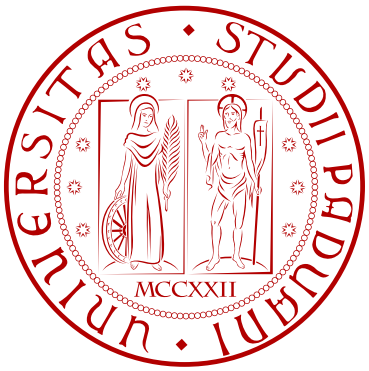
Università degli Studi di Padova

facoltà di ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica



Tesi di Laurea Magistrale

Un'architettura di database per la virtualizzazione di dispositivi embedded tramite web service

Laureando
Alberto Boccato

Relatore
Michele Zorzi

Correlatori
Nicola Bui
Moreno Dissegna

Anno Accademico 2012–2013

Ai miei genitori,
per avermi sostenuto
sempre
in questo mio percorso

Indice

Sommario

L'Internet of Things (IoT), con la sua crescente popolarità, ha portato ad un nuovo modo di comunicare attraverso Internet. L'interoperabilità tra dispositivi ha assunto un'importanza fondamentale, così come la possibilità di accedere e gestire in modo programmatico dispositivi, messaggi ed interfacce. Inoltre, la progettazione di hardware e protocolli efficienti, si scontra con la necessità di utilizzare componenti di rete aventi basso consumo energetico e limitate capacità di calcolo. La pila di protocolli per l'IoT proposta dagli organismi di standardizzazione, è una delle soluzioni più promettenti per le comunicazioni tramite dispositivi embedded e, in particolare, il protocollo Constrained Application Protocol (CoAP) consente di incorporare dei web service nei dispositivi che soddisfano questi requisiti. Da questa evoluzione nasce il bisogno di memorizzare, organizzare e reperire i dati provenienti dai dispositivi in modo semplice e scalabile, mediante un'interfaccia ad alto livello. In questo lavoro viene presentata un'architettura di database che permette di collegare assieme reti di sensori eterogenee, virtualizzandone i dispositivi e rendendoli accessibili ovunque attraverso il Web. Nella prima parte del lavoro infatti, è stato progettato un modello per mappare le informazioni associate ai dispositivi reali, in dispositivi virtuali all'interno nel database. Le risorse presenti nei dispositivi devono essere conformi al paradigma RESTful: in questo modo il modello dati risulta sufficientemente flessibile per gestire dispositivi generici, ed è abbastanza semplice per interfacciare risorse CoAP. Inoltre, è stato implementato un sistema di caching per limitare il numero di comunicazioni verso i nodi fisici, al fine di migliorarne l'efficienza energetica. Nella seconda parte, è stato sviluppato un web service che consente agli utenti di accedere ai dispositivi virtuali attraverso un'interfaccia REST. In questo contesto, sono stati tenuti in considerazione diversi aspetti di sicurezza, tra cui la gestione dei permessi sui dispositivi e l'autenticazione degli utenti che eseguono le richieste all'API server. Infine l'architettura è stata testata con dei reali sensori COAP, lasciando per il futuro la possibilità di renderla fruibile da altri utenti come software-as-a-service.

Capitolo 1

Introduzione

In questo capitolo vengono presentati alcuni concetti riguardanti le reti di sensori e l'Internet of Things in modo da contestualizzare il lavoro svolto. Successivamente viene descritto lo scopo della tesi, definendo i componenti chiave necessari alla realizzazione del progetto.

1.1 Internet of Things

Fino ad oggi, la maggior parte delle comunicazioni veicolate attraverso il Web, ha avuto come fonte l'interazione fra attori umani. Lo scenario sta però cambiando, e ci si sta addentrando in un era dove l'ambiente che ci circonda è pervaso da oggetti che comunicano e collaborano tra loro, l'era dell'*Internet of Things (IoT)* [1] [2]. Per la prima volta è infatti possibile interagire, non solo con le persone, ma anche con l'ambiente intorno a noi, e ricevere informazioni sul suo stato in un modo prima impensabile. Inoltre, questi oggetti "intelligenti" possono ora scambiarsi informazioni senza la necessità del diretto intervento umano, assumendo così un ruolo attivo e orientando sempre più la comunicazione verso il paradigma *machine to machine (M2M)*. Indubbiamente, il principale punto di forza di questa rivoluzione tecnologica, è il grande impatto che avrà in molti aspetti della vita di tutti i giorni. Domotica, sanità, logistica, ambienti intelligenti e *social networking* sono solo alcuni dei domini applicativi in cui gli effetti dell'IoT saranno visibili [1].

1.2 Wireless Sensor Networks

L'idea di Internet of Things pone nuovi problemi in termini comunicazione: gli oggetti, nell'accezione di IoT, sono infatti caratterizzati da poche risorse. Per *reti constrained* si intendono appunto reti in cui i nodi e il loro collegamento presenta delle limitazioni. Trattandosi tipicamente di dispositivi embedded di piccole dimensioni, tali

limiti riguardano la potenza di calcolo e la memoria del dispositivo. Inoltre, ci sono dei vincoli anche nella comunicazione, dato che spesso i nodi sono localizzati in reti wireless caratterizzate dal basso throughput e da un'alta probabilità di perdita dei pacchetti.

Un sottoinsieme delle reti constrained sono le reti di sensori o *Wireless Sensor Networks (WSN)*. Esse consistono di un insieme di dispositivi, detti nodi, il cui scopo è quello di monitorare determinati fenomeni, per esempio temperatura, luminosità, umidità e comunicare le informazioni raccolte attraverso un collegamento wireless ad una stazione centrale detta *gateway* o *sink node* [3]. Questi ultimi hanno in genere una capacità di calcolo e di memoria più elevata. Le reti possono essere composte da centinaia o migliaia di nodi, anche eterogenei tra loro e che potenzialmente possono usare stack diversi di protocolli per la comunicazione.

L'interconnessione di WSN ad Internet è stata studiata a lungo negli ultimi anni. All'inizio la ricerca era focalizzata su sistemi dedicati: protocolli non standard e altamente specializzati venivano usati all'interno delle reti, dove uno o più gateway si occupavano della traduzione dei messaggi e della connessione dei sensori al mondo esterno, che sfruttava il protocollo IP. Tali sistemi mancavano però di flessibilità, e lo sviluppo di nuove applicazioni su questa base era complesso e costoso. Per rimediare a questo, l'Internet Engineering Task Force ha recentemente proposto il protocollo 6LoWPAN che permette di portare il protocollo IPv6 nelle reti di sensori [4]. Questo comporta numerosi vantaggi tra cui la possibilità di interagire con i dispositivi per mezzo di web service come avviene nelle reti basate su IP.

1.3 Web service e architettura REST

Un web service è un sistema software che permette l'interazione tra diversi dispositivi attraverso la rete. Possiede un'interfaccia che può essere descritta attraverso un file, tipicamente in formato Web Service Description Language (WSDL), con il quale specifica l'insieme delle funzionalità offerte. Come si è detto i web service possono essere usati come base per la connessione dei sensori al web. Esistono tuttavia due tipologie di web service: SOAP (Simple Object Access Protocol) e REST (Representational State Transfer). I primi introducono però dei costi considerevoli in termini di performance in quanto prevedono connessioni di tipo stateful, e lo scambio e parsing di grandi file XML. Recentemente, REST è invece emerso come una alternativa leggera alla classica tipologia di web service, poiché non prevede la definizione di messaggi SOAP e di interfacce WSDL. L'aspetto più interessante del paradigma REST per quanto riguarda questa tesi, è il concetto di "risorsa". Qualsiasi informazione che possa essere referenziata da un identificatore, un Uniform Resource Identifier (URI) come specificato nell'RFC 3986 [5], può essere infatti considerata risorsa: ad esempio un documento, una immagine, la temperatura di una stanza, una collezione di altre risorse, ecc.

L'architettura REST prevede poi, che l'interazione con le risorse avvenga attraverso un insieme limitato di operazioni ben definite: nel caso di HTTP, ad esempio, queste sono GET, POST, PUT, DELETE. Il formato usato per la rappresentazione di una risorsa è chiamato *media type* ed in genere vengono usati, a questo scopo, linguaggi come XML oppure JSON.

REST risulta dunque adatto allo sviluppo di applicazioni per reti di sensori poiché le tipiche operazioni previste sui nodi, per esempio la loro abilitazione oppure la lettura di un dato, ben si adattano al concetto di risorsa previsto dal paradigma.

1.4 Dati e database

Con l'espandersi delle reti di sensori si è reso sempre più importante gestire le grandi quantità di dati generate. Tutte queste informazioni necessitano infatti di essere processate, archiviate e interpretate al fine di renderle utili agli utenti. La raccolta dei dati dalle reti di sensori prevede due diverse modalità: può essere subordinata ad una esplicita richiesta da parte di un client, oppure può essere un flusso periodico di dati dai dispositivi verso la base di dati. A questo scopo sono in via di sviluppo dei meccanismi di notifica [6] che permettono di eseguire una registrazione alle risorse per cui si vuole essere notificati. I dispositivi usati in questo progetto, tuttavia, non implementano ancora queste funzionalità dunque si sono sfruttati dei sistemi alternativi per ottenere l'aggiornamento periodico di un dato.

1.5 Scopo della tesi

Questa tesi è nata presso l'azienda Patavina Technologies, azienda specializzata nella progettazione e realizzazione di reti di sensori wireless e, più in generale, di sistemi di comunicazione e sistemi embedded. Obbiettivo di questo lavoro è stato quello di progettare e implementare una piattaforma che permette di virtualizzare e rendere accessibili ovunque, attraverso il Web, le reti di sensori di Patavina Technologies. L'azienda dispone infatti di dispositivi resource-constrained che comunicano mediante il protocollo CoAP. Ciascuna rete può essere composta da centinaia di nodi e produrre grandi quantità di dati eterogenei. Da tale situazione, è nata la necessità di memorizzare, organizzare e reperire i dati provenienti dai dispositivi in modo semplice e scalabile, attraverso un servizio web accessibile da qualsiasi posizione. La realizzare di questo software ha previsto l'implementazione di diversi componenti:

- un simulatore di dispositivi CoAP, necessario per testare passo dopo passo le funzionalità della piattaforma in modo rapido e indipendente dai dispositivi fisici.
- un gateway che rappresenta, per i dispositivi non direttamente connessi ad Internet, il punto di accesso alle reti locali in cui questi sono dislocati. Il suo

compito è quello di inoltrare i messaggi provenienti dai dispositivi verso il database e viceversa

- una architettura di database che permette di mappare le informazioni associate ai dispositivi reali, in dispositivi virtuali. Oltre a questo, sono previste una serie di altre funzionalità che permettono di creare una piattaforma multiutente per la gestione dei dispositivi. Per esempio si vuole dare la possibilità all'utente di effettuare delle ricerche tra le risorse e i dispositivi registrati, memorizzare lo storico dei valori ricevuti dai sensori, abilitare notifiche al verificarsi di particolari condizioni sui dispositivi ecc.

- un RESTFul web service che, appoggiandosi al database, fornisce delle API di alto livello per interagire con i dispositivi virtuali. Per API si intende una interfaccia accessibile via HTTP che, seguendo il paradigma client-server, permette di accedere ai servizi offerti dal server. Tale server dispone quindi di 2 interfacce: una verso gli utenti, con cui comunica attraverso messaggi XML o JSON, e una verso i dispositivi con i quali interagisce tramite il protocollo CoAP. La parte di comunicazione con i nodi fisici è comunque facilmente estendibile anche ad altri protocolli, purché rispettino il paradigma REST. Nell'implementazione del web service inoltre, si sono tenuti in considerazione diversi aspetti di sicurezza, tra cui l'autenticazione e la crittografia di ogni richiesta REST in entrata, e lo studio di un sistema di permessi che possa permettere agli utenti in un futuro di condividere i propri dispositivi.

Capitolo 2

Stato dell'arte

2.1 Il protocollo CoAP

Il Constrained RESTful Environments (CoRE) Working Group sta creando il protocollo CoAP (Constrained Application Protocol) [7] allo scopo di fornire un framework per le reti contrained che aderisca al paradigma REST, e quindi sia adatto ad applicazioni orientate al concetto di risorsa. É stato progettato infatti, per reti caratterizzate da alta probabilità di perdita di pacchetti e dispositivi a basso consumo energetico, con i quali voglia interagire remotamente attraverso una architettura client-server di tipo REST. Si interfaccia in modo semplice con HTTP, dato che mantiene lo stesso metodo di interazione ma implementa un sottoinsieme dei suoi metodi. Non si tratta tuttavia di una copia compressa di HTTP, in quanto il protocollo è stato ridisegnato per essere più semplice. Si appoggia ad UDP anziché a TCP e necessita dunque di un substrato tra il livello trasporto e il livello applicazione che si occupi di garantire connessioni affidabili. Prevede inoltre overhead molto bassi, multicasting e servizi di resource-discovery che lo rendono particolarmente adatto in applicazioni M2M.

2.1.1 Struttura dei messaggi

Un pacchetto CoAP che rispetti la versione 14, quella a cui si farà riferimento in questa tesi, è così strutturato:

- un header con dimensione fissa di 4 byte così composto:
  - i primi 2 bit per la versione del protocollo;
  - i successivi 2 bit servono per indicare il tipo di pacchetto che si manda. I tipi supportati sono 4: confirmable, non confirmable, ack e reset;

  - 4 bit sono usati per indicare la lunghezza variabile del campo token. Attualmente sono supportate lunghezze da 0 a 8 byte.
  - 8 bit vengono usati per indicare il tipo di risposta (1-31) e di richiesta (64-191);
  - i rimanenti 16 bit formano il message ID. Esso è un numero necessario per associare un pacchetto al relativo ack, e identificare eventuali pacchetti duplicati.

- un campo token la cui dimensione varia da 0 ad 8 byte e che serve per correlare richieste e risposte;
- eventuali opzioni. Una opzione può essere seguita da un'altra opzione, dalla fine del messaggio oppure dal payload;
- eventuale payload che, se presente, ha come prefisso un payload-marker (0xFF) di un byte;

L'affidabilità della connessione viene gestita attraverso il tipo del messaggio: un pacchetto confirmable richiede infatti la conferma della ricezione attraverso un ack, a differenza di un pacchetto non confirmable. Le risposte ad una particolare richiesta possono poi essere inserite all'interno di un ack di risposta (piggy-backed response) oppure inviate in un pacchetto separato (separate response), nel caso il server interrogato non disponga in quel momento del dato.

L'URL della risorsa richiesta viene specificata mediante le opzioni `Uri-Host`, `Uri-Port`, `Uri-Path`, `Uri-Query`. I primi 2 indicano il nome del server e la porta su cui la richiesta deve essere inviata, l'`Uri-Path` è invece una opzione ripetibile che rappresenta le singole porzioni di cui si compone l'URL. `Uri-Query` specifica infine eventuali parametri aggiuntivi della richiesta.

2.1.2 CoRE Link Format

L'architettura REST è stata riconosciuta dai gruppi di standardizzazione come la più promettente nell'ambito delle reti constrained. In particolare la fase di discovery delle risorse disponibili all'interno di un server è molto importante nelle applicazioni machine-to-machine, dove non è presente l'intervento umano. La funzione principale del meccanismo di discovery è quella di fornire degli Universal Resource Identifiers (URIs) che identificano le risorse presenti nel server, specificando per ciascuna di queste le proprietà ed eventualmente le relazioni con altre risorse. La soluzione proposta nel RFC 6690 [8] dal CoRE Working Group, prevede che questa collezione di URIs, chiamata CoRE Link Format, sia fornita dal server come una propria risorsa. Richiedendo la particolare URI relativa `/.well-known/core` è infatti possibile ottenere nel payload della risposta la descrizione delle risorse presenti nel server.

I principali casi d'uso sono dunque quelli legati al discovery e alla descrizione delle risorse. Si pensi ad un dispositivo che possiede la lista di allarmi di una abitazione: in tal caso il CoRE Link Format può essere usato per identificare il punto di accesso alla collezione di risorse e scorrere i suoi membri.

Core Link Format prevede la definizione di una serie di attributi per ciascuna risorsa. Fra questi, quelli principali sono:

- **title**: quando presente è usato per etichettare la destinazione del link e quindi la risorsa, in modo comprensibile da parte di un essere umano. Questo parametro può apparire una sola volta e le ulteriori occorrenze devono essere ignorate dai parser;

- **type**: dovrebbe indicare il tipo di dato contenuto della risorsa associata. Si tratta comunque di un suggerimento, infatti questo attributo non va a sovrascrivere l'header Content-Type della risposta HTTP;

- **rt**: l'attributo "Resource Type" viene usato per assegnare una semantica, specifica per una data applicazione, alla risorsa. Nel caso di un sensore di temperatura questo potrebbe essere, per esempio, "outdoor-temperature". Si tratta dunque di una sorta di tag per la particolare risorsa e ce ne può essere più d'uno;

- **if**: l'attributo "Interface Description" si usa per definire una interfaccia con cui interagire con la risorsa. Si può pensare come la descrizione delle azioni, o meglio i metodi dell'interfaccia REST, usabili per interagire con la particolare risorsa. Ci si aspetta che la stessa interfaccia venga riusata da diversi "Resource Type". Per esempio le risorse il cui attributo "Resource Type" corrisponde a "outdoor-temperature", "indoor-temperature", "rel-humidity" potrebbero essere tutte accessibili mediante l'interfaccia `http://www.example.org/myapp.wadl#sensor`. L'attributo "Interface Description" può essere infatti una URI dove è presente questa descrizione in un particolare linguaggio, per esempio Web Application Description Language (WADL);

- **sz**: indica la massima dimensione stimata, ottenibile eseguendo un metodo GET sulla specifica URI. Non ci si aspetta che questo attributo sia incluso in risorse di piccole dimensioni e che possano essere trasportate all'interno di una singola Maximum Transmission Unit (MTU).

Il lavoro presentato in questa tesi è compatibile con questa specifica e richiede, ai fini di registrare i dispositivi in modo automatico al sistema, che questi implementino il Core Link Format fornendo perlomeno la lista delle risorse e il loro attributo "title". Inoltre, questa specifica è applicabile sia al protocollo CoAP che a qualsiasi altro protocollo che rispetti il paradigma REST, per esempio HTTP. Viene presentato ora un esempio di richiesta del file Core Link Format:

```
REQ: GET /.well-known/core

RES: 2.05 Content
</sensors>;title="Sensor Index",
</sensors/temp>;title="temperature";rt="outdoor-temperature";type
    =40,
</sensors/light>;title="light";rt="indoor-light";type=40
```

2.2 Altre piattaforme per l'IoT

Vengono presentate in questa sezione i software presenti sul mercato che mirano ad obbiettivi simili a quelli presentati in questo progetto di tesi.

2.2.1 iDigi

La piattaforma *iDigi* [9] è una soluzione proprietaria, e quindi a pagamento, basata sul cloud per gestire dispositivi di rete. Supporta molte tipologie di dispositivo, da quelli connessi via cavo, a quelli dotati di connessione cellulare o satellitare. *iDigi* commercializza anche dispositivi proprietari che si interfacciano nativamente con la piattaforma, per tutti gli altri è invece necessario installare un componente software chiamato *iDigi Connector*.

2.2.2 Cosm

Cosm [10] è una piattaforma sicura e scalabile che permette di connettere ad Internet dispositivi e più in generale flussi di dati. L'implementazione ruota infatti attorno al concetto di *feed*, ossia un ambiente in cui confluiscono più flussi di dati detti *datastream*. Questi possono provenire da dispositivi e sensori, oppure da altre applicazioni: per esempio è supportata l'integrazione con i post di Twitter. *Cosm* mette a disposizione delle API per permettere a ciascun sviluppatore di creare la propria applicazione. É fruibile gratuitamente a patto di rispettare dei vincoli in termini di frequenza di richieste alle API.

2.2.3 Nimbits

Nimbits [11] è un software open source per la registrazione di serie storiche di dati nel cloud. Mette a disposizione una interfaccia REST per inviare i dati al sistema in formato JSON oppure XML. Sui valori ricevuti permette poi di eseguire calcoli, statistiche e programmazione di notifiche. Oltre a questo fornisce una interfaccia grafica per visualizzare grafici e informazioni sui dati raccolti. Nimbits è predisposto per essere inserito all'interno di una architettura cloud come Google App Engine, Amazon EC2

oppure qualsiasi altra che possa ospitare una macchina virtuale basata su Ubuntu Linux KVM.

2.2.4 Paraimpu

Paraimpu [12], a differenza delle altre piattaforme è maggiormente orientato all'aspetto *social*. É un software che permette agli utenti di connettere, usare, condividere e comporre dispositivi ma anche servizi, al fine di creare applicazioni personalizzate.

2.2.5 Open.sen.se

Open.sen.se è l'ultimo tra i software che sono stati presi come modello per lo sviluppo di questo progetto. Fornisce le stesse funzionalità messe a disposizione da *Cosm* anche se cambiano i nomi con cui i dispositivi vengono virtualizzati nel sistema. In *Cosm* si ha una suddivisione *feed*>*datastream*>*datapoint*, in *Open.sen.se* le stesse entità sono chiamate *device*>*feed*>*events* [13]

2.3 Architetture database simili

2.3.1 REnvDB

Anche in letteratura sono stati presentati dei modelli di database attinenti agli scopi di questa tesi. É il caso per esempio di REnvDB [14], un database dotato di una interfaccia REST specificamente progettato per applicazioni di monitoraggio di reti di sensori. Nel caso specifico, il software è applicato all'ambito agricolo, dove reti di sensori che controllano temperatura, umidità e luminosità sono state dislocate nei terreni da monitorare. L'architettura usata è molto simile a quella che si vuole implementare con questo progetto di tesi. I sensori comunicano infatti con un gateway, il quale inoltra le informazioni ricevute al database dove vengono memorizzate. Il database può poi interagire con un web server che, attraverso una interfaccia REST, permette all'utente di accedere ai dati osservati. La comunicazione con i sensori è limitata però al protocollo HTTP, vincolo troppo stringente per i sensori di Patavina Technologies che comunicano invece mediante CoAP. Inoltre questo progetto di tesi vuole fornire degli ulteriori servizi avanzati che permettano per esempio di gestire notifiche, allarmi e prevedere la creazione di applicazioni da parte degli utenti.

Capitolo 3

Architettura del sistema

In questo capitolo vengono descritte le linee guida seguite in fase di progettazione del software e viene data una visione d'insieme del sistema che si andrà a realizzare, descrivendo inoltre gli strumenti usati per l'implementazione

3.1 Linee guida di progetto

Durante la fase iniziale di analisi sono state definite un insieme di linee guida da seguire durante l'intero sviluppo del software. In linea generale, i principi a cui si è cercato di attenersi sono:

- Modularità: la possibilità di aggiungere nuove funzionalità al sistema, senza doverne modificare la struttura, è uno dei requisiti fondamentali. Trattandosi di un progetto appena nato, infatti, è difficile immaginare fin da subito tutte le funzioni che il sistema può offrire. Inoltre, il fatto di mantenere separati fisicamente componenti che, dal punto di vista logico, svolgono funzioni diverse, permette di ottenere una maggiore mantenibilità del software. Per raggiungere questi obbiettivi si è usato Spring Framework (vedi 3.4.4), il quale, per sua natura, facilita il programmatore obbligandolo a suddividere l'applicazione, a livello logico, in diversi strati, ciascuno dei quali ha un compito diverso;

- Scalabilità: è importante che il sistema mantenga inalterate le proprie performance all'aumentare degli utenti e dei dispositivi registrati. Un maggior numero di sensori e di utenti implica una maggiore quantità di dati da memorizzare, oltre ad un maggior numero di richieste da servire. Per questo motivo si è progettato il software in modo da massimizzarne il parallelismo ove possibile: a livello di API server le richieste in entrata vengono gestite da thread separati (vedi 5.3), e per l'accesso ai dati è prevista una gestione concorrente delle connessioni, in modo

che più processi possano eseguire contemporaneamente la propria transazione nel database (vedi 5.5).

- Flessibilità: come accennato precedentemente il sistema è stato progettato per lavorare con dispositivi che comunicano mediante il protocollo COAP, ma deve essere facilmente estendibile ad altri protocolli, per esempio HTTP. Per questo motivo tutte le scelte fatte in fase di implementazione mirano a mantenere il software quanto più possibile svincolato dalla particolare tipologia di protocollo utilizzato.

3.2 Struttura di sistema

Nel paragrafo 1.5 si è descritto in linea generale lo scopo della tesi. Verrà ora presentata più dettagliatamente la struttura del sistema, così come è stata concepita durante la fase di analisi.

Si vuole realizzare una piattaforma che permetta di interconnettere reti di sensori diverse ad Internet, virtualizzando e rendendo accessibili ovunque i singoli dispositivi attraverso una semplice interfaccia web. Dopo un'attenta fase di analisi e ricerca (vedi capitolo 2), si è deciso che la strada migliore per raggiungere gli obiettivi prefissati fosse quella di implementare un web service che fornisca un insieme di API pubbliche, con le quali accedere a tutte le funzionalità del sistema. Il software deve attenersi all'architettura REST e la comunicazione deve avvenire attraverso semplici richieste HTTP, il cui contenuto sarà formattato in JSON oppure XML. Attraverso un'architettura di questo tipo, qualsiasi sviluppatore è libero di creare la propria applicazione sfruttando le API messe a disposizione.

Alla base dell'API server è presente un database che memorizza, organizza e rende disponibili i dati provenienti dalle diverse reti. Per raggiungere tale scopo viene introdotto il concetto di virtualizzazione dei dispositivi. Virtualizzare una rete di sensori significa mappare i nodi reali in entità che vivono all'interno della piattaforma, e quindi all'interno del database. Questo prevede innanzitutto una registrazione di queste entità al sistema, e la creazione di un livello di astrazione che mascheri all'utente finale i dettagli della comunicazione con i sensori. Dal punto di vista dell'utilizzatore del framework infatti, richiedere un particolare dato attraverso le API pubbliche deve essere equivalente alla richiesta dello stesso dato all'interno della rete locale, in presenza del sensore stesso. Come questo mapping viene effettuato verrà descritto nel capitolo 4. Inoltre, la base di dati deve permettere di gestire tutte le funzionalità che l'API server mette a disposizione: dalla fase di autenticazione delle richieste REST, all'interrogazione di valori monitorati dai sensori, fino all'impostazione di allarmi che possano notificare il verificarsi di particolari eventi.

Per quanto riguarda la parte di comunicazione tra device fisici e API server, è necessario permettere ai dispositivi di interfacciarsi al sistema mediante il protocollo CoAP sopra ad UDP, mantenendo il sistema abbastanza flessibile da prevedere in futuro l'uso di altri protocolli, ad esempio HTTP. Come detto nella sezione 2.1 i device espongono le proprie risorse attraverso un URI che le identifica univocamente. Il sistema prevede che essi usino come metodo di indirizzamento IPv4 oppure IPv6, e che possano essere direttamente connessi al web oppure passare attraverso un gateway.

3.3 Suddivisione logica e fisica

Dal punto di vista logico, il software può essere suddiviso in diversi componenti, ciascuno dei quali ha una diversa funzione, così come è presentato in figura 3.1. Al livello più basso si trova il *Sensors Component*, composto da tutti quei dispositivi che si occupano di raccogliere le informazioni nelle reti locali, ossia sensori e gateway. Il *Core Component*, è costituito dell'API server: esso si occupa di tutte le elaborazioni del sistema, per esempio autenticazione, gestione delle notifiche, ecc. interagendo, ove serve, con le altre parti del sistema. Il *Data Component*, è adibito alla memorizzazione di tutte le informazioni necessarie al funzionamento del software, e in particolar modo quelle necessarie alla virtualizzazione dei sensori. Infine il *Presentation Component* implementa l'interfaccia utente ed è costituito dai client della piattaforma. Poiché si fornisce all'utente un API server, il compito di programmare applicazioni per accedere ai servizi offerti dal sistema è demandato agli sviluppatori di terze parti. Eventualmente è possibile prevedere lo sviluppo di una applicazione web proprietaria di Patavina Technologies che vada ad inserirsi in questo componente. Una suddivisione logica così fatta, permette di dislocare i componenti del sistema su macchine diverse, anche quando non strettamente necessario: *Data Component* e *Core Component* potrebbero per esempio essere implementati sullo stesso server. Il diagramma UML di deployment in figura 3.3 mostra nel dettaglio come sono dislocati questi componenti e come comunicano.

Il cuore del framework si trova dunque nel *Core Component*. Esso comunica con tutti gli altri ed è a sua volta suddiviso, a livello logico, in diversi moduli [1] come mostrato in figura 3.2. L'*Access Module* fornisce un'interfaccia per l'accesso degli utenti alla piattaforma. Il *Processing Module* costituisce il nucleo del software: interoperando con gli altri componenti, permette infatti di realizzare la logica del sistema. Al *Communication Module* vengono demandati tutti i compiti di comunicazione da e verso i sensori. Infine il *Data Component* si occupa dell'accesso ai dati, gestendo le connessioni al database. Per una discussione più dettagliata di questi moduli si rimanda al capitolo 5.

[1] Nella definizione di questi moduli si è mantenuta la stessa nomenclatura usata in [15].

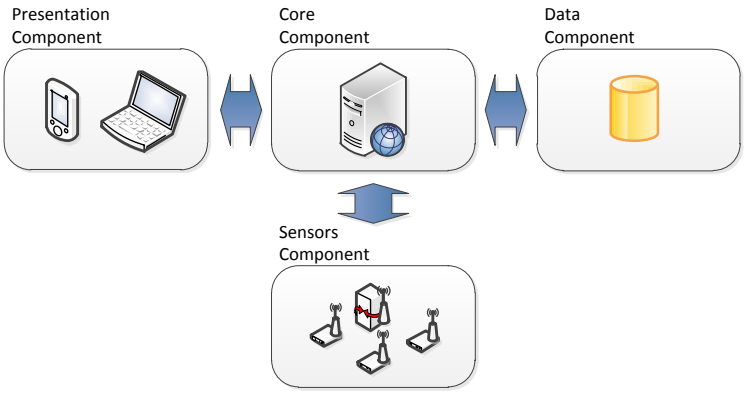


Figura 3.1: Suddivisione logica del sistema sulla base delle funzionalità dei vari componenti.

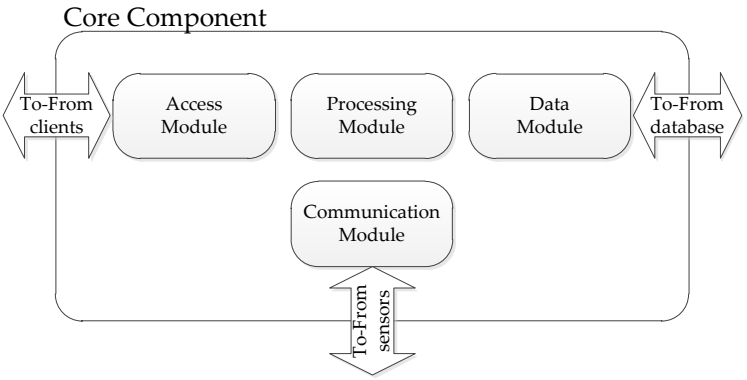


Figura 3.2: Suddivisione a livello logico del Core del sistema, ossia dell'API server.

3.4 Strumenti usati

3.4.1 JCoAP

Per creare i simulatori e gestire la comunicazione dall'API server verso i sensori è stato necessario realizzare dei server e dei client CoAP in Java. Per far questo senza dover implementare da zero il protocollo, si è usata JCoAP, una libreria che implementa appunto il protocollo CoAP fornendo le seguenti funzionalità:

- comunicazione affidabile e non affidabile;
- supporta entrambi i tipi di risposta definiti dal protocollo: *piggy-backed* ossia risposta inserita all'interno dell'ACK, oppure *separate* cioè quando l'ACK è vuoto e la risposta vera e propria viene inviata separatamente;
- semplicità d'uso poiché fornisce tutti i blocchi fondamentali per creare server e client;
- implementa le funzionalità di proxy (CoAP-CoAP, CoAP-HTTP, HTTP-CoAP)

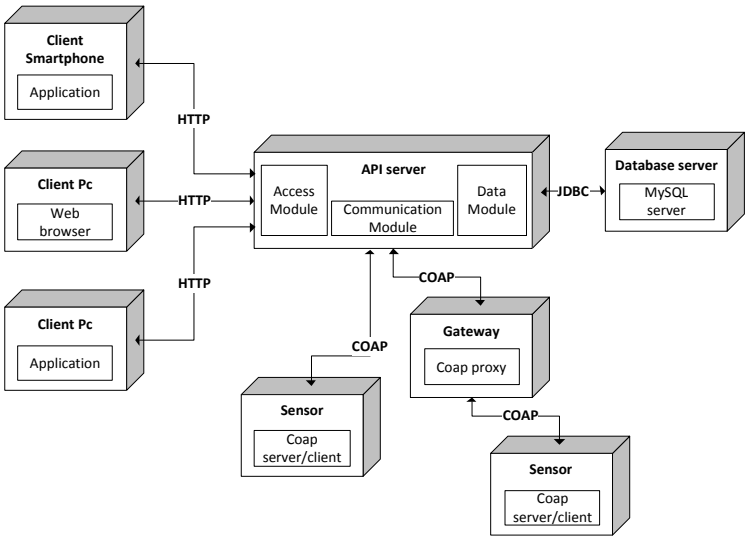


Figura 3.3: Il diagramma UML di deployment e come sono suddivisi dislocati i componenti logici all'interno di quelli fisici.

3.4.2 MySQL

MySQL è un sistema per la gestione di basi di dati relazionale (RDBMS - Relational database management system) disponibile sia per sistemi operativi Windows che Linux. Questo database supporta gran parte della sintassi standard SQL e fruisce di varie interfacce per diversi linguaggi di programmazione attualmente esistenti. In particolare per quanto riguarda il linguaggio Java, è possibile interfacciarsi al database per mezzo dei driver JDBC, messi a disposizione dalla libreria MySQL Connector/J. Si è scelto di usare MySQL in quanto open source, ben documentato e compatibile con la piattaforma di Hibernate. Inoltre il suo motore di salvataggio dati InnoDB fornisce il supporto alle transazioni. Per transazione si intende una unità elementare di lavoro che adotta una politica *all-or-nothing*, ossia o viene completata interamente oppure non ha alcun effetto. Si tratta di una tecnica che permette di mantenere il database consistente anche in caso di guasti: se la transazione non va a buon fine, tutte le modifiche eseguite fino a quel momento vengono eliminate. Inoltre un'altra proprietà importante delle transazioni è quella di fornire l'isolamento: transazioni diverse non interferiscono infatti le une con le altre.

3.4.3 Hibernate

Hibernate è un framework Java open source che fornisce un servizio di Object Relational Mapping, ovvero gestisce la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java. La caratteristica principale di Hibernate è quella di mappare le classi Java nelle tabelle di un database relazionale. Sulla base di

questa associazione viene gestito il salvataggio degli oggetti di tali classi sul database. Inoltre permette di astrarre ed automatizzare le procedure per le operazioni CRUD (Create, Read, Update, Delete), sollevando il programmatore dal compito di inserire codice SQL nell'applicazione, in quanto questo viene generato on-the-fly e direttamente eseguito. Altra importante caratteristica è il fatto di essere completamente indipendente dall'architettura di database utilizzata. Attraverso un file di configurazione è possibile specificare infatti tutti i dettagli relativi alla connessione al database.

Per quanto riguarda il mapping tra classi Java e tabelle del database, le classi devono rispettare delle convenzioni, in particolare devono essere dei `JavaBean`. Un `JavaBean` è una classe che rispetta i seguenti vincoli:

- la classe deve avere un costruttore senza argomenti;
- le sue proprietà devono essere accessibili usando get, set utilizzando una convenzione standard per i nomi;
- la classe dovrebbe essere serializzabile (capace di salvare e ripristinare il suo stato in modo persistente);

Il collegamento logico tra `JavaBean` e tabelle del database viene poi gestito per mezzo di annotazioni, ossia delle istruzioni per il compilatore precedute dal simbolo "@", inserite all'interno del `JavaBean`. In figura 3.1 viene presentato un esempio di questo mapping:

- `@Entity` specifica il fatto che la classe deve essere mappata in una entità del database;
- `@Table` con i suoi attributi definisce il nome del database detto anche catalogo e il nome della tabella a cui la classe deve fare riferimento;
- `@Column`, precedendo i metodi get specifica a quale colonna della tabella deve essere associato il campo della classe Java in questione;
- `@Id`, `@GeneratedValue` descrivono il fatto che l'id viene generato automaticamente dal database;
- `@ManyToOne` e `@JoinColumn` sono esempi di come vengono definite le relazioni con gli altri `JavaBean`, e dunque con le altre tabelle del database;

Codice 3.1: Esempio di come una classe viene mappata in una tabella del database

```
1 @Entity
2 @Table(name = "data_entry", catalog = "iot")
3 public class DataEntry implements java.io.Serializable {
4
5  private Long id;
```

```
6   private Resource resource;
7   private byte[] value;
8   private Date valueTimestamp;
9
10  public DataEntry() {
11  }
12
13  @Id
14  @GeneratedValue(strategy = IDENTITY)
15  @Column(name = "id", unique = true, nullable = false)
16  public Long getId() {
17   return this.id;
18  }
19
20  public void setId(Long id) {
21   this.id = id;
22  }
23
24  @ManyToOne(fetch = FetchType.LAZY)
25  @JoinColumn(name = "resource_id", nullable = false)
26  public Resource getResource() {
27   return this.resource;
28  }
29
30  //...
31 }
```

3.4.4 Spring Framework

Spring è un framework open source nato con l'intento di semplificare lo sviluppo di applicazioni java di tipo enterprise [16]. Si tratta di un framework "leggero" e grazie alla sua architettura estremamente modulare è possibile utilizzare solo le componenti di cui si necessita. Per comprendere in che modo il framework renda più semplice e rapida l'implementazione, è necessario introdurre i concetti di *Inversion of Control (IoC)* e *Dependency Injection(DI)*.

Per Inversion of Control si intende un pattern di programmazione, secondo il quale si tende a tener disaccoppiati i singoli componenti di un sistema, in cui le eventuali dipendenze non vengono definite all'interno del componente stesso, ma gli vengono iniettate dall'esterno. Nella programmazione tradizionale il flusso logico è definito esplicitamente dallo sviluppatore, che si occupa tra le altre cose di tutte le operazioni di creazione, inizializzazione ed invocazione dei metodi degli oggetti. Nell'IoC invece, questo flusso viene invertito, essendo un componente esterno a prendersi carico di questi compiti. La Dependency injection è una delle tecniche con le quali si può attuare l'IoC prendendo il controllo su tutti gli aspetti di creazione degli oggetti e delle loro dipendenze. Le principali tipologie di Injection sono:

- Constructor Injection, dove la dipendenza viene iniettata tramite l'argomento del costruttore;
- Setter Injection, dove la dipendenza viene iniettata attraverso un metodo "set";

Spring usa molto diffusamente la Dependency Injection attraverso la definizione di `beans` e l'uso dell'annotazione `@autowired`, con il risultato di eliminare dal codice applicativo ogni logica di inizializzazione. Il termine `bean`, all'interno di Spring, sta ad indicare un componente del sistema ossia una qualsiasi classe Java, da non confondere con i `JavaBean` definiti precedentemente. In Spring l'utente deve specificare un file XML dove è definita la configurazione di tutti questi componenti. In particolare tale file ha lo scopo di:

- creare i `bean` necessari all'applicazione
- inizializzare le loro dipendenze attraverso l'utilizzo dell'injection
- gestirne l'intero ciclo di vita

Nel listato 3.2 è riportato un esempio di scheletro del file di configurazione.

Codice 3.2: Esempio di file di configurazione di Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="..." class="...">
  <!-- eventuali risoluzioni di dipendenze e proprietà -->
</bean>

<bean id="..." class="...">
  <!-- eventuali dipendenze e proprietà -->
</bean>

<!-- altri bean applicativi -->
```

Dopo l'intestazione del file XML, racchiuse tra i tag `<beans>`, si trovano le definizioni dei `bean` e, per ognuno di questi, le eventuali proprietà che ne descriveranno la struttura e il comportamento.

Altro motivo importante per cui si è scelto di usare Spring framework è il supporto che viene fornito per la realizzazione di web service di tipo REST. Spring implementa infatti il pattern Model View Control (MVC) che è fortemente orientato alle richieste e permette di separare la logica di presentazione, dai dati e dalla logica di business. In figura 3.4 si può vedere il flusso di elaborazione di una richiesta.

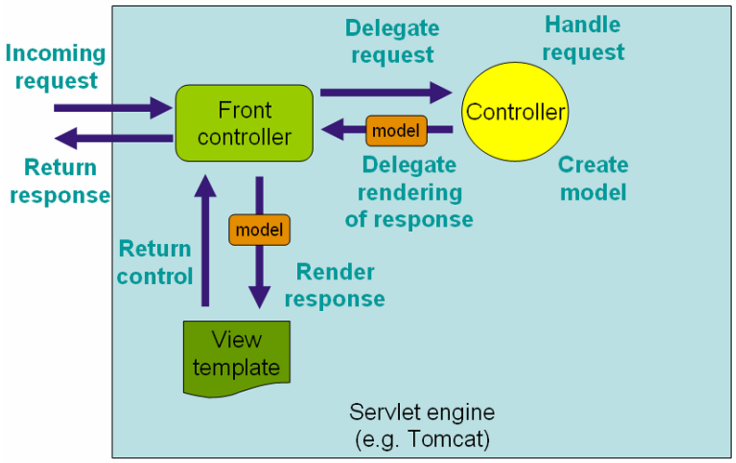


Figura 3.4: Flusso di elaborazione di una richiesta da parte di Spring MVC.

Questo modello è progettato sulla base di una servlet chiamata `DispatcherServlet` che agisce da "front controller", ossia smista le richieste HTTP in ingresso sulla base della loro URI ad altri Spring MVC controller. Un controller è un componete che si occupa di elaborare le richieste. Una tipica applicazione può avere però molti controller e la `DispatcherServlet` ha bisogno di uno strumento chiamato "handler mapping" per decidere, sulla base della URL trasportata nella richiesta, dove indirizzarla. Quando la richiesta giunge al controller, ne viene fatto il parsing e viene inoltrata a dei componenti denominati service, che implementano la logica di business. Generalmente, le operazioni eseguite prevedono poi la restituzione di qualche dato al client. Tali informazioni sono rappresentate mediante un oggetto "model". Ma ritornare all'utente dei risultati in formato grezzo non è sufficiente. Per questo motivo, le informazioni devono essere passate ad una vista ("view") che le presenta in un formato user-friendly come HTML. Nel caso di questo progetto, volendo implementare un web service, le risposte verranno presentate in formato XML oppure JSON.

3.5 Simulatori

Prima di passare all'analisi approfondita di come sono stati realizzati database e API server, descriviamo brevemente come è stata realizzata la comunicazione fra API server e i dispositivi CoAP, rimandando a [15] per i dettagli tecnici.

3.5.1 Device

Il primo passo per la realizzazione del software è stato quello di implementare un simulatore di device CoAP che permettesse di testare passo dopo passo le funzionalità della piattaforma in modo rapido, senza la necessità di disporre di una moltitudine

di dispositivi reali. Il simulatore è stato implementato in Java, sfruttando la libreria JCoAP che permette di creare un server CoAP ed esporre, come previsto dal protocollo, un insieme di risorse per mezzo di un web service. La definizione delle risorse avviene tramite un file nel formato Core Link Format (come già spiegato nel paragrafo 2.1.2), il quale deve specificare per ciascuna di esse almeno la proprietà "title", che ne identifica il nome, e la URI associata.

3.5.2 Gateway

Altro tassello importante del sistema è il Gateway, ossia quel componente che si interpone tra la rete di sensori e l'API server. Si è discusso molto sulle funzioni che questo componente avrebbe dovuto ricoprire. In particolare le alternative prese in considerazione erano le seguenti:

- implementare un gateway passivo il cui unico scopo fosse quello di inoltrare le richieste provenienti dall'API server verso la rete di sensori e viceversa. Questo implica che l'API server deve conoscere il particolare protocollo con cui i sensori comunicano per poter forgiare le richieste. Inoltre nel caso di dispositivi direttamente connessi ad Internet la sua presenza non sarebbe indispensabile;

- spostare parte della complessità del sistema nel gateway. Per esempio si era pensato di sfruttare il protocollo HTTP dall'API server fino al gateway ed assegnare a quest'ultimo il compito di tradurre dal protocollo HTTP allo specifico protocollo usato dalla rete di sensori, come può essere CoAP. In questa situazione la presenza di un gateway sarebbe sempre necessaria e sarebbe inoltre possibile implementare funzionalità aggiuntive, come l'aggregazione di richieste da inviare verso l'API server ecc.

Alla fine la scelta è ricaduta sulla prima soluzione in quanto non si vuole vincolare il sistema alla presenza di un gateway, permettendo per esempio ad un dispositivo CoAP di comunicare direttamente con l'API server.

Capitolo 4

Database

Il cuore di questo lavoro di tesi è la progettazione e l'implementazione della base di dati relazionale sottostante all'API server, in cui le informazioni sui dispositivi vengono mappate. La metodologia di progettazione a cui si è fatto riferimento è descritta in [17] e si articola in tre fasi: la progettazione concettuale, la progettazione logica e la progettazione fisica. Verrà ora presentata nel dettaglio ciascuna fase.

4.1 Progettazione concettuale

Il primo passo è stato quello di analizzare e descrivere sotto forma di specifiche formali la realtà di interesse che si vuole modellare, al fine di produrre un modello concettuale dei dati. In questo modo è possibile descrivere l'organizzazione dei dati ad un alto livello di astrazione, senza tenere in considerazione gli aspetti implementativi. Il prodotto di questa fase è lo schema Entità-Relazione.

4.1.1 Analisi informale dei requisiti

Come si è detto precedentemente, lo scopo è quello di progettare una base di dati per gestire in modo centralizzato, attraverso una web service, reti di sensori ubicate in diverse posizioni geografiche, anche distanti fra loro. Con il termine device si intende un qualsiasi dispositivo che sfrutti l'architettura REST, rendendo le proprie risorse fruibili tramite un web service. I dispositivi sono dotati di un indirizzo IPv4 oppure IPv6, e possono essere direttamente connessi ad Internet oppure passare attraverso un gateway. Per ciascun dispositivo reale si vogliono memorizzare due tipologie di informazioni:

- fisiche, ossia tutte quelle indipendenti dalla funzione per cui il device viene utilizzato. Sono un esempio le informazioni relative alla comunicazione come

l'indirizzo ip e la porta oppure quelle relative alla sua posizione geografica come latitudine e longitudine;

- logiche, cioè tutte quelle che riguardano l'attuale funzione che il dispositivo sta ricoprendo. Sono un esempio di informazioni logiche il nome, la descrizione ed eventualmente i tag associati per rendere semplice la ricerca e il raggruppamento di dispositivi;

Dato che i device rispettano il paradigma REST, ad ognuno di essi è associato un insieme di risorse. Ogni risorsa è identificata univocamente dal suo percorso (o URI) all'interno del dispositivo e può possedere un insieme di ulteriori attributi come l'unità e il simbolo di misura. Per ciascuna risorsa si vuole inoltre mantenere lo storico dei valori osservati, prevedendo la possibilità di usufruire di un sistema di caching in modo da limitare il numero di richieste verso i dispositivi e riducendo così il loro consumo energetico. Si vuole poi gestire un sistema di notifiche o allarmi. A questo scopo è possibile definire dei trigger associati alle risorse. Ogni trigger definisce una condizione da verificare ogniqualvolta un valore viene monitorato, e un insieme di azioni da intraprendere quando tale condizione risultasse verificata. Le azioni possono prevedere per esempio l'invio di email o SMS. Tutte queste informazioni devono infine essere rese fruibili in un sistema multiutente dunque è necessario memorizzare per ciascuno di questi le informazioni di contatto e quelle necessarie all'autenticazione delle richieste in entrata al web service. Inoltre deve essere implementato un sistema di permessi che permetta ad ognuno di loro di interagire solo con i device registrati alla piattaforma di cui possiede i privilegi.

Dunque un tipico caso d'uso del sistema è il seguente:

- un utente si registra al sistema;
- l'utente registra i dispositivi di cui è in possesso, creandone così dei profili virtuali;
- ai dispositivi creati vengono registrate le risorse di cui si compongono, manualmente oppure richiedendole ai dispositivi stessi;
- l'utente esegue ricerche, interrogazioni, aggiornamenti sui dispositivi registrati;

4.1.2 Entità

Sulla base di quanto appena esposto in termini generali, vengono ora definite in modo strutturato le entità che descrivono la base di dati.

User

L'entità User rappresenta tutte le informazioni associate agli utenti registrati al sistema. Si definiscono gli attributi: `id` per identificare univocamente ciascun utente,

`password` memorizza l'MD5 della password, `registration_date` rappresenta la data di registrazione al sistema, `name`, `surname`, `phone` ed `email` per salvare le restanti informazioni associate all'utente.

Physical Device

Come si era descritto precedentemente in linea generale, i dispositivi possiedono delle informazioni di carattere fisico e altre di carattere logico. Per questo motivo si è deciso di suddividere le informazioni relative ad un device in 2 entità distinte. L'entità Physical Device descrive un dispositivo dal punto di vista fisico, memorizzando gli attributi: `ip`, l'indirizzo IPv4 oppure IPv6 del dispositivo, `mac`, l'indirizzo MAC, `port`, la porta su cui è in ascolto, `lat` e `lon` che rappresentano rispettivamente latitudine e longitudine, necessarie al fine di geolocalizzare il dispositivo, `protocol` infine descrive il protocollo usato nella comunicazione.

Gateway

L'entità Gateway rappresenta, per ciascun sensore in una rete locale, il punto di uscita verso Internet e l'API server. Si tratta di una entità debole poiché non possiede un proprio identificatore. Un gateway viene infatti identificato dagli attributi `ip`, `port` congiuntamente al device fisico a cui è associato. Dato che un device fisico può possedere molti gateway, si pensi ad esempio ad un dispositivo che viene spostato spesso, l'attributo `priority` definisce l'ordine con cui l'API server dovrà scegliere il gateway per tentare la comunicazione.

Logical Device

L'entità Logical Device memorizza le informazioni logiche di un device, ossia quelle legate alla funzione che sta svolgendo, e viene identificata da un attributo `id`. In pratica rappresenta la virtualizzazione del dispositivo all'interno del nostro sistema. Gli attributi `name` e `description`, assegnano una semantica al ruolo ricoperto dal dispositivo. Come si è detto più device logici possono far riferimento a un device fisico, ma solamente una di queste associazioni può essere attiva in un dato istante. Si pensi per esempio ad un sensore di temperatura: questo può essere posizionato in un edificio e poi spostato all'esterno. In tale situazione il dispositivo fisico non è cambiato, ma la sua funzione logica non è più la stessa. Inoltre i valori monitorati prima all'interno e poi all'esterno non hanno nulla a che fare, e questa è la motivazione per cui si è introdotta questa separazione concettuale. Questa associazione viene modellata per mezzo di una relazione (vedi 4.1.3). Infine l'attributo `enabled` specifica se il dispositivo virtuale è abilitato, cosa che prescinde dal fatto che il device sia acceso o spento, il suo scopo è per esempio quello di impedire che task periodici del server vadano a richiedere informazioni al dispositivo fisico.

Resource

Ogni dispositivo, aderendo all'architettura REST, espone delle risorse che vengono mappate nell'entità Resource. Un device logico può quindi possedere molte risorse, ognuna delle quali è identificata dall'attributo `path`, il percorso della risorsa, congiuntamente al device di appartenenza. Per ogni risorsa si tiene traccia di: `name`, `description`, `measure_unit` e `measure_symbol` che identificano l'unità di misura del sensore nel caso fosse presente. Come si può vedere queste informazioni rispecchiano una parte di quelle previste dal Core Link Format (vedi 2.1.2). Si vuole infatti permettere, oltre a quella manuale, la registrazione automatica delle risorse in modo che l'API server ne richieda la lista direttamente ai dispositivi. Infine nell'entità Resource si è deciso di salvare l'ultimo valore monitorato e il suo timestamp associato, mediante i campi `value` e `value_timestamp`. Sfruttando poi l'attributo `expiration` è stata implementata una sorta di cache, permettendo all'utente di specificare un periodo di validità per tale valore in funzione delle caratteristiche della risorsa.

Data Entry

L'entità Data Entry memorizza lo storico dei valori di una particolare risorsa. Ogni data entry è identificata univocamente dall'attributo `id` e tiene traccia del valore ricevuto e del timestamp associato tramite `value` e `value_timestamp`.

Trigger

L'entità Trigger rappresenta un allarme associabile ad una particolare risorsa. Viene identificato da un `id` e per ognuno di essi si tiene traccia di: nome e descrizione attraverso i campi `name` e `description`, data di creazione tramite l'attributo `creation_time`. Inoltre `is_active` permette di memorizzare lo stato del trigger e `java_condition` rappresenta la condizione che deve essere verificata affinché vengano eseguite l'insieme di azioni associate al trigger. Per esempio la condizione

Trigger Action

L'entità Trigger Action rappresenta le operazioni da compiere quando la condizione del trigger associato è verificata. Ogni azione è identificata da un `id` e si compone di 3 attributi: `type` specifica il tipo di operazione da eseguire, per esempio i valori "email" o "sms" specificano il tipo di notifica, `content` indica il contenuto informativo dell'allarme e `target` indica il destinatario dell'operazione. Nel caso di una notifica è previsto che il campo `target` corrisponda all'email o al numero di cellulare di un utente. Non sono stati imposti vincoli sul campo type e target poiché si prevede che in un futuro i trigger possano eseguire una notifica su una particolare URL, per esempio

attraverso una chiamata HTTP POST. In questo modo sarebbe possibile dotare la piattaforma di funzionalità di interazione M2M.

Tag

Per device logici e risorse è prevista l'assegnazione di tag, che ne permettono il raggruppamento sulla base di una parola chiave e dunque una semplice ricerca. L'entità TAG è costituita da un unico attributo `tag` che lo identifica univocamente. Si vorrà prevedere la possibilità di assegnare dei tag di default, condivisi da tutti gli utenti, e dei tag proprietari associati ad uno specifico utente. Deve essere permesso l'assegnamento della stessa parola chiave sia come tag di default che come tag proprietario dato che la ricerca deve poter essere implementata su questi due insiemi separatamente.

App

L'entità APP rappresenta le applicazioni registrate al sistema e serve a memorizzare informazioni utili per la fase di autenticazione delle richieste REST verso l'API server (vedi 5.6.1). Si prevede infatti che qualsiasi utente del sistema possa implementare la propria applicazione, sfruttando le API messe a disposizione. Per far questo, ad ogni applicazione deve essere assegnata una chiave che dovrà essere inclusa nell'header delle richieste HTTP che l'applicazione stessa invia all'API server. L'attributo `id` la identifica univocamente, mentre il nome, la descrizione e lo stato dell'applicazione vengono rappresentati dai campi `name`, `description` ed `enabled`. L'attributo `private_access_key` memorizza la chiave che identifica l'applicazione nelle richieste REST e deve essere unica per ciascuna applicazione. Non viene usata come identificatore univoco per l'applicazione perché questa si suppone possa cambiare nel tempo. Infine `access_key_id` rappresenta l'identificatore della chiave ed è stato introdotto per compatibilità col protocollo di autenticazione utilizzato (vedi 5.6.1) che prevede la possibilità di assegnare più chiavi alla stessa applicazione. In ogni caso, anche se non indispensabile, si è deciso di mantenere l'attributo.

4.1.3 Associazioni

Identificate le entità che costituiscono il modello, si descriveranno ora le relazioni tra di esse definendo i blocchi che andranno a comporre lo schema E-R. Si ricorda che le cardinalità nelle relazioni sono espresse come numero di occorrenze con cui l'entità partecipa alla relazione.

Use Gateway

Questa relazione è definita tra le entità PHYSICAL DEVICE e GATEWAY e descrive il fatto che ad un device fisico possono essere associati più gateway, eventualmente

nessuno se il dispositivo è direttamente connesso ad Internet. Ad ogni gateway che, come si è detto, è stato modellato come entità debole può essere associato uno ed un solo device fisico.

Physical Association

Questa relazione è definita tra le entità Physical Device e Logical Device. Ogni device logico deve essere associato alle informazioni fisiche necessarie per la comunicazione. Dunque un device logico deve essere collegato ad uno ed un solo device fisico mentre lo stesso dispositivo fisico può essere associato a molti dispositivi logici. L'attributo di relazione `active_association` definirà qual'è il dispositivo logico attualmente attivo sullo specifico dispositivo fisico. Si è deciso di modellare con una relazione uno a molti questo comportamento perché si vuole mantenere traccia di tutti i dispositivi logici che sono stati associati in passato ad un certo device fisico. Tornando all'esempio fatto durante la descrizione dell'entità Logical Device, si supponga di aver registrato al sistema un sensore di temperatura la cui funzione è quella di monitorare la temperatura interna di un edificio. Questo a livello di base di dati implica l'inserimento di un nuovo device fisico (per semplicità di spiegazione chiamiamolo "temp"), la creazione di un device logico ("indoor-temp") e la loro associazione, marcandola come attiva. Se il dispositivo viene spostato all'esterno dell'edificio cambia la sua funzione logica e dunque devono essere eseguite le seguenti operazioni:

- creare un nuovo device logico chiamato "outdoor-temp" che identifica la nuova funzione di monitoraggio della temperatura esterna;
- marcare come inattiva l'associazione tra il device logico "indoor-temp" e il device fisico "temp";
- creare una nuova associazione tra il device fisico "temp" e il device logico "outdoor-temp";

In questo modo l'associazione che si era creata fra "temp" e "indoor-temp" rimane memorizzata nella base di dati ed è eventualmente riattivabile in un secondo momento giocando con operazioni di associazione e dissociazione tra device logici e device fisici. In questo modo si mantiene inoltre la consistenza dei dati contenuti nello storico dei valori monitorati. Infatti, i valori relativi alle temperature interne saranno agganciati al device logico "indoor-temp", mentre quelli relativi alle temperature esterne saranno agganciati al device "outdoor-temp".

Expose

Questa relazione è definita tra le entità Logical Device e Resource e modella l'architettura REST dei dispositivi, associando a ciascuno di essi un insieme di risorse.

Ad una risorsa è associato invece uno ed un solo dispositivo logico. Si tratta dunque di una relazione "uno a molti".

History

Questa relazione è definita tra le entità Resource e Data Entry e lega i valori ricevuti dai dispositivi alla risorsa a cui fanno riferimento. Per ciascuna risorsa possono essere registrate molte data entry mentre ogni data entry è associata ad una ed una sola risorsa.

Resource Trigger

Questa relazione è definita tra le entità Resource e Trigger e modella la possibilità di associare delle notifiche o allarmi al verificarsi di determinate condizioni sui valori delle risorse. Un trigger può essere associato ad una ed una sola risorsa mentre più trigger possono essere associati alla stessa risorsa.

Composition

Come già descritto in precedenza un trigger può essere composto da molte azioni, mentre una singola azione può essere associata ad uno ed un solo trigger. La relazione "uno a molti" Composition esprime proprio questo comportamento.

Physical Device Creator

Questa relazione è definita tra le entità User e Physical Device ed è necessaria per tener traccia del creatore del device logico. Ad ogni device logico può essere associato uno ed uno solo creatore mentre lo stesso utente può essere il creatore di molti device logici.

Logical Device Creator

Come per la relazione Physical Device Creator tiene traccia del creatore del dispositivo logico dunque si tratta di una "uno a molti" tra le entità User e Logical Device.

Trigger Creator

Come per la relazione Physical Device Creator tiene traccia del creatore del trigger dunque si tratta di una "uno a molti" tra le entità User e Trigger.

User App

Si tratta di una relazione "molti a molti" tra le entità User e App. Essa memorizza a quali applicazioni sono registrati i vari utenti. Un utente infatti può registrarsi a

più applicazioni e una stessa applicazione può essere usata da più utenti. Questa relazione contiene inoltre informazioni necessarie per la fase di autenticazione delle richieste. Per ciascun utente e per specifica applicazione vengono salvati, sotto forma di attributo di relazione: `refresh_token`, l'`access_token` e la sua scadenza `access_token_expiration`. Il primo rappresenta una stringa permanente che va a sostituirsi a nome utente e password dell'utente per una particolare applicazione, e permette di ottenere una chiave di accesso temporanea, l'`access_token`. Quest'ultimo ha una durata limitata definita da `access_token_expiration`. Per comprendere meglio il protocollo di autenticazione, e quindi il significato di questi attributi, si veda la sezione 5.6.1.

Developer

Per ciascuna App si vuole tener traccia dello sviluppatore, il quale deve essere un membro dell'entità User. Questa relazione è necessaria perché si prevede che lo sviluppatore abbia dei privilegi particolari nella gestione delle applicazioni, cosa che gli altri utenti registrati al sistema non possono fare. Si tratta di una relazione "uno a molti" dato che, per ciascuna applicazione ci può essere uno ed un solo sviluppatore, mentre un utente può essere il creatore di più applicazioni.

Default Device Tag

Questa relazione è definita tra le entità Tag e Logical Device e descrive il fatto che per ciascun device logico è possibile assegnare delle parole chiave di default ossia relative solamente al dispositivo e condivise tra tutti gli utenti. Ad un dispositivo logico è possibile assegnare più tag e uno stesso tag può essere assegnato a molti dispositivi logici. Si tratta dunque di una relazione "molti a molti".

Device Tag

Si tratta di una relazione ternaria tra le entità Tag, Logical Device e User. Permette di modellare la possibilità, per ciascun utente, di definire un insieme di tag associati ad un particolare device logico. Tutte le entità possono partecipare alla relazione con molte occorrenze.

Default Resource Tag

Questa relazione è definita tra le entità Tag e Resource e rappresenta lo stesso concetto della relazione Default Device Tag, ma trasportato all'ambito delle risorse.

Resource Tag

Si tratta di una relazione ternaria tra le entità Tag, Resource e User. Anche in questo caso il concetto è lo stesso della relazione Device Tag trasportato all'ambito delle risorse.

4.1.4 Dizionario dei dati

Al fine di dare una descrizione sintetica dei concetti rappresentati, viene presentato nelle tabelle 4.1 e 4.2 il dizionario dei dati.

Tabella 4.1: Dizionario dei dati: entità

Entità	Descrizione	Attributi	Identificatore
User	Utente registrato al sistema	`id, name, surname, email, password, phone, registration_date`	`id`
Physical Device	Informazioni fisiche di un dispositivo	`id, ip, port, mac, protocol, lat, lon`	`id`
Gateway	Punto di accesso alla rete locale di sensori	`ip, port, priority`	`ip, port,` Physical Device(`id`)
Logical Device	La virtualizzazione di un dispositivo, ne memorizza le informazioni logiche	`id, name, description, enabled`	`id`
Resource	La virtualizzazione delle risorse dei dispositivi reali	`path, name, description, measure_unit, measure_symbol, value, value_timestamp, expiration`	`path,` Logical Device(`id`)
Data Entry	Storico dei valori ricevuti dai dispositivi	`id, value, value_timestamp`	`id`
Trigger	Allarme o notifica associata ad una certa condizione	`id, name, description, creation_time, is_active, java_condition`	`id`
Trigger Action	Insieme di azioni di cui si compone un allarme	`id, content, type, target`	`id`
Tag	Parola chiave associata a risorse o dispositivi logici	`tag`	`tag`
App	Applicazioni di terze parti registrate al sistema	`id, name, description, access_key_id, private_access_key, enabled`	`id`

Tabella 4.2: Dizionario dei dati: relazioni

Relazione	Descrizione	Entità coinvolte	Attributi
Use Gateway	Associa ad un device fisico i suoi gateway	Physical Device(0,N), Gateway(1,1)	
Physical Association	Associa ad un device fisico i dispositivi logici	Physical Device(0,N), Logical Device(1,1)	`is_active`
Expose	Associa ad un dispositivo logico le sue risorse	Logical Device(0,N), Resource(1,1)	
History	Associa ad una risorsa i valori ricevuti dal dispositivo	Resource(0,N), Data Entry(1,1)	
Resource Trigger	Associa ad una risorsa i suoi trigger	Resource(0,N), Trigger(1,1)	
Composition	Associa ad un trigger le azioni da compiere	Trigger(1,N), Trigger Action(1,1)	
Physical Device Creator	Associa ad un dispositivo fisico il suo creatore	Physical Device(1,1), User(0,N)	
Logical Device Creator	Associa ad un dispositivo logico il suo creatore	Logical Device(1,1), User(0,N)	
Trigger Creator	Associa ad un dispositivo logico il suo creatore	Trigger(1,1), User(0,N)	
User App	Registra un utente ad una applicazione	User(1,N), App(1,N)	`refresh_token`, `access_token`, `access_token_expiration`
Developer	Associa gli sviluppatori alle applicazioni	User(0,N), App(1,1)	
Default Device Tag	Associa un dispositivo logico ai suoi tag di default	Tag(0,N), Logical Device(0,N)	
Device Tag	Associa dispositivi logici a parole chiave ed utenti	Tag(0,N), Logical Device(0,N), User(1,N)	
Default Resource Tag	Associa un dispositivo fisico ai suoi tag di default	Tag(0,N), Physical Device(0,N)	
Resource Tag	Associa dispositivi fisici a parole chiave ed utenti	Tag(0,N), Physical Device(0,N), User(0,N)	

4.1.5 Schema Entità Relazione

Sulla base dell'analisi appena descritta il modello risultante è dato dallo schema E-R in figura 4.1.

4.2 Progettazione logica

La fase di progettazione logica consiste nella traduzione dello schema concettuale definito nella fase precedente, nel modello di rappresentazione dei dati adottato dal DBMS. Il prodotto di questa seconda fase è lo schema logico e prevede una prima ristrutturazione dello schema Entità Relazione sulla base di criteri di semplificazione e ottimizzazione, e successivamente la traduzione vera e propria verso il modello logico.

4.2.1 Schema E-R ristrutturato

Come si è detto, prima di tradurre il modello concettuale, è necessario attuare alcune semplificazioni. In particolare in questo processo si analizzano le ridondanze, si accorpano o suddividono concetti e si scelgono gli identificatori primari.

La principale ottimizzazione introdotta in questa fase riguarda la gestione dei tag. Le operazioni più frequenti che si vogliono prevedere sull'entità Tag sono:

- **operazione 1**: data una lista di tag come parametro, ricerca tutti i dispositivi logici a cui siano assegnate tutte quelle parole chiave come tag di default;
- **operazione 2**: data una lista di tag e un particolare utente come parametri, ricerca tutti i dispositivi logici che sono associati a quell'insieme di tag e a quello specifico utente;
- **operazione 3**: data una lista di tag come parametro, ricerca tutte le risorse a cui siano assegnate tutte quelle parole chiave come tag di default;
- **operazione 4**: data una lista di tag e un particolare utente come parametri, ricerca tutte le risorse che sono associate a quell'insieme di tag e a quello specifico utente;

Come si può vedere, ciascuna operazione esegue una ricerca sulla base di una sola delle 4 tipologie di tag. Inoltre l'entità Tag si compone di un solo attributo `tag`. Per questo motivo si è deciso di suddividere l'entità Tag in 4 particolari entità deboli, in cui la parola chiave diventa parte dell'identificatore. Questo implica una introduzione di ridondanza nei dati ma è una scelta giustificata dal fatto che, al momento della ricerca, questa viene eseguita all'interno del particolare sottoinsieme di tag richiesto e risulta dunque più rapida. In genere non si vogliono infatti cercare risorse e dispositivi contemporaneamente, anche se questo è comunque possibile. Inoltre, il fatto di lasciare il tag stesso come parte della chiave primaria di queste nuove entità, e non doverlo reperire attraverso pesanti operazioni di `join` da una tabella separata, dovrebbe garantire una maggior velocità d'esecuzione delle procedure descritte. In figura 4.2 si può vedere come è stato modificato lo schema concettuale sulla base di queste considerazioni.

Per quanto riguarda la scelta degli identificatori, si è deciso di sfruttare delle chiavi primarie surrogate per le entità che prevedono l'uso di chiavi primarie composte da più attributi. In particolare le entità deboli GATEWAY e RESOURCE sono quelle interessate da questa ottimizzazione. Viene infatti introdotto per entrambe le entità l'attributo *id*, che le identificherà univocamente e permetterà di alleggerire le chiavi esterne che devono referenziare queste entità. Nonostante questo, si prevede comunque un indice con vincolo di unicità sui i campi che costituiscono l'identificatore naturale di entità. Lo schema E-R ristrutturato di figura 4.2 rispecchia questa analisi.

4.2.2 Traduzione verso il modello relazionale

La seconda fase della progettazione logica consiste nel tradurre lo schema E-R ristrutturato e costruire lo schema logico equivalente. La traduzione naturale nel modello relazionale prevede:

- per ogni entità, una tabella con lo stesso nome avente per attributi i medesimi attributi dell'entità e per chiave il suo identificatore;
- per l'associazione, una tabella con lo stesso nome avente per attributi gli attributi dell'associazione e gli identificatori delle entità coinvolte; tali identificatori formano la chiave della relazione

Viene ora presentato lo schema relazionale ottenuto:

USER(id, name, surname, email, password, phone, registration_date)

PHYSICAL DEVICE(id, ip, port, mac, protocol, lat, lon, created_by)

GATEWAY(id, ip, port, priority, physical_id)

LOGICAL DEVICE(id, name, description, enabled, created_by, physical_id)

RESOURCE(id, path, name, description, measure_unit, measure_symbol, value, value_timestamp, expiration, device_id)

DATA ENTRY(id, value, value_timestamp, resource_id)

TRIGGER(id, name, description, creation_time, is_active, java_condition, created_by, resource_id)

TRIGGER ACTION(id, content, target, type, trigger_id)

DEFAULT DEVICE TAG(device_id, tag)

DEVICE TAG(device_id, tag)

DEFAULT RESOURCE TAG(resource_id, tag, user_id)

RESOURCE TAG(resource_id, tag, user_id)

User App(app_id, user_id, access_token, access_token_expiration, refresh_token)

App(id, name, description, access_key_id, private_access_key, enabled)

In figura 4.3 si ha una rappresentazione grafica dello schema relazionale nel suo complesso. Per una sua migliore comprensione viene proposta una legenda della sintassi utilizzata:

- gli attributi che costituiscono la chiave primaria sono affiancati da una chiave;
- gli attributi affiancati da indicatore pieno non possono essere nulli, a differenza di quelli affiancati da indicatore vuoto;
- gli indicatori rossi indicano gli attributi che sono chiave esterna;
- i collegamenti sono tratteggiati quando il vincolo di integrità referenziale non è parte della chiave primaria nella tabella referenziata, in caso contrario la linea è continua;
- per ogni associazione è indicata inoltre la molteplicità con cui ciascuna tabella partecipa alla relazione[1].

4.3 Implementazione

Per quanto riguarda l'implementazione dello schema logico in MySQL si rimanda all'appendice A.

[1] La notazione usata a livello di schema logico per le cardinalità è diversa da quella usata per lo schema concettuale. Infatti le cardinalità delle associazioni binarie nello schema E-R considerano il numero di occorrenze di una entità ad una data relazione. A questo livello le cardinalità risultano invece invertite.

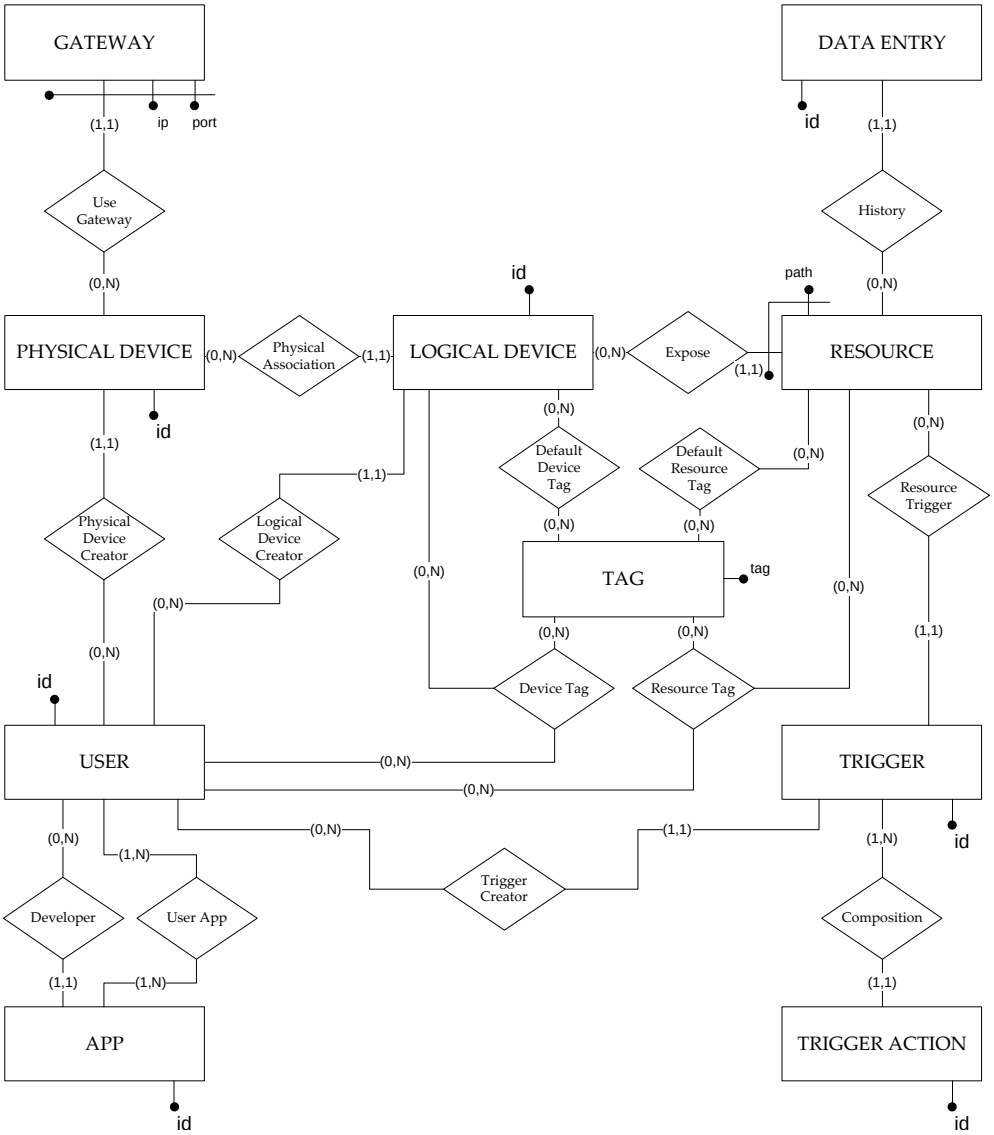


Figura 4.1: Schema Entità Relazione complessivo.

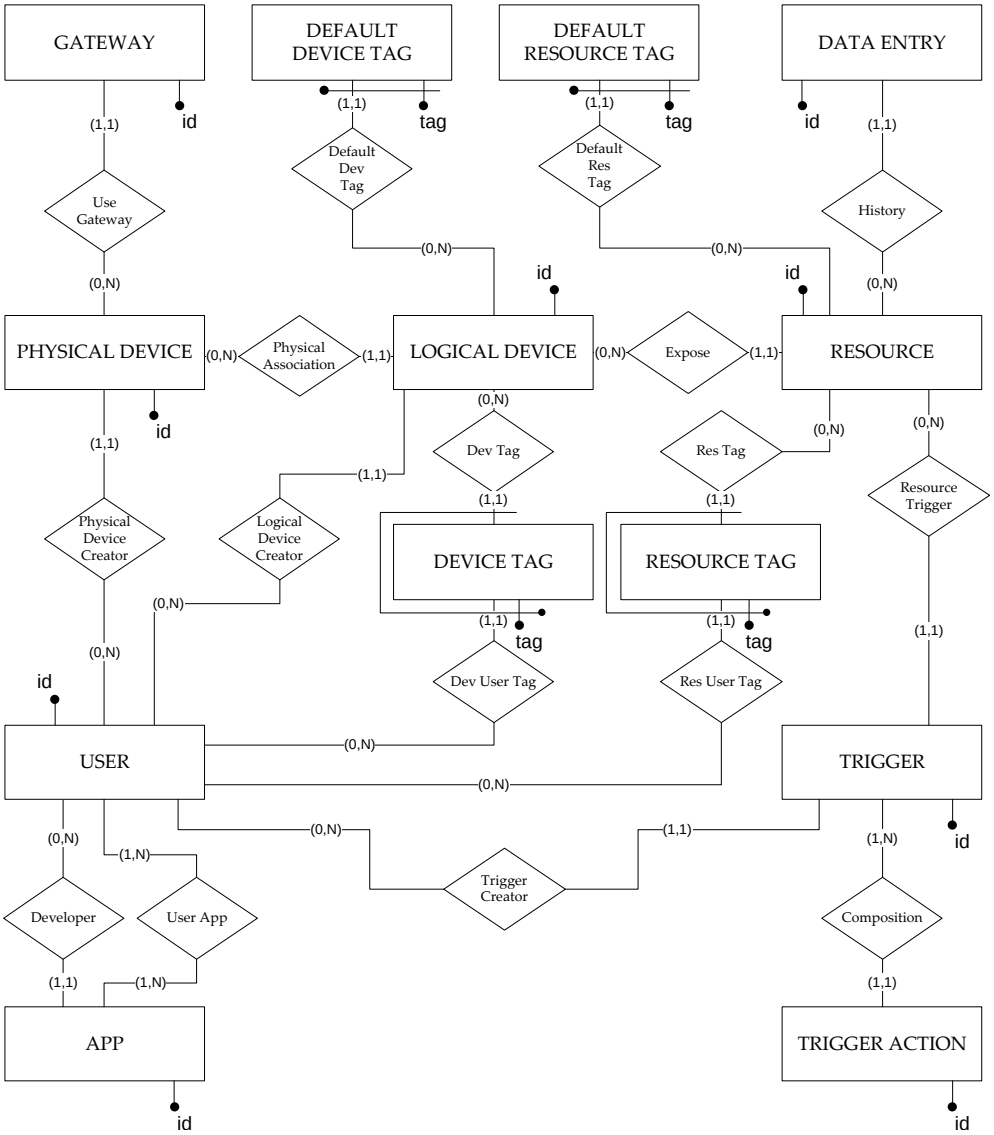


Figura 4.2: Schema Entità Relazione ristrutturato.

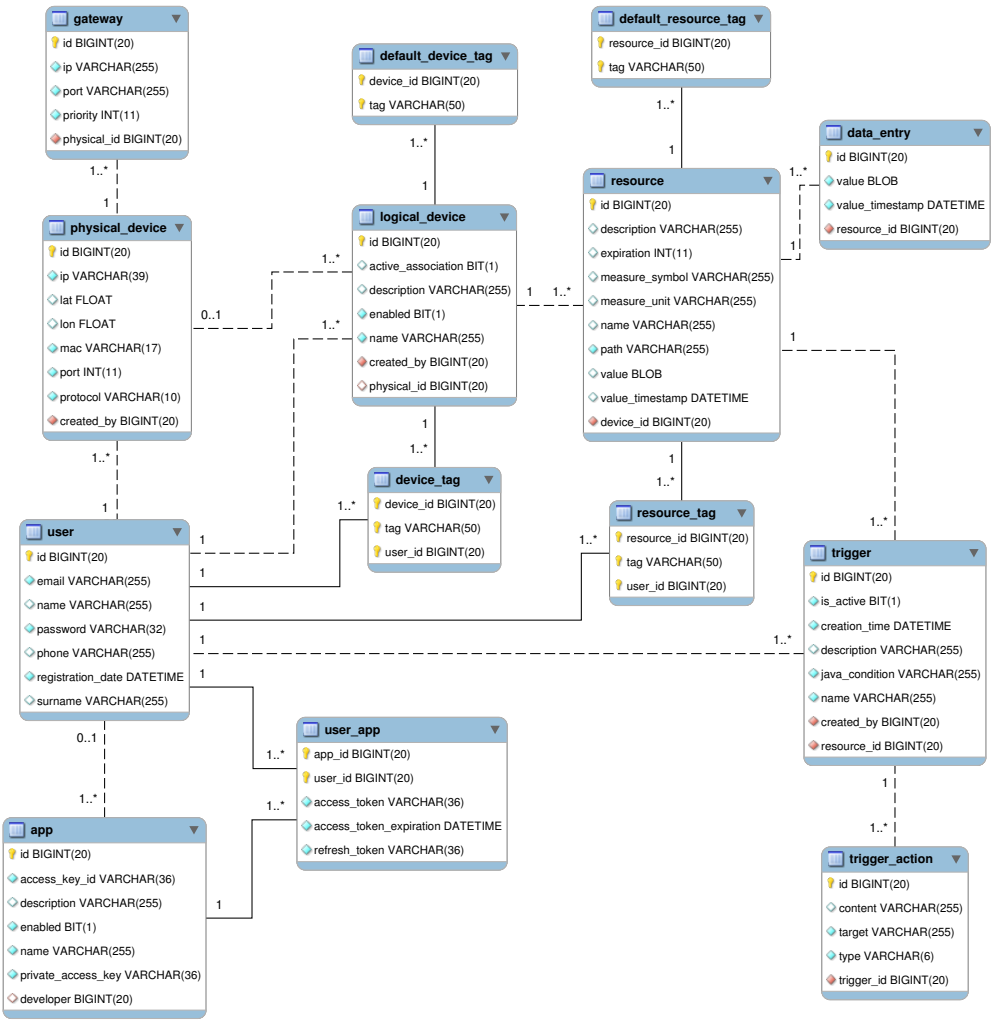


Figura 4.3: Schema logico complessivo.

Capitolo 5

API server

In questo capitolo viene descritto come è stato realizzato l'API server e vengono illustrati nel dettaglio i vari moduli di cui si compone il sistema.

5.1 Architettura

Nel capitolo 3 sono stati definiti i 4 moduli di cui l'API server è costituito: l'*Access Module*, il *Processing Module*, il *Data Module* e il *Communication Module*.

Quando una richiesta HTTP da parte di un client giunge all'API server, viene presa in carico dall'Access Module, che si occupa di farne il parsing e passarla al Processing module. Questo si occupa di tutte le elaborazioni necessarie andando a richiedere le informazioni riguardanti i device virtualizzati al Data Module. Se la richiesta prevede l'interrogazione diretta dei dispositivi fisici, il Processing Module demanda i compiti relativi alla comunicazione al Communication Module. Una volta creata la risposta, questa viene passata nuovamente all'Access Module che la restituisce al client nel formato richiesto da quest'ultimo.

Dal punto di vista implementativo: l'Access Module è associato agli Spring Controller, i quali dipendono da oggetti detti service che racchiudono la logica di business e rappresentano il Processing Module. I service infine dipendono da altri oggetti, chiamati DAO che si occupano dell'accesso ai dati. Si crea dunque una catena di dipendenze che viene gestita dallo Spring Framework attraverso l'uso della dependency injection. Nei prossimi paragrafi sarà descritto nel dettaglio questo meccanismo.

5.2 DTO pattern

Gli oggetti che mappano le tabelle del database e che vengono usati per implementare la logica dell'API server sono dette classi di dominio. Volendo implementare un web

service la cosa più semplice per comunicare con i client sarebbe quella di fare l'unmarshalling in fase di ricezione e il marshalling in fase di risposta di tali oggetti, in formato XML o JSON. Tuttavia ci sono dei problemi. Il primo è che a volte le informazioni che si vogliono presentare all'utente non sono esattamente la rappresentazione completa dell'oggetto di dominio. L'altro problema è che spesso, lavorando con interfacce remote, è necessario ridurre il numero di comunicazioni, accorpando informazioni associate internamente ad oggetti di dominio diversi in un unico messaggio. Per questi motivi esiste un pattern di programmazione chiamato Data Transfer Object (DTO). Questo prevede la creazione di un ulteriore strato di oggetti, strutturati diversamente da quelli di dominio, che non contengono alcuna logica di business ma hanno il solo scopo di trasportare i dati. Applicando questa tecnica, a livelli diversi dell'API server si hanno oggetti diversi: i Controller ricevono e rispondono con dei DTO, i Service fanno la traduzione dei DTO in oggetti di domino e viceversa, i DAO usano solamente gli oggetti di dominio.

5.3 Access Module

L'Access Module si occupa di tutti gli aspetti di comunicazione verso i client, costituendo il punto di ingresso dell'API server. Le sue principali funzionalità sono dunque:

- validare le richieste XML e JSON in entrata al sistema;
- fare il parsing delle richieste HTTP e trasformarle in oggetti manipolabili dal Processing Module;
- associare alle risposte in uscita il corretto formato, ossia quello definito dall'utente nell'header HTTP `Accept`;
- gestire le eccezioni e segnalarle attraverso delle risposte a cui è associato uno specifico HTTP status code.

L'Access Module viene implementato per mezzo degli Spring Controller, ciascuno dei quali, fornisce un insieme ben definito di funzionalità accessibili attraverso richieste REST su specifiche URL. Questo percorso deve essere significativo dal punto di vista semantico, e dunque descrivere, insieme al metodo HTTP, la risorsa su cui si vuole eseguire l'azione. Nel listato 5.1 è possibile osservare l'implementazione del controller relativo all'operazione B.1.3 definita in appendice.

Codice 5.1: Esempio di Spring Controller

```
1 @Controller
2 @RequestMapping("/physical_device")
3 public class PhysicalDeviceController {
```

```
4
5   @Autowired
6   private PhysicalDeviceService pDevService;
7
8   @Autowired
9   private AuthenticationService authService;
10
11  // ...
12
13  /**
14   * Returns a detailed description of the queried device.
15   * @param response the response that has to be sent to the client.
16   * @param key the user authentication key
17   * @param pDevId the id of the physical device.
18   * @return a DeviceDescription object if the device is stored on the
         database
19   */
20  @RequestMapping(method=RequestMethod.GET, value="/{pDevId}")
21  @ResponseStatus(value = HttpStatus.OK)
22  @ResponseBody
23  public PhysicalDeviceDTO physicalDeviceDescription(
24    HttpServletRequest request,
25    @RequestHeader("key") String key,
26    @PathVariable String pDevId){
27
28   User authUser = authService.authenticate(key);
29   log.info(request.getMethod() + " " + request.getRequestURI() + " from
         user: " + authUser.getEmail());
30
31   return pDevService.getPhysicalDevice( pDevId, authUser );
32  }
33
34  // ...
35 }
```

- `@Autowired` è l'annotazione che viene usata per iniettare la dipendenza dagli oggetti di livello service: `AuthenticationService`, il quale si occuperà della fase di autenticazione, e `PhysicalDeviceService`, che valuterà i permessi che l'utente autenticato ha sulla particolare risorsa, e demanderà al Data Module il compito di recuperare i dati;
- `@RequestMapping` è una annotazione che serve per associare, sulla base della URL, la richiesta in entrata alla classe e al metodo a cui spetta la sua gestione;
- `@ResponseStatus` definisce il codice HTTP da associare alla risposta in caso l'esecuzione vada a buon fine;
- `@ResponseBody` permette di includere nella risposta la serializzazione dell'oggetto `PhysicalDeviceDTO`) ritornato dal metodo. In questo modo la scelta della

"view", come descritto nel modello MVC in 3.4.4, viene eseguita automaticamente sulla base dell'header `Accept` incluso nella richiesta.

- `@RequestHeader` permette di accedere ad un header specificandone il nome. In questo caso si vuole accedere all'header `key`;

- `@PathVariable` viene usato per mappare le variabili presenti nel percorso della risorsa in parametri del metodo;

Per rispondere a tutte le esigenze del sistema sono stati implementati i seguenti controller:

- **AuthenticationController** si occupa di gestire l'autenticazione degli utenti registrati al sistema. Il sistema implementa lo stesso protocollo usato da *SugarSync*[18];

- **PhysicalDeviceController** fornisce tutte le operazioni di creazione, lettura, aggiornamento e cancellazione sui dispositivi fisici;

- **LogicalDeviceController** fornisce tutte le operazioni di creazione, lettura, aggiornamento e cancellazione sui dispositivi logici. Permette inoltre di dissociare e riassociare dispositivi logici a dispositivi fisici;

- **ResourceController** si occupa della gestione delle risorse associate ai dispositivi. Oltre a fornire le operazioni di creazione, lettura, aggiornamento e cancellazione su ciascuna risorsa, permette di interrogare i dispositivi direttamente per ottenere letture aggiornate, ed anche la possibilità di accedere allo storico dei valori monitorati;

- **TriggerController** permette di definire delle condizioni sui valori delle risorse, ed associare a ciascuna di esse una serie di azioni. Queste possono essere l'invio di notifica via email o sms, ma anche l'esecuzione di una chiamata HTTP su una particolare URL della piattaforma o di un server esterno;

- **SchedulingController** permette di istruire il server per eseguire delle richieste periodiche a determinate risorse;

- **AsyncNotificationController** permette di gestire comunicazione asincrona con il Communication Module;

Per quanto riguarda gli aspetti legati alla comunicazione e alle notifiche, questi vengono descritti nel dettaglio in [15] e sono fuori dagli scopi di questa tesi.

5.4 Processing Module

Il Processing module è il componente centrale dell'API server. Si occupa di mettere in comunicazione gli altri moduli traducendo i DTO provenienti dall'Access Module in oggetti di dominio, sui quali eseguire tutte le elaborazioni. Come si è detto i componenti di Spring che gestiscono questo modulo sono degli oggetti detti service, i quali combinano insieme operazioni legate a moduli diversi al fine di implementare la logica di business. Per esempio quando si vuole ottenere il valore di una risorsa viene invocato un metodo della classe `ResourceService` il quale si occupa di:

- interrogare il database per verificare l'esistenza della risorsa e ottenere l'ultimo valore monitorato;
- se tale valore non è più valido deve demandare al `Communication Module` il compito di interrogare direttamente il dispositivo;
- una volta ottenuto il valore aggiornato deve incapsularlo nell'oggetto DTO predisposto per essere inviato al client, e passarlo all'Access Module;

Oltre a ciò nel Processing Module risiede tutta la logica relativa alle gestione delle notifiche ed alla gestione dei thread periodici programmati che vanno a monitorare le risorse.

5.5 Data Module

5.5.1 DAO pattern

Il Data Module è quello che si occupa dell'accesso ai dati. Esso è stato implementato sfruttando il framework Hibernate e il pattern Data Access Object (DAO). Per quanto concerne Hibernate, nella sezione 3.4.3 è stato spiegato come le tabelle del database vengono mappate nelle classi dell'API server, le classi di dominio. Il pattern DAO è invece un paradigma di programmazione che mira a definire un'interfaccia per le operazioni di persistenza (Creazione, lettura, aggiornamento, cancellazione e ricerca) slegandole completamente dalla specifica implementazione e dallo specifico DBMS utilizzato. In questo modo è possibile sostituire il sistema di persistenza (per esempio passare da un database MySQL ad uno PostreSQL senza il bisogno di riscrivere completamente l'applicazione). Inoltre, obbliga il programmatore a riunire il codice riguardante le operazioni su una specifica tabella del database.

Come suggerito in [19], il livello di persistenza è stato realizzato con 2 gerarchie di classi parallele: una contenente le interfacce e una contenente l'implementazione. Le operazioni comuni a tutte le entità del database sono raggruppate in una generica superinterfaccia e vengono implementate da una superclasse che usa una particolare tecnologia di persistenza (in questa tesi si è usato Hibernate ma si sarebbe potuto

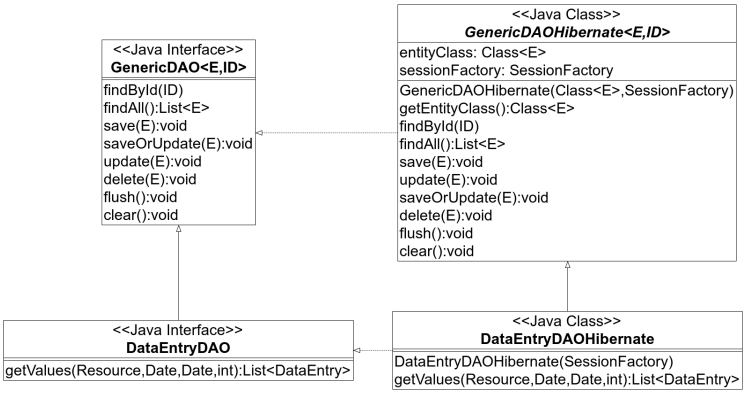


Figura 5.1: Esempio di come vengono estese la superclasse astratta e la superinterfaccia.

sfruttare un qualsiasi altro sistema per memorizzare i dati, per esempio una libreria che li salvi su file XML). L'interfaccia generica viene poi estesa da altre interfacce, una per ogni particolare entità del database che richieda ulteriori operazioni sui dati. Di nuovo, queste interfacce hanno una rispettiva classe che le implementa come nell'esempio in figura 5.1.

Codice 5.2: Superinterfaccia che definisce le generiche operazioni sulle entità

```
1  public interface GenericDAO<E, ID extends Serializable> {
2
3      E findById(ID id);
4
5      List<E> findAll();
6
7      void save(E entity);
8
9      void saveOrUpdate(E entity);
10
11     void update(E entity);
12
13     void delete(E entity);
14
15     void flush();
16
17     void clear();
18
19 }
```

Nel listato 5.2 si può vedere come viene definita la superinterfaccia `GenericDAO`, utilizzabile per implementare le operazioni di persistenza. Il primo parametro `E` rappresenta l'entità per cui si vuole implementare il DAO. Il secondo parametro `ID` definisce il tipo dell'identificatore usato nel database: entità diverse possono avere infatti identificatori diversi. Tale interfaccia viene implementata facendo uso di Hibernate,

come mostrato nel listato 5.3.

Codice 5.3: Superclasse che implementa la generica interfaccia usando le API di Hibernate

```
/**
 * Implements the generic CRUD data access operations using Hibernate
 * APIs. To write a DAO, subclass and parameterize this class with your
 * persistent class. Of course, assuming that you have a traditional
 * 1:1 appraoch for Entity:DAO design.
 */
public abstract class GenericDAOHibernate<E, ID extends Serializable>
        implements GenericDAO<E, ID> {

 private final Class<E> entityClass;
 private final SessionFactory sessionFactory;

 public GenericDAOHibernate(Class<E> entityClass,
   SessionFactory sessionFactory) {
  this.entityClass = entityClass;
  this.sessionFactory = sessionFactory;
 }

    protected Session getSession() {
     return sessionFactory.getCurrentSession();
    }

    public Class<E> getEntityClass() {
        return entityClass;
    }

    @SuppressWarnings({ "unchecked" })
    public E findById(ID id) {
     return (E) getSession().get(getEntityClass(), id);
    }

    public List<E> findAll() {
        return findByCriteria();
    }

    public void save(E entity) {
        getSession().save(entity);
    }

    public void update(E entity) {
     getSession().update(entity);
    }

    public void saveOrUpdate(E entity) {
        getSession().saveOrUpdate(entity);
    }

```

```
48      public void delete(E entity) {
49          getSession().delete(entity);
50      }
51
52      public void flush() {
53          getSession().flush();
54      }
55
56      public void clear() {
57          getSession().clear();
58      }
59
60      /**
61       * Use this inside subclasses as a convenience method.
62       */
63      @SuppressWarnings("unchecked")
64      protected List<E> findByCriteria(Criterion... criterion) {
65          Criteria crit = getSession().createCriteria(getEntityClass());
66          for (Criterion c : criterion) {
67              crit.add(c);
68          }
69          return crit.list();
70      }
71  }
```

Per ogni entità si definisce poi una interfaccia che estende il `GenericDAO`, definendo eventualmente altre operazioni specifiche (vedi listato 5.4), e una sua implementazione che estende il `GenericDAOHibernate` (vedi listato 5.5) ereditandone i metodi della superclasse. In questo ultimo esempio si può osservare inoltre come l'implementazione attraverso le *Criteria API* di Hibernate permetta di eseguire delle query, anche complesse, senza il bisogno di scrivere codice SQL. I significato dei metodi messi a disposizione inoltre è di semplice comprensione: nel caso specifico si stanno selezionando le data entry relative ad una risorsa passata come parametro, eventualmente limitando i valori ritornati nel loro numero e all'interno di un determinato periodo di tempo.

Codice 5.4: Esempio di estensione della generica interfaccia GenericDAO

```
1  public interface DataEntryDAO extends GenericDAO<DataEntry, Long> {
2
3   List<DataEntry> getValues(Resource resource, Date start, Date end, int
        limit);
4  }
```

Codice 5.5: Esempio di classe che implementa l'interfaccia definita per la specifica entità

```
1  @Repository
2  public class DataEntryDAOHibernate
3   extends GenericDAOHibernate<DataEntry, Long>
4   implements DataEntryDAO {
```

```
5
6  @Autowired
7  public DataEntryDAOHibernate(SessionFactory sessionFactory) {
8   super(DataEntry.class, sessionFactory);
9  }
10
11 @SuppressWarnings("unchecked")
12 public List<DataEntry> getValues(Resource resource, Date start, Date end,
       int limit) {
13  Criteria crit = getSession().createCriteria(getEntityClass());
14        crit.add( Restrictions.eq("resource", resource));
15        if (start!=null) crit.add( Restrictions.ge("valueTimestamp", start
              ));
16        if (end!=null) crit.add( Restrictions.le("valueTimestamp", end));
17        if (limit!=0) crit.setMaxResults(limit);
18        crit.addOrder(Order.desc("id"));
19        return crit.list();
20 }
21 }
```

5.5.2 La sessione di Hibernate

Come si vede nell'implementazione dei DAO, tutti le operazioni col database passano attraverso un oggetto *Session*. Questo è un oggetto che rappresenta una singola unità di lavoro verso il database e fornisce le API di Hibernate per caricare e salvare gli oggetti. Internamente è composto da una coda di comandi SQL che devono essere, in qualche momento, lanciati nel database per sincronizzare gli oggetti di dominio con le tabelle. Per ottenere una sessione è necessario avere una *SessionFactory*, ossia un oggetto che rappresenta una particolare configurazione di Hibernate che comprende:

- sorgente di dati da utilizzare per la connessione con il database;
- dialetto SQL da usare per l'ottimizzazione delle query;
- mapping tra le classi e le tabelle del database (ottenuto attraverso le annotazioni dei `JavaBean`);
- ulteriori altre informazioni;

5.5.3 Integrazione di Spring con Hibernate

A questo punto entra in gioco l'integrazione con Spring Framework che, fornendo supporto ad Hibernate, permette di iniettare nei DAO un oggetto *SessionFactory* (si veda il concetto di dependency injection in 3.4.4). Tutto questo processo si traduce nell'introduzione nel file di configurazione di Spring delle definizioni presenti nel listato 5.6.

Codice 5.6: Esempio di configurazione della SessionFactory

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/iot" />
  <property name="username" value="xxx" />
  <property name="password" value="xxx" />
  <!-- POOL PROPERTIES -->
  <property name="removeAbandoned" value="true"/>
  <property name="initialSize" value="10" />
  <property name="maxActive" value="30" />
</bean>

<bean id="mySessionFactory"
  class="org.springframework.orm.hibernate3.annotation
     .AnnotationSessionFactoryBean">
  <property name="dataSource" ref="myDataSource" />
  <property name="packagesToScan">
    <list>
      <value>com.patavina.apiserver.domain</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect
         .MySQL5InnoDBDialect</prop>
      <prop key="hibernate.format_sql">true</prop>
      <prop key="hibernate.show_sql">false</prop>
      <prop key="hibernate.use_sql_comments">true</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
      <prop key="hibernate.connection.autocommit">false</prop>
    </props>
  </property>
</bean>
```

Come si può vedere il componente `id` "myDataSource" definisce le caratteristiche della sorgente dei dati, specificando il driver da usare, il nome utente, la password e la URL a cui connettersi per avere accesso al database. Viene definito inoltre un *connection pool*, ossia una sorta di cache che mantiene aperte le connessioni con il database e minimizza i costi di aprirle e chiuderle. Generalmente, infatti, non è efficiente creare una connessione ogni volta che si vuole interagire con il database. Con l'uso di un *connection pool* ogni thread dell'applicazione che abbia bisogno di interagire con la base di dati, può chiedere una connessione al *pool* e restituirla quando le sue operazioni SQL sono terminate. Nella definizione del componente con `mySessionFactory` si osservano:

- l'iniezione della sorgente dei dati appena definita;
- la definizione del `package` dove trovare le classi di dominio annotate per il

mapping con le tabelle del database;

- la descrizione di un insieme di proprietà fra cui il dialetto del linguaggio SQL da utilizzare;

5.6 Sicurezza

Nella realizzazione del sistema si sono tenute in considerazione anche questioni relative alla sicurezza, in particolare nella fase di comunicazione tra client e API server. É stato implementato infatti un sistema di autenticazione delle richieste in ingresso e di gestione dei permessi sui dispositivi registrati al sistema. Inoltre si prevede in futuro di rendere sicure tutte le richieste sfruttando SSL.

5.6.1 Autenticazione

Il modello tradizionale di autenticazione Client-Server usa le credenziali dell'utente (tipicamente username e password) per accedere alle risorse private che sono ospitate nel server. Con questo metodo di autenticazione, l'uso di applicazioni sviluppate da terzi parti pone chiari problemi di sicurezza:

- la password non è più un segreto tra servizio e utente, viene salvata anche dall'applicazione;
- l'applicazione client ha totale accesso alle informazioni riservate;
- l'autorizzazione rimane in possesso dell'applicazione client fino al cambiamento della password da parte dell'utente;

Per ovviare a questi problemi si è scelto di implementare la stessa soluzione usata da *SugarSync* [18], il noto servizio di file hosting e sincronizzazione automatica di file tramite web. Un utente che voglia creare la propria applicazione deve per prima cosa registrarsi come sviluppatore al servizio offerto da Patavina Technologies. Successivamente attraverso una interfaccia web deve registrare la propria applicazione al sistema. Quest'ultimo genera automaticamente una coppia di chiavi chiamate Access Key ID e Private Access Key che forniscono una connessione autorizzata esclusivamente per l'applicazione che l'utente ha registrato.

L'applicazione ha poi bisogno di essere autorizzata per accedere alle risorse dell'utente gestite dalla piattaforma attraverso le API. A questo scopo, per ogni utente per una data applicazione, è necessario creare un *refresh token*, ossia un identificatore persistente che permette alle applicazioni di autenticare gli utenti alla piattaforma senza che l'applicazione debba memorizzare le sue credenziali. Quando un utente esegue l'applicazione per la prima volta, viene creato un refresh token inviando una richiesta `POST` che include le credenziali dell'utente all'API server (vedi B.4.1). Se la

richiesta va a buon fine, l'API server autorizza l'applicazione ad accedere all'account dell'utente e ritorna un refresh token.

Dopo che l'applicazione ha ottenuto un refresh token, può usarlo per richiedere un *access token* (vedi richiesta B.4.2), ossia una chiave temporanea che consente l'accesso alle risorse dell'utente per un periodo limitato di tempo (in genere un'ora). Il refresh token è pensato per essere salvato all'interno dell'applicazione e riusato per richiedere di volta in volta un access token. Per questo motivo l'applicazione non avrà bisogno di salvare le credenziali degli utenti ma solo i refresh token.

Capitolo 6

Test e performace

In questa sezione viene descritto come è stata organizzata la fase di test e vengono presentati i risultati ottenuti.

6.1 Test funzionali

In un primo momento si è valutata la correttezza del software attraverso dei test funzionali, ossia si è verificato, richiesta per richiesta, che le risposte ottenute dal web service fossero quelle che ci si aspettava. Per far questo è stato necessario realizzare uno script per popolare il database con dei valori casuali verosimili, in modo da simulare una situazione reale. Dunque si è creata una classe Java che genera uno script SQL `populate.sql` che contiene tutte le istruzioni di inserimento dati. Tale classe riceve come parametri il numero di: utenti, device fisici, device logici, risorse e data entry da inserire. In questo modo è possibile popolare il database con istanze di diverse dimensioni, cosa utile per valutare successivamente la scalabilità del sistema. Sono stati poi creati degli script in Python, che simulano il funzionamento dei client, interrogando il server mediante richieste HTTP strutturate come mostrato in appendice B.

6.2 Test prestazionali

Oltre ai test funzionali, si è deciso di valutare anche le prestazioni del sistema, concentrandosi in particolar modo su quelle del database. Per far questo è stato simulato un ambiente ideale, dove database e API server erano posizionati sulla stessa macchina, in modo che il loro tempo di comunicazione fosse trascurabile rispetto al tempo di esecuzione delle query. Nella predisposizione dell'ambiente si è inoltre cercato di disabilitare ogni tipo di cache, in quanto queste introducono variazioni sensibili e non controllabili nei valori osservati. La cache di MySQL è stata disabilitata dal server mentre per Hibernate non è stato possibile, poiché prevista dal suo funzionamento. Si

ritiene che questo sia il principale motivo per cui si ottengono alti valori di varianza nei risultati.

6.2.1 Sessione tipica

Nell'esecuzione dei test delle performance si sono tenute in considerazione le operazioni più frequenti che un client esegue in una tipica sessione. Queste sono:

- **operazione 1**: visualizzazione dei device logici associati all'utente che esegue la richiesta. Solo i device logici che attualmente hanno una associazione attiva con un device fisico vengono mostrati. Questo operazione viene eseguita attraverso la richiesta B.2.5;
- **operazione 2**: dalla lista dei dispositivi logici ricevuta, visualizzare le risorse associate ad un particolare device. Questo avviene per mezzo della richiesta B.3.6;
- **operazione 3**: visualizzazione degli ultimi 100 valori monitorati da una particolare risorsa. Questo avviene attraverso la richiesta B.3.8;

I test sono stati organizzati in diverse fasi, ad ognuna delle quali è stata aumentata la dimensione dell'istanza del database. Affinché i test fossero verosimili si è scelto di imporre la presenza di 15 risorse per ciascun dispositivo logico registrato e 100 data entry associate a ciascuna risorsa. Fissati questi vincoli, si è fatto variare il numero di dispositivi logici registrati. Le configurazioni risultanti prevedono così le seguenti dimensioni:

- **configurazione 1**: 1000 dispositivi logici, 15000 risorse e 1500000 data entry
- **configurazione 2**: 2000 dispositivi logici, 30000 risorse e 3000000 data entry
- **configurazione 3**: 3000 dispositivi logici, 45000 risorse e 4500000 data entry
- **configurazione 4**: 4000 dispositivi logici, 60000 risorse e 6000000 data entry
- **configurazione 5**: 5000 dispositivi logici, 75000 risorse e 7500000 data entry

Il numero di utenti e applicazioni registrate al sistema è invece rimasto sempre invariato, poiché questo influisce solamente sul tempo impiegato per l'autenticazione. Per ciascuna operazione, nelle varie configurazioni, si è valutato il tempo medio di risposta a 100 richieste. In tutti i casi considerati il sistema presenta dei tempi di risposta che si comportano in modo sublineare o, al peggio, lineare con l'aumentare delle dimensioni del database. Vengono presentati di seguito i risultati ottenuti e i relativi grafici.

Operazione 1

La prima operazione visualizza i device logici associati all'utente che esegue la richiesta, aventi una associazione attiva con un dispositivo fisico. Per rispondere a una tale richiesta il sistema deve eseguire le seguenti query al database:

- sulla base dell'access token presente nell'header della richiesta, recuperare dalla tabella User App l'id dell'utente che l'ha eseguita;

- selezionare dalla tabella Logical Device tutti i dispositivi logici di cui quell'utente possiede i permessi (ossia tutti quelli di cui è il creatore) e che hanno una associazione logica attiva.

In tabella 6.1 vengono riportati i valori medi ottenuti eseguendo 100 richieste e in figura 6.1 viene riportato il grafico.

Tabella 6.1: Media e varianza osservate per l'operazione 1

Configurazione	$E[T_{op1}]$	$\sigma^2(T_{op1})$
1	19.049	48,318
2	30.438	116,390
3	35.308	113,017
4	39.268	118,998
5	43.617	145,755

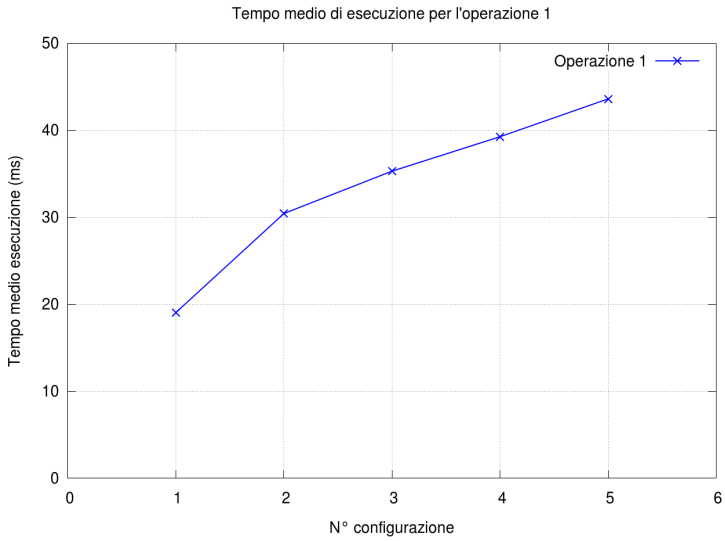


Figura 6.1: Tempo di esecuzione medio su 100 richieste per l'operazione 1.

Operazione 2

Dalla lista dei dispositivi logici ricevuta attraverso l'operazione 1, visualizza le risorse associate ad un particolare device. Per rispondere a una tale richiesta il sistema deve eseguire le seguenti query al database:

- sulla base dell'access token presente nell'header della richiesta, recuperare dalla tabella USER APP l'id dell'utente che l'ha eseguita;
- verificare l'esistenza del device logico richiesto nella tabella LOGICAL DEVICE;
- verificare che l'utente possieda i permessi e quindi sia il creatore del dispositivo;
- selezionare dalla tabella RESOURCE le risorse del dispositivo specificato;

In tabella 6.2 vengono riportati i valori medi ottenuti eseguendo 100 richieste e in figura 6.2 viene riportato il grafico.

Tabella 6.2: Media e varianza osservate per l'operazione 2

Configurazione	$E[T_{op2}]$	$\sigma^2(T_{op2})$
1	14.559	24,822
2	16.999	30,390
3	18.953	51,651
4	18.312	31,721
5	20.319	40,721

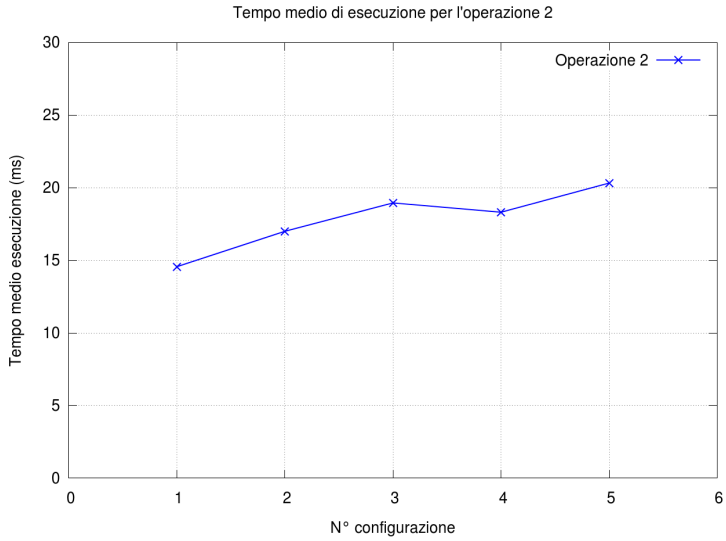


Figura 6.2: Tempo di esecuzione medio su 100 richieste per l'operazione 2.

Operazione 3

La terza operazione, dalla risorsa selezionata con l'operazione 2, visualizza gli ultimi 100 valori da essa monitorati. Le query necessarie al database sono:

- sulla base dell'access token presente nell'header della richiesta, recuperare dalla tabella User App l'id dell'utente che l'ha eseguita;
- verificare l'esistenza del device logico richiesto all'interno della tabella Logical Device;
- verificare che l'utente possieda i permessi e quindi sia il creatore del dispositivo;
- verificare l'esistenza della risorsa richiesta all'interno della tabella Resource;
- selezionare i valori osservati per la risorsa richiesta limitandola ai 100 più recenti.

In tabella 6.3 vengono riportati i valori medi ottenuti eseguendo 100 richieste e in figura 6.3 viene riportato il grafico.

Tabella 6.3: Media e varianza osservate per l'operazione 3

Configurazione	$E[T_{op3}]$	$\sigma^2(T_{op3})$
1	38.651	223,214
2	98.710	2126,967
3	142.552	3699,621
4	258.750	8195,583
5	328.154	11819,459

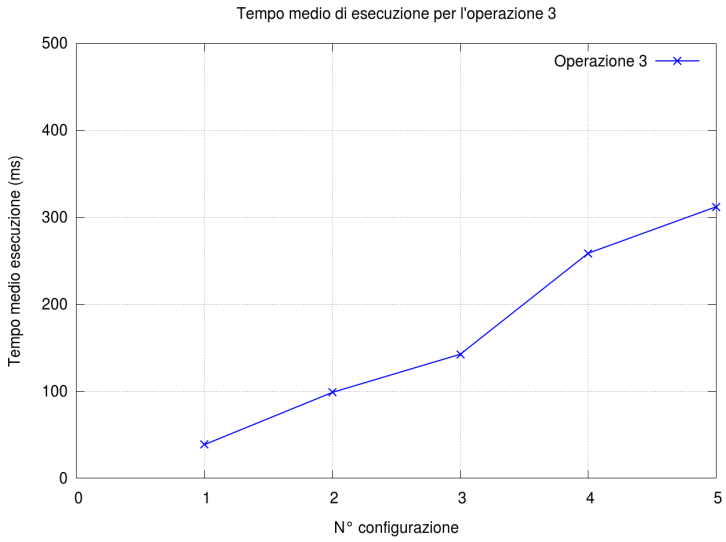


Figura 6.3: Tempo di esecuzione medio su 100 richieste per l'operazione 3.

6.2.2 Connection Pool

Si è deciso inoltre di valutare quanto incide l'uso di un connection pool per gestire le connessioni al database. Per far questo sono state eseguite 1000 richieste del tipo B.3.8, che richiedono i valori osservati a device scelti casualmente tra quelli registrati. Tali richieste sono state prima inviate in modo sequenziale da un singolo thread e successivamente da 10 e 20 thread parallelamente. In queste condizioni si è valutato il tempo impiegato dal sistema a rispondere alle 1000 richieste al variare della dimensione del connection pool. Avere un connection pool di dimensione 1 equivale a non utilizzarlo dato che esisterà un'unica connessione verso il database condivisa tra tutti i processi dell'API server che richiedono l'accesso ai dati. Con valori maggiori di 1 invece, il software mantiene aperto un certo numero di connessioni assegnando quelle libere ai processi che ne fanno richiesta.

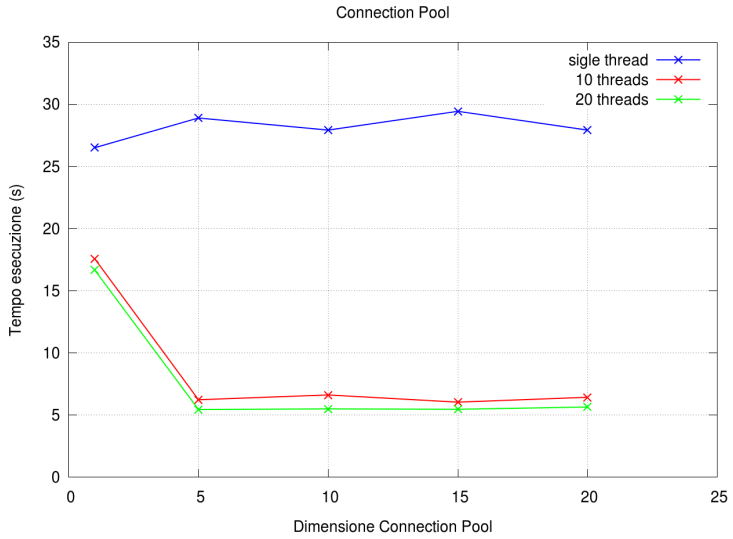


Figura 6.4: Tempo di risposta a 1000 richieste provenienti rispettivamente da 1, 10 e 20 thread in parallelo.

Come si può osservare dal grafico 6.4, nel caso di singolo thread l'aumento della dimensione del connection pool non porta alcun beneficio in quanto, essendo le richieste sequenziali, queste troveranno sempre una connessione libera verso il database. Nel caso di più richieste concorrenti invece, già con un pool di dimensione 5 il tempo di esecuzione diminuisce notevolmente. Il fatto che l'ulteriore aumento della sua dimensione non abbia effetti è da imputare al fatto che la velocità con cui giungono le richieste dagli script python, essendo i test eseguiti sulla macchina locale, non permettano di generare una mole di richieste verso database tale da occupare più di 5-10 connessioni contemporaneamente.

Capitolo 7

Conclusioni

L'Internet of Things è una rivoluzione tecnologica che sta cambiando completamente il modo con cui interagiamo con il mondo che ci circonda. L'espandersi delle reti di sensori ha infatti permesso di osservare e controllare fenomeni del mondo reale in un modo prima impensabile. Questa evoluzione però non è avvenuta in modo omogeneo: l'idea di Internet of Things è in effetti nata dalla convergenza di diversi filoni di ricerca e, per questo motivo, molte sono le tecnologie sviluppate parallelamente nel corso degli anni, ciascuna con le proprie caratteristiche e il proprio dominio applicativo. In un tale scenario, la sfida più grande risulta quella di riuscire ad integrare i diversi tipi di dispositivi, tecnologie e servizi, interconnettendoli ad un unica piattaforma accessibile al Web. Questo tesi vuole essere un esempio di come si può raggiungere questo obbiettivo, in particolare focalizzando l'attenzione sul protocollo CoAP, uno tra i più recenti e promettenti protocolli per la gestione di dispositivi constrained, caratterizzati da grandi limiti in termini di risorse. Il progetto è nato dalla collaborazione con l'azienda Patavina Technologies ed ha previsto la realizzazione di una piattaforma per la virtualizzazione e gestione di reti di sensori attraverso un web service. Nella prima fase, descritta nel capitolo 2, si è studiato il protocollo CoAP e sono state analizzate le soluzioni software aventi scopi simili, presenti sul mercato. Successivamente si è passati alla fase di analisi e si è definita l'architettura del sistema, che permette di creare un ponte tra le reti di sensori ed Internet. Per far questo il software è stato suddiviso in diverse parti, come è stato illustrato nel capitolo 3. Al livello più basso si trova il *Sensors Component*, composto da tutti i dispositivi che si occupano di raccogliere i dati nelle reti locali, ossia sensori e gateway. Essi sono stati forniti direttamente da Patavina Technologies ma, per ragioni di agilità dei test, sono stati implementati dei simulatori. Il *Data Component*, è poi adibito alla memorizzazione di tutte le informazioni che questi dispositivi inviano al sistema, creando per ciascuno di essi un profilo virtuale. In questo contesto si è dovuto definire il concetto di virtualizzazione, ossia in quale modo i device e le loro informazioni, sia logiche che fisiche, vengono mappate nel database. Il

cuore del sistema o *Core Component*, è costituito dall'API server, il quale si occupa di fornire una interfaccia HTTP aderente al paradigma REST che permette agli utenti di interagire con i profili virtuali dei sensori registrati. L'API server esegue inoltre tutte le elaborazioni pesanti computazionalmente, per esempio autenticazione, gestione delle notifiche, ecc. interagendo, ove serve, con le altre parti del sistema. Infine l'interfaccia utente risiede nel *Presentation Component*, la cui implementazione non era scopo di questa tesi. Si è scelto invece, di fornire un API server con il quale, attraverso lo scambio di messaggi in formato XML oppure JSON, sviluppatori di terze parti possano creare le proprie applicazioni.

Nello sviluppo di questo software si sono sempre tenuti in considerazione i principi di modularità, scalabilità e flessibilità. Per quanto riguarda la modularità, l'uso dei framework Spring ed Hibernate, il cui uso è ormai consolidato in ambito aziendale per la realizzazione di applicazioni così complesse, ha permesso di mantenere separati i componenti del sistema sulla base della loro funzione. Dal punto di vista della scalabilità, i risultati presentati nel capitolo 6 mostrano che i tempi di risposta delle operazioni che accedono ai dati hanno un andamento sublineare all'aumentare delle dimensioni del database. Questo significa che il sistema mantiene inalterate le proprie prestazioni all'aumentare di utenti e dispositivi registrati.

Per quanto riguarda gli sviluppi futuri si prevede l'implementazione della condivisione dei dispositivi, in modo che ciascun utente possa permettere l'accesso alle proprie risorse anche ad altri utenti della piattaforma. Oltre a ciò, ulteriori meccanismi di sicurezza devono essere presi in considerazione al fine di autenticare il dispositivo all'API server: non deve infatti essere possibile, per una entità malevola, impersonare un dispositivo ed inviare informazioni per suo conto. Infine, l'obbiettivo ultimo sarebbe quello di trasportare l'architettura realizzata in una piattaforma cloud-based capace di reggere condizioni di carico ben più pesanti di quelle con cui si sono eseguiti i test.

Appendice A

Implementazione SQL del database

Viene qui riportato lo script per la creazione del database.

```
DROP SCHEMA IF EXISTS `iot`;
CREATE SCHEMA `iot` DEFAULT CHARACTER SET latin1 ;
USE `iot` ;

-- -----------------------------------------------------
-- Table `iot`.`user`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`user` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `email` VARCHAR(255) NULL DEFAULT NULL ,
  `name` VARCHAR(255) NULL DEFAULT NULL ,
  `password` VARCHAR(32) NULL DEFAULT NULL ,
  `phone` VARCHAR(255) NULL DEFAULT NULL ,
  `registration_date` DATETIME NOT NULL ,
  `surname` VARCHAR(255) NULL DEFAULT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  UNIQUE INDEX `email` (`email` ASC) )
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`app`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`app` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `access_key_id` VARCHAR(36) NOT NULL ,
  `description` VARCHAR(255) NULL DEFAULT NULL ,
```

```
  `enabled` BIT(1) NOT NULL ,
  `name` VARCHAR(255) NOT NULL ,
  `private_access_key` VARCHAR(36) NOT NULL ,
  `developer` BIGINT(20) NULL DEFAULT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  UNIQUE INDEX `access_key_id` (`access_key_id` ASC) ,
  UNIQUE INDEX `private_access_key` (`private_access_key` ASC) ,
  INDEX `FK17A2188A2D071` (`developer` ASC) ,
  CONSTRAINT `FK17A2188A2D071`
    FOREIGN KEY (`developer` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`physical_device`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`physical_device` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `format` VARCHAR(10) NULL DEFAULT NULL ,
  `ip` VARCHAR(39) NOT NULL ,
  `lat` FLOAT NOT NULL ,
  `lon` FLOAT NOT NULL ,
  `mac` VARCHAR(17) NOT NULL ,
  `port` INT(11) NOT NULL ,
  `protocol` VARCHAR(10) NOT NULL ,
  `created_by` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  UNIQUE INDEX `mac` (`mac` ASC) ,
  UNIQUE INDEX `mac_2` (`mac` ASC) ,
  INDEX `FK1FCFEABEDF1591B5` (`created_by` ASC) ,
  CONSTRAINT `FK1FCFEABEDF1591B5`
    FOREIGN KEY (`created_by` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`logical_device`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`logical_device` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `active_association` BIT(1) NULL DEFAULT NULL ,
  `description` VARCHAR(255) NULL DEFAULT NULL ,
  `enabled` BIT(1) NULL DEFAULT NULL ,
  `name` VARCHAR(255) NOT NULL ,
```

```
  `created_by` BIGINT(20) NOT NULL ,
  `physical_id` BIGINT(20) NULL DEFAULT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  UNIQUE INDEX `name` (`name` ASC, `created_by` ASC) ,
  INDEX `FKA9D6934C4E54956C` (`physical_id` ASC) ,
  INDEX `FKA9D6934CDF1591B5` (`created_by` ASC) ,
  CONSTRAINT `FKA9D6934CDF1591B5`
    FOREIGN KEY (`created_by` )
    REFERENCES `iot`.`user` (`id` ),
  CONSTRAINT `FKA9D6934C4E54956C`
    FOREIGN KEY (`physical_id` )
    REFERENCES `iot`.`physical_device` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`resource`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`resource` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `description` VARCHAR(255) NULL DEFAULT NULL ,
  `expiration` INT(11) NULL DEFAULT NULL ,
  `measure_symbol` VARCHAR(255) NULL DEFAULT NULL ,
  `measure_unit` VARCHAR(255) NULL DEFAULT NULL ,
  `name` VARCHAR(255) NULL DEFAULT NULL ,
  `path` VARCHAR(255) NOT NULL ,
  `value` BLOB NULL DEFAULT NULL ,
  `value_timestamp` DATETIME NULL DEFAULT NULL ,
  `device_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  UNIQUE INDEX `device_id` (`device_id` ASC, `path` ASC) ,
  INDEX `FKEBABC40E7D292207` (`device_id` ASC) ,
  CONSTRAINT `FKEBABC40E7D292207`
    FOREIGN KEY (`device_id` )
    REFERENCES `iot`.`logical_device` (`id` )
    ON DELETE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`data_entry`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`data_entry` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `value` BLOB NOT NULL ,
  `value_timestamp` DATETIME NOT NULL ,
```

```
  `resource_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  INDEX `FK5FC2739D80BA7DD6` (`resource_id` ASC) ,
  CONSTRAINT `FK5FC2739D80BA7DD6`
    FOREIGN KEY (`resource_id` )
    REFERENCES `iot`.`resource` (`id` )
    ON DELETE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`default_device_tag`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`default_device_tag` (
  `device_id` BIGINT(20) NOT NULL ,
  `tag` VARCHAR(50) NOT NULL ,
  PRIMARY KEY (`device_id`, `tag`) ,
  INDEX `FK4DD8AEEF7D292207` (`device_id` ASC) ,
  CONSTRAINT `FK4DD8AEEF7D292207`
    FOREIGN KEY (`device_id` )
    REFERENCES `iot`.`logical_device` (`id` )
    ON DELETE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`default_resource_tag`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`default_resource_tag` (
  `resource_id` BIGINT(20) NOT NULL ,
  `tag` VARCHAR(50) NOT NULL ,
  PRIMARY KEY (`resource_id`, `tag`) ,
  INDEX `FK4663812780BA7DD6` (`resource_id` ASC) ,
  CONSTRAINT `FK4663812780BA7DD6`
    FOREIGN KEY (`resource_id` )
    REFERENCES `iot`.`resource` (`id` )
    ON DELETE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`device_tag`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`device_tag` (
  `device_id` BIGINT(20) NOT NULL ,
  `tag` VARCHAR(50) NOT NULL ,
```

```
  `user_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`device_id`, `tag`, `user_id`) ,
  INDEX `FK2E94F23184ACD6B6` (`user_id` ASC) ,
  INDEX `FK2E94F2317D292207` (`device_id` ASC) ,
  CONSTRAINT `FK2E94F2317D292207`
    FOREIGN KEY (`device_id` )
    REFERENCES `iot`.`logical_device` (`id` )
    ON DELETE CASCADE,
  CONSTRAINT `FK2E94F23184ACD6B6`
    FOREIGN KEY (`user_id` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`gateway`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`gateway` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `ip` VARCHAR(255) NULL DEFAULT NULL ,
  `port` VARCHAR(255) NULL DEFAULT NULL ,
  `priority` INT(11) NULL DEFAULT NULL ,
  `physical_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  INDEX `FKF4BA46444E54956C` (`physical_id` ASC) ,
  CONSTRAINT `FKF4BA46444E54956C`
    FOREIGN KEY (`physical_id` )
    REFERENCES `iot`.`physical_device` (`id` )
    ON DELETE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`resource_tag`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`resource_tag` (
  `resource_id` BIGINT(20) NOT NULL ,
  `tag` VARCHAR(50) NOT NULL ,
  `user_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`resource_id`, `tag`, `user_id`) ,
  INDEX `FKE91BFBE984ACD6B6` (`user_id` ASC) ,
  INDEX `FKE91BFBE980BA7DD6` (`resource_id` ASC) ,
  CONSTRAINT `FKE91BFBE980BA7DD6`
    FOREIGN KEY (`resource_id` )
    REFERENCES `iot`.`resource` (`id` )
    ON DELETE CASCADE,
  CONSTRAINT `FKE91BFBE984ACD6B6`
```

```
    FOREIGN KEY (`user_id` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`trigger`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`trigger` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `is_active` BIT(1) NULL DEFAULT NULL ,
  `creation_time` DATETIME NOT NULL ,
  `description` VARCHAR(255) NULL DEFAULT NULL ,
  `java_condition` VARCHAR(255) NOT NULL ,
  `name` VARCHAR(255) NOT NULL ,
  `created_by` BIGINT(20) NULL DEFAULT NULL ,
  `resource_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  INDEX `FKC0D355B8DF1591B5` (`created_by` ASC) ,
  INDEX `FKC0D355B880BA7DD6` (`resource_id` ASC) ,
  CONSTRAINT `FKC0D355B880BA7DD6`
    FOREIGN KEY (`resource_id` )
    REFERENCES `iot`.`resource` (`id` )
    ON DELETE CASCADE,
  CONSTRAINT `FKC0D355B8DF1591B5`
    FOREIGN KEY (`created_by` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`trigger_action`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`trigger_action` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `content` VARCHAR(255) NULL DEFAULT NULL ,
  `target` VARCHAR(255) NOT NULL ,
  `type` VARCHAR(6) NOT NULL ,
  `trigger_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  INDEX `FKF62AC69D35907C5E` (`trigger_id` ASC) ,
  CONSTRAINT `FKF62AC69D35907C5E`
    FOREIGN KEY (`trigger_id` )
    REFERENCES `iot`.`trigger` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;
```

```
-- -----------------------------------------------------
-- Table `iot`.`trigger_log`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`trigger_log` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT ,
  `activation_timestamp` DATETIME NOT NULL ,
  `message_sent` VARCHAR(255) NOT NULL ,
  `trigger_id` BIGINT(20) NOT NULL ,
  PRIMARY KEY (`id`) ,
  UNIQUE INDEX `id` (`id` ASC) ,
  INDEX `FKD76E8C3D35907C5E` (`trigger_id` ASC) ,
  CONSTRAINT `FKD76E8C3D35907C5E`
    FOREIGN KEY (`trigger_id` )
    REFERENCES `iot`.`trigger` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;


-- -----------------------------------------------------
-- Table `iot`.`user_app`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `iot`.`user_app` (
  `app_id` BIGINT(20) NOT NULL ,
  `user_id` BIGINT(20) NOT NULL ,
  `access_token` VARCHAR(36) NOT NULL ,
  `access_token_expiration` DATETIME NOT NULL ,
  `refresh_token` VARCHAR(36) NOT NULL ,
  PRIMARY KEY (`app_id`, `user_id`) ,
  UNIQUE INDEX `access_token` (`access_token` ASC) ,
  UNIQUE INDEX `refresh_token` (`refresh_token` ASC) ,
  INDEX `FKF022B7AD84ACD6B6` (`user_id` ASC) ,
  INDEX `FKF022B7AD872A29BE` (`app_id` ASC) ,
  CONSTRAINT `FKF022B7AD872A29BE`
    FOREIGN KEY (`app_id` )
    REFERENCES `iot`.`app` (`id` ),
  CONSTRAINT `FKF022B7AD84ACD6B6`
    FOREIGN KEY (`user_id` )
    REFERENCES `iot`.`user` (`id` ))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;
```

Appendice B

Documentazione API

In questa sezione vengono documentate le API con cui i client, attraverso richieste HTTP, possono interagire con la piattaforma. In ciascuna richiesta deve essere presente l'header `key`, contenente l'`access_token` che identifica utente e applicazione da cui la richiesta proviene.

B.1 Device fisici

B.1.1 Registrazione

Registra un device fisico al sistema.

Richiesta

URL	`/physical_device`
ACTION	`POST`
FORMAT	`XML, JSON`

Corpo della richiesta

```
<?xml version="1.0" encoding="UTF-8"?>
<physicalDevice>
  <mac>11:22:33:44:55:66</mac>
  <ip>192.168.1.10</ip>
  <port>5683</port>
  <lat>45.087096</lat>
  <lon>11.6922</lon>
  <protocol>coap</protocol>
  <gateway>
    <ip>1.2.3.4</ip>
    <port>5555</port>
  </gateway>
```

```
</physicalDevice>
```

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `201 Created` e un messaggio contenente l'id fornito per quel device fisico.

```
<?xml version="1.0" encoding="UTF-8"?>
<registrationResponse>
   <provisionedId>132</provisionedId>
</registrationResponse>
```

B.1.2 Aggiornamento

Aggiorna le informazioni fisiche di un dispositivo.

Richiesta

URL	/physical_device/*{physicalDeviceId}*
ACTION	`PUT`
FORMAT	`XML, JSON`

Corpo del messaggio

Lo stesso usato in fase di registrazione.

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.1.3 Descrizione

Recupera la descrizione del dispositivo fisico. L'id non viene incluso nel file XML di risposta in quanto ridondante.

Richiesta

URL	/physical_device/*{physicalDeviceId}*
ACTION	`GET`
FORMAT	`XML, JSON`

Risposta

```
<?xml version="1.0" encoding="UTF-8"?>
<physicalDevice>
  <mac>11:22:33:44:55:66</mac>
  <ip>192.168.1.10</ip>
  <port>5683</port>
  <lat>45.087096</lat>
  <lon>11.6922</lon>
  <protocol>coap</protocol>
  <gateway>
    <ip>1.2.3.4</ip>
    <port>5555</port>
  </gateway>
</physicalDevice>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK` e un messaggio contenente la descrizione completa del device fisico.

B.1.4 Cancellazione

La cancellazione è permessa a patto che non ci sia alcun device logico associato. Dunque è necessario eliminare prima tutti i device logici associati, prima di poter eliminare un device fisico.

Richiesta

URL	`/physical_device/{physicalDeviceId}`
ACTION	`DELETE`
FORMAT	`XML, JSON`

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.2 Device Logici

B.2.1 Registrazione

Registra un device logico al sistema. Nel file xml di registrazione deve essere indicato il device fisico a cui associare il device logico che si sta registrando. L'associazione viene eseguita in modo automatico. Se il device fisico ha già un device logico associato viene restituito un errore. In tal caso è necessario dissociare il precedente device logico e ritentare la registrazione.

Richiesta

URL	`/logical_device`
ACTION	`POST`
FORMAT	`XML, JSON`

Corpo della richiesta

```
<?xml version="1.0" encoding="UTF-8"?>
<logicalDeviceRequest>
  <name>logical dev 1</name>
  <description>the temperature and the pressure of a room</description>
  <enabled>true</enabled>
  <physicalDeviceId>1</physicalDeviceId>
  <defaultTags>
    <tag>default tag 1</tag>
    <tag>default tag 2</tag>
    <tag>default tag 3</tag>
  </defaultTags>
  <myTags>
    <tag>my tag 1</tag>
    <tag>my tag 2</tag>
  </myTags>
</logicalDeviceRequest>
```

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `201 Created` e un messaggio contenente l'id fornito per quel device logico.

```
<?xml version="1.0" encoding="UTF-8"?>
<registrationResponse>
   <provisionedId>132</provisionedId>
</registrationResponse>
```

B.2.2 Aggiornamento

Aggiorna le informazioni logiche di un dispositivo.

Richiesta

URL	`/logical_device/{logicalDeviceId}`
ACTION	`PUT`
FORMAT	`XML, JSON`

Corpo del messaggio

Lo stesso usato in fase di registrazione.

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.2.3 Descrizione

Recupera la descrizione del dispositivo logico identificato dal `logicalDeviceId`. L'id non viene incluso nel file XML di risposta in quanto ridondante.

Richiesta

URL	`/logical_device/{logicalDeviceId}`
ACTION	`GET`
FORMAT	`XML, JSON`

Risposta

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<logicalDeviceResponse>
  <id>3</id>
  <name>logical dev 1</name>
  <description>the temperature and the pressure of a room</description>
  <creatorId>1</creatorId>
  <physicalDeviceId>1</physicalDeviceId>
  <enabled>true</enabled>
  <defaultTags>
    <tag>default tag 1</tag>
    <tag>default tag 2</tag>
    <tag>default tag 3</tag>
  </defaultTags>
  <myTags>
    <tag>my tag 1</tag>
    <tag>my tag 2</tag>
  </myTags>
</logicalDeviceResponse>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK` e un messaggio contenente la descrizione completa del device logico.

B.2.4 Cancellazione

La cancellazione di un device logico comporta la cancellazione di tutte le risorse associate e di conseguenza anche le data entry e i trigger associati.

Richiesta

URL	/logical_device/{logicalDeviceId}
ACTION	DELETE
FORMAT	XML, JSON

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status 200 OK.

B.2.5 Lista dispositivi attivi

Recupera la lista dei dispositivi logici registrati dall'utente identificato dall'header key e che hanno attualmente una associazione attiva con un device fisico.

Richiesta

URL	/logical_device/
ACTION	GET
FORMAT	XML, JSON

Parametri

tags	lista dei tag separati da virgola per raffinare la ricerca

Risposta

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<logicalDeviceList>
  <logicalDeviceResponse>
    <name>logical dev 1</name>
    <description>the temperature and the pressure of a room</description>
    <enabled>true</enabled>
    <physicalDeviceId>1</physicalDeviceId>
    <defaultTags>
      <tag>default tag 1</tag>
      <tag>default tag 2</tag>
      <tag>default tag 3</tag>
    </defaultTags>
    <myTags>
      <tag>my tag 1</tag>
      <tag>my tag 2</tag>
    </myTags>
  </logicalDeviceResponse>
  <logicalDeviceResponse>
    <name>logical dev 2</name>
    <description>humidity of another room</description>
    <enabled>true</enabled>
```

```
    <physicalDeviceId>2</physicalDeviceId>
    </defaultTags>
    <myTags>
      <tag>my tag 1</tag>
    </myTags>
  </logicalDeviceResponse>
</logicalDeviceList>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.2.6 Dissociazione da device fisico

Con l'operazione di dissociazione il device logico viene scollegato dal device fisico. In questo modo è possibile "montare" un altro device logico su quel particolare device fisico. Il database tiene traccia dell'associazione con il device fisico ma lo marca come inattivo (questo per poter riassociare successivamente quel device logico).

Richiesta

URL	`/logical_device/`*`{logicalDeviceId}`*`/active_association`
ACTION	`DELETE`
FORMAT	`XML, JSON`

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.2.7 Associazione a device fisico

Con l'operazione di associazione è possibile riagganciare un device logico ad un device fisico. Affinchè l'operazione vada a buon fine, il device fisico deve essere "libero" ossia non deve essere già associato ad un altro device logico. L'operazione di associazione è implicita al momento della registrazione di un nuovo device logico.

Richiesta

URL	`/logical_device/`*`{logicalDeviceId}`*`/active_association/`*`{physicalDeviceId}`*
ACTION	`POST`
FORMAT	`XML, JSON`

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.3 Risorse

B.3.1 Registrazione manuale

Richiesta

URL	/resource/{logicalDeviceId}?source=user
ACTION	POST
FORMAT	XML, JSON

Corpo della richiesta

```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
  <resource>
    <path>/sensor/temp/</path>
    <name>temperature</name>
    <description>the temperature of the room</description>
    <measureUnit>celsius</measureUnit>
    <measureSymbol>°C</measureSymbol>
    <expiration>60000</expiration>
    <defaultTags>
      <tag>default tag 1</tag>
    </defaultTags>
  </resource>
  <resource>
    <path>/sensor/pressure/</path>
    <name>pressure</name>
    <description>the pressure of the room</description>
    <measureUnit>celsius</measureUnit>
    <measureSymbol>°C</measureSymbol>
    <expiration></expiration>
    <myTags>
      <tag>my tag 1</tag>
      <tag>my tag 2</tag>
    </myTags>
  </resource>
  <resource>
    <path>/sensor/humy/</path>
    <name>humy</name>
    <description>the humidity of the room</description>
    <measureUnit>celsius</measureUnit>
    <measureSymbol>°C</measureSymbol>
    <expiration></expiration>
  </resource>
</resources>
```

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `201 Created`.

B.3.2 Registrazione automatica

Con questa chiamata l'API server richiede al device il file `/.well-known/core` contenente la descrizione delle risorse.

Richiesta

URL	`/resource/{logicalDeviceId}?source=coapDevice`
ACTION	`POST`
FORMAT	`XML, JSON`

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `201 Created`.

B.3.3 Aggiornamento

Aggiorna le informazioni logiche di una risorsa

Richiesta

URL	`/resource/{logicalDeviceId}`
ACTION	`PUT`
FORMAT	`XML, JSON`

Corpo del messaggio

Una singola risorsa tra quelle definite nel file di registrazione manuale.

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.3.4 Descrizione

Recupera la descrizione della risorsa identificata dal `path` all'interno del device logico identificato da `logicalDeviceId`.

Richiesta

URL	`/resource/{logicalDeviceId}/{path}`
ACTION	`GET`
FORMAT	`XML, JSON`

Risposta

```
<?xml version="1.0" encoding="UTF-8"?>
<resource>
  <id>35</id>
  <logicalDeviceId>2</logicalDeviceId>
  <path>/sensor/temp/</path>
  <name>temp</name>
  <description>the temperature of the room</description>
  <measureUnit>celsius</measureUnit>
  <measureSymbol>$^\circ$C</measureSymbol>
  <value>24</value>
  <valueTimestamp>2013-04-23T12:01:50+02:00</valueTimestamp>
  <expiration>60000</expiration>
  <defaultTags>
    <tag>default tag 1</tag>
  </defaultTags>
</resource>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK` e un messaggio contenente la descrizione completa della risorsa.

B.3.5 Cancellazione

La cancellazione di una risorsa comporta la cancellazione di tutti i trigger e delle data entry associate.

Richiesta

URL	`/resource/{logicalDeviceId}/{path}`
ACTION	`DELETE`
FORMAT	`XML, JSON`

Risposta

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.3.6 Lista delle risorse di un dispositivo

Recupera la lista delle risorse associate ad un particolare dispositivo

Richiesta

URL	`/resource/{logicalDeviceId}`
ACTION	`GET`
FORMAT	`XML, JSON`

Risposta

Una lista di risorse come sono state definite nella sezione B.3.4. Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.3.7 Interrogazione stato della risorsa

Richiede al sistema l'ultimo valore osservato per la risorsa. Se il valore nel database è ancora valido non è necessario richiederlo ad dispositivo.

Richiesta

URL	`/resource/lastValue/{logicalDeviceId}/{path}`
ACTION	`GET`
FORMAT	`XML, JSON`

Risposta

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dataEntry>
  <expiration>2013-04-23T16:00:34+01:00</expiration>
  <value>value</value>
  <valueTimestamp>2013-04-23T15:45:34+01:00</valueTimestamp>
</dataEntry>
```

Nonostante nell'esempio sia in formato stringa, il contenuto del tag `<value>` viene ritornato come array di byte codificato in base 64.

Parametri

force	flag che specifica se forzare la richiesta del dato al dispositivo

B.3.8 Interrogazione valori di una risorsa

Richiesta

URL	`/resource/values/{logicalDeviceId}/{path}`
ACTION	`GET`
FORMAT	`XML, JSON`

Parametri

`limit`	indica il numero di valori che devono essere ritornati
`start`	da che punto temporale restituire i valori
`stop`	fino a che punto temporale restituire i valori

Risposta

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dataEntries>
  <dataEntry>
    <value>value1</value>
    <valueTimestamp>2013-04-23T15:45:34+01:00</valueTimestamp>
  </dataEntry>
  <dataEntry>
    <value>value2</value>
    <valueTimestamp>2013-04-23T15:46:10+01:00</valueTimestamp>
  </dataEntry>
<dataEntries>
```

B.4 Autenticazione

Le seguenti richieste non prevedono l'header `key` settato dato che mirano proprio a richiedere un `access_token`.

B.4.1 Creare un Refresh Token

Richiesta

URL	`/app-authorization`
ACTION	`POST`
FORMAT	`XML, JSON`

Corpo del messaggio

```
<refreshTokenRequest>
  <username>user@patavinatech.it</username>
  <password>password</password>
  <applicationId>1</applicationId>
  <accessKeyId>
     af5831be-22cd-42bf-b733-3b857596546d
  </accessKeyId>
  <privateAccessKey>
     12636aa0-5fff-4f0c-8641-7d86a0906df9
  </privateAccessKey>
</refreshTokenRequest>
```

Nel corpo del messaggio sono presenti le credenziali dell'utente, l'id dell'applicazione e la coppia di chiavi che sono state associate all'applicazione.

Risposta

```
<refreshTokenResponse>
  <refreshToken>3629f5ae-8d61-4125-9316-ff8e0db424c9</refreshToken>
</refreshTokenResponse>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

B.4.2 Ottenere un Access Token

Richiesta

URL	`/authorization`
ACTION	`POST`
FORMAT	`XML, JSON`

Corpo del messaggio

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessTokenRequest>
  <refreshToken>
     3629f5ae-8d61-4125-9316-ff8e0db424c9
  </refreshToken>
  <accessKeyId>
     514c3710-3969-4460-91fa-02b17a1215d1
  </accessKeyId>
  <privateAccessKey>
     5933e6fb-3974-441f-8d48-543ca8235b97
  </privateAccessKey>
</accessTokenRequest>
```

Nel corpo del messaggio sono presenti il refresh token e la coppia di chiavi.

Risposta

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessTokenResponse>
  <accessToken>fcdab729-ff59-47d3-85d3-1919deb06951</accessToken>
  <expiration>2013-04-23T11:23:08+01:00</expiration>
  <user>user@patavinatech.it</user>
</accessTokenResponse>
```

Se l'operazione va a buon fine alla risposta è associato lo status `200 OK`.

Bibliografia

[1] Luigi Atzori, Antonio Iera e Giacomo Morabito. "The internet of things: A survey". In: *Computer Networks* 54.15 (2010), pp. 2787–2805 (cit. a p. 1).

[2] Michele Zorzi et al. "From today's intranet of things to a future internet of things: a wireless-and mobility-related view". In: *Wireless Communications, IEEE* 17.6 (2010), pp. 44–51 (cit. a p. 1).

[3] C. Buratti. *Sensor Networks with Ieee 802.15.4 Systems: Distributed Processing, Mac, and Connectivity*. Signals and communication technology. Springer Berlin Heidelberg, 2011. ISBN: 9783642174902 (cit. a p. 2).

[4] Angelo P Castellani et al. "Architecture and protocols for the internet of things: A case study". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. IEEE. 2010, pp. 678–683 (cit. a p. 2).

[5] Tim Berners-Lee, Roy Fielding e Larry Masinter. *Uniform resource identifiers (URI): generic syntax*. 2005 (cit. a p. 2).

[6] Klaus Hartke. "Observing Resources in CoAP". In: (2012) (cit. a p. 3).

[7] Zach Shelby, Klaus Hartke e Carsten Bormann. "Constrained application protocol (coap)". In: (2013) (cit. a p. 5).

[8] Zach Shelby. "Constrained RESTful Environments (CoRE) Link Format". In: (2012) (cit. a p. 6).

[9] *iDigi - Any app, Anything, Anywhere*. 2013. URL: http://www.idigi.com/ (cit. a p. 8).

[10] *Cosm - connect your world*. 2013. URL: http://cosm.com/ (cit. a p. 8).

[11] *Nibits - the open source internet of things on a distributed cloud*. 2013. URL: http://www.nimbits.com/ (cit. a p. 8).

[12] *Paraimpu - The Web of Things is more than Things in the Web*. 2013. URL: http://paraimpu.crs4.it/ (cit. a p. 9).

[13] *Open.sen.se - feel. act. make sense*. 2013. URL: http://open.sen.se.com/ (cit. a p. 9).

[14] Filippo De Stefani et al. “Renvdb, a restful database for pervasive environmental wireless sensor networks”. In: *Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International Conference on*. IEEE. 2010, pp. 206–212 (cit. a p. 9).

[15] Enrico Costanzi. “Managing Constrained Devices Into The Cloud: A Restful Web Service”. 2013 (cit. alle pp. 13, 19, 40).

[16] Craig Walls. *Spring in Action, Third Edition*. Manning Publications Company, 2011. ISBN: 9781935182351 (cit. a p. 17).

[17] Paolo Atzeni et al. *Basi di dati, seconda edizione*. McGraw-Hill, 1999 (cit. a p. 21).

[18] *SugarSync - Your Cloud*. 2013. URL: http://sugarsync.com/ (cit. alle pp. 40, 47).

[19] C. Bauer e G. King. *Java Persistence With Hibernate*. Manning Publications Company, 2009. ISBN: 1-932394-88-5 (cit. a p. 41).