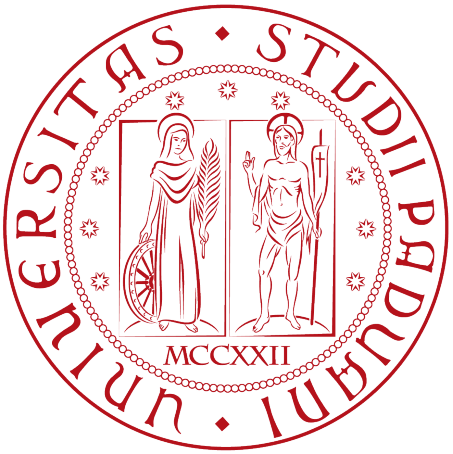
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Un prototipo di sistema di Home Automation basato su microservizi

Tesi di laurea triennale

Relatore

Prof. Tullio Vardanega

Laureando

Nicola Dal Maso

Anno Accademico 2017-2018

Struttura del documento

Il presente documento è articolato in quattro capitoli:

Nel primo capitolo presento i temi su cui ho svolto lo stage, elencando i rischi che ho valutato per le scelte intraprese.

Nel secondo capitolo descrivo con maggior precisione il progetto di stage, elencandone gli obiettivi curricolari, formativi, tecnici e di prodotto.

Nel terzo capitolo approfondisco il lavoro svolto durante lo svolgimento dello stage, esaminando le attività di analisi, progettazione, codifica e test.

Nel quarto capitolo fornisco una valutazione degli obiettivi raggiunti, motivando la presenza di eventuali obiettivi non raggiunti; inoltre esamino le conoscenze acquisite e i rischi descritti nel primo capitolo.

Convenzioni tipografiche

Riguardo la stesura del testo, relativamente al documento ho adottato le seguenti convenzioni tipografiche:

- ho definito un glossario, presente alla fine del presente documento, contenente gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati nel corso del documento;
- ho utilizzato per la prima occorrenza dei termini riportati nel glossario la seguente nomenclatura: parola[g];
- ho evidenziato i termini in lingua straniera o facenti parti del gergo tecnico con il carattere *corsivo*.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 L'idea

In questa relazione descrivo lo svolgimento dello stage effettuato nel contesto del Corso di Laurea di Informatica dell'Università di Padova.

Ho intrapreso lo stage nella sua forma interna individuale, in cui con il proponente, Prof. Tullio Vardanega, ho redatto un piano di lavoro nel quale lo stage abbia una durata pianificata su 300 ore.

L'obiettivo principale dello stage consiste nello sviluppo e nella realizzazione di un prototipo per la gestione di dispositivi interconnessi (IoT) attraverso un'interfaccia *web*. Questo centro di controllo attraverso cui l'utente del sistema gestisce i dispositivi *smart* presenti nella propria rete domestica dovrebbe permettere operazioni quali:

- avvio/spegnimento di un dispositivo;
- monitoraggio dei dispositivi collegati;
- collegamento all'eventuale interfaccia proprietaria del dispositivo (es. supporto tecnico).

Ho sviluppato e realizzato il progetto di stage con l'idea di unificare in un unico centro di controllo tutti gli eventuali dispositivi connessi alla rete domestica dell'utente, permettendo tuttavia allo stesso di accedere all'interfaccia proprietaria di ciascun dispositivo. L'obiettivo di una tale *dashboard* non è quindi confinare l'utente in un unico ecosistema domotico, bensì quello di facilitare la consultazione delle informazioni più frequentemente richieste dall'utente provenienti da più ecosistemi distinti.
Grazie alla natura prototipale del prodotto sviluppato, ho inoltre potuto sperimentare l'approccio architetturale a microservizi, al fine di garantire scalabilità all'applicazione.

1.2 IoT: definizione e caratteristiche generali

Internet Of Things è un paradigma tecnologico diffusosi nell'ultimo decennio. Questa locuzione fa riferimento a un insieme di oggetti, di varia natura e utilizzo, che interagiscono tra loro e che permettono all'utente di interagire con essi.

Un dispositivo *smart* è un dispositivo elettronico, generalmente connesso ad altri dispositivi, che può svolgere le proprie funzioni in maniera autonoma. Un esempio lampante di dispositivi *smart* sono gli smartphone, prodotti che hanno arricchito di funzionalità avanzate, interagendo con l'utente in modi precedentemente non possibili, i predecessori telefoni (*phone*).[1]

Studenti e professori del Dipartimento di Informatica dell'Università di Carnegie Mellon presero in considerazione l'idea di una rete di dispositivi *smart* quando modificarono un distributore automatico di bibite del Dipartimento per accedere al suo inventario di bibite. Quel distributore automatico divenne il primo apparecchio collegato ad Internet.[2]

Nel corso degli anni '90 il mondo accademico e il mondo dell'industra legata alla produzione continuarono a sperimentare evolvendo il *concept* iniziale, arrivando alla conclusione che l'*Ubiquitous Computing* non si debba riferire solamente ai *computer*, ma debba espandersi agli oggetti di utilizzo quotidiano. Questa visione dovette scontrarsi con i limiti della microelettronica di allora: la produzione di semiconduttori non era ancora pronta a supportare la potenziale domanda e i costi per sostenere l'ampliamento degli impianti non erano facilmente assorbibili in breve tempo.[3]

Kevin Ashton coniò il termine *"Internet of Things"* nel 1999.[4] L'origine dell'espressione, riassumendo le parole dell'autore, deriva dal fatto che la maggior parte delle informazioni presenti su Internet sono state e sono inserite da utenti "umani", soggetti quindi a concentrazione, precisione e tempo limitate; se queste informazioni fossero invece inserite da macchine senza l'aiuto di un utente umano, la maggior qualità delle stesse garantirebbe:

- maggiore capacità di tracciamento delle risorse;
- minor spreco di risorse;
- minor costo per la gestione delle risorse.

Grazie agli avanzamenti nei processi di produzione dei semiconduttori, dovuti alla crescita dei mercati del consumo di massa, allo sviluppo di un numero sempre maggiore di tecnologie volte a migliorare l'efficienza e l'affidabilità dei circuiti integrati e alla sempre maggior diffusione di tecnologie per la trasmissione di informazioni senza fili, dai primi anni 2000 un numero sempre maggiore di aziende, provenienti dagli ambiti più disparati, ha sviluppato il concetto alla base dell'IoT, estendendolo a settori quali *home automation*, *manufacturing*, *smart agriculture*, etc..[5]

[1] *Smart device*. URL: https://en.wikipedia.org/wiki/Smart_device.

[2] *The Carnegie Mellon University Computer Science Department Coke Machine*. URL: https://www.cs.cmu.edu/~coke/history_long.txt.

[3] Mark Weiser. *The computer for the 21st century*. New York, NY, USA, 1999. URL: https://web.archive.org/web/20150311220327/http://web.media.mit.edu/~anjchang/ti01/weiser-sciam91-ubicomp.pdf.

[4] Kevin Ashton. *That 'Internet of Things' Thing*. 2009. URL: http://www.rfidjournal.com/articles/view?4986.

[5] Christian Floerkemeier Friedemann Mattern. «From the Internet of Computers to the Internet of Things». In: (2010). URL: http://www.vs.inf.ethz.ch/publ/papers/Internet-of-things.pdf.

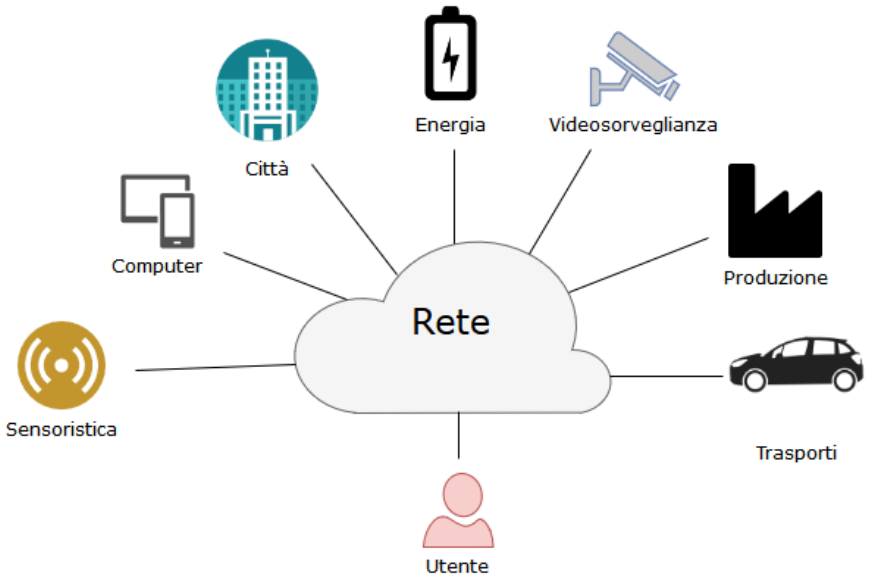


Figura 1.1: Rappresentazione figurata del concetto di *"internet of things"*.

Posso sintetizzare il concetto di *Internet of Things* come quello di una rete di dispositivi interconnessi, individuabili in modo univoco e che possono comunicare informazioni. I dispositivi presenti in una rete possono comunicare con due tipologie di attori diverse:

- se la comunicazione avviene con altri dispositivi si parla di comunicazione M2M (*Machine to Machine*, ovvero comunicazione tra macchine);
- se la comunicazione avviene interagendo con il mondo reale si parla di comunicazione M2H (*Machine to Human*, ovvero comunicazione tra macchina e utente).

Il termine "Things" nel contesto IoT si riferisce a una varietà di dispositivi come ad esempio: videocamere di sorveglianza, automobili a guida autonoma e assistita oppure piccoli e grandi elettrodomestici casalinghi. Questi dispositivi, soprannominati anche *smart object*, raccolgono informazioni utili in base agli attori con cui comunicano:

- se i dispositivi comunicano in modo M2M, le informazioni supportano tecnologie esistenti, integrandosi nel flusso di informazioni esistente;
- se i dispositivi comunicano in modo M2H, le informazioni aiutano le persone che interagiscono con essi.

L'interesse verso il tema IoT è cresciuto esponenzialmente sia nel mercato consumer che in quello enterprise e secondo Forbes ([6]) diventerà nel prossimo quinquennio uno dei settori dell'ITC più redditizi. É interessante osservare anche la popolarità dei termini di ricerca correlati all'IoT: i dati sono stati ottenuti interrogando il servizio https://trends.google.com/trends, il quale consente di effettuare analisi sulla popolarità delle stringhe di ricerca immesse nel motore di ricerca di Google. Ciascun grafico a linea (*line chart* in inglese) presenta nelle ordinate il grado di popolarità della

[6] Louis Columbus. *2017 Roundup Of Internet Of Things Forecasts*. 10 Dic. 2017. URL: https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts.

query di ricerca, valutato da 0 (popolarità minima) a 100 (popolarità massima), e nelle ascisse l'arco temporale in analisi.

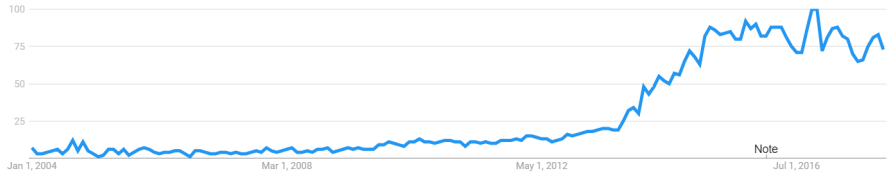


Figura 1.2: Andamento dell'interesse per la stringa di ricerca *"internet of things"*.
URL: https://trends.google.com/trends/explore?date=2004-01-01%202017-12-31&q=internet%20of%20things

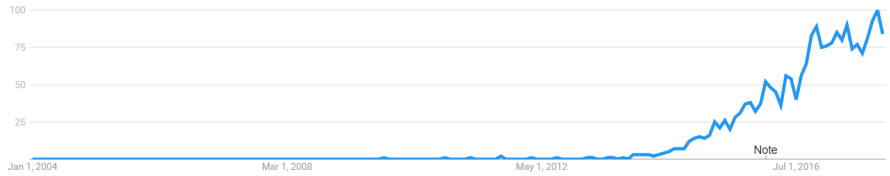


Figura 1.3: Andamento dell'interesse per la stringa di ricerca *"iot devices"*.
URL: https://trends.google.com/trends/explore?date=2004-01-01%202017-12-31&q=iot%20devices

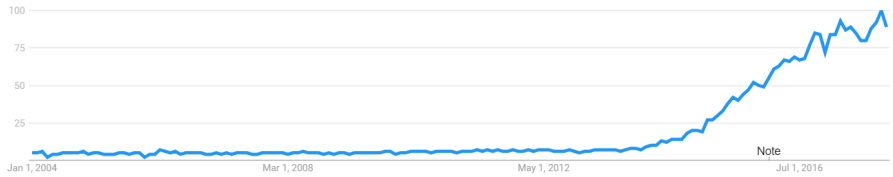


Figura 1.4: Andamento dell'interesse per la stringa di ricerca *"iot"*.
URL: https://trends.google.com/trends/explore?date=2004-01-01%202017-12-31&q=iot

Sintetizzando le previsioni di Forbes con l'andamento dei termini di ricerca legati all'IoT, visibili alle figure 1.2, 1.3 e 1.4, posso evidenziare che l'interesse verso l'argomento IoT stia generalmente aumentando o nel caso peggiore rimanga stabile con l'interesse degli anni precedenti.

1.3 Architettura a microservizi: definizione e caratteristiche

L'espressione **Architettura a microservizi** è sempre più comune tra gli sviluppatori di applicazioni *enterprise* per descrivere un metodo di progettazione delle applicazioni

come insiemi di servizi eseguibili indipendentemente, che comunicano tra loro grazie a meccanismi di comunicazione "leggeri" (solitamente attraverso Application Program Interface (API)[g] HTTP). Nella concezione originale in cui l'architettura a microservizi è nata, ogni servizio doveva essere progettato per eseguire in un processo indipendente dagli altri; con la nascita e la diffusione dei *container* questo paradigma sta cambiando, associando sempre più l'esecuzione dei microservizi in altrettanti *container*. La *containerization* (containerizzazione) è un metodo di virtualizzazione posto al livello del sistema operativo per la distribuzione ed esecuzione di applicazioni all'interno di *container*.[7] Un *container* è un unità *software* standardizzata, distribuibile in un unico pacchetto composto da:

- l'applicazione da eseguire;
- l'ambiente d'esecuzione configurato correttamente per l'applicazione da eseguire, che a sua volta specifica:
  - le dipendenze dell'applicazione;
  - i file di configurazione dell'applicazione.

Una delle tecnologie di containerizzazione che si è più diffusa è Docker (`https://www.docker.com/what-docker`), sviluppata dall'omonima azienda; Docker ha reso disponibile la propria tecnologia di containerizzazione su molteplici piattaforme, sia locali (*computer* con i sistemi operativi *Windows*, *macOS* e i sistemi operativi basati su *Linux*) sia in *cloud* (con ad es. servizi come *Amazon Web Services* e *Microsoft Azure*). Per implementare il concetto di *container*, la piattaforma di Docker installa un insieme di servizi che comunicano con il Kernel[g] del sistema operativo su cui è in esecuzione (*host*) e con i quali gli utenti interagiscono per creare e gestire *container*.[8] Ho illustrato l'architettura sopra citata in figura 1.5.

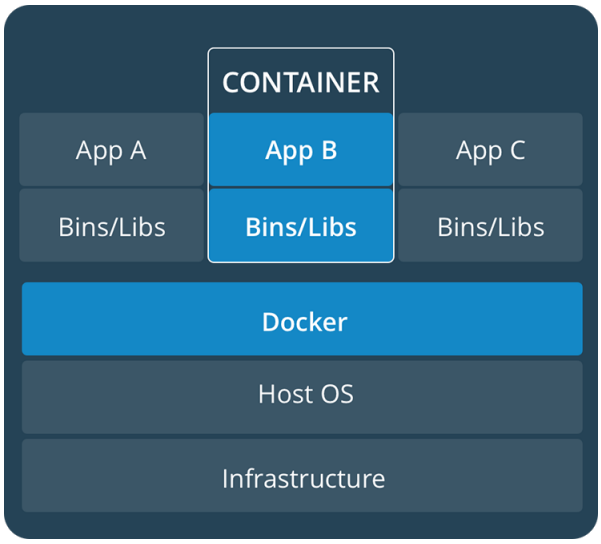


Figura 1.5: Architettura sviluppata da Docker per la sua piattaforma di containerizzazione. *What is a container*. URL: `https://www.docker.com/what-container`

[7] *Containerization*. URL: `https://en.wikipedia.org/wiki/Operating-system-level_virtualization`.

[8] *What is a container*. URL: `https://www.docker.com/what-container`.

Caratteristiche delle architetture monolitiche Un'applicazione monolitica è progettata e costruita per essere una singola unità in esecuzione. L'applicazione sviluppata con architettura monolitica è responsabile della visualizzazione delle informazioni in un'interfaccia utente (pagine web o *software* nativi), del reperimento delle informazioni da una sorgente di dati (solitamente un *database*) e dell'esecuzione delle logiche di business della stessa.[9]

Nelle applicazioni monolitiche la modularità del sistema si ottiene sfruttando i costrutti fondamentali dell'orientamento ad oggetti presente nei linguaggi di programmazione:

- funzioni;
- classi;
- *namespace* o *package*.

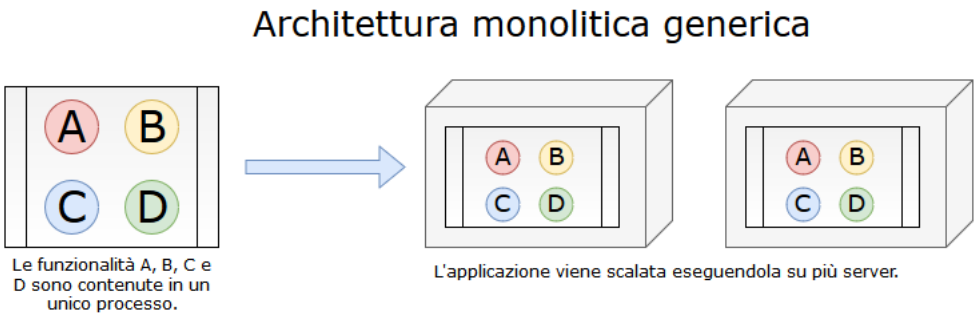


Figura 1.6: Caratteristiche di una generica architettura monolitica.
Martin Fowler. *Microservices, a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html

Per aumentare la disponibilità delle applicazioni monolitiche si usa replicare istanze dell'applicazione in molteplici server, bilanciando il traffico verso le applicazioni per mezzo di un load balancer[g]. Tra i difetti delle applicazioni monolitiche posso evidenziare:

- modifiche a una piccola parte all'applicazione richiedono la ricompilazione e la ridistribuzione dell'applicazione;
- all'accrescere della complessità dell'applicazione aumenta anche la difficoltà nel mantenere le modifiche isolate ai moduli di competenza;
- scalare l'applicazione richiede l'esecuzione di istanze multiple della stessa applicazione, ignorando di fatto eventuali requisiti di efficienza (solitamente alcune componenti del sistema non richiedono un aumento di throughput[g]).

Caratteristiche delle architetture a microservizi Per lo stile architetturale a microservizi non esistono definizioni formali, tuttavia gli informatici più esperti in materia, tra i quali annovero Martin Fowler, hanno dedotto le caratteristiche che hanno accomunato i progetti diventati nel tempo esempi di best-practice. Non tutte le architetture a microservizi hanno tutte le caratteristiche elencate in seguito, ma ci si aspetta che la maggior parte delle architetture esibisca quante più caratteristiche possibili.[10]

[9] Rod Stephens. *Beginning Software Engineering.* John Wiley & Sons, 2015.
[10] Martin Fowler. *Microservices, a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html.

L'aspetto cruciale delle architetture a microservizi verte sulla definizione di componente: la definizione comunemente accettata di componente è quella di "unità di software che è indipendentemente aggiornabile e sostituibile in un sistema". Le architetture a microservizi usano i servizi per realizzare tale definizione di componente. A titolo di confronto con gli approcci di sviluppo tradizionali introduco la nozione di libreria. Le librerie sono componenti insiti in un'applicazione tanto da risiedere nello stesso spazio di memoria dell'applicazione e che per essere invocate richiedono una chiamata di funzione in memoria. I servizi sono componenti che vivono nel sistema come processi separati, sfruttando vari tipi di comunicazione interprocesso: richieste web, chiamate di funzione remote (RPC).[11]

Il vantaggio principale dei servizi rispetto alle librerie consiste nel fatto che i servizi sono rilasciabili indipendentemente dal sistema. Data la natura dell'architettura a microservizi, modifiche a un singolo servizio comportano il rilascio di una nuova versione solamente per quel servizio e non dell'intera applicazione. Una buona architettura a microservizi quindi mira a progettare e implementare servizi che circoscrivano chiaramente il loro scopo.

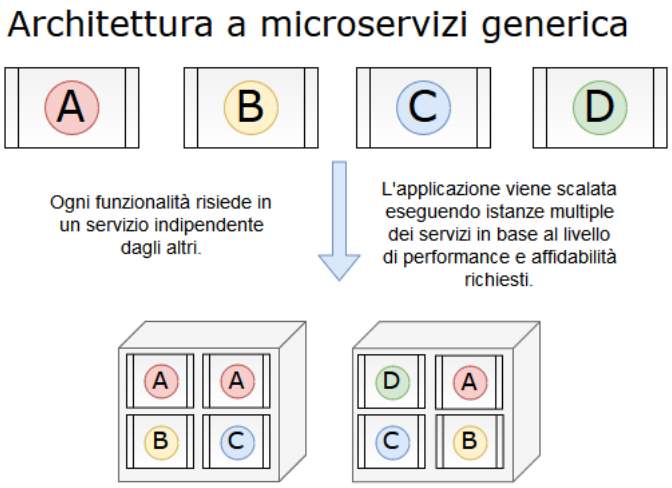


Figura 1.7: Caratteristiche di una generica architettura a microservizi.
Martin Fowler. *Microservices, a definition of this new architectural term.* 2014.
URL: https://martinfowler.com/articles/microservices.html

L'uso di servizi come componenti consente inoltre di rendere esplicita l'interfaccia dei componenti. Spesso solamente la documentazione e la disciplina prevengono usi impropri di una componente da parte di uno sviluppatore esterno, rischiando di causare un alto accoppiamento tra componenti. I servizi facilitano il rispetto delle interfacce pubblicate attraverso l'uso di meccanismi di chiamate remote esplicite. Il difetto che Fowler attribuisce all'uso di servizi come componenti risiede nell'utilizzo di chiamate remote per la comunicazione tra servizi: esse richiedono più risorse rispetto alle chiamate di funzione intraprocesso e quindi è necessario progettare le API di ciascun servizio rivolgendo maggiore attenzione all'aspetto prestazionale delle stesse.[12]

[11] Ibid.
[12] Ibid.

Nella sua trattazione, Fowler inoltre riscontra ed evidenzia le differenze dal punto di vista della suddivisione delle persone impegnate nello sviluppo dell'applicazione. Solitamente applicazioni complesse sviluppate seguendo l'architettura monolitica sono divise in *team* con competenze isolate:

- *team* esperto in UI;
- *team* specializzato in DB Management;
- uno o più *team* specializzati a realizzare la logica di business.

Ho illustrato graficamente questo ambiente lavorativo in figura 1.8. L'origine di una tale suddivisione risale alla Legge di Conway, enunciata nel 1967 dallo sviluppatore Melvin Conway, la quale afferma:

> "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."

La legge di Conway ragiona sul fatto che un sistema *software* complesso per funzionare richieda lo sforzo congiunto di più attori; dal momento che questi attori devono comunicare tra loro, la complessità insita nella comunicazione tra gli attori si riflette nelle componenti del sistema.[13]

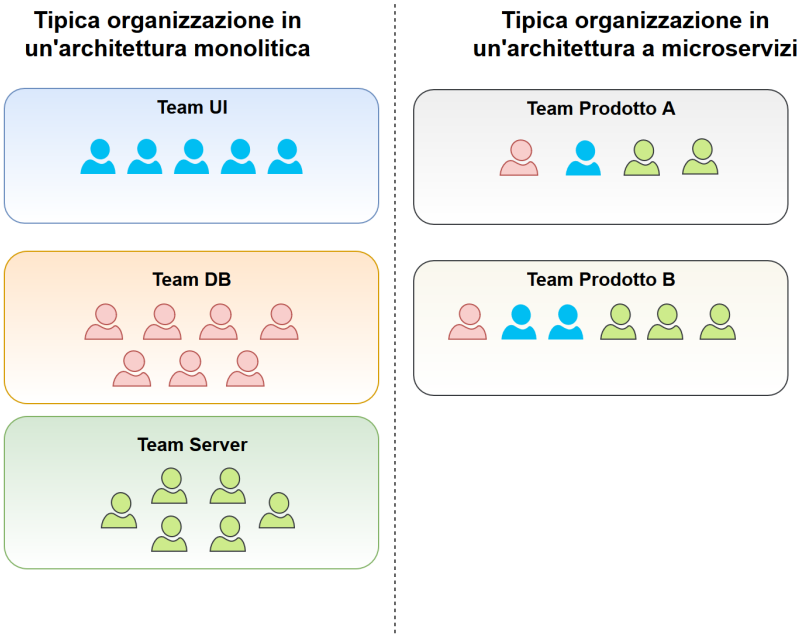


Figura 1.8: Illustrazione che mostra la differente organizzazione aziendale in un ambiente di sviluppo monolitico e in un ambiente di sviluppo a microservizi.
Martin Fowler. *Microservices, a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html

Quando le persone sono così isolate, anche una semplice modifica può richiedere l'intervento di altre persone in *team* diversi. La maggiore richiesta di pianificazione e organizzazione tra gruppi di sviluppo diversi causa un peggioramento dell'efficienza del processo di sviluppo.

[13] Melvin Conway. *Conway's Law.* URL: http://www.melconway.com/Home/Conways_Law.html.

L'approccio orientato ai microservizi con la suddivisione dell'applicazione invece pone l'accento sulle capacità di business: ogni *team* inerente un particolare settore di business si occupa dell'intero prodotto per quel settore (sviluppando interamente UI, DB, ecc.). I *team* in questo approccio sono multidisciplinari e gli scambi con altri settori riflettono le effettive dipendenze tra un settore e un altro all'interno dell'azienda.[14]

Un esempio di quest'approccio alla suddivisione lo si ritrova in Amazon, dove vige il motto "you build, you run it" ("tu lo costruisci, tu lo esegui"). In Amazon ogni *team* ha completa responsabilità del prodotto anche in ambiente di produzione, mettendo in comunicazione diretta sviluppatori e utenti del prodotto per le attività di supporto e manutenzione.[15]

Le comunicazioni tra servizi sono orchestrate usando semplici protocolli basati su REST. REST, acronimo di REpresentational State Transfer, è un tipo di architettura *software* per lo sviluppo di applicazioni distribuite, introdotta nel 2000 nella tesi di dottorato di Roy Fielding. Le architetture basate su REST prevedono che la scalabilità delle applicazioni sia conseguenza di pochi principi di progettazione:

- separazione tra *client* e *server*: i ruoli delle due componenti sono ben distinti utilizzando un insieme di interfacce comuni per la comunicazione, permettendo quindi uno sviluppo indipendente di queste componenti (se l'interfaccia comune non viene alterata);
- *stateless*: la comunicazione *client-server* è vincolata in modo che nessuna informazione sullo stato del *client* venga memorizzata dal *server*;
- *cacheable*: ogni *client* deve poter memorizzare le risposte inviate dal *server* per minimizzare le comunicazioni *client-server*. Ogni risposta deve comunicare al *client* implicitamente o esplicitamente se essa è memorizzabile;
- *layered system*: un *client* non deve poter discernere un *server* di basso livello da uno intermedio, dedicato a migliorare le prestazioni o introdurre politiche di sicurezza;
- *uniform interface*: la comunicazione tra *client* e *server* deve avvenire con un'interfaccia omogenea, disaccoppiando le due componenti ma degradando potenzialmente l'efficienza, dal momento che le informazioni vengono trasferite in una forma standardizzata invece di una più affine alla loro struttura.

[16][17]

L'esperienza di Fowler mostra come siano due le tipologie di protocolli più usati nelle architetture a microservizi:

- protocolli basati su richieste/risposte HTTP secondo API ben dettagliate;
- protocolli basati sullo scambio di messaggi in un canale di comunicazione snello. I servizi producono e consumano i messaggi che circolano nel canale di comunicazione, secondo regole di accesso definite.

Quando un'applicazione è suddivisa in molteplici componenti sorgono naturalmente dubbi sulla gestione delle informazioni che ciascuna componente gestisce. Solitamente nelle architetture monolitiche gli analisti astraggono i domini dell'applicazione scegliendo una fra le tecniche di modellazione disponibili e applicandola a tutti i domini; i modelli prodotti sono poi veicolati su singoli *storage* di dati (ad es. unico *database*). L'architettura a microservizi invece propone di concepire i modelli in autonomia per

[14] Fowler, *Microservices, a definition of this new architectural term*.

[15] *A Conversation with Werner Vogels - Learning from the Amazon technology platform*. 2006. URL: https://queue.acm.org/detail.cfm?id=1142065.

[16] *REpresentational State Transfer*. URL: https://en.wikipedia.org/wiki/Representational_state_transfer.

[17] Roy Fielding. «Representational State Transfer (REST)». in: (). URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

ogni singolo servizio, utilizzando le tecniche ritenute più appropriate. Questa decentralizzazione dell'astrazione dei modelli si riflette anche sulla possibilità di decentralizzare le decisioni relative a quale *storage* dei dati utilizzare per ciascun servizio. Nell'architettura a microservizi si preferisce che ogni servizio gestisca il proprio *database* in base ai requisiti che il servizio deve soddisfare: il database di un servizio potrebbe essere un'istanza di una stessa piattaforma tecnologica, una piattaforma specifica e ottimizzata per il caso d'uso del servizio oppure potrebbe non essere utilizzato (servizi puramente funzionali). Questo approccio alla gestione della persistenza è chiamato *Polyglot Persistence* ed è utilizzabile anche in architetture monolitiche, malgrado appaia con maggior frequenza in architetture a microservizi.[18]

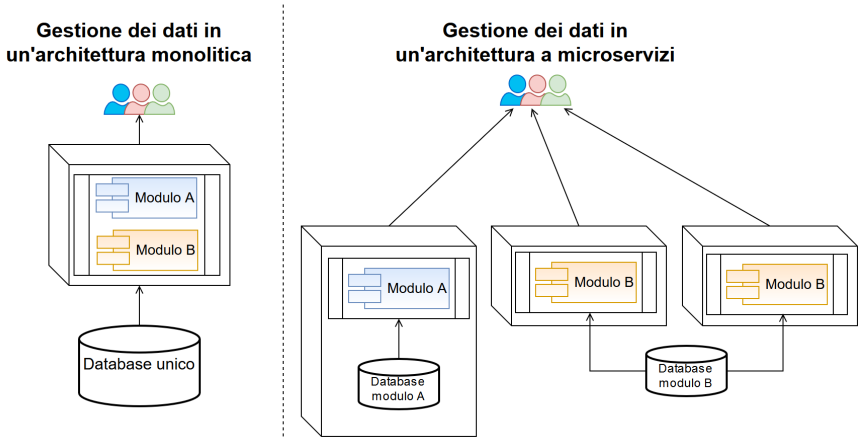


Figura 1.9: Illustrazione che mostra la differente gestione dell'architettura di persistenza dei dati tra prodotti software con architettura monolitica e con architettura a microservizi.
Martin Fowler. *Microservices, a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html

Le decisioni di *storage* decentralizzate implicano una maggior attenzione verso gli aggiornamenti dei dati. Fowler rileva che l'approccio comune agli aggiornamenti in un'architettura monolitica è quello di usare le transazioni per garantire la consistenza dei dati prima e dopo ciascun aggiornamento. L'utilizzo di transazioni è un grave limite per l'architettura a microservizi, in quanto le transazioni impongono un ordine temporale che potrebbe non essere rispettato, causando inconsistenze dei dati salvati. É per questo che le architetture a microservizi enfatizzano l'utilizzo di comunicazioni non vincolanti (*transactionless*): eventuali inconsistenze vengono segnalate e risolte grazie a operazioni correttive.[19]

Una conseguenza nell'utilizzo dei servizi come componenti è che le applicazioni devono prevedere e tollerare malfunzionamenti nei servizi. L'utilizzatore dei servizi deve quindi rispondere ai malfunzionamenti nel modo più elegante possibile. É naturale quindi attribuire l'aumento di complessità della gestione dei malfunzionamenti tra i difetti delle architetture a microservizi. Dal momento che i servizi possono malfungere in ogni momento, è fondamentale riuscire a:

- monitorare il servizio,

[18] Fowler, *Microservices, a definition of this new architectural term.*
[19] Ibid.

- segnalare il malfunzionamento e
- ripristinare automaticamente il servizio

nel più breve tempo possibile. Conseguentemente, ogni servizio deve essere progettato focalizzando l'attenzione sulle attività di monitoring, individuando le metriche rilevanti (ad es. *throughput*, latenza, ecc.).

Uno dei temi cruciali che ho riscontrato nella trattazione di Fowler consiste nel chiedersi quante responsabilità debba avere ciascun servizio: secondo Fowler la caratteristica fondamentale da osservare è la nozione di sostituzione e aggiornamento indipendenti. Un buon segnale lo si ritrova quando ad ogni modifica di un servizio, questa modifica non richiede adattamenti in altri servizi (a meno di modifiche alle funzionalità offerte). Se due o più servizi vengono aggiornati spesso insieme probabilmente essi dovrebbero essere uniti.[20]

[20] Ibid.

1.4 Valutazione dei rischi considerati nello stage

1.4.1 Valutazione dei rischi di uno stage interno

Lo stage formativo viene previsto dal CdL triennale di Informatica in due modalità:

- stage aziendali, riferiti anche come stage esterni, per i quali il proponente del progetto di stage è un'azienda;
- stage non aziendali, riferiti anche come stage interni individuali, per i quali il proponente del progetto di stage è un docente dell'Ateneo di Padova.

Sebbene lo stage aziendale sia la modalità preferita per svolgere l'attività, mi sono imbattuto in due ostacoli che mi hanno portato ad avviare uno stage interno.
Il mio stato di studente lavoratore causa il primo ostacolo: l'azienda per cui sono assunto non poteva offrire stage riguardanti il settore IoT. Nel momento in cui ho iniziato a cercare proposte di stage e mi sono quindi informato sulle modalità con cui avrei potuto assentarmi da lavoro, l'ufficio per le risorse umane dell'azienda per cui sono assunto mi ha comunicato che avrei avuto opzioni limitate, dipendenti dal motivo dell'assenza.
Se l'assenza avesse comportato l'inizio di attività lavorative per altre aziende (concorrenti oppure non concorrenti) sarei stato costretto a scegliere tra due opzioni:

- il licenziamento dall'azienda per cui sono assunto;
- l'avvio della procedura di aspettativa[g] del lavoro come prevista dalla legge 53 recante data 8 marzo 2000.[21].

Quindi fin dall'inizio la ricerca di un progetto di stage aziendale è stata fortemente messa in discussione dalle menzionate opzioni.
Malgrado il primo ostacolo, ho proseguito la ricerca dello stage aziendale al fine di ponderare se gli aspetti negativi legati al congedo lavorativo potessero essere in qualche modo bilanciati da eventuali esperienze formative.
Dato il periodo di inizio dello stage (ottobre/novembre 2017), molti degli stage aziendali riguardanti il settore IoT proposti erano già stati svolti da altri studenti del corso. Le poche proposte di stage aziendale legate al settore IoT rimanenti erano interessate alla integrazione con prodotti già esistenti: malgrado l'opportunità offerta da queste aziende fosse interessante, ho riflettuto a lungo sul fatto che l'attività formativa non fosse adeguatamente supportata.
Nei progetti proposti infatti le aziende proponenti hanno enfatizzato l'utilizzo degli strumenti interni da loro sviluppati, da una parte per motivarmi ad essere assunto al termine dello stage, dall'altra per effettivamente semplificarmi il lavoro nel processo di sviluppo.

L'aspetto formativo è stato l'ambito più discusso per effettuare la scelta: malgrado la presenza di esperti nel settore mi interessasse, per l'opportunità di approfondire l'ambito d'uso reale dei dispositivi *smart*, la scelta di legarmi a un singolo prodotto, non particolarmente diffuso e utilizzato, mi ha spinto ad iniziare a valutare il percorso di stage interno.

Ho riassunto i rischi considerati per lo svolgimento di uno stage interno individuale nella tabella 1.1.

[21] *Disposizioni per il sostegno della maternità e della paternità, per il diritto alla cura e alla formazione e per il coordinamento dei tempi delle città*. 2000. URL: http://www.camera.it/parlam/leggi/000531.htm.

Tabella 1.1: Tabella di analisi dei rischi correlati allo svolgimento di uno stage interno

Rischio	**Probabilità di accadimento**	**Gravità del danno potenziale**
Il lavoro svolto individualmente, senza possibilità di essere guidato da esperti nel settore, potrebbe risultare ininfluente. La presenza di una persona con più esperienza in un determinato tema è utile in quanto può focalizzare l'attenzione su problematiche reali, le quali potrebbero non essere correttamente valutate nel momento in cui si dispone di scarsa esperienza in materia.	Molto probabile	Grave
Il lavoro svolto individualmente, senza possibilità di essere guidato da esperti nel settore, potrebbe procedere più lentamente, causando il non raggiungimento degli obiettivi preposti.	Molto probabile	Media

1.4.2 Valutazione dei rischi del tema IoT

Nuove tecnologie contengono sempre una determinata quantità di rischi: mentre la maggior parte degli sviluppatori trovano utilizzi per i dispositivi IoT, altri cercano modi per usarli per scopi meno nobili.
I dispositivi IoT stanno sempre più diffondendosi in molti aspetti su cui basiamo la società moderna:

- trasporti;
- comunicazione;
- settore energetico.

Attacchi informatici contro questi dispositivi possono portare al caos: dalla distruzione di proprietà alla messa in discussione della propria sicurezza, con l'accezione che il termine *safety* possiede nella lingua inglese. A peggiorare la situazione, gli acquirenti di questi dispositivi pretendono che essi continuino a funzionare per una quantità di tempo superiore a quella che il consumismo tecnologico ha abituato.
La quantità di protocolli sviluppati per l'IoT porta ad aumentare la complessità insita in questi dispositivi. Una complessità maggiore implica un maggior costo per le società per aggiornare i prodotti rilasciati, portando quindi i produttori ad abbandonare i dispositivi rilasciati da più tempo, ignorando l'insorgere di nuove vulnerabilità.

Date le suddette premesse, per il settore IoT ho posto numerose osservazioni relative ai rischi collegati allo sviluppo di prodotti software e hardware. I rischi considerati sono riassunti nella tabella 1.2.

Tabella 1.2: Tabella di analisi dei rischi correlati al tema IoT

Rischio	**Probabilità di accadimento**	**Gravità del danno potenziale**
Lo sviluppo di prodotti legati all'IoT espone l'utente degli stessi a possibili rischi di cybersicurezza: il furto di dati ma soprattutto la perdita di controllo dell'utente sui propri dispositivi sono scenari possibili e con conseguenze disastrose per lo sviluppatore di tale prodotto.	Probabile	Molto grave
La mole di dati raccolta dai dispositivi inseriti in un contesto IoT potrebbe essere tale da richiedere investimenti consistenti per la loro elaborazione e per mantenere elevata la loro confidenzialità. Inoltre la diversità dei dispositivi presenti in una rete aumenta la complessità del trattamento delle informazioni.	Probabile	Media
Dal momento che l'IoT è un ambito emergente nel contesto ITC ho osservato la nascita di una multitudine di protocolli per la raccolta e la trasmissione delle informazioni, nessuno dei quali è stato indicato o si è imposto come standard globale.	Molto probabile	Media

1.4.3 Valutazione dei rischi dell'architettura a microservizi

L'architettura a microservizi permette di sviluppare un'applicazione complessa partendo da piccole e relativamente isolate componenti; in questo modo i cambiamenti effettuati sono facilmente verificabili. La frammentazione dell'architettura del prodotto rende tuttavia più complesse le attività di test funzionali, perché per eseguire un tipico caso d'uso di una funzionalità completa sono richiesti molteplici servizi, correttamente configurati per comunicare tra loro ed eseguire simultaneamente.

Lo sviluppo di servizi indipendenti rende inoltre difficoltosa l'attività di integrazione delle modifiche: nel momento in cui molti *team* di sviluppo lavorano ciascuno nel proprio servizio non è possibile integrare queste modifiche senza una attenta pianificazione, che coinvolga la comunicazione dei cambiamenti effettuati.
Nel mio caso questo non si applica, lavorando individualmente, tuttavia mi sono comunque richieste le attività di allineamento delle funzionalità tra i servizi che necessitano di comunicare tra loro.

Anche in questo caso, la relativa novità dell'argomento causa una generale mancanza di documentazione pratica, lasciando lo spazio ad esempi semplici, che non riflettono applicazioni d'uso reale, oppure documentazione teorica, che non si spinge ad analizzare problematiche reali. La tabella 1.3 riassume e sintetizza i rischi analizzati in precedenza.

Tabella 1.3: Tabella di analisi dei rischi correlati all'utilizzo dell'architettura a microservizi

Rischio	**Probabilità di accadimento**	**Gravità del danno potenziale**
Spostamento di alcune problematiche di progettazione da un livello di modulo a un livello di architettura del sistema.	Probabile	Media
Le performance dell'applicazione sviluppata potrebbero non essere sufficienti passando da un'architettura monolitica a una a microservizi.	Scarsamente probabile	Grave
Difficoltà nel reperimento delle informazioni, data la relativa novità dell'argomento. Assume maggior significato nel momento in cui l'esperienza con un tale paradigma risulti scarsa o nulla.	Molto probabile	Grave

1.5 Organizzazione dello Stage

Ho pianificato le attività, in termini di quantità di ore di lavoro, secondo la tabella 1.4 e le ho inserite in un Piano di Lavoro. Dal momento che ho già avuto modo di trattare il lato tecnologico del progetto nel corso del percorso di studi, ho allocato 40 ore di formazione negli argomenti meno conosciuti e li ho suddivisi in due parti:

- una prima parte in cui ho studiato le caratteristiche comuni delle architetture a microservizi e i Design Pattern nati per facilitare l'implementazione di un sistema con questo tipo di architettura;
- una seconda parte pratica di applicazione dei princìpi teorici in Node.js e il ripasso della caratteristiche tecniche del *framework* React.

Durante quest'insieme di attività di formazione, per mitigare i rischi elencati in tabella 1.1, ho focalizzato la mia attività di studio alla ricerca di esempi sviluppati da aziende con esperienza in materia per risolvere i problemi che hanno dovuto affrontare. Il secondo macroblocco di attività riguarda l'analisi, la progettazione e l'implementazione dei servizi di comunicazione con i dispositivi. A queste attività ho allocato 120 ore, in quanto per mitigare le probabilità di incorrere nei rischi elencati in tabella 1.2 ho assegnato risorse per l'attività di ricerca del protocollo di comunicazione da implementare. Durante le attività di analisi inoltre mi sono preparato per contrastare i rischi più probabili che ho menzionato in tabella 1.3, analizzando i requisiti funzionali del sistema e mettendoli in relazione con le attività di ricerca effettuate durante le attività di formazione. Ho incluso nelle attività di progettazione di questo blocco anche la progettazione e realizzazione dei dispositivi "virtualizzati", l'implementazione dei quali mi ha permesso di effettuare i primi test dei servizi di comunicazione. Il terzo macroblocco di attività riguarda l'analisi dell'interazione che un *client* può avere con i servizi di comunicazione, permettendomi di arrivare alla progettazione e realizzazione del servizio che compone le funzionalità da offrire alla *dashboard* e la *dashboard* stessa. L'ultimo blocco di attività riguarda la revisione del prototipo implementato e la sua possibile pubblicazione sulla piattaforma di Heroku[22].

[22] *Heroku è una piattaforma in* cloud *per l'esecuzione di applicazioni che supporta diversi linguaggi di programmazione.* URL: https://www.heroku.com/.

Tabella 1.4: Tabella che illustra la pianificazione delle attività dello stage

<table>
<tr><th colspan="2">Durata in ore</th><th>Descrizione dell'attività</th></tr>
<tr><td colspan="2">40</td><td>Formazione
Architettura microservizi
Node.JS orientato ai microservizi
React</td></tr>
<tr><td rowspan="3">120</td><td></td><td rowspan="3">Analisi, sviluppo e implementazione servizi di comunicazione con i dispositivi IoT
Analisi dei protocolli open source esistenti per i diversi dispositivi IoT
Stima implementazione eventuali nuovi protocolli
Progettazione dei servizi di comunicazione con i dispositivi
Progettazione dei test dei servizi di comunicazione
Realizzazione dei test e dei servizi di comunicazione in Node.js</td></tr>
<tr><td>40</td></tr>
<tr><td>80</td></tr>
<tr><td rowspan="3">120</td><td></td><td rowspan="3">Analisi, sviluppo e implementazione servizio di presentazione delle informazioni agli utenti
Analisi interazione utente con la dashboard
Progettazione del servizio di presentazione informazioni
Progettazione dei test del servizio di presentazione
Realizzazione dei test e del servizio di presentazione in React, HTML5 e CSS3.</td></tr>
<tr><td>40</td></tr>
<tr><td>80</td></tr>
<tr><td colspan="2">20</td><td>Review dei servizi, deploy dei servizi su Heroku.</td></tr>
<tr><td colspan="2">Totale ore</td><td>300</td></tr>
</table>

1.6 *Milestone*

In questa sezione presento le milestone[g] previste per il progetto su base settimanale, associando a ciascuna *milestone* i prodotti che devono essere sviluppati entro la corrispondente scadenza.

- Prima settimana: Completamento delle attività di autoformazione con produzione di una breve relazione riguardante la stessa;
- Seconda e terza settimana: Analisi dei protocolli di comunicazione esistenti, primo ciclo di progettazione e implementazione del servizio di comunicazione, mirato all'implementazione dei dispositivi *virtualizzati*;
- Quarta e quinta settimana: Revisione analisi sui protocolli, secondo ciclo di progettazione e implementazione del servizio di comunicazione, mirato all'implementazione dei dispositivi fisici (Raspberry Pi);
- Sesta e settima settimana: Analisi dell'interazione utente con la *dashboard*, progettazione e implementazione del servizio di presentazione;
- Ottava settimana: Revisione dei servizi, stesura del Manuale d'Uso e deploy (opzionale) di un ambiente di simulazione della *dashboard* su Heroku.

Capitolo 2

Progetto di stage

2.1 Descrizione generale

Lo stage interno è una forma di stage individuale in cui uno studente, in concerto con un docente, redige un piano delle attività da svolgere nell'intervallo di tempo specificato. Nel presente progetto di stage ho iniziato le attività di stage in data 6/11/2017 e le ho terminate in data 2/1/2018, con un monte ore totale di circa 312 ore.
In questo periodo di svolgimento dello stage ho definito, con l'assistenza del Prof. Tullio Vardanega, gli obiettivi dello stage dai seguenti punti di vista:

- obiettivi di prodotto: questi obiettivi coincidono con le caratteristiche e le funzionalità importanti per gli utenti del prodotto;
- obiettivi formativi: questi obiettivi comprendono le conoscenze che mi aspetto di acquisire e le abilità mi aspetto di apprendere durante lo svolgimento dello stage.
- obiettivi tecnici: questi obiettivi comprendono le tecniche e le tecnologie specifiche che mi aspetto di dover padroneggiare al fine di completare il progetto di stage.

L'ambito su cui ho focalizzato l'attenzione dello stage include dispositivi per l'automazione domestica dedicati alla gestione dell'illuminazione e alla gestione termica dell'abitazione.
I dispositivi che ho considerato per l'illuminazione domestica consistono solamente in varianti di un solo dispositivo, la lampada *smart*.
I dispositivi considerati invece per la gestione termica dell'abitazione sono sostanzialmente due:

- sensori, i quali devono inviare informazioni relative alla temperatura di un determinato ambiente;
- termostati, i quali devono interfacciarsi con i sensori da una parte, per ricevere e sintetizzare la distribuzione termica dell'abitazione, e con l'impianto di riscaldamento dall'altra, per applicare le variazioni di temperatura richieste dall'utente.

2.2 Obiettivi di prodotto

L'obiettivo principale del prodotto è quello di fornire un'interfaccia unificata per la gestione dei dispositivi connessi, consentendo all'utente l'accesso all'interfaccia proprietaria di ciascun dispositivo.

Come riporta il titolo della presente relazione, per il progetto di stage ho preferito concentrarmi su funzionalità per certi aspetti innovative restando nell'ambito di sviluppo di un prototipo; come tale, il prototipo non è una soluzione pronta alla distribuzione sul mercato (*production-ready*), quanto un modo per sperimentare con l'automazione domestica introducendo funzionalità non diffuse nei prodotti presenti sul mercato.

Le funzionalità chiave che il prodotto sviluppato vuole offrire all'utente sono riportate in tabella 2.1, che illustra l'identificativo assegnato all'obiettivo, una descrizione dell'obiettivo e la sua importanza, valutata in una scala di importanza crescente da 1 a 5, in cui 1 indica l'importanza minima e 5 l'importanza massima.

Tabella 2.1: Tabella delle funzionalità offerte dal prototipo all'utente

Id obiettivo	**Descrizione obiettivo**	**Importanza funzionalità**
OP1	Visualizzare lo stato generale del sistema.	4
OP2	Visualizzare quali dispositivi sono collegati al sistema.	5
OP3	Visualizzare le informazioni trasmesse dai dispositivi collegati al sistema.	5
OP4	Implementare sistemi di autenticazione dell'utente.	2
OP5	Visualizzare e gestire le preferenze dell'utente.	3

L'obiettivo "OP1" consente all'utente di verificare l'operatività del sistema, mostrando gli eventuali malfunzionamenti che causano un disservizio alla *dashboard*; per questo credo sia una funzionalità relativamente importante per l'utente, che potrebbe voler conoscere se un disservizio della *dashboard* è legato ad un malfunzionamento del sistema sottostante e non ad un errore dell'interfaccia.

Gli obiettivi "OP2" e "OP3" costituiscono funzionalità fondamentali per l'utilizzo della *dashboard* perché permettono all'utente di conoscere quanti e quali dispositivi sono correttamente riconosciuti dal sistema e per ciascun dispositivo le informazioni messe a disposizione dallo stesso.

Malgrado l'obiettivo "OP4" sia fondamentale per un prodotto distribuibile al pubblico, ho assegnato una importanza medio-bassa a questo obiettivo: data la natura di prototipo del prodotto e considerato che il tema sicurezza richiede una elevata formazione per la sua corretta implementazione, ho preferito dare priorità agli obiettivi riguardanti alle funzionalità che possono dare un valore di innovazione aggiunto rispetto alle soluzioni presenti sul mercato.

L'obiettivo "OP5" aggiunge una componente di personalizzazione alla *dashboard* che potrebbe essere utile all'utente: l'esempio a cui ho fatto riferimento riguarda la scelta delle unità di misura predefinite con cui il sistema visualizza le informazioni.

2.3 Obiettivi formativi

Per analizzare al meglio gli obiettivi formativi del presente stage, ho deciso di suddividere gli obiettivi formativi legati all'ambito IoT da quelli relativi alle architetture a microservizi.
Dal punto di vista del tema IoT, ho concentrato l'attenzione su due aspetti essenziali:

1. l'analisi e la definizione delle caratteristiche e delle funzionalità di cui sono dotati i dispositivi IoT esistenti;
2. la ricerca e l'implementazione di un protocollo di comunicazione le cui qualità siano adeguate al contesto di utilizzo.

L'aspetto citato in 1 risulta fondamentale per apprendere, in aggiunta alle abilità di analisi imparate durante il corso di studi, abilità specifiche nella comprensione delle funzionalità e caratteristiche richieste dal mercato presente dei dispositivi IoT.
L'aspetto citato in 2 mi permette di acquisire conoscenze specifiche relative a protocolli di comunicazione non approfonditi durante il corso di Reti e Sicurezza del percorso di studi del CdL di Informatica. Il mio obiettivo in questo frangente è analizzare e studiare i protocolli di comunicazione esistenti, utilizzabili liberamente e le cui specifiche siano accessibili pubblicamente e gratuitamente.
Gli obiettivi precedentemente citati corrispondono agli obiettivi "OF1" e "OF2" definiti nella tabella 2.2.
Dal punto di vista delle architetture a microservizi, data la mia totale inesperienza riguardo le architetture *software* orientate ai servizi ho posto come obiettivi formativi l'acquisizione dei concetti alla base di queste architetture, analizzandone:

1. i princìpi generali che definiscono questo insieme di architetture;
2. le tecnologie che consentono di sviluppare sistemi *software* con le caratteristiche richieste da queste architetture;
3. pregi e difetti delle architetture orientate ai servizi, dando particolare risalto ai pregi e difetti delle architetture a microservizi.

Mentre le voci 1 e 3 corrispondono all'obiettivo "OF3" riportato in tabella 2.2, la voce 2 è una generalizzazione dell'obiettivo "OF4" (riportato nella stessa tabella citata precedentemente). Tra i princìpi delle architetture orientate ai servizi e ai microservizi, uno degli aspetti su cui mi sono concentrato con maggior attenzione è l'analisi della corretta dimensione di un servizio tale per cui esso possa essere definito "*micro*". Un altro concetto importante su cui ho dovuto informarmi con attenzione consiste nella scelta delle tecnologie di persistenza dei dati per ciascun servizio, specialmente in relazione alla già citata *Polyglot persistence* (riferimento 1.3).

Tabella 2.2: Tabella degli obiettivi formativi del progetto

Id obiettivo	Descrizione obiettivo
OF1	Apprendere abilità elementari per la comprensione delle funzionalità richieste dal mercato IoT, specialmente nel campo della automazione domestica.
OF2	Acquisire conoscenze adeguate alla scelta e implementazione di un protocollo di comunicazione adeguato al campo di utilizzo del progetto.
OF3	Comprendere il concetto di architettura a microservizi, con i pregi e i difetti caratteristici di una tale architettura.
OF4	Acquisire le nozioni legate alla containerizzazione di un sistema *software* in un contesto architetturale basato su microservizi.

2.4 Obiettivi tecnici

Sin dall'inizio delle attività di stage è stata mia intenzione distribuire i prodotti di queste attività in modo che chiunque potesse consultarli liberamente e pubblicamente: per offrire questa possibilità il primo obiettivo tecnico del progetto consiste nell'adozione di una licenza di distribuzione permissiva, che permetta:

- la visualizzazione,
- l'utilizzazione e
- la modifica

del codice sorgente e della documentazione associata senza vincoli legali. Questo obiettivo corrisponde all'obiettivo "OT1" indicato in tabella 2.3.
Collegato all'obiettivo precedente, il secondo obiettivo tecnico consiste nella pubblicazione delle istruzioni per facilitare l'esecuzione del prototipo in una macchina di sviluppo locale. Dal momento che il prototipo è testabile da un pubblico potenzialmente vasto, questo secondo obiettivo tecnico assicura che il sistema sviluppato possa essere eseguito in maniera ripetibile, semplificando la ricerca e la segnalazione di malfunzionamenti e permettendo a chiunque di valutare le idee sviluppate nel prototipo. Questo obiettivo corrisponde all'obiettivo "OT2" indicato in tabella 2.3.
L'obiettivo "OT3" indicato in tabella 2.3 si riferisce alla possibilità di far funzionare il prototipo in un ambiente realistico: in questo ambiente realistico vi sono dispositivi, con cui l'utente può interagire, che trasmettono le informazioni raccolte a dispositivi in grado di elaborare queste informazioni e metterle a disposizione degli altri dispositivi in maniera strutturata. Dal momento che nella rete questi dispositivi devono poter identificarsi, con l'obiettivo "OT3" evidenzio le caratteristiche di modularità e configurabilità che il prototipo deve possedere.
In maniera complementare a quanto appena detto, il prototipo deve poter essere eseguito in un unico dispositivo che sia in grado di simulare l'esecuzione nell'ambiente realistico precedentemente citato. Questo obiettivo, indicato come "OT4" in tabella 2.3, è fortemente collegato agli obiettivi "OT1" e "OT2", perché non è assicurato che gli utenti che desiderano provare il prototipo posseggano un insieme di dispositivi che possa eseguire tutte le componenti che formano la *dashboard*.
Da un punto di vista tecnico, gli obiettivi "OT3" e "OT4" vincolano la scelta del protocollo di comunicazione da implementare, perché è necessario che tale protocollo sia applicabile sia in ambito di esecuzione in un ambiente reale, sia nell'ambito di simulazione, nel quale non ci sono comunicazioni all'esterno della macchina nella quale esegue il prototipo.
L'obiettivo "OT5" in tabella 2.3 si riferisce alla possibilità di conoscere quali componenti del sistema siano in esecuzione e cambiare lo stato delle componenti al fine di aumentare o diminuire la disponibilità di una componente, fino alla completa disattivazione della stessa. Evidenzio la correlazione tra questo obiettivo ("OT5") con l'obiettivo "OF4": l'acquisizione corretta delle conoscenze delle tecnologie di containerizzazione dovrebbe semplificare il soddisfacimento dell'obiettivo "OT5", dato il contesto attinente con cui si sono sviluppate le tecnologie di containerizzazione.

Gli obiettivi tecnici "OT6" e "OT7", indicati in tabella 2.3, impostano dei vincoli tecnologici per l'implementazione del prototipo.
L'obiettivo "OT6" è strettamente correlato con l'obiettivo "OT3", il quale richiede l'utilizzo di tecnologie multipiattaforma per la corretta esecuzione del prototipo. Ho scelto di vincolare lo sviluppo del prototipo adottando *Node.js* come *framework* per l'implementazione della parte *backend* per due motivi:

1. dal momento che gli argomenti trattati nello stage mi sono completamente nuovi, ho voluto appoggiarmi dal punto di vista tecnico a una tecnologia già utilizzata per l'implementazione del progetto formativo del corso di Ingegneria del Software per abbassare la quantità di argomenti nuovi trattati;
2. *Node.js* utilizza JavaScript[g] come linguaggio di programmazione, rendendo lo sviluppo delle applicazioni più veloce, grazie ad una sintassi semplice da imparare.

L'obiettivo "OT7" è il risultato di un altro vincolo tecnologico che ho imposto con lo scopo di semplificare lo sviluppo dell'interfaccia grafica dell'applicazione *web* grazie all'esperienza già acquisita a riguardo con il progetto formativo del corso di Ingegneria del Software.

Tabella 2.3: Tabella degli obiettivi tecnici del progetto

Id obiettivo	Descrizione obiettivo
OT1	Rilascio del codice sorgente del prototipo e della documentazione associata nei termini di una licenza open source[g].
OT2	La documentazione associata al progetto deve includere le istruzioni necessarie all'esecuzione del prototipo.
OT3	Il prototipo deve essere eseguibile su dispositivi presenti in una rete, previa corretta configurazione.
OT4	Il prototipo deve essere eseguibile su un dispositivo di test, che simuli l'esecuzione in un ambiente reale.
OT5	Il prototipo deve prevedere strumenti per gestire la scalabilità del sistema e per monitorarne lo stato.
OT6	Il prototipo deve essere implementato in Node.js (`https://nodejs.org/en/about/`) per il lato server.
OT7	L'interfaccia utente del prototipo deve essere implementata in React (`https://reactjs.org/`).

Capitolo 3

Svolgimento dello stage

3.1 Ambiente di sviluppo

In questa sezione descrivo le tecnologie e gli strumenti di sviluppo che ho utilizzato per lo sviluppo del progetto, includendo la motivazione per cui ho fatto la scelta. Nella tabella 3.1 presento il sommario delle tecnologie e degli strumenti che ho utilizzato per lo sviluppo del prototipo, indicandone:

- il nome;
- una breve descrizione;
- la versione utilizzata o l'intervallo di versioni utilizzate nel caso in cui abbia aggiornato le tecnologie o gli strumenti utilizzati ad una versione successiva;
- il grado di conoscenza pregressa in una scala crescente che parte da 0 (esperienza nulla) e che arriva a 5 (esperienza consolidata), abbreviato in "Esp.".

Tabella 3.1: Tabella con il sommario delle tecnologie e degli strumenti utilizzati

Tecnologia	Descrizione	Versioni	Esp.
Node.js	Node.js è un'ambiente d'esecuzione utilizzato per l'implementazione di applicazioni server in JavaScript.	v9.2.0, v9.2.1	4
React	React è una libreria per il linguaggio JavaScript il cui scopo è costruire interfacce grafiche.	v16.2.0	3
ECMAScript 2017	ECMAScript è un linguaggio di programmazione la cui implementazione standard più conosciuta è JavaScript.	06/2017	2
Jest	Jest è un *framework* per l'implementazione di test per codice JavaScript.	v21.2.1	3
ESLint	ESLint è uno strumento *open source* per l'analisi statica del codice JavaScript prodotto.	v4.12.0, v4.12.1	4
HTML5	HTML5 è un linguaggio di *markup* per la formattazione e impaginazione delle pagine *web*.	N.D.	4
CSS3	CSS3 è un linguaggio di formattazione delle pagine *web*.	N.D.	2
Docker Engine	Docker Engine è la combinazione dell'implementazione della tecnologia di containerizzazione con gli strumenti di gestione del ciclo di vita dei *container*.	v17.09.0, v17.12.0	4
Docker Compose	Docker Compose è lo strumento attraverso cui è possibile coordinare applicazioni eseguite su *container* multipli.	v1.18.0	1
Atom	Atom è un *editor* di testo sviluppato da GitHub[g].	v1.21.2, v1.23.1	4
VS Code	Visual Studio Code è un *editor* di testo sviluppato da Microsoft[g].	v1.18.1, v1.19.1	1

3.2 Analisi dei Requisiti

Ho iniziato l'analisi dei requisiti cercando di capire quali funzionalità dovesse offrire il prototipo; per affrontare al meglio la definizione di queste funzionalità, ho esplorato il mercato dei dispositivi IoT esistenti per individuare i limiti del dominio applicativo in cui questi dispositivi si inseriscono. Ho osservato che una delle caratteristiche desiderabili che il sistema deve offrire riguarda l'identificazione precisa di quali dispositivi sono collegati al sistema; per questo ho vincolato l'accesso dei dispositivi al sistema: per accedervi i dispositivi devono fornire informazioni relative a modello, revisione, produttore e anno di produzione del dispositivo, oltre a un seriale univoco.

Ho inoltre suddiviso i dispositivi in due tipi principali in base alle funzionalità che essi offrono:

- sensori;
- dispositivi che ho definito "attivi".

La funzionalità principale offerta dai sensori è l'invio periodico di informazioni legate a ciò che il sensore misura. L'invio di informazioni periodiche avviene in automatico, secondo i parametri impostati dal produttore del sensore.

Nella mia analisi delle funzionalità richieste dai dispositivi attivi, ho considerato le seguenti funzionalità:

- pubblicazione di una lista degli eventi gestiti dal dispositivo;
- generazione di risposte agli eventi esterni;
- invio di informazioni sullo stato energetico del dispositivo;
- spegnimento del dispositivo.

Il dispositivo espone la lista degli eventi gestiti; questa lista permette di conoscere le funzionalità *smart* del dispositivo. La funzionalità di risposta agli eventi gestiti è automatica, avviene a ogni evento occorso ed è gestita direttamente dal produttore del dispositivo. L'invio delle informazioni sullo stato energetico del dispositivo richiede che il produttore abbia dotato il dispositivo di unità di *power management* e perciò potrebbe non essere disponibile per tutti i dispositivi collegati.

Acquisita conoscenza del dominio applicativo in cui mi stavo muovendo, ho introdotto il concetto di "centro di controllo": nella mia analisi, il centro di controllo consiste in quell'insieme di dispositivi responsabili della coordinazione tra i vari dispositivi connessi alla rete. I dispositivi facenti parte del "centro di controllo" presentano le seguenti funzionalità:

- gestione dei dispositivi collegati al sistema;
- ricezione, elaborazione e memorizzazione delle informazioni utili provenienti dai dispositivi (anche per fini diagnostici);
- pubblicazione delle informazioni raccolte per i *client* che interrogano il centro di controllo.

Dal momento che i dati ricevuti dal centro di controllo potrebbero essere raccolti dai dispositivi in una forma grezza, ho concluso che il centro di controllo necessiti di capacità di elaborazione al fine di rendere i dati raccolti comprensibili anche agli umani.

Individuate le necessità dei dispositivi coinvolti, ho iniziato lo studio dei protocolli di comunicazione. MQTT[g] è un protocollo di messaggistica leggero basato sul *Design Pattern Publish/Subscribe*. É un protocollo nato per l'utilizzo con sensori a basso consumo energetico ed è stato progettato tra la fine degli anni '90 e l'inizio degli anni 2000 per ambienti in cui l'affidabilità della rete non era garantita.[1] I dispositivi che ho considerato comunicano con il sistema utilizzando il protocollo MQTT (**??**), perché

[1] *MQTT* - Frequently Asked Questions. URL: http://mqtt.org/faq.

ho ritenuto questo protocollo il più adatto per il sistema. I motivi che mi hanno spinto a scegliere MQTT sono elencati di seguito:

- è un protocollo molto diffuso, caratteristica che mi ha facilitato molto nella ricerca di documentazione che dimostri il suo utilizzo;
- è un protocollo *data agnostic*, ossia che non pone vincoli sulla struttura dei dati scambiati nella rete;
- è un protocollo efficiente in quanto trasmette informazioni con un *overhead* minimo;
- è un protocollo che permette l'aggiunta e la rimozione di dispositivi dinamicamente, richiedendo intervento manuale minimo all'utente.

Nella mia analisi dell'interazione dell'utente con la *dashboard* ho specificato che l'accesso alle funzionalità del prodotto avvenga attraverso una interfaccia *web*, opportunamente progettata per essere reattiva (ottimizzata per *mobile*). Ho considerato anche la realizzazione di applicazioni native per i dispositivi *mobile*, tuttavia per la realizzazione del prototipo ho preferito concentrare le mie attività nell'implementazione di una soluzione multipiattaforma. Durante le attività di analisi dei requisiti ho scritto un documento di Analisi dei Requisiti, nel quale ho indicato gli obiettivi e le funzionalità del prototipo e dei dispositivi. Per raccogliere e specificare i requisiti del sistema e integrarli nel documento di Analisi dei Requisiti ho utilizzato i seguenti strumenti:

- casi d'uso, che permettono di valutare ogni requisito sulla base degli attori che interagiscono con il sistema, rispettando le aspettative che l'attore tiene dall'avvio dell'interazione al termine della stessa;
- diagrammi Unified Modeling Language (UML)[g] dei casi d'uso, che permettono di rappresentare i casi d'uso attraverso una definizione delle relazioni tra sistema, attori e funzionalità che il sistema offre agli attori.

Nella stesura della'Analisi dei Requisiti ho utilizzato *template* per velocizzare la definizione dei casi d'uso e dei requisiti. Nei *template* che ho utilizzato, ho catalogato i casi d'uso nella forma seguente:

$$UC[numero][caso]$$

UC	specifica che si sta parlando di un caso d'uso;
numero	è assoluto e rappresenta un riferimento univoco al caso d'uso in questione;
caso	individua eventuali diramazioni all'interno dello stesso caso d'uso.

La breve descrizione di ciascun caso d'uso presenta:

- gli attori del caso d'uso;
- lo scopo e la descrizione del caso d'uso.

Gli attori che ho considerato in sede di analisi consistono in:

- Utente: rappresenta l'utente che interagisce con la *dashboard*;
- Dispositivi: rappresentano l'insieme di apparati collegati al sistema che forniscono i dati per popolare la *dashboard*;
- Dispositivo: rappresenta uno dei dispositivi collegati al sistema;
- Interfaccia proprietaria del dispositivo: rappresenta l'interfaccia proprietaria progettata dal produttore di un generico dispositivo.

Lo scenario principale che ho immaginato, descritto nel caso d'uso UC0 e illustrato utilizzando il linguaggio UML in figura 3.1, prevede che l'utente abbia correttamente installato il prototipo per effettuare le prove in una macchina in locale e abbia eseguito

l'accesso alla *dashboard*. A questo punto ho previsto che l'utente possa utilizzare tre funzionalità della *dashboard*:

- visualizzare una pagina con il riepilogo dei dispositivi collegati;
- visualizzare una pagina che permetta di gestire i dispositivi collegati;
- visualizzare una pagina che indichi lo stato del sistema.

UC0: Scenario principale
Attori Principali: Utente, Dispositivi.

Precondizioni: L'utente ha correttamente installato il prototipo e ha aperto la *dashboard* in un *browser*.

Descrizione: La *dashboard* mostra lo stato del sistema ed evidenzia i dispositivi collegati.

Postcondizioni: La *dashboard* consente un'altra interazione con il sistema.

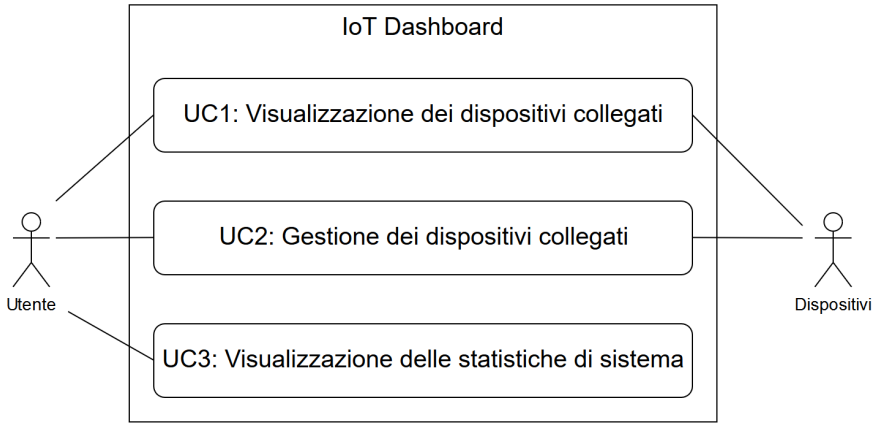


Figura 3.1: Use Case - UC0: Scenario principale

Durante le attività di analisi, ho utilizzato la tecnica del *top down* a partire dallo scenario principale menzionato precedentemente per raggiungere, attraverso iterazioni in cui ho raffinato i problemi in sottoproblemi più specifici, uno stato in cui mi è risultato chiaro quali requisiti dovesse soddisfare il prototipo. Per non allungare eccessivamente questa sezione, riporto l'analisi di uno dei casi d'uso sviluppati: la visualizzazione dei dispositivi collegati al sistema. La visualizzazione dei dispositivi collegati, caso d'uso UC1 descritto testualmente in UC1 e illustrato utilizzando il linguaggio UML in figura 3.2, mette in relazione l'utente con i dispositivi collegati al sistema. Lo scopo di questa interazione è far conoscere all'utente alcune informazioni elementari riguardanti i dispositivi collegati. In questo caso d'uso ho incluso un'insieme di funzionalità che non cambiano lo scopo del caso d'uso, bensì specificano eventuali interazioni che l'utente può avviare con il sistema. Una delle particolarità del caso d'uso UC1 riguarda il modo con cui ho gestito l'interazione dell'utente con un singolo dispositivo (UC1.3): se l'utente interagisce con un singolo dispositivo nell'ambito di questo caso d'uso, allora l'utente desidera conoscere le informazioni che riguardano quel dispositivo in maggiore dettaglio. Dal momento che le informazioni presentate all'utente in questo *step* riguardano i dispositivi collegati, allora significa che il dispositivo attore in UC1.3 è incluso nella lista dei dispositivi collegati. Da questo pensiero nasce la relazione di

inclusione tra l'attore "Dispositivo" e l'attore più esteso "Dispositivi".

UC1: Visualizzazione dei dispositivi collegati
Attori Principali: Utente, Dispositivi.

Precondizioni: L'utente ha scelto di visualizzare tutti i dispositivi collegati.

Descrizione: L'utente interroga la *dashboard* per conoscere lo stato dei dispositivi collegati.

Postcondizioni: L'utente conosce lo stato di tutti i dispositivi collegati al sistema.

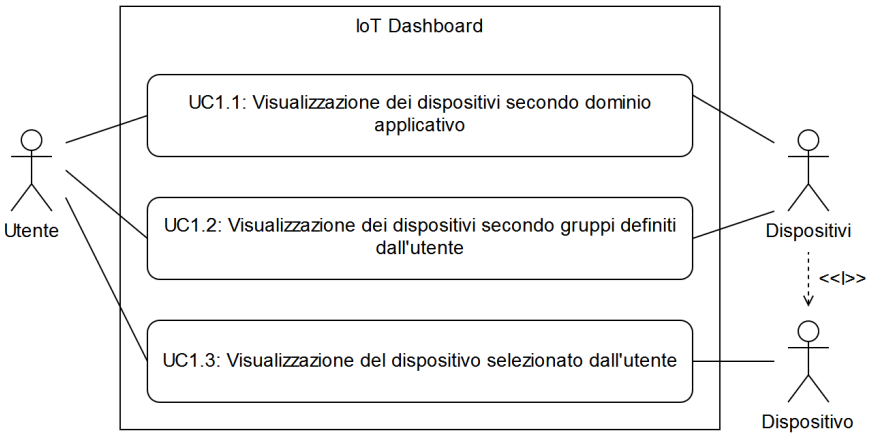


Figura 3.2: Use Case - UC1: Visualizzazione dei dispositivi collegati

Presento di seguito i requisiti emersi durante l'analisi dei casi d'uso. Per permetterne una consultazione agevole, ho deciso di inserire i requisiti in un insieme di tabelle dei requisiti suddivise in base alla categoria degli stessi. In tutte le tabelle cui presenti presento i requisiti indicando:

- Identificativo (secondo le regole indicate successivamente);
- Categoria di appartenenza fra:
  - Obbligatorio, per i requisiti irrinunciabili;
  - Desiderabile, per i requisiti non strettamente necessari ma che offrono un valore aggiunto riconoscibile;
  - Opzionale, per i requisiti relativamente utili o contrattabili in seguito.
- Descrizione esaustiva del requisito;

Ho scelto di identificare i requisiti seguendo la forma seguente:

$$R[Categoria][Tipo][numero]$$

dove

R	= specifica che si tratta di un requisito
$Categoria$	= indica se si tratta di un requisito tra quelli definiti in tabella 3.2
$Tipo$	= indica la tipologia del requisito tra quelli definiti in tabella 3.3
$Numero$	= è assoluto e rappresenta un riferimento univoco al requisito in questione

Per abbreviare le categorie utilizzate per identificare i requisiti ho associato nella tabella 3.2 un identificativo a ciascuna categoria.

Tabella 3.2: Tabella recante le categorie dei requisiti

Identificativo	Descrizione e origine
M	Obbligatorio (*mandatory*)
A	Desiderabile ((*advisable*)
O	Opzionale (*optional*)

Nella tabella 3.3 presento i tipi dei requisiti: il tipo indica l'origine del requisito e specifica se il requisito soddisfa una richiesta dell'utente del sistema oppure se esso deriva da una caratteristica del sistema.

Tabella 3.3: Tabella recante i tipi dei requisiti

Identificativo	Descrizione e origine
O	Di vincolo (*obligation*)
F	Funzionale (*functional*)
Q	Di qualità (*quality*)

In tabella 3.4 indico i requisiti di vincolo che ho individuato durante le attività di analisi dei requisiti. Con questi requisiti indico i vincoli, tecnologici e non, che il sistema deve soddisfare al fine di realizzare un sistema efficiente nella sua implementazione[2], nel funzionamento e nella manutenzione.

Tabella 3.4: Tabella dei requisiti di vincolo

Identificativo	Categoria	Descrizione
RMO1	Obbligatorio	Il sistema deve essere progettato secondo lo stile di progettazione a microservizi.
RAO2	Desiderabile	Il sistema può essere implementato utilizzando il linguaggio JavaScript secondo lo standard ECMAScript 2017.
RAO3	Desiderabile	Il sistema può essere implementato utilizzando il *framework* Node.js per il *backend* e React per il *frontend*.
RMO4	Obbligatorio	Il sistema deve utilizzare il protocollo MQTT.

In tabella 3.5 indico i requisiti funzionali che ho individuato durante le attività di analisi dei requisiti. Con questi requisiti indico le funzionalità che il sistema deve offrire all'utente dello stesso e le classifico in base all'importanza che esse possono avere per l'utente che interagisce con il sistema.

[2]Con la locuzione efficienza nell'implementazione mi riferisco alla quantità di risorse spese per soddisfare i requisiti funzionali del sistema (ad es. quantità di tempo investito per l'implementazione delle funzionalità *software*) rispetto al totale delle risorse allocate.

Tabella 3.5: Tabella dei requisiti funzionali

Identificativo	Categoria	Descrizione
RMF1	Obbligatorio	L'utente deve poter visualizzare tutti i dispositivi collegati al sistema.
RMF2	Obbligatorio	L'utente deve poter visualizzare i dispositivi collegati secondo dominio applicativo.
RAF3	Desiderabile	L'utente può visualizzare i dispositivi collegati secondo gruppi personalizzati.
RMF4	Obbligatorio	L'utente deve poter selezionare uno dei dispositivi collegati per visualizzarne le informazioni.
RAF5	Desiderabile	L'utente può creare un gruppo di dispositivi personalizzato.
RAF6	Desiderabile	L'utente può modificare uno dei gruppi personalizzati esistenti.
RAF7	Desiderabile	L'utente può rimuovere uno dei gruppi di dispositivi personalizzati esistenti.
RMF8	Obbligatorio	L'utente deve poter visualizzare le operazioni messe a disposizione dal dispositivo selezionato.
RMF9	Obbligatorio	L'utente deve poter selezionare una delle operazioni disponibili.
RMF10	Obbligatorio	L'utente deve poter visualizzare le statistiche di utilizzo del sistema sistema.

In tabella 3.6 indico i requisiti di qualità che ho individuato durante le attività di analisi dei requisiti. Con questi requisiti indico le caratteristiche che un sistema *software* deve soddisfare per essere definito *software* di qualità. Nel corso di Ingerneria del Software[3] ho studiato che i parametri con cui si può misurare la qualità di un *software* possono essere suddivisi in due categorie:

- i parametri esterni si riferiscono alla qualità del *software* percepita dagli utenti (correttezza, affidabilità, efficienza, ecc.);
- i parametri interni si riferiscono alla qualità del *software* percepita dagli autori del sistema e dai suoi manutentori (manutenibilità, riusabilità, leggibilità, ecc.).

Tabella 3.6: Tabella dei requisiti di qualità

Identificativo	Categoria	Descrizione
ROQ1	Opzionale	Il sistema deve essere testato, raggiungendo i seguenti obiettivi: • *statement coverage* $>$ 80 % • *branch coverage* $>$ 90 %

In tabella 3.7 riepilogo i requisiti individuati in sede di analisi, suddividendoli in base al tipo e alla loro categoria d'importanza.

[3] *Qualità del* software. URL: http://www.math.unipd.it/~tullio/IS-1/2017/Dispense/L13.pdf.

Tabella 3.7: Tabella di riepilogo dei requisiti

Tipo	Obbligatorio	Opzionale	Desiderabile
Funzionale	6	0	4
Qualitativo	0	1	0
Di vincolo	2	0	2
Totale	8	1	6

3.3 Progettazione

In questa sezione del documento definisco la progettazione dell'architettura ad alto livello del progetto di stage. La sezione include la descrizione dell'architettura del sistema e delle relative componenti *software* e i Design Pattern utilizzati per la progettazione.

L'architettura scelta per il sistema segue lo stile architetturale a microservizi con l'obiettivo di approfondire questo stile architetturale e implementarlo in uno scenario plausibile. Lo stile architetturale a microservizi descrive un metodo di progettazione delle applicazioni come insiemi di servizi eseguibili indipendentemente, che comunicano tra loro grazie a meccanismi di comunicazione leggeri. Nel mio caso, la comunicazione tra dispositivi avviene per mezzo del protocollo MQTT. Il protocollo MQTT si basa sul principio che ogni *client* pubblica messaggi, i quali hanno uno o più argomenti, nel gergo tecnico *topic*. Ogni *client* può registrarsi a determinati argomenti per ricevere tutti i messaggi che altri *client* pubblicano per quell'argomento. Molti *client* si connettono a un broker[g] che funziona da intermediario, ricevendo i messaggi pubblicati e inoltrandoli a tutti i *client* sottoscritti ai rispettivi argomenti. Gli argomenti in MQTT sono trattati gerarchicamente. Questo permette la creazione di argomenti e sottoargomenti, simili alla struttura ad albero di un *filesystem*. MQTT definisce 3 livelli di qualità della comunicazione in base a quanto *broker* e *client* si impegneranno a ricevere un messaggio. I *client* decidono il livello massimo di QoS[g] che riceveranno. La scala della QoS definisce i livelli 0, 1 e 2 con affidabilità crescente ma minori performance:

0 : *broker* e/o *client* invieranno il messaggio al massimo una volta senza richiesta di conferma. A questo livello i messaggi vengono persi se una delle parti si disconnette;

1 : *broker* e/o *client* invieranno il messaggio almeno una volta con la richiesta di conferma;

2 : *broker* e/o *client* invieranno il messaggio una sola volta effettuando una trasmissione in 4 step.

Per esempio, se un messaggio è pubblicato con QoS 2 e il *client* è sottoscritto all'argomento con QoS 0, il *client* riceverà quel messaggio con QoS 0 (niente richieste di conferma, ecc). Se un altro *client* è sottoscritto allo stesso argomento con QoS 2 allora riceverà il messaggio con QoS 2 (handshake[g] in 4 step). Alla connessione il *client* imposta un parametro logico che rappresenta una "sessione pulita" in base a come il *client* ritiene affidabile la connessione ('false' indica una connessione affidabile). Se il *client* si disconnette, in tutte le sottoscrizioni con QoS 1 o QoS 2 i messaggi verranno salvati e inviati alla prossima riconnessione del *client*.

Ho progettato le componenti del sistema seguendo il paradigma orientato agli oggetti: in questo modo ho potuto organizzare il codice in moduli riutilizzabili dalle diverse componenti del *software*, diminuendo la possibilità che implementi funzionalità comuni

in moduli diversi e con comportamento diverso che potrebbe causare malfunzionamenti nel sistema. Le attività di progettazione ad alto livello del sistema hanno richiesto da parte mia la redazione di un documento di Specifica Tecnica del sistema, nel quale ho indicato:

- le tecnologie e gli strumenti utilizzati;
- una panoramica dell'architettura del sistema;
- gli approfondimenti riguardanti le componenti definite all'interno dei servizi che compongono il sistema;
- i Design Pattern utilizzati all'interno delle componenti dei servizi del sistema.

Durante la progettazione delle componenti del prototipo ho utilizzato i diagrammi UML delle classi per rappresentare i tipi delle entità ed evidenziare le relazioni tra queste: in questo modo ho individuato le dipendenze tra le entità e sono quindi stato in grado di modularizzare alcune funzionalità in componenti isolate e riutilizzabili. Per snellire la procedura di inserimento della specifica delle classi all'interno della Specifica Tecnica ho utilizzato *template*.

Nell'immagine 3.3 illustro la panoramica delle componenti di cui è composta l'architettura utilizzando la notazione dei diagrammi delle componenti UML.

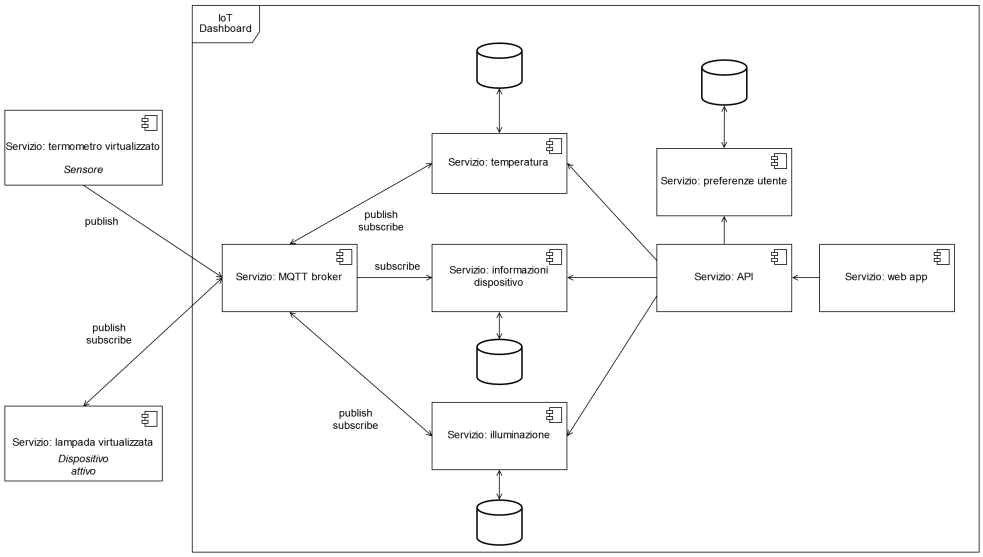


Figura 3.3: Panoramica dell'architettura ad alto livello progettata per il prototipo

Il *broker* MQTT è il servizio responsabile alla ricezione di tutti i messaggi, alla loro catalogazione e all'invio delle notifiche verso i *client* sottoscritti a ciascuna categoria. Il *broker* memorizza lo stato di tutti i *client* a lui connessi, inclusi i messaggi non ancora inviati o il cui invio è fallito.

Il termometro "virtualizzato" è il servizio responsabile della simulazione di un sensore che invii dati sulla temperatura dell'ambiente in cui si trova. Esso pubblica periodicamente la temperatura rilevata secondo l'argomento *temperature*, mentre invia secondo l'argomento *hw_info* i propri dati identificativi, quali produttore, modello, ecc.. Data la relativa importanza i dati vengono inviati con un QoS di livello 0 nella categoria *temperature*, mentre con QoS di livello 1 nella categoria *hw_info*. In questo modo al collegamento del dispositivo "virtualizzato" viene effettuato almeno un tentativo di

trasmissione delle informazioni relative alle specifiche del dispositivo. Anche se nel diagramma è disegnato individualmente, è possibile che ve ne siano molteplici. Nel diagramma 3.4 ho illustrato il diagramma delle classi UML a cui sono giunto durante le attività di progettazione ad alto livello. Vista la necessità di generare dati riguardanti la temperatura in maniera psuedocasuale, ho utilizzato il Design Pattern Abstract Factory[g] per permettere la creazione di funzioni per la generazione della temperatura. Queste funzioni per la generazione della temperatura sono incapsulate in una classe `TemperatureCurve`, il cui scopo è definire la funzione matematica che restituisce la temperatura in base alla data e ora specificata. L'implementazione di `TemperatureCurve` progettata per il termometro virtualizzato è `SineTemperatureCurve`, nella quale restituisco la temperatura sulla base di una temperatura di base specificata e applicando a tale temperatura una funzione sinusoidale. Ho applicato il Design Pattern Abstract Factory definendo una classe `TemperatureCurveFactory`, il cui unico scopo è creare oggetti che implementino `TemperatureCurve`. Nella tabella 3.8 ho elencato le classi sviluppate per soddisfare i requisiti di funzionamento del dispositivo.

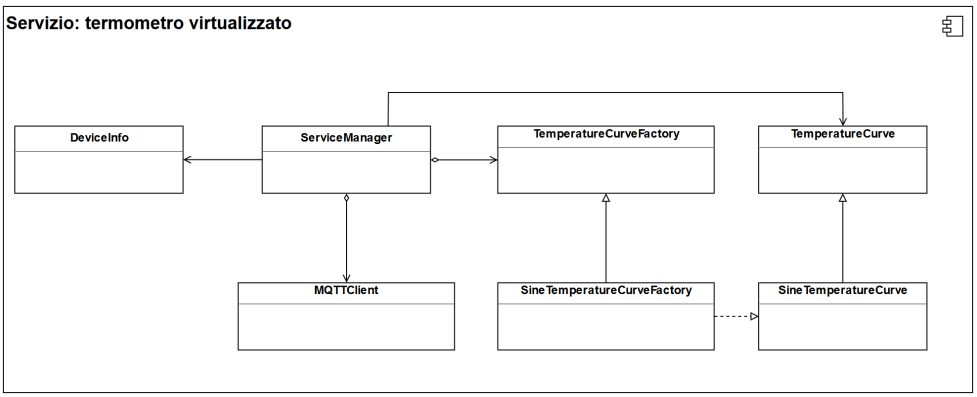


Figura 3.4: Architettura ad alto livello progettata per il termometro virtualizzato

Il servizio relativo alla temperatura si occupa di raccogliere tutti i dati provenienti dai sensori di temperatura, memorizzandoli e mettendoli a disposizione in un formato strutturato per gli altri servizi del sistema. Il servizio si sottoscrive alla categoria *temperature* e comunica con un QoS di livello 0, inoltre può pubblicare messaggi con la sottocategoria *temperature/active* per usufruire delle funzionalità aggiuntive presenti in dispositivi attivi legati alla temperatura, con un QoS di livello 1.

Con il servizio della lampada "virtualizzata" simulo la presenza nella rete di un dispositivo "attivo": una lampada in grado di comunicare il proprio assorbimento energetico e che è controllabile da remoto. La lista delle operazioni disponibili è la seguente:

- accensione della lampada (QoS di livello 2);
- spegnimento della lampada (QoS di livello 2);
- richiesta assorbimento energetico (QoS di livello 0);

Le operazioni di accensione e spegnimento della lampada necessitano di una affidabilità più elevata delle altre operazioni per evitare l'invio di richieste di accensione e spegnimento multiple. L'argomento a cui la lampada si sottoscrive è *light/active* e al primo collegamento la lampada invia i propri dati identificativi, pubblicandoli nella categoria *hw_ info*.

Tabella 3.8: Panoramica delle classi del servizio di simulazione del termometro

Classe	Funzionalità
`DeviceInfo`	Classe che indica le informazioni del dispositivo (produttore, modello, revisione, ecc.) pubblicate nel *topic hw_ info*.
`ServiceManager`	Classe responsabile dell'integrazione tra generazione dei dati di temperatura, gestione delle informazioni del dispositivo e invio delle informazioni tramite protocollo MQTT.
`MQTTClient`	Classe utile all'inizializzazione del *client* MQTT.
`TemperatureCurveFactory`	Classe Factory che espone la funzionalità di creazione della curva di temperatura, rappresentata dalla classe `TemperatureCurve`.
`SineTemperatureCurveFactory`	Implementazione della factory `TemperatureCurveFactory` per la creazione di oggetti `SineTemperatureCurve`.
`TemperatureCurve`	Classe che espone le funzionalità di creazione di una funzione, di aggiunta di rumore pseudocasuale nella funzione creata e di simulazione della temperatura data l'ora corrente.
`SineTemperatureCurve`	Classe che implementa `TemperatureCurve` definendo una funzione di simulazione sinusoidale parametrizzabile in ampiezza, frequenza e fase.

Il servizio relativo all'illuminazione si occupa di raccogliere e memorizzare tutti i dati pubblicati dai dispositivi nella categoria *light* e permette il controllo dei dispositivi sottoscritti alla categoria *light/active*. Questo servizio utilizza trasmissioni con tutti i livelli di QoS definiti nel protocollo MQTT: la comunicazione delle operazioni che l'utente esegue vengono trasmessi con il livello di affidabilità maggiore (Qos livello 2) per avere la garanzia che i dati trasmessi siano arrivati ai dispositivi atomicamente, la trasmissione delle specifiche tecniche del dispositivo vengono inviate con un livello di affidabilità intermedio (QoS livello 1) mentre le misurazioni vengono inviate nel modo più efficiente possibile (QoS livello 0).

Il servizio relativo alle informazioni dei dispositivi si occupa di raccogliere e memorizzare tutti i dati pubblicati secondo l'argomento *hw_ info*; utilizza esclusivamente un livello di QoS pari a 1 per aumentare l'affidabilità del sistema a fronte delle attività di identificazione dei dispositivi collegati.

Il servizio di gestione delle preferenze utente si occupa di salvare informazioni quali ad esempio gruppi personalizzati, unità di misura preferite, ecc. Il servizio non utilizza il protocollo MQTT in quanto non richiede la comunicazione con i dispositivi connessi alla rete, quindi viene utilizzato solamente dal servizio API.

Il servizio API svolge un ruolo da intermediario tra il servizio che fornisce l'applicazione *web* e i microservizi che raccolgono i dati dei dispositivi. Esso interroga i servizi "illuminazione", "temperatura" e "informazioni dispositivo" definiti dal sistema per fornire una interfaccia unificata ai dati. Per la progettazione del servizio API ho utlizzato uno dei Design Pattern specifici per le architetture a microservizi, ossia il *Gateway Pattern*. Ho progettato il servizio API in modo *stateless* grazie alla composizione delle API esposte dagli altri servizi: le chiamate effettuate al servizio API vengono

dirottate ai rispettivi servizi e opportunamente decorate con informazioni aggiuntive, utili durante le attività di *debug* del prototipo.

Il servizio "*web app*" comprende l'applicazione *web* per la consultazione della *dashboard* e include le componenti sviluppate in React per la costruzione dell'interfaccia grafica della *dashboard* e le classi necessarie alla sua pubblicazione sul *web*. Per conferire un'elevato grado di riutilizzabilità ai componenti creati ho utilizzato il Design Pattern Composite[g] (rappresentato in figura 3.5): nella progettazione dell'interfaccia ho individuato dapprima componenti elementari che estendono la classe `React.Component` (ad esempio componenti per visualizzare il testo specificato) e gradualmente ho assemblato componenti più complesse, sempre estensioni della classe `React.Component` (ad esempio unendo la componente di visualizzazione del testo con la componente di visualizzazione di una immagine). Attraverso questo procedimento *bottom up* ho concluso la progettazione delle componenti dell'interfaccia, giungendo alla componente `UIPage`, la quale è anch'essa una componente React che specifica quali componenti e con quale struttura visualizzare a schermo una determinata pagina. Ciascuna pagina dell'applicazione *web* è quindi un *collage* di componenti più o meno complesse che mi hanno permesso di riutilizzare i componenti elementari in molte pagine dell'applicazione.

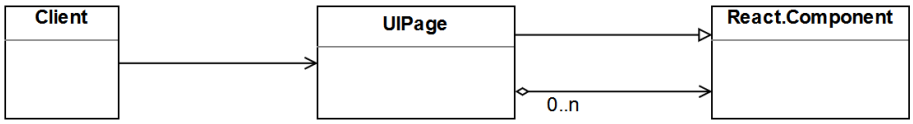


Figura 3.5: Rappresentazione semplificata del Design Pattern Composite

Nel diagramma 3.6 indico l'architettura ad alto livello progettata per il servizio di presentazione delle informazioni e nella tabella 3.9 descrivo le classi sviluppate.

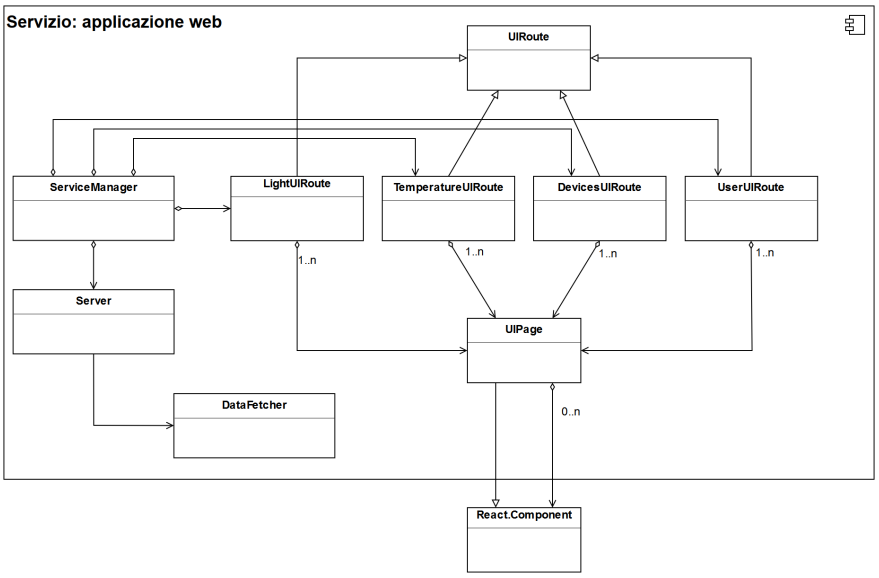


Figura 3.6: Architettura ad alto livello progettata per il servizio *web app*

Tabella 3.9: Panoramica delle classi del servizio API

Classe	Funzionalità
`ServiceManager`	Classe responsabile dell'integrazione tra istanza del server, pagine esposte e interfaccia di richeista dati.
`DataFetcher`	Modulo che si occupa di effettuare le richieste al servizio API secondo le definizioni fornite dal servizio.
`Server`	Classe responsabile del ciclo di vita del server Node.js. Effettua le richieste definite dalle istanze di `UIRoute` per ricevere i dati, utilizzando un'istanza di `DataFetcher`.
`UIRoute`	Interfaccia utilizzata per definire le richieste da effettuare per ricevere le informazioni che popolano le pagine della rotta.
`UserUIRoute`	Implementazione di `UIRoute` che definisce le richieste per ottenere o modificare le preferenze dell'utente ed espone le pagine di visualizzazione e modifica delle preferenze utente.
`DevicesUIRoute`	Implementazione di `UIRoute` che definisce le richieste per ottenere informazioni sui dispositivi collegati ed espone le pagine di visualizzazione di questi.
`TemperatureUIRoute`	Implementazione di `UIRoute` che definisce le richieste per ottenere dati legati alla temperatura, visualizzare ed eseguire operazioni con dispositivi attivi e ne permette la visualizzazione.
`LightUIRoute`	Implementazione di `UIRoute` che definisce le richieste per ottenere dati legati all'illuminazione, visualizzare ed eseguire operazioni con dispositivi attivi e ne permette la visualizzazione.
`UIPage`	Implementazione di un componente React (`React.Component`) che rappresenta una pagina. La pagina visualizzata può contenere più figli anch'essi componenti React.
`React.Component`	Classe che rappresenta una componente grafica nel *framework* React.

3.4 Implementazione e Verifica

Durante le attività di implementazione delle specifiche del sistema ho scritto, oltre al codice sorgente dei servizi, anche documenti in cui indico le istruzioni per eseguire le singole componenti del sistema e specifico le risorse esposte dalle stesse. I documenti in cui indico le risorse esposte dai servizi sono rivolti agli utenti più esperti che vogliono approfondire le funzionalità offerte dai servizi. Nelle mie previsioni questi utenti grazie ai documenti di specifica dovrebbero essere in grado di implementare le altre componenti del sistema, sostituendo le componenti da me implementate. Ho scritto questi documenti contestualmente all'implmentazione delle funzionalità nel codice sorgente, adeguandoli manualmente ad ogni modifica compiuta nel codice sorgente.
Ho deciso di scrivere i documenti in cui indico le istruzioni per eseguire il progetto sia in lingua italiana, sia in lingua inglese dal momento che il progetto è disponibile pubblicamente su GitHub. Nel farlo ho seguito alcuni consigli che ho tratto dalla lista degli articoli presenti al link `https://github.com/matiassingers/awesome-readme#articles`: in questo repository[g] l'autore ha raccolto collegamenti a:

- esempi di pagine informative dei rispettivi progetti caratterizzate da una buona struttura e da contenuti adeguati agli utenti a cui questi progetti sono indirizzati;
- articoli di autori che espongono la loro opinione riguardo a come dovrebbero essere scritti i documenti che gli utenti devono leggere in sede di esplorazione del progetto;
- strumenti che facilitano la scrittura dei documenti informativi.

Nelle istruzioni ho incluso i riferimenti per installare i *software* richiesti e ho fatto in modo che i procedimenti da seguire fossero i più brevi possibili (procedimenti con al massimo 5 istruzioni da seguire). Ho scritto queste istruzioni solamente quando mi sono accertato che le componenti fossero abbastanza stabili nel loro funzionamento da poter essere potenzialmente usate da altri utenti.
Ho scritto la documentazione del codice sorgente contestualmente al codice sorgente; la documentazione del codice sorgente consiste in commenti nei sorgenti JavaScript in cui spiego:

- lo scopo delle proprietà degli oggetti;
- il tipo delle proprietà degli oggetti: in questo caso ho preferito annotare il tipo delle proprietà degli oggetti nella documentazione in quanto JavaScript non è staticamente tipizzato e quindi la dichiarazione di variabili non consente di specificare un tipo con cui il compilatore e l'interprete validino il codice sorgente;
- gli algoritmi utilizzati per l'implementazione delle funzionalità.

Ho scritto i commenti nel codice sorgente, come da *best practice* per i progetti distribuiti pubblicamente, in lingua inglese.

Prima di iniziare le attività di codifica ho utilizzato Yeoman (`http://yeoman.io/`) per generare un *template* che ho utilizzato per implmentare tutti i servizi. Nel *template* ho specificato le dipendenze comuni a tutti i servizi e ho configurato la struttura dei progetti in modo che fosse ripetibile per tutti i servizi. Ogni servizio, grazie alla generazione per mezzo del *template*, ha al suo interno gli strumenti di analisi statica e dinamica del codice, un documento di *README* contenente le informazioni specifiche per il servizio e un *Dockerfile* responsabile della creazione dei *container*. Lo strumento principale che ho utilizzato durante la codifica è l'*editor* Visual Studio Code, indicato in tabella **??**, grazie al quale ho utilizzato gli strumenti di analisi statica e dinamica del codice durante la sua scrittura.

Le attività di verifica garantiscono che l'esecuzione delle attività pianificate nel corso dello svolgimento di un progetto non introducano errori. Durante le attività di verifica

dei requisiti ho controllato la loro consistenza e la loro completezza, accertandomi che fossero chiaramente definiti, tuttavia ho tralasciato il controllo della loro realizzabilità confidando nelle mie capacità implementative. Durante le attività di progettazione ad alto livello del prototipo ho iterato le attività di verifica sui diagrammi delle classi UML al fine di verificare che essi rispondessero a due criteri:

- il grado di accoppiamento medio delle classi, calcolato come differenza in valore assoluto tra le relazioni in ingresso e le relazioni in uscita per ogni classe, sia nell'intervallo [0, 2] (ogni classe può utilizzare in media fino a un massimo di due altre classi);
- le classi all'interno di ciascun microservizio offrano le funzionalità essenziali per soddisfare i requisiti correlati e non dipendano dagli altri servizi per funzionare.

Ai due criteri ho associato gradi di importanza non equivalenti: ho preferito concentrarmi sul secondo criterio, che ho potuto applicare a tutti i servizi ad eccezione del servizio API intermediario con i *client*) per acquisire esperienza nell'applicazione del concetto *micro* ai servizi richiesto dalle archietture a microservizi. Ho svolto le attività di verifica durante l'implementazione del *software* da due lati:

- ho pianificato il primo insieme di attività del processo di verifica del prodotto utilizzando gli strumenti di analisi statica per prevenire errori comuni e facilmente risolvibili automaticamente;
- ho pianificato il secondo insieme di attività del processo di verifica del prodotto effettuando prove durante l'esecuzione del *software*.

Ho condotto l'analisi dinamica del prodotto sia con l'ausilio di strumenti di test automatici, sia effettuando prove manuali, simulando l'interazione di un *client*, inteso come attore, con i servizi. Ho progettato i test per raggiungere la copertura del codice fissata durante l'attività di Analisi dei Requisiti (riferimento 3.6). Durante la progettazione dei test ho scelto di approfondire la progettazione dei test d'unità di tutte le componenti progettate come primo *step* mentre ho tralasciato la progettazione dei test d'integrazione dei servizi: ho perseguito questa scelta perché ho ritenuto più importante aumentare l'affidabilità intrinseca di ciascun servizio indipendentemente dagli altri. Uno dei fattori che mi ha consentito di effetuare questa scelta consiste nell'individualità dell'implementazione del progetto: se al progetto avessero collaborato uno o più sviluppatori, avrebbe assunto maggiore importanza la corretta interazione tra i servizi. Inoltre, dal momento che i servizi che ho progettato hanno un flusso dei dati ben definito, mi ha permesso di tenere allineate le modifiche attraverso i servizi in efficienza. Nei test ho usato ampiamente *mock* e *stub* per isolare le eventuali dipendenze tra un modulo e un altro all'interno dello stesso servizio. Come avevo previsto durante lo studio degli argomenti, la componente in cui ho avuto alcune difficoltà nell'implementazione dei test consiste nei moduli che dialogano con un *database*: per arginare il problema ho utilizzato librerie accessorie presenti su npm per simulare le risposte che i *database* inviano nel caso in cui si verifichino errori. In questo modo ho potuto testare con successo anche eventuali malfunzionamenti relativi al collegamento con i *database*. Ho elencato i risultati della copertura del codice raggiunta al termine dello stage nella tabella 3.10.

Tabella 3.10: Tabella che specifica la copertura del codice raggiunta per ciascun servizio

Servizio	***Statement coverage***	***Branch coverage***
MQTT Broker	93 %	90 %
Sensore di temperatura "virtualizzato"	97 %	85 %
Temperatura	92 %	81 %
Lampada *smart* "virtualizzata"	83 %	67 %
Illuminazione	91 %	65 %
Informazioni dispositivi	90 %	77 %
Preferenze utente	90 %	79 %
API	70 %	70 %
Applicazione *web*	50 %	63 %
Totale	85,44 %	75,22 %

3.5 Validazione dei Requisiti

Nella tabella 3.11) ho elencato lo stato dell'implementazione dei requisiti citati in Analisi (riferimento 3.2)).

Tabella 3.11: Tabella dei requisiti funzionali

Identificativo	**Categoria**	**Esito**
RMF1	Obbligatorio	Soddisfatto
RMF2	Obbligatorio	Soddisfatto
RAF3	Desiderabile	Omesso
RMF4	Obbligatorio	Soddisfatto
RAF5	Desiderabile	Omesso.
RAF6	Desiderabile	Omesso.
RAF7	Desiderabile	Omesso
RMF8	Obbligatorio	Soddisfatto
RMF9	Obbligatorio	Soddisfatto
RMF10	Obbligatorio	Parzialmente soddisfatto
ROQ1	Opzionale	Soddisfatto
RMO1	Obbligatorio	Soddisfatto
RAO2	Desiderabile	Soddisfatto
RAO3	Desiderabile	Soddisfatto
RMO4	Obbligatorio	Soddisfatto

Durante lo svolgimento dello stage ho deciso di non implementare i requisiti relativi alla personalizzazione dei gruppi di dispositivi perché ho incontrato alcune difficoltà relative alla composizione dei servizi che hanno rallentato l'implementazione delle funzionalità dei servizi. Come avevo previsto in sede di valutazione dei rischi (riferimento 1.3), la difficoltà nel reperimento delle informazioni correlate ad esempi pratici delle architetture a microservizi ha causato un generale rallentamento delle attività di sviluppo. In particolare, ho avuto difficoltà in due momenti:

- durante lo sviluppo di una soluzione sperimentale che permettesse di utilizzare la tecnica dello sharding[g] tra molteplici *database* in un contesto di utilizzo basato sui *container*;

- durante lo sviluppo del servizio di integrazione delle API (servizio API citato in 3.3) fornite dai singoli microservizi.

Ho riscontrato la prima problematica a causa della documentazione legata allo *sharding* del *database* considerato: dal momento che essa è indirizzata per installazioni dei *database* direttamente sui calcolatori ho avuto difficoltà a trasportare quelle istruzioni nell'ambito della composizione di *container*, richiedendo una quantità superiore a quella che avevo valutato per essere implementata.
Proseguendo con la seconda criticità, questa si è verificata a causa di una mia incomprensione del meccanismo di condivisione della rete nell'ambito dell'orchestrazione dei servizi attraverso Docker Compose: nella pratica i servizi legati alla temperatura, all'illuminazione, alla gestione delle informazioni dei dispositivi e alle preferenze utente potevano essere interrogati individualmente, tuttavia non riuscivano a comunicare con il servizio di integrazione (servizio API) perchè Docker Compose internamente isolava i *container* in sottoreti virtuali senza possibilità di comunicare tra loro. Ho risolto il problema definendo manualmente una nuova sottorete virtuale, comune a tutti i servizi, che ha permesso al sistema di funzionare correttamente.
Per quanto riguarda il requisito RMF10, relativo alla visualizzazione delle statistiche di sistema, ho deciso di implementare solamente la componente di controllo della salute dei servizi che compongono il sistema *healthcheck*, tralasciando le componenti di integrazione dei dati raccolti dai servizi e le componenti di visualizzazione dei dati citati.
A causa dei rallentamenti menzionati ho scelto di non implementare alcune funzionalità perchè, sebbene le avessi catalogate come desiderabili, la quantità di risorse rimanenti non sarebbe stata sufficiente alla loro corretta implementazione, quindi ho preferito allocare tempo per migliorare la qualità del *software* sviluppato aumentando la copertura del codice sorgente, implementando un numero maggiore di test di unità. Mi è ora chiaro che alcuni dei requisiti da me classificati avrebbero richiesto ulteriori passaggi di verifica: la fattibilità e la valutazione più attenta del rapporto importanza della funzionalità paragonato alle risorse a disposizione avrebbero evidenziato che alcuni requisiti non erano implementabili nelle ore destinate al progetto di stage.

Capitolo 4

Valutazione retrospettiva

Nelle sezioni di questo capitolo parlerò dell'esperienza avuta durante lo svolgimento dello stage, parlando delle aspettative descritte nel primo capitolo e raffrontandole con le reali attività svolte

4.1 Valutazione raggiungimento degli obiettivi

Per ogni obiettivo definito precedentemente valuterei il suo soddisfacimento e descriverei le problematiche rilevate durante lo svolgimento dello stage.

4.2 Conoscenze acquisite

Autovalutazione delle conoscenze acquisite, ragionando in termini di aspettative iniziali e menzionando le parti che hanno causato difficoltà nello svolgimento del progetto.

4.3 Conclusioni

Glossario

Abstract Factory è un Pattern[g] creazionale che fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte dei *client* di specificare quale classe istanziare. Questo pattern permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il *client*, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti. 33

API[g] in informatica con il termine *Application Programming Interface API* (ing. interfaccia di programmazione di un'applicazione) si indica ogni insieme di

procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 43

Aspettativa è un periodo di astensione dal lavoro, previsto dalla legge, che il datore di lavoro può concedere ad un proprio lavoratore per motivi familiari o personali, generalmente non retribuito. 12

Broker un broker nell'ambito MQTT è un server che reindirizza i messaggi pubblicati verso i client sottoscritti all'argomento di ciascun messaggio. 31

Composite è un Pattern[g] strutturale che permette di effettuare operazioni su gruppi di oggetti come se essi fossero l'istanza di un singolo oggetto e permette di manipolare oggetti singoli e loro composizioni in modo uniforme. 35

GitHub è un servizio che permette di mantenere documenti e progetti *software* all'interno di un controllo di versione distribuito. Il servizio è sviluppato dall'omonima azienda. 24

Handshake in informatica rappresenta il processo attraverso cui due elaboratori stabiliscono un insieme di regole comuni per la comunicazione di dati. 31

JavaScript è un linguaggio di programmazione nato per aggiungere dinamicità e interattività alle pagine renderizzate dai browser attraverso l'esecuzione di semplici istruzioni che manipolino la struttura della pagina *web* visualizzata. 21

Kernel è il sottosistema di un sistema operativo che fornisce ai processi in esecuzione sull'elaboratore l'accesso alle risorse fisiche, in modo controllato e sicuro. 5

Load balancer in informatica, il load balancer è lo strumento hardware o software attraverso cui viene distribuito un carico di lavoro su più risorse di elaborazione (*load balancing*). 6

Microsoft è un'azienda statunitense che opera nel campo dell'informatica, sviluppando sistemi operativi, *software* per la produttività aziendale e domestica ed elettronica di consumo. 24

Milestone nell'ambito della pianificazione di un progetto, indica il raggiungimento di obiettivi stabiliti in fase di definizione del progetto. 16

MQTT[g] è un protocollo di comunicazione leggero basato sullo scambio di messaggi tra dispositivi caratterizzati da risorse di elaborazione limitate in una rete inaffidabile. 25

Open source indica un prodotto *software* in cui il codice sorgente è pubblicato dagli autori al fine di favorirne lo studio, aumentarne la diffusione e permettere alla comunità di apportarvi modifiche. 21

Pattern in informatica è una soluzione testata e dimostrata ad un problema ricorrente. La sua accezione principale consiste in quella di *Design Pattern* (soluzioni progettuali). 41, 42

QoS[g] nell'ambito delle reti di telecomunicazioni, il QoS indica i parametri usati per qualificare le prestazioni e l'affidabilità di un servizio offerto dalla rete. 31

Repository è un sistema *software* che archivia un insieme di *file* e cartelle e le relative informazioni al fine di permettere la distribuzione sicura e affidabile di questo insieme di dati. 37

Sharding nell'ambito delle tecnologie per i *database*, è il procedimento attraverso cui un *database* principale partiziona i dati ricevuti su molteplici istanze del *database*, definite *slave*. I dati in una architettura di questo tipo sono frammentati tra *database* separati, potenzialmente aumentando le prestazioni e la resilienza dei dati. 39

Throughput indica la capacità di un canale di comunicazione di processare o trasmettere dati in uno specifico periodo di tempo. É una misura di produttività.. 6

UML[g] in ingegneria del software *UML, Unified Modeling Language* (ing. linguaggio di modellazione unificato) è un linguaggio di modellazione e specifica basato sul paradigma object-oriented. L'*UML* svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa tale linguaggio per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. 43

Acronimi

API Application Program Interface[g]. 5, 41

MQTT MQ Telemetry Transport. 42

QoS Quality of Service. 42

UML Unified Modeling Language[g]. 26, 43

Bibliografia

Riferimenti bibliografici

Stephens, Rod. *Beginning Software Engineering*. John Wiley & Sons, 2015 (cit. a p. 6).