POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Utilizzo dell'approccio architetturale a microservizi nella realizzazione di una soluzione di Industry 4.0

Relatore	Candidato
Prof. Riccardo SISTO	Edoardo DAZZARA

Aprile 2021

Sommario

Esistono due modalità con cui realizzare una applicazione distribuita: l'approccio architetturale monolitico e l'approccio architetturale a microservizi. L'approccio monolitico prevede che dall'insieme di funzionalità di interesse si definisca una interfaccia applicativa e la si affidi ad un servizio. L'applicazione quindi è composta esclusivamente da quel servizio. L'approccio a microservizi, invece, si basa sul dividere l'insieme di funzionalità in sottoinsiemi, da ognuno definire una distinta interfaccia applicativa e affidarla ad un servizio apposito. L'applicazione quindi è composta da un insieme di servizi che collaborano. Se si applica quest'ultimo approccio si ottiene che l'applicazione risultante può essere sviluppata non tutta assieme, essere modificata velocemente, essere scalata con un livello di granularità più basso della applicazione stessa e se ne possono riutilizzare parti per risolvere altri problemi. Adottando invece l'approccio monolitico non si ottiene niente di tutto ciò, ma l'applicazione risulta di più facile realizzazione. Questo vuol dire che in un caso semplice è bene non adottare l'approccio a microservizi, mentre in un caso complesso è doveroso adottarlo. Con il termine Industry 4.0 si fa riferimento alla quarta rivoluzione industriale nell'ambito della manifattura, in cui si ha la tendenza all'automazione del processo produttivo grazie all'utilizzo di soluzioni Internet of Things (IoT). Internet of Things è una terminologia usata per indicare soluzioni distribuite che permettono di collegare ad internet qualsiasi oggetto tangibile. Una delle tecnologie IoT più diffuse è la tecnologia Radio-Frequency IDentification (RFID). Il punto di partenza del lavoro di tesi qui descritto è una soluzione Industry 4.0 esistente che automatizza la logistica di uno specifico processo produttivo tramite un particolare utilizzo della tecnologia RFID. Questa applicazione è stata realizzata seguendo l'approccio monolitico. Col tempo, a causa del fatto che l'automazione della logistica è la funzionalità di base per automatizzare un qualsiasi processo produttivo, le sono state aggiunte altre funzionalità che si basano su essa, impattano su essa e sono volte all'automazione di tale processo. Si sono riscontrati limiti di un certo rilievo in tale applicazione, come l'impossibilità di scalarla con una granularità fine o di modificarla velocemente. La natura di tali limiti ha fatto comprendere che la causa di essi è l'approccio architetturale scelto, evidentemente errato. È nato quindi il desiderio di avere tali funzionalità

realizzate seguendo l'approccio a microservizi. Il modo in cui realizzarle, però, si è scelto dover essere incrementale, ovvero in modo da avere soluzioni diverse in cui si aggiunge man mano una funzionalità. Il motivo di ciò è che in questo modo si hanno a disposizione soluzioni che permettono, adattandole per applicarle ad un altro processo, di poter soddisfare esigenze aziendali diverse. L'obiettivo del lavoro qui descritto è quello di creare, seguendo l'approccio a microservizi, una applicazione distribuita che automatizzi la logistica dello stesso processo produttivo dell'applicazione monolitica. Si è deciso che tale applicazione a microservizi debba realizzare tale funzionalità utilizzando la tecnologia RFID nello stesso modo in cui è utilizzata nella applicazione monolitica. La ragione di ciò è perché tale modo è robusto, performante ed indipendente dallo specifico processo produttivo (per cui ottimo nell'ottica del riadattamento). Siccome l'approccio a microservizi favorisce i cambiamenti e siccome tale modo di utilizzare la tecnologia RFID è indipendente dal processo produttivo si è deciso di porre come obiettivo anche il fatto che la soluzione da realizzare debba essere adattabile con facilità ad un qualsiasi altro processo produttivo.

Indice

Elenco delle figure

Acronimi

IoT
Internet of Things

RFID
Radio-Frequency IDentification

REST
REpresentational State Transfer

HATEOAS
Hypermedia As The Engine of Application State

URI
Uniform Resource Identifier

web API
web Application Programming Interface

OSI
Open Systems Interconnection

DBMS
DataBase Management System

SOA
Service-Oriented Architecture

SPOF
Single Point Of Failure

CRUD

Create-Read-Update-Delete

Capitolo 1

Introduzione

1.1 Scopo di questo capitolo

Inizialmente verrà introdotto lo scenario di interesse. Successivamente si presenterà l'obiettivo che si è voluto perseguire, proseguendo quindi con un breve accenno alla metodologia applicata per raggiungerlo. In conclusione verrà indicato quali sono i contenuti dei capitoli successivi.

1.2 Scenario di interesse

Grazie a concetti quali Industry 4.0 e Internet of Things i campi di applicazione dell'ingegneria informatica, per altro già notevolmente estesi, sono tuttora oggetto di una notevole espansione.

Con il termine Industry 4.0 si vuol fare riferimento a quella che viene considerata come la quarta rivoluzione industriale nell'ambito della manifattura. La prima rivoluzione industriale vede la produzione mediante macchinari sostituire la produzione manuale, grazie all'utilizzo della forza del vapore o dell'acqua. Nel corso della seconda rivoluzione industriale, invece, grazie all'installazione di reti ferroviarie estese, all'utilizzo dell'elettricità e al concetto della catena di montaggio, si ha il passaggio alla produzione di massa. L'avvento dei calcolatori è alla base della terza rivoluzione industriale, detta anche rivoluzione digitale a causa del fatto che da tecnologie meccaniche o analogiche si passa a tecnologie digitali. Semplificando la questione, se nel corso della terza rivoluzione industriale si aveva una concezione del calcolatore come quella di uno strumento a sé stante, nella quarta rivoluzione industriale la connessione e la comunicazione tra calcolatori diventano il centro di un discorso in cui un calcolatore viene quindi visto come un nodo di una rete. Ciò che definisce la quarta rivoluzione industriale è la tendenza all'automazione del processo produttivo, grazie all'utilizzo di soluzioni Internet of Things (IoT).

IoT è una terminologia usata per indicare soluzioni distribuite che permettono di collegare ad internet qualsiasi oggetto tangibile. Il motivo per cui valga la pena fare ciò è che l'oggetto in questione così acquisisce nuove capacità come l'elaborazione dati, la localizzazione, l'interazione con l'ambiente esterno etc. Una delle tecnologie IoT più diffuse (e una pietra miliare del mondo IoT) è la tecnologia Radio-Frequency IDentification (RFID), che verrà trattata in seguito. Il collegare ad internet qualsiasi oggetto tangibile permette opportunità praticamente infinite, a molte delle quali oggi non si pensa neppure.

Esistono due modalità con cui realizzare una applicazione distribuita, indipendentemente dal fatto che sia una soluzione IoT (eventualmente applicata all'Industry 4.0) o no: l'approccio architetturale monolitico e l'approccio architetturale a microservizi. L'approccio monolitico prevede che dall'insieme di funzionalità di interesse si definisca una interfaccia applicativa e la si affidi ad un servizio. L'applicazione quindi è composta esclusivamente da quel servizio. L'approccio a microservizi, invece, si basa sul dividere l'insieme di funzionalità in sottoinsiemi, da ognuno definire una distinta interfaccia applicativa e affidarla ad un servizio apposito. L'applicazione quindi è composta da un insieme di servizi che collaborano. Se si applica quest'ultimo approccio nel modo opportuno si ottiene che l'applicazione risultante può essere sviluppata non tutta assieme, può essere modificata velocemente, può essere scalata sulla base della domanda attuale con un livello di granularità più basso della applicazione stessa (ovvero il singolo servizio) e se ne possono riutilizzare le singole funzionalità per risolvere altri problemi. Adottando invece l'approccio monolitico non si ottiene niente di tutto ciò, ma l'applicazione risulta di più facile realizzazione. Questo vuol dire che in un caso semplice, in cui non sono necessarie le capacità che l'approccio a microservizi offrirebbe è bene non adottarlo per non aggiungere complessità non necessaria, mentre in un caso complesso, in cui sono necessarie tali capacità, è doveroso adottarlo.

I concetti sopra esposti se da un lato aprono le porte a nuove opportunità, dall'altro chiaramente le aprono anche a nuove sfide.

1.3 Obiettivo del lavoro svolto

Nel contesto precedentemente descritto, quindi, il compito dell'ingegneria informatica è quello di rendere possibile e supportare l'automazione di un qualche processo produttivo di una azienda manifatturiera sfruttando la rete e la possibilità di connettere a questa qualsiasi oggetto fisico.

Considerando la situazione attuale, in cui dato un problema ciò che ci si trova a dover fare è scegliere quale delle diverse soluzioni a questo adottare, si ritiene importante far notare che le motivazioni che guidano tale scelta riguardano in generale aspetti quali la qualità della soluzione ottenuta e le prestazioni del risultato

prodotto. Sulla base di ciò, quindi, nell'eseguire la propria parte, l'ingegneria informatica deve tenere in considerazione anche tali aspetti al fine di vedere il frutto del suo lavoro effettivamente utilizzato.

All'interno dello scenario precedentemente descritto si colloca il lavoro di tesi svolto. Il punto di partenza è una soluzione Industry 4.0 esistente che automatizza la logistica di uno specifico processo produttivo tramite un particolare utilizzo della tecnologia RFID. Questa applicazione è stata realizzata seguendo l'approccio architetturale monolitico. Col tempo, a causa del fatto che l'automazione della logistica è la funzionalità di base per far tendere all'automazione completa un qualsiasi processo produttivo, le sono state aggiunte altre funzionalità che si basano su essa, impattano su essa e sono volte all'aumentare il supporto all'automazione di tale processo. Si sono riscontrati alcuni limiti di un certo rilievo in tale applicazione, come ad esempio l'impossibilità di scalarla con una granularità fine oppure l'impossibilità di modificarla velocemente o ancora l'impossibilità di riutilizzarne alcune funzionalità in problemi diversi che le necessitano. La natura di tali limiti ha fatto comprendere che la causa di essi è da ritrovarsi nell'approccio architetturale scelto, evidentemente errato. È nato quindi il desiderio di avere tali funzionalità realizzate seguendo l'approccio architetturale a microservizi. Il modo in cui realizzarle, però, si è scelto dover essere incrementale, ovvero in modo da avere soluzioni diverse in cui si aggiunge man mano una funzionalità alla volta. Il motivo di ciò è che in questo modo si hanno a disposizione soluzioni diverse che permettono, adattandole per applicarle ad un altro processo specifico del caso, di poter soddisfare esigenze aziendali diverse, in cui il livello di automazione richiesto può non essere lo stesso. In questa tesi l'attenzione è posta sulla funzionalità di automazione della logistica. Si può realizzare una soluzione Industry 4.0 che automatizzi la logistica di quel processo produttivo seguendo l'approccio architetturale a microservizi in due modi: da zero oppure migrandola dalla applicazione monolitica. Qui si è scelta la seconda alternativa, ovvero si è deciso di dover di soddisfare i requisiti della applicazione monolitica esclusivamente per quanto riguarda l'automazione della logistica seguendo l'approccio a microservizi. La ragione di ciò è perché il particolare utilizzo che si fa in tale applicazione monolitica della tecnologia RFID per realizzare la funzionalità di automazione della logistica è robusto, performante e non dipendente dallo specifico processo produttivo (per cui ottimo nell'ottica del riadattamento).

L'obiettivo del lavoro di questa tesi pertanto è stato il progettare e quindi poi realizzare, seguendo l'approccio architetturale a microservizi, una applicazione distribuita che automatizzasse la logistica di quel processo produttivo aziendale tramite quel particolare utilizzo della tecnologia RFID. Siccome l'approccio a microservizi favorisce i cambiamenti e siccome tale modo di utilizzare la tecnologia RFID è indipendente dal processo produttivo si è deciso di porre come obiettivo anche il fatto che la soluzione da realizzare dovesse essere adattabile con facilità ad un qualsiasi altro processo produttivo. La natura del lavoro svolto, in cui per

ragioni aziendali si è scelto di non implementare tutte le funzionalità che sono presenti nella applicazione monolitica, fa si che confronti quantitativi tra le due soluzioni non abbiano ragione di essere posti.

Si espone qui che cosa voglia dire in termini concreti avere una soluzione che automatizza la logistica di un processo produttivo: si migliora il processo su cui la si usa in termini di produttività, costi e tempistiche, quindi, più in generale, la qualità stessa del processo. Risulta infatti evidente che se il problema della logistica è automatizzato richiede meno tempo per essere svolto, pertanto fa si che si abbiano meno costi e permette di impiegare tale tempo risparmiato in una ulteriore produzione dei beni in questione, aumentando la produttività.

1.4 Metodologia applicata e tematiche fondamentali trattate

La metodologia applicata nel conseguire l'obiettivo del lavoro qui esposto consiste in primo luogo in un'analisi qualitativa critica esclusivamente della parte dell'applicazione monolitica atta alla funzionalità di automazione della logistica, con il fine di evidenziare i limiti ma non solo. In secondo luogo quindi si è svolto un lavoro di design della soluzione a microservizi, che costituisce il cuore del lavoro qui descritto. Considerando quanto esposto in precedenza, si fa ora notare che nella progettazione di tale soluzione si è prestata particolare attenzione non soltanto al risolvere il problema in sé e alla modalità con cui lo si sarebbe risolto, ma anche agli aspetti legati alla qualità di ciò che si stava creando, come la possibilità di riutilizzo, la flessibilità, la robustezza e l'eleganza, nonché alle sue performance. Alla fase di progettazione è seguita la fase di realizzazione di tale soluzione e quindi una fase di testing esclusivamente funzionale di cui si riporteranno risultati ottenuti. Infine è stata analizzata in modo critico la soluzione proposta valutando la sua capacità o incapacità di raggiungere l'obiettivo iniziale.

Per poter raggiungere l'obiettivo di questa tesi è stata necessaria la conoscenza di alcuni concetti teorici che verranno affrontati nei successivi capitoli, esponendo di ognuno di questi esclusivamente le tematiche utili al fine di comprendere il lavoro che è stato svolto. Tra questi troviamo: le applicazioni distribuite, dove verrà indicato in cosa consiste la loro realizzazione, in particolare nei casi specifici di interesse per il lavoro svolto; le differenti modalità di realizzazione possibili, ovvero l'approccio monolitico e l'approccio a microservizi, dove verranno evidenziati i vantaggi e gli svantaggi di entrambe; un'introduzione al mondo IoT, in cui l'attenzione verrà posta sulla tecnologia RFID.

1.5 Contenuto dei capitoli successivi

Questa trattazione è stata pensata per permettere al lettore inizialmente di acquisire progressivamente la conoscenza dei concetti teorici necessari alla comprensione del lavoro svolto. In particolare nel prossimo capitolo, il secondo, si tratteranno le applicazioni distribuite. Dopodiché verrà affrontato il tema delle modalità di realizzazione di queste, nel terzo capitolo. Il quarto capitolo è dedicato al mondo IoT e alla tecnologia RFID. Una volta acquisiti tali concetti si è voluto fare in modo che il lettore potesse comprendere a pieno il problema che si vuole risolvere e i requisiti della soluzione da creare, scopo a cui è dedicato il quinto capitolo. Nel sesto capitolo è riportato il risultato dell'analisi dell'applicazione monolitica. Il settimo capitolo è il cuore di questo scritto: consiste infatti nell'esposizione del frutto del lavoro di design. I risultati ottenuti dai test funzionali che si sono eseguiti sono riportati nell'ottavo capitolo. Nel nono capitolo si espone il frutto dell'analisi condotta sulla soluzione proposta e la valutazione del raggiungimento degli obiettivi iniziali. L'ultimo capitolo, il decimo, contiene possibili sviluppi futuri per il lavoro svolto.

Capitolo 2

Applicazioni distribuite

2.1 Scopo di questo capitolo

Questo capitolo e il successivo sono volti a dare al lettore le conoscenze che si sono rivelate necessarie per attuare l'analisi o la creazione di un'applicazione distribuita. In particolare in questo capitolo si tratteranno le applicazioni distribuite e in cosa consista la realizzazione di una di esse, nel successivo invece le diverse modalità di realizzazione esistenti. Inizialmente verrà quindi data la definizione di applicazione, specializzandola nel caso distribuito. Successivamente si esporrà in cosa consiste la realizzazione di una applicazione di tale tipo, portando due esempi specifici che risulteranno essere esattamente quelli adottati nel perseguire l'obiettivo di questa tesi. Esula dagli scopi di questo testo essere esaustivo in merito alla tematica trattata, obiettivo che sarebbe in ogni caso irraggiungibile per un singolo scritto vista la complessità e l'estensione dell'argomento, ragion per cui ove si ritiene necessario che il lettore approfondisca le tematiche qui introdotte verranno indicati riferimenti utili a tale scopo.

2.2 Definizione di applicazione

Un essere umano vuole raggiungere un qualche obiettivo in uno specifico ambito. Ogni ambito ha un gergo specifico, un insieme di concetti fondamentali su cui si basa e definisce un insieme di regole. L'essere umano raggiunge l'obiettivo attraverso un processo, ovvero una serie di task. Un task è un'operazione che manipola i concetti del dominio in questione sottostando alle regole da quest'ultimo definite.

L'aspetto interessante è notare che in tutto ciò non c'è alcun riferimento al concetto di applicazione: un'applicazione è un qualcosa di non necessario. Il ruolo di un'applicazione, nel contesto sopra indicato, è quello di permettere all'essere umano di fare uno o più task in modo migliore rispetto a come farebbe senza

questa. Gli assi lungo i quali misurare questo miglioramento dipendono dall'ambito specifico e possono includere misure oggettive quali il tempo impiegato o il numero di errori (critici e non) commessi, oppure misure soggettive come l'impegno mentale richiesto, lo sforzo fisico richiesto o la soddisfazione percepita. Un'applicazione, nella sua accezione ultima, consente all'utente di interagire in modo controllato con dei dati. I dati rappresentano i concetti fondamentali del dominio e l'utente vi interagisce in modo controllato tramite essa, ovvero manipola tali dati secondo le regole dell'ambito specifico. In questa trattazione distinguiamo le applicazioni in applicazioni standalone e applicazioni distribuite.

2.3 Applicazioni distribuite: caratteristiche

Un'applicazione standalone è un'applicazione che è eseguita su un solo calcolatore. Un'applicazione distribuita è un'applicazione, invece, che è eseguita in un ambiente distribuito, ovvero su un sistema costituito da una collezione di calcolatori autonomi interconnessi tramite una rete. Da ciò si evince facilmente che ognuna di queste tipologie ha delle caratteristiche sue proprie. In questo scritto l'attenzione è posta esclusivamente sulle applicazioni distribuite e sulle loro peculiarità.

È utile notare che per essere definita tale, quindi, un'applicazione distribuita deve essere data da un insieme di calcolatori, di almeno due elementi, interconnessi tramite una rete, tale che su ognuno di questi, semplificando la questione, sia eseguito un processo riguardante la suddetta applicazione. Tali processi comunicano tra loro mediante la rete. Questo fa emergere le seguenti considerazioni. Nel software distribuito l'esecuzione è sempre parallela. Consideriamo infatti, in modo semplicistico ma accurato, un calcolatore come una risorsa e un processo come una entità richiedente una risorsa. Ovviamente un processo può richiedere esclusivamente la risorsa calcolatore sui cui si trova. Dal punto di vista della applicazione quindi si ha che i richiedenti possono essere soddisfatti contemporaneamente grazie al fatto che non ci siano più richiedenti per la stessa risorsa e che per ogni richiedente ci sia una risorsa. Un'applicazione distribuita è eseguita in modo reattivo: al verificarsi di eventi di interesse reagisce nel modo che ritiene opportuno. Gli eventi di interesse tipicamente riguardano l'utente oppure la rete. Questa è la modalità con cui un'applicazione distribuita fornisce la funzionalità per cui è stata concepita. Il fatto che la comunicazione tra i processi avvenga tramite la rete implica che risultino necessari meccanismi per la localizzazione dei processi all'interno del sistema distribuito. Far comunicare processi in questo modo impatta maggiormente sulle prestazioni rispetto a farli comunicare in un qualsiasi altro modo. Inoltre è sempre bene ricordare che la comunicazione tramite la rete può fallire a causa di problemi di varia natura. Si vuole anche permettere che i calcolatori del sistema distribuito possano essere piattaforme tra loro eterogenee in

merito a hardware e software e questo incide notevolmente sulla realizzazione di questa categoria di applicazioni. Infine va considerato che il software distribuito è maggiormente esposto ad attacchi che compromettono la sicurezza a causa del fatto che si basi sull'utilizzo della rete.

2.4 Realizzare un'applicazione distribuita

Consideriamo il modello Open Systems Interconnection (OSI) di stack di rete. Per realizzare un'applicazione distribuita si deve inizialmente decidere di quale livello, almeno a partire dal livello trasporto andando verso l'alto, sfruttare i servizi offerti per per fornire a propria volta una qualche funzionalità che, in ultima analisi, serve all'essere umano. Indipendentemente dal livello sopra al quale si è deciso di porsi capita che l'interfacciarsi con esso sia ora di fatto indipendente da una qualche specifica piattaforma grazie ad un processo di standardizzazione avvenuto nel corso del tempo. Sulla base dello strato a cui ci si affida si ottengono funzionalità sempre più di alto livello e sempre più complesse. Una volta scelto il livello, è necessario quindi scegliere quale dei protocolli possibili utilizzare. Tale scelta è dettata dal fatto che i servizi forniti da protocolli diversi dello stesso livello sono tra loro differenti. A questo punto tipicamente ci si trova di fronte ad un bivio: utilizzare il protocollo scelto così com'è oppure definire un protocollo di livello superiore, che a sua volta si serve del protocollo scelto, e quindi utilizzare questo nuovo protocollo. Se si sceglie la seconda strada allora si dovrà definire e implementare un protocollo di comunicazione e renderlo utilizzabile mediante interfacce.

Nel definire un protocollo di comunicazione si deve definire il modello di interazione adottato. Tipicamente la scelta è tra il modello client-server e il modello peer-to-peer. In entrambi i casi i ruoli possibili sono quello di cliente, colui il quale fa una richiesta, e servitore, colui il quale a fronte di una richiesta fornisce una risposta. Nell'architettura client-server vi è un host sempre attivo, chiamato server, che risponde alle richieste di servizio di molti altri host, detti client. I client non comunicano direttamente tra loro. Nell'architettura peer-to peer ogni host è uguale a ogni altro host, è intermittente e può sia fare una richiesta ad un qualsiasi altro host sia rispondere ad una richiesta di un qualsiasi altro host, quindi gli host (peer) comunicano direttamente tra loro. In questa trattazione ci concentriamo sui protocolli ad architettura client-server.

Indipendentemente dalla strada intrapresa, alla fine si arriva ad avere un protocollo da utilizzare. Utilizzare un protocollo vuol dire definire una qualche logica che se ne serva per fornire la funzionalità richiesta alla applicazione distribuita. Le interfacce che un qualsiasi protocollo fornisce sono utili lato client per poter creare e mandare una richiesta e quindi riceverne la risposta, lato server per poter ricevere una richiesta e quindi crearne la risposta e mandarla al client che l'ha

fatta. Il modo in cui si definisce la logica detta sopra è tipicamente tramite la definizione dell'interfaccia applicativa e quindi poi la sua implementazione lato server e il suo utilizzo lato client. L'interfaccia applicativa è l'interfaccia che il server espone ad ogni client e tramite la quale è in grado di fornire le funzionalità richieste alla applicazione. Tale interfaccia dipende dal protocollo che si è scelto di utilizzare. Semplificando la questione essa dice, data una funzionalità, cosa mettere nel pacchetto di richiesta che si deve usare per ottenerla ed, eventualmente, cosa aspettarsi di ricevere nel pacchetto di risposta a questa. Dalla figura 2.1 possiamo riconoscere la differenza tra l'interfaccia applicativa e un'interfaccia tra due livelli OSI: l'interfaccia applicativa dice, dato un obiettivo, ovvero una funzionalità, come servirsi dello stack di rete al fine di utilizzarla. In seguito verranno trattati rispettivamente il protocollo HTTP e il protocollo MQTT, in particolare verrà trattato il loro utilizzo diretto.

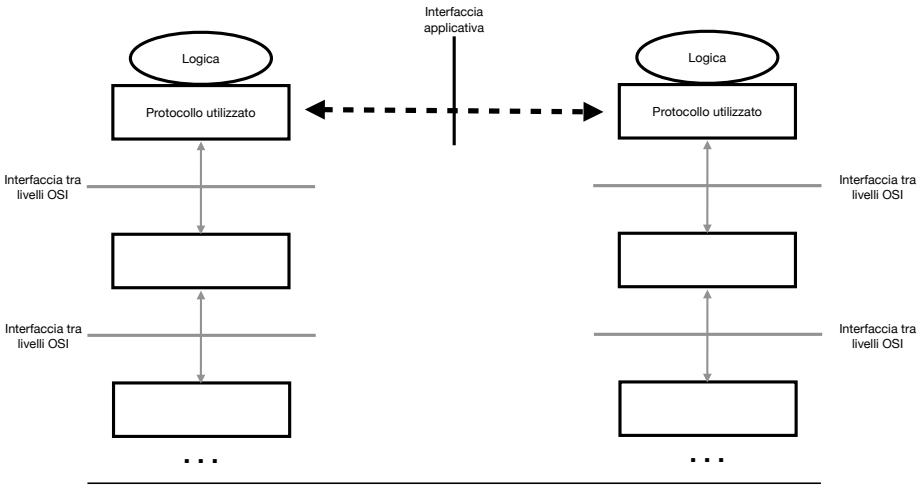


Figura 2.1: Interfacce: OSI e applicativa

2.5 HTTP

2.5.1 Premessa

Uno dei casi qui trattati è l'HTTP. All'inizio verrà fatta una breve introduzione al protocollo e in seguito si specificherà come servirsene per realizzare un'applicazione distribuita. Nel proseguire, sarà data una visione di più alto livello dell'HTTP e sarà fornito uno strumento da utilizzare quando si ha a che fare con tale protocollo: il modello di maturità di Richardson. Si concluderà quindi l'argomento con alcune considerazioni di carattere generale, utili sia in fase di analisi di un'applicazione esistente sia in fase di realizzazione di una nuova applicazione e derivanti dagli argomenti a queste precedentemente introdotti.

2.5.2 Introduzione al protocollo

L' HTTP è un protocollo di livello applicazione semplice e leggero. È di tipo client server, è privo di stati (ovvero la risposta ad una richiesta di un client non dipende dalle richieste precedenti di quel client) ed è in grado di trasportare dati arbitrari in termini di contenuto, dimensione e codifica. Fare una richiesta HTTP corrisponde al chiedere di eseguire una certa operazione su di una certa risorsa.

Secondo [1] una risorsa è una funzione del tempo tale che in un certo istante corrisponde ad un insieme di valori tra loro semanticamente equivalenti ed in alternativa. I valori trovati ad un certo istante di tempo sono istanze dei tipi di rappresentazione possibili per la risorsa. Volendo fare un esempio, se si considera la risorsa "la prima squadra del massimo torneo del campionato di calcio femminile italiano" allora si ha che questa può cambiare in funzione del tempo. Sono possibili diverse tipologie di rappresentazione per tale risorsa: una stringa indicante il nome della squadra o una immagine raffigurante lo stemma della squadra. In ogni caso il significato non cambia. Se ad un certo istante di tempo la prima squadra del massimo torneo del campionato di calcio femminile italiano fosse X allora in quell'istante l'insieme dei valori di tale risorsa sarebbe costituito dalla stringa "X" e dall'immagine con lo stemma di X. Ad un altro istante di tempo potrebbe essere ancora così o non essere più così. Consideriamo ora la risorsa "la squadra X del massimo torneo del campionato di calcio femminile italiano" e definiamo come tipi di rappresentazione possibili per essa una stringa indicante il nome o una immagine raffigurante lo stemma: abbiamo che ad un certo istante l'insieme dei valori di tale risorsa è costituito dalla stringa "X" e dall'immagine con lo stemma di X. Questo per far notare come sia possibile (e non crei alcun tipo di problema) il fatto che che due o più risorse distinte abbiano in un certo istante di tempo una o più rappresentazioni possibili identiche.

Ogni risorsa ha una semantica definita da chi ha definito la risorsa stessa. Le risorse possibili sono illimitate, non definite dal protocollo e sono identificate da un Uniform Resource Identifier (URI), il quale deve essere univoco a livello globale per evitare ambiguità. Le operazioni possibili sono limitate, definite dal protocollo e identificate da una stringa. Ogni operazione ha una sua semantica definita dal protocollo stesso. Per una trattazione completa su tale argomento si rimanda a [2].

2.5.3 Realizzare un'applicazione distribuita tramite HTTP

Avendo compreso ciò si vuole ora indicare come servirsi del protocollo HTTP per realizzare un'applicazione distribuita: si attua il processo di design della interfaccia applicativa, specializzandolo nel caso corrente ovvero HTTP. Dato l'insieme di funzionalità che la applicazione deve fornire si deve mappare ognuna di queste verso una coppia di informazioni: una risorsa ed una operazione (metodo) da eseguire su questa. Per poter fare ciò il punto di partenza è il design delle risorse. Abbiamo precedentemente definito un'applicazione come uno strumento che consente di interagire in modo controllato con i concetti fondamentali del dominio di business di interesse. Semplificando la questione si ha che una risorsa tipicamente corrisponde ad un concetto fondamentale del dominio in questione. Una volta individuate le risorse è necessario assegnare ad ognuna un URI univoco a livello globale ed è necessario definire per ognuna almeno un tipo di rappresentazione. Arrivati a questo punto si può quindi far corrispondere ad ogni funzionalità della applicazione una coppia univoca di informazioni, costituita appunto dall'URI di una risorsa e da uno dei metodi dell'HTTP. La risorsa è data da quella che indica il concetto su cui si vuole operare mentre il metodo va scelto in base alla semantica dell'operazione che si vuole eseguire. Il contenuto della richiesta, dove previsto, è auspicabilmente la rappresentazione futura della risorsa oppure un insieme di dati a lei destinati per una qualche elaborazione. Il contenuto della risposta, invece, dove previsto, è la rappresentazione corrente della risorsa oppure il frutto dell'elaborazione dell'insieme di dati ricevuto. Definita l'interfaccia applicativa la si implementa lato server, mentre lato client la si usa a fronte di input provenienti dall'utilizzatore.

2.5.4 REST

Quanto descritto in precedenza si basa sulla condotta secondo la quale dato uno strumento è doveroso servirsene esclusivamente secondo l'uso per cui tale strumento è stato pensato. L'HTTP, per come è stato pensato e per come è stato presentato qui, può essere visto come un esempio, probabilmente il più importante, di applicazione della filosofia REpresentational State Transfer (REST). I concetti su cui si basa l'HTTP quali risorsa, rappresentazione di una risorsa etc, infatti, sono i concetti definiti dalla filosofia REST, la quale è un modo di progettare

applicazioni distribuite. È quindi possibile applicare tali concetti utilizzando un qualsiasi altro protocollo. Si potrebbe quindi pensare che, data una qualsiasi applicazione distribuita che usi l'HTTP, questa sia classificabile come applicazione conforme alla filosofia REST. Se la condotta descritta in precedenza fosse scontata allora sarebbe corretto pensare in questo modo, ma purtroppo ciò non è. Per questo motivo è opportuno riformulare la precedente frase affermando che una qualsiasi applicazione distribuita che usi HTTP nel modo corretto è classificabile come applicazione conforme alla filosofia REST. Leonard Richardson, nel 2008, ha definito un modello di maturità per una web Application Programming Interface (web API), il quale permette di classificare un'applicazione distribuita sulla base della sua conformità allo standard REST. Per poterlo comprendere, però, risulta necessario trattare un altro concetto definito dalla filosofia REST e applicabile nell'HTTP: Hypermedia As The Engine of Application State (HATEOAS).

2.5.5 HATEOAS

HATEOAS è un principio secondo il quale lo stato di un'interazione tra un client ed un server non deve essere memorizzato dal server, ma deve essere memorizzato dal client. Con stato della interazione si intende la risorsa su cui il client si trova in quel momento. Se fosse compito del server memorizzare tale informazione, allora si dovrebbero avere messaggi per permettere al client di cambiare posizione ed avere messaggi per permettere al client di operare nella posizione in cui si trova. Memorizzando l'informazione corrispondente allo stato sul client, invece, risultano necessari e sufficienti messaggi esclusivamente operativi in cui però deve essere specificata sia l'operazione che si vuole fare sia la posizione in cui la si vuole fare. Per adottare il principio HATEOAS c'è bisogno di un meccanismo per permettere al client di sapere dove posizionarsi e di un meccanismo che, data una risorsa, permetta di scoprire quali sono le operazioni su questa possibili. Il primo meccanismo sfrutta il fatto che i concetti di un determinato dominio sono tutti in relazione e si basa sulla possibilità di includere nella rappresentazione di una qualsiasi risorsa i riferimenti alle risorse a questa collegate (indicando per ognuno il significato del collegamento). Esso consiste nello sfruttare tali riferimenti per spostarsi da una risorsa all'altra. Il secondo meccanismo richiede l'esistenza di un'operazione atta a risolvere tale scopo ed applicabile su qualsiasi risorsa (nell'HTTP è costituita dal metodo OPTIONS) e consiste quindi nello sfruttare tale operazione. Adottando questo principio si permette quindi al server di essere senza stato e alle applicazioni distribuite di essere auto descrittive.

2.5.6 Modello di maturità di Richardson

Il modello di maturità di Richardson prevede quattro possibili livelli, da zero a tre, dentro cui collocare un'applicazione distribuita.

Al livello zero troviamo le applicazioni distribuite che usano l'HTTP in questo modo: esiste una sola risorsa su cui è implementato un solo metodo (tipicamente POST). All'interno del corpo della richiesta viene di fatto specificata l'operazione di interesse e vengono collocati i dati di input per tale operazione. Nel corpo della risposta si trova il risultato della operazione richiesta.

Al livello uno, detto anche livello delle risorse, troviamo le applicazioni che usano l'HTTP in questo modo: ogni concetto del dominio di interesse ha una risorsa a lui corrispondente su cui è implementato un solo metodo (tipicamente POST). All'interno del corpo della richiesta viene di fatto specificata l'operazione di interesse sulla risorsa in questione e vengono collocati i dati di input per tale operazione. Nel corpo della risposta si trova il risultato della operazione richiesta.

Al livello due, detto anche livello dei verbi, troviamo le applicazioni che usano l'HTTP in questo modo: ogni concetto del dominio di interesse ha una risorsa a lui corrispondente su cui sono implementati diversi metodi HTTP. L'implementazione di ognuno di tali metodi è conforme alla semantica definita dal protocollo e si utilizzano i codici di stato, in modo opportuno, per esprimere gli esiti delle operazioni. All'interno del corpo della richieste e delle risposte si hanno le rappresentazioni delle risorse.

Al livello tre, detto anche dei collegamenti ipermediali, troviamo le applicazioni che usano l'HTTP in questo modo: sono conformi al livello due e inoltre capita che all'interno della rappresentazione di una risorsa si inseriscano i riferimenti alle risorse a questa collegate etichettandoli in modo da esplicitare la semantica di tale relazione e che si implementi in modo opportuno il metodo OPTIONS.

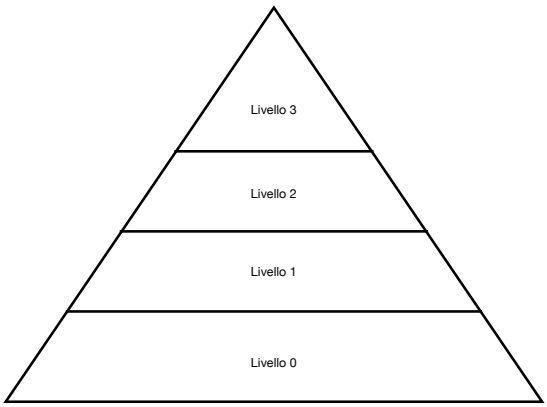


Figura 2.2: Modello di maturità di Richardson

2.5.7 Considerazioni generali

Nel perseguire l'obiettivo di questa tesi inizialmente si sono dovute giudicare le scelte altrui fatte nel realizzare la applicazione monolitica esistente, in seguito si sono dovute compiere delle scelte nel creare la applicazione a microservizi. Uno degli aspetti fondamentali su cui ci si è focalizzati è il seguente: l'aderenza ad una o più filosofie oppure la non aderenza ad alcuna filosofia. Il conformarsi ad uno standard non deve essere visto né come una seccatura fine a se stessa né come una limitazione alle proprie azioni. Seguire una filosofia, indipendentemente da quale essa sia, deve essere visto come il modo in cui ottenere una serie di benefici che solitamente si rivelano molto utili, a volte al punto tale da spingere a definirne una propria quando ci si accorge che è troppo tardi per seguirne una esistente. Tra i vantaggi ottenibili troviamo: la chiarezza, la capacità di essere compresi senza il bisogno di spiegarsi e il favorire evoluzioni future. La chiarezza la si ha grazie al fatto che uno standard pone dei vincoli da seguire. Tali vincoli possono sì essere visti come una limitazione, ma in realtà sono anche e soprattutto ciò che permette di evitare ambiguità. Il fatto di essere in grado di farsi comprendere senza la necessità di spiegarsi è davvero un punto saliente: se si è compreso lo standard allora automaticamente si comprenderanno tutti i lavori che a questo sono conformi perché non è necessario alcuno sforzo aggiuntivo oltre a quello fatto inizialmente per comprendere tale standard. Questo è il motivo per cui seguendo una certa filosofia si va verso l'auto descrizione. Un requisito fondamentale che un'opera di qualsiasi tipo deve avere per poter durare nel tempo è la capacità di evolversi per potersi adattare al contesto che la circonda, che via via cambia. Se ogni opera ha la sua filosofia o non ne ha affatto allora capita che, per poter far fronte ad un cambiamento, vada considerata singolarmente ognuna di queste, ripartendo quindi da zero nel passaggio dall'una all'altra. Adottando uno standard, invece, ci si può astrarre dall'opera specifica e si può cercare di definire un modo in cui lo standard in sé deve affrontare tale cambiamento. In questo modo automaticamente per ogni lavoro a questo conforme si sa già come comportarsi.

2.6 MQTT

2.6.1 Premessa

L'altro caso qui trattato è l'MQTT. Si comincerà con una breve introduzione al protocollo e in seguito si specificherà come realizzare un'applicazione distribuita che se ne serva.

2.6.2 Introduzione al protocollo

L'MQTT è un protocollo di livello applicazione di tipo client server publish-subscribe. È un protocollo leggero ed è stato pensato per essere facile da implementare lato client: è quindi ideale per essere utilizzato in ambienti con vincoli come quelli dell'IoT, dove è richiesto codice che occupi poca memoria e dove la banda è scarsa. È leggero poiché è un protocollo binario che aggiunge molto poco ai pacchetti che deve trasportare. Per una trattazione completa a riguardo si rimanda a [3]. Qui di seguito verranno affrontati i concetti fondamentali per questo protocollo. La trattazione sarà di alto livello e riguarderà esclusivamente gli aspetti necessari alla comprensione del lavoro di questa tesi.

L'architettura client server publish-subscribe è molto diversa da quella client server tradizionale. In questo caso si hanno tre attori coinvolti: pubblicatori (publisher), sottoscrittori (subscriber) e broker. Il broker è il server. Un pubblicatore è un client. Un sottoscrittore è un client. L'obiettivo è quello di far comunicare tra loro un pubblicatore con tutti e soli i sottoscrittori opportuni. Per realizzarlo si fa in questo modo: la comunicazione, per altro uno a molti, non avviene direttamente tra le parti interessate, ma avviene in primo luogo tra un pubblicatore ed il broker e, eventualmente, in secondo luogo tra il broker e tutti e soli i sottoscrittori opportuni. Il concetto fondamentale è quello di topic. Un topic, semplificando la questione, è un canale di comunicazione identificato da una stringa. Un client diventa un pubblicatore nel momento in cui pubblica qualcosa su un topic, ovvero nel momento in cui manda un messaggio di pubblicazione al broker indicando all'interno del campo per il topic la stringa corrispondente. Un client diventa un sottoscrittore nel momento in cui si sottoscrive ad un topic, ovvero nel momento in cui manda un messaggio di sottoscrizione al broker specificando il topic di interesse. Da quel momento, fino a che non verrà esplicitamente detto il contrario, quel client è considerato come interessato a quel topic. Il broker riceve un messaggio da un pubblicatore e quindi lo manda a tutti e soli i sottoscrittori opportuni, ovvero a quelli in quel momento interessati al topic su cui è stato pubblicato tale messaggio. Un pubblicatore non sa se esiste qualcuno che è interessato ai messaggi che pubblica e il fatto che li pubblichi non dipende da nessuna condizione a lui esterna, ma dipende esclusivamente dalla sua volontà. Un sottoscrittore non sa se esiste qualcuno che

pubblica sul topic a cui si è iscritto e il fatto che si sia iscritto a quel topic non è legato ad alcuna circostanza esterna, ma è legato esclusivamente alla sua volontà.

2.6.3 Realizzare un'applicazione distribuita tramite MQTT

Per realizzare un'applicazione distribuita servendosi dell'MQTT il procedimento è lo stesso che si adotterebbe per un qualsiasi altro protocollo: si attua il processo di design dell'interfaccia applicativa, specializzandolo nel caso corrente ovvero MQTT. Le caratteristiche precedentemente dette fanno capire che questo protocollo fornisce un meccanismo di trasporto dati unidirezionale uno a molti. Sebbene esso permetta anche la realizzazione di una applicazione distribuita con un flusso di informazioni bidirezionale, ripetendo il medesimo meccanismo nei due versi della comunicazione, risulta quindi sicuramente più adatto alla realizzazione di una applicazione distribuita con un flusso di informazioni unidirezionale, in cui di fatto quello che interessa non è il fare una richiesta per ricevere una risposta, ma è, a fronte di una generazione di dati, consegnarli a chi è incaricato di gestirli e ne è in grado. Per questo motivo l'insieme di funzionalità che la applicazione distribuita deve fornire in questo ultimo caso è tipicamente dato da un insieme di tipi diversi di informazioni, ciascuno dei quali deve essere trattato diversamente dagli altri e magari anche in più modalità tra loro differenti. In questo caso quindi, a partire da questo insieme, si deve mappare ogni tipo di informazione in un tipo di dato e dedicare a esso un topic specifico: chi genera informazioni di un certo tipo sarà un pubblicatore per quel topic, chi deve trattare secondo una certa modalità quelle informazioni sarà un sottoscrittore per quel topic. Indipendentemente dal tipo di flusso dati con cui si ha a che fare si ha che tipicamente nell'utilizzare l'MQTT si vanno ad implementare client MQTT, siano questi pubblicatori o sottoscrittori. Ci si potrebbe ora chiedere il server dove si collochi in questo discorso. Il server, ovvero il broker, ha compiti detti sopra: sono operazioni particolari, ma che non cambiano al cambiare del dominio applicativo in cui si lavora. Per questo motivo si tende ad utilizzare un broker già implementato da altri piuttosto che ad implementarne uno. Il lavoro che però è richiesto è quindi quello di configurare tale broker esistente, in particolare in termini di sicurezza.

Capitolo 3

Metodologie per la realizzazione di applicazioni distribuite

3.1 Scopo di questo capitolo

Si è detto in precedenza che, a partire dall'insieme di funzionalità richieste alla applicazione distribuita, si deve definire l'interfaccia applicativa e quindi la si deve implementare. Adesso estendiamo questo discorso dicendo che è possibile, a partire dall'insieme di funzionalità detto sopra, che una parte venga mappata verso un certo protocollo, un'altra verso un altro protocollo, un'altra ancora verso un terzo protocollo e così via: si ha quindi un'unica interfaccia applicativa in cui compaiono diversi protocolli. Ciò tipicamente accade o perché una funzionalità viene espressamente richiesta come da ottenersi mediante un certo protocollo oppure perché risulta più adatto un certo protocollo rispetto ad un altro per fornirla. Questo fatto fa si che la applicazione distribuita non poggi esclusivamente su un protocollo di un certo livello, ma sfrutti contemporaneamente protocolli diversi, potenzialmente anche di livelli diversi. L'estensione fatta non cambia quanto detto prima: l'interfaccia applicativa così ottenuta è da implementare lato server e da utilizzare lato client. Indichiamo ora con il termine "servizio" un'interfaccia applicativa e con l'espressione "fornitore del servizio" chi la implementa. Indichiamo quindi con l'espressione "fruitore del servizio" o con l'espressione "consumatore del servizio" chi si serve di tale interfaccia. Nel seguito, per brevità e per estensione del concetto, si utilizzerà il termine servizio per indicare congiuntamente una interfaccia applicativa e la sua implementazione.

Questo capitolo ha come obiettivo quello di trattare le diverse modalità che si

hanno a disposizione, partendo dall'insieme di funzionalità detto sopra, per realizzare un'applicazione distribuita che le fornisca, indipendentemente dai protocolli che si è scelto di utilizzare. Esistono principalmente due modalità: seguire l'approccio architetturale monolitico oppure seguire l'approccio architetturale a microservizi. Verranno trattate qui di seguito in tale ordine, infine si avrà modo di confrontarle evidenziando i vantaggi e gli svantaggi di ognuna.

3.2 Approccio architetturale monolitico

Seguire l'approccio monolitico vuol dire non fare niente di più e niente di meno di quanto detto finora: definire l'interfaccia applicativa che fornisca tutte le funzionalità richieste alla applicazione e implementarla. In questo modo l'architettura di ciò che ne risulta è costituita da un unico servizio, incaricato di ogni responsabilità. Questo approccio è storicamente quello arrivato per primo ed è il più adatto per applicazioni poco complesse.

3.3 Approccio architetturale a microservizi

3.3.1 Idea di base

Seguire l'approccio a microservizi vuol dire partire dall'insieme di funzionalità richieste alla applicazione, dividerlo in sottoinsiemi e quindi per ognuno di questi definire una sua propria interfaccia applicativa ed implementarla. In questo modo l'architettura che si ottiene è costituita da un insieme di servizi, ciascuno incaricato di una parte limitata delle funzionalità richieste applicazione, che collaborano tra loro per fornire l'insieme di funzionalità detto sopra. Questa modalità storicamente è quella arrivata dopo ed è indicata per applicazioni complesse.

Ora verranno analizzate le motivazioni per cui questo approccio è nato, dopodiché verrà definito il concetto di microservizio. Si proseguirà dando delle indicazioni su come realizzare una applicazione distribuita seguendo questa modalità, fornendo anche qualche considerazione utile a riguardo. Dopodiché si analizzerà la comunicazione tra un microservizio e il resto del mondo e si presenteranno le diverse modalità con cui dei microservizi possono collaborare tra loro. In conclusione si farà notare come il seguire i concetti introdotti da questa modalità permetta il raggiungimento degli obiettivi per cui è nata. Per una trattazione completa a riguardo si rimanda a [4] e a [5].

3.3.2 Motivazioni

Prima di addentrarsi nella trattazione dei concetti che questa modalità di costruzione di applicazioni distribuite introduce, è utile capire se valga la pena o no di iniziare questo percorso. Per questo motivo viene qui indicato ciò che sta dietro alla nascita di tale approccio. Secondo [5] l'approccio a microservizi è nato con lo scopo di bilanciare l'essere veloci nell'attuare modifiche ad un prodotto e il non fare danni irreparabili nel momento in cui le si attua, il tutto in un'ottica secondo la quale la domanda per tale prodotto può crescere oltre le aspettative iniziali. L'essere veloci nell'attuare modifiche si traduce nell'avere un prodotto in grado di evolvere adattandosi a ciò che ci sta intorno, che cambia col tempo. Il non fare danni nel momento in cui si attua una modifica vuol dire lavorare in un certo modo ed implica dunque delle tempistiche non trascurabili. La ragione per cui è nato l'approccio architetturale a microservizi è quindi quella di fornire una soluzione che permetta allo stesso tempo di raggiungere entrambi questi obiettivi dandogli eguale peso. Tutto ciò tenendo sempre presente che la domanda per il prodotto in questione può crescere, anche di molto, oltre le aspettative. Il modo in cui si risolve quest'ultima sfida, secondo questa modalità, non è quello di dimensionare il prodotto in base ad aspettative aumentate, ma è quello di dargli la capacità di crescere o decrescere a seconda della domanda corrente. Dopo aver introdotto i concetti fondamentali per questa modalità si indicherà come sia possibile, aderendo a tali concetti, ottenere gli obiettivi qui detti.

3.3.3 Definizione di microservizio

È quindi fondamentale comprendere cosa sia un microservizio. Un microservizio è un servizio piccolo, con il quale si può comunicare solo ed esclusivamente tramite la rete, che deve avere un suo specifico DataBase Management System (DBMS), che può essere sviluppato in modo indipendente e di cui può essere fatto il deploy in modo indipendente. Il fatto che sia piccolo più che nel numero di linee di codice o nel tempo impiegato per scriverlo deve essere misurato in termini di funzionalità che ha. Ovviamente, però, il fatto di dover avere poche funzionalità ha un impatto sul tempo necessario per scriverlo e sul numero di righe di codice di cui è fatto. Il motivo per cui si è scelto di imporre come unico meccanismo di comunicazione la rete è perché in questo modo si ottengono una serie di benefici che vanno nell'ottica dell'autonomia: il fatto di dover progettare qualcosa che sia accessibile dal mondo esterno solo tramite la rete forza a vedere questo qualcosa come una entità a se stante. In merito al fatto di richiedere che un microservizio abbia un DBMS ad uso esclusivamente proprio si potrebbe a primo impatto pensare ad una esagerazione non necessaria, tuttavia se si considera che di questo deve poter essere fatto lo sviluppo, la modifica e il deploy dovunque e ogni qual volta lo si desideri, il tutto in modo indipendente, allora si può notare che tale vincolo ha motivo di essere.

Poter sviluppare del codice in modo indipendente da fattori esterni è sicuramente una condizione desiderabile. Richiedere che che un microservizio possa essere sviluppato in modo indipendente vuol dire rendere tale condizione necessaria. Un aspetto rilevante in generale, ma ancor di più in questo contesto, è il velocizzare le tempistiche soprattuto della fase di mantenimento del software: il fatto di richiedere che di un microservizio debba poter essere fatto il deploy in modo indipendente va in questa direzione.

Date queste caratteristiche si indica qui come sia possibile ottenerle. Il fatto che un microservizio debba essere piccolo lo si ottiene dandogli poche responsabilità. La comunicazione tramite la rete e il fatto di avere un DBMS ad uso esclusivo sono vincoli a cui ci si deve semplicemente conformare. Per permettere di poter sviluppare un microservizio in modo indipendente bisogna attuare la fase di definizione dei microservizi nel modo opportuno, come verrà indicato in seguito. Infine per poter fare il deploy in modo indipendente oltre a fare in modo che si verifichino le proprietà dette sopra è necessario impacchettare e rilasciare ogni microservizio in un ambiente in cui risulti autonomo e isolato. Questo ambiente deve essere conforme alle aspettative del microservizio in termini di sistema operativo, stack di rete, librerie (della versione opportuna) etc. Per fare ciò una soluzione può essere quella dei container: un microservizio deve essere rilasciato per mezzo di un singolo ed esclusivamente suo container. Per una trattazione completa riguardo al mondo dei container e al loro utilizzo in problemi reali si rimanda a testi specifici a riguardo.

3.3.4 Realizzare un'applicazione distribuita seguendo l'approccio a microservizi

Realizzare un'applicazione distribuita seguendo questo approccio si traduce nel progettarla come costituita da un insieme di microservizi che collaborano tra loro per fornire le funzionalità a questa richieste. Si parte quindi dalla definizione dei microservizi e si conclude con la loro implementazione. Precedentemente si è detto che a partire dall'insieme di funzionalità lo si deve dividere in sottoinsiemi e per ognuno di questi definire un suo microservizio apposito: è proprio questo il modo in cui si devono definire i microservizi, ovvero a partire dalle funzionalità richieste alla applicazione. Come fare a operare tale suddivisione in sottoinsiemi è un quesito che non ha una risposta definita, quello che invece è definito è una linea guida da seguire nell'attuare tale suddivisione. Tale linea guida è anche nota come principio di singola responsabilità di Robert C. Martin, che afferma: "Metti assieme quelle cose che cambiano per la stessa ragione, separa quelle che cambiano per ragioni diverse". Da questa linea guida emergono i seguenti valori: debole accoppiamento tra microservizi diversi e alta coesione all'interno di uno stesso microservizio. Fare in modo che due microservizi siano debolmente accoppiati

vuol dire fare in modo che una modifica in uno dei due non implichi una modifica nell'altro. Se si fa riferimento a due microservizi tali che uno dei due è il fornitore di un servizio e l'altro è un consumatore di tale servizio (e fornitore a sua volta di un altro servizio) allora questo presuppone che l'interfaccia applicativa di un microservizio debba cambiare molto raramente, mentre la sua implementazione può variare come meglio si ritiene. Alta coesione all'interno di un microservizio implica che i concetti manipolati da questo siano tutti fortemente collegati e che non esistano altri concetti al di fuori di questi che risultino fortemente collegati a questi. In questo modo si ottiene che una modifica alla applicazione impatti esclusivamente in un punto: l'unico microservizio che tratta l'oggetto della modifica. Seguendo questo approccio la fase di definizione dei microservizi li vede solitamente delineati, all'interno del dominio di business in cui si opera, attorno ai sotto domini di questo. Una volta definiti i microservizi li si implementa andando a definire l'interfaccia applicativa di ognuno e quindi realizzandola.

3.3.5 Considerazioni sulla fase di realizzazione

Prima o poi, nel corso della progettazione di una soluzione a microservizi, si arriverà alla fase di definizione del database di ogni microservizio. Si vogliono qui fare alcune considerazioni a riguardo. Qualsiasi sia il dominio in cui la applicazione in questione operi capita che tutti i concetti da questo definiti siano tra loro legati. Alcuni saranno legati in modo forte, altri in modo debole: è proprio identificando questi legami più forti che si riescono a separare tra loro i sotto domini esistenti. Una volta fatta tale separazione, però, esistono ancora ed esisteranno sempre legami tra concetti di sotto domini differenti e sono proprio questi tipicamente a costituire il punto di interesse maggiore. In generale si è abituati a far corrispondere legami tra concetti a relazioni tra entità. Nel caso monolitico tutte le entità di interesse per la applicazione sono mappate su tabelle di un unico database: è possibile quindi implementare legami tra concetti di sotto domini diversi con relazioni tra le tabelle corrispondenti utilizzando il meccanismo della chiave esterna e affidandosi al DBMS per la gestione di tali relazioni. Nel caso a microservizi le entità di interesse della applicazione vengono sì mappate su tabelle, ma queste tabelle sono suddivise tra i database dei vari microservizi. Questo potrebbe far sorgere la domanda su come fare a legare tra loro tabelle in database diversi: si utilizza anche in questo caso il meccanismo della chiave esterna perché è il migliore nel collegare due dati (è bidirezionale, non dipende dall'implementazione etc), ma non ci si deve affidare separatamente ad ogni singolo DBMS e basta per la gestione di relazioni di questo tipo perché altrimenti si arriverebbe ad una situazione di inconsistenza per il sistema. Quello che si deve fare è dare questo compito di gestione alla applicazione, la quale se ne può occupare direttamente o affidarsi a soluzioni appositamente create per questo e altri tipi di problemi. Un caso particolare, però, può essere

costituito da un insieme di concetti a cui molti sotto domini diversi sono legati, tali per cui i dati di questi non variano frequentemente. In questa situazione è possibile duplicare tali dati per ogni microservizio in modo da evitare l'utilizzo della rete, però facendo attenzione a mantenere le copie tra loro consistenti.

3.3.6 Comunicazione tra un microservizio ed il resto del mondo

Definire l'interfaccia applicativa di un servizio, come indicato in precedenza in questo scritto, vuol dire scegliere, eventualmente definendoli, dei protocolli da sfruttare per fornire le funzionalità che sono richieste. In questo contesto, ovvero quello dei microservizi, si vogliono fornire delle indicazioni in merito a questa fase della progettazione. Risulta infatti utile notare che, al fine di ottenere i valori detti in precedenza, ci sono delle linee guida da seguire. Innanzitutto si deve cercare il più possibile di definire una interfaccia applicativa che non dipenda da specifici protocolli. Questo potrebbe sembrare contraddittorio poiché una interfaccia applicativa, ricordiamo, è esattamente corrispondente all'utilizzo che si deve fare di un insieme di protocolli. In realtà quello che si vuole suggerire è di seguire delle filosofie piuttosto che non seguirne nessuna o inventarne di proprie, quindi di scegliere dei protocolli che permettano di conformarsi ad esse. In questo modo, volendo, in futuro si potrà cambiare un protocollo scelto con un altro che permetta comunque di seguire la stessa filosofia di quello di partenza, il tutto senza impattare notevolmente sulla applicazione. Nella scelta di un protocollo bisogna poi controllare che questo permetta di non dover per forza condividere dettagli implementativi. È infatti molto importante non condividere dettagli interni ad un microservizio con il resto del mondo, questo sia per non dare modo a forme di accoppiamento forte di emergere sia per conformarsi alla buona pratica di programmazione nota come information hiding. Nel definire una interfaccia applicativa è bene porsi dal punto di vista di colui il quale dovrà poi utilizzarla e quindi fare il possibile per semplificargli il lavoro. Precedentemente si è detto che il fatto di avere due microservizi, un fornitore e un consumatore, debolmente accoppiati si traduce nel fatto di avere l'interfaccia applicativa in questione che cambia molto raramente. Qui si dice inoltre che, a fronte di cambiamenti minimi in tale interfaccia, il consumatore non deve rendersene conto. L'applicazione risulta quindi costituita dall'insieme dei microservizi così definiti, i quali collaborano per fornire nel complesso ciò che è richiesto alla applicazione. Prima di definire le interfacce applicative, in realtà, poiché ci troviamo ad operare nell'ambito dei microservizi, è necessario svolgere un passo fondamentale che condiziona la definizione di queste: scegliere il tipo di collaborazione.

3.3.7 Diverse modalità di collaborazione tra microservizi

Tipicamente la scelta è tra due alternative: orchestrazione o coreografia. Nel caso in cui si scelga la prima opzione capita che, data una funzionalità che richieda l'utilizzo di più microservizi, esista un microservizio corrispondente al cervello dell'elaborazione, il quale dice ad ogni altro microservizio coinvolto cosa deve fare. Se si sceglie la seconda possibilità si ha invece che, data una funzionalità che richieda l'utilizzo di più microservizi, ogni microservizio capisca in autonomia che cosa debba fare in relazione a tale funzionalità e quindi lo faccia. L'orchestrazione è più facile da progettare, ma va più verso l'accoppiamento tra microservizi. La coreografia è più difficile da progettare, ma permette un disaccoppiamento maggiore. Il primo caso si presta bene ad essere mappato verso un paradigma di comunicazione del tipo richiesta-risposta, nel secondo caso, invece, il paradigma ad eventi risulta il più appropriato. Nel paradigma ad eventi qualcuno si occupa di dichiarare che è capitato un evento e qualcun altro, potenzialmente anche più di uno, in relazione a ciò reagisce sapendo già come comportarsi. Adottando la modalità richiesta-risposta si hanno due strade ulteriori tra cui scegliere: comunicazione sincrona o asincrona. La comunicazione sincrona prevede l'invio di una richiesta e quindi il rimanere bloccati in attesa della risposta. È semplice da progettare, ma è meno performante in caso di operazioni molto lunghe. Nella comunicazione asincrona, invece, si manda una richiesta senza rimanere bloccati, segnandosi eventualmente come comportarsi alla ricezione della risposta. È più difficile da progettare ma, poiché non prevede attese, è più performante in caso di operazioni molto lunghe.

Scelto quindi il meccanismo di collaborazione per il fornire una data funzionalità lo si implementa andando a scegliere una filosofia che, seguendo tale meccanismo, permetta di realizzarlo. Tipicamente aver scelto il pattern richiesta-risposta porta all'adottare la filosofia REST con l'utilizzo del protocollo HTTP, questo perché il fatto di aderire al concetto HATEOAS al suo interno favorisce il disaccoppiamento tra fornitore e consumatore di un servizio. Scegliere invece il pattern ad eventi tipicamente si traduce nell'utilizzare un protocollo pensato per un flusso unidirezionale uno a molti di dati, come ad esempio l'MQTT, in cui chi genera il dato è chi deve dichiarare il verificarsi di un evento mentre chi lo processa è chi è interessato al verificarsi di tale evento e sa di suo di conseguenza come comportarsi.

3.3.8 Raggiungimento degli obiettivi iniziali

È utile ora notare che gli obiettivi per cui l'approccio a microservizi è nato vengono raggiunti grazie a tutti i vincoli e le caratteristiche sopra dette. Infatti, a fronte di un cambiamento da dover fare, capita tipicamente che questo sia localizzato in un unico microservizio: quello che gestisce l'oggetto di tale modifica. Questo microservizio, grazie alle sue caratteristiche, può essere modificato in modo indipendente dal resto del sistema e può essere messo in campo senza dover rimettere in campo l'intero

sistema, il che si traduce nella capacità di attuare modifiche velocemente. Il fatto di potersi comportare come descritto sopra favorisce un atteggiamento tale per cui, singolarmente per ogni cambiamento richiesto, si attui la corrispondente modifica e la si metta in campo. In questo modo viene raggiunta la capacità di non fare danni irreparabili nel momento in cui si attua una modifica: se si scoprissero problemi sarebbe facile tornare indietro o individuare la fonte di questi poiché c'è stata solo una modifica rispetto alla versione precedente funzionante.

Nell'approccio monolitico, invece, sia a causa del fatto che si deve per forza rimettere in campo l'intero sistema anche se è stata modificata solo una sua parte sia a causa del fatto che una modifica possa teoricamente impattare dappertutto, si tende a effettuare più modifiche contemporaneamente e quindi a mettere in campo il risultato di ciò. Questo porta a correre il rischio, a fronte di problemi, di non poter più tornare indietro facilmente o di dover impiegare del tempo per capire quale tra le modifiche fatte rispetto alla versione precedente sia la fonte di questi.

3.4 Confronto tra i due approcci

Si ritiene utile confrontare i due approcci precedentemente detti. Nella realtà capita, tipicamente, che l'insieme delle funzionalità richieste alla applicazione sia tale per cui lo si implementi con più di un protocollo. Nel caso in cui si usi l'approccio monolitico non si ha scelta: l'applicazione poggerà su più protocolli diversi, eventualmente anche di livelli diversi, contemporaneamente. Se invece si adotta l'approccio a microservizi, sebbene sia possibile che un microservizio debba fornire funzionalità differenti attraverso protocolli differenti, tipicamente capita che ogni microservizio si appoggi ad un singolo protocollo. In questo ultimo caso quindi si ha minore complessità. Uno degli aspetti più importanti in un sistema software è la sua capacità di essere resiliente oppure no, ovvero di sopportare fallimenti o errori di una o più sue parti senza rompersi del tutto. Nel caso monolitico risulta notevolmente complesso perseguire questo obiettivo, mentre invece nel caso a microservizi risulta più semplice. Si è precedentemente detto che il modo in cui una soluzione a microservizi affronta la sfida corrispondente ad avere una domanda che cresce oltre le aspettative corrisponde al dare al sistema la capacità di crescere e decrescere in funzione della domanda attuale. Questa capacità può averla anche un sistema monolitico, ma la differenza sta nel fatto che nell'approccio monolitico si è obbligati a scalare l'intero sistema in blocco, mentre nell'approccio a microservizi si può scalare ogni sua parte in modo indipendente dalle altre. Ad esempio, a fronte di una richiesta maggiore per una parte della applicazione, come ad esempio un singolo microservizio, lo si può spostare su una macchina host con più risorse, senza dover fare altrettanto per gli altri, oppure se ne può fare il deploy più volte, senza dover fare altrettanto per gli altri. Questo porta un vantaggio perché ottimizza

l'uso delle risorse disponibili. Il fatto che si possa fare il deploy di un singolo microservizio indipendentemente dal resto del sistema fa si che a fronte di una modifica già attuata sia più veloce la sua messa in campo: non c'è bisogno di rimettere in campo l'intero sistema, ma solo la parte di questo in cui è avvenuta tale modifica. Un aspetto chiave con cui solitamente si valuta una soluzione è dato dalla possibilità di riutilizzarla per risolvere altri problemi diversi da quello di partenza ma con delle caratteristiche in comune. Già a colpo d'occhio si può notare che l'approccio a microservizi, lavorando ad una granularità più fine rispetto a quello monolitico, permetta il riuso a livello di singole funzionalità piuttosto che dell'intera soluzione, che è più utile e più facile da attuare. Oltre a ciò l'approccio a microservizi permette di vedere una applicazione come un pezzo da costruire collegando tra loro pezzi già fatti e pezzi da fare. È esattamente il concetto che sta dietro alla filosofia Service-Oriented Architecture (SOA) di cui l'approccio a microservizi può essere considerato un esempio, il migliore, di implementazione.

Non è verosimile pensare che l'approccio a microservizi porti solo vantaggi rispetto a quello monolitico perché altrimenti lo avrebbe rimpiazzato invece che esserne un'alternativa. È infatti evidente che, a fronte di una soluzione richiesta, il sistema risultante da un progetto a microservizi sarà inevitabilmente più complesso del sistema risultante da un progetto monolitico. Va però sottolineato che, anche se il sistema a microservizi risultante è più complesso, la sua comprensione risulterà tipicamente più semplice rispetto al sistema monolitico in uno scenario di applicazione reale in cui la mole di codice è spropositata, grazie al fatto che nel caso a microservizi sia possibile procedere pezzo per pezzo. Inoltre l'approccio a microservizi introduce un problema che non si ha nell'approccio a monolitico: la gestione delle transazioni. Consideriamo di avere una sequenza di operazioni che necessita di essere atomica, quindi tale che non la si possa eseguire in parte ma si la debba eseguire del tutto o, eventualmente, non eseguire per niente. Nel caso monolitico questo problema si risolve facilmente indicando tale sequenza di operazioni come una transazione. Nel caso a microservizi, invece, se la sequenza di operazioni coinvolge più di un microservizio allora si devono adottare accortezze diverse. Una possibilità è quella di definire ed implementare dei meccanismi ad hoc per la nostra applicazione che hanno lo scopo di renderla classificabile come eventualmente consistente, ovvero tale per cui non è detto che ad un certo istante sia consistente ma è detto che sicuramente lo diventerà in futuro. Questa soluzione è adatta per problemi semplici. Tra questi meccanismi i più semplici sono quelli in cui si riprova più tardi ad eseguire un'operazione che è fallita. Un'altra possibilità è quella di definire ed implementare dei meccanismi ad hoc per renderla sempre consistente. Questa strategia si basa sull'implementazione e utilizzo di azioni compensative fatte su misura. Non è una strada consigliabile e questo perché non è mai bene in generale reinventarsi delle soluzioni a problemi comuni per cui ne esistono già di ottime. Infatti l'ultima possibilità è quella di adottare transazioni

distribuite servendosi di soluzioni già esistenti. Questa soluzione introduce una notevole complessità ulteriore al sistema per cui è indicata per la gestione di problemi complessi.

Capitolo 4

Introduzione al mondo IoT e all'utilizzo della tecnologia RFID

4.1 Scopo di questo capitolo

L'applicazione distribuita da creare ha come obiettivo l'automazione della logistica di un processo produttivo aziendale sfruttando la tecnologia RFID del mondo IoT secondo quanto fatto dalla applicazione monolitica. Per comprendere l'insieme di specifiche che tale applicazione deve soddisfare (che sono parte di quelle che la applicazione monolitica soddisfa) si ritiene fondamentale fornire una conoscenza basilare di tale tecnologia, scopo a cui questo capitolo è dedicato. Inizialmente verrà data una visione generale del mondo IoT, dopodiché si introdurrà la tecnologia RFID e il suo utilizzo. Per una trattazione completa sulle tematiche qui indicate si rimanda a [6] e a testi specifici sul tema RFID.

4.2 Introduzione al mondo IoT

Secondo [6] lo scenario in cui si colloca l'IoT è quello in cui un attore e una entità fisica interagiscono. L'attore è un essere umano, il quale ha un certo obiettivo. L'entità fisica è qualsiasi cosa si possa toccare, da un'automobile ad un animale o ad una lampadina, persino un'altro essere umano. Tramite l'interazione con l'entità fisica l'attore realizza il suo obiettivo. Un'interazione in generale può verificarsi con o senza un intermediario. Un caso particolare di intermediario è costituito dal mondo IoT. Tipicamente si sceglie un'interazione con il mondo IoT a fare da tramite poiché si vogliono ottenere i vantaggi che solo in questo modo

sono possibili. Tra questi, ad esempio, il dare la possibilità che l'attore e l'entità fisica siano fisicamente distanti. L'utilizzo dell'IoT come intermediario si basa sul fatto che l'entità fisica venga rappresentata nel mondo digitale associandole una entità virtuale. Esistono diverse tipologie di entità virtuale: una tupla in una tabella di un database relazionale, un account di un social network, un oggetto di una classe di un linguaggio orientato agli oggetti, etc. L'associazione di cui sopra si realizza dando ad ogni entità fisica un identificativo che deve essere quindi usato nella corrispondente entità virtuale. Il modo in cui si dà ad un'entità fisica un identificativo è per mezzo di un dispositivo, il quale deve esserle fisicamente attaccato o posto vicino. Questo dispositivo può essere una semplice etichetta (tag), la quale permette di identificare univocamente l'entità fisica su cui è posta, oppure un sensore, il quale permette sia di identificare un'entità fisica sia di ottenere informazioni in tempo reale sullo stato fisico di questa. L'identificazione avviene grazie al fatto che l'etichetta o il sensore sono in grado di memorizzare dati al loro interno: come minimo una stringa alfanumerica utilizzata come identificativo univoco. Un lettore (reader) è una calcolatore in grado di leggere dati da tale memoria e di scrivere dati su tale memoria. Infine, in uno scenario completo, compaiono anche gli attuatori, ovvero macchine in grado di attuare una modifica allo stato fisico dell'entità fisica su cui sono attaccati o vicino alla quale sono posti. In questo caso un lettore è anche in grado di comunicare con un attuatore al fine di comandarlo. Una volta che l'entità fisica è associata ad un'entità virtuale quello che capita è che l'essere umano e l'entità virtuale interagiscono tra loro e l'entità fisica e l'entità virtuale interagiscono tra loro: l'essere umano e l'entità fisica non interagiscono più tra loro. Una soluzione IoT permette tali interazioni con l'entità virtuale e mantiene sincronizzate la rappresentazione e la cosa rappresentata (ossia se cambia una cambia l'altra e viceversa): l'essere umano raggiunge il proprio obiettivo, ma indirettamente.

4.3 Tecnologia RFID

In questa trattazione non verranno considerati né i sensori né gli attuatori e verrà considerato esclusivamente il meccanismo di lettura da parte di un reader dei dati contenuti nella memoria di un tag. La tecnologia RFID è un esempio di implementazione di tale meccanismo. In questo contesto si ha quindi che un tag RFID è un'etichetta con un transponder e un chip che, se sottoposta ad un campo elettromagnetico, si attiva ed emette le informazioni che contiene. Una antenna RFID è un dispositivo che emette un campo elettromagnetico, legge le informazioni dei tag all'interno di tale campo e le comunica al reader RFID a cui è collegata (il collegamento avviene mediante un cavo). Un reader RFID è un dispositivo che controlla e gestisce un certo numero di antenne ed è un host ovvero un calcolatore.

4.4 Utilizzo della tecnologia RFID

Lo scenario in cui la tecnologia RFID trova applicazione è quello in cui un essere umano ha come obiettivo il sapere se una certa entità fisica si trova in un certo posto. Se non si ricorresse ad una soluzione IoT, allora capiterebbe che l'essere umano, per raggiungere il suo obiettivo, dovrebbe essere fisicamente nello stesso posto in cui si trova l'entità fisica. Tramite una soluzione IoT che sfrutti la tecnologia RFID, invece, è possibile che tale interazione avvenga a distanza, ovvero che l'essere umano sappia localizzare l'entità fisica senza essere fisicamente nello stesso posto di questa. Per ottenere ciò si procede nel seguente modo. Si attacca ad ogni entità fisica un tag RFID contenente un identificativo univoco. Si posiziona un reader RFID in in tale posto, collocando opportunamente una antenna RFID a questo collegata. Dopodiché si scrive del codice da eseguire sul reader in modo che, a fronte di una lettura di un tag, la comunichi ad un sistema software che modifichi di conseguenza la rappresentazione dell'entità fisica identificata, ovvero l'entità virtuale a questa collegata tramite l'identificativo letto. In questo modo si ha quindi che l'entità fisica interagisce con l'entità virtuale a lei corrispondente tramite la soluzione IoT. Dall'altro lato si ha che l'essere umano interagisce, sempre per mezzo di tale sistema software, con l'entità virtuale corrispondente alla entità fisica di interesse, raggiungendo quindi il proprio obiettivo. La soluzione IoT comprende quindi sia il software dei reader RFID sia il sistema software.

Capitolo 5

Funzionalità richieste all'applicazione

5.1 Scopo di questo capitolo

In questo capitolo si vogliono esporre al lettore le specifiche dell'applicazione a microservizi che si vuole creare, che corrispondono alla parte di specifiche della applicazione monolitica riguardanti la funzionalità di automazione della logistica. Poiché, come si vedrà più avanti, la gestione della logistica di un processo produttivo qualsiasi corrisponde ad un task svolto da un essere umano, l'applicazione da creare e (parte del) l'applicazione monolitica devono essere viste come uno strumento per migliorare il modo in cui tale essere umano svolge tale task, al punto tale da farlo al posto suo. Per permettere al lettore di comprendere ciò si è scelto di procedere nel seguente modo. Inizialmente si definirà il contesto in cui si inserisce tale task, ovvero quello costituto da un generico processo produttivo. Dopodiché si esporrà il task in questione considerando di non poter utilizzare una applicazione. In questo modo sarà possibile individuare più chiaramente e semplicemente le problematiche esistenti che l'applicazione monolitica risolve e che devono essere risolte dall'applicazione da creare. A questo punto si dirà come sono state risolte tali questioni dalla applicazione monolitica e quindi a cosa deve conformarsi l'applicazione a microservizi da creare. Infine si esporranno le motivazioni per cui si è scelto che le specifiche della applicazione a microservizi dovessero essere esattamente le stesse della parte a questa corrispondente dell'applicazione monolitica.

Si fa ora notare che, considerando le applicazioni legate all'automazione, la definizione di applicazione data nel secondo capitolo va estesa, dicendo che il modo migliore in cui un applicazione di tale tipo permette all'essere umano di fare un certo task è evitandogli di doverlo fare, perché è essa stessa che lo svolge al posto suo.

5.2 Definizione di processo produttivo

Un'azienda manifatturiera è caratterizzata dal fatto di avere un suo specifico processo di business. Tale azienda può avere diverse sedi (location). Una sede può avere una o più aree al suo interno e la stessa area può trovarsi replicata in sedi diverse. Il processo di business della azienda in questione tratta scatole (box). Una scatola è un contenitore che contiene un certo numero di pezzi di un certo tipo di prodotto. Di una scatola interessa sapere in particolare la terna di informazioni costituita dalla area e dalla location in cui questa si trova e dalla condizione in cui si trova in tale area in tale location. L'insieme di queste tre informazioni per una scatola verrà indicato come lo stato di tale scatola. All'interno di un'area all'interno di una location si trovano delle macchine aziendali. Le macchine aziendali sono i macchinari che servono all'azienda per attuare il suo processo di business, ovvero sono le macchine che trattano le scatole. Una macchina aziendale è caratterizzata dalla categoria di macchinari di cui fa parte e dalla location e area in cui si trova. Essa può produrre, consumare o modificare una scatola. Quando una scatola viene trattata da una macchina aziendale il suo stato cambia. La sequenza di stati che una scatola attraversa è detta "ciclo di vita" di tale scatola e corrisponde quindi ad una sequenza di trattamenti che riguardano tale scatola fatti su essa da varie macchine aziendali. Data una scatola e una macchina aziendale che la tratti si ha che lo stato futuro raggiunto dipende in primis dal tipo di tale scatola. L'informazione sul tipo è un'altra informazione caratteristica di una scatola che serve ad associarla ad uno specifico ciclo di vita. Esistono infatti diversi cicli di vita, ovvero diverse sequenze di stati possibili, e ad ognuno corrisponde un tipo di scatola. Il fatto che una scatola sia di un certo tipo fa si che la sequenza di stati che deve attraversare sia quella corrispondente a tale tipo. Una volta considerato il tipo di una scatola si è quindi a conoscenza del ciclo di vita che questa deve compiere. Tornando al punto in cui si ha una scatola e una macchina aziendale che la tratti, dato quindi il ciclo di vita di tale scatola, lo stato futuro raggiunto si determina considerando lo stato corrente, in modo da posizionarsi nel punto del ciclo di vita in cui la scatola si trovava prima di essere trattata, e poi la categoria, location e area della macchina aziendale che ora la tratta. La figura 5.1, qui di seguito, mostra graficamente un esempio di possibili diversi cicli di vita.

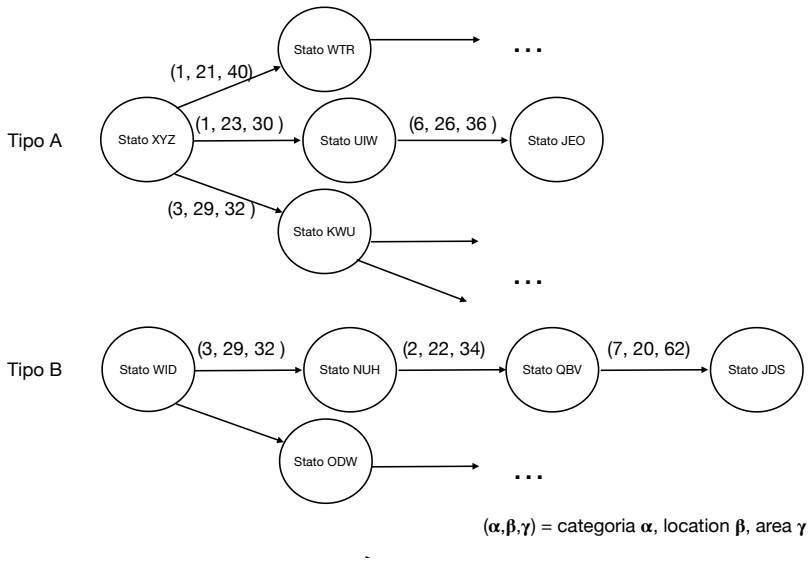


Figura 5.1: Cicli di vita scatole

5.3 Gestione della logistica

Per capire le specifiche della applicazione da creare (che sono parte di quelle soddisfatte dall'applicazione monolitica) immaginiamo ora di dover gestire la logistica di tale processo, ovvero di dover mantenere (aggiornate) le informazioni sulle scatole, e di non poter utilizzare la tecnologia per fare ciò. C'è bisogno di avere un documento cartaceo associato ad ogni scatola su cui riportare e tenere aggiornate le informazioni di tale scatola. Consideriamo in primis il fatto che si debba avere un documento cartaceo associato ad ogni scatola. Per fare ciò è necessario che ogni scatola sia univocamente identificata all'interno dell'azienda: stabiliamo che ogni scatola debba avere un identificativo univoco all'interno dell'azienda. Questo identificativo decidiamo di scriverglielo sopra e di scriverlo su quello che deve essere il documento ad essa associato: il problema sopra detto è risolto. Consideriamo quindi ora il fatto che si debbano riportare e mantenere aggiornate le informazioni sulla scatola in tale documento ad essa associato. Per fare ciò c'è bisogno di capire quali sono le informazioni che possono cambiare di una scatola e quando possono cambiare. Le informazioni soggette a cambiamento sono quelle che costituiscono lo stato di una scatola: location, area e condizione. Lo stato di una scatola cambia quando questa è trattata da una macchina aziendale, ovvero quando si trova fisicamente nei pressi di una macchina aziendale. Decidiamo quindi che quando una scatola si trova nei pressi di una macchina aziendale un impiegato ha il compito di aggiornarne lo stato. Un impiegato deve quindi fisicamente trovarsi nei pressi di una scatola quando questa è trattata da una macchina aziendale e deve aggiornare il documento cartaceo a questa associato scrivendoci il nuovo stato da essa ora raggiunto. Lo stato futuro di una scatola, ricordiamo, dipende dal tipo e dallo stato corrente di tale scatola e dalla location area e categoria della macchina aziendale che la tratta. Le informazioni sul tipo e sullo stato corrente della scatola l'impiegato le trova nel documento cartaceo a questa associato. Le informazioni sulla macchina aziendale potrebbe saperle di suo, ma è inutile e potenzialmente fonte di errori richiedere che ogni impiegato sappia le informazioni sulle macchine aziendali. Decidiamo quindi di avere un documento cartaceo associato ad ogni macchina aziendale su cui riportare le informazioni di tale macchina aziendale. È un problema analogo a quello già risolto sulle scatole quindi si sa già come comportarsi: decidiamo di dare ad ogni macchina aziendale un identificativo univoco all'interno dell'azienda (scrivendoglielo sopra) e di usarlo in un documento cartaceo per associare tale documento a tale macchina aziendale. Fatto ciò, tornando all'aggiornamento dello stato di una scatola, si ha quindi che l'impiegato trova le informazioni sulla macchina aziendale nel documento cartaceo a questa associato.

5.4 Automazione della logistica: specifiche dell'applicazione da creare

Nello scenario sopra descritto si ha quindi un essere umano che ha un obiettivo, ovvero quello di mantenere aggiornate le informazioni sulle scatole, e che per realizzarlo svolge un certo task. Questo task consiste nell'essere fisicamente nei pressi di una scatola quando questa viene trattata da una macchina aziendale, recuperare i documenti cartacei sulla scatola e sulla macchina in questione, calcolare (dalle informazioni a loro interno) il nuovo stato raggiunto dalla scatola e quindi scriverlo sul documento cartaceo ad essa associato.

Consideriamo ora di voler automatizzare la gestione della logistica sopra descritta. Questo si traduce nel migliorare, rendendolo automatizzato, il task a questa corrispondente. È stato ideato un modo per realizzare ciò che è il seguente. Si utilizza la tecnologia RFID come segue. Si posiziona almeno una antenna RFID nei pressi di ogni macchina aziendale, in questo modo ad ogni macchina aziendale è associata almeno una antenna ad uso suo esclusivo. Si collega ogni antenna ad un reader RFID (più antenne sono collegate allo stesso reader). Si attacca ad ogni scatola un tag RFID: si stabilisce che l'identificativo della scatola è quello contenuto nel tag. In questo modo il fatto che una antenna legga un tag si traduce nell'informazione che la macchina aziendale associata a tale antenna ha ora vicino a lei la scatola associata a tale tag, ovvero che è necessario aggiornare lo stato di quella scatola considerando che è stata trattata da quella macchina aziendale. Si dà ad ogni reader un indirizzo ip facendo in modo che non ci siano sovrapposizioni all'interno della stessa area della stessa location (ma invece sono possibili tra aree diverse della stessa location o nella stessa area in location diverse o in aree diverse in location diverse). Si fa in modo che una antenna RFID sia caratterizzata dalla location e area in cui si trova, dall'indirizzo ip del reader RFID a cui è collegata e dal numero di porta di questo a cui è collegata. Il numero di porta non riguarda il livello OSI trasporto, ma è semplicemente un numero che permette di identificare una antenna tra quelle collegate ad uno stesso reader. Si definisce un sistema software composto da due parti: del codice da eseguire sui reader e una applicazione distribuita. Il codice da eseguire sui reader deve essere tale per cui ognuno di essi, nel momento in cui una delle antenne a lui collegate legge un tag, pubblichi un messaggio MQTT su un broker in cui specifica il suo indirizzo ip, il numero di porta corrispondente all'antenna che ha eseguito la lettura e l'identificativo contenuto nel tag letto (e volendo altre informazioni). Si definisce un topic per ogni terna di informazioni costituita da categoria, location e area delle macchine aziendali. Un reader deve pubblicare sul topic opportuno sulla base della macchina aziendale associata all'antenna a lui collegata che ha letto il tag. L'applicazione distribuita, vista come una black box,

deve essere tale per cui è in sottoscrizione su i vari topic e offre le funzionalità di processamento di un messaggio MQTT, di consultazione delle informazioni sulle scatole e delle informazioni sull'esito del processamento dei messaggi tramite HTTP e di consultazione e modifica delle informazioni sui device, ovvero sulle macchine aziendali e sulle antenne RFID, sempre tramite HTTP. L'elaborazione che deve fare nel momento in cui riceve un messaggio MQTT è la seguente: deve recuperare i dati relativi alla macchina aziendale associata all'antenna di cui riceve indirizzo ip e numero di porta del reader e i dati relativi alla scatola in questione, quindi calcolare lo stato futuro di tale scatola e conseguentemente aggiornarne i dati relativi. La figura **??** esprime graficamente il tutto.

Il modo per automatizzare la logistica di un processo produttivo qualsiasi detto sopra è stato quindi realizzato nel concreto per uno specifico processo produttivo aziendale (i cicli di vita rappresentati dalla figura 5.1 sono una parte dei cicli di vita di tale processo). Nella realizzazione si è commesso l'errore di adottare l'approccio sbagliato per l'implementazione dell'applicazione distribuita: è proprio questa l'applicazione monolitica da cui il lavoro di questa tesi è partito. Col tempo è stato deciso che tale applicazione monolitica dovesse fornire altre funzionalità utili al supporto all'automazione del processo di business specifico del caso, le quali si basano sulla funzionalità di automazione della logistica e impattano su questa. L'obiettivo di questa tesi pertanto è quello di concepire e realizzare una soluzione a microservizi che possa svolgere unicamente la funzionalità di automazione della logistica e che sia conforme alle specifiche sopra descritte.

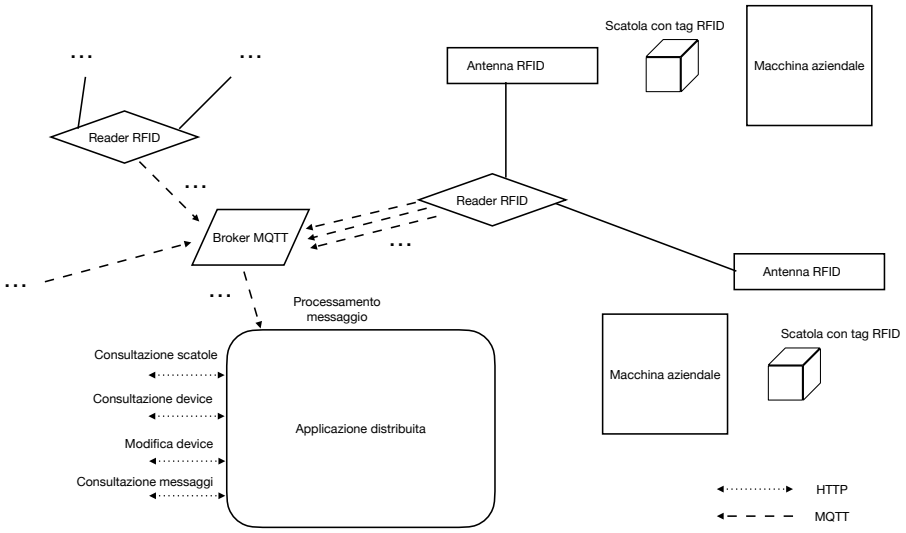


Figura 5.2: Architettura del sistema

5.5 Motivazioni a supporto delle specifiche

Analizzando quanto precedentemente esposto emergono le considerazioni seguenti. Innanzitutto il modo ideato con cui usare la tecnologia RFID per risolvere il problema dell'automazione della logistica, o più in generale il modo ideato con cui risolvere l'automazione della logistica, non ha richiesto di porre limiti o vincoli sul processo produttivo in questione, quindi può essere applicato ad un qualsiasi processo di tale tipo. Dopodiché si ha che si è aderito al modo corretto con cui utilizzare la tecnologia RFID, ovvero quello precedentemente spiegato secondo cui si devono avere il codice dei reader da un lato e il codice di una applicazione distribuita dall'altro. Si fa qui notare che i motivi per cui è utile conformarsi a ciò sono che così la manipolazione delle entità virtuali, che avviene nell'applicazione distribuita, è centralizzata, quindi meno complessa da gestire rispetto al caso distribuito che si avrebbe se avvenisse in ogni reader e che in questo modo si permette ad ogni reader di non dover fare elaborazioni complesse, che sarebbero problematiche visto che è un dispositivo da campo IoT (quindi con poca memoria, poca CPU e tipicamente poca banda). Avendo quindi tale separazione è possibile adottare in ogni parte accorgimenti diversi e appropriati per la funzionalità che svolge. In questo caso la scelta dell'MQTT per la comunicazione tra i reader e l'applicazione distribuita è ottima sia perché esso è molto leggero sia perché, volendo evitare di processare i dati sul campo (cioè dove vengono generati) e non necessitando di ricevere una risposta sul processamento di questi, esso è ottimo per gestire quello che a tutti gli effetti è un flusso unidirezionale di dati uno a (eventualmente) molti. La scelta dell'HTTP per la comunicazione tra l'applicazione distribuita e il resto del mondo, ovvero chiunque voglia consultare le informazioni sulle entità virtuali, è ottima perché oggi giorno chiunque ha un client HTTP su qualsiasi dispositivo e perché il suo utilizzo, anche se non ottimo in termini di performance, è estremamente facile a causa del fatto che è un protocollo testuale. Infine si ha che la scelta fatta per la configurazione dei topic è ottima perché questa è quella che bilancia meglio la complessità del codice dei reader (per cui l'ottimo sarebbe tutti che pubblicano su un solo topic) e la complessità del codice del sistema software (in cui l'ottimo sarebbe avere topic diversi per ogni insieme di informazioni costituito da categoria, location e area di una macchina aziendale e un tipo di scatola). Tali considerazioni sono i motivi per cui si è voluto migrare tale funzionalità dal mondo monolitico a quello a microservizi piuttosto che definirla da zero. Esistendo già il codice dei reader ed essendo già configurato il broker MQTT il lavoro di questa tesi è consistito esclusivamente nel concepimento e nella realizzazione di una applicazione distribuita a microservizi che aderisse alle specifiche sopra dette. Si fa qui notare che il meccanismo con cui utilizzare la tecnologia RFID sopra definito è tale per cui durante il periodo in cui la scatola resta nel campo di una antenna (e quindi il suo tag viene continuamente letto) vengano continuamente pubblicati messaggi

MQTT relativi alla lettura del suo tag. Ciò implica che una volta che lo stato di una scatola è stato aggiornato opportunamente una successiva la lettura del suo tag da parte della stessa antenna non debba avere effetti su tale stato. Il fatto detto sopra deve essere tenuto in considerazione perché fa capire che la mole di messaggi ricevuti dal sistema software con il passare del tempo diventa enorme e che pertanto c'è bisogno di adottare dei meccanismi che ne rendano l'elaborazione il più veloce possibile.

Capitolo 6

Analisi applicazione monolitica

6.1 Scopo di questo capitolo

Lo scopo di questo capitolo è quello di analizzare in modo critico e qualitativo la soluzione monolitica costituente il punto di partenza del lavoro di tesi qui descritto, esclusivamente in merito al compito di automazione della logistica e alle funzionalità corrispondenti ad esso. Inizialmente verranno esposti i problemi riscontrati dovuti al fatto che si è seguito l'approccio architetturale sbagliato, dopodiché verranno trattati i limiti emersi dovuti ad altri fattori. Infine verranno esposti i punti di forza di tale soluzione.

6.2 Limiti dovuti alla scelta errata dell'approccio architetturale

Tra le problematiche dovute al fatto di aver scelto l'approccio monolitico compare, ad esempio, il fatto di dover per forza avere una applicazione che poggi contemporaneamente su più protocolli diversi (per venire incontro alle specifiche precedentemente definite), il che implica una maggiore complessità. Un altro aspetto fondamentale da analizzare è la capacità di crescere o decrescere della applicazione sulla base della richiesta attuale (senza ovviamente modificarne il codice). Se la richiesta attuale salisse di molto per una certa funzionalità allora sarebbe necessario scalare l'intera applicazione, non essendo possibile operare ad una granularità più fine quale quella delle singole funzionalità. Questo risulterebbe in costi aggiuntivi necessari maggiori rispetto al minimo teoricamente possibile. Se la richiesta attuale per una certa funzionalità scendesse di molto, invece, non

si potrebbe fare niente di conseguenza perché altrimenti si ridurrebbero le risorse anche alle altre funzionalità. Il che corrisponderebbe quindi ad avere un utilizzo delle risorse non ottimizzato. La applicazione qui descritta è eseguita su un solo calcolatore, che pertanto costituisce un Single Point Of Failure (SPOF) per il sistema: se questo va giù, allora l'intera applicazione va giù. Consideriamo ora la fase di mantenimento del sistema software qui analizzato. Il fatto che tale soluzione sia costituita da un unico pezzo enorme (e complesso come già sottolineato) fa si che il modificarla sia problematico perché potenzialmente una modifica potrebbe impattare dappertutto. Questo fa si che si tenda ad evitare di modificarla o che se la si modifica si facciano più modifiche assieme, il che potrebbe facilmente portare a fare danni irreparabili. Tale applicazione pertanto si rivela incapace di stare al passo con ciò che la circonda, che via via cambia. A seguito di una modifica già effettuata non è possibile rimettere in campo solo le parti del sistema che sono state toccate da tale modifica, ma è necessario rimettere in campo l'intero sistema in blocco, con conseguente periodo di non disponibilità elevato. Si consideri ora il seguente aspetto. Quando si ha a che fare con una applicazione che deve essere utilizzata in un contesto aziendale non è verosimile pensare che questa elabori completamente un input prima di dedicarsi al successivo, perché altrimenti sarebbe di una lentezza troppo elevata per poter effettivamente essere messa in campo. Se l'applicazione in questione è di tipo monolitico l'unica alternativa al meccanismo sopra detto è la concorrenza, in cui gli input condividono sia l'uso delle risorse del calcolatore su cui è eseguita l'applicazione sia l'uso dell'unico database di questa e vengono elaborati un po' alla volta. Questo però fa si che anche due input che non richiedono l'uso della stessa porzione del database impattino l'uno sull'altro, se non altro per l'utilizzo delle risorse del calcolatore. Se l'applicazione in questione è a microservizi , invece, si ha automaticamente parallelismo tra microservizi diversi per un certo tipo di input. Infatti se un input richiede un solo microservizio e un'altro input richiede un solo altro microservizio (e se questi, come accade tipicamente, sono eseguiti su calcolatori diversi) allora l'uno non impatta in alcun modo sull'altro. Considerando poi input che richiedono lo stesso microservizio si deve fare uso della concorrenza all'interno di questo per aumentarne l'efficienza. Se si volesse adattare tale soluzione per utilizzarla in un contesto diverso da quello in essere ma comunque in cui i problemi da risolvere sono gli stessi si avrebbero grosse difficoltà nel fare ciò, secondo quanto detto precedentemente. Tale applicazione pertanto risulta fine a se stessa. Infine una tematica piuttosto rilevante nell'analisi di una soluzione software è la sua capacità di poterne riutilizzare esclusivamente delle funzionalità offerte per risolvere problemi diversi, che le richiedono. Riutilizzare una soluzione software qui è inteso come il servirsene così com'è senza doverne toccare il codice. L'applicazione monolitica qui trattata non possiede questa capacità: essendo un blocco unico, enorme e indivisibile non permette di riutilizzarne esclusivamente singole funzionalità.

6.3 Altri limiti

L'analisi effettuata ha evidenziato un altro limite rilevante che si sarebbe potuto evitare pur seguendo l'approccio monolitico: lo si espone qui di seguito. Esso corrisponde al fatto che la soluzione qui analizzata è stata realizzata, in merito alle funzionalità di consultazione e modifica, servendosi del protocollo HTTP come specificato nei requisiti, ma non utilizzandolo nel modo opportuno. È infatti emerso che in tale applicazione monolitica non si sia seguita la filosofia REST nell'utilizzare tale protocollo. Se si volesse utilizzare il modello di maturità di Richardson allora si troverebbe che tale applicazione è classificabile come di livello uno. Questo fa si che l'interfaccia applicativa di questa non sia auto descrittiva, risultando quindi di difficile comprensione per qualsiasi utilizzatore del servizio, ma non solo. Supponiamo, infatti, che in futuro sia necessario cambiare il protocollo attraverso cui fornire queste funzionalità. Se si fosse seguita la filosofia REST nell'utilizzare l'HTTP, allora, a fronte di tale cambiamento richiesto, basterebbe comportarsi con questa applicazione nello stesso modo in cui ci si comporterebbe con una qualsiasi altra applicazione conforme a REST ed utilizzante l'HTTP con lo stesso problema. Visto che non si è seguita tale filosofia, nel caso precedentemente definito, risulterebbe invece necessario un lavoro su tale applicazione fatto su misura. Questo si traduce in termini pratici in tempistiche, costi e conoscenze necessarie maggiori rispetto al caso aderente a REST, costituendo di fatto un problema non di poco conto. Essendo classificata come di livello uno l'applicazione qui analizzata non segue il principio HATEOAS. Questo fortunatamente non si è tradotto in una gestione lato server dello stato della conversazione con un client, ma ha comunque portato al problema per cui un client deve conoscere a priori le risorse esistenti e i loro URI e quindi come usare il metodo POST su queste per ottenere le varie funzionalità per lui di interesse. Tale fatto fa si che, in seguito a cambiamenti sul server anche banali come una modifica degli URI, sia necessario modificare di conseguenza i client già esistenti. Questo è un problema che si cerca solitamente di evitare a tutti i costi.

6.4 Punti di forza

L'applicazione analizzata si è dimostrata resiliente rispetto ad una problematica comune nell'ambito RFID: il fatto che nonostante un tag sia nei pressi di una antenna questo possa non venir letto. Se ciò accade allora non viene generato alcun messaggio MQTT, il che corrisponde ad avere la scatola reale in questione che cambia stato ma la scatola virtuale a lei associata no. Le due entità quindi non sono più sincronizzate e, se non si corregge tale errore, non lo saranno mai più. Infatti se questa scatola dovesse essere letta successivamente da un'altra antenna

collegata ad un'altra macchina aziendale che giustamente ha trattato tale scatola, allora l'applicazione, senza opportune contromisure, riconoscerebbe come errata quella lettura perché rispetto allo stato corrente della scatola virtuale e al suo tipo non esistono archi verso altri stati corrispondenti alla categoria, location e area della macchina aziendale associata all'antenna che ha fatto tale lettura. Il modo in cui è stato risolto tale problema è il seguente. Si testa se lo stato corrente, errato rispetto alla lettura corrente, è precedente rispetto a quello che dovrebbe essere lo stato corretto per la lettura corrente, considerando il tipo della scuola in questione. Se così è allora si esegue prima del codice che rimetta le cose a posto e dopo il codice conseguente alla lettura corrente.

Capitolo 7

Design dell'applicazione a microservizi

7.1 Scopo di questo capitolo

In questo capitolo si esporrà il cuore del lavoro di questa tesi: il design del sistema software realizzato seguendo l'approccio a microservizi. Inizialmente verrà indicata l'architettura scelta, ovvero verranno indicati i microservizi che la costituiscono. In seguito verrà trattato singolarmente ogni suo componente. In tale trattazione sarà possibile riconoscere i meccanismi con cui la soluzione proposta realizza le varie funzionalità richieste. Infine si faranno alcune considerazioni in merito all'architettura proposta.

7.2 Design dell'architettura a microservizi

La figura 7.1 esprime il risultato della fase di design della architettura a microservizi.

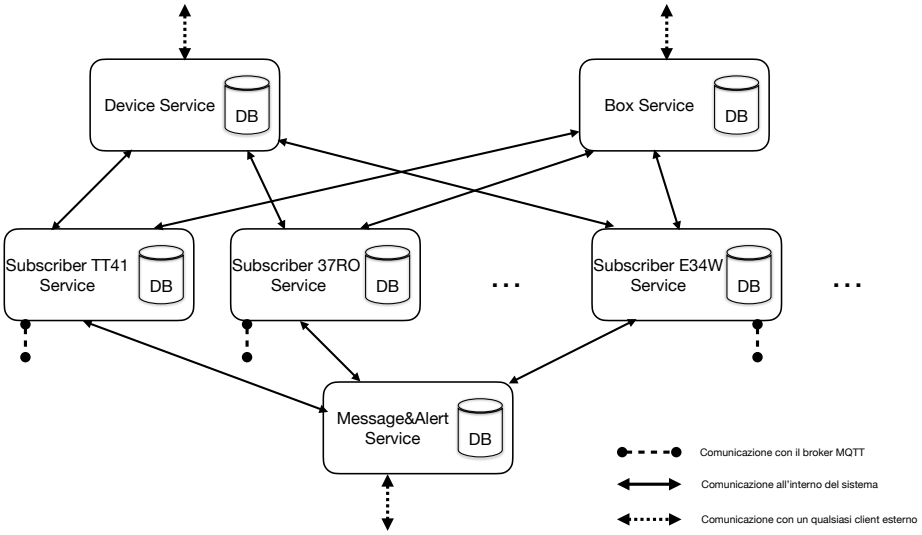


Figura 7.1: Architettura a microservizi

Il device service è il microservizio che si occupa di memorizzare le informazioni sulle macchine aziendali e sulle antenne RFID e di renderle manipolabili. Il box service, invece, è il microservizio che si occupa di memorizzare le informazioni sulle scatole e sullo storico dei cambiamenti di stato di ognuna di esse e di renderle consultabili. Il message&alert service è il microservizio che si occupa di memorizzare e rendere consultabili le informazioni sull'esito del processamento dei messaggi MQTT generati sul campo. La configurazione dei topic è, si ricorda, tale per cui esiste un topic per ogni categoria, location e area di macchine aziendali. Un subscriber service è un microservizio in sottoscrizione su un topic. Esso è l'unico e il solo in sottoscrizione su quel topic e ha il compito di processare solo ed esclusivamente ogni messaggio pubblicato su tale topic e di mandarne l'esito al message&alert service. Processare un messaggio si è detto che corrisponde a recuperare i dati relativi alla macchina aziendale associata all'antenna di cui si riceve indirizzo ip e numero di porta del reader collegato e i dati relativi alla scatola di cui si riceve l'identificativo, quindi calcolare lo stato futuro di tale scatola e conseguentemente aggiornarne i dati relativi. È evidente che recuperare i dati sulle scatole e aggiornarli siano operazioni che un subscriber fa collaborando con il box service, il quale quindi deve anche rendere le informazioni sulle scatole manipolabili. È altrettanto evidente che il recupero delle informazioni sulla macchina aziendale in questione venga fatto da tale subscriber collaborando con il device service. Si ha quindi che sulla base della categoria, location e area della macchina aziendale associata all'antenna da cui arriva un messaggio capita che l'elaborazione di questo è affidata ad un subscriber piuttosto che ad un altro. Si sfrutta ciò quindi per evitare che ogni subscriber abbia lo stesso comportamento (ovvero di avere più istanze dello stesso subscriber), cosa che non andrebbe bene perché il processamento di un messaggio è enorme quindi tale microservizio non sarebbe né piccolo né semplice, comportandosi come segue. Per ogni tipo di scatola che deve essere trattato da una macchina aziendale con location, area e categoria pari a quelle del topic sui cui un subscriber è in sottoscrizione capita che tale subscriber è il solo ed unico in grado di far passare una scatola di tale tipo da un certo stato ad un certo altro stato. Questa coppia di stati (partenza e arrivo) è l'unica per quel tipo di scatola per cui è necessario il trattamento da parte di una macchina aziendale con le caratteristiche corrispondenti al topic su cui il subscriber è in sottoscrizione. Ogni subscriber quindi ha lo stesso scheletro di base in termini di comportamento (corrispondente alla sequenza di passi da svolgere per processare un messaggio), il quale viene poi specializzato in modo diverso per ognuno di essi a seconda della porzione dei messaggi che deve processare. La figura 7.2, in cui si fa riferimento al contenuto della figura 5.1, esprime graficamente l'associazione tra un subscriber e un topic cosa comporti in termini di tipi di scatole trattati da quel subscriber. Quindi, secondo questo meccanismo, una scatola con un certo tipo deve essere trattata dai subscriber che si occupano di tale tipo nell'ordine definito da tale ciclo di vita. La

figura 7.3 corrisponde alla figura 5.1 in cui si è fatto uso di quanto espresso nella figura 7.2 ed esprime graficamente il tutto.

Identificativo Subscriber	Campo di applicazione	Tipi di scatole trattati
E34W	(3, 29, 32)	Tipo A, Tipo B
RT53	(1, 23, 30)	Tipo A
37RO	(6, 26, 36)	Tipo A
901A	(1, 21, 40)	Tipo A
TT41	(2, 22, 34)	Tipo B
U739	(7, 20, 62)	Tipo B

(α,β,γ) = categoria α, location β, area γ

Figura 7.2: Associazione subscriber-topic

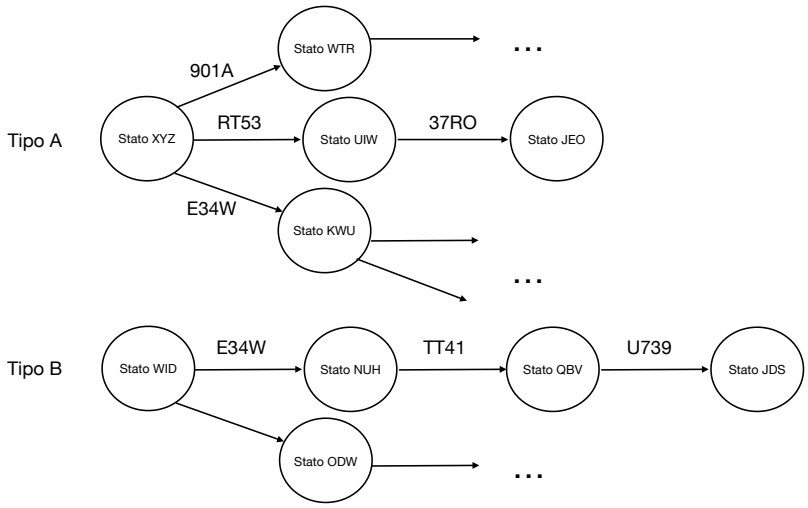


Figura 7.3: Cicli di vita scatole con subscriber

Nella figura 7.4 si riprende la figura 5.2 inserendovi quanto espresso nella figura 7.1.

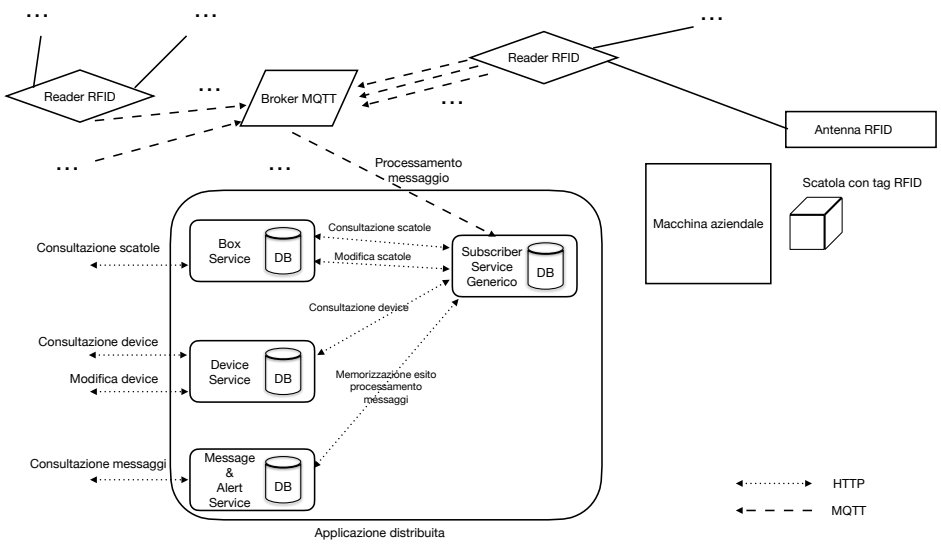


Figura 7.4: Architettura del sistema con applicazione a microservizi

7.3 Device service

7.3.1 Introduzione al microservizio

Il device service è il componente che ha il compito di gestire i device dell'azienda. Il termine device è utilizzato per indicare due tipologie di dispositivi: le antenne RFID e le macchine aziendali. Una antenna RFID è univocamente identificata dal suo id ed è caratterizzata dallo stato ok/ko, dalla location e area in cui si trova, dall'indirizzo ip del reader RFID a cui è collegata e dal numero di porta del reader a cui è collegata. Una macchina aziendale, invece, è univocamente identificata dal suo id ed è caratterizzata dallo stato ok/ko, dalla categoria di macchinari di cui fa parte e dalla location e area in cui si trova. Una macchina aziendale è associata ad una o più antenne. Lo stato di una macchina aziendale viene calcolato a partire dallo stato delle antenne a questa associate: se almeno una antenna associata è ok allora la macchina aziendale è ok. Si è scelta una politica che permetta l'associazione tra una macchina aziendale ed una antenna solo ed esclusivamente se queste si trovano nella stessa area e nella stessa location. Il device service offre quindi le funzionalità atte a gestire lo scenario appena descritto, dalla creazione e aggiornamento di macchine aziendali o di antenne, la creazione e distruzione di associazioni tra loro, fino alle funzionalità di interrogazione e ricerca. Per realizzare le funzionalità sopra dette si è scelto di utilizzare l'HTTP (come da specifiche) aderendo alla filosofia REST in un modo che è classificabile come di livello 3 secondo il modello di maturità di Richardson. Di seguito verrà quindi dapprima trattato il design delle risorse di questo microservizio, dopodiché la definizione della sua interfaccia applicativa e infine la progettazione del suo database.

7.3.2 Design delle risorse

Di seguito si propone in forma tabellare il frutto della fase di design delle risorse e quindi l'insieme dei tipi di dato da scambiare con questo microservizio.

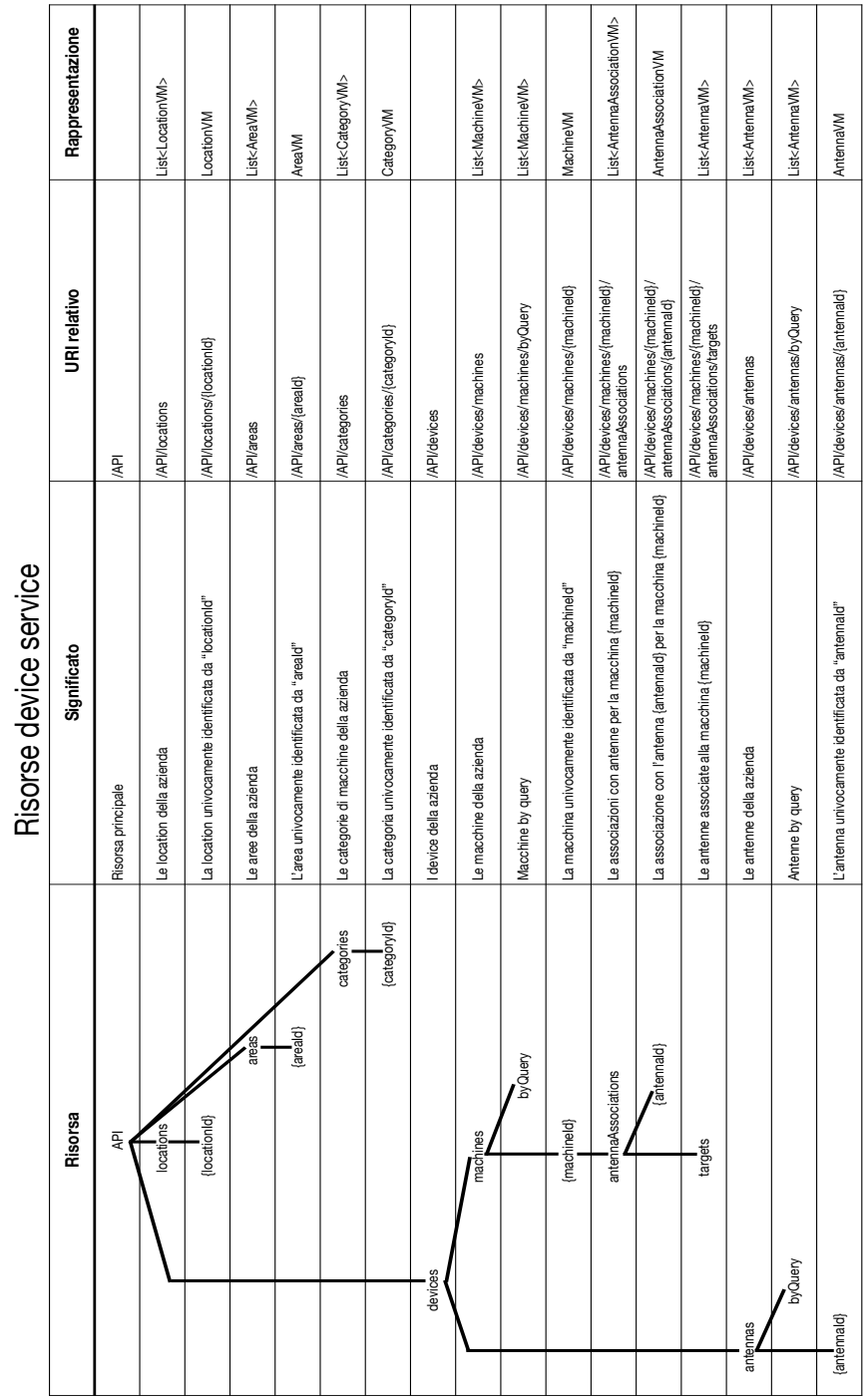


Figura 7.5: Risorse device service

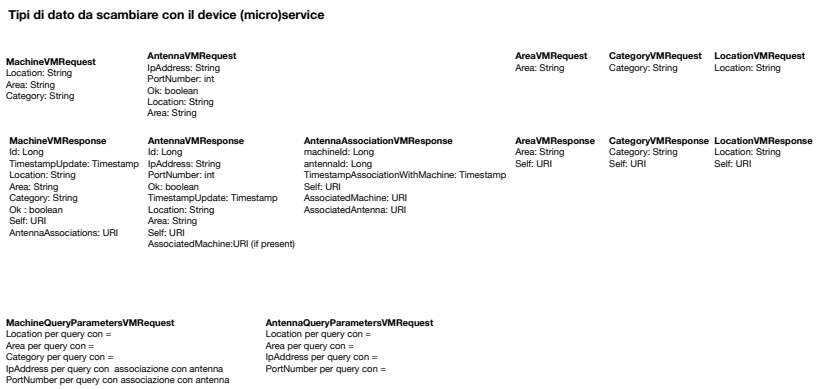


Figura 7.6: Tipi di dato device service

7.3.3 Definizione dell'interfaccia applicativa

Qui si espone l'interfaccia applicativa definita in modo tale da conformarsi alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Si riportano solo i casi di successo.

Interfaccia applicativa device service (parte 1 di 2)

Operazione	Risorsa	Verbo	Corpo Richiesta	Query Parameters	Success status	Corpo Risposta
Crea location	/API/locations	POST	LocationVMRequest		201 CREATED	LocationVMResponse
Dammi la location dato il suo identificativo	/API/locations/{locationId}	GET			200 OK	LocationVMResponse
Aggiorna location dato il suo identificativo	/API/locations/{locationId}	PUT	LocationVMRequest		204 NO CONTENT	
Crea area	/API/areas	POST	AreaVMRequest		201 CREATED	AreaVMResponse
Dammi l'area dato il suo identificativo	/API/areas/{areaId}	GET			200 OK	AreaVMResponse
Aggiorna area dato il suo identificativo	/API/areas/{areaId}	PUT	AreaVMRequest		204 NO CONTENT	
Crea categoria	/API/categories	POST	CategoryVMRequest		201 CREATED	CategoryVMResponse
Dammi la categoria dato il suo identificativo	/API/categories/{categoryId}	GET			200 OK	CategoryVMResponse
Aggiorna categoria dato il suo identificativo	/API/categories/{categoryId}	PUT	CategoryVMRequest		204 NO CONTENT	
Dammi le location	/API/locations	GET			200 OK	List<LocationVMResponse>
Dammi le aree	/API/areas	GET			200 OK	List<AreaVMResponse>
Dammi le categorie	/API/categories	GET			200 OK	List<CategoryVMResponse>
Crea antenna	/API/devices/antennas	POST	AntennaVMRequest		201 CREATED	AntennaVMResponse
Dammi l'antenna dato il suo identificativo	/API/devices/antennas/{antennaId}	GET			200 OK	AntennaVMResponse
Aggiorna antenna dato il suo identificativo	/API/devices/antennas/{antennaId}	PUT	AntennaVMRequest		204 NO CONTENT	

Figura 7.7: Interfaccia applicativa device service - parte 1 di 2

Interfaccia applicativa device service (parte 2 di 2)

Operazione	Risorsa	Verbo	Corpo Richiesta	Query Parameters	Success Status	Corpo Risposta
Crea macchina	/API/devices/machines	POST	MachineVMRequest		201 CREATED	MachineVMResponse
Dammi la macchina dato il suo identificativo	/API/devices/machines/{machineId}	GET			200 OK	MachineVMResponse
Aggiorna macchina dato il suo identificativo	/API/devices/machines/{machineId}	PUT	MachineVMRequest		204 NO CONTENT	
Dammi le macchine che soddisfano questa query	/API/devices/byQuery	POST	MachineQueryParametersVMRequest		200 OK	List<MachineVMResponse>
Dammi le antenne che soddisfano questa query	devices/antennas/byQuery	POST	AntennaQueryParametersVMRequest		200 OK	List<AntennaVMResponse>
Associa questa macchina a questa antenna	/API/devices/machines/{machineId}/antennaAssociations/{antennaId}	PUT			204 NO CONTENT	
Disassocia questa macchina da questa antenna	/API/devices/machines/{machineId}/antennaAssociations/{antennaId}	DELETE			200 OK	AntennaAssociationVMResponse
Dammi tutte le associazioni con antenne di questa macchina	/API/devices/machines/{machineId}/antennaAssociations	GET			200 OK	List< AntennaAssociationVMResponse>
Dammi l'associazione tra questa macchina e questa antenna	/API/devices/machines/{machineId}/antennaAssociations/{antennaId}	GET			200 OK	AntennaAssociationVMResponse
Dammi le antenne associate a questa macchina	/API/devices/machines/{machineId}/antennaAssociations/targets	GET			200 OK	List<AntennaVMResponse>

Figura 7.8: Interfaccia applicativa device service - parte 2 di 2

7.3.4 Progettazione del database

Segue il diagramma entità-relazioni corrispondente al modello concettuale del database di questo microservizio.

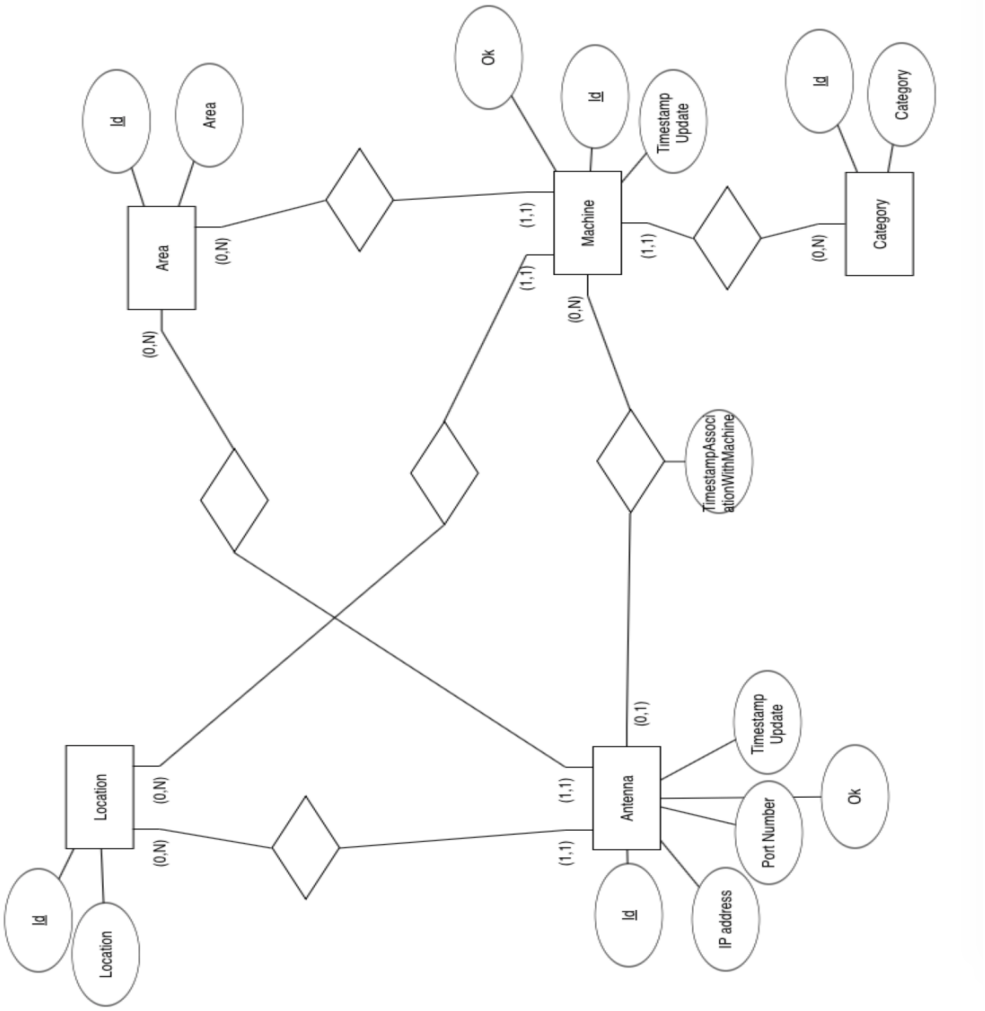


Figura 7.9: Database device service

7.4 Box service

7.4.1 Introduzione al microservizio

Il box service è il componente che ha il compito di gestire le scatole e il loro ciclo di vita. Una scatola è un contenitore che contiene un certo numero di pezzi di un certo tipo di prodotto e sul quale è applicato un tag RFID. Una scatola ha uno stato, che evolve nel corso del tempo ed indica l'area e la location in cui si trova tale scatola e la condizione in cui si trova in tale area in tale location. Una scatola, quindi, è univocamente identificata dal suo id (corrispondente al contenuto del tag RFID applicatole) ed è caratterizzata dal suo tipo, dallo stato (che corrisponde alle informazioni dette sopra), dal codice dell'ordine di cui fa parte, dal codice della spedizione di cui fa parte, dalla destinazione, dal codice del prodotto contenuto al suo interno e dal numero di pezzi di tale prodotto che contiene al suo interno. Si vuole avere l'informazione corrispondente allo storico del ciclo di vita di una scatola, per questo motivo si tiene traccia di ogni aggiornamento riguardante una scatola considerando l'istante temporale in cui tale aggiornamento si verifica. Il box service offre quindi le funzionalità atte a gestire lo scenario appena descritto, dalla creazione e aggiornamento di scatole fino alle funzionalità di interrogazione e ricerca. In questo caso i requisiti chiedevano esplicitamente di fornire la funzionalità di consultazione delle informazioni relative alle scatole tramite HTTP, per cui è stato adottato tale protocollo per fornire l'intero insieme di funzionalità di questo microservizio. Anche in questo caso si è aderito alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Di seguito verranno trattati rispettivamente il design delle risorse di questo microservizio, la definizione della sua interfaccia applicativa e infine la progettazione del suo database.

7.4.2 Design delle risorse

Di seguito si propone in forma tabellare il frutto della fase di design delle risorse e quindi l'insieme dei tipi di dato da scambiare con questo microservizio.

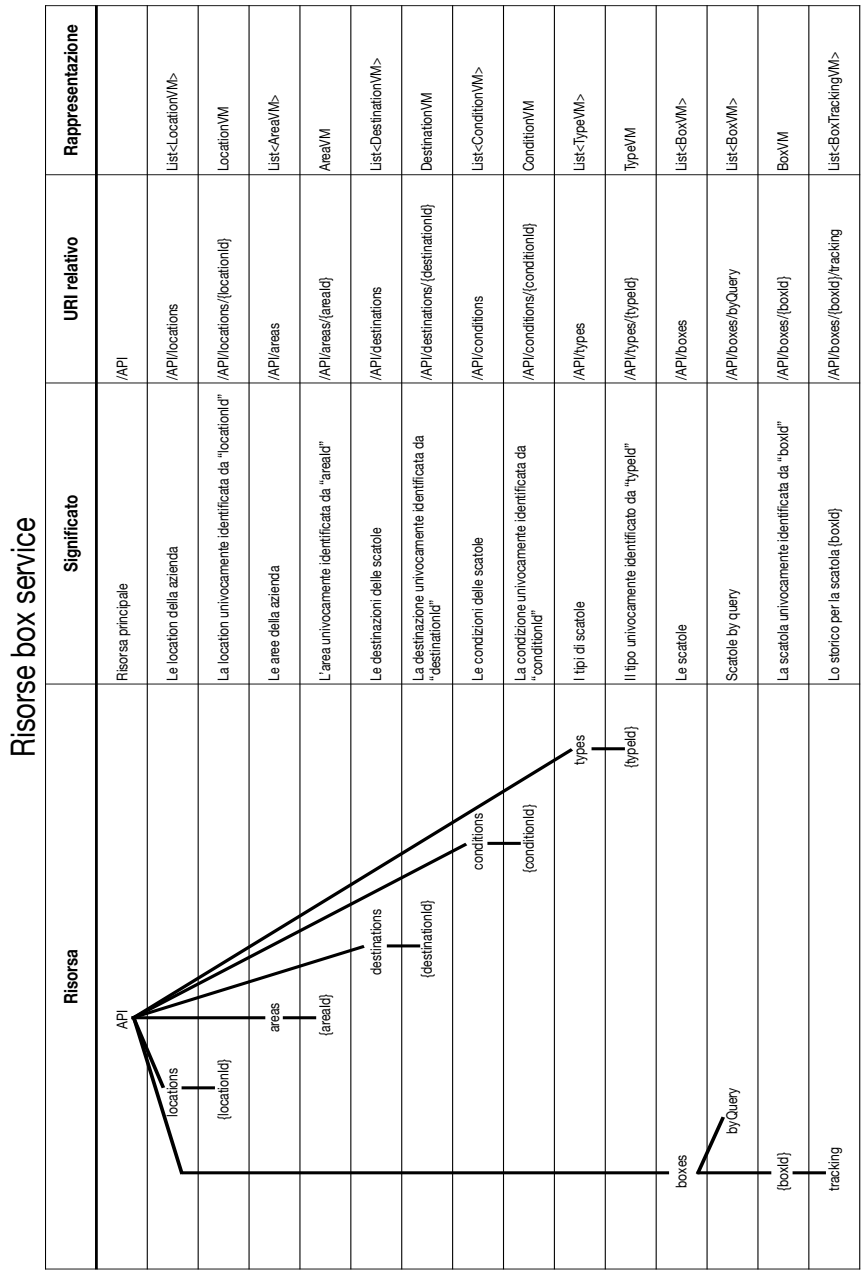


Figura 7.10: Risorse box service

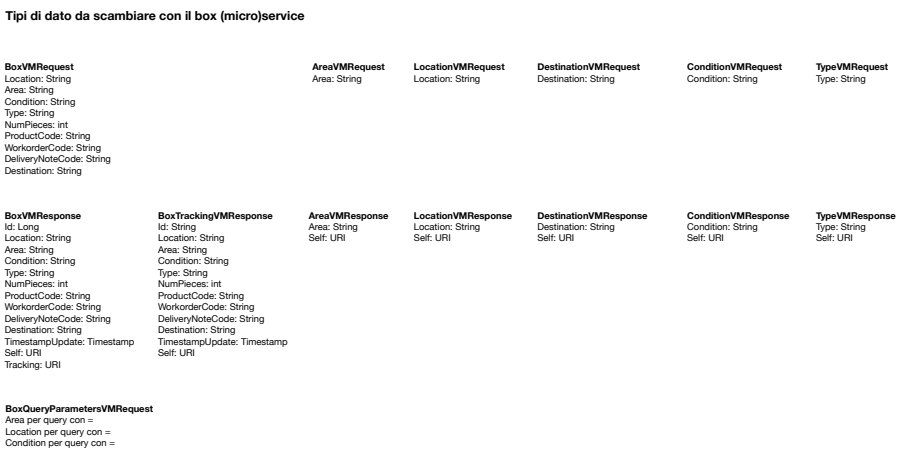


Figura 7.11: Tipi di dato box service

7.4.3 Definizione dell'interfaccia applicativa

Qui si espone l'interfaccia applicativa definita in modo tale da conformarsi alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Si riportano solo i casi di successo.

Interfaccia applicativa box service (parte 1 di 2)

Operazione	Risorsa	Verbo	Corpo Richiesta	Query Parameters	Success status	Corpo Risposta
Crea location	/API/locations	POST	LocationVMRequest		201 CREATED	LocationVMResponse
Dammi la location dato il suo identificativo	/API/locations/{locationId}	GET			200 OK	LocationVMResponse
Aggiorna location dato il suo identificativo	/API/locations/{locationId}	PUT	LocationVMRequest		204 NO CONTENT	
Crea area	/API/areas	POST	AreaVMRequest		201 CREATED	AreaVMResponse
Dammi l'area dato il suo identificativo	/API/areas/{areaId}	GET			200 OK	AreaVMResponse
Aggiorna area dato il suo identificativo	/API/areas/{areaId}	PUT	AreaVMRequest		204 NO CONTENT	
Crea destinazione	/API/destinations	POST	DestinationVMRequest		201 CREATED	DestinationVMResponse
Dammi la destinazione dato il suo identificativo	/API/destinations/{destinationId}	GET			200 OK	DestinationVMResponse
Aggiorna destinazione dato il suo identificativo	/API/destinations/{destinationId}	PUT	DestinationVMRequest		204 NO CONTENT	
Dammi le location	/API/locations	GET			200 OK	List<LocationVMResponse>
Dammi le aree	/API/areas	GET			200 OK	List<AreaVMResponse>
Dammi le destinazioni	/API/destinations	GET			200 OK	List<DestinationVMResponse>

Figura 7.12: Interfaccia applicativa box service - parte 1 di 2

Interfaccia applicativa box service (parte 2 di 2)

Operazione	Risorsa	Verbo	Corpo Richiesta	Query Parameters	Success status	Corpo Risposta
Crea condizione	/API/conditions	POST	ConditionVMRequest		201 CREATED	ConditionVMResponse
Dammi la condizione dato il suo identificativo	/API/conditions/{conditionId}	GET			200 OK	ConditionVMResponse
Aggiorna condizione dato il suo identificativo	/API/conditions/{conditionId}	PUT	ConditionVMRequest		204 NO CONTENT	
Dammi le condizioni	/API/conditions	GET			200 OK	List<ConditionVMResponse>
Crea type	/API/types	POST	TypeVMRequest		201 CREATED	TypeVMResponse
Dammi il type dato il suo identificativo	/API/types/{typeId}	GET			200 OK	TypeVMResponse
Aggiorna type dato il suo identificativo	/API/types/{typeId}	PUT	TypeVMRequest		204 NO CONTENT	
Dammi i types	/API/types	GET			200 OK	List<TypeVMResponse>
Crea la risorsa o aggiorna la rappresentazione della risorsa scatola con questo identificativo	/API/boxes/{boxId}	PUT	BoxVMRequest		204 NO CONTENT	
Dammi la rappresentazione della scatola dato il suo identificativo	/API/boxes/{boxId}	GET			200 OK	BoxVMResponse
Dammi la rappresentazione delle scatole che soddisfano questa query	/API/boxes/byQuery	POST	BoxQueryParametersVMRequest			List<BoxVMResponse>
Dammi lo storico della scatola dato il suo identificativo	/API/boxes/{box id}/tracking	GET			200 OK	List<BoxTrackingVMResponse>

Figura 7.13: Interfaccia applicativa box service - parte 2 di 2

7.4.4 Progettazione del database

Segue il diagramma entità-relazioni corrispondente al modello concettuale del database di questo microservizio.

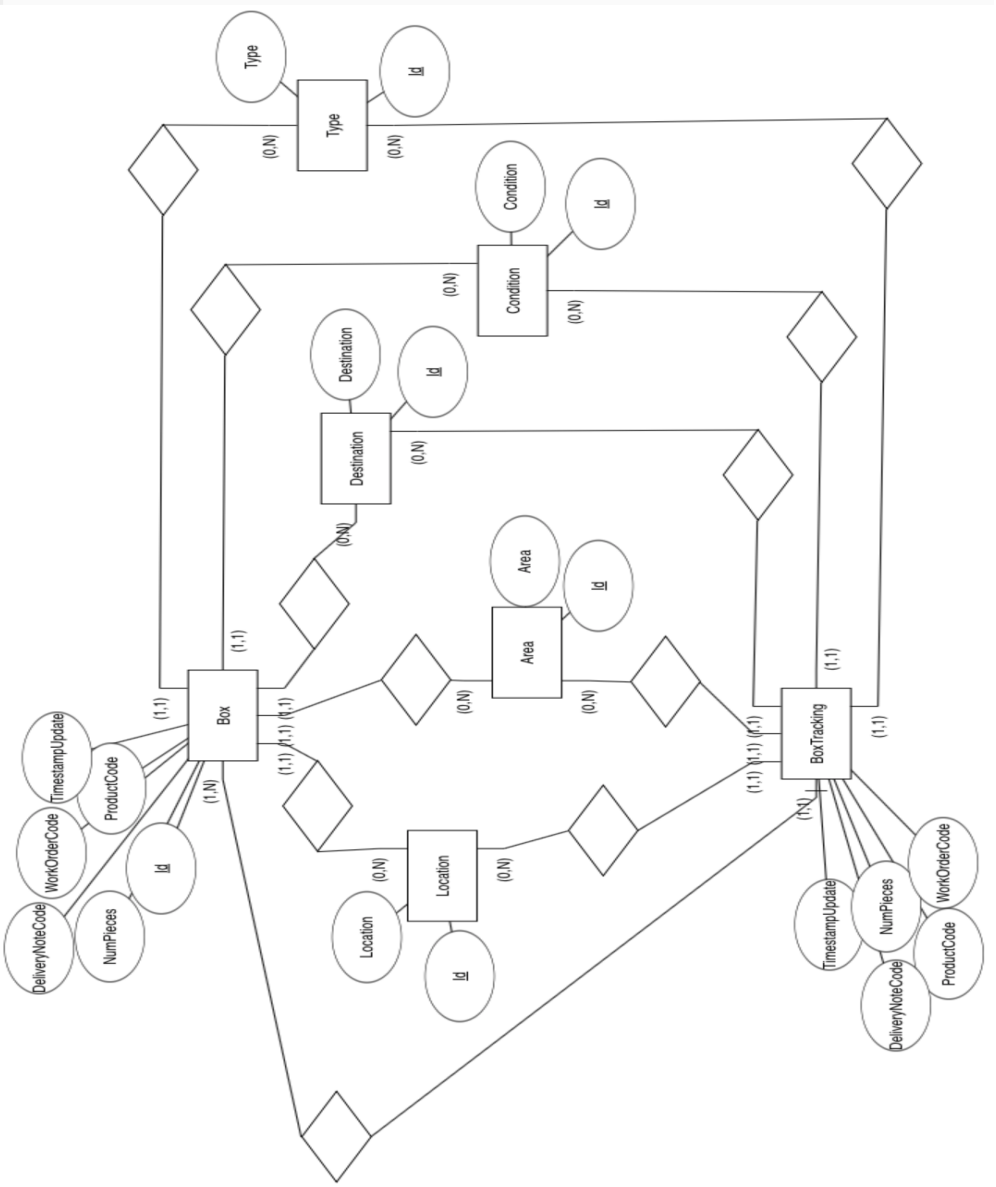


Figura 7.14: Database box service

7.5 Message&Alert service

7.5.1 Introduzione al microservizio

Il message&alert service è il componente che si occupa della memorizzazione di un messaggio processato (ed eventualmente dell'alert a questo relativo) e che rende disponibili messaggi e alert al mondo esterno fornendo funzionalità di ricerca. Un alert indica un errore verificatosi nel corso del processamento di un messaggio. Il processamento di un messaggio termina positivamente o negativamente: se termina positivamente non avrà un alert associato, se termina negativamente avrà un alert associato. Nonostante basti già questo a dare l'informazione sull'esito del processamento, si dà la possibilità di indicare nello status di un messaggio questa stessa informazione, ovvero il risultato del suo processamento, ma con un livello maggiore di dettaglio. Un messaggio quindi è univocamente identificato dal suo id ed è caratterizzato dalla antenna che lo ha mandato, dalla macchina a cui in quel momento è associata quella antenna, dalla scatola interessata, da alcuni timestamp utili per il dominio di business, dallo status del suo processamento ed, eventualmente, dall'alert a lui associato. L'architettura progettata ha tra i suoi obiettivi quello di essere resiliente, ragion per cui si cerca di fare in modo che un errore non grave verificatosi nel corso del processamento di un messaggio possa essere successivamente recuperato. Il fatto che un errore non grave venga recuperato viene qui considerato come corrispondente al fatto che di quell'errore non grave non si debba tenere più traccia. La gravità di un errore si esprime quindi attraverso la possibilità o impossibilità che ha questo di essere cancellato o meno in futuro. Un alert quindi è univocamente identificato dal suo id ed è caratterizzato dal timestamp in cui è stato creato, dalla possibilità di cancellarlo in futuro, da una descrizione testuale del problema in questione e dal messaggio a cui è associato. Un messaggio quindi viene mandato (eventualmente insieme ad un alert che lo riguarda) a tale service da un subscriber al termine del processamento di tale messaggio da parte di tale subscriber. La politica adottata è la seguente: se si riceve un messaggio il cui processamento è terminato positivamente allora si cancellano tutti gli alert cancellabili relativi alla scatola indicata in tale messaggio. Il dominio applicativo considerato è tale per cui la mole di informazione che questo servizio deve gestire è molto grande. Per questo motivo si è scelto di fare in modo che tale servizio, nell'offrire le funzionalità precedentemente indicate, faccia uso della paginazione dei risultati, con l'obiettivo di non sovraccaricarsi e di non sovraccaricare un qualsiasi suo client. Il message&alert service offre quindi le funzionalità atte a gestire lo scenario appena descritto, dalla memorizzazione di messaggi e alert fino alle funzionalità di interrogazione e ricerca. In questo caso i requisiti chiedevano esplicitamente di fornire la funzionalità di consultazione delle informazioni relative ai messaggi tramite HTTP. È stato quindi adottato tale protocollo per fornire

l'intero insieme di funzionalità di questo microservizio aderendo alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Di seguito si espongono rispettivamente il design delle risorse di questo microservizio, la definizione della sua interfaccia applicativa e infine la progettazione del suo database.

7.5.2 Design delle risorse

Di seguito si propone in forma tabellare il frutto della fase di design delle risorse e quindi l'insieme dei tipi di dato da scambiare con questo microservizio.

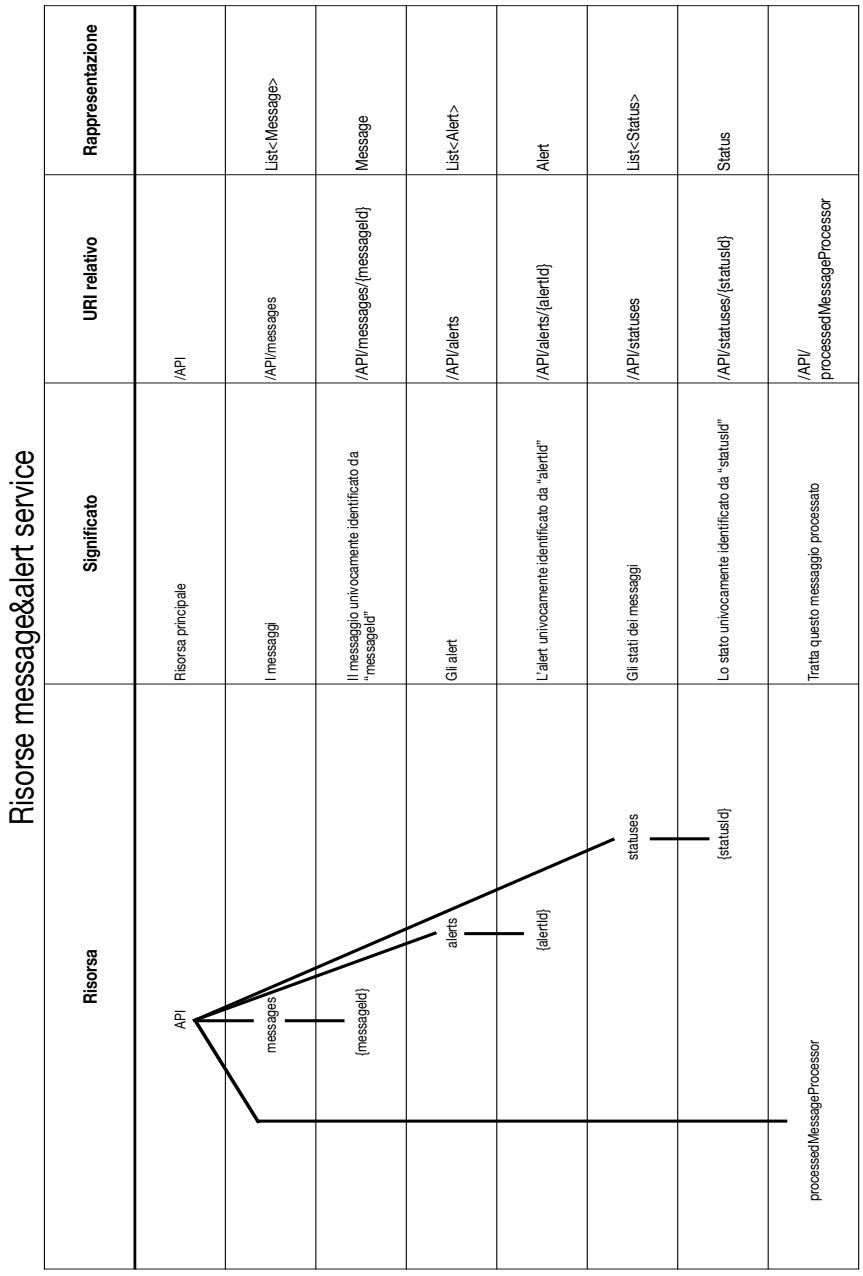


Figura 7.15: Risorse message&alert service

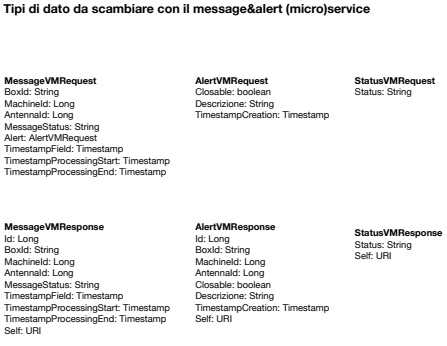


Figura 7.16: Tipi di dato message&alert service

7.5.3 Definizione dell'interfaccia applicativa

Qui si espone l'interfaccia applicativa definita in modo tale da conformarsi alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Si riportano solo i casi di successo.

Interfaccia applicativa Message&Alert service

Operazione	Risorsa	Verbo	Corpo Richiesta	Query Parameters	Success status	Corpo Risposta
Crea status	/API/statuses	POST	StatusVMRequest		201 CREATED	StatusVMResponse
Dammi lo status dato il suo identificativo	/API/statuses/{statusId}	GET			200 OK	StatusVMResponse
Aggiorna status dato il suo identificativo	/API/statuses/{statusId}	PUT	StatusVMRequest		204 NO CONTENT	
Dammi gli stati	/API/statuses	GET			200 OK	List<StatusVMResponse>
Processa questi dati	/API/processedMessageProcessor	POST	MessageVMRequest		200 OK	
Dammi la rappresentazione dei messaggi che soddisfano questa query	/API/messages	GET		• page=NUM_PAGINA (Si parte da zero) • size=NUM_ITEM_PAGINA (Numero itemi in una pagina) • boxId=BOX_ID • machineId=MACHINE_ID • antennaId=ANTENNA_ID	200 OK	Page<MessageVMResponse>
Dammi la rappresentazione degli alert che soddisfano questa query	/API/alerts	GET		• page=NUM_PAGINA (Si parte da zero) • size=NUM_ITEM_PAGINA (Numero itemi in una pagina) • boxId=BOX_ID • machineId=MACHINE_ID • antennaId=ANTENNA_ID	200 OK	Page<AlertVMResponse>
Dammi il messaggio dato il suo identificativo	/API/messages/{messageId}	GET			200 OK	MessageVMResponse
Dammi l'alert dato il suo identificativo	/API/alerts/{alertId}	GET			200 OK	AlertVMResponse

Figura 7.17: Interfaccia applicativa message&alert service

7.5.4 Progettazione del database

Segue il diagramma entità-relazioni corrispondente al modello concettuale del database di questo microservizio.

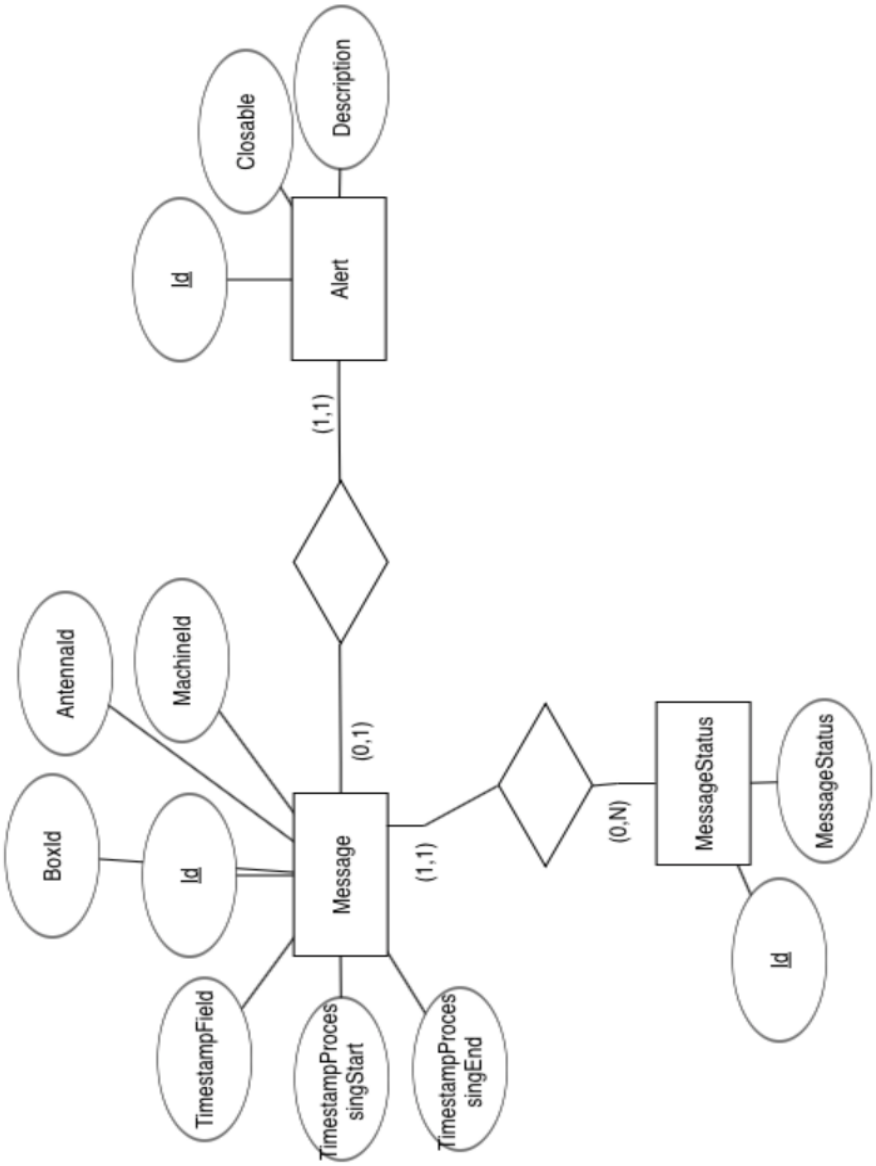


Figura 7.18: Database message&alert service

7.6 Subscriber service

7.6.1 Introduzione al microservizio

Un subscriber è un componente che è posto in sottoscrizione su un topic ed è l'unico in sottoscrizione su quel topic. Per ogni messaggio che riceve si occupa di processarlo e quindi di mandarlo, assieme al risultato del processamento, al message&alert service. Un messaggio ricevuto da un subscriber è caratterizzato dall'identificativo della scatola di cui è stato letto il tag, dall'indirizzo ip del reader che lo ha mandato, dal numero di porta della antenna di tale reader che ha eseguito tale lettura e da un timestamp. Un messaggio inviato da un subscriber al message&alert service, eventualmente con un alert associato, è del tipo ricevuto dal message&alert service pertanto non si ripete qui l'insieme di informazioni che lo caratterizza. Potenzialmente ogni subscriber può fare qualsiasi cosa ritenga opportuna a seguito della ricezione di un messaggio, ma visto che il processamento di un messaggio da parte del sistema software complessivo deve portare all'aggiornamento dello stato di una scatola, quello che capita è che il comportamento dei subscriber abbia uno scheletro di base comune costituito dalla sequenza di passi indicata nelle specifiche della applicazione, e poi ognuno di essi lo specializza sulla base della porzione dei messaggi che deve processare. Di seguito si tratterà quindi tale comportamento di base, evitando di particolareggiarlo per questo o quel subscriber specifico poiché si ritiene non utile e potenzialmente fuorviante scendere troppo nel dettaglio.

7.6.2 Introduzione al meccanismo di processamento di un messaggio

Il processamento di un messaggio ha come obiettivo quello di aggiornare lo stato della scatola di cui è stato letto l'identificativo e deve essere tenuta traccia dell'esito di ciò. Per capire tale meccanismo di aggiornamento si rammenti che lo stato futuro di una scatola dipende dal suo tipo e dal suo stato corrente e dalla categoria, location e area della macchina aziendale che la tratta. Un subscriber qualsiasi quindi, a seguito della ricezione di un messaggio, ha bisogno di recuperare tali informazioni per poter calcolare, nel modo suo specifico, lo stato futuro della scatola lì indicata. Dopo averle recuperate ed aver calcolato tale stato futuro, il subscriber ha bisogno quindi di aggiornare ad esso lo stato della scatola (ed infine di mandare il messaggio ricevuto, con l'esito del suo processamento, al message&alert service). Consideriamo la prima fase, ovvero quella di recupero delle informazioni sopra dette: essa è articolata come segue. Si chiede al device service la rappresentazione della macchina aziendale che si trova nella area specificata e nella location specifica ed è associata all'antenna di cui vengono specificati indirizzo ip e numero di porta del reader associato (tali informazioni vengono prese dal messaggio ricevuto e dal topic

del subscriber). Si controlla che la rappresentazione della macchina ottenuta sia conforme a ciò a cui il topic di questo subscriber corrisponde, ovvero si controllano location, area e categoria della macchina aziendale ricevuta. Si chiede al device service la rappresentazione della antenna che si trova nella area specificata e nella location specifica ed ha indirizzo ip e numero di porta del reader associato pari a quelli specificati (tali informazioni vengono prese dal messaggio ricevuto e dal topic del subscriber). Si chiede al box service la rappresentazione della scatola di cui viene mandato l'identificativo. Si controlla che il tipo di tale scatola sia uno di quelli che questo subscriber sa trattare. Per capire come si procede da ora in avanti è necessario prima affrontare il tema del recupero delle letture saltate, che verrà trattato qui di seguito.

7.6.3 Recupero delle letture saltate

Il sistema complessivo vogliamo che sia resiliente rispetto ad una problematica comune nell'ambito RFID già introdotta nell'analisi dell'applicazione monolitica: il fatto che nonostante un tag sia nei pressi di una antenna questo possa non venir letto. Tale fallimento si traduce nel fatto che la scatola reale in questione cambia stato ma la scatola virtuale a lei corrispondente no. Questo implica che le due entità non siano sincronizzate e che, se non si corregge tale errore, non lo saranno mai più. Infatti se questa scatola dovesse essere letta successivamente da un'altra antenna collegata ad un'altra macchina aziendale che giustamente ha trattato tale scatola, allora il subscriber del caso troverebbe che il tipo della scatola è conforme a ciò che lui sa trattare, ma non interverrebbe poiché lo stato corrente di questa non è quello tale per cui lui può intervenire. Per recuperare il fallimento parziale sopra descritto si deve sfruttare una lettura di un tag di una scatola che è avvenuta per recuperare tutte le precedenti letture che la scatola in questione potrebbe aver saltato. Recuperare una lettura che non è avvenuta non può corrispondere semplicemente a far finta che ci sia stata e quindi a lasciar operare il subscriber che ha ricevuto il messaggio corrispondente alla lettura avvenuta. Questo perché se si facesse così una scatola virtuale, invece di seguire il suo ciclo di vita attraversando tutti i vari stati che lo costituiscono, lo seguirebbe saltandone alcuni. Il fatto di saltare uno stato vuol dire saltare l'elaborazione di un subscriber ed è da evitare perché, tipicamente, un subscriber interagisce con software terzi nel portare a termine il suo compito. Il non eseguire l'elaborazione di un subscriber per una scatola quindi potrebbe implicare l'avere inconsistenze con eventuali software terzi e pertanto non deve capitare. Questo discorso fa capire che sia necessario fare in modo che il recupero di una mancata lettura si traduca nell'attuare un meccanismo che porti il sistema e il mondo esterno in una condizione non differente da quella a cui si sarebbe arrivati se ci fosse stata. Nella versione monolitica era possibile e anche appropriato eseguire del codice compensativo che portasse il sistema e il

mondo esterno nella condizione che si avrebbe avuto se tali letture non fossero state saltate, per poi eseguire il codice conseguente alla lettura avvenuta. In questo caso, invece, non è possibile e nemmeno appropriato fare ciò perché vorrebbe dire sovrapporre (anche se solo in parte) i compiti dei subscriber, introducendo quindi delle dipendenze tra questi che creerebbero problemi nella fase di mantenimento del sistema. Parte del lavoro di questa tesi pertanto è consistita nell'ideare una soluzione a tale problema. Di seguito la si espone. Il meccanismo che si è scelto di adottare prevede che ogni subscriber, oltre alla funzionalità corrispondente al processare un messaggio, offra anche la funzionalità corrispondente al processare un messaggio rispondendo a chi glielo ha mandato in merito all'esito di tale processamento. A primo impatto si potrebbe pensare di sfruttare tale funzionalità aggiunta nel seguente modo. Un subscriber, quando riceve la rappresentazione di una scatola di un tipo che sa trattare, sulla base del tipo e dello stato corrente di questa, simula un messaggio (creandolo e mandandolo) verso il subscriber che deve farle compiere il primo cambiamento di stato che essa ha saltato. Se riceve risposta affermativa procede quindi con la simulazione di un messaggio verso il subscriber responsabile del secondo cambiamento di stato che tale scatola ha perso e così via fino ad arrivare alla situazione in cui la scatola ha lo stato per cui lui può intervenire, quindi interviene. Risulta evidente che il fatto di rispondere con l'esito del processamento è necessario per poter procedere con l'elaborazione. Così facendo, però, si introdurrebbe nel sistema una dipendenza molto forte tra i diversi subscriber: se, ad esempio, un ciclo di vita necessitasse di una modifica allora tutti i subscriber che vi operano dovrebbero essere modificati di conseguenza. Per questo motivo non deve essere adottata tale strategia. Quello che si deve fare, invece, è sfruttare la funzionalità aggiunta come indicato qui di seguito. In generale si ha che ogni subscriber, per ogni tipo di scatola che sa trattare, è in grado di far passare una scatola di quel tipo da un certo stato (di partenza) ad un altro (di arrivo). L'alternativa sopra descritta si basa sulla concezione secondo cui un subscriber, per ogni tipo di scatola che sa trattare, sa trattarne esclusivamente uno stato (quello detto di partenza). Per fare in modo che tra i subscriber ci sia meno dipendenza si deve cambiare tale concezione. Un subscriber deve essere visto come in grado di trattare effettivamente, per ogni ciclo di vita che lo compete, un singolo stato (quello di partenza), ma deve essere visto anche come in grado di trattare, per ogni ciclo di vita che lo compete, ogni stato dall'inizio fino a quello che effettivamente è in grado di trattare. Si ha quindi che un subscriber, quando riceve la rappresentazione di una scatola di un tipo che sa trattare, se nota che lo stato di questa (sulla base del suo tipo) è tra quelli precedenti a quello che lui sa effettivamente trattare, allora simula un messaggio (creandolo e mandandolo) verso il subscriber che sa trattare tutti gli stati dall'inizio fino a quello precedente a quello che lui sa effettivamente trattare. Se riceve risposta affermativa al messaggio simulato significa che la scatola in questione è stata processata correttamente dal subscriber contattato. Questo

implica che ora lo stato di tale scatola sia proprio pari a quello su cui il subscriber che ha ricevuto il messaggio dal campo ha competenza e che quindi esso ora possa processare tale messaggio arrivato dal campo. Anche in questo caso risulta evidente che il fatto di rispondere con l'esito del processamento sia necessario per poter procedere con l'elaborazione (il fatto che non arrivi risposta deve essere considerato come il caso in cui si riceve risposta negativa da parte del subscriber contattato). È importante notare che il meccanismo appena descritto può ripetersi a ritroso: un subscriber che riceve un messaggio simulato, se scopre a sua volta che lo stato della scatola in questione (sulla base del suo tipo) è tra i precedenti rispetto a quello di sua competenza, allora a sua volta simula un messaggio verso il subscriber che sa trattare gli stati fino a quello precedente al suo, il quale a sua volta può fare lo stesso e così via. Il fatto di sfruttare una lettura per recuperare le eventuali precedenti letture saltate implica necessariamente l'introdurre dipendenze tra i subscriber del sistema. La ragione per cui si deve adottare questo meccanismo è perché è quello che richiede la minore dipendenza possibile tra di essi. In tale scenario un subscriber, infatti, deve conoscere soltanto, per ogni ciclo di vita che tratta, il subscriber responsabile di trattare tutti gli stati dall'inizio fino a quello precedente al suo e non tutti i subscriber coinvolti in tale ciclo di vita. Ritornando al caso in cui si dovesse, per esempio, modificare un ciclo di vita allora con questo meccanismo basterebbe modificare solo ed esclusivamente il subscriber subito a valle della modifica e lasciare il resto invariato. Nella progettazione e nella realizzazione della soluzione a microservizi, pertanto, è stato adottato tale meccanismo. Il modo in cui è stato implementato è tramite l'utilizzo dell'HTTP in un modo conforme alla filosofia REST secondo il livello 3 del modello di maturità di Richardson. Il motivo per cui si è scelto l'HTTP è perché tra i protocolli già utilizzati nell'applicazione è quello più adatto ad implementare un pattern del tipo richiesta-risposta sincrono. La figura 7.19 esprime graficamente entrambi i meccanismi, quello scartato e quello adottato, che si sono esposti qui, utilizzando un esempio riferito al contenuto della figura 7.3. Ora verrà ripreso il processamento di un messaggio tenendo conto di ciò.

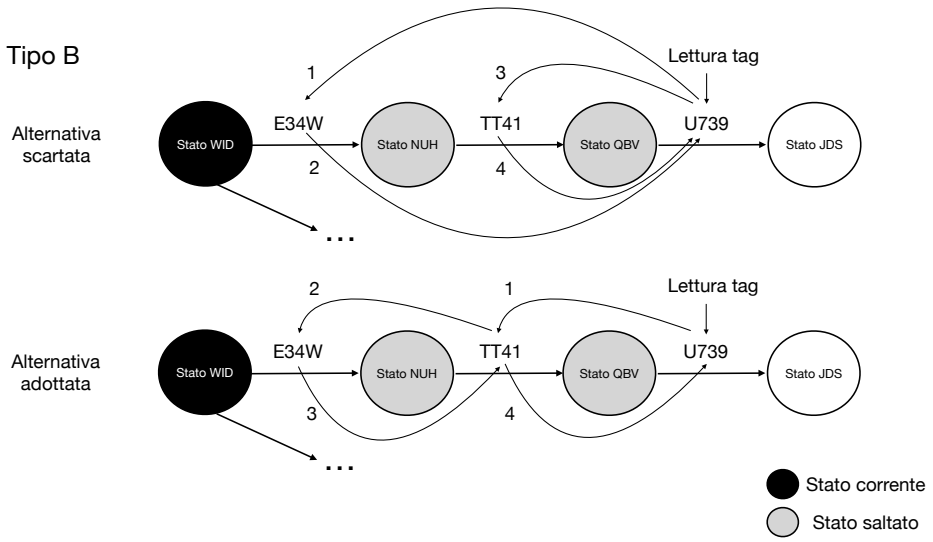


Figura 7.19: Alternative per il recupero delle letture saltate

7.6.4 Meccanismo di processamento di un messaggio

Si è detto che il processamento di un messaggio ha come obiettivo quello di aggiornare lo stato della scatola di cui è stato letto l'identificativo e che deve essere tenuta traccia dell'esito di ciò. Ricordiamo che lo stato futuro di una scatola dipende dal suo tipo e dal suo stato corrente e dalla categoria, location e area della macchina aziendale che la tratta. Un subscriber quindi, a seguito della ricezione di un messaggio, ha bisogno di recuperare tali informazioni per poter calcolare, nel modo suo specifico, lo stato futuro della scatola lì indicata. Dopo averle recuperate ed aver calcolato tale stato futuro, il subscriber ha bisogno quindi di aggiornare ad esso lo stato della scatola (ed infine di mandare il messaggio ricevuto, con l'esito del suo processamento, al message&alert service). Il processo si articola nel seguente modo. Innanzitutto si chiede al device service la rappresentazione della macchina aziendale che si trova nella area specificata e nella location specifica ed è associata all'antenna di cui vengono specificati indirizzo ip e numero di porta del reader associato (tali informazioni vengono prese dal messaggio ricevuto e dal topic del subscriber). Quindi si controlla che la rappresentazione della macchina ottenuta sia conforme a ciò a cui il topic di questo subscriber corrisponde, ovvero si controllano location, area e categoria della macchina aziendale ricevuta. Si chiede ora al device service la rappresentazione della antenna che si trova nell'area specificata e nella location specifica ed ha indirizzo ip e numero di porta del reader associato pari a quelli specificati (tali informazioni vengono prese dal messaggio ricevuto e dal topic del subscriber). Si chiede quindi al box service la rappresentazione della scatola di cui viene mandato l'identificativo. A questo punto si controlla se si rientra in uno dei seguenti due casi: il tipo della scatola è tra quelli che il subscriber sa trattare e lo stato corrente della scatola è tra gli stati precedenti a quello che esso è effettivamente in grado di trattare oppure il tipo della scatola è tra quelli che il subscriber sa trattare e lo stato corrente della scatola è proprio quello su cui tale subscriber è effettivamente in grado di lavorare. Nel primo caso si simula un messaggio per il subscriber che sa trattare tutti gli stati fino a quello precedente a quello di questo subscriber e, in caso di risposta affermativa, si ricomincia il processo dal principio. Nel secondo caso, invece, si procede con il processamento come segue. Si calcola lo stato futuro della scatola in questione e, interagendo con il box service, si aggiorna la scatola virtuale a tale stato appena calcolato. Infine si manda al message&alert service il messaggio ricevuto con l'esito del suo processamento.

Ogni operazione sopra detta può fallire per ragioni riconducibili alla non disponibilità di uno o più microservizi. I controlli detti sopra possono fallire o può capitare che non si rientri nei casi in cui è possibile proseguire per le ragioni più diverse. Consideriamo dapprima il problema della non disponibilità di uno o più microservizi. Nell'architettura proposta ogni microservizio è un SPOF per la

funzionalità che offre, visto che non esistono due o più microservizi che forniscono la stessa funzionalità. Si è voluto fare in modo che il sistema progettato fosse in grado di tollerare l'indisponibilità di uno o più microservizi nel seguente modo: durante il periodo in cui almeno un microserivizio non è disponibile il sistema deve continuare a funzionare per quanto possibile, eventualmente raggiungendo uno stato di inconsistenza tra i vari database dei vari microservizi o di non sincronizzazione con la realtà, nonappena tutti i microservizi ritornano disponibili deve però essere possibile portarlo allo stato di consistenza e corrispondenza conseguente a ciò che è successo. È evidente che, dato un microservizio non disponibile, la sua funzionalità non è sfruttabile ma quelle di tutti gli altri sì. Per le funzionalità che riguardano un singolo microservizio pertanto si raggiunge quanto detto sopra senza accorgimenti speciali. Per la funzionalità di processamento di un messaggio (che comprende il recupero delle letture saltate), ovvero l'unica che riguarda più microservizi, è necessaria qualche accortezza in più che verrà trattata qui di seguito. Consideriamo dapprima il caso in cui siano indisponibili uno o più microservizi subscriber e l'impatto di ciò su tale funzionalità: il sistema risulta in grado di portarla avanti solo parzialmente (tramite i subscriber che sono ancora attivi). Se un subscriber non è disponibile allora non riceverà i messaggi MQTT a lui destinati dal broker MQTT e non riceverà i messaggi HTTP corrispondenti a letture simulate. Nel primo caso si ha quindi che i messaggi MQTT non vengono processati, nel secondo caso si è deciso che il subscriber chiamante, a fronte della indisponibilità del subscriber chiamato (o a fronte di risposta con errore), deve terminare con errore il processamento del messaggio in questione, comunicando l'esito al message&alert service. Ogni volta in cui una scatola reale cambia stato e il subscriber corrispondente non è disponibile capita che la rappresentazione virtuale e la scatola reale non siano più sincronizzate: in questo caso è questo il problema che deve essere possibile risolvere una volta che il sistema ha di nuovo tutti i componenti attivi. Considerando che tipicamente i fallimenti dei microservizi sono temporanei, si permette al sistema di portarsi allo stato di sincronizzazione conseguente a ciò che è capitato se si effettua almeno una lettura, una volta che tutti i microservizi saranno tornati disponibili, per ogni scatola che ha cambiato stato durante il periodo di non completa disponibilità. Grazie al meccanismo adottato per il recupero delle letture saltate, infatti, tale lettura porterà alla risincronizzazione tra la scatola reale e la sua rappresentazione virtuale. Ciò è stato ritenuto sufficiente a considerare il sistema tollerante rispetto al fallimento di uno o più microservizi subscriber. In questa soluzione è stato deciso di gestire l'irraggiungibilità di almeno uno tra i microservizi device service e box service facendo terminare il processamento del messaggio con errore e comunicandone quindi l'esito al message&alert service. Anche in questo caso le scatole reali che cambiano stato durante l'indisponibilità e quelle virtuali a queste corrispondenti non saranno più sincronizzate. Anche in qui, però, il meccanismo definito per il recupero delle letture saltate rimetterà a posto le cose nonappena

entrambi tali microservizi saranno, assieme al resto del sistema, di nuovo attivi e sarà effettuata almeno una lettura per ogni scatola che ha cambiato stato mentre almeno uno di essi non era attivo. Questo è stato anche qui ritenuto sufficiente a considerare il sistema come tollerante rispetto al fallimento di almeno uno di tali microservizi. Per quanto riguarda invece l'irraggiungibilità del message&alert service è necessaria qualche accortezza in più. Si contatta il message&alert service o per dargli un messaggio o per dargli un messaggio assieme ad un alert. Nel primo caso il processamento di tale messaggio è andato a buon fine, il che implica per il sistema che ci sia stata esclusivamente una operazione di scrittura in un solo database, corrisponde all'aggiornamento dello stato della scatola in questione. Nel secondo caso, invece, il processamento di tale messaggio non è andato a buon fine, il che implica che non sia stata fatta alcuna operazione di scrittura su alcun database. Contattando il message&alert service si chiede di fatto una scrittura sul suo database, indipendentemente dal caso. Questo vuol dire che il processamento di un messaggio, qualsiasi cosa accada, corrisponde ad almeno una operazione di scrittura sul database di un microservizio. La sequenza di operazioni da svolgere corrispondente al processamento di un messaggio deve essere ovviamente tale per cui o è eseguita tutta o non è eseguita per niente, ovvero costituisce una transazione che riguarda più microservizi (l'unica per l'applicazione). Visto che non è necessario per il dominio in questione che il sistema sia consistente ad ogni istante, ma è sufficiente che se non lo è ora sia garantito che lo sarà entro un certo intervallo di tempo, si è deciso di fare in modo che una volta partita una transazione corrispondente al processamento di un messaggio sia garantito che prima o poi questa finisca. Questo in realtà è abbastanza semplice da garantire perché richiede esclusivamente che l'operazione del message&alert service corrispondente alla memorizzazione dell'esito del processamento di un messaggio sia garantito che prima o poi avverrà. Questo accade perché la prima operazione di scrittura della transazione può anche non avvenire, quindi la seconda (cioè tale memorizzazione) è l'unica per cui si deve garantire che avverrà prima o poi. L'indisponibilità di un microservizio (in questo caso del mesage&alert service visto che è il diretto interessato) è qualcosa di temporaneo tipicamente, pertanto si è scelto di risolvere tale problema nel seguente modo automatico. Un subscriber mantiene memorizzato ogni messaggio con l'esito del processamento che avrebbe dovuto mandare al message&alert service (e non è riuscito a mandargli) e periodicamente riprova a mandare tutti tali messaggi memorizzati a tale service, eliminando man mano quelli che riesce effettivamente a mandargli. Questo è stato ritenuto sufficiente per considerare il sistema tollerante rispetto al fallimento di tale microservizio. Chiaramente sono possibili altri casi non enunciati, ma quanto detto è sufficiente a far comprendere al lettore come si comporterebbe il sistema di conseguenza. Non si è ancora detto cosa capita se un controllo tra quelli del processo fallisce o se non si rientra nei casi in cui è possibile proseguire: in tutti questi casi il processamento del messaggio termina con

errore e viene comunicato l'esito al message&alert service. La figura 7.20 esprime graficamente il meccanismo di processamento di un messaggio precedentemente definito.

Di seguito si espongono rispettivamente il design delle risorse di un microservizio subscriber generico (per la parte HTTP ovviamente), l'insieme dei tipi di dato da scambiare con esso, la definizione della sua interfaccia applicativa e infine la progettazione del suo database.

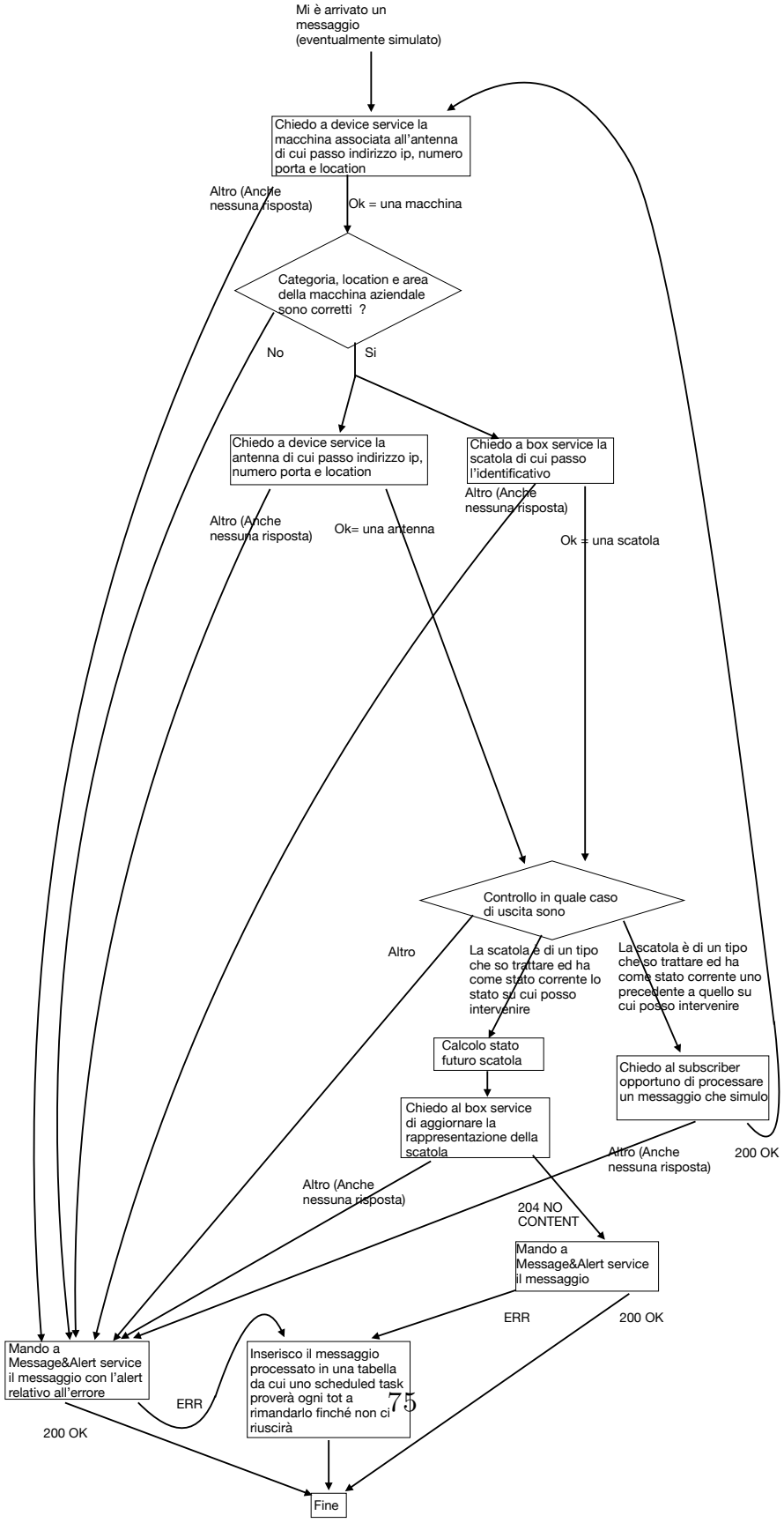


Figura 7.20: Processamento di un messaggio

7.6.5 Design delle risorse

Di seguito si propone in forma tabellare il frutto della fase di design delle risorse e quindi l'insieme dei tipi di dato da scambiare con questo microservizio.

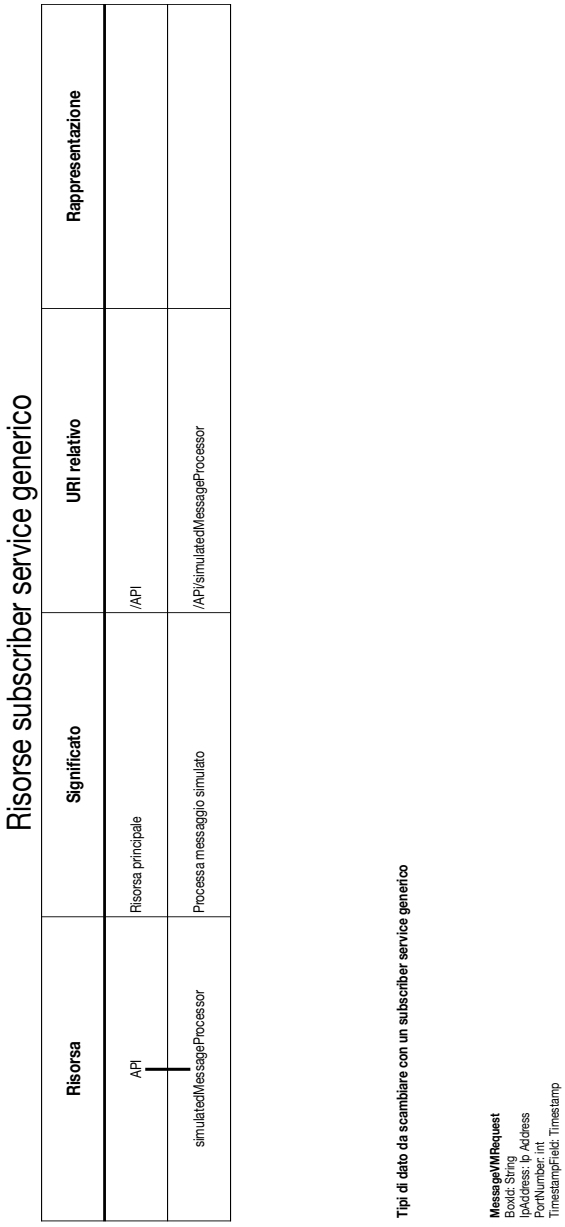


Figura 7.21: Risorse e tipi di dato subscriber service generico

7.6.6 Definizione dell'interfaccia applicativa

Qui si espone sia l'interfaccia applicativa HTTP definita in modo tale da conformarsi alla filosofia REST secondo il livello 3 del modello di maturità di Richardson (si riportano solo i casi di successo) sia l'interfaccia applicativa MQTT.

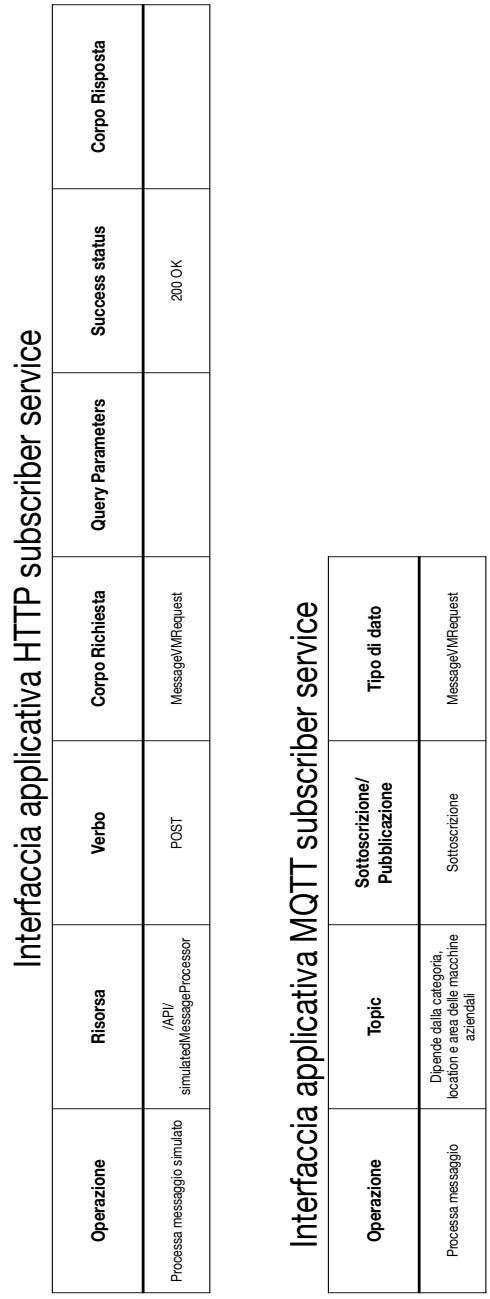


Figura 7.22: Interfaccia applicativa subscriber service generico

7.6.7 Progettazione del database

Segue il diagramma entità-relazioni corrispondente al modello concettuale del database di questo microservizio.

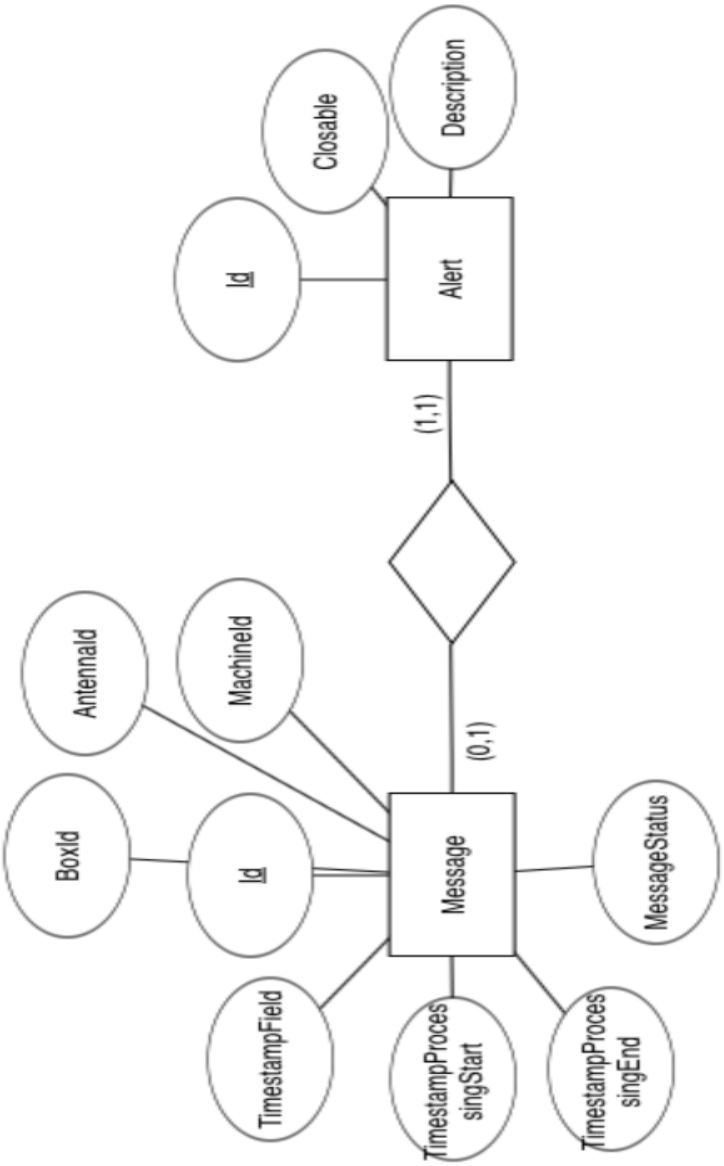


Figura 7.23: Database subscriber service generico

7.7 Considerazioni sulla soluzione proposta

Ogni microservizio definito è piccolo poiché ha poche responsabilità, è raggiungibile solo mediante la rete ed è dotato di un suo esclusivo DBMS. L'architettura proposta è tale per cui la funzionalità di consultazione delle informazioni sui messaggi è affidata ad un unico microservizio: il message&alert service. Anche la funzionalità di consultazione delle informazioni sulle scatole è affidata ad un unico microservizio, ovvero il box service. Le funzionalità di consultazione e modifica delle informazioni sui device sono affidate al device service. Se ci si fermasse a quanto detto finora si considererebbe la suddivisione delle funzionalità richieste alla applicazione, ovvero la definizione dei microservizi, come fatta in modo corretto, visto che ogni microservizio tra quelli sopra detti ha funzionalità che non hanno niente a che vedere con quelle degli altri. Manca però all'appello la funzionalità di processamento di un messaggio. Per quanto riguarda tale funzionalità, invece, si ha che, secondo la definizione dei microservizi che è stata fatta, essa richieda la collaborazione di alcuni di essi per poter essere fornita al mondo esterno: il device service, almeno un subscriber, il box service e il message&alert service. Si fa qui notare come la modalità di collaborazione scelta sia l'orchestrazione: il subscriber che riceve il messaggio è il microservizio che dice ad ogni altro microservizio coinvolto che cosa deve fare, costituendo quindi il cervello dell'elaborazione. In generale adottando un approccio del tipo ad orchestrazione è più facile creare dipendenze forti tra microservizi. In questo caso chiaramente il microservizio subscriber, essendo il centro dell'elaborazione, è dipendente dall'esistenza congiunta di tutti gli altri microservizi per fornire la sue funzionalità (anche se si possono tollerare indisponibilità), ma gli altri microservizi si sottolinea qui che non dipendono dall'esistenza né di un subscriber né di un qualsiasi altro microservizio, ovvero ognuno di essi è in grado di fornire le sue proprie funzionalità senza necessitare dell'aiuto di nessuno. Il paradigma richiesta-risposta verso cui è stata mappata tale orchestrazione è di tipo sincrono: si fa una richiesta e si attende fino a quando non arriva la corrispondente risposta. È stato scelto questo tipo di comunicazione poiché i compiti per cui si fa richiesta sono tipicamente molto brevi e di bassa complessità (poiché corrispondo ad operazioni Create-Read-Update-Delete (CRUD)). Visto che ogni singolo microservizio (per un subscriber solo metà di esso) è stato realizzato seguendo la filosofia REST (nel concreto usando l'HTTP) in un modo classificabile come di livello tre secondo il modello di maturità di Richardson, capita che la comunicazione tra i microservizi coinvolti avvenga seguendo l'approccio HATEOAS. In questo modo si è favorito il disaccoppiamento tra le parti coinvolte nella realizzazione della funzionalità di processamento di un messaggio. Si fa qui notare che se si vuole che l'applicazione sia realizzata seguendo l'approccio a microservizi, come nel nostro caso, allora è inevitabile che tale funzionalità di processamento di un messaggio richieda la collaborazione tra diversi microservizi qualsiasi sia la loro definizione, perché

consiste in una elaborazione in cui sono coinvolti tutti i dati gestiti dalla applicazione. Quanto esposto sopra fa quindi capire che la definizione dei microservizi che è stata fatta risulta corretta anche considerando quest'ultima funzionalità, poiché la collaborazione che si è definita è ottima perché prevede poche dipendenze e, per quanto possibile, lasche. È quindi possibile sviluppare, modificare e mettere in campo ogni microservizio indipendentemente dagli altri (se non si modificano le interfacce applicative ovviamente). Il fatto di aver seguito una filosofia per la realizzazione di un microservizio qualsiasi fa si che se si dovesse voler cambiare il protocollo di comunicazione pur restando all'interno di quelli che permettono di aderire a tale filosofia, allora si potrebbe fare con uno sforzo minimo. Si ha inoltre che le interfacce applicative definite non richiedono ad alcun microservizio di condividere dettagli implementativi, permettendo quindi di seguire la buona pratica di programmazione di information hiding nel realizzarle. Si considera ora la persistenza dei dati. Si è già fatto notare che l'architettura proposta è conforme alle specifiche del mondo a microservizi per cui ogni microservizio ha un suo esclusivo database. Si fa notare che in tale architettura in generale ogni database non è legato a nessun altro database, a dimostrazione del fatto che la suddivisione in sottoinsiemi e la delineazione dei microservizi è stata fatta in un modo corretto. Tuttavia ci sono due casi di dipendenze tra database diversi che si ritiene utile trattare nel dettaglio. Il primo caso riguarda il message&alert service. Il database del message&alert service contiene i messaggi ricevuti dalla applicazione (con eventualmente il rispettivo alert). Un messaggio contiene dei riferimenti alla antenna da cui è arrivato, alla macchina aziendale associata a tale antenna e alla scatola coinvolta. Si vuol far notare che tali riferimenti sono stati fatti utilizzando il meccanismo della chiave esterna, che essendo bidirezionale e non dipendente dall'implementazione era la scelta migliore possibile. Chiaramente se dovesse essere eliminata una chiave primaria (ad esempio una macchina aziendale) la applicazione di per sé si ritroverebbe in uno stato non consistente. Per questo motivo si è scelto di vietare la cancellazione di macchine aziendali, antenne e scatole. Per quanto riguarda i device questa è una scelta abbastanza tipica in ambito aziendale, per quanto riguarda le scatole oltre ad essere tipica è anche una scelta necessaria perché si deve tenere traccia delle informazioni di una scatola per sempre, almeno teoricamente. Il secondo caso riguarda i dati corrispondenti alle location e alle aree aziendali. Questi dati sono necessari sia al device service, per dire dove si trova fisicamente un device, sia al box service, per dire dove si trova fisicamente una scatola. Visto che sono dati che, dopo una fase di configurazione iniziale, si presuppone rimangano stabili nel tempo si è scelto di duplicarli e quindi fare in modo che ogni microservizio dei due sopra detti abbia la sua copia. Questo implica che nella fase di configurazione iniziale sia richiesto di configurare entrambe le copie e che, in generale, vengano mantenute sincronizzate. Tale scelta è stata adottata sia per la natura stessa dei dati, che l'ha consentita, sia perché così si evita di

utilizzare la rete per leggerli (la lettura appunto è l'operazione più comune su essi), il che si traduce quindi in prestazioni migliori.

Capitolo 8

Test della soluzione a microservizi

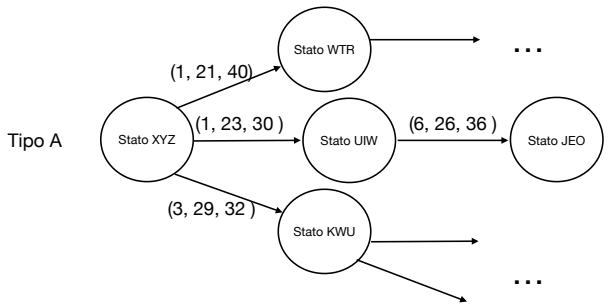
8.1 Scopo di questo capitolo

Nello sviluppo di una soluzione ci sono due passi imprescindibili da compiere. Il primo consiste nel validare la soluzione, il secondo nel verificarla. Validare una soluzione vuol dire confrontare i bisogni dell'utente con i requisiti definiti per tale soluzione e ha come obiettivo quello di stabilire se tali requisiti soddisfano tali bisogni. Verificare una soluzione, invece, corrisponde al confrontare i requisiti validati con quello che è l'artefatto prodotto per soddisfarli e ha come obiettivo quello di stabilire se tale artefatto soddisfa o meno tali requisiti. Esistono diverse tecniche per realizzare tali passi, la cui esposizione esula dagli scopi di questo testo. Considerando il lavoro svolto in questa tesi, pertanto, lo scopo di questo capitolo è quello di esporre il risultato dell'applicazione di tali fasi alla soluzione proposta. Il lavoro qui svolto è una migrazione parziale di una applicazione dall'approccio architetturale monolitico a quello a microservizi. Da ciò emergono due considerazioni. In primo luogo si ha che la fase di validazione dei requisiti della applicazione a microservizi non è necessaria, perché è già stata fatta quando è stata creata l'applicazione monolitica. Come si è già indicato in precedenza l'applicazione monolitica svolge la funzionalità di automazione della logistica di un processo produttivo aziendale, ma non solo. Essa esegue anche altre funzionalità che si basano su essa ed impattano su essa ed altre ancora che si basano su queste e impattano su queste e così via, in modo gerarchico. In secondo luogo si ha quindi che, a causa del fatto che nell'applicazione a microservizi per ragioni aziendali si è scelto di non implementare tutte le funzionalità che sono presenti nella applicazione monolitica, qualsiasi confronto quantitativo tra queste sarebbe inconsistente di per sé. Una volta compreso ciò si ha quindi che quello che manca da fare è verificare

che la soluzione a microservizi proposta soddisfi i requisiti indicati, ovvero verificare che essa sia in grado di automatizzare la logistica del processo produttivo trattato dalla applicazione monolitica. In questo capitolo si esporrà il frutto di tale verifica.

8.2 Metodologia applicata

Per verificare la soluzione proposta si sono eseguiti dei test di tipo black box. In un test black box si fornisce un input al sistema e si confronta l'output prodotto da questo con quello atteso. L'output atteso dato un certo input può essere ottenuto in modi diversi, in questo caso si è scelto di utilizzare un oracolo umano a sua volta basato sui requisiti della applicazione. Si è deciso di svolgere un test per ogni funzionalità che la applicazione svolge. Per quanto riguarda le funzionalità trasportate dall'HTTP si è utilizzato il software client Postman, per quanto riguarda l'MQTT, invece, si è utilizzato un broker MQTT già esistente ed il software client MQTTfx per effettuare le pubblicazioni. Inizialmente pertanto, oltre al sistema da testare, è stato fatto partire l'intero insieme di software sopra detti, dopodiché si sono eseguiti i test: di seguito non si riportano tutti i risultati ottenuti, ma solo quelli sulle funzionalità più importanti. Poiché la funzionalità di processamento di un messaggio (eventualmente simulato) richiede per essere fornita che tutte le altre (sia interne sia verso il mondo esterno) siano fornite, lo stesso tipo di relazione si ha con i test corrispondenti. Con riferimento alla figura 5.1 si è voluto testare la funzionalità di processamento di un messaggio MQTT arrivato da una antenna associata alla macchina aziendale (1, 23, 30) su una scatola già esistente, del tipo A, che si trova nello stato XYZ ed ha id "BOX_ID_TEST_MESSAGGIO_MQTT". Per quanto riguarda invece la funzionalità di recupero delle letture saltate si è voluto testarla tramite un messaggio MQTT arrivato da una antenna associata alla macchina aziendale (6, 26, 36) su una scatola già esistente, del tipo A, che si trova nello stato XYZ e ha id "BOX_ID_TEST_MESSAGGIO_HTTP". Per fare ciò si sono dovute quindi prima testare le altre funzionalità, di modo che potessero a loro volta essere utilizzate per tali due test e di modo che creassero uno scenario in cui poter eseguire tali due test. La figura 8.1 riprende parte della figura 5.1 e indica per gli stati di interesse a cosa corrispondono in termini più dettagliati. Di seguito si riportano quindi i test eseguiti nell'ordine in cui si è dovuto eseguirli.

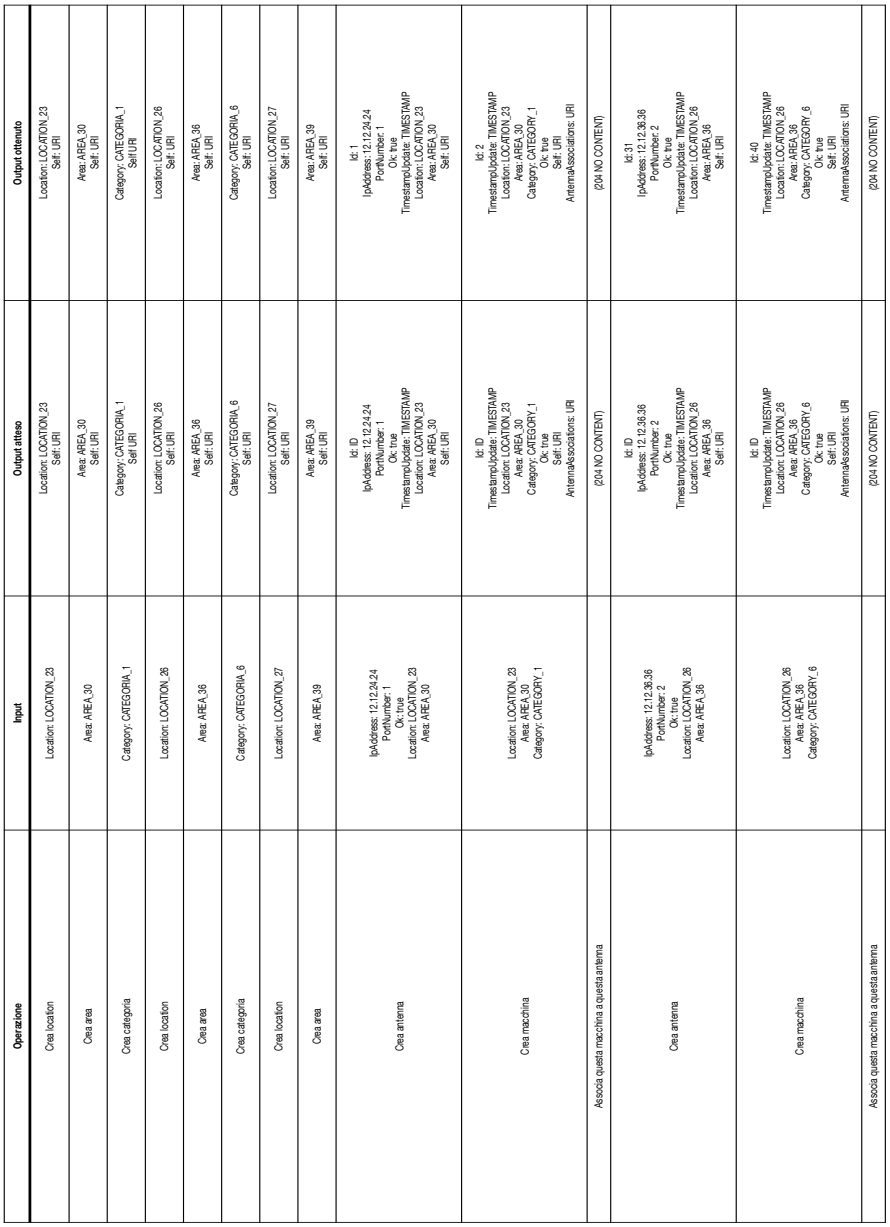


(**α**,**β**,**γ**) = categoria **α**, location **β**, area **γ**

Stato NOMESTATO = (location, area, condizione)
Stato XYZ = (23, 30, A)
Stato UIW = (26, 36, B)
Stato JEO = (27, 39, K)

Figura 8.1: Dettaglio stati scatole di tipo A

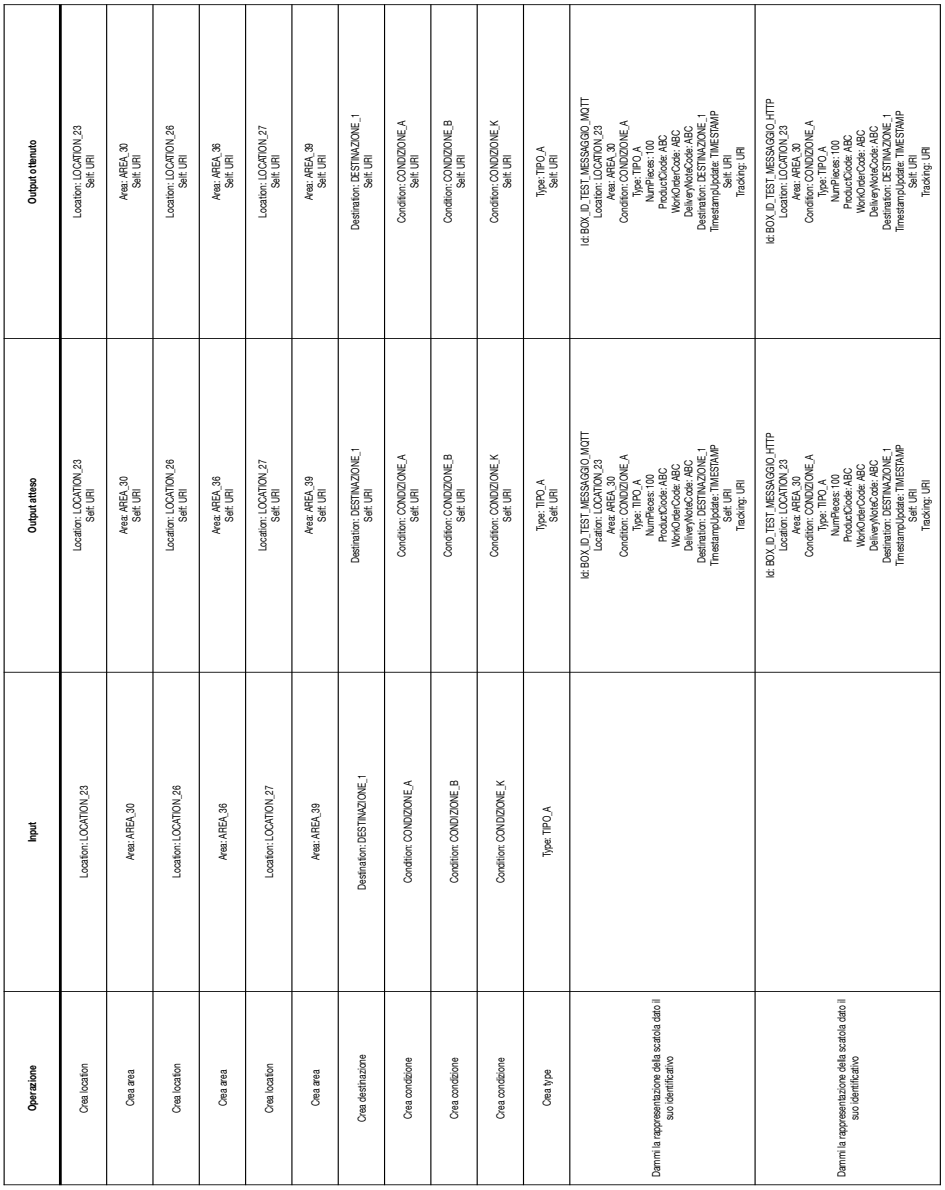
8.3 Test gestione dei device



Operazione	Input	Output atteso	Output ottenuto
Crea location	Location: LOCATION_23	Location: LOCATION_23 Self: URI	Location: LOCATION_23 Self: URI
Crea area	Area: AREA_30	Area: AREA_30 Self: URI	Area: AREA_30 Self: URI
Crea categoria	Category: CATEGORIA_1	Category: CATEGORIA_1 Self:URI	Category: CATEGORIA_1 Self:URI
Crea location	Location: LOCATION_26	Location: LOCATION_26 Self: URI	Location: LOCATION_26 Self: URI
Crea area	Area: AREA_36	Area: AREA_36 Self: URI	Area: AREA_36 Self: URI
Crea categoria	Category: CATEGORIA_6	Category: CATEGORIA_6 Self: URI	Category: CATEGORIA_6 Self: URI
Crea location	Location: LOCATION_27	Location: LOCATION_27 Self: URI	Location: LOCATION_27 Self: URI
Crea area	Area: AREA_39	Area: AREA_39 Self: URI	Area: AREA_39 Self: URI
Crea antenna	Id: ID IpAddress: 12.12.24.24 PortNumber: 1 Ok: true Location: LOCATION_23 Area: AREA_30	Id: ID IpAddress: 12.12.24.24 PortNumber: 1 Ok: true TimestampUpdate: TIMESTAMP Location: LOCATION_23 Area: AREA_30 Self: URI	Id: 1 IpAddress: 12.12.24.24 PortNumber: 1 Ok: true TimestampUpdate: TIMESTAMP Location: LOCATION_23 Area: AREA_30 Self: URI
Crea macchina	Location: LOCATION_23 Area: AREA_30 Category: CATEGORY_1	Id: ID TimestampUpdate: TIMESTAMP Location: LOCATION_23 Area: AREA_30 Category: CATEGORY_1 Ok: true Self: URI AntennaAssociations: URI	Id: 2 TimestampUpdate: TIMESTAMP Location: LOCATION_23 Area: AREA_30 Category: CATEGORY_1 Ok: true Self: URI AntennaAssociations: URI
Associa questa macchina a questa antenna		(204 NO CONTENT)	(204 NO CONTENT)
Crea antenna	IpAddress: 12.12.36.36 PortNumber: 2 Ok: true Location: LOCATION_26 Area: AREA_36	Id: ID IpAddress: 12.12.36.36 PortNumber: 2 Ok: true TimestampUpdate: TIMESTAMP Location: LOCATION_26 Area: AREA_36 Self: URI	Id: 31 IpAddress: 12.12.36.36 PortNumber: 2 Ok: true TimestampUpdate: TIMESTAMP Location: LOCATION_26 Area: AREA_36 Self: URI
Crea macchina	Location: LOCATION_26 Area: AREA_36 Category: CATEGORY_6	Id: ID TimestampUpdate: TIMESTAMP Location: LOCATION_26 Area: AREA_36 Category: CATEGORY_6 Ok: true Self: URI AntennaAssociations: URI	Id: 40 TimestampUpdate: TIMESTAMP Location: LOCATION_26 Area: AREA_36 Category: CATEGORY_6 Ok: true Self: URI AntennaAssociations: URI
Associa questa macchina a questa antenna		(204 NO CONTENT)	(204 NO CONTENT)

Figura 8.2: Test gestione dei device

8.4 Test gestione delle scatole



Operazione	Input	Output atteso	Output ottenuto
Crea location	Location: LOCATION_23	Location: LOCATION_23 Self: URI	Location: LOCATION_23 Self: URI
Crea area	Area: AREA_30	Area: AREA_30 Self: URI	Area: AREA_30 Self: URI
Crea location	Location: LOCATION_26	Location: LOCATION_26 Self: URI	Location: LOCATION_26 Self: URI
Crea area	Area: AREA_36	Area: AREA_36 Self: URI	Area: AREA_36 Self: URI
Crea location	Location: LOCATION_27	Location: LOCATION_27 Self: URI	Location: LOCATION_27 Self: URI
Crea area	Area: AREA_39	Area: AREA_39 Self: URI	Area: AREA_39 Self: URI
Crea destinazione	Destination: DESTINAZIONE_1	Destination: DESTINAZIONE_1 Self: URI	Destination: DESTINAZIONE_1 Self: URI
Crea condizione	Condition: CONDIZIONE_A	Condition: CONDIZIONE_A Self: URI	Condition: CONDIZIONE_A Self: URI
Crea condizione	Condition: CONDIZIONE_B	Condition: CONDIZIONE_B Self: URI	Condition: CONDIZIONE_B Self: URI
Crea condizione	Condition: CONDIZIONE_K	Condition: CONDIZIONE_K Self: URI	Condition: CONDIZIONE_K Self: URI
Crea type	Type: TIPO_A	Type: TIPO_A Self: URI	Type: TIPO_A Self: URI
Dammi la rappresentazione della scatola dato il suo identificativo		Id: BOX_ID_TEST_MESSAGGIO_MQTT Location: LOCATION_23 Area: AREA_30 Condition: CONDIZIONE_A Type: TIPO_A NumPieces: 100 ProductCode: ABC WorkOrderCode: ABC DeliveryNoteCode: ABC Destination: DESTINAZIONE_1 TimestampUpdate: TIMESTAMP Self: URI Tracking: URI	Id: BOX_ID_TEST_MESSAGGIO_MQTT Location: LOCATION_23 Area: AREA_30 Condition: CONDIZIONE_A Type: TIPO_A NumPieces: 100 ProductCode: ABC WorkOrderCode: ABC DeliveryNoteCode: ABC Destination: DESTINAZIONE_1 TimestampUpdate: TIMESTAMP Self: URI Tracking: URI
Dammi la rappresentazione della scatola dato il suo identificativo		Id: BOX_ID_TEST_MESSAGGIO_HTTP Location: LOCATION_23 Area: AREA_30 Condition: CONDIZIONE_A Type: TIPO_A NumPieces: 100 ProductCode: ABC WorkOrderCode: ABC DeliveryNoteCode: ABC Destination: DESTINAZIONE_1 TimestampUpdate: TIMESTAMP Self: URI Tracking: URI	Id: BOX_ID_TEST_MESSAGGIO_HTTP Location: LOCATION_23 Area: AREA_30 Condition: CONDIZIONE_A Type: TIPO_A NumPieces: 100 ProductCode: ABC WorkOrderCode: ABC DeliveryNoteCode: ABC Destination: DESTINAZIONE_1 TimestampUpdate: TIMESTAMP Self: URI Tracking: URI

Figura 8.3: Test gestione delle scatole

8.5 Test processamento di un messaggio

Per quanto riguarda il processamento di un messaggio si hanno due casi possibili: messaggio MQTT oppure messaggio HTTP (il secondo è il caso di un messaggio simulato). Nel primo caso non c'è un output da verificare, perché il processamento di un messaggio MQTT ha solo effetti collaterali. Il secondo caso, invece prevede un output e degli effetti collaterali.

Operazione	Input	Output atteso	Output ottenuto
Processa messaggio	BoxId: BOX_ID_TEST_MESSAGGIO_MQTT IpAddress: 12.12.24.24 PortNumber: 1 TimestampField: TIMESTAMP		
Processa messaggio simulato	BoxId: BOX_ID_TEST_MESSAGGIO_HTTP IpAddress: 12.12.36.36 PortNumber: 2 TimestampField: TIMESTAMP	(200 OK)	(200 OK)

Figura 8.4: Test processamento messaggio (normale e simulato)

Gli effetti collaterali sul box service si riportano qui, quelli sul message&alert service invece verranno riportati in seguito.

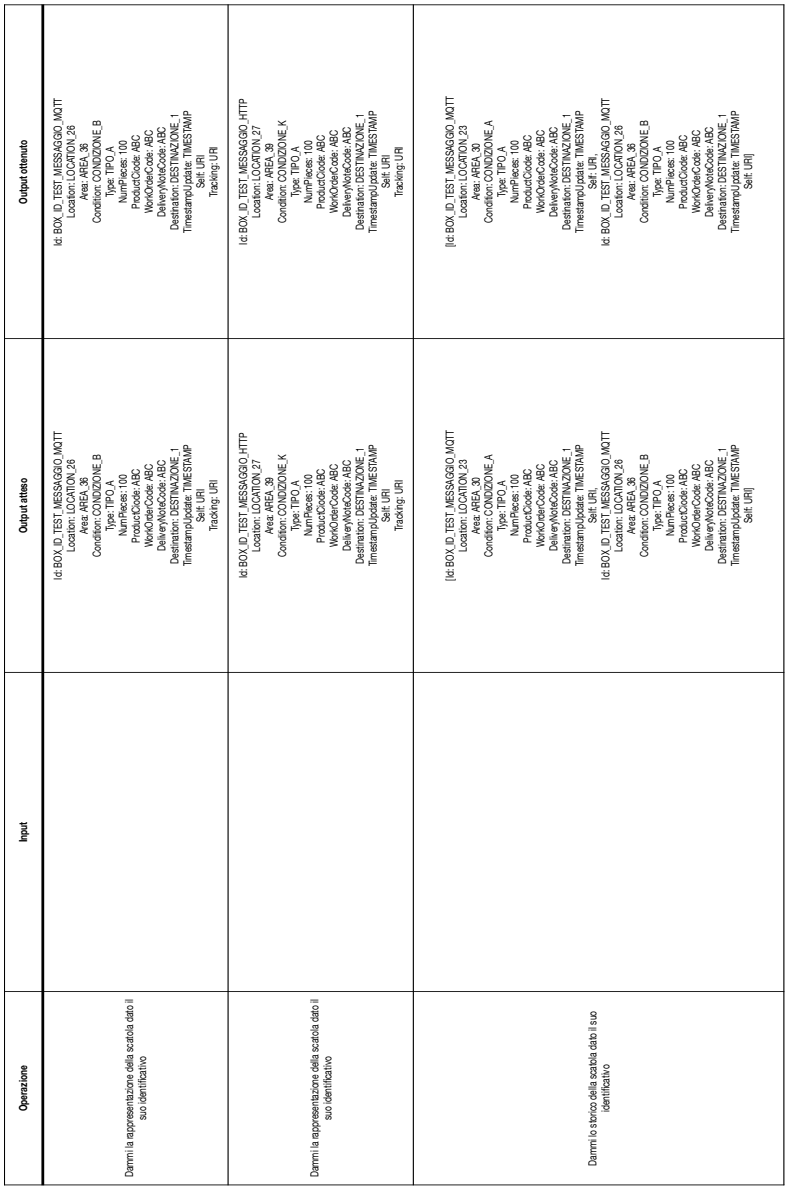


Figura 8.5: Effetti collaterali box service

8.6 Test gestione dei messaggi

Si fa qui notare che l'operazione di creazione degli status di un messaggio è stata testata (e quindi eseguita) prima del test di processamento di un messaggio MQTT e di un messaggio HTTP.

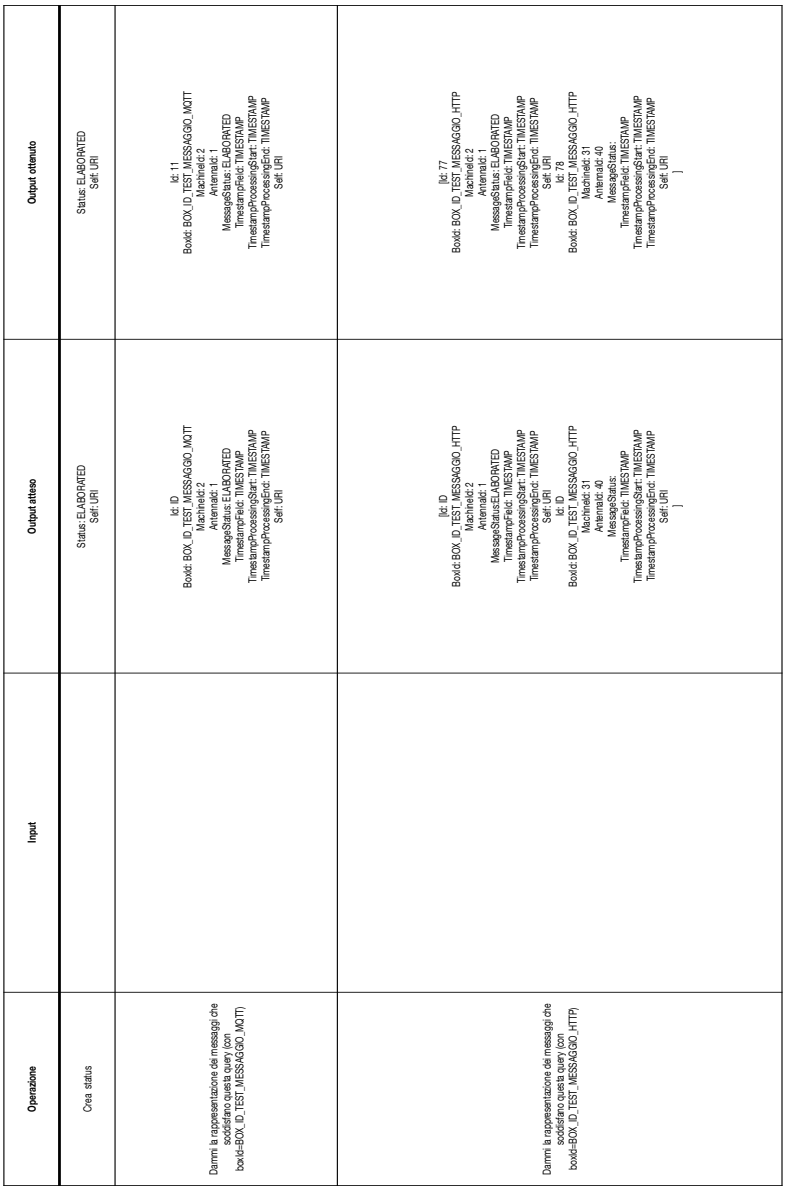


Figura 8.6: Test gestione messaggi

8.7 Considerazioni in merito ai risultati ottenuti

Sulla base dei risultati ottenuti la soluzione proposta si è dimostrata in grado di soddisfare i requisiti precedentemente detti.

Capitolo 9

Analisi della soluzione a microservizi e valutazione raggiungimento obiettivi iniziali

9.1 Scopo di questo capitolo

Questo capitolo ha come obiettivo quello di valutare, tramite una analisi qualitativa della soluzione proposta, il raggiungimento degli obiettivi inizialmente posti. Si ricorda che per raggiungere gli obiettivi iniziali è necessario che la soluzione proposta superi i limiti di quella monolitica dovuti alla scelta errata dell'approccio architetturale (ovvero che abbia le caratteristiche che si ottengono se si segue opportunamente l'approccio architetturale a microservizi), che in essa si utilizzi la tecnologia RFID nel modo in cui la si usa nella soluzione monolitica e che sia facilmente adattabile ad un qualsiasi processo produttivo aziendale. Il fatto che nella soluzione a microservizi si usi la tecnologia RFID nel modo in cui è usata nella applicazione monolitica è evidente perché i requisiti della soluzione proposta comprendevano esattamente il conformarsi a tale uso, per cui si considera tale questione come archiviata e non la si riprenderà in seguito. Restano da considerare le altre due questioni sopra dette. Inizialmente si esporrà quindi il frutto dell'analisi della soluzione proposta, dopodiché si valuterà se questa, adattandosi leggermente, è in grado di automatizzare la logistica di un qualsiasi processo produttivo.

9.2 Analisi della soluzione proposta

9.2.1 Caratteristiche ottenute grazie alla scelta dell'approccio architetturale opportuno

Alla fine del settimo capitolo si è dimostrato che si è seguito opportunamente l'approccio architetturale a microservizi nella fase di design della soluzione proposta. Ciò fa si che tale soluzione abbia le caratteristiche che mancano alla soluzione monolitica e che costituiscono per quest'ultima limiti notevoli. Di seguito si espongono tali caratteristiche. Innanzitutto è utile notare che, se si considera singolarmente ogni microservizio, si ha che, tranne per i subscriber, questo si appoggia su un unico protocollo, il che lo rende un software distribuito poco complesso. La soluzione proposta segue l'approccio architetturale a microservizi. Essendo perciò costituita da un insieme di componenti con cui si può interagire solo tramite la rete, rende possibile collocare ogni componente su un calcolatore diverso (e ogni database in modo tale da non condividere risorse con gli altri database). Se si opera in questo modo si ha che il sistema complessivo è in grado di crescere o decrescere sulla base della domanda attuale con una granularità pari a quella del singolo microservizio. A fronte di una richiesta maggiore per le funzionalità di un singolo microservizio, è infatti possibile spostarlo su un calcolatore più performante senza dover toccare gli altri. Similmente, nel caso contrario, è possibile spostarlo su un calcolatore meno performante senza impattare sul resto del sistema. In questo modo si può reagire in fretta a cambiamenti della domanda e al tempo stesso ottimizzare l'utilizzo delle risorse. Ogni microservizio si è detto essere un SPOF per le funzionalità che offre, ma il fatto che si sia deciso di fare in modo che l'intera applicazione non debba essere fermata se uno o più suoi componenti sono indisponibili fa si che il sistema complessivo risulti più robusto rispetto ad uno SPOF unico. La definizione dei microservizi si è dimostrato che è stata fatta in modo corretto. Questo fa si che sia possibile modificare velocemente l'applicazione e al tempo stesso non fare danni irreparabili nel momento in cui la si modifica. Infatti, a fronte di un cambiamento da dover fare, grazie alla modalità con cui sono stati definiti i microservizi, tipicamente capita che questo sia localizzato in uno solo di essi. Questo microservizio, grazie alle sue caratteristiche, può essere modificato in modo indipendente dal resto del sistema e può essere messo in campo senza dover rimettere in campo l'intero sistema: ciò si traduce nella capacità di attuare modifiche velocemente. Il fatto di poter modificare il sistema velocemente spinge ad adottare un comportamento secondo cui si attua una modifica alla volta. Questo permette a sua volta di non fare danni irrecuperabili nell'attuare modifiche perché è più facile tornare indietro in caso di problemi. Il sistema risulta quindi in grado di evolversi per stare al passo dei cambiamenti di ciò che lo circonda. È bene precisare che per modifica si intende una modifica nel modo in cui una

funzionalità viene realizzata e non nell'interfaccia con cui la si fornisce. È evidente infatti che, nel caso in cui sia necessario modificare l'interfaccia con cui si fornisce una certa funzionalità, allora necessariamente debba essere modificato non soltanto chi realizza tale funzionalità ma anche chi se ne serve. Ogni microservizio è un componente a se stante e, per la maggior parte di quelli del sistema precedentemente definito, è in grado di fornire le proprie funzionalità senza dipendere da altri. Questo fa si che uno qualsiasi di tali microservizi possa essere riutilizzato nella risoluzione di altri problemi che richiedono la funzionalità che esso offre. Nonostante il sistema risultante sia più complesso di quello monolitico, fatto del resto inevitabile, risulta però di più facile comprensione perché è possibile procedere pezzo per pezzo.

Le considerazioni sopra dette fanno capire che la questione riguardante il superamento dei limiti della applicazione monolitica dovuti alla scelta errata dell'approccio architetturale possa considerarsi archiviata.

9.2.2 Caratteristiche ottenute grazie all'utilizzo appropriato dei protocolli di comunicazione

Nella soluzione proposta è stato utilizzato il protocollo HTTP nel modo opportuno: essa è infatti classificabile come di livello tre secondo il modello di maturità di Richardson. Questo fa si che l'interfaccia applicativa dei diversi microservizi, almeno per la parte HTTP, sia auto descrittiva, risultando quindi di immediata comprensione per qualsiasi utilizzatore che conosca la filosofia REST, ma non solo. Supponiamo che in futuro si voglia passare dal'HTTP ad un altro protocollo. Grazie al fatto di aver seguito la filosofia REST il lavoro che sarebbe necessario fare per questa applicazione non sarebbe diverso da quello necessario per una qualsiasi altra applicazione conforme a REST e utilizzante l'HTTP avente lo stesso problema. Questo si traduce nell'avere costi, tempistiche e necessità di conoscenze pari al minimo possibile. Essendo classificata come di livello tre l'applicazione qui analizzata segue il principio HATEOAS. Se fossero necessari cambiamenti sui fornitori dei servizi (anche banali come una modifica degli URI), grazie a ciò non si dovrebbero modificare di conseguenza anche gli utilizzatori già esistenti, vantaggio non di poco conto.

Quanto detto sopra fa capire che anche i limiti che non dipendevano dalla scelta architetturale che sono emersi analizzando l'applicazione monolitica sono stati superati dalla soluzione proposta.

9.3 Valutazione della soluzione proposta

Si è dimostrato che si è seguito nel modo opportuno l'approccio a microservizi, il che corrisponde a dire che si è operato in modo tale da avere alta coesione all'interno

di un microservizio e debole accoppiamento tra microservizi. Si è dimostrato che i servizi costituenti l'applicazione funzionano correttamente. Ciò che è interessante domandarsi è se il risultato ottenuto sia di valore. Per capire ciò si valuta ora il design che è stato fatto da un punto di vista qualitativo.

Consideriamo per prima cosa ogni funzionalità richiesta alla applicazione e valutiamo il modo in cui il design concepito la fornisce. La funzionalità di gestione dei device è stata affidata ad un microservizio apposito. Il vantaggio nell'aver fatto tale scelta si manifesta nel fatto che tale funzionalità non debba necessariamente condividere risorse di alcun tipo con una qualsiasi altra funzionalità. Questo è stato considerato più importante rispetto al fatto di aver introdotto il ritardo della rete nel processamento di un messaggio corrispondente al fatto che chi si occupa di processarlo non ha già "in casa" le informazioni sui device. Il modo in cui si fornisce tale funzionalità quindi è stato considerato ottimo. Si può fare un discorso simile per la funzionalità di gestione delle scatole. Restano da considerare le funzionalità di processamento di un messaggio e di consultazione dei messaggi. Visto che la mole di messaggi con cui l'applicazione complessiva ha a che fare si è detto essere enorme era richiesto che il processamento di un messaggio fosse il più performante possibile, ovvero in termini di design che la progettazione della applicazione tenesse conto di ciò favorendo le prestazioni di tale funzionalità piuttosto che ostacolandole. La valutazione di una soluzione, tra gli altri criteri, deve tenere sicuramente conto dell'utilizzabilità di questa da parte di un qualsiasi client. Considerando questi due aspetti il meccanismo con cui si è deciso di gestire il processamento di un messaggio e la consultazione dei messaggi risulta ottimo per le seguenti ragioni. In primo luogo si ha che la funzionalità di processamento dei messaggi non è a carico di una sola entità, questo vuol dire favorirne le performance perché non si hanno colli di bottiglia. Per di più le entità coinvolte non sono tutte capaci di fare tutto, ma sono stati divisi i compiti: questo fa si che ognuna di esse sia piccola e semplice da realizzare. In secondo luogo si ha che la funzionalità di consultazione dei messaggi è affidata ad un unica entità nonostante il loro processamento sia distribuito tra diversi attori. Per un client esterno qualsiasi è la modalità più facile che ci sia l'usufruire di una funzionalità rivolgendosi ad una sola entità, per cui di meglio non si sarebbe potuto fare. Le considerazioni sopra dette fanno emergere che la soluzione proposta abbia un certo valore per come fornisce le funzionalità che le sono richieste. A maggior ragione quindi si vorrebbe che fosse in grado (se la si adatta) di automatizzare la logistica di un qualsiasi processo produttivo aziendale. Si considera quindi ora tale aspetto.

Si ha che il sistema complessivo, indipendentemente da come sia realizzata all'interno l'applicazione distribuita, è utilizzabile per automatizzare la logistica di un qualsiasi processo produttivo. Quello su cui si vuole porre l'attenzione adesso è il capire se il modo in cui è stata progettata l'applicazione distribuita è tale per cui, se si cambia processo produttivo, non si deve buttare via tale applicazione

ma è sufficiente adattarla. Consideriamo quindi in cosa può cambiare un processo produttivo, ovvero un processo che rientra nella definizione data nel quinto capitolo, rispetto a quello specifico supportato dalla applicazione distribuita. Può capitare ad esempio che, in merito alla localizzazione degli impianti, le sedi siano diverse o le aree siano diverse. Per quanto riguarda le macchine aziendali potrebbe capitare che ne esistano di categorie diverse da quelle del caso trattato. I messaggi che vengono mandati dal campo, oltre alle informazioni necessarie per identificare l'antenna RFID da cui hanno origine, potrebbero dover trasportare dati diversi rispetto a quelli indicati. Può essere necessario che le scatole abbiano, oltre all'identificativo corrispondente al contenuto del tag RFID, informazioni associate diverse. Sia che succeda almeno una delle condizioni sopra dette, sia che non ne succeda nessuna ciò che capita è che sicuramente i cicli di vita delle scatole, ovvero i tipi di queste, saranno sicuramente diversi. Il motivo è che essi sono l'astrazione del processo produttivo, quindi se fossero identici i processi sarebbero identici (ovvero ci sarebbe un solo processo). Questo fatto quindi è il primo problema di cui preoccuparsi (ed è anche il più complesso) quando si vuole adattare la soluzione proposta per utilizzarla in un processo produttivo di un'altra azienda. Il design che è stato concepito richiede, per risolvere questo problema, di non utilizzare i subscriber esistenti, ma di utilizzarne dei nuovi che vanno realizzati appositamente. Si fa ora notare il seguente aspetto. Visto che il design fatto separa il comportamento da adottare nel processare un messaggio dalla gestione dei diversi tipi di dato relativi alla applicazione, si ha che a fronte di tale problema non soltanto il design può (e deve visto il suo valore) restare invariato, ma anche che il lavoro richiesto per risolverlo è semplicemente corrispondente a realizzare (e non a concepire) in modo diverso i subscriber. Se poi, oltre a tale problema, ci fossero anche uno o più altri problemi corrispondenti ai casi sopra detti ciò non sarebbe complicato da gestire. Tali altri casi, infatti, richiedono cambiamenti risibili e banali da fare nel resto della applicazione.

Queste considerazioni fanno emergere che la soluzione proposta sia facilmente adattabile per supportare un qualsiasi altro processo produttivo aziendale, per cui anche l'ultimo obiettivo inizialmente posto risulta raggiunto.

Capitolo 10

Conclusione

10.1 Scopo di questo capitolo

In questo capitolo si vuole concludere l'esposizione del lavoro svolto. Inizialmente si riassumeranno i risultati ottenuti, dopodiché si esporranno possibili sviluppi futuri per la soluzione che tale lavoro ha prodotto.

10.2 Risultati ottenuti

Ciò che si è ottenuto tramite il lavoro qui descritto è quindi il back-end di una applicazione distribuita, realizzato seguendo l'approccio architetturale a microservizi, che, tramite l'utilizzo della tecnologia RFID del mondo IoT, è in grado di automatizzare la logistica di un processo produttivo aziendale, costituendo quindi una una soluzione nel contesto dell'Industry 4.0. Tale applicazione distribuita si è dimostrata essere facilmente adattabile per poter essere utilizzata nell'automatizzazione della logistica di un qualsiasi processo produttivo aziendale. Questo vuol dire che essa ha un campo di applicazione potenzialmente illimitato costituito appunto dall'insieme dei processi produttivi aziendali.

10.3 Possibili sviluppi futuri

La soluzione proposta non è assolutamente pensabile come priva di possibilità di miglioramento.Sono infatti possibili diversi miglioramenti al sistema precedentemente definito.Si è detto che a livello di architettura si è lavorato in modo tale da non ostacolare le prestazioni nella realizzazione delle funzionalità richieste, ma anzi in modo tale da favorirle. In questo contesto sono possibili diversi miglioramenti futuri.

Un esempio può essere costituito dallo sfruttare il caching dell'HTTP, il quale permetterebbe di alleggerire il carico sui singoli microservizi senza richiedere modifiche complesse al sistema. Per ottenere l'utilizzo ottimizzato delle risorse da parte del sistema sulla base della domanda corrente è necessario, come si è detto, avere una architettura a microservizi perché in questo modo si può scalare il sistema con un livello di granularità non corrispondente al sistema stesso, ma più piccolo (ovvero il singolo microservizio).Ciò però non è sufficiente.È infatti necessario anche un sistema di monitoraggio che permetta di capire come evolve la domanda nel tempo per le diverse funzionalità esistenti, sistema che tuttora non esiste.Anche in questo caso si fa notare che tale funzionalità aggiuntiva può essere inserita senza aggiungere troppa complessità perché si può attuare separatamente e in un modo specifico per ogni microservizio.

La soluzione proposta si è detto avere un campo di applicazione potenzialmente illimitato costituito dall'insieme dei processi produttivi aziendali, il che è ottimo. Oltre a ciò va anche considerato che, visto che la funzionalità di automazione della logistica è la base per fornire supporto all'automazione completa di un processo produttivo, tale applicazione non risulta un prodotto fine a se stesso. È infatti possibile considerarla la base da cui partire aggiungendole via via altre funzionalità, siano esse quelle fornite dalla applicazione monolitica o altre (ovviamente continuando a seguire correttamente l'approccio architetturale a microservizi), che facciano tendere sempre più il processo produttivo in questione all'automazione completa.

Bibliografia

[1] Fielding R. «Architectural Styles and the Design of Network-based Software Architectures». Tesi di dott. 2000 (cit. a p. 10).

[2] Fielding R. e Reschke J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 2014 (cit. a p. 11).

[3] Banks A. e Gupta R. *MQTT Version 3.1.1*. 2014 (cit. a p. 16).

[4] Newman S. *Building Microservices*. O'Reilly Media, Inc., 2015 (cit. a p. 19).

[5] Nadareishvili I., Mitra R., McLarty M. e Amundsen M. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 2016 (cit. alle pp. 19, 20).

[6] Abiy Biru Chebudie, Roberto Minerva e Domenico Rotondi. «Towards a definition of the Internet of Things (IoT)». Tesi di dott. 2014 (cit. a p. 28).