non sono web, come Uber, WhatsApp, Postmates, and Wash.io. Le API REST permettono la facile implementazione su molte piattaforme che potranno essere sviluppate in un secondo momento, trasformando il progetto iniziale in un progetto indipendente dalla piattaforma utilizzata.

Possiamo iniziare a descrivere i dati che dobbiamo memorizzare:

- Dati degli utenti
- Informazioni dei vari dispositivi
- Dati dei beacon
- GPIO(General Purpose Input Output)
- Impostazioni del sistema

Diamo una sguardo alla prima collection:

```
1 {  
2   "_id": ObjectId("523b1153a2aa6a3233a913f8"),  
3   "username": "admin",  
4   "firstname": "Saverio",  
5   "lastname": "Tosi",  
6   "password": "123456",  
7   "permission": 0,  
8   "theme": "red",  
9   "block": false,  
10  "photo": "codificata_in_base_64"  
11 }
```

Codice 4.1: User collection

Quando si parla di Database Relazionali si parla di Tabelle, righe e colonne. In MongoDB, database non relazionale, c'è un collegamento con la maggior parte dei concetti dei Database Relazionali. Ad alto livello, MongoDB supporta uno o più database. Un database contiene una o più collezioni, che sono simili alle tabelle dei database relazionali. Le collezioni, a loro volta, contengono documenti. Ogni documento in una collezione è simile ad una riga di una tabella di un database relazionale. Con la differenza che un documento non segue uno schema specifico con colonne predefinite e con variabili semplici. Ogni documento consiste in una o più coppie chiave-valore dove il contenuto può essere o una variabile semplice oppure qualcosa di più complicato come un array.

Il documento JSON rappresenta un utente del nostro sistema. Lo scopo di ogni singolo campo è abbastanza intuitivo. Il campo più importante è `_id`. In MongoDB, questo campo rappresenta la chiave primaria, se si salva un documento senza questo campo, esso verrà creato direttamente da MongoDB.

```
1 {  
2   "_id": ObjectId("523b1153a2aa6a3233a913f8"),  
3   "uuid": "ACFD065E-C3C0-11E3-9BBE-1A514932AC01",  
4   "major": "0",  
5   "minor": "0",  
6   "state": "1"  
7 }
```

Codice 4.2: Beacon collection

Il documento che descrive i beacon conterrà soltanto 4 campi: i primi tre sono quelli descritti nel capitolo precedente e rappresentano il payload del beacon; l'ultimo rappresenta lo stato del beacon, ovvero se esso è stato solo segnalato, se è stato registrato o se è stato eliminato e non deve essere più segnalato dagli utenti

```

1 {
2   "_id": ObjectId("523b1153a2aa6a3233a913f8"),
3   "type": "Lampada",
4   "io": "output",
5   "permission": "0",
6   "name": "Lampada_Saverio",
7   "description": "Lampada_verde_sulla_scrivania",
8   "automatic": false,
9   "_GPIO": id_GPIO,
10  "_Beacon": id_beacon,
11  "properties": {}
12 }
```

Codice 4.3: Device collection

Il documento che descrive il device memorizza: il tipo, se è di output o di input; il livello dei permessi; se deve essere comandato in maniera automatica; un riferimento al GPIO ed un riferimento al beacon associato. I permessi vanno da 0 a 10. 0 identifica l'admin, più è alto il valore di permission, più utenti potranno accedere al dispositivo. In altre parole, un utente può accedere ad un dispositivo se e solo se il numero che identifica i permessi dell'utente (sempre da 0 a 10) è compreso tra 0 e il valore di permission del device

```

1 {
2   "_id": ObjectId("523b1153a2aa6a3233a913f8"),
3   "type": "output",
4   "GPIO": 3,
5   "_Device": id_device,
6   "value": true
7 }
```

Codice 4.4: GPIO collection

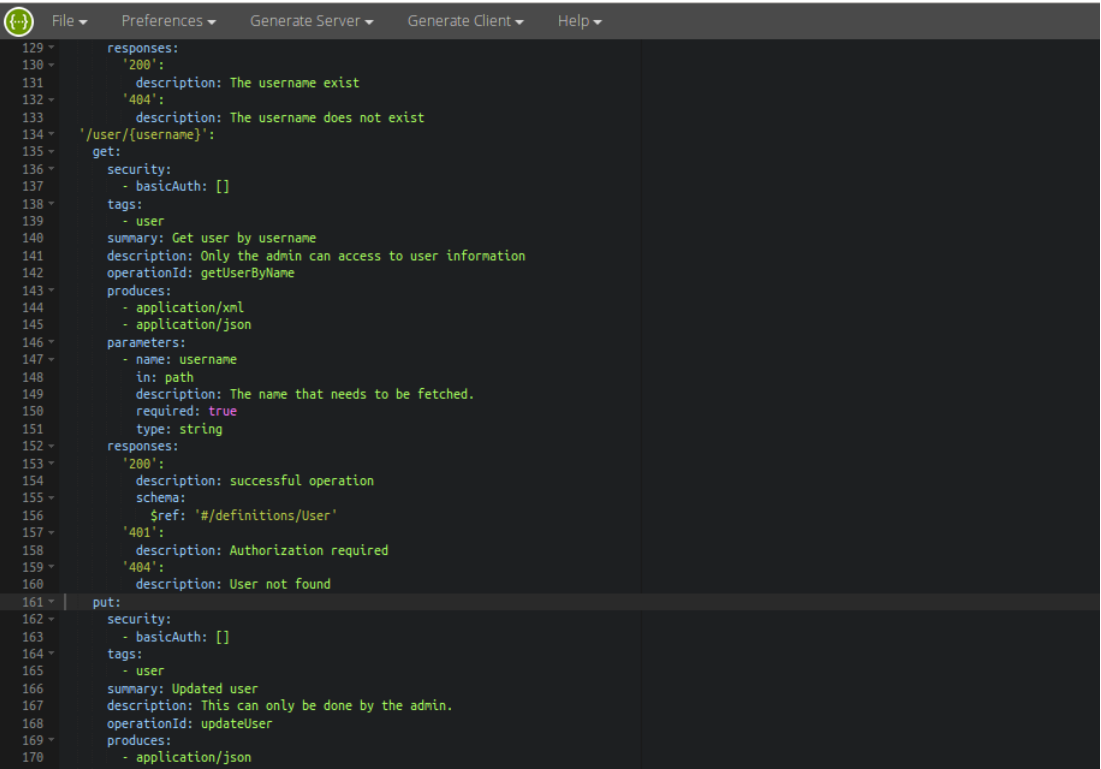
Un GPIO avrà soltanto le seguenti informazioni: il tipo, output o input; GPIO, che identifica il piedino nel raspberry; _Device, un riferimento al dispositivo associato; value, che rappresenta il valore attuale del GPIO.

section

4.3 Documentazione

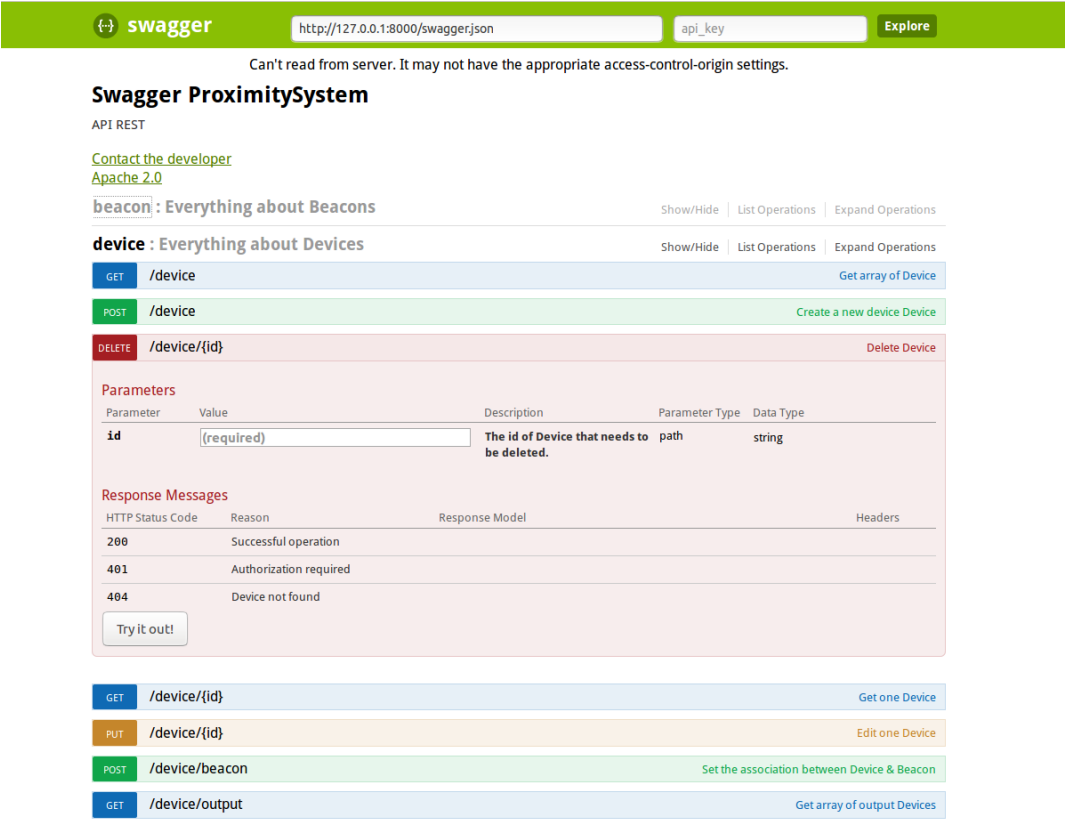
Una parte fondamentale della progettazione di API REST è la documentazione. Per questa funzione abbiamo utilizzato i tool messi a disposizione da swagger[[Swa](#)].

L'editor permette di descrivere le proprie API tramite un linguaggio chiamato YAML che rende la scrittura della documentazione semplice e veloce. Quello che si ottiene è una documentazione interattiva, con la quale è anche possibile testare le API.



```
129 ~ responses:
130 ~   '200':
131 ~     description: The username exist
132 ~   '404':
133 ~     description: The username does not exist
134 ~ '/user/{username}':
135 ~   get:
136 ~     security:
137 ~       - basicAuth: []
138 ~     tags:
139 ~       - user
140 ~     summary: Get user by username
141 ~     description: Only the admin can access to user information
142 ~     operationId: getUserByName
143 ~     produces:
144 ~       - application/xml
145 ~       - application/json
146 ~     parameters:
147 ~       - name: username
148 ~         in: path
149 ~         description: The name that needs to be fetched.
150 ~         required: true
151 ~         type: string
152 ~     responses:
153 ~       '200':
154 ~         description: successful operation
155 ~         schema:
156 ~           $ref: '#/definitions/User'
157 ~       '401':
158 ~         description: Authorization required
159 ~       '404':
160 ~         description: User not found
161 ~ put:
162 ~   security:
163 ~     - basicAuth: []
164 ~   tags:
165 ~     - user
166 ~   summary: Updated user
167 ~   description: This can only be done by the admin.
168 ~   operationId: updateUser
169 ~   produces:
170 ~     - application/json
```

Figura 4.1: Swagger editor



swagger Explore

Can't read from server. It may not have the appropriate access-control-origin settings.

Swagger ProximitySystem

API REST

[Contact the developer](#)
[Apache 2.0](#)

beacon: Everything about Beacons Show/Hide | List Operations | Expand Operations

device: Everything about Devices Show/Hide | List Operations | Expand Operations

Method	Path	Action
GET	/device	Get array of Device
POST	/device	Create a new device Device
DELETE	/device/{id}	Delete Device

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	(required)	The id of Device that needs to be deleted.	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Successful operation		
401	Authorization required		
404	Device not found		

Try it out!

GET	/device/{id}	Get one Device
PUT	/device/{id}	Edit one Device
POST	/device/beacon	Set the association between Device & Beacon
GET	/device/output	Get array of output Devices

Figura 4.2: Swagger ui

4.4 NodeJS

Node.js[[Nod](#)] è un interprete javascript per applicazioni di rete lato server. È disponibile per diverse piattaforme, come Linux, Microsoft Windows e Apple OS X. Le applicazioni Node.js vengono costruite utilizzando librerie e moduli che sono disponibili per l'ecosistema ed alcuni di essi li abbiamo utilizzati per la nostra applicazione. Per iniziare ad utilizzare Node.js, dobbiamo per prima cosa creare un file `package.json` che descrive la nostra applicazione ed elenchi tutte le sue dipendenze. NPM (Node.js Package Manager) installa una copia delle librerie in una sotto-cartella chiamata `node_modules/`, della cartella principale dell'applicazione. Questo comporta diversi benefici, ad esempio isola le diverse versioni delle librerie, evitando così i problemi di compatibilità che si sarebbero avuti se avessimo installato tutto in una cartella standard come `/usr/lib`.

Il comando `npm install` creerà la cartella `node_modules/`, con dentro tutte le librerie richieste

```

1 {
2   "name": "proximity-system",
3   "version": "1.0.0",
4   "description": "Backend di Proximity System",
5   "scripts": {
6     "start": "sudo_node_server.js",
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "author": "Marco D'Argenio - Saverio Tosi",
10  "license": "Apache 2.0",
11  "dependencies": {
12    "async": "^2.0.0-rc.6",
13    "body-parser": "^1.4.3",
14    "cors": "^2.7.1",
15    "express": "^4.13.4",
16    "frisby": "^0.8.5",
17    "mongoose": "^4.5.2",
18    "morgan": "^1.7.0",
19    "rpi-gpio": "^0.7.0"
20  }
21 }
```

Codice 4.5: package.json

Il nome della nostra applicazione è `proximity-system`. La versione è la `1.0.0`. Sono presenti due script: il primo serve ad avviare il webserver, i permessi di root sono necessari per utilizzare i GPIO nel raspberry; ed un secondo script per eseguire i test. Poi sono segnati l'autore, la licenza e l'elenco delle dipendenze. I test dell'applicazione li abbiamo scritti con Frisby ed eseguiti con Jasmine-node.

Una libreria particolarmente interessante è `async`. Node.js è stato progettato per essere asincrono. Qualsiasi operazione che ha a che fare con blocchi di I/O, come leggere un file o interrogare un database, prenderà una callback come ultimo parametro e continuerà l'esecuzione del programma, solo quando l'operazione sarà completa il programma richiamerà la callback. Ecco un semplice esempio per spiegare meglio il tutto.

```

1 function foo() {
2   someAsyncFunction(params, function(err, results) {
3     console.log("one");
4   });
5   console.log("two");
6 }
```

Codice 4.6: operazioni asincrone

Dal codice 4.6 ci si potrebbe aspettare che il risultato sia: `onetwo`. In realtà potrebbe capitare che l'output sia `twoone`. Questa situazione di incertezza dei programmi asincroni viene chiamata esecuzione non deterministica. Questo caratteristica permette al programma di essere particolarmente performante, ma ci sono alcuni casi in cui è desiderabile eseguire determinate azioni in un determinato ordine.

Per questo scopo ci viene incontro `Async`. `Async` è un modulo che fornisce delle funzioni per lavorare con il JavaScript asincrono. Ce ne sono circa venti tra cui molte classiche come: `map`, `reduce`, `filter`, `each`, etc. Altre più comuni per il controllo del flusso asincrono come `parallel` e `series`. Tutte queste funzioni seguono la convenzione di `Node.js` di mettere una singola callback come ultimo parametro della funzione. L'esempio seguente illustra come stampare i due numeri nell'ordine corretto utilizzando la libreria `async`:

```
1 actionArray = [  
2   function one(cb) {  
3     someAsyncFunction(params, function(err, results) {  
4       if (err) {  
5         cb(new Error("There_was_an_error"));  
6       }  
7       console.log("one");  
8       cb(null);  
9     });  
10  },  
11  function two(cb) {  
12    console.log("two");  
13    cb(null);  
14  }  
15 ]  
16 async.series(actionArray);
```

Codice 4.7: operazioni sincrone

In questo caso siamo sicuri che la prima funzione venga eseguita prima della seconda.

Lo sviluppo del backend, comporta inizialmente l'utilizzo di test. Se pur affrontati in modo superficiale nella nostra applicazione, parleremo dei principali vantaggi che essi comportano:

- Aiuta lo sviluppatore a capire come verranno consumati i servizi. In modo tale da poterli modificare prima di farli utilizzare ai vari client.
- Scrivendo i test prima di costruire l'applicazione, il paradigma diventa "rotto/non implementato finché non passa i test" invece di assumere che funzioni finché qualcosa non va storto.

Prima di scrivere il nostro primo test, per iniziare andiamo a definire il file di configurazione.

```
1 module.exports = {  
2   url : 'http://localhost:8000/api/v2.0'  
3 }
```

Codice 4.8: test/config/test_config.js

Il nostro server, almeno in fase di sviluppo, sarà eseguito in locale sulla porta 8000. Questo solo per i test iniziale. Quando il sistema andrà in produzione e quindi sul RaspberryPi, basterà cambiare questi parametri.

Per preparare i nostri casi di test, per prima cosa ci dobbiamo assicurare di avere un sistema per farli che funzioni correttamente. Il seguente codice fa questo per noi

```

1 function connectDB(callback) {
2   mongoClient.connect(dbConfig.testDBURL, function(err, db) {
3     assert.equal(null, err);
4     proximitysystem_test_db = db;
5     console.log("Connesso al server");
6     callback(0);
7   });
8 }

```

Codice 4.9: test/config/setup_tests.js - connectDB

Di seguito, eliminiamo la collection user. In questo modo ci assicuriamo di trovarci in uno stato che già conosciamo.

```

1 function dropUserCollection(callback) {
2   console.log("dropUserCollection");
3   user = proximitysystem_test_db.collection('User');
4   if (undefined !== user) {
5     user.drop(function(err, reply) {
6       console.log('user_collection_dropped');
7       callback(0);
8     });
9   } else {
10    callback(0);
11  }
12 }

```

Codice 4.10: test/config/setup_tests.js - dropUserCollection

In modo simile eliminiamo poi tutte le altre collection. Chiudiamo la connessione al database

```

1 function closeDB(callback) {
2   proximitysystem_test_db.close();
3 }

```

Codice 4.11: test/config/setup_tests.js - closeDB

Infine, chiamiamo `async.series` per assicurarci che tutte le funzioni vengano eseguite nell'ordine corretto

```

1 async.series([
2   connectDB,
3   dropUserCollection,
4   insertAdminInUserCollection,
5   dropDeviceCollection,
6   dropBeaconCollection,
7   dropGPIOCollection,
8   insertGPIOInGPIOCollection,
9   closeDB]);

```

Codice 4.12: test/config/setup_tests.js - async

Per testare i casi di test useremo `frisby` come segue:

```

1 TU1_UN = "test6"
2 TU1_FN = "Test";
3 TU1_LN = "User10";
4 TU1_PW = "testUser123";
5 SP_APP_NAME = 'Proximity_System';

```

Codice 4.13: test/user/create_accounts_error_spec.js

Iniziamo con testare la registrazione. In questo caso noi deliberatamente non inseriamo il primo nome, in questo modo ci aspettiamo che il server risponda con uno stato 400 e una stringa che identifichi il problema.

```
1 frisby.create('POST_missing_firstName')
2   .post(tc.url + '/user',
3     { 'lastname' : TU1_LN,
4       'username' : TU1_UN,
5       'password' : TU1_PW })
6   .expectStatus(400)
7   .expectHeader('Content-Type', 'application/json;_charset=utf-8')
8   .expectJSONTypes({'msg' : String})
9   .toss()
```

Codice 4.14: test/user/create_accounts_error_spec.js

Adesso, andiamo a vedere alcuni esempi di casi di test che dovrebbero andar a buon fine. per prima cosa definiamo tre utenti.

```
1 TEST_USERS = [{ 'fn' : 'Test', 'ln' : 'User1',
2                 'un' : 'testuser1', 'pwd' : 'testUser123'},
3                 { 'fn' : 'Test', 'ln' : 'User2',
4                 'un' : 'testuser2', 'pwd' : 'testUser123'},
5                 { 'fn' : 'Test', 'ln' : 'User3',
6                 'un' : 'testuser3', 'pwd' : 'testUser123'}]
7
8 SP_APP_NAME = 'Reader_Test';
9
10 var frisby = require('frisby');
11 var tc = require('./config/test_config');
```

Codice 4.15: test/user/create_accounts_spec.js

In questo caso, stiamo inviando 3 utenti e ci aspettiamo che il server risponda 201 in tutti e tre i casi.

```
1 TEST_USERS.forEach(function createUser(user, index, array) {
2   frisby.create('POST_user_' + user.un)
3     .post(tc.url + '/user',
4       { 'firstname' : user.fn,
5         'lastname' : user.ln,
6         'username' : user.un,
7         'password' : user.pwd })
8     .expectStatus(201)
9     .expectHeader('Content-Type', 'application/json;_charset=utf-8')
10    .toss()
11 });
```

Codice 4.16: test/user/create_accounts_spec.js

In questo esempio proviamo a creare un utente con un username già esistente.

```
1 frisby.create('POST_duplicate_user_')
2   .post(tc.url + '/user',
3     { 'firstname' : TEST_USERS[0].fn,
4       'lastname' : TEST_USERS[0].ln,
5       'username' : TEST_USERS[0].un,
6       'password' : TEST_USERS[0].pwd })
7   .expectStatus(400)
8   .expectHeader('Content-Type', 'application/json;_charset=utf-8')
9   .toss()
```

Codice 4.17: test/user/create_accounts_spec.js

Questi test sono stati scritti solo per spiegare il loro funzionamento.

Prima di iniziare a scrivere le nostre API REST, abbiamo bisogno di definire meglio la configurazione del sistema. Per prima cosa, dobbiamo definire come la nostra applicazione si conatterà ad un database. Mettendo queste informazioni in un file di configurazione possiamo aggiungere differenti URL dei database a seconda se ci troviamo in fase di sviluppo oppure in produzione.

```

1 module.exports = {
2   url : 'mongodb://localhost/proximitysystem_test'
3 }

```

Codice 4.18: config/db.js

4.5 Express

Con `express.js`, noi abbiamo creato la "applicazione" vera e propria. Questa applicazione ascolta in una particolare porta le richieste HTTP in ingresso. Quando arriva una richiesta, questa passa attraverso una catena di middleware. Ogni nodo nella catena è definito da un `req` (request) object e un `res` (results) object per salvare i risultati. Ogni nodo può decidere di eseguire delle operazioni, o di passare i due oggetti al nodo successivo. Per aggiungere un nuovo middleware usiamo `app.use()`. Il middleware principale è chiamato "router", il quale guarda l'URL e il tipo di richiesta per indirizzarlo ad una specifica funzione.

Possiamo finalmente iniziare a scrivere il codice della nostra applicazione, che sarà relativamente piccolo in quanto i vari handler saranno in file separati.

```

1 var fs = require('fs');
2 var express = require('express');
3 var cors = require('cors');
4 var bodyParser = require('body-parser');
5 var mongoose = require('mongoose');
6 var userRoutes = require('./app/routes/user');
7 var deviceRoutes = require('./app/routes/device');
8 var beaconRoutes = require('./app/routes/beacon');
9 var settingRoutes = require('./app/routes/setting');
10 var gpioRoutes = require('./app/routes/gpio');
11 var gpio = require('./app/services/gpio');
12 var db = require('./config/db');
13 var security = require('./config/security');
14 var environment = process.env.NODE_ENV;
15 console.log("Environment: ", environment);
16 var app = express();
17 var morgan = require('morgan');
18 var accessLogStream=fs.createWriteStream(__dirname + '/access.log', {flags: 'a'})
19 var port = 8000;
20 mongoose.connect(db.url);
21 app.use(express.static(__dirname + '/angular'));
22 app.use(bodyParser.json());
23 app.use(bodyParser.urlencoded({extended: true}));
24 app.use(cors());
25 app.use([userRoutes.authentication,morgan('common',{stream:accessLogStream})]);
26 userRoutes.addAPIRouter(app);
27 deviceRoutes.addAPIRouter(app, environment);
28 beaconRoutes.addAPIRouter(app);
29 settingRoutes.addAPIRouter(app);
30 gpioRoutes.addAPIRoutes(app, environment);

```

Codice 4.19: server.js

Definiamo il nostro middleware alla fine della catena per la gestione degli URL sbagliati

```

1 app.use(function(req, res, next){
2   res.status(404);
3   res.json({ error: 'Invalid_URL' });
4 });
5 // Express route to handle errors
6 app.use(function (err, req, res, next) {
7   if (req.xhr) {

```

```

8   res.status(500).send('Oops, something went wrong!');
9   } else {
10    next(err);
11  }
12 });

```

Codice 4.20: server.js - middleware

Mettiamo la nostra app in ascolto sulla porta 8000 e gestiamo il segnale SIGINT per interrompere correttamente il webserver

```

1 //catches ctrl+c event
2 process.on('SIGINT', function() {
3   console.log("Stop webserver");
4   gpio.unexportPins(environment);
5 });
6 gpio.init(environment);
7 app.listen(port);
8 console.log('GPIO setup completed and server listening on port ' + port);

```

Codice 4.21: server.js - listen

4.6 Mongoose

Il passo successivo è quello di modellare i nostri dati con mongoose. Utilizzeremo mongoose anche per mappare gli oggetti in Node.js nei document in MongoDB [\[Mon\]](#). Le collection da definire sono le seguenti:

- Beacon
- Device
- GPIO
- User

Per questo definiamo uno schema per ciascuna delle 4 collezioni. Iniziamo con lo schema degli utenti. Notiamo dallo schema, che non solo c'è la tipizzazione dei dati, ma è anche possibile formattare i dati e porre dei vincoli, ad esempio con l'utilizzo di trim.

```

1 var userSchema = new mongoose.Schema({
2   block: { type: Boolean, default: false },
3   username: { type: String, trim: true, required: true },
4   firstname: { type: String, trim: true, required: true },
5   lastname: { type: String, trim: true, required: true },
6   permission: { type: Number, default: 10 },
7   created: { type: Date, default: Date.now },
8   photo: { type: String, default: "img/account.jpg" },
9   password: String,
10  theme: { type: String, default: "altTheme" }
11 },
12 {collection: 'User'}
13 );
14 userSchema.index({username : 1}, {unique:true});
15 var User = mongoose.model('User', userSchema);

```

Codice 4.22: userSchema

Nel codice dello userSchema, definiamo con mongoose la presenza di un indice. Possiamo assicurare che gli indici vengano creati anche se essi non esistono nel nostro

database. La regola unique assicura che i duplicati non siano permessi. "username: 1" fa in modo che gli username siano mantenuti in ordine crescente. -1 avrebbe indicato l'ordine decrescente.

Adesso abbiamo bisogno di definire un handler per ogni combinazione di URL/METHOD. Qui ne descrivo solo uno ma vi ricordo che il resto del codice è disponibile su GitHub

```

1 exports.addAPIRouter = function(app, mongoose) {
2   var router = express.Router();
3   router.get('/', function(req, res) {
4     if(req.user.block === false && req.user.permission === 0) {
5       User.find(function(err, users) {
6         if(err) {
7           res.status(500).send({msg: err.errmsg});
8         } else if(users && users.length>0) {
9           res.status(200).send(users);
10        } else {
11          res.status(200).send([]);
12        }
13      });
14    } else {
15      res.status(401).send({msg: 'Authorization_required'})
16    }
17  });

```

Codice 4.23: get user

I principali metodi messi a disposizione dall'oggetto mongoose. Schema sono find, save, findAndRemove e update. Nel codice 4.23 vediamo come sia semplice accedere all'elenco degli utenti. Basta infatti richiamare User.find().

5. Websocket

Nel capitolo 4 abbiamo parlato di come e con quali tecnologie sono state implementate le API REST di proximity system. Queste API danno la possibilità a chiunque di implementare un proprio client per qualsiasi dispositivo. Il problema di questi servizi è che sono di tipo pull, ovvero dev'essere il client a chiedere informazioni al server. Il server non può inviare i dati aggiornati appena sono disponibili, ma soltanto quando un client li richiede.

In questo capitolo andremo a vedere come risolvere questo problema con il metodo di comunicazione più potente introdotto con la specifica HTML5: le WebSocket. Queste non sono altro che un canale di comunicazione full-duplex operante su una singola socket che permette di migliorare drasticamente sia la quantità di traffico che la latenza di tutte le applicazioni web basate sulla visualizzazione di dati in tempo reale e sugli eventi.

5.1 Panoramica della applicazione real-time e HTTP

Prima di andare nel dettaglio nella specifica WebSocket, vediamo quali sono e come funzionano i principali "hack" utilizzati per simulare un'applicazione real-time con il protocollo HTTP.

Nei tradizionali siti web quando un server HTTP riceve la richiesta, il server genera la pagina richiesta personalizzata per il client, la inoltra ed è il client che poi fa il render della risposta e la visualizza a video. Esistono molti casi, come la disponibilità di biglietti per un evento, dati azionari etc. in cui i dati inviati dal server potrebbero già essere deprecati nel momento in cui il client fa il render della pagina. Per avere gli ultimi dati, bisogna aggiornarli continuamente, ad esempio manualmente, anche se ovviamente non rappresenta una soluzione ottimale.

Comet è un modello per le web application nel quale una particolare gestione delle richieste HTTP permette al web server di inviare notifiche push al browser, senza che il browser faccia una richiesta specifica. Esistono diverse tecniche che permettono di implementare questo modello basato sugli eventi.

La più diffusa tra le tecniche comet è il polling, che consiste nell'invio di richieste HTTP a intervalli regolari con una risposta immediata da parte del server. Questa tecnica è efficiente nei casi in cui si conosce a prescindere ogni quanto tempo i dati verranno aggiornati dal server. Come è facile immaginare non è sempre semplice prevedere con quale cadenza i dati verranno aggiornati, causando un numero elevato di richieste non necessarie. Quello che ne deriva è l'apertura e la chiusura di un alto numero di connessioni e quindi l'utilizzo di risorse non strettamente necessarie.

Una possibile soluzione è l'implementazione del long-polling, dove il client invia una richiesta al server che risponderà soltanto se i dati sono stati aggiornati e farà scadere

la richiesta altrimenti. È importante far notare che se il numero di aggiornamenti è alto, non si hanno vantaggi rispetto al polling tradizionale. Quello in Figura 5.1 è un esempio di long-polling in azione preso da messenger.

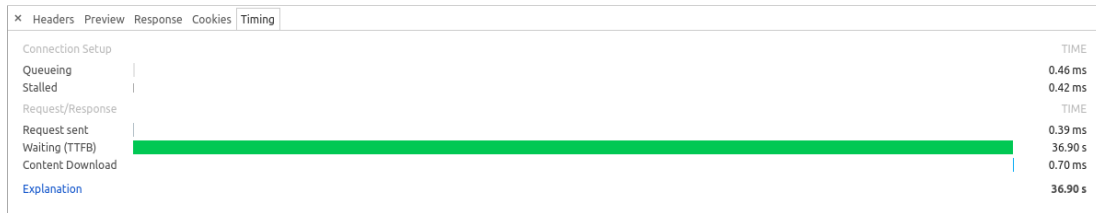


Figura 5.1: Esempio di long-polling

Un'altra tecnica, che prende il nome di streaming, consiste in una connessione completa tra client e server, in cui il server mantiene la connessione aperta con un aggiornamento costante delle informazioni per un tempo indeterminato, per poi aggiornare la risposta con l'arrivo di nuovi dati, sempre mantenendo aperta la connessione.

In questo modo la connessione rimane aperta per i messaggi seguenti. Il problema è dato dal fatto che lo streaming è incapsulato in una richiesta HTTP e in quanto tale un eventuale firewall o proxy nel mezzo potrebbero interferire bufferizzando la risposta. Per questo motivo molti sistemi passano automaticamente al long-polling se rilevano la presenza di qualche proxy. Una possibile soluzione alternativa è quella di utilizzare connessioni TLS per impedire il buffering della risposta, ma in questo caso bisogna prendere in considerazione l'utilizzo di maggiori risorse da parte del server.

Tutte le tecniche sopra descritte hanno in comune due problemi:

- prevedono l'utilizzo dell'header HTTP
- prevedono l'invio di messaggi solo dal server al client

Il primo problema consiste nel fatto che l'header HTTP contiene molte informazioni inutili al fine della comunicazione stessa, occupando banda e aumentando la latenza della connessione.

Il secondo consiste nel fatto in cui se l'utente ha la necessità di inviare dati al server, dovrà instaurare una nuova connessione per poterlo fare. Ciò comporta tutta una serie di problematiche per mantenere i dati sincronizzati tra tutte le connessioni.

In altre parole, il protocollo HTTP 1.1 è un protocollo pull e non push. In quanto tale, realizzare una webapp con il modello publish/subscribe attraverso comunicazioni HTTP half-duplex è più complicato di quanto sembri, soprattutto per soluzioni scalabili.

5.2 Introduzione alle WebSocket

Ian Hickson, uno dei principali responsabili della specifica HTML5, inizialmente definì le WebSocket nella sezione specifica dedicata ai metodi di comunicazione con il nome di "TCPConnection". Questa specifica si è evoluta poi a tal punto da far definire le WebSocket in una specifica indipendente come la geolocalizzazione, i Web Worker etc. per mantenere la discussione focalizzata.

Il protocollo WebSocket viene descritto nel RFC 6455[[IETF11](#)] ed è diviso in due parti: una che descrive l'handshake e una per il trasferimento dei dati.

Una volta che il client e il server hanno completato l'handshake, se è andato a buon fine, allora i due possono iniziare a comunicare. Quello che si crea è un canale di comunicazione full-duplex dove sia il client che il server, indipendentemente l'uno dall'altro, possono inviare dei dati.

Dopo l'handshake, i dati che vengono scambiati prendono il nome di "messaggi". A livello fisico, un messaggio è composto da uno o più frame. Quindi un messaggio WebSocket non corrisponde necessariamente ad un particolare frame, in quanto il messaggio può essere frammentato e poi ricomposto da qualsiasi intermediario tra le due parti.

Un frame ha associato un determinato tipo. Ogni frame dello stesso messaggio contiene lo stesso tipo di dati. Possiamo identificare dei dati di tipo testo(UTF-8), dati binari e frame di controllo. La versione corrente del protocollo definisce sei tipi di frame e ne lascia altri dieci per future implementazioni.

5.2.1 L'handshake WebSocket

L'handshake è stato progettato per essere compatibile con i server basati sul protocollo HTTP, in modo tale che una singola porta possa essere utilizzata sia dai client che vogliono comunicare con il WebServer sia dai client WebSocket. Per questo motivo l'handshake è una richiesta di HTTP upgrade.

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6 Origin: http://example.com
7 Sec-WebSocket-Protocol: chat, superchat
8 Sec-WebSocket-Version: 13
```

Codice 5.1: client handshake

Come stabilito nel RFC 2616[IET99], i campi dell'header possono essere inviati in qualsiasi ordine. Il metodo GET viene utilizzato sia per identificare la WebSocket corretta, sia per far in modo che domini multipli possano essere serviti da un singolo indirizzo IP e per permettere la gestione di più WebSocket in un singolo server.

Il client include nell'header il nome dell'host del server così come descritto in RFC2616, in modo tale che sia il client che il server possano verificare che l'host in uso sia quello corretto.

Campi dell'header addizionali possono essere aggiunti per selezionare delle opzioni del protocollo WebSocket. Le opzioni tipiche che vengono utilizzate sono:

- **Sec-WebSocket-Protocol:** lista dei sotto protocolli disponibili nel client;
- **Sec-WebSocket-Extension:** lista delle estensioni disponibili nel client;
- **Origin:** per la gestione del CORS.

Il campo *Origin* viene utilizzato come contromisura per l'utilizzo non autorizzato del server WebSocket da script eseguiti nel browser. Se il server non vuole accettare una connessione da una determinata origine, può scegliere di rifiutare la connessione inviando un codice di errore HTTP appropriato. Questo campo ha senso solo per i client che sono browser, in quanto tutti gli altri possono modificarlo a loro piacimento.

Adesso, il server deve dimostrare al client che ha ricevuto un messaggio di handshake e deve stare attento a non accettare richieste che non siano per le WebSocket. In modo

tale da prevenire eventuali attacchi generati con un XMLHttpRequest o l'invio di dati da un form.

Per dimostrare che ha ricevuto l'handshake corretto, il server deve prendere due dati e metterli insieme per generare una risposta. Il primo dato viene preso dal campo *Sec-WebSocket-Key*. il secondo è la stringa che rappresenta il GUID(Globally Unique Identifier RFC4122[[IETF05](#)]) "258EAF5E-E914-47DA-95CA-C5AB0DC85B11". Ad esempio, se prendiamo i dati dal handshake precedente, il server deve concatenare il GUID e il *Sec-WebSocket-Key* ottenendo la stringa "dGhlIHhXbXZsZSBub25jZQ==258EAF5E-E914-47DA-95CA-C5AB0DC85B11". Poi il server deve prendere l'hash SHA-1 di questa stringa, ovvero 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea. Questo valore deve essere poi codificato in base64, ottenendo "s3pPLMBiTxaQ9kYGzZhZRbK+xOo=". Questo valore dovrà essere inserito nell'header nel campo Sec-WebSocket-Accept".

L'handshake del server è molto più semplice rispetto a quello del client.

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzZhZRbK+xOo=
```

Codice 5.2: server handshake

Se il valore di Sec-WebSocket-Accept non è quello aspettato, se manca qualche header, o se lo stato HTTP è diverso da 101, la connessione non verrà stabilita e i frame WebSocket non verranno inviati. L'header può anche includere i campi delle opzioni e settera eventuale cookie.

5.2.2 L'interfaccia

Utilizzare le WebSocket in un'applicazione JavaScript è molto semplice. Per la connessione con l'host remoto è sufficiente creare una nuova istanza WebSocket, fornendo al nuovo oggetto l'URL con il quale si desidera aprire la connessione. I prefissi `ws://` e `wss://` indicano rispettivamente una connessione WebSocket e una connessione WebSocket sicura. La connessione WebSocket è stabilita passando dal protocollo HTTP a quello WebSocket, durante l'handshake iniziale tra client e server, attraverso la stessa connessione TCP/IP. Dopo aver stabilito la connessione i frame di dati WebSocket possono essere inviati e ricevuti in modalità full-duplex. La connessione stessa è esposta tramite l'evento `message` e il metodo `send` definiti nell'interfaccia WebSocket. Scrivendo il codice è possibile utilizzare dei *listener* asincroni per gestire ogni fase del ciclo di connessione.

```
1 url = "ws://example.com/echo";
2 socket = new WebSocket(url);
3 socket.onopen = function() {
4     console.log("WebSocket_aperta");
5     socket.send("Grazie_per_aver_accettato_questa_WebSocket");
6 }
7 socket.onmessage = function(e) {
8     console.log(e.data);
9 }
10 socket.onclose = function(e) {
11     console.log("connessione_chiusa");
12 }
```

Codice 5.3: interfaccia WebSocket

5.2.3 Aumento delle prestazioni

Le WebSocket HTML offrono vantaggi tali da spingere Ian Hickson (Google) a dichiarare:

*"Reducing kilobytes of data to 2 bytes is more than "a little more byte efficient", and reducing latency from 150ms (TCP round trip to set up the connection plus a packet for the message) to 50ms (just the packet for the message) is far more than marginal. In fact, these two factors alone are enough to make WebSocket seriously interesting to Google."*¹

Per illustrare quanto possono essere efficienti le WebSocket, andremo a confrontare il traffico e la latenza di un'applicazione web prima implementata tramite polling e poi tramite WebSocket.

Dato che il miglioramento delle performance in Proximity System è minimo, prenderemo in considerazione un'ipotetica applicazione che visualizza dati in tempo reale i dati di una macchina da corsa: velocità; giri del motore; tempo del giro, posizione, km percorsi. Supponiamo che le 2 nostre applicazioni, quella basata sul polling e quella basata sulle WebSocket, ricevano i dati tramite il seguente JSON:

```

1 {
2   "lap_time": "35.256",
3   "speed": "124",
4   "gear": "4",
5   "RPM": "15600",
6   "mileage": "12563.02"
7 }
```

Codice 5.4: JSON dati macchina

In questo caso la quantità di dati che il server invia al client è pari al numero di caratteri presenti nel JSON, escludendo gli spazi, le tabulazioni e i ritorni a capo che si possono omettere contiamo 81 caratteri e quindi 81 byte di dati. Il problema è che oltre agli 81 byte dei dati effettivi, ogni richiesta XMLHttpRequest deve considerare sia l'header inviato dal client che quello inviato dal server in risposta. Giusto per far un esempio, osserviamo i due header di una richiesta fatta utilizzando ProximitySystem

```

1 GET /api/v2.0/gpio HTTP/1.1\r\n
2 Host: localhost:8000
3 Connection: keep-alive
4 Cache-Control: max-age=0
5 Accept: application/json, text/plain, */*
6 If-Non-Match: If-None-Match: W/"4dc-Uz68k8B/IopHdTlOHcquQA"
7 User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
8               Chrome/52.0.2743.116 Safari/537.36
9 Authorization: Basic YWRtaW46MTIzNDU2
10 DNT: 1
11 Referer: http://localhost:8000/
12 Accept-Encoding: gzip, deflate, sdhc
13 Accept-Language: it-IT,it;q=0.8,en-US;q=0.6,en;q=0.4
```

Codice 5.5: header client

Solo nell'header di richiesta ci sono 484 caratteri

```

1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 1246
5 Etag: W/"4de-K40aEhIMswWt/3FPFcGKCQ"
6 Date: Mon, 29 Aug 2016 15:05:16 GMT
```

¹www.ietf.org/mail-archive/web/hybi/current/msg00784.html

7 Connection: keep-alive

Codice 5.6: header server

Aggiungendo i 210 della risposta otteniamo 694 byte di dati non necessari per ogni richiesta. È vero che questo è solo un esempio, e ci saranno situazioni con meno dati nell'header, ma è bene sapere che in alcuni casi le informazioni nell'header possono arrivare fino a 2000 byte. Andando a ricapitolare possiamo vedere come il rapporto tra dati inutili(694 byte) e dati utili(81 byte) sia di circa 8:1, un gran spreco di dati. Vediamo cosa accadrebbe distribuendo questa applicazione ad un ampio numero di utenti. ipotizziamo tre scenari di utilizzo:

1. 1.000 client che eseguono il polling ogni secondo. il traffico equivale a: $694 \times 1.000 = 694.000 \text{ byte} = 5.552.000 \text{ bit al secondo} = 5,2 \text{ Mbps}$;
2. 10.000 client che eseguono il polling ogni secondo. il traffico equivale a: $694 \times 10.000 = 6.940.000 \text{ byte} = 55.520.000 \text{ bit al secondo} = 52 \text{ Mbps}$;
3. 100.000 client che eseguono il polling ogni secondo. il traffico equivale a: $694 \times 100.000 = 69.400.000 \text{ byte} = 555.200.000 \text{ bit al secondo} = 529 \text{ Mbps}$.

Adesso che abbiamo visto la quantità di banda che si spreca utilizzando il polling, andiamo ad analizzare come si comporta la stessa applicazione con l'utilizzo delle WebSocket. Osservando il frame si può evincere che le informazioni non necessarie possono

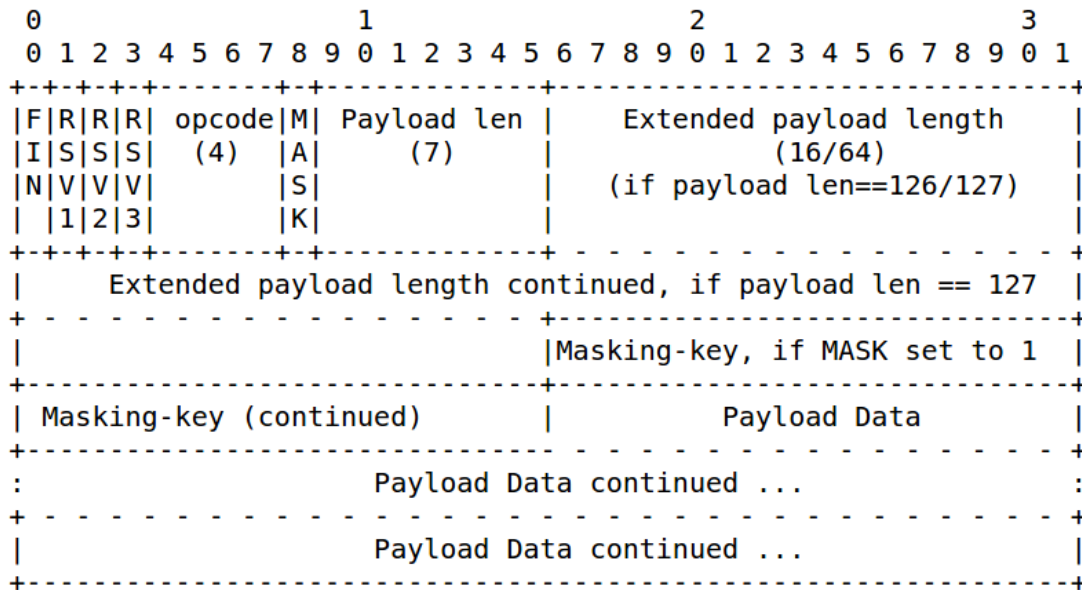


Figura 5.2: Frame WebSocket

variare da un minimo di 2 byte ad un massimo di 16 byte. Ecco lo scenario del traffico prendendo in considerazione il caso peggiore:

1. 1.000 client che ricevono un messaggio al secondo. il traffico equivale a: $16 \times 1.000 = 16.000 \text{ byte} = 128.000 \text{ bit al secondo} = 0,12 \text{ Mbps}$;
2. 10.000 client che ricevono un messaggio al secondo. il traffico equivale a: $16 \times 10.000 = 160.000 \text{ byte} = 1.280.000 \text{ bit al secondo} = 1,22 \text{ Mbps}$;

3. 100.000 client che ricevono un messaggio al secondo. il traffico equivale a: $16 \times 100.000 = 1.600.000 \text{ byte} = 12.800.000 \text{ bit al secondo} = 12,20 \text{ Mbps}$.

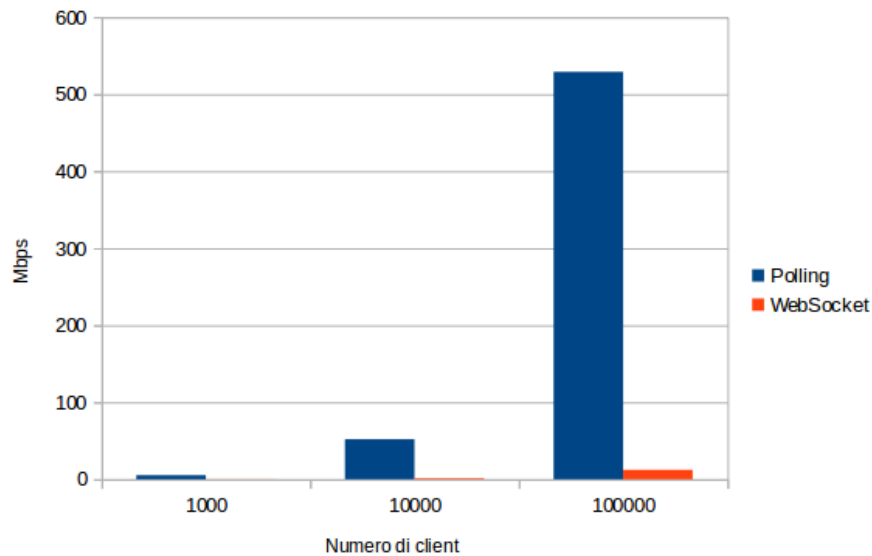


Figura 5.3: Traffico dati non necessario

Basta guardare la Figura 5.3 per rendersi conto dell'inutile mole di dati che viene generata se si realizza un'applicazione real-time con il polling.

Per quanto riguarda la latenza, l'aumento delle performance è illustrato in Figura 5.4. Se ipotizziamo che un pacchetto per arrivare dal browser al server impieghi 50 ms, l'applicazione con il polling genera molta latenza in più rispetto a quella implementata tramite WebSocket, perché per ogni risposta completa deve essere inviata una nuova richiesta.

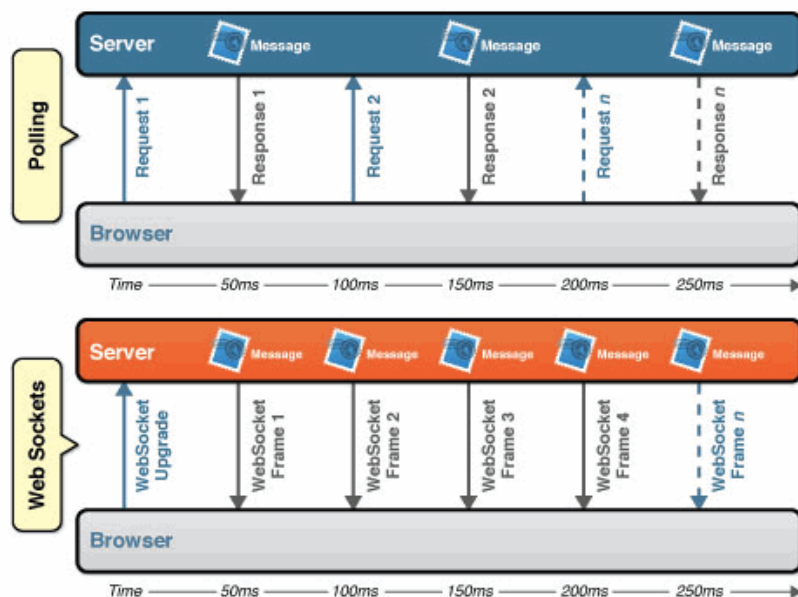


Figura 5.4: Latenza

5.2.4 Supporto dei browser

La specifica WebSocket ormai ha qualche anno, per la precisione, la versione standard definita nel RFC6455 è di dicembre 2011. Nonostante ciò, le WebSocket sono supportate solo nei browser più recenti.

Versione	Chrome	Firefox	Internet Explorer	Opera	Safari
76	6	4.0	No	11.00(disabilitato)	5.0.1
7	No	6.0	No	No	No
10	14	7.0	HTML5 Labs	?	?
RFC 6455	16	11.0	10	12.10	6.0

Tabella 5.1: Supporto dei browser

Versione	Android	Firefox Mob.	IE Mob.	Opera Mob.	Safari Mob.
76	?	?	?	?	?
7	?	?	?	?	?
10	?	7.0	?	?	?
RFC 6455	16(Chrome)	11.0	?	12.10	6.0

Tabella 5.2: Supporto dei browser Mobile

Nelle Tabelle 5.1 e 5.2 è possibile vedere, rispettivamente per desktop e per mobile, il supporto dei vari browser per le diverse specifiche delle WebSocket.

Quando si crea una WebApp con le WebSocket è importante prendere in considerazione anche la possibilità che esse non siano supportate dal client e quindi implementare anche una soluzione alternativa, per l'appunto, nel caso non sia possibile utilizzare le WebSocket. Esistono diversi framework che forniscono supporto completo sia per l'implementazione del client che del server che risolvono tutti i problemi di compatibilità che potrebbero presentarsi. Tra i framework più popolari troviamo:

- Kaazing;
- LightStreamer;
- Socket.io.

Nel paragrafo successivo vedremo in dettaglio come implementare sia il client ed il server di un'applicazione real-time con Socket.io

5.3 Socket.io

Socket.io permette di sviluppare applicazione client-server real-time, senza doversi preoccupare delle problematiche relative alle WebSocket. Ad esempio, se un client non supporta le WebSocket, socket.io implementerà lo stesso servizio con il polling, il tutto in maniera trasparente al programmatore.

In questa sezione scriveremo il codice di una chat, lo stesso proposto nel sito di Socket.io ma utilizzando Angular nel client, per farne vedere sia l'utilizzo lato frontend che backend.

5.3.1 Backend

Le uniche dipendenze di cui ha bisogno il nostro backend sono: `express` e `socket.io`. Il codice è molto semplice e non ha bisogno di particolari spiegazioni. Il server è in ascolto sulla porta 3000 sia di connessioni HTTP sia WebSocket. Se riceve una richiesta http risponderà con la WebApp in Angular, altrimenti, gestirà la WebSocket.

```

1 var express = require('express');
2
3 var app = express();
4 var http = require('http').Server(app);
5 var io = require('socket.io')(http);
6 var port = 3000;
7
8 app.use(express.static(__dirname + '/angular'));
9
10 io.on('connection', function(socket){
11   console.log('a_user_connected');
12
13   socket.on('disconnect', function(){
14     console.log('user_disconnected');
15   });
16
17   socket.on('chat_message', function(msg){
18     console.log('message:' + msg);
19     io.emit('chat_message', msg);
20   });
21 });
22
23 http.listen(port, function(){
24   console.log("Server_listening_on_port_" + port);
25 });

```

Codice 5.7: backend chat socket.io

L'unica cosa che fa il server è inviare in broadcast a tutte le socket connesse i messaggi che riceve dalle singole connessioni. Se non avessimo utilizzato Socket.io avremmo dovuto implementare soluzioni alternative nel caso in cui le WebSocket non fossero disponibili nel client o l'handshake non andasse a buon fine, in questo caso, invece, è Socket.io che si occupa di gestire tutti i possibili problemi e fornire una soluzione equivalente, in maniera sempre trasparente al programmatore.

5.3.2 Frontend

Il frontend, molto semplicemente, invierà i messaggi scritti dall'utente tramite Socket.io al server e visualizzerà a video i messaggi ricevuti, il tutto, real-time. Per utilizzare Socket.io come se fosse un servizio di Angular, abbiamo installato il plugin `angular-socket-io`, disponibile tramite bower e inizializzato come segue.

```

1 angular.module('socketApp.services.socket', [])
2 .factory('mySocket', function(socketFactory) {
3   var myIoSocket = io.connect();
4
5   mySocket = socketFactory({
6     ioSocket: myIoSocket
7   });
8
9   return mySocket;
10 });

```

Codice 5.8: service mySocket

A livello di controller basta iniettare il servizio come dipendenza e si possono utilizzare i listener e i metodi messi a disposizione da Socket.io. Nel codice che segue semplicemente se arriva un nuovo messaggio viene aggiunto all'array di messaggi.

```
1 angular.module('socketApp.controllers.home', [])
2 .controller('HomeCtrl', function($scope, mySocket) {
3   $scope.messages = [];
4   $scope.message = "Scrivi_qualcosa";
5
6   mySocket.on('chat_message', function(data) {
7     $scope.messages.push({text: data});
8   })
9
10  $scope.send = function() {
11    mySocket.emit('chat_message', $scope.message);
12    $scope.message = "";
13  };
14 });
```

Codice 5.9: controller mySocket

Questo è tutto, il codice completo dell'esempio è disponibile sul mio GitHub².

²<https://github.com/save91/angular-chat-socket.io>

6. Conclusioni

NOTA *** CONCLUSIONI RIGUARDO
IOT, API REST E BEACON - INTRO-
DUZIONE BMC CHE VERRÀ DESCRIT-
TO DA MARCO ***

A. Il codice

Bibliografia

- [App14] Apple. *Getting started with iBeacon*. Giu. 2014. URL: <https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf>.
- [Blu] BlueUp. *BlueUp - Pagina di supporto*. URL: <http://www.blueupbeacons.com/index.php?page=support>.
- [Dana] Radigan Dan. *Kanban*. URL: <https://www.atlassian.com/agile/scrum>.
- [Danb] Radigan Dan. *Scrum*. URL: <https://www.atlassian.com/agile/scrum>.
- [Ect16] Maarten Ectors. *The Future Of your Smart Home*. Set. 2016. URL: http://www.huffingtonpost.co.uk/maarten-ectors/the-future-of-your-smart-_b_11874856.html.
- [IET05] IETF. *A Universally Unique Identifier (UUID) URN Namespace*. Lug. 2005. URL: <https://tools.ietf.org/html/rfc4122>.
- [IET11] IETF. *The WebSocket Protocol*. Dic. 2011. URL: <https://tools.ietf.org/html/rfc6455>.
- [IET99] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. Giu. 1999. URL: <https://tools.ietf.org/html/rfc2616>.
- [JJN07] Fawcett Joe, McPeak Jeremy e C.Zakas Nicholas. *Ajax Guida per lo sviluppatore*. Milano: Hoepli, 2007.
- [Lei15] Norberto Leite. *Building your first application with MongoDB: Creating a REST API using the MEAN Stack*. Apr. 2015. URL: https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1?jmp=docs&_ga=1.200080494.1978830653.1465371972.
- [Mon] MongoDB. *MongoDB*. URL: <https://www.mongodb.com/nosql-explained?jmp=footer>.
- [Nod] Node. *Node*. URL: <https://nodejs.org/en/>.
- [PBF11] Lubbers Peter, Albers Brian e Salim Frank. *HTML5 Tecniche professionali*. Milano: Hoepli, 2011.
- [Swa] Swagger. *Swagger*. URL: <http://swagger.io/>.
- [Tre] Trello. *Trello*. URL: <https://trello.com/>.

Indice analitico

B		
BLE	22	
BlueBeacon	24	
C		
Comet	41	
D		
Database	30	
E		
Express	37	
I		
iBeacon	19	
J		
JSON	27	
K		
kanban	12	
L		
LAMP	27	
M		
MEAN	11, 27	
MongoDB	38	
R		
REST	13	
RSSI	21	
S		
Scrum	12	
Sprint	12	
Swagger	31	
T		
Trello	12	
U		
UUID	23	
X		
		XML
		27
		Y
		YAML
		31