POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Progettazione e implementazione del backend di un'applicazione di prospecting

Relatore
prof. Guido Albertengo

Candidato
Giovanni Salviati
matricola: 280188

Supervisore aziendale
Akkodis
dott. ing. Marco Mastropasqua

Anno accademico 2023-2024

Sommario

Questo studio presenta il progetto software backend di un'applicazione aziendale di prospecting, sviluppato all'interno dell'azienda di consulenza *Akkodis*, come parte di un'esperienza di tesi aziendale. Il progetto è stato concepito per indirizzare la necessità di migliorare il processo di prospecting aziendale, il cui fine è quello di identificare, valutare e approcciare eventuali aziende clienti, con lo scopo di instaurare nuove collaborazioni lavorative. Più in particolare, il software è destinato all'uso da parte dei business manager, i quali sono responsabili di cercare e stipulare accordi di consulenza con aziende esterne. Il progetto si è proposto di introdurre funzionalità che semplificano il processo svolto dagli utenti finali, tra cui l'invio di email in maniera automatica tramite l'uso di un template dinamico pre impostato dall'utente, il salvataggio di ricerche effettuate e la generazione di statistiche con granularità temporale personalizzabile. La fase di ricerca dei prospect, ossia le aziende clienti, e dei relativi dipendenti da contattare è stata implementata integrando le funzionalità offerte da Apollo.io, una piattaforma simile a LinkedIn. Inizialmente, si prevedeva di utilizzare le API di LinkedIn; tuttavia, a causa della loro rimozione da parte dell'azienda, si è scelto di adottare le API di Apollo.io. Il processo di sviluppo è stato guidato dalla necessità di realizzare un backend basato su un'architettura a microservizi, con alcune componenti software ospitate sulla piattaforma Microsoft Azure, il servizio cloud di Microsoft. I risultati indicano che un'applicazione aziendale simile, soprattutto quando indirizzata ai singoli manager aziendali, potrebbe notevolmente semplificare e accelerare il processo di ricerca di potenziali clienti. Tuttavia, risulta fondamentale aggiornare il software per gestire attentamente la memorizzazione dei dati sensibili relativi ai dipendenti cercati, al fine di garantire la conformità con le disposizioni del GDPR.

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutte le persone che mi hanno sostenuto e aiutato durante questa esperienza in azienda.

In primo luogo, desidero ringraziare il mio supervisore aziendale, Marco Mastropasqua, che, oltre ad avermi offerto questa opportunità, ha evidenziato in modo sincero quali sono gli aspetti su cui concentrarmi per affrontare una carriera lavorativa nel migliore dei modi.

Inoltre, ringrazio il Prof. Guido Albertengo per aver accettato di essere il relatore della mia tesi.

Un grazie di cuore va anche alla mia famiglia e ai miei amici per il loro costante sostegno emotivo e per avermi incoraggiato nei momenti difficili.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Nel corso di questa tesi, verranno esaminati approfonditamente gli aspetti teorici e pratici relativi al progetto affrontato. Il percorso è strutturato in modo da offrire un'introduzione teorica nei prossimi tre capitoli, che forniscono una base solida per comprendere appieno il contesto e gli obiettivi della ricerca. Successivamente, a partire dal capitolo cinque, ci si immergerà nell'applicazione pratica, esplorando in dettaglio il modo in cui tali concetti teorici sono stati implementati nel contesto del progetto. Prima di proseguire con questa analisi dettagliata, viene presentato un riassunto di ogni capitolo, per offrire una panoramica completa dei temi trattati e preparare il terreno per l'approfondimento successivo.

Nel prossimo capitolo, intitolato *Architettura software*, vengono esaminati dettagliatamente le caratteristiche e le differenze tra due approcci architetturali: il modello monolitico e quello a microservizi.

L'analisi comprende una valutazione dei rispettivi vantaggi e svantaggi. In particolare, viene approfondita la tecnica di decomposizione in microservizi, che consente di identificare ogni singolo microservizio a partire da un'applicazione esistente o dal dominio di un software in fase di sviluppo. Per offrire un approccio pratico alla decomposizione viene esaminato l'algoritmo sviluppato da Chris Richardson, noto come 3-step microservices decomposition.

Successivamente, vengono esposte le sfide legate alla gestione dei dati all'interno di un'architettura a microservizi, insieme alle relative tecniche risolutive.

Infine, vengono analizzate le principali tecniche di comunicazione all'interno di un sistema distribuito.

Il capitolo successivo, dal titolo *Cloud Computing*, discute l'evoluzione del cloud computing, esplorando le sue caratteristiche principali e le sue proprietà distintive.

Inoltre, verrà analizzato il ruolo dei Cloud Service Provider (CSP), con particolare attenzione rivolta a Microsoft Azure, la piattaforma utilizzata nel contesto del progetto aziendale in esame.

In seguito, nel capitolo dedicato al *Processo di sviluppo software*, vengono esaminate le fasi essenziali di tale processo, con un'attenzione particolare riservata alla fase dei

requisiti software. Successivamente, verrà approfondito l'approccio agile e, infine, verrà presentato uno dei framework agile più diffusi al momento, ossia Scrum.

Nel quinto capitolo, intitolato *Il progetto*, si inizia con una descrizione dettagliata del progetto aziendale, fornendo un contesto completo e illustrando gli obiettivi prefissati. Le sezioni successive si concentrano sull'analisi approfondita della progettazione dell'applicazione, includendo la definizione dei requisiti e la progettazione sia dell'architettura a microservizi che dell'architettura cloud.

Il capitolo *Tecnologie e strumenti* presenta i principali strumenti impiegati nel progetto, fornendone una panoramica delle caratteristiche principali e spiegandone brevemente l'applicazione all'interno dello stesso.

Successivamente, nel settimo capitolo, *Test e Validation*, viene esaminata l'importanza dei test automatizzati nelle applicazioni software moderne, illustrando le principali tipologie di test utilizzate e concludendo con un approfondimento sull'approccio Test-Driven Development (TDD).

Il capitolo finale della tesi è dedicato alle *Conclusioni*, dove si discutono i risultati emersi durante il progetto e le possibili modifiche che, se implementate correttamente, potrebbero migliorare l'applicazione.

Capitolo 2

Architettura software

Il seguente capitolo offre un'esplorazione dettagliata delle fondamenta e delle pratiche di progettazione architetturale nel contesto dello sviluppo software moderno.

Nella prima sezione, *Concetti teorici dell'architettura backend*, vengono presentati i principi di base dell'architettura software lato server. Saranno esaminati i diversi approcci architetturali, prestando attenzione all'architettura monolitica e all'architettura a microservizi.

Successivamente, ci concentreremo sulla *Decomposizione in microservizi*, una pratica che favorisce la suddivisione di un'applicazione in componenti indipendenti e scalabili.

Nella sezione *Data management*, saranno esaminate le caratteristiche dei dati gestiti all'interno di un sistema informatico, seguite dalle sfide e dalle soluzioni pratiche per la gestione dei dati sia in un contesto di architettura monolitica che in un contesto di architettura a microservizi.

Infine, nell'ultima sezione, dal titolo *Comunicazione tra microservizi*, saranno analizzati i modelli e gli strumenti utilizzati per facilitare la comunicazione e il coordinamento tra i diversi microservizi all'interno di un'architettura distribuita.

2.1 Concetti teorici dell'architettura backend

Nel corso degli anni si è dibattuto molto riguardo la definizione di architettura software.

Molti esperti del campo la definiscono come l'insieme degli elementi più significativi e di alto livello del sistema, ed il modo in cui essi interagiscono tra loro.

Tuttavia questa definizione non descrive al meglio il concetto, in quanto diversi stakeholder[1] potrebbero ritenere importanti componenti differenti del sistema. Ad esempio, un customer non ritiene rilevanti gli stessi elementi che invece sono fondamentali per un team di sviluppo.

Una definizione più precisa può quindi essere quella data dall'informatico statunitense *Ralph Johnson*, che definisce l'architettura di un sistema software come una visione condivisa dei concetti e componenti rilevanti, di alto livello, del sistema nel suo ambiente, comprensibili da tutti gli sviluppatori ma definiti da quelli più esperti.

Con quest'ultima definizione si coglie perfettamente chi definisce e chi può comprendere un'architettura. Difatti Johnson approfondisce ulteriormente il concetto dicendo testualmente: 'Architecture is about the important stuff. Whatever that is', in italiano: 'L'architettura riguarda le cose importanti. Qualsiasi esse siano', giustificando il motivo per cui devono essere gli sviluppatori esperti a definirla.

Dunque, nel processo di design dell'architettura di un software si affrontano e definiscono tutti quegli elementi che, nel dominio preso in considerazione, sono importanti e impattanti in modo considerevole nella continuazione del progetto. Spesso i fattori determinanti su cui si basa il design sono la difficoltà nel cambiamento e il rallentamento nello sviluppo e distribuzione di nuove feature.

Le applicazioni moderne, soprattutto se si tratta di applicazioni web, possono essere suddivise in due componenti principali. Il *frontend*, ovvero la parte che si occupa di tutto ciò che l'utente finale vede e con cui può interagire. Il *backend*, detto anche server-side, è il cuore dell'applicazione, invisibile agli utenti finali, che comprende la logica del software, la gestione dei dati e le funzionalità di sicurezza. Ogni qual volta un utente richiede, tramite l'interfaccia grafica, l'esecuzione di un'operazione, il frontend invia una richiesta al backend, che dopo averla elaborata, invia indietro una risposta comprensibile dalla controparte.

Per *architettura backend* si intende la struttura e la logica propria del backend, come si comportano i vari componenti software e come interagiscono tra loro. I componenti più importanti richiesti da un backend sono:

- *Server*: è un computer o una macchina virtuale in grado di ospitare l'app e gestire tutte le risorse necessarie per eseguirla;
- *App*: è il codice che viene eseguito nel server e che implementa la logica per interpretare le richieste, eseguire le funzionalità richieste e inviare indietro le risposte;
- *Database*: è una collezione organizzata di dati generati dagli utenti o richiesti per il funzionamento dell'applicazione. Risulta essere molto utile per ridurre il carico

[1] In generale, un stakeholder è una persona, un gruppo o un'organizzazione che ha interesse, coinvolgimento o influenza su un progetto. Questi individui o gruppi possono essere influenzati dalle azioni e dalle decisioni prese all'interno del progetto o dell'azienda e possono influenzarle a loro volta.

sulla memoria principale del server e soprattutto per memorizzare dati in modo persistente;

- *API*: è l'acronimo di Application Programming Interface e rappresenta un insieme di regole e protocolli che consentono a diversi software o componenti software di comunicare tra loro e scambiare dati in modo efficiente e standardizzato. Nel contesto di un'architettura backend le API rappresentano l'interfaccia esposta verso l'esterno, col fine di fornire ai client l'accesso alle risorse e servizi in maniera standardizzata.

Esaminando l'architettura del backend con una visione più ampia, emergono diverse tipologie di architetture che rappresentano modelli consolidati e innovativi nel panorama dello sviluppo software:

- *Architettura monolitica*: è l'architettura in cui si ha una singola app che garantisce il funzionamento dell'intera applicazione;
- *Architettura a microservizi*: in questa tipologia di architettura l'app del backend viene suddivisa in unità indipendenti chiamate microservizi;
- *Architettura serverless*: consiste nell'utilizzare servizi cloud che forniscono tutte le funzionalità backend senza richiedere la gestione di server e infrastrutture.

Nei prossimi paragrafi verranno analizzate l'architettura monolitica e l'architettura a microservizi per poi concentrarsi su alcuni concetti relativi a quest'ultima.

2.1.1 Architettura monolitica vs architettura a microservizi

L'architettura monolitica è l'architettura nata per prima tra le due, in quanto è il risultato più logico e naturale che può venire in mente. Essa è conseguenza del classico modello a tre tier o livelli, che prevede l'esistenza di tre tier logici e fisici separati: il tier di presentazione, ovvero l'interfaccia grafica, il tier dell'applicazione in cui vengono processati i dati e il tier dei dati in cui questi ultimi vengono memorizzati e gestiti. Il *monolite* coincide perciò con il tier dell'applicazione del modello di cui sopra ed è un insieme di servizi e funzionalità messi a disposizione dell'utente ma facenti parte tutti della stessa codebase[2]. In altri termini un'applicazione basata su un'architettura monolitica è una singola unità autosufficiente e indipendente, eseguita come un unico processo all'interno del sistema operativo.

Vantaggi dell'architettura monolitica Di seguito i vantaggi più importanti che si possono trarre sviluppando un monolite:

- Semplicità a partire dalla fase di progettazione rispetto ad architetture distribuite;

[2]Il codebase è l'insieme dei file sorgente, eventuali file di configurazione e documentazione.

- Semplicità in fase di testing, debugging, deploying e monitoring in quanto la codebase è unica ed esiste un solo processo allo start-up del programma;
- Non è necessario gestire la sincronizzazione tra dati memorizzati in database differenti poiché il database è unico;
- Nessun rallentamento causato dalla comunicazione tra processi dato che il programma consiste in un singolo processo.

Svantaggi dell'architettura monolitica Al crescere della grandezza e complessità dell'applicazione potrebbero presentarsi diverse problematiche traducibili in svantaggi del pattern architetturale:

- Gli sviluppatori sono più lenti nell'apportare modifiche al software;
- Cambiamenti apportati ad un modulo potrebbero influenzare il corretto funzionamento di altri moduli e nel caso estremo dar vita ad errori a cascata;
- Il tempo di start-up aumenta notevolmente ostacolando il deployment dell'applicazione;
- È necessario un re-deploy dell'intera applicazione ad ogni aggiornamento, anche se quest'ultimo riguarda solamente un modulo;
- Un bug in un modulo potrebbe causare il crash dell'intera applicazione;
- L'adozione di nuove tecnologie risulta ardua poiché potrebbe richiedere un cambiamento da effettuare nell'intera applicazione.

Architettura a microservizi e i suoi principi L'esigenza di mitigare questi inconvenienti ha determinato la ricerca della così detta *separation of concerns*, che in italiano può essere tradotta come separazione delle preoccupazioni o compiti, e quindi alla nascita del pattern architetturale a microservizi. Per separation of concerns si intende l'abilità di decomporre e organizzare un sistema in moduli logicamente coesi (*cohesive*) e poco dipendenti tra di loro (*loosely-coupled*), i quali nascondono la propria implementazione l'uno dall'altro e espongono dei servizi tramite delle interfacce ben definite. Detto ciò, un'architettura a microservizi può essere descritta come un'architettura che decompone un dominio di business in piccole unità, ovvero i microservizi, delimitate in modo coerente e il più indipendenti tra loro.

Un esempio è quello mostrato in Figura 2.1, la quale rappresenta i microservizi di una applicazione web nell'ambito dell'e-commerce. È evidente che non è un singolo componente software ad implementare l'intera applicazione ma vi è una suddivisione in più componenti, ognuno dei quali offre delle specifiche funzionalità all'esterno. *Order* si occupa della gestione degli ordini, *Payment* dei pagamenti, *Customer* delle informazioni dei clienti e così via. Ogni microservizio è indipendente dagli altri e nasconde la propria implementazione. Allo stesso tempo è possibile che alcuni microservizi abbiano bisogno di comunicare tra loro per scambiarsi informazioni o per delegare parte della richiesta

proveniente da un client. Gli archi presenti nella figura stanno a rappresentare queste eventuali interazioni. Va notato che nella realtà un'applicazione e-commerce spesso richiede molti più microservizi di quelli illustrati nell'esempio.

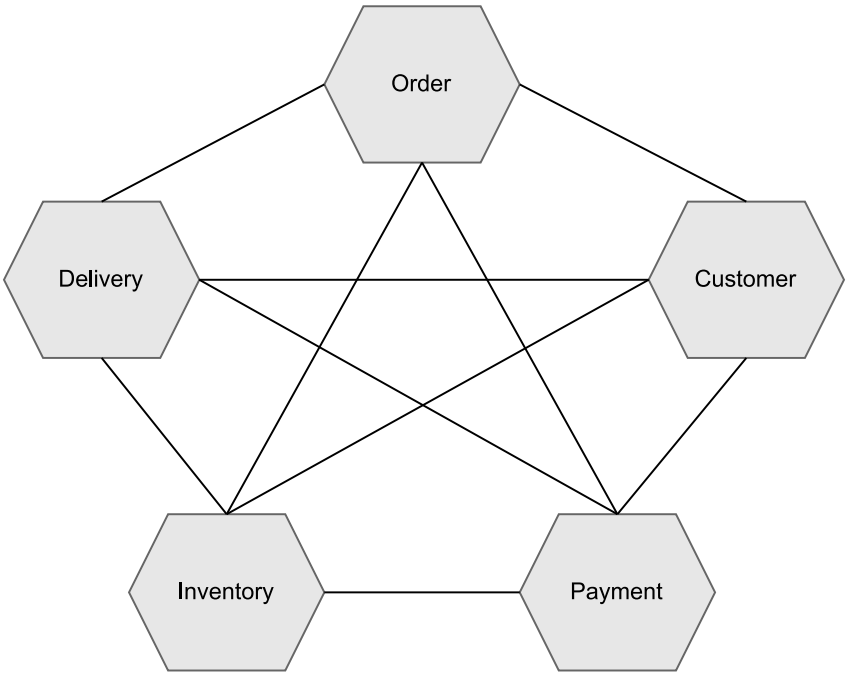

Figura 2.1. Architettura a microservizi di un'applicazione web e-commerce

L'architettura a microservizi prevede una serie di principi:

- *Autonomous services*: questa principio si può decomporre in due sotto proprietà dei microservizi: self-contained, cioè auto-contenuti, e independently deployable, ovvero distribuibili indipendentemente. La prima indica che ogni servizio contiene tutto ciò di cui ha bisogno per funzionare correttamente, inclusi dati, librerie e dipendenze; la seconda denota che ogni servizio deve essere implementabile e distribuibile indipendentemente dagli altri;
- *High cohesion and low coupling*: ogni microservizio deve implementare delle funzioni strettamente correlate tra loro (high cohesion) e deve dipendere poco da altri servizi (low coupling). In questo modo, eventuali cambiamenti da apportare al codice sono confinati ad un singolo microservizio e non influenzano il funzionamento di altri microservizi;
- *Single responsibility per service*: ogni servizio deve avere una singola responsabilità, dunque non deve verificarsi che due servizi condividano la stessa responsabilità né che un servizio ne abbia più di una. Aderendo a questo principio si riduce la complessità dei singoli servizi e, di conseguenza, la difficoltà nell'effettuare modifiche;

- *Common Closure Principle*: proviene dal Object Oriented Design e dice che le cose che cambiano insieme dovrebbero essere raggruppate insieme in, modo tale da garantire che ogni modifica influenzi un singolo servizio;
- *Design to failure*: ogni microservizio deve essere progettato in modo cercando di evitare che un suo guasto impatti il funzionamento degli altri microservizi, garantendo così un alto livello di availability dell'applicazione;
- *Build around business capabilities*: ogni servizio deve essere responsabile di una business capability specifica e tutti i servizi insieme devono coprire tutte le business capability necessarie per l'applicazione. Facendo ciò, i servizi ottenuti avranno una dimensione ridotta e risulterà semplice introdurre cambiamenti. In aggiunta, in caso di guasto di un servizio le altre business capability saranno comunque soddisfatte;
- *Data decentralization*: ogni servizio deve avere il proprio meccanismo di storage, non condiviso con altri servizi. Benché seguire questo principio possa generare ridondanza di informazioni in database differenti, provvede a mantenere la proprietà di low coupling. Inoltre ogni servizio può usare il tipo di database che meglio si adatta alle sue esigenze;
- *Deployment process automation*: deve essere automatizzato il processo di deployment dei vari servizi cosicché gli sviluppatori possano concentrarsi su aspetti di maggior rilievo riguardanti lo sviluppo;
- *Inter-Service Communication*: l'eventuale uso di tecnologie differenti per servizio richiede una progettazione accurata della comunicazione tra di essi, garantendo sempre che la proprietà di incapsulamento sia rispettata;
- *Monitoring*: devono essere impiegati tool di monitoraggio automatici in quanto quello manuale non è fattibile per progetti di una data grandezza;
- *Transportable microservices*: ogni microservizio deve poter essere migrato da un ambiente di esecuzione ad un altro con poco sforzo. Attualmente la soluzione migliore da adottare è l'impiego dei container.

Vantaggi dell'architettura a microservizi Dai suddetti principi è possibile dedurre i vantaggi di tale architettura:

- La decomposizione di un'applicazione complessa in componenti più piccoli rende lo sviluppo, la gestione e il mantenimento più semplici rispetto ad un'architettura monolitica;
- Fintantoché le interfacce dei servizi non cambiano, le modifiche interne al componente risultano più semplici e meno costose da applicare;
- Essendo i microservizi autonomi e poco dipendenti dagli altri, possono essere sviluppati con tecnologie differenti;

- Per lo stesso motivo, vi è una maggiore resilienza ai guasti;
- Incoraggia lo sviluppo agile poiché i cambiamenti sono più semplici da applicare e deployare in tempi brevi, in base alle esigenze del cliente. Oltretutto, permette ai diversi team di lavorare in modo indipendente su microservizi distinti, minimizzando la comunicazione tra team (inter-team communication), spesso causa di rallentamenti;
- Maggiore naturalezza nello scalare l'applicazione in base al carico di lavoro. Più precisamente, è possibile scalare i singoli servizi in modo indipendente dagli altri e in poco tempo, grazie alla velocità con cui può essere effettuato il deploy;
- La possibilità di distribuire ogni servizio in modo indipendente consente l'utilizzo della strategia del continuous deployment.

Svantaggi dell'architettura monolitica La maggior parte degli svantaggi dell'architettura a microservizi sono determinati dalla complessità extra dovuta alla sua natura distribuita:

- Depolyment, scaling e monitoring sono complicati e conviene impiegare l'uso di tool per CI/CD[3];
- Necessità di gestire l'IPC (Inter-Process Communication) e scrivere codice per gestire guasti parziali e non. In aggiunta, l'IPC degrada le prestazioni;
- Nel caso in cui un'operazione implica l'aggiornamento di database appartenenti a più microservizi è utile gestire una transazione distribuita o una forma di sincronizzazione dei dati se vi è ridondanza;
- Se occorre attuare dei cambiamenti riguardanti più servizi serve un'attenta pianificazione per effettuare correttamente il deployment;

In conclusione non esiste un pattern architetturale migliore dell'altro. La scelta spesso viene effettuata ponderando le esigenze e gli obiettivi a breve e lungo termine del software e dell'azienda che lo sviluppa, oltre sicuramente alla disponibilità di budget.

2.2 Decomposizione in microservizi

I *decomposition pattern* sono dei modelli molto utili nell'ambito di applicazioni basate su microservizi. Essi vengono utilizzati durante la progettazione di un software o per effettuare la migrazione da un'architettura monolitica ad un'architettura a microservizi di un'applicazione funzionante. Il loro obiettivo è quello di fornire delle linee guida per l'identificazione di microservizi partendo da un dominio di alto livello.

I decomposition pattern più utilizzati sono due:

[3]CI: Continuous Integration, CD: Continuous Deployment. Sono tecniche automatizzate per integrare e distribuire software in modo semplice e veloce.

- *Decomposition by business capability*: si basa su un concetto proveniente dal business architecture modeling, ovvero quello di *business capability*, traducibile come capacità aziendali. Una business capability è ciò che un business fa per generare valore. Ogni azienda ha più business capability, per esempio vendite, servizio clienti, marketing e altre, le quali variano a seguito del settore in cui l'azienda stessa opera. Nell'ambito dello sviluppo software queste business capability possono essere viste come i concetti e le funzionalità di alto livello indispensabili e che generano valore per l'applicazione. Avendo ben chiare quali siano le business capability relative al software che si vuole sviluppare e seguendo questo pattern si ottiene un'architettura a microservizi stabile se le business capability sono relativamente stabili. L'applicazione di questo pattern permette ai team di sviluppo di concentrarsi sul fornire business value invece che unicamente funzionalità tecniche;

- *Decomposition by subdomain*: si basa sui *subdomain* del Domain-Driven Design (DDD). Questo approccio prevede la suddivisione del dominio dell'applicazione in sottoinsiemi più piccoli, chiamati subdomain, ognuno dei quali rappresenta un'area funzionale ben definita. I subdomain possono differenziarsi in tre tipologie:

  - *Core*: sono i subdomain più importanti, essenziali per il business, che corrispondono alla parti più preziose dell'applicazione;
  - *Supporting*: sono i subdomain meno importanti per il business ma dei quali il business non può farne a meno. Questi possono essere implementati internamente o esternalizzati;
  - *Generic*: sono i subdomain di meno valore, talmente generici che spesso vengono implementati acquistando software di terzi poi integrati nell'applicazione.

Benché i due decomposition pattern utilizzino concetti differenti come base di partenza, il risultato finale è pressoché lo stesso e si ottiene facendo corrispondere ad ogni business capability nel primo caso, e ad ogni subdomain nel secondo, un microservizio.

2.2.1 L'algoritmo 3-step microservices decomposition

La progettazione di un'architettura a microservizi non prevede soltanto l'individuazione di questi ultimi ma è necessario definire anche le operazioni che il sistema fornisce e quindi le relative interfacce che i servizi espongono sia all'esterno del sistema sia internamente ad esso.

Chris Richardson ha proposto all'interno del suo libro *Microservices Patterns: With Examples in Java* un algoritmo a tre step per definire un'architettura a microservizi. Lo scopo di Richardson non è fornire un processo da seguire meccanicamente, piuttosto suggerire delle linee guida generiche; difatti questo è un algoritmo che richiede anche un pizzico di creatività e flessibilità.

Le tre fasi del processo sono:

1. Identificazione delle *system operation*, in italiano operazioni di sistema;
2. Identificazione dei *microservizi*;
3. Definizione delle *API* dei servizi ed eventuali collaborazioni.

Prima di affrontare in modo approfondito l'algoritmo nelle sue tre fasi conviene avere un'idea generale di cosa sono i requisiti scritti di un software, in quanto indispensabili per il primo step.

In molti progetti software vengono impiegati dei riquisiti tipici dell'approccio di sviluppo agile (al quale dedicheremo un capitolo), ovvero le *user story* e le *acceptance criteria*. Entrambe sono scritte seguendo un formato ben preciso; facendo ciò sono comprensibili da tutti i membri facenti parte del progetto e allo stesso tempo dal cliente finale.

Le user story sono delle specifiche generali del software, scritte dal punto di vista di un utente, che descrivono quali funzionalità devono essere implementate e perché. L'utilizzatore del software è posto al centro dell'attenzione e vien data più importanza alla funzionalità e al valore che essa genera rispetto ai soli dettagli tecnici. Lo scopo è quindi quello di descrivere come un progetto software fornirà valore all'utente finale e con quale priorità una funzionalità deve essere sviluppata rispetto alle altre. Una lista di user story viene condivisa all'interno del team di sviluppo che decide in seguito come implementarle dal punto di vista tecnico.

Le acceptance criteria, invece, si immergono più a fondo nei dettagli tecnici, e sono un insieme di condizioni che il software deve rispettare per essere accettato dall'utente finale o dal cliente. Possono essere viste come dei dettagli aggiuntivi relativi ad una user story; più precisamente come delle condizioni che devono essere rispettate per considerare una user story come completa, in inglese chiamata *definition of done*. Oltre a ciò, possono essere facilmente tradotte in casi di test manuali o automatici.

Di seguito la descrizione dei tre step dell'algoritmo di Richardson. Degli esempi saranno illustrati nel Capitolo 5, tramite l'applicazione dell'algoritmo all'interno del progetto svolto in azienda.

1 Identificazione delle system operation Lo scopo del primo step è quello di identificare le *system operation*, le quali possono essere viste come un'astrazione delle richieste che il sistema deve poter gestire.

Prima di poter individuare le system operation, alle quali naturalmente si deve fornire un nome, vi è la necessità di creare un vocabolario comune. Esso è utilizzato dal team di sviluppo durante il progetto, col fine di utilizzare una nomenclatura coerente, e in fase di progettazione, per attribuire i nomi alle system operation. Pertanto, il primo step prevede innanzitutto la creazione di un *high level domain model*, il quale non è altro che un class diagram di alto livello, e il cui scopo è rappresentare le classi principali dell'intero sistema e definire il vocabolario comune di cui abbiamo parlato

poco fa. Per realizzare ciò conviene individuare i nomi all'interno delle user story e delle acceptance criteria e tradurli nelle varie classi che compongono il class diagram.

Avendo a disposizione il class diagram, si può procedere con l'individuazione delle system operation. Anche in questo caso è fondamentale consultare le user story e le acceptance criteria ma, questa volta, analizzando i verbi. Dunque, se per definire il class diagram si usano i sostantivi individuati nei requisiti, in questo caso si utilizzano i verbi. Il nome completo di una system operation è solitamente composto dal verbo e da un complemento oggetto, il quale corrisponde a un'entità dell'high level domain model.

Potrebbe essere d'aiuto schematizzare il risultato ottenuto in più tabelle. La prima elenca le varie system operation e per ognuna di esse aggiunge ulteriori informazioni quali l'attore (utente) che può effettuare la richiesta, la user story da cui è derivata e una breve descrizione. Le restanti tabelle, una per ogni system operation, forniscono dettagli più tecnici utilizzabili al momento dell'implementazione, come ad esempio parametri di input e output, precondizioni e post condizioni.

2 Identificazione dei servizi Il secondo step consiste nell'applicare uno dei due decomposition pattern di cui abbiamo già discusso, decomposition by business capability e decomposition by subdomain. Il risultato rimane invariato, ad eccezione di qualche dettaglio, a seconda del decomposition pattern utilizzato e comprende una lista dei microservizi.

3 Definizione delle API Il terzo e ultimo step riguarda la definizione delle API di ogni servizio. Queste si ottengono associando ad ogni microservizio un'operazione di sistema. In aggiunta si indicano eventuali collaborazioni tra più servizi se necessarie per completare una richiesta. Anche in questo step il risultato può essere schematizzato in una tabella.

2.3 Data management

Durante lo sviluppo di un'applicazione, è essenziale tenere in considerazione la gestione dei dati. È buona prassi valutare quali proprietà, come l'affidabilità o la coerenza, sono prioritarie per i dati gestiti dall'applicazione.

Alcune delle proprietà più basilari e importanti sono:

- *Data persistence*: la persistenza dei dati è la capacità di conservare e mantenere i dati nel tempo, anche dopo che il processo che li ha creati è terminato. In altre parole, i dati devono essere scritti su una memoria non volatile;

- *Data integrity*: l'integrità dei dati si riferisce all'accuratezza e alla validità dei dati durante tutto il loro ciclo di vita. L'integrità dei dati può essere compromessa in diversi modi, per esempio ogni volta che i dati vengono replicati o trasferiti; essi dovrebbero rimanere intatti e non alterati tra gli aggiornamenti. Generalmente

vengono impiegate procedure di validazione e controllo degli errori per assicurarsi che i dati siano integri in ogni momento;

- *Data availability*: la disponibilità dei dati cerca di assicurare l'accessibilità ai dati quando necessario, nonostante possibili interruzioni, guasti hardware o situazioni di emergenza. Per garantire questa proprietà si possono usare diverse tecniche in combinazione tra esse come ad esempio la replicazione dei dati, sistemi di backup e ripristino, tolleranza ai guasti ed altre ancora;
- *Data consistency*: la data consistency si riferisce alla garanzia che i dati all'interno di un sistema siano sempre allineati e, di conseguenza, quando letti, riflettano sempre il valore più recente.

Di solito, è consigliabile valutare con attenzione quale livello di ciascuna proprietà garantire, tenendo conto dell'architettura software, del contesto operativo dell'applicazione e delle risorse disponibili per l'implementazione.

Attualmente, i DBMS[4] relazionali eseguono transazioni ACID, le quali sono essenziali per garantire sia persistenza sia integrità dei dati, ma per ottenere un alto livello di availability e consistency sono necessarie soluzioni aggiuntive, specialmente in ambienti distribuiti complessi.

ACID proviene dall'acronimo Atomicity, Consistency, Isolation, Durability che rappresentano le proprietà logiche che caratterizzano una transazione. Di seguito delle brevi definizioni:

- *Atomicity*: una transazione deve essere un'unità indivisibile. Tutte le operazioni all'interno di una transazione vengono eseguite con successo o nessuna di esse viene eseguita. Nel caso in cui anche solo una singola operazione fallisce si esegue il rollback, annullando l'intera transazione;
- *Consistency*: assicura che dopo il completamento di una transazione il database si trovi in uno stato consistente, rispettando tutte le regole di integrità e vincoli definiti. Una transazione non può portare il database da uno stato consistente a uno inconsistente;
- *Isolation*: garantisce che l'esecuzione simultanea di più transazioni non interferisca con il risultato complessivo. Le modifiche apportate da una transazione devono essere isolate dalle altre transazioni fino a quando non vengono completate. Questo evita effetti indesiderati dovuti all'interferenza tra transazioni concorrenti;
- *Durability*: assicura che una volta completata con successo, una transazione persista anche in caso di guasti del sistema o interruzioni di alimentazione. Le modifiche apportate dal commit di una transazione devono essere permanentemente salvate nel database, garantendo che i dati non vadano persi e rimangano disponibili anche dopo un riavvio del sistema o un guasto hardware.

[4]DBMS è l'acronimo di Database Management System, ovvero il software per la gestione di database.

2.3.1 Data management in architettura monolitica

Nel contesto di un software monolitico, l'approccio comune consiste nell'utilizzare un singolo database per gestire l'intera applicazione. I moduli e le funzionalità condividono lo stesso schema del database e accedono ai dati tramite le medesime tabelle. La valutazione richiesta durante questo processo risulta quindi abbastanza semplice e si concentra principalmente sulla scelta del database da impiegare.

Il DBMS, mediante le transazioni ACID, assicura l'integrità e la persistenza dei dati. Inoltre, in questo specifico scenario, assicura anche la coerenza dei dati, poiché tutte le informazioni risiedono in un singolo database. È facilmente comprensibile che in questo caso non vi è garanzia di un elevato livello di data availability, che si potrebbe ottenere replicando i dati in vari database. La replicazione dei dati causa inevitabilmente la necessità di sincronizzare i vari database ogni qual volta un'operazione di write, update e delete venga effettuata su un database. Il modo in cui questa sincronizzazione viene effettuata determina il livello di data consistency finale.

Quest'ultimo aspetto sarà esaminato più approfonditamente nel prossimo paragrafo, poiché si tratta di una situazione più comune all'interno di un'architettura a microservizi, che è intrinsecamente distribuita.

2.3.2 Data management in architettura a microservizi

Contrariamente all'approccio monolitico, l'architettura a microservizi spinge all'adozione del *database per service* pattern, in cui ogni servizio ha il proprio database, indipendente dagli altri, responsabile per la gestione dei dati specifici del servizio. Evitando di introdurre dipendenze dirette tra microservizi si mantiene un alto livello di autonomia e indipendenza tra essi, obiettivo cardine dell'architettura stessa. Ulteriori vantaggi sono quelli di poter scegliere la tipologia di database più adatta per ogni servizio e di ridurre le comunicazioni tra i vari team di sviluppatori.

In un'architettura distribuita del genere, i dati godono di un alto livello di disponibilità e scalabilità poiché ogni microservizio ha il proprio ambiente di esecuzione e il proprio data store. Tuttavia, emergono complicazioni nella gestione dei dati, specialmente riguardo le transazioni e la coerenza di dati eventualmente duplicati su più database. Come già discusso nel precedente paragrafo, in un'applicazione monolitica un singolo database condiviso garantisce la data consistency tramite le transazioni ACID. D'altra parte, in un'architettura a microservizi, ogni servizio ha un suo database, eventualmente di tipologia diversa, dunque non esiste una singola unità di lavoro che possa eseguire transazioni ACID sulla totalità dei dati. L'applicazione però, generalmente, richiede comunque le proprietà ACID e di conseguenza la data consistency.

I due principali approcci per risolvere tale problema sono:

- Distributed transactions;
- Eventual consistency.

Distributed transactions Tradotto in italiano come transazioni distribuite, è una tecnica per eseguire delle transazioni su più risorse. L'integrity e la consistency dei dati su più database sono garantite grazie ad un coordinatore centrale che gestisce le transazioni distribuite.

Esiste un protocollo bloccante chiamato Two-Phase Commit (2PC), il quale assicura che tutte le transazioni coinvolte in una transazione distribuita abbiano esito positivo o, in caso contrario, tutte falliscano. Il processo è complesso e bloccante, ciò provoca alta latenza e scarso throughput[5]. Inoltre il coordinatore è un single point of failure[6].

Senza ombra di dubbio, utilizzando questo metodo, si ottiene un alto livello di data consistency, a discapito però della data availability. Perciò non è consigliabile usarlo per casi in cui i dati in questione devono essere disponibili sempre ed in tempi brevi. Spesso in applicazioni basate su un'architettura a microservizi si tende ad accettare un livello più basso di data consistency per assicurare un alto livello di availability dei dati, da ciò il termine eventual consistency, cioè il secondo approccio.

Eventual consistency In italiano, coerenza o consistenza eventuale, è un approccio utilizzato nei sistemi distribuiti per garantire un elevato livello di data availability. In un sistema che adotta l'eventual consistency si accetta che i dati possano essere non allineati per un breve periodo di tempo, finché, appunto, tutte le repliche non convergono ad uno stato in cui condividono lo stesso valore. Questo consente ai dati di essere sempre disponibili all'utente finale benché possa capitare che esso non legga il valore più aggiornato. In questa situazione, è responsabilità dello sviluppatore informare chiaramente l'utente nel momento in cui il dato raggiunge il valore più recente.

L'eventual consistency rappresenta un compromesso che garantisce un elevato livello di data availability a scapito della data consistency, la quale viene garantita solo dopo un breve periodo di tempo.

Il SAGA pattern è un modello comunemente usato per implementare l'eventual consistency. Nel SAGA la transazione distribuita viene compiuta eseguendo diverse transazioni locali asincrone su tutti i microservizi interessati. La natura asincrona di questa tecnica permette ai vari microservizi di non bloccarsi, garantendo un livello elevato di data availability. Allo stesso tempo una transazione asincrona distribuita non si assicura che i dati vengano aggiornati immediatamente dai vari microservizi, causando inevitabilmente un decremento della data consistency.

Le implementazioni del SAGA pattern più comunemente utilizzate sono:

- Choreography-based SAGA: il microservizio responsabile dell'aggiornamento del dato produce un evento. Quest'ultimo viene gestito da un message broker che inoltra il messaggio a tutti i microservizi interessati. Questi ultimi provvedono a

[5]Per throughput si intende la capacità trasmissiva effettiva di un canale.

[6]In italiano, singolo punto di vulnerabilita. Si tratta di un componente di un sistema il cui malfunzionamento può causare il fallimento o il blocco dell'intero sistema.

consumare il messaggio e a reagire correttamente, eventualmente apportando le giuste modifiche alla replica del dato interessato nel proprio database;

- Orchestration-based SAGA: in questo caso vi è un servizio che ricopre il ruolo di coordinatore. Esso gestisce la sequenza di transazioni che devono essere eseguite dai vari microservizi ed attua azioni di compensazione, se necessarie.

2.4 Comunicazione tra microservizi

Quando si ha a che fare con un cluster di microservizi è spesso necessario stabilire una collaborazione tra di essi. Questa collaborazione può essere richiesta per garantire un adeguato livello di data consistency, come descritto nella sezione dedicata al data management, oppure per soddisfare una richiesta proveniente da un client. Difatti, può accadere che un client invii una richiesta ad un singolo microservizio, il quale potrebbe non essere in grado di completarla senza delegare ulteriori azioni ad altri microservizi.

Come sappiamo, un software monolitico viene eseguito come un unico processo, pertanto tutte le comunicazioni tra i vari componenti e moduli avvengono all'interno di tale processo. Si parla perciò di *in-process communication*. Nel contesto di un software a microservizi, al contrario, ogni microservizio è un processo a se stante e la comunicazione tra essi viene chiamata *inter-process communication*.

Il primo aspetto rilevante che può essere analizzato tra i due tipi di comunicazione sono le prestazioni che, logicamente, sono molto diverse.

Nel caso della in-process communication, due componenti software possono collaborare insieme e scambiarsi informazioni invocando l'uno i metodi dell'altro. La chiamata di un metodo all'interno dello stesso processo viene ottimizzata in diversi modi dal compilatore e dalle librerie di runtime per ridurre drasticamente l'impatto sul tempo di esecuzione. Inoltre, il passaggio di parametri, soprattutto nel caso di strutture dati complesse, avviene passando un puntatore al dato originale piuttosto che effettuando una copia dei dati. Ciò permette di risparmiare tempo e spazio; il tempo che sarebbe stato utilizzato per copiare i dati, rispetto ad un singolo puntatore, e lo spazio in memoria per memorizzare la copia stessa.

Per quanto riguarda la inter-process communication, e più specificamente la comunicazione tra due microservizi, è ragionevole prevedere che il costo in termini di tempo è significativamente più alto. Considerando due microservizi in esecuzione su due host differenti, il primo fattore che determina il degrado delle prestazioni è la rete. I dati scambiati, infatti, devono essere trasmessi sotto forma di pacchetti attraverso una connessione via cavo, il quale introduce uno specifico Round Trip Time, ovvero il tempo che impiega un pacchetto di dati a viaggiare dal mittente al destinatario sommato al tempo impiegato dal pacchetto di risposta per tornare indietro. Oltre a questo, i dati devono essere serializzati in un formato adatto ad essere trasmesso in rete e, arrivati a destinazione, devono essere deserializzati. Il tempo impiegato per serializzare e deserializzare dei dati dipende dalla loro grandezza.

Si deduce quindi che la inter-process communication introduce ulteriore utilizzo di risorse, quali il tempo e lo spazio, degradando nettamente le prestazioni. Inoltre, agli sviluppatori è richiesto uno sforzo extra nell'affrontare situazioni complesse, come la gestione degli errori, che nel contesto dell'inter-process communication potrebbe non risultare così immediata come sembra.

Soffermandoci sulla comunicazione tra microservizi, è bene dare delle brevi definizioni prima di approfondire i concetti nei paragrafi successivi. Innanzitutto, è importante notare che esistono due modelli principali riguardanti la comunicazione tra microservizi:

- *Comunicazione sincrona*: un microservizio invia un messaggio ad un altro microservizio e rimane in attesa, bloccando il proprio flusso di esecuzione, sino a quando non riceve la risposta. Questo tipo di comunicazione è comunemente denominato anche comunicazione sincrona bloccante;

- *Comunicazione asincrona*: un microservizio invia un messaggio ad un altro senza la necessità di attendere una risposta. In altre parole, il mittente prosegue la sua esecuzione senza essere bloccato, mentre il messaggio viene consegnato ed elaborato dal destinatario. La ricezione di una risposta dipende dalle esigenze specifiche della comunicazione e può anche non essere prevista. Questo tipo di comunicazione è nota anche come comunicazione asincrona non bloccante.

Questi due modelli costituiscono la base per i diversi stili di comunicazione. Tra i più importanti vi sono:

- *Request-response*: un microservizio invia una richiesta ad un altro microservizio chiedendo che venga eseguita un'azione e si aspetta di ricevere una risposta che lo informi del risultato;

- *Event-driven*: i microservizi emettono eventi, che altri microservizi consumano reagendo di conseguenza. Il microservizio che emette l'evento non sa quali altri microservizi, se ce ne sono, consumano gli eventi che esso emette;

- *Common data*: i microservizi collaborano tramite una qualche fonte di dati condivisa.

Nella fase di progettazione di un sistema basato su microservizi, è cruciale considerare quali stili di comunicazione meglio si adattano ai requisiti del sistema e poi selezionare le tecnologie più adeguate per implementarli. Inoltre, è importante notare che spesso è comune utilizzare diversi stili di comunicazione all'interno dello stesso gruppo di microservizi. Ad esempio, alcune interazioni possono seguire lo stile di comunicazione request-response, mentre altre possono adottare lo stile event-driven. La presenza di più tecnologie di comunicazione non è problematica se è il risultato di una scelta ben ponderata e giustificata.

2.4.1 Comunicazione sincrona

Come menzionato in precedenza, nella comunicazione sincrona bloccante, un microservizio invia una richiesta ad un altro microservizio e rimane in attesa di una risposta, prima di proseguire con ulteriori operazioni.

Questo modello di comunicazione risulta utile sia quando è necessario utilizzare le informazioni contenute nella risposta per le operazioni successive, sia quando si desidera ricevere un feedback che indichi l'esito positivo o negativo dell'operazione richiesta. Se uno di questi scenari è rilevante per i requisiti dell'interazione desiderata, l'adozione dello stile di comunicazione request-response diventa spesso una scelta naturale.

Prima di esplorare in dettaglio lo stile di comunicazione request-response sincrona, è opportuno analizzare gli svantaggi associati al modello di comunicazione sincrona.

La principale sfida di questo approccio risiede nella creazione di una dipendenza temporale tra il microservizio mittente e quello destinatario. Se il microservizio destinatario non è accessibile, il mittente deve gestire tecniche di retry o timeout. In caso contrario, se il mittente diventa irraggiungibile dopo l'invio della richiesta, la risposta potrebbe andare persa. In questo caso, è conveniente che una replica del microservizio mittente sia in grado di gestire la risposta. Inoltre, nel caso in cui il servizio destinatario è lento nell'elaborazione, il mittente potrebbe rimanere bloccato per un periodo di tempo non accettabile.

Nella pratica, queste situazioni non generano complicazioni in architetture a microservizi relativamente semplici. Tuttavia, la criticità sorge quando si verificano catene di chiamate tra diversi microservizi.

L'esempio illustrato in Figura 2.2 nel contesto di un'applicazione e-commerce chiarisce meglio il concetto. I quattro microservizi rappresentati collaborano tra loro per servire il completamento di un'ordine, accertandosi però che non ci sia rischio di frode. Il microservizio *Order* invia una richiesta al servizio *Payment* per eseguire il pagamento. Esso però può concludere il pagamento soltanto se non vi è rischio di frode, quindi contatta il microservizio *Fraud detection* che, a sua volta, invia una richiesta al microservizio *Customer*, in quanto ha bisogno dei dati del cliente per eseguire il controllo. Dato che le varie interazioni sono sincrone, ogni microservizio aspetta una risposta dal successivo. Per esempio, Fraud detection può completare il check solo dopo aver ricevuto i dati dal microservizio Customer; dopo aver ricevuto la risposta esegue il controllo antifrode e risponde al servizio Payment, che nel frattempo è rimasto bloccato in attesa di un riscontro. Si può dedurre che i microservizi a monte mantengono una connessione aperta per un tempo prolungato finché le varie risposte non sono giunte a destinazione facendo collassare la catena di chiamate. Oltretutto, in caso di guasti riguardanti un microservizio o una connessione tra due microservizi, l'intera comunicazione potrebbe fallire.

Per mitigare questo problema senza adottare il modello asincrono, una possibile soluzione potrebbe essere quella di ridurre la lunghezza della catena di chiamate. In questo specifico caso si potrebbe togliere dal flusso principale di comunicazione il servizio Fraud detection, facendo eseguire il controllo antifrode in parallelo non appena

la richiesta arriva da un client. Facendo ciò, si riduce il numero di connessioni aperte contemporaneamente e di dipendenze. Allo stesso tempo si riduce la probabilità che un guasto possa influenzare il successo dell'intera comunicazione.

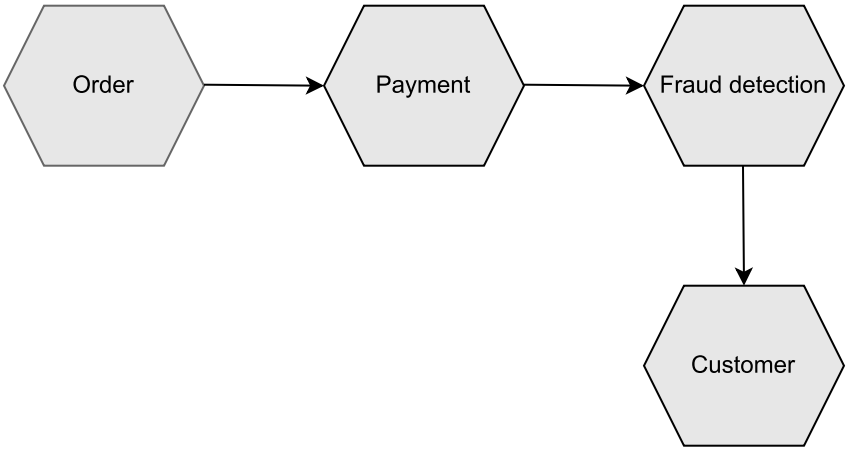

Figura 2.2. Catena di chiamate sincrone

Analizziamo ora lo stile request-response sincrono.

Request-response La modalità di comunicazione nota come request-response sincrona è ampiamente utilizzata, specialmente quando le interazioni sono di breve durata e richiedono una risposta contenente dati richiesti o l'esito dell'operazione in corso.

Questo approccio implica l'apertura di una connessione tra il microservizio mittente e il destinatario. Il mittente invia una richiesta e rimane in attesa di una risposta, mantenendo la connessione aperta fino al suo arrivo. Da notare che il microservizio responsabile dell'invio della risposta non ha bisogno di sapere a quale microservizio inviarla, poiché la invierà semplicemente sulla stessa connessione dalla quale ha ricevuto la richiesta. Tuttavia, se la connessione dovesse interrompersi prima del completamento dell'interazione, si potrebbero verificare complicazioni significative.

Naturalmente, questo stile di comunicazione eredita gli svantaggi associati alla comunicazione sincrona, derivanti dalla dipendenza temporale tra mittente e destinatario.

Nel paragrafo successivo, esamineremo anche l'approccio request-response asincrono, che affronta completamente questa dipendenza temporale mediante l'uso di code di messaggi.

2.4.2 Comunicazione asincrona

Riprendendo la definizione di comunicazione asincrona, il microservizio che invia la richiesta iniziale non interrompe il proprio flusso di esecuzione per attendere la risposta. Anzi, esso può procedere tranquillamente con le operazioni successive.

La comunicazione asincrona porta con se diversi vantaggi dato che non vi è dipendenza temporale tra i due microservizi. Grazie a questo, il destinatario non deve essere necessariamente raggiungibile nel momento in cui viene inviata la richiesta.

L'approccio asincrono è ideale per richieste che prevedono task di lunga durata da parte del microservizio destinatario, dato che il mittente non deve rimanere in attesa per tutto il tempo. Oltre a ciò, è molto utile per risolvere il problema delle catene di chiamate esposte nel paragrafo precedente.

Il principale svantaggio emerge dalla diversificazione delle tecnologie utilizzate per implementare i vari stili di comunicazione asincrona; infatti, a differenza del modello sincrono, essa porta a una molteplicità di stili comunicativi, tra cui il request-response asincrono, l'event-driven e il common data, ognuno dei quali è stato sviluppato nel tempo da diverse soluzioni tecnologiche. Pertanto, è necessario acquisire una conoscenza approfondita di tali opzioni e selezionare quella più adatta alle proprie esigenze.

Ora passiamo ad esaminare più approfonditamente i tre stili di comunicazione precedentemente menzionati: il common data, il request-response asincrono e l'event-driven.

Common data Il common data rappresenta il più semplice tra i tre stili di comunicazione asincrona menzionati. In questo approccio, un microservizio inserisce informazioni in una locazione condivisa e successivamente uno o più microservizi le utilizzano dopo averle recuperate.

Per implementare questo stile di comunicazione è necessario disporre di una locazione di memoria persistente, ad esempio il filesystem o un database.

Risulta essere un pattern molto semplice da implementare in quanto le tecnologie necessarie per farlo sono ampiamente conosciute e stabili. Inoltre, è ideale per muovere grandi volumi di dati, ad esempio file molto pesanti. Al contrario, è caldamente sconsigliato in applicazioni real-time poiché il microservizio destinatario utilizza un meccanismo di polling per verificare se i dati sono stati memorizzati o meno. Ciò comporta ritardi tra la scrittura dei dati da parte del mittente e la loro lettura da parte del destinatario.

Request-response L'obiettivo del request-response asincrono è eliminare la limitazione legata al temporal coupling tipica della controparte sincrona. In questo approccio, invece di stabilire direttamente una connessione con il destinatario e inviare la richiesta, il microservizio mittente trasmette la richiesta sotto forma di messaggio a un *message broker*. Il message broker inserisce il messaggio in una coda di messaggi, in inglese *message queue*. Il microservizio destinatario consuma continuamente i messaggi presenti nella coda, processandoli uno alla volta. Successivamente, la risposta viene accodata in una coda separata, dalla quale sarà il mittente a consumare i messaggi.

La presenza di due code, una per le richieste e l'altra per le risposte, permette di spezzare la dipendenza temporale presente nella tipica interazione sincrona. Oltre a questo, anche un destinatario lento non si troverà mai inondato di richieste dato che esse sono mantenute all'interno della coda e consumate una alla volta.

Come per lo stile sincrono, anche in questo caso è utile usufruire della request-response se è necessario un dato richiesto o un feedback rappresentate l'esito del task effettuato.

Dall'esempio illustrato nella Figura 2.3, è evidente come il microservizio *Order* inserisce una richiesta nella coda dei messaggi per la prenotazione dei prodotti di un ordine. Il servizio *Warehouse*, noto anche come Magazzino, estrae e elabora i messaggi in coda finché non raggiunge il messaggio attuale. Dopo aver riservato le quantità necessarie, risponde creando un messaggio di risposta e lo inserisce in una coda di messaggi separata, destinata esclusivamente alle risposte di questo tipo di interazione.

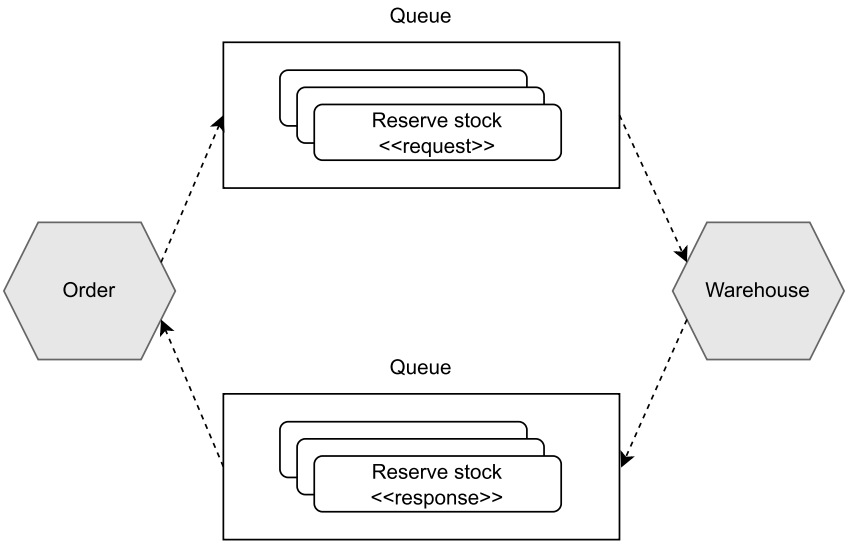

Figura 2.3. Interazione request-response asincrona

Event-driven Lo stile di comunicazione event-driven differisce notevolmente dal modello request-response e, in alcuni aspetti, adotta un approccio opposto, rendendolo potenzialmente meno intuitivo.

In questo contesto, anziché inviare una richiesta diretta a un destinatario, un microservizio emette un *evento*. Tale evento può essere consumato da uno o più microservizi, o addirittura da nessuno, in alcuni casi. Un evento rappresenta un'affermazione riguardo qualcosa che è accaduto, di solito nel contesto del microservizio che lo ha emesso. Il microservizio che emette l'evento non è a conoscenza del modo in cui le informazioni all'interno dell'evento saranno utilizzate, né quale microservizio le utilizzerà. Anzi, in generale, non conosce affatto l'esistenza degli altri microservizi. Se in una interazione request-response, il mittente deve conoscere necessariamente il microservizio destinatario e in più anche ciò che esso è in grado di fare, in modo tale da poter inviare una richiesta congrua, in un'interazione event-driven la conoscenza del mittente è ridotta e la sua responsabilità finisce nel momento in cui emette l'evento. Questa caratteristica

è anche denominata *inversion of responsibility* in quanto la responsabilità passa dal microservizio mittente ai vari destinatari che consumano l'evento.

Tutte le caratteristiche proprie di questo stile di comunicazione permettono al sistema di raggiungere un alto livello di low-coupling, proprietà fondamentale di un'architettura a microservizi.

Dal punto di vista implementativo, una soluzione tecnologica ampiamente adottata è quella dei message broker che implementano il pattern *publish-subscribe*. In questa soluzione, alcuni microservizi pubblicano dei messaggi con uno specifico *topic*, un canale logico avente un nome. Tali messaggi incapsulano un evento e sono pubblicati tramite un'interfaccia esposta dal broker. In questo modo tali microservizi assumono il ruolo di publisher per quello specifico topic. Altri microservizi si sottoscrivono a quel topic assumendo il ruolo di subscriber. Il message broker memorizza i vari topic esistenti e quali microservizi sono sottoscritti ad ognuno di essi. Ogni qual volta un microservizio pubblica un messaggio per quel topic, il message broker provvede ad inoltrarlo ai vari subscriber.

Analizzando la Figura 2.4 è possibile osservare come il microservizio *Warehouse* pubblica un evento per il topic *Order status*, utilizzato per segnalare che l'ordine è pronto per la spedizione. I due microservizi a destra della figura, *Notifications* e *Inventory*, essendo sottoscritti a quel topic, ricevono l'evento tramite il message broker e reagiscono di conseguenza. Il primo invia un'email al cliente per comunicare che l'ordine è pronto, mentre il secondo si occupa di aggiornare le quantità relative ai prodotti appena impacchettati. È evidente che il publisher, Warehouse, non interagisce direttamente con i subscriber e nemmeno ne è a conoscenza. Sono i microservizi destinatari che sanno come gestire quel tipo di evento ed eseguire le operazioni necessarie, utilizzando le informazioni contenute in esso.

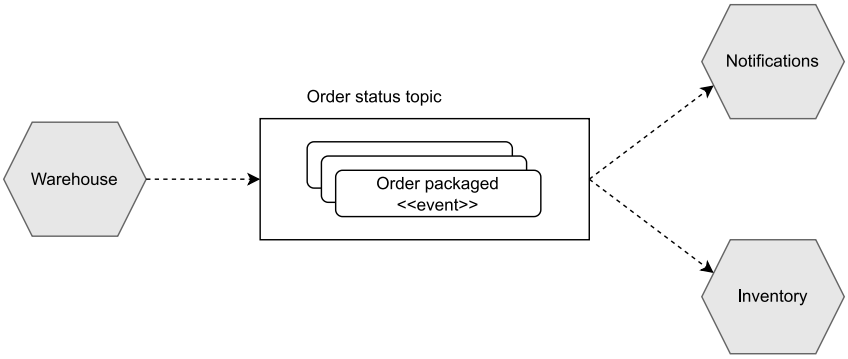

Figura 2.4. Interazione event-driven

In conclusione, l'approccio event-driven è molto conveniente nei casi in cui il focus principale è sviluppare un architettura in cui i diversi microservizi siano il più indipendenti possibili. Tuttavia, è fondamentale considerare attentamente la complessità aggiuntiva introdotta da questa soluzione e le differenze tra le varie implementazioni tecnologiche disponibili.

Capitolo 3

Cloud Computing

Il cloud computing rappresenta un paradigma di distribuzione delle risorse informatiche attraverso Internet, consentendo l'accesso a risorse IT on-demand e scalabili.

Inizialmente, con la sezione *Teoria del cloud*, saranno esaminate le fondamenta teoriche del cloud computing, illustrando i concetti e i modelli architetturali che ne costituiscono la base.

In seguito, nella sezione *Cloud Service Provider*, ci concentreremo sui fornitori di servizi cloud, analizzando le varie opzioni disponibili sul mercato e le caratteristiche distintive di ciascuno.

In conclusione, con la sezione intitolata *Microsoft Azure*, una delle piattaforme cloud leader nel settore, saranno esaminati i tipi di servizi e le funzionalità offerti da Azure, esplorando le sue caratteristiche più imporanti e il suo impatto nel panorama del cloud computing.

3.1 Teoria del cloud

Il concetto di *cloud computing* ha radici che risalgono agli anni '60, principalmente grazie al lavoro svolto dall'informatico statunitense *Joseph Carl Robnett Licklider* durante le ricerche svolte all'interno dell'*ARPA* (*Advanced Research Projects Agency*), un'agenzia del dipartimento della difesa degli Stati Uniti. In questo periodo rilasciò diversi report in cui venivano esposti concetti riguardanti una rete di computer globale, da cui poi è nata ARPAnet, precursore del moderno Internet, e una rete simile all'attuale cloud. Benché egli non avesse mai utilizzato l'espressione cloud computing aveva concepito l'idea di consentire alle persone di accedere ai dati da qualsiasi luogo e in qualsiasi momento, utilizzando i computer.

La moderna implementazione e diffusione di questo principio, invece, è iniziata nel 2006 quando AWS, una divisione di Amazon, ha lanciato i propri servizi di cloud computing. Tuttavia, è importante notare che anche altre aziende stavano sviluppando tecnologie e concetti simili in quel periodo.

Una definizione di cloud computing è stata data in un breve documento dal *NIST* (*National Institute of Standards and Technology*).

Il cloud viene definito come un modello che consente l'accesso ubiquo, comodo e on-demand, tramite una rete, ad un insieme di risorse computazionali configurabili (ad esempio reti, server, storage, applicazioni e servizi) che possono essere rapidamente fornite e rilasciate con un minimo sforzo di gestione ed interazione con il fornitore di servizi.

Sempre all'interno dello stesso documento vengono poi date delle brevi definizioni riguardanti le caratteristiche essenziali del modello cloud, i modelli di servizio e i modelli di distribuzione che analizzeremo nei prossimi paragrafi. Verranno mantenuti i nomi in inglese dei vari concetti appena citati, quindi *Essential Characteristics*, *Service Models* e *Deployment Models*.

Essential Characteristics

- *On-demand self-service*: il consumatore può accedere alle capacità computazionali offerte dal servizio cloud in modo autonomo, senza richiedere interazioni umane con il fornitore del servizio;

- *Broad network access*: le risorse sono disponibili in rete e accessibili tramite protocolli di comunicazione standard. In tal modo il servizio è accessibile da dispositivi eterogenei quali smartphone, tablet, laptop, desktop PC e altri;

- *Resource pooling*: le risorse computazionali del provider sono raggruppate insieme per servire diversi consumatori utilizzando il modello multi-tenant. Questo modello prevede l'accesso allo stesso insieme di risorse da parte di più consumatori in modo indipendente; ogni consumatore non può vedere e accedere alle risorse di un altro. Le risorse vengono assegnate e riassegnate dinamicamente in base alla richiesta dei vari tenant (clienti). Inoltre, in generale, il consumatore non ha controllo sulle risorse né conosce la loro posizione precisa, ma può specificarne la posizione a un livello di astrazione più alto, ad esempio a livello di regione, stato o datacenter;

- *Rapid elasticity*: le risorse possono essere allocate e rilasciate in modo flessibile per adattarsi rapidamente al carico di richieste in corso. Dal punto di vista del cliente le risorse appaiono come se fossero illimitate e possono essere acquisite in qualsiasi quantità in qualsiasi momento;

- *Measured service*: le risorse sono controllate e ottimizzate in modo automatizzato sfruttando tecniche di misurazione diverse per ogni risorsa. Ciò consente di monitorare, controllare e generare report sull'uso delle risorse, fornendo trasparenza sia al fornitore che al consumatore del servizio utilizzato.

Service Models

- *Software as a Service (SaaS)*: si tratta di offrire al cliente un'applicazione funzionante eseguita su un'infrastruttura cloud[1] e accessibile tramite un'interfaccia grafica web, come ad esempio la web-based email di Google, oppure attraverso un'interfaccia di un programma installabile sul proprio dispositivo. Il cliente non ha il controllo sull'infrastruttura cloud sottostante né sulle funzionalità dell'applicazione, fatta eccezione per alcune impostazioni di configurazione specifiche per l'utente;
- *Platform as a Service (PaaS)*: il servizio fornito in questo caso permette di avviare su un'infrastruttura cloud le applicazioni create dal cliente, utilizzando linguaggi di programmazione, librerie, servizi e strumenti supportati dal fornitore. Il consumatore non gestisce o controlla l'infrastruttura cloud sottostante ma ha il controllo sulle applicazioni avviate e, se necessario, sulle impostazioni di configurazione per l'ambiente di hosting delle applicazioni;
- *Infrastructure as a Service (IaaS)*: consiste nel mettere a disposizione del cliente risorse di elaborazione, storage, reti e altre risorse, su cui il cliente può configurare ed eseguire software arbitrario, che può includere sistemi operativi e applicazioni. Anche in questo caso, il cliente non gestisce o controlla l'infrastruttura cloud, ma ha il controllo sui sistemi operativi, lo storage e le applicazioni distribuite ed eventualmente ha anche un controllo limitato su determinati componenti di rete.

Deployment Models

- *Private cloud*: l'infrastruttura cloud è dedicata ad una singola organizzazione di utenti. Questo modello si oppone al tradizionale modello multi-tenant con cui vengono gestite le risorse computazionali. Il private cloud può essere ospitato nel datacenter dell'organizzazione stessa o in un datacenter di terze parti fornito da un private cloud provider. In generale, l'organizzazione è responsabile del corretto funzionamento del sistema come nel caso di una classica infrastruttura on-premise;
- *Community cloud*: si tratta di una soluzione molto simile alla precedente ma in questo caso l'infrastruttura è destinata a utenti provenienti da organizzazioni che condividono obiettivi comuni, come un progetto. Da qui il termine community cloud. Anche in questo caso, l'infrastruttura può essere gestita da una o più organizzazioni della community o affidata a terze parti;
- *Public cloud*: l'infrastruttura cloud è fornita per un utilizzo aperto al pubblico in generale. Essa si trova presso le strutture del fornitore di servizi cloud;

[1] L'infrastruttura cloud è l'insieme di hardware e software che garantisce le cinque caratteristiche essenziale esposte in precedenza. L'infrastruttura può essere divisa in layer fisico e layer logico. Il primo è composto ad esempio dai server, connettività di rete e storage. Il layer logico invece è composto da tutti i componenti software allocati al di sopra del layer fisico, che servono per supportare l'intero sistema e che manifestano le caratteristiche essenziali.

- *Hybrid cloud*: è una composizione di due o più infrastrutture cloud distinte (private, community o public) che rimangono entità uniche, ma connesse attraverso tecnologie standard o proprietarie che consentono la portabilità dei dati e delle applicazioni.

3.2 Cloud Service Provider

I *Cloud Service Provider*, spesso abbreviati con l'acronimo *CSP*, sono aziende o organizzazioni che implementano e forniscono servizi di cloud computing. I CSP gestiscono e mantengono l'infrastruttura necessaria per fornire questi servizi, inclusi datacenter, server, reti e storage. Forniscono anche servizi di sicurezza, monitoraggio, supporto tecnico e aggiornamenti continui per garantire che i servizi siano sicuri, affidabili e in linea con le esigenze degli utenti. I servizi offerti dai provider differiscono in base al modello di servizio, come discusso nella sezione precedente (SaaS, PaaS, IaaS). Inoltre, i CSP possono offrire servizi cloud di tipo pubblico, privato, ibrido o community. I datacenter dei provider sono sparsi per tutto il mondo; in tal modo chiunque, ovunque e in qualsiasi momento può accedere alla piattaforma del vendor ed acquistare le risorse necessarie nella regione che desidera. Il costo varia da servizio a servizio e il pagamento segue solitamente il modello *pay-as-you-go*, ovvero con tariffazione basata sull'utilizzo.

In pratica, i fornitori di servizi cloud (CSP) offrono un'opzione rapida e agevole per effettuare l'*outsourcing* di diverse risorse, a seconda del modello di servizio scelto. Questa scelta porta con sé numerosi vantaggi, sia dal punto di vista produttivo che economico.

Esploreremo qui di seguito tali vantaggi, prendendo in considerazione la prospettiva aziendale:

- grazie all'outsourcing di server, rete, storage e altre risorse, l'azienda non deve sostenere grandi investimenti iniziali riguardanti l'infrastruttura ma paga una bassa tariffa in base al loro utilizzo;
- per lo stesso motivo l'azienda guadagna tempo dovuto all'installazione, al testing e alla manutenzione di tale infrastruttura investendolo maggiormente nell'aspetto produttivo;
- entrare in mercati nuovi o mai esplorati prima, diventa semplice e rapido per un'azienda. Non è necessario sostituire o aggiornare la propria infrastruttura ma basta rilasciare e allocare nuove risorse sulla piattaforma del provider;
- un vantaggio è quello di poter scegliere tra i vari modelli di servizio, ovvero SaaS, PaaS e IaaS che determinano, in pratica, quale livello di outsourcing si desidera raggiungere;
- un'azienda può scegliere tra i quattro modelli di deployment esistenti: private, public, community e hybrid in base alle proprie esigenze;

- il pagamento riguarda solo le risorse utilizzate. Non son previsti pagamenti anticipati o contratti a lungo termine;
- i propri dati così come le proprie applicazioni possono essere facilmente georeplicate in più datacenter per eseguire disaster recovery e per assicurare un elevato livello di availability.

Non bisogna però sottovalutare le eventuali problematiche dovute al cloud computing:

- se con un datacenter on-premise si ha il pieno controllo su tutto, compresi dati sensibili come quelli degli utenti e segreti aziendali, affidarsi a un provider significa accettare inevitabilmente una minore garanzia di confidenzialità in generale;
- anche la sicurezza è un aspetto critico; infatti, utilizzando servizi cloud si trasferiscono dati su Internet, che è una rete pubblica. Inoltre i CSP sono bersagli di attacchi informatici. Questo può causare perdita o compromissione dei dati con conseguente danno d'immagine e reputazionale;
- uno dei problemi più diffusi nel contesto del cloud è il cosiddetto *vendor lock-in*, che rappresenta la difficoltà di effettuare una migrazione da un fornitore di servizi a un altro, se necessario. Questa transizione può risultare lunga e costosa se non si ha già adottato dal principio una strategia di diversificazione delle risorse su più cloud provider.

Il mercato attuale dei CSP è dominato dai cosiddetti *Big Three Cloud Providers*, ovvero *Amazon Web Services* (*AWS*), *Microsoft Azure* e *Google Cloud Platform* (*GCP*) che insieme occupano il 66% del mercato e sono in continua crescita da diversi anni. La restante fetta di mercato è occupata da cloud provider di minor rilievo. Tra i più importanti vi sono Alibaba Cloud, Oracle Cloud, IBM Cloud e Tencent Cloud.

È di grande utilità analizzare attentamente i cloud provider in base a più criteri e successivamente selezionare il più adatto alle proprie esigenze. Alcuni di questi criteri potrebbero essere:

- Modello di Servizio (IaaS, PaaS, SaaS);
- Costi;
- Prestazioni e scalabilità;
- Sicurezza;
- Posizione dei datacenter;
- Servizi aggiuntivi;
- Reputazione e affidabilità;
- Assistenza e supporto;
- Esperienza utente e interfaccia.

3.3 Microsoft Azure

Considerata come seconda solo a AWS, *Microsoft Azure* detiene il 23% del market share. Insieme ad *Amazon Web Services* e *Google Cloud Platform* domina il settore occupandone il 66%.

Nasce nel 2010 con il nome di Windows Azure, offrendo inizialmente servizi per lo sviluppo e la distribuzione di applicazioni web. Nel corso degli anni ha costantemente evoluto e ampliato la sua gamma di servizi, includendo capacità computazionali avanzate, soluzioni orientate all'IoT, implementazioni serverless e, infine, strumenti dedicati all'intelligenza artificiale. Queste migliorie mirate hanno permesso a Microsoft Azure di adattarsi alle esigenze di aziende di diverse dimensioni, garantendo una risposta completa e efficace alle sfide tecnologiche contemporanee.

Tra tutti i cloud provider, Azure è quello che possiede il maggior numero di *region*[2] e *availability zone*[3], garantendo, quindi, un elevato livello di disponibilità e resilienza alle applicazioni distribuite sui propri datacenter.

I servizi sono accessibili tramite un portale web abbastanza semplice da utilizzare o tramite API se risulta necessario integrare un servizio all'interno della propria applicazione. Sono inoltre disponibili tool per il monitoring delle risorse e dei servizi.

I vari servizi offerti dalla piattaforma possono essere raggruppati in più tipologie:

- Servizi di calcolo: si tratta dei servizi che riguardano virtual machine, container e applicazioni web o mobile;
- Servizi di archiviazione: sono tutti quei servizi che possono archiviare e gestire i dati nel cloud come ad esempio Azure Blob Storage;
- Servizi di gestione dati: in questa categoria vi sono tutti i servizi che implementano database relazionali e non;
- Servizi di rete: comprendono tutti quei servizi che permettono di connettere e proteggere le risorse allocate in azure;
- Servizi di sicurezza: forniscono protezione, in particolare autenticazione e autorizzazione;

[2]Una region rappresenta un'area geografica in cui un cloud provider posiziona i propri datacenter. Le region sono posizionate in diverse parti del mondo e consentono ai clienti di distribuire le loro risorse informatiche e applicazioni vicino ai loro utenti, migliorando la latenza e la resilienza.

[3]Una availability zone, in italiano zona di disponibilità, rappresenta una partizione di una region. Ogni region può avere più availability zone e ogni availability zone può essere composta da più datacenter. Esse sono progettate in modo tale che se una zona riscontra un'interruzione energetica o viene affetta da un'evento catastrofico, i servizi, la capacità e l'high availability dell'intera regione sono supportati dalle zone rimanenti. Queste, infatti, sono disposte a distanze tali da escludere che tutte siano danneggiate contemporaneamente. Inoltre dispongono di infrastrutture indipendenti per l'alimentazione, il raffreddamento e la rete. Allo stesso tempo la distanza tra le zone è pensata per garantire una bassa latenza, facilitando la sincronizzazione dei dati replicati nel minor tempo possibile.

- Servizi di intelligenza artificiale: permettono di integrare sia l'intelligenza artificiale sia il machine learning all'interno delle proprie applicazioni;
- Servizi Internet of Things: comprendono i servizi per collegare e gestire dispositivi IoT;
- Servizi per gli sviluppatori: attraverso questi servizi, è possibile sviluppare, implementare e gestire applicazioni nel cloud. Tra i servizi di questa categoria uno tra i più usati é sicuramente Azure DevOps.

In generale, Microsoft Azure offre servizi appartenenti a tutti e tre i modelli di servizio. Per esempio, i vari database presenti nella piattaforma seguono il modello *IaaS* in quanto l'infrastruttura è gestita interamente da Microsoft. Per quanto riguarda il *PaaS*, Azure fornisce un ambiente di lavoro completo e tool di sviluppo all'interno della sua piattaforma, semplificando lo sviluppo e la distribuzione delle applicazioni. Infine, il modello *SaaS* rappresenta il tratto distintivo di Microsoft Azure rispetto agli altri cloud provider. Azure si specializza nel fornire software funzionante e immediatamente utilizzabile, rivolto sia al livello personale che, in particolare, a quello aziendale. Questa specializzazione si traduce in una vasta gamma di soluzioni, comprendenti strumenti per la gestione aziendale, la collaborazione, l'analisi dei dati e molto altro.

Capitolo 4

Processo di sviluppo software

Il processo di sviluppo software è un insieme di attività organizzate e pianificate che guidano la creazione di un'applicazione software, dall'ideazione alla distribuzione e manutenzione. Coinvolge fasi come l'analisi dei requisiti, il design, l'implementazione, il testing e il deployment. Un efficace processo di sviluppo è essenziale per garantire la qualità, l'affidabilità e la riuscita del prodotto software finale.

La sezione *Fasi del processo di sviluppo software* fornisce una panoramica completa delle fasi coinvolte nella creazione di un'applicazione software, dal concepimento alla distribuzione e manutenzione del software.

In seguito, nella sezione dedicata ai *Requisiti software*, l'attenzione si focalizza sulla descrizione dei diversi tipi di requisiti e sul processo attraverso il quale possono essere derivati da requisiti di più alto livello.

La sezione *Sviluppo agile del software*, invece, descrive in maniera sintetica e coincisa l'approccio di sviluppo agile, mettendo in luce i valori e i principi fondamentali su cui si fonda.

In conclusione, nell'ultima sezione dal titolo *Il framework Scrum*, viene presentato uno dei framework agile più diffusi in ambito lavorativo, ossia il framework Scrum, delineando la struttura del team, gli eventi e gli artefatti che lo caratterizzano.

4.1 Fasi del processo di sviluppo software

Il processo di sviluppo software si potrebbe suddividere in diverse fasi:

1. *Selezione di una metodologia di sviluppo*: consiste nel selezionare un framework per la gestione del ciclo di sviluppo software. Tra i più utilizzati e conosciuti vi sono Waterfall, Agile, Rapid Application Development ed altri;

2. *Raccolta dei requisiti*: si tratta di raccogliere i vari requisiti del software coinvolgendo i vari *stakeholder*. Una possibile traduzione di stakeholder è *parte interessata*, ovvero una persona o un gruppo di persone che ha un interesse, coinvolgimento o influenza su un particolare progetto, processo o iniziativa;

3. *Scelta dell'architettura*: avendo a disposizione i requisiti di alto livello è possibile scegliere su quale architettura il software verrà sviluppato e quindi eseguito;

4. *Sviluppo di un progetto*: partendo dai requisiti di alto livello e dall'architettura selezionata è possibile progettare nei dettagli tutti gli aspetti rilevanti del sistema creando eventualmente modelli, diagrammi e documentazione in generale;

5. *Implementazione del codice*: è semplicemente la fase di sviluppo del codice;

6. *Testing*: potrebbe essere inclusa nella fase di sviluppo del codice in base alla metodologia selezionata in principio. Riguarda la scrittura e l'esecuzione di casi di test derivanti solitamente da requisiti funzionali e non funzionali;

7. *Distribuzione*: è il processo in cui il software funzionante viene distribuito agli utenti finali o agli ambienti operativi in cui sarà utilizzato, rendendolo accessibile e fruibile in modo appropriato;

8. *Manutenzione del software*: prevede di apportare modifiche, miglioramenti, correzioni e aggiornamenti al software distribuito per garantire che continui a funzionare in modo efficace, efficiente e sicuro nel corso del tempo.

Nei paragrafi successivi, esploreremo in dettaglio i concetti dei requisiti software, concentrandoci attentamente sui vari tipi di requisiti, sulle loro differenze e sugli scopi per i quali vengono introdotti durante la fase di progettazione.

4.2 Requisiti software

I *requisiti software* rappresentano specifiche e condizioni necessarie che definiscono le funzionalità, le prestazioni, gli standard e le restrizioni del software. Queste specifiche derivano dalle esigenze degli utenti finali e degli stakeholder. Avere dei requisiti chiari ed esaustivi è un elemento fondamentale per la riuscita di un progetto software.

Esistono diversi tipi di requisiti, con l'obiettivo comune di ottenere un prodotto finale, entro le scadenze previste, con funzionalità e qualità che rispettino le aspettative. In generale, è possibile individuare principalmente tre tipologie di requisiti:

- *Requisiti di business*
- *Requisiti utente*
- *Requisiti del prodotto*

Nella Figura 4.1 sono rappresentate le tre tipologie di requisiti appena menzionate, disposte in sequenza da sinistra a destra. Si nota che i *product requirements* si suddividono ulteriormente in *functional requirements* (requisiti funzionali) e *non-functional requirements* (requisiti non funzionali). Lo scopo principale di questo schema è però un altro. La freccia nella parte inferiore indica che il livello di dettaglio aumenta procedendo da sinistra a destra.

In particolare, i *business requirements* descrivono il sistema software in modo generale e meno tecnico, mentre i *product requirements* offrono dettagli più specifici. Gli *user requirements*, invece, si collocano a un livello intermedio di astrazione. D'altra parte, le frecce tra le tipologie di requisiti indicano che dai requisiti di alto livello si possono derivare quelli più dettagliati. Per essere più precisi, dai requisiti di business si possono derivare i requisiti utente o direttamente i requisiti del prodotto, mentre dai requisiti utente è possibile derivare i requisiti del prodotto.

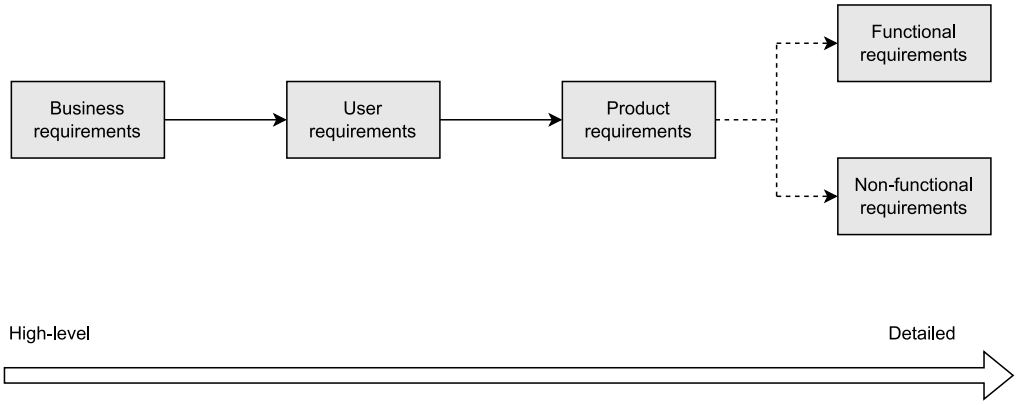

Figura 4.1. Tipologie di requisiti

Ognuna di queste categorie di requisiti riveste un ruolo fondamentale nello sviluppo del software, ciascuna con uno scopo ben definito.

Requisiti di business I requisiti di business sono i requisiti con il più alto livello di astrazione, come già menzionato in precedenza. Rappresentano i primi requisiti da definire nel processo di sviluppo del software e vengono identificati attraverso interviste, sondaggi, riunioni con i vari stakeholder e incontri aziendali interni. Questo perché il cliente necessita di un software come soluzione per soddisfare un bisogno aziendale o risolvere un problema specifico. Attraverso queste interazioni, diventa più semplice comprendere *cosa* il software deve implementare e, soprattutto, il motivo per cui dovrebbe farlo. Se si pone l'attenzione sul *perché* del progetto, significa che si ha cura dei requisiti di business. I requisiti di business, quindi, delineano in modo generale le caratteristiche del software, senza scendere nei dettagli, e ne indicano i benefici derivanti dal suo utilizzo.

Generalmente, il formato più efficiente per documentare i requisiti di business raccolti è quello del *Business Requirement Document*. Si tratta di un documento diviso in più sezioni:

- *Executive summary*: è una breve introduzione del documento in cui si cerca di riassumere il contesto del progetto e i principali obiettivi;
- *Project objectives*: è una vera e propria lista di obiettivi che si vogliono raggiungere con l'utilizzo del software in questione;
- *Project scope*: questa sezione è divisa in due liste. La prima elenca tutte le funzionalità o problematiche che il progetto deve affrontare e gestire mentre la seconda tutto ciò che non deve essere incluso o considerato;
- *Stakeholders*: è la lista di tutti gli stakeholder coinvolti nel progetto, sia interni sia esterni;
- *Constraints*: è la sezione che si occupa di elencare tutte le restrizioni previste per il progetto. Queste possono essere di diversa natura, per esempio monetaria, legale, morale ed altre.

Nella realtà, in base al contesto del progetto software, se vi è poca disponibilità di tempo e di risorse si può optare per un documento più semplice, in cui vi è un paragrafo introduttivo seguito da una lista di funzionalità software o problematiche da risolvere.

Per chiarire le idee, di seguito un esempio molto semplificato di requisito di business relativo ad un'azienda che necessita di un sistema per la prenotazione di postazioni per i propri dipendenti:

> *Abbiamo bisogno di un tool integrato nella piattaforma aziendale che permetta di prenotare una postazione in un open space riservato ai consulenti poiché il sistema attuale risulta prono ad errori in quanto gestito da dipendenti e non automatizzato.*

Requisiti utente I requisiti utente rappresentano le necessità e le aspettative degli utenti finali nei confronti del sistema software. Questi requisiti definiscono cosa gli utenti desiderano ottenere o esperire attraverso l'utilizzo del prodotto. Sono formulati in un linguaggio accessibile e descrivono le funzionalità dal punto di vista dell'utente, evitando eccessivi dettagli tecnici, per essere compresi anche da persone non formate nell'ambito informatico. Spesso, questi requisiti sono redatti da figure di alto livello, come il product manager, per garantire una chiara comprensione e allineamento con le necessità dell'utente.

I requisiti utente possono essere definiti in vari modi. Un'opzione semplice è quella di usare un linguaggio comune per descrivere in modo superficiale le azioni che l'utente può eseguire. Tuttavia, in alcuni casi, è utile arricchire la documentazione con requisiti più dettagliati. Tra questi, spiccano le *user story* e gli *use case*.

Le user story sono ampiamente utilizzate nell'approccio agile. Rappresentano un modo efficace e rapido per definire le funzionalità del software dal punto di vista dell'utente, mettendo in evidenza i benefici ottenuti dall'utilizzo della funzionalità richiesta. Sono scritte in modo accessibile e mirano a stimolare conversazioni approfondite durante le riunioni periodiche con gli stakeholder.

Gli use case, invece, offrono un livello di dettaglio maggiore rispetto alle user story. Concentrandosi sul flusso di operazioni che l'utente deve compiere per raggiungere un determinato obiettivo, facilitano lo sviluppo e il testing da parte dello sviluppatore. Gli use case rappresentano una soluzione intermedia tra i requisiti utente e i requisiti di prodotto funzionali, di cui si parlerà nel prossimo paragrafo.

Riprendendo l'esempio del requisito di business del paragrafo precedente, dei possibili requisiti utente potrebbero essere i seguenti:

> *L'utente deve poter autenticarsi al sistema usando il single sign-on aziendale.*
>
> *L'utente deve poter essere in grado di prenotare una postazione in sede.*
>
> *L'utente deve poter essere in grado di annullare una prenotazione effettuata in passato.*

Requisiti di prodotto I requisiti di prodotto possono essere derivati dai requisiti di più alto livello. Solitamente, vengono redatti da membri interni del progetto che possiedono una buona conoscenza tecnica nello sviluppo software. Essi rappresentano una descrizione dettagliata e inequivocabile del comportamento di ogni funzionalità del software, da implementare per soddisfare i requisiti di business e utente. Sono quindi requisiti ricchi di dettagli tecnici e, di conseguenza, poco comprensibili da manager di alto livello o dalla maggior parte degli stakeholder. Risultano invece fondamentali per sviluppatori e alcuni membri del progetto con conoscenze specifiche.

Come già menzionato, i requisiti di prodotto si suddividono:

- *Requisiti funzionali*: rappresentano le specifiche delle funzionalità o delle azioni che il sistema software deve essere in grado di eseguire. Essi descrivono cosa il sistema deve fare in risposta a determinate condizioni, input o eventi. In altre parole, i requisiti funzionali indicano le operazioni che il sistema deve supportare, definendo il comportamento del software. Esempi di requisiti funzionali potrebbero includere: operazioni come autenticazione degli utenti, gestione degli ordini, calcolo delle statistiche ed altri;

- *Requisiti non funzionali*: rappresentano gli aspetti di prestazione, qualità, usabilità e sicurezza del sistema. Questi requisiti definiscono le caratteristiche globali del sistema che non sono legate a specifiche azioni o funzionalità, ma, piuttosto, a come il sistema dovrebbe comportarsi o operare nel suo complesso. Ciò include prestazioni, affidabilità, sicurezza, usabilità, manutenibilità, compatibilità e altre qualità generali del sistema. Esempi di requisiti non funzionali possono essere: tempo di risposta massimo accettabile, livello di sicurezza richiesto, interfaccia utente intuitiva, compatibilità con i browser ed altri.

Dal secondo requisito utente esposto nel paragrafo precedente è possibile derivare i seguenti requisiti funzionali e non funzionali.

Requisiti funzionali:

Il sistema deve permettere all'utente di selezionare una data per cui riservare una postazione.

Il sistema deve permettere all'utente di selezionare un open space tra tutti quelli agibili.

Il sistema deve permettere all'utente di scegliere una postazione tra quelle libere all'interno dell'open space prescelto.

Il sistema deve permettere all'utente di confermare la propria scelta o annullarla.

Requisiti non funzionali:

L'open space deve essere selezionato da una piantina della struttura riprodotta digitalmente.

La postazione deve essere selezionata da una piantina dell'open space riprodotta digitalmente.

Le postazioni libere e occupate devono essere distinte tramite opportuni colori non fastidiosi per gli occhi.

I requisiti di prodotto possono essere dettagliati ulteriormente, a seconda del livello di flessibilità implementativa desiderata per gli sviluppatori. Ad esempio, considerando l'ultimo requisito funzionale, si potrebbe specificare che il sistema debba chiedere un ulteriore conferma prima di annullare effettivamente la prenotazione.

4.3 Sviluppo agile del software

La *metodologia agile* per lo sviluppo software è nata nei primi anni 2000 per tentare di risolvere i problemi legati agli approcci di sviluppo tradizionali, primo su tutti, quello a cascata, in inglese waterfall. Quest'ultimo prevede che le fasi del processo siano eseguite sequenzialmente e in modo lineare, caratteristica che comporta diversi problemi:

- *Rigidità al cambiamento*: ogni fase deve essere completata prima di passare alla successiva causando difficoltà e costi elevati per le modifiche;
- *Difficoltà nel rappresentare i requisiti*: in generale, è molto difficile definire in modo completo ed esaustivo tutti i requisiti all'inizio di un progetto, il che, può portare ad ambiguità e incomprensioni da parte degli sviluppatori;
- *Mancanza di coinvolgimento del cliente*: il coinvolgimento del cliente è limitato principalmente alla fase di raccolta dei requisiti, non vi è un coinvolgimento periodico durante tutto il processo. Ciò può portare a un prodotto finale che non soddisfa pienamente le esigenze e le aspettative del cliente;

- *Lungo ciclo di sviluppo*: a causa della natura sequenziale, il ciclo di sviluppo tende ad essere lungo e il cliente può visualizzare il prodotto completo solo alla fine del processo, con il rischio che il prodotto non corrisponda alle aspettative;

- *Rischio di consegna tardiva*: a causa della rigidità del modello e delle difficoltà nel comprendere completamente i requisiti iniziali, c'è un rischio significativo che il prodotto finale venga consegnato in ritardo;

- *Test e validazione posticipati*: probabilmente il problema più grave è quello dei test, i quali vengono eseguiti solo dopo che lo sviluppo è stato completato, comportando il rischio di individuare i problemi solo in fase di testing tardiva e integrata, rendendo quindi più complesso e costoso apportare correzioni.

La metodologia agile risulta essere un approccio di sviluppo iterativo, in cui ogni iterazione è composta da fasi predefinite, rendendo il progetto più orientato alle esigenze dei clienti, flessibile e reattivo ai cambiamenti.

Nel 2001 è stato pubblicato l'*agile manifesto*, un documento stilato da diverse figure di spicco del mondo informatico, che si basa su quattro valori fondamentali e su dodici principi guida, che delineano il modo in cui gli sviluppatori dovrebbero affrontare il processo di sviluppo.

I 4 valori del manifesto agile Dal manifesto agile emerge chiaramente un cambio di prospettiva rispetto a determinati aspetti propri degli approcci tradizionali, sottolineato dai quattro valori cardine:

- *Individuals and interactions over processes and tools*: si tende a dare più valore e importanza agli individui e alle interazioni rispetto ai processi e agli strumenti;

- *Working software over comprehensive documentation*: é più importante ottenere un software funzionante rispetto ad una documentazione completa e comprensibile;

- *Customer collaboration over contract negotiation*: la collaborazione con il cliente deve sostituire la contrattazione;

- *Responding to change over following a plan*: è importante saper rispondere ai cambiamenti rispetto al seguire un piano in modo fiscale.

I 12 principi del manifesto agile I dodici principi rendono chiari gli obiettivi dell'approccio agile. Di seguito una loro traduzione:

- La massima priorità è soddisfare il cliente attraverso la consegna tempestiva e continua di software di valore;

- Accogliere i cambiamenti nei requisiti, anche in fase avanzata di sviluppo. I processi agile sfruttano il cambiamento per il vantaggio competitivo del cliente;

- Consegnare frequentemente software funzionante, con una frequenza che può variare dalle poche settimane ai pochi mesi, con una preferenza per un breve arco temporale;
- Le persone del business e gli sviluppatori devono collaborare quotidianamente per l'intera durata del progetto;
- Costruire i progetti attorno a individui motivati. Fornire loro l'ambiente e il supporto di cui hanno bisogno, e fidarsi che completeranno il lavoro;
- Il metodo più efficiente ed efficace di comunicare informazioni a un team di sviluppo e all'interno di esso è la conversazione faccia a faccia;
- Il progresso è principalmente valutato in base al funzionamento effettivo del software;
- I processi agile promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere un ritmo costante indefinitamente;
- Un'attenzione continua all'eccellenza tecnica e al buon design migliora l'agility;
- La semplicità, ovvero l'arte di massimizzare la quantità di lavoro non fatto, è essenziale;
- Le migliori architetture, requisiti e progettazioni emergono dai team auto-organizzati;
- A intervalli regolari, il team riflette su come diventare più efficace, quindi adatta e migliora il proprio comportamento di conseguenza.

Negli ultimi due decenni, la metodologia agile è stata adottata sempre più frequentemente all'interno delle aziende del settore. Esistono diversi framework per la gestione di progetti software che implementano i principi agile, tra i quali i più conosciuti sono: Scrum, Kanban, eXtreme Programming ed altri. Nonostante seguano tutti lo stesso approccio di base, presentano alcune differenze sostanziali. Nei paragrafi successivi, verrà esaminato il framework Scrum, analizzandone gli elementi chiave e le sue fasi principali.

4.4 Il framework Scrum

Scrum è un framework agile il cui obiettivo è quello di aiutare persone, team e organizzazioni nel generare valore, trovando soluzioni a problemi più o meno complessi in modo adattativo. Non mira a fornire dettagliate linee guida per l'intero ciclo di sviluppo software; al contrario, delinea solo alcune regole fondamentali basate sulla teoria Scrum, offrendo un alto grado di libertà ai suoi utilizzatori.

Questa scelta deriva dall'intento di attribuire gran parte del successo relativo a un progetto al gruppo di lavoro. Grazie alla propria intelligenza e autoanalisi, il gruppo

può evolversi iterazione dopo iterazione acquisendo nuove conoscenze, skill e sinergia. Scrum rende visibile l'efficacia della gestione, dell'ambiente e delle tecniche di lavoro, consentendo così di apportare miglioramenti.

Ciò detto, Scrum si basa sui concetti di *empirismo* e di *lean thinking*, cioè pensiero snello. Il primo afferma che l'esperienza è la prima fonte di conoscenza, mentre il secondo sottolinea che è importante concentrarsi sugli aspetti essenziali, evitando sprechi di risorse su elementi poco rilevanti.

La teoria Scrum è fondata su tre principi cardine chiamati *pilastri della teoria Scrum*:

- *Transparency*: la trasparenza, come principio fondamentale, stabilisce che il processo e il lavoro devono essere visibili sia a chi lo sta effettivamente conducendo che a chi ne sta ricevendo i risultati. È quindi evidente che il cliente stesso debba essere informato sull'andamento dello sviluppo, avendo piena conoscenza degli obiettivi raggiunti e delle eventuali problematiche incontrate. Tale trasparenza favorisce una corretta fase di ispezione. In mancanza di essa, l'ispezione risulterebbe fuorviante, aumentando quindi la probabilità di non raggiungere perfettamente l'obiettivo finale;

- *Inspection*: gli artefatti del processo e il progresso devono essere ispezionati periodicamente e con grande attenzione, al fine di individuare eventuali problemi indesiderati. In questo contesto, un'ispezione accurata agevola l'adattamento che risulta essere cruciale in quanto gli eventi Scrum sono progettati per stimolare il cambiamento;

- *Adaptation*: se l'ispezione ha evidenziato aspetti del processo che deviano al di fuori di limiti accettabili o se il prodotto risultante non è accettabile, è importante prendere delle azioni correttive. Questi aggiustamenti devono essere effettuati il prima possibile. L'adattamento è strettamente correlato al concetto di team auto-gestito, concetto esposto in uno dei dodici principi dello sviluppo agile. Il framework Scrum sottolinea come un team auto-gestito e con potere decisionale abbia la capacità di reagire e adattarsi facilmente ai problemi non appena questi ultimi vengono individuati.

Se questi tre principi descrivono quali sono le caratteristiche principali del processo di sviluppo software, Scrum ha delineato anche i *valori* che ogni membro del team dovrebbe seguire per raggiungere gli obiettivi e migliorare le proprie competenze. In totale sono cinque, ovvero:

- *Commitment*: il team Scrum si impegna a raggiungere i propri obiettivi e a supportarsi reciprocamente;

- *Focus*: la loro attenzione è principalmente rivolta al lavoro da svolgere nello Sprint[1] per fare il miglior progresso possibile verso gli obiettivi prefissati;
- *Openness*: i membri del team e gli stakeholder sono aperti riguardo al lavoro e alle problematiche;
- *Respect*: i membri del team Scrum si rispettano reciprocamente come persone capaci e indipendenti;
- *Courage*: i membri del team hanno il coraggio di fare ciò che è giusto e di affrontare problemi non banali.

4.4.1 Scrum team

Uno *Scrum team* è un team unito, privo di gerarchie, che lavora per raggiungere lo stesso obiettivo, ovvero il Product Goal. Uno Scrum team è composto da un *Product Owner*, uno *Scrum Master* e i *Developer*. Si tratta di un gruppo di poche persone, circa dieci, abbastanza piccolo per essere autosufficiente e comunicare in modo efficiente, riuscendo così ad essere molto produttivo. Abbastanza grande da poter terminare con successo il lavoro programmato nello Sprint senza troppi problemi. Lo Scrum Team ha la responsabilità di gestire tutto ciò che è legato al prodotto, partendo dalla creazione dei task per implementare le varie funzionalità software, fino alla collaborazione e comunicazione con gli stakeholder.

Developers Gli sviluppatori sono responsabili per:

- creare un piano per il successivo Sprint, lo Sprint Backlog;
- promuovere un certo livello di qualità aderendo alla Definition of Done;
- se necessario, adattare il loro piano quotidianamente allo Sprint Goal;
- essere responsabili gli uni verso gli altri come professionisti.

Product Owner L'obiettivo principale del Product Owner è quello di massimizzare il valore generato dal lavoro complessivo conseguito dal team. Il modo con cui esso riesca a massimizzarlo può variare in base all'azienda e al progetto su cui si sta lavorando. Nella pratica, si occupa di gestire il Product Backlog[2], quindi di:

- stabilire e comunicare al team il Product Goal;

[1] Uno Sprint è un periodo di tempo, di solito di due o quattro settimane, durante il quale un team di sviluppo lavora intensamente per completare un insieme di attività concordate, producendo un incremento di lavoro funzionante.

[2] Il Product Backlog è una lista dinamica e ordinata per importanza, di tutte le funzionalità, requisiti, miglioramenti e correzioni di bug che devono essere sviluppati per un prodotto software.

- definire e comunicare in modo chiaro e coinciso il Product Backlog al resto del team;
- ordinare gli elementi del Product Backlog secondo un criterio di valore generato;
- assicurarsi che il Product Backlog sia sempre chiaro, comprensibile e trasparente.

Scrum Master Lo Scrum Master è la figura che stimola il team e l'organizzazione ad applicare in modo corretto la teoria Scrum, aiutandoli nel corso dei processi aziendali o di sviluppo. Lo Scrum Master aiuta lo Scrum Team nei seguenti modi:

- ad essere auto sufficiente e multidisciplinare (cross-functional);
- a concentrarsi sul generare progressi di valore seguendo la Definition of Done;
- a superare eventuali problemi ed ostacoli;
- ad assicurare che tutti gli eventi previsti dal framework siano produttivi e conseguiti entro i limiti di tempo programmati.

Inoltre, lo Scrum Master serve il Product Owner:

- aiutandolo ad identificare tecniche per definire il Product Goal e per gestire opportunamente il Product Backlog;
- sensibilizzando lo Scrum Team sull'importanza di avere elementi chiari e concisi nel Product Backlog;
- aiutando a definire una pianificazione empirica, basata su esperienza e dati del passato;
- favorendo la collaborazione degli stakeholder, se richiesta.

Infine, esso aiuta l'intera organizzazione in diversi modi, tra cui:

- guidando l'organizzazione nell'adozione del framework Scrum;
- pianificando l'implementazione di Scrum all'interno dell'organizzazione;
- aiutando i dipendenti e gli stakeholder a comprendere ed attuare un approccio empirico;
- rimuovendo le barriere tra gli stakeholder e il team.

4.4.2 Scrum events

Tutti gli *eventi* presenti nel framework Scrum sono contenuti nell'evento principale chiamato Sprint. Ogni evento deve essere visto come l'occasione per poter ispezionare gli artifact e, quindi, adattare il proprio modo di lavorare, se necessario. Oltre a questo, gli eventi Scrum sono ideati in modo tale da non aver necessità di aggiungere altre riunioni, minimizzando il più possibile il tempo impiegato nella comunicazione e sincronizzazione.

The Sprint Lo *Sprint* è il punto focale dell'intero framework Scrum e contiene all'interno tutti gli altri eventi. Rappresenta l'iterazione base di Scrum che può durare un mese o qualche settimana. Non appena vien concluso uno Sprint si inizia il successivo. Si preferisce uno Sprint della durata di alcune settimane poiché un'iterazione relativamente breve garantisce un'analisi periodica di durata ridotta del progresso effettuato. Questo approccio permette di individuare tempestivamente eventuali problemi non appena si presentano. Inoltre, il team cresce più rapidamente grazie all'esperienza acquisita di Sprint in Sprint.

Sebbene durante uno Sprint siano permessi cambiamenti, non sono ammesse modifiche che possano compromettere lo Sprint Goal. La qualità, e quindi il valore generato da uno Sprint singolo, non può discostarsi troppo da quanto inizialmente pianificato. Il Product Backlog può essere ridefinito se necessario, e possono essere avviate discussioni e negoziazioni con il Product Owner per chiarire alcuni elementi e obiettivi.

Sprint Planning Il primo evento dello Sprint è lo Sprint Planning che consiste nel pianificare quale progresso deve essere raggiunto nello Sprint attuale. Lo Sprint Planning risponde in pratica alle seguenti domande:

- *Why is this Sprint valuable?* Il Product Owner comunica al team come il prodotto potrebbe acquisire valore aggiunto durante lo Sprint corrente. Il team, quindi, collabora per definire uno Sprint Goal, il cui scopo è spiegare che beneficio lo Sprint corrente fornisce agli stakeholder. Naturalmente, lo Sprint Goal deve essere raggiunto entro la fine dell'iterazione;
- *What can be Done this Sprint?* Il Product Owner e gli sviluppatori discutono per selezionare alcuni degli elementi (item) del Product Backlog da inserire nello Sprint. Questi item possono essere raffinati e leggermente modificati con il permesso del Product Owner. La quantità di elementi inseriti nello Sprint dipende dalle previsioni che gli stessi developer effettuano. Basandosi sull'esperienza e sulla fiducia acquisite in passato, possono stimare l'effort richiesto per ciascun elemento e, quindi, selezionare quelli più appropriati per lo Sprint;
- *How will the chosen work get done?* Ogni elemento selezionato per lo Sprint viene decomposto in Increment, che rispettino la Definition of Done, della durata di un giorno o meno. Questo compito è affidato completamente agli sviluppatori. Nessun'altra figura dello Scrum Team può metter parola sul come si debbano decomporre gli elementi del Product Backlog.

È importante notare che lo Sprint Backlog è composto dall'unione dello Sprint Goal, degli elementi selezionati dal Product Backlog e del piano che indica la loro scomposizione in task più piccoli. Inoltre, è comune organizzare lo Sprint Planning con una durata di otto ore per Sprint di un mese. Nel caso di Sprint più brevi, la durata della pianificazione viene ulteriormente ridotta.

Daily Scrum Come suggerisce il nome, il Daily Scrum è un breve incontro giornaliero della durata di quindici minuti. Per comodità, è bene che si svolga sempre alla stessa

ora e nello stesso luogo, durante la giornata lavorativa. L'obiettivo di questo incontro è analizzare i progressi raggiunti nell'arco della giornata e apportare eventuali modifiche allo Sprint Backlog. Il modo e le tecniche con cui si conduce questo incontro non sono rigidamente definiti, purché il team mantenga la concentrazione nel progredire verso lo Sprint Goal, pianificando come procedere con il lavoro nel giorno successivo. In aggiunta, i Daily Scrum favoriscono l'auto-gestione.

In sintesi, queste piccole riunioni giornaliere facilitano la comunicazione, l'identificazione dei problemi e il quick decision making. Va sottolineato che non è l'unico momento in cui gli sviluppatori si allineano e discutono eventuali questioni relative al progresso dello Sprint. Questi scambi possono avvenire anche durante il resto della giornata, anche se non prendono il nome di Daily Scrum.

Sprint Review Lo Sprint Review è uno degli ultimi eventi in cui il team presenta il lavoro completato durante lo Sprint corrente a stakeholder chiave. L'obiettivo principale è valutare quali obiettivi dello Sprint Backlog sono stati raggiunti e quali sono rimasti incompleti, al fine di pianificare eventuali azioni correttive. Durante questa revisione, è possibile apportare modifiche al Product Backlog per cogliere nuove opportunità, ad esempio l'implementazione di una nuova funzionalità. Solitamente, per uno Sprint di un mese, la durata prevista dello Sprint Review è di circa quattro ore.

Sprint Retrospective Il momento dello Sprint Retrospective segna la conclusione dello Sprint, focalizzandosi sull'incremento della qualità e dell'efficienza del team di sviluppo. Durante questa fase, si effettua un'ispezione critica del lavoro svolto durante lo Sprint in corso, esaminando da vicino il contributo individuale, le interazioni, i processi e gli strumenti utilizzati. Identificando gli eventuali ostacoli, il team delinea come affrontarli nelle prossime iterazioni, inserendo, se possibile, i task relativi ai problemi più importanti, nello Sprint successivo. La durata media, sempre per uno Sprint di un mese, è di circa tre ore.

4.4.3 Scrum artifacts

Gli *artefatti Scrum* sono documenti creati per rendere chiari e trasparenti i dettagli più importanti del processo. Forniscono informazioni cruciali per valutare il progresso del lavoro in un determinato momento. Il numero di artefatti previsti dal framework è tre, ognuno dei quali può essere utilizzato per misurare un progresso differente all'interno del processo di sviluppo. Il *Product Backlog* fornisce informazioni per misurare il Product Goal, ossia il progresso generale del prodotto finale. D'altra parte, lo *Sprint Backlog* offre informazioni per valutare il progresso di uno Sprint singolo. Infine, l'*Increment* rappresenta il grado di completamento rispetto alla Definition of Done.

Product Backlog Il Product Backlog è una lista ordinata di elementi, essenziali per creare valore e migliorare il prodotto. Tutti gli elementi valutati di dimensioni adeguate sono considerati pronti per essere discussi durante lo Sprint Planning da parte del team. Se ciò non avviene, il Product Backlog viene adeguatamente modificato per

suddividere gli elementi troppo grandi per uno Sprint in elementi più gestibili. Questo processo è noto come affinamento del Product Backlog. Come accennato in precedenza, esaminando il Product Backlog è possibile valutare lo stato attuale del Product Goal, che rappresenta l'obiettivo a lungo termine dello Scrum Team.

Sprint Backlog Lo Sprint Backlog è composto dallo Sprint Goal, che spiega il motivo per cui l'attuale Sprint può portare benefici agli Stakeholder, dall'insieme di elementi selezionati dal Product Backlog, i quali rappresentano cosa si intende implementare nello Sprint, e da un piano operativo per realizzare l'Increment. È una rappresentazione in tempo reale del lavoro che gli sviluppatori intendono completare durante lo Sprint per raggiungere lo Sprint Goal. Dovrebbe presentare un livello di dettaglio tale da poter essere ispezionato per comprendere il progresso degli sviluppatori durante il Daily Scrum.

Increment Un Increment è un progresso tangibile verso il Product Goal. Ciascuno di essi si somma a tutti gli Increment precedenti garantendo che tutti funzionino insieme come previsto. Per apportare valore, l'Increment deve essere utilizzabile. Durante uno Sprint possono essere creati più Increment, e l'insieme di essi viene presentato durante la Sprint Review. In alcuni casi, un Increment può essere consegnato agli stakeholder anche prima della fine dello Sprint.

Tuttavia, il lavoro non può essere considerato parte di un Increment se non soddisfa la Definition of Done. Quest'ultima è una descrizione formale dello stato dell'Increment quando raggiunge il livello di qualità richiesto per il prodotto. Non appena un elemento del Product Backlog soddisfa la Definition of Done, un nuovo Increment si concretizza. Al contrario, se un elemento del Product Backlog non soddisfa la Definition of Done, non può essere rilasciato o presentato durante la Sprint Review. In questo caso, viene reinserito nel Product Backlog per future valutazioni.

Capitolo 5

Il progetto

Questo capitolo offre un'analisi approfondita del progetto software affrontato, focalizzandosi sugli obiettivi, sul design software e sull'architettura cloud adottata.

La prima sezione, *Descrizione del progetto e obiettivi formativi*, presenta una descrizione dettagliata del progetto e del suo dominio, insieme agli obiettivi formativi che ne hanno guidato lo sviluppo.

La seconda sezione, intitolata *Software design*, approfondisce il design del software, focalizzandosi su due aspetti cruciali del processo di sviluppo, ossia la definizione dei requisiti a partire da quelli di alto livello e la progettazione dell'architettura a microservizi.

Nella terza e ultima sezione, *Definizione dell'architettura cloud*, si analizza l'architettura cloud adottata nel progetto descrivendo come le varie componenti sono interconnesse tra loro.

5.1 Descrizione del progetto e obiettivi formativi

Descrizione del progetto Il progetto in questione riguarda lo sviluppo di un software backend per una web app utilizzando il linguaggio di programmazione JavaScript tramite il runtime system Node.js e basandosi su un'architettura a microservizi, partendo dalla fase di progettazione, fino alla scrittura del codice e testing.

Lo scopo principale dell'applicazione è facilitare il processo di prospecting all'interno di un'azienda di consulenza. Questo processo comprende la ricerca, la valutazione e l'approccio alle aziende che potrebbero essere interessate a stabilire relazioni lavorative e, di conseguenza, adottare servizi di consulenza.

Dalla definizione è evidente che tale processo può essere scomposto in più fasi:

- *Identificazione dei prospect*: è la prima fase che consiste nella ricerca e individuazione potenziali aziende clienti;

- *Qualificazione dei prospect*: in seguito vi è la valutazione dei prospect per determinare se soddisfano i criteri desiderati e se ci sono opportunità di collaborazione;
- *Approccio e comunicazione*: riguarda l'avvio del contatto e interazione con i prospect, presentando l'azienda, i servizi offerti o le opportunità di collaborazione;
- *Gestione dei rapporti*: infine c'è la fase di mantenimento di relazioni positive con i prospect attraverso comunicazioni continue.

In un'azienda di consulenza, è consuetudine che figure aziendali quali i business manager o professionisti con responsabilità nella gestione commerciale e nello sviluppo aziendale, si occupino dell'identificazione di potenziali clienti, stabiliscano relazioni, presentino offerte, conducano le negoziazioni contrattuali e promuovano l'acquisizione di nuovi progetti per l'azienda.

Uno strumento che negli ultimi anni è diventato indispensabile per il processo di prospecting è Linkedin. Comunemente, un manager avvia una prima fase di ricerca per individuare aziende del settore desiderato, potenzialmente interessate a instaurare rapporti lavorativi. Successivamente, identifica all'interno di tali aziende figure professionali idonee a coltivare tali rapporti. Ad esempio, è comune cercare business manager, project manager, responsabili delle opportunità e altre figure correlate. Queste operazioni costituiscono la prima fase del processo di prospecting, e LinkedIn offre un avanzato sistema di ricerca, con opzioni di filtraggio sia per le aziende che per i dipendenti. Successivamente, dopo aver selezionato le persone ritenute più adatte, queste vengono contattate tramite email o telefonicamente, se tali informazioni sono disponibili sul loro profilo, oppure direttamente tramite LinkedIn.

In sostanza, il backend di questa applicazione utilizza le API messe a disposizione da una piattaforma simile a Linkedin, denominata *Apollo.io*, per eseguire ricerche di aziende e dipendenti utilizzando filtri avanzati. A questa caratteristica si affiancano altre funzionalità utili per gli utenti, tra cui:

- *Invio email semplificato*: rende possibile inviare email a tutti i dipendenti individuati tramite una ricerca o a un sottoinsieme di essi senza dover inserire manualmente il testo o l'indirizzo email per ogni singola persona da contattare. Questo è reso possibile grazie a un testo di base creato dall'utente e memorizzato nel proprio account, chiamato tecnicamente *email template* dinamico. Un email template rappresenta un modello predefinito di un messaggio email che può essere personalizzato in modo automatico, mediante l'inserimento dinamico di informazioni relative al destinatario. Per esempio nome, cognome, indirizzo email;
- *Salvataggio ricerche*: vi è la possibilità di salvare una ricerca effettuata, in modo tale da poterla recuperare in futuro e continuare a contattare altre persone;
- *Statistiche*: l'utente può consultare diverse tipologie di statistiche con granularità temporale personalizzabile, generate automaticamente dal sistema con cadenza giornaliera.

Prima di discutere degli obiettivi formativi del progetto e, in seguito, della progettazione software, è utile distinguere due termini fondamentali usati all'interno della documentazione stilata durante le fasi iniziali. In questo contesto per *prospect* si intende una potenziale azienda cliente con cui l'utente cercherà di coltivare rapporti lavorativi. D'altra parte, le figure aziendali dell'azienda prospect coinvolte nella fase di approccio e comunicazione, sono chiamate *contact*, in quanto rappresentano il punto di contatto tra le due aziende.

Obiettivi formativi Gli obiettivi riguardano prettamente la formazione personale nell'ambito della progettazione e implementazione di software a microservizi. I dettagli specifici di tali obiettivi sono illustrati nella seguente lista:

- Progettazione di software a microservizi seguendo con cura le best practice, al fine di garantire un'architettura efficiente e scalabile per l'applicazione;
- Approfondire la conoscenza dell'ambiente cloud di Microsoft Azure per comprendere in modo dettagliato le sue funzionalità, strumenti e servizi, al fine di poter efficacemente progettare, implementare e gestire soluzioni basate su questa piattaforma;
- Esplorare e sperimentare l'impiego di nuovi strumenti e librerie che non sono stati utilizzati in precedenza, al fine di arricchire le competenze tecniche e ampliare l'orizzonte delle risorse disponibili per lo sviluppo di soluzioni software innovative.

5.2 Software design

La progettazione software ha avuto inizio con l'analisi di requisiti di alto livello, dai quali sono stati derivati requisiti più dettagliati, come le user story e le acceptance criteria. Questi ultimi hanno svolto un ruolo fondamentale, prima come punto di partenza per la definizione dell'architettura a microservizi e, successivamente, per chiarire le funzionalità e il comportamento desiderato del software durante la fase di scrittura del codice.

Il passo successivo, come anticipato, è stato quello di progettare l'architettura a microservizi. A tal scopo è stato utilizzato l'algoritmo per la decomposizione in microservizi proposto da *Chris* Richardson, dettagliato nel Capitolo 2 (consultabile alla Sezione 2.2.1). Questo processo ha portato all'identificazione di diversi microservizi, ognuno rappresentante una parte specifica del dominio, con le rispettive operazioni, API e collaborazioni. L'emergere della necessità di collaborazioni tra questi microservizi ha richiesto uno sforzo aggiuntivo nella fase finale del design dell'architettura. La conclusione di questo processo è stata resa possibile mediante l'elaborazione di un sistema di comunicazione efficace, essenziale per garantire una sinergia ottimale tra i vari componenti del sistema.

5.2.1 Definizione delle user story e delle acceptance criteria

La prima fase affrontata nel corso del progetto è stata quella riguardante la definizione dei requisiti utente, più in particolare le user story, a partire da una lista di requisiti di alto livello. Le user story sono poi state arricchite con la scrittura delle acceptance criteria.

I requisiti di alto livello risultano essere simili ai requisiti di business benché non seguano fedelmente la loro struttura e principi. Per comodità sono stati raggruppati in tre macro categorie: *Search*, *Contact* e *Storage and statistics*. La prima include tutti i requisiti riguardanti la ricerca di aziende e persone, la seconda racchiude i requisiti relativi alla comunicazione mentre l'ultima, quella più corposa, comprende tutti i requisiti inerenti alla memorizzazione di dati e alla produzione di statistiche.

- *Search*:
  - The user must be able to search contacts of a single prospect company filtering them by location and one or multiple roles;
  - The mandatory data returned by the search must be: name, surname, location, role, email, phone number, linkedIn URL.
- *Contact*:
  - The user must be able to select people obtained from the search and send them a standard prospecting email.
- *Storage and statistics*:
  - The user must be able to create and save the email template;
  - The user must be able to save the search result or some contacts obtained from it in order to contact them in the future;
  - The user must be able to see which contacts have been contacted and when;
  - The user must be able to attach a feedback to a contact to which an email has been sent;
  - The user must be able to add a note to a contact;
  - The system must highlight already reached contacts when a search result is shown;
  - The system must warn the user when an email is about to be sent to an already reached contact.
  - The user must be able to see the following statistics:
    * Number of prospects searched and number of prospects reached;
    * Number of contacts reached (filtered by time frames);
    * List of prospects ordered by number of contacts reached (filtered by time frames).

Le user story illustrate nella Tabella 5.1 derivano dai requisiti di alto livello e seguono il formato più ampiamente utilizzato nell'ambito agile: *As - I want - So that*. Tale formato evidenzia chiaramente le esigenze dell'utente e il valore aggiunto che può derivare da una specifica funzionalità. Inoltre, sono state disposte in ordine di priorità, considerando il valore complessivo che ciascuna funzionalità apporta all'applicazione.

Tabella 5.1: User story.

Title	**User story**
Search contacts	AS a user I WANT to search for company contacts SO THAT I can see a list of them on the screen
Save email template	AS a user I WANT to save a prospecting email template in the system SO THAT I don't have to write an email for each contact
Send email	AS a user I WANT to send prospecting emails to some or all contacts of a research SO THAT I don't have to send them manually from my email client
Retrieve reached contacts	AS a user I WANT to retrieve reached contacts SO THAT I can see when I've reached them and I can attach feedbacks
Save contacts of a search	AS a user I WANT to save some or all contacts of a search SO THAT I can retrieve them in the future
Retrieve saved contacts of a search	AS a user I WANT to retrieve saved searches SO THAT I can continue to send emails

Continua ...

... Continuazione della tabella

Title	**User story**
Receive already reached alert	AS a user I WANT to receive an alert before sending an email to an already reached contact SO THAT I can decide carefully if the email should be sent again
Attach note	AS a user I WANT to attach a note to a contact SO THAT I can add any kind of information about the contact
Attach feedback	AS a user I WANT to attach feedback to an already reached contact SO THAT I can summarize the interest of the contact to my proposal
Retrieve prospects statistics	AS a user I WANT to see the statistics of number of prospects searched and contacted SO THAT I can report to my superiors
Retrieve contacts statistics	AS a user I WANT to see statistics of number of contacts reached SO THAT I can report to my superiors
Retrieve list of contacted prospects	AS a user I WANT to see a list of prospects ordered by number of contacts reached SO THAT I can report to my superiors
Login	AS a user I WANT to login in the system
Logout	AS a user I WANT to logout from the system

Successivamente, per ciascuna user story, sono state definite le acceptance criteria illustrate nella Tabella 5.2, che rappresentano requisiti più dettagliati e tecnici. Esse forniscono una descrizione approfondita del comportamento del sistema software. Possono essere consultate per valutare l'implementazione corretta della rispettiva user story e costituiscono una solida base per la creazione dei casi di test.

Anche in questo caso è stato utilizzato uno dei formati più comunemente usati per questa tipologia di requisito: *Given - When - Then*. In sintesi, il formato in questione

è composto da tre sezioni che delineano rispettivamente le precondizioni, ossia lo stato del sistema prima dell'azione che implementa la funzionalità, l'azione stessa che si intende descrivere e le post condizioni del sistema, ovvero le modifiche che è ragionevole attendersi a seguito dell'azione eseguita.

Tabella 5.2: Acceptance criteria.

User story	**Acceptance criteria**
Search contacts	GIVEN I'm a logged-in user and I'm in the "search" page and I've filled the company input field and I've possibly filled the filters input field WHEN I click the "search" button THEN a search of contacts is performed by the system and a list of contacts is shown on the screen having name, surname, location, email, phone number, verified email, verified phone number, linkedIn URL details
Save email template	GIVEN I'm a logged-in user and I'm in the "email template" page and I fill the text box following the instructions to insert the parameters that the system can parse WHEN I click the "save" button THEN the email template is saved by the system and it is associated to my account
Send email	GIVEN I'm a logged-in user and I've configured the email template and I'm in the "search" page or I'm in the "saved searches" page and I see a list of contacts on the screen WHEN I select some or all the contacts and I click the "send email" button THEN an email sending is taken over by the system and a success or failure message is shown on the screen to warn the user if the email has been sent successfully

Continua ...

... Continuazione della tabella

User story	**Acceptance criteria**
Retrieve reached contacts	GIVEN I'm a logged-in user WHEN I'm in the "reached contacts" page THEN I can see a list of reached contacts with contact details, a button to attach feedbacks, a timestamp that represents the time when the email has been sent and I can filter them by time frames, role, name, surname, prospect name
Save contacts of a search	GIVEN I'm a logged-in user and I'm in the "search" page and I see a list of contacts WHEN I select some or all the contacts and I click the "save contacts" button THEN the selected contacts are saved as a saved search by the system
Retrieve saved contacts of a search	GIVEN I'm a logged-in user and I'm in the "saved searches" page and I've possibly filtered saved searches by time frames and prospect name WHEN I click on a saved search THEN I can see the list of contacts belonging to the search
Receive already reached alert	GIVEN I'm a logged-in user and I'm in the "search" page or I'm in the "saved searches" page and I see a list of contacts WHEN I select some or all the contacts and I click the "send email" button THEN the system send me an alert if I've already sent an email to any contact and it provides the option to resend the email or not for each contact

Continua ...

... Continuazione della tabella

User story	**Acceptance criteria**
Attach note	GIVEN I'm a logged-in user and I'm in the "saved searches" page or I'm in the "reached contacts" page and I see a list of contacts WHEN I click the "attach note" button and I fill the text box and I click the "save" button THEN the note is attached to the selected contact by the system
Attach feedback	GIVEN I'm a logged-in user and I'm in the "saved searches" page or I'm in the "reached contacts" page and I see a list of contacts WHEN I click the "attach feedback" button and I fill the text box and I click the "save" button THEN the feedback is attached to selected contact by the system
Retrieve prospects statistics	GIVEN I'm a logged-in user WHEN I'm in the "home" page THEN I can see the statistics regarding the number of prospects searched and the number of prospects reached
Retrieve contacts statistics	GIVEN I'm a logged-in user WHEN I'm in the "home" page THEN I can see the statistics regarding the number of contacts reached and I can filter them by time frames
Retrieve list of contacted prospects	GIVEN I'm a logged-in user WHEN I'm in the "home" page THEN I can see a list of prospects ordered by number of contacts reached and I can filter them by time frames
Login	GIVEN I'm a logged-out user WHEN I click the "login" button and I fill the input fields with my credentials THEN the system signs me in

Continua ...

... Continuazione della tabella

User story	Acceptance criteria
Logout	GIVEN I'm a logged-in user WHEN I click the "logout" button THEN the system signs me out

5.2.2 Design dell'architettura a microservizi

Il processo di progettazione dell'architettura a microservizi è stato affrontato in primis applicando l'algoritmo di decomposizione di Chris Richardson, come discusso dettagliatamente nel Capitolo 2. Successivamente, una volta definite le responsabilità di ciascun microservizio e le tipologie di richieste che ciascuno di essi può gestire, è stato delineato il sistema di comunicazione tra di essi. In questa sezione, si procederà prima con l'approfondimento dell'algoritmo di decomposizione in microservizi. In seguito, si fornirà una descrizione delle responsabilità attribuite ai microservizi individuati e al contempo si illustrerà il sistema di comunicazione.

Algoritmo di decomposizione in microservizi

Per riassumere brevemente i concetti chiave di questo algoritmo, esso si articola in tre fasi principali. La prima fase mira a creare un vocabolario comune e coerente che sarà seguito per l'intero sviluppo del software, identificando contemporaneamente le operazioni di sistema. La seconda fase si occupa di individuare i microservizi, utilizzando uno dei due approcci disponibili: decomposition by subdomain e decomposition by business capability. Infine, la terza e ultima fase riguarda la definizione delle API dei singoli microservizi e le eventuali interazioni collaborative. Da questo punto in poi, approfondiremo i procedimenti e i risultati relativi a quattro obiettivi chiave:

- Definizione del vocabolario comune
- Definizione delle system operation
- Identificazione dei microservizi (utilizzando la decomposition by business capability)
- Definizione delle API dei microservizi e collaborazioni

Va notato che i primi due punti costituiscono il primo passo dell'algoritmo di decomposizione ma per chiarezza è conveniente trattarli come due punti distinti.

Definizione del vocabolario comune Riprendendo quanto già illustrato nel Capitolo 2, per creare il vocabolario comune del progetto è conveniente disporre di requisiti quali le user story e le acceptance criteria, analizzarli per identificare i sostantivi presenti all'interno di essi e, infine, creare un class diagram di alto livello, più propriamente

chiamato *High level domain model*, nel quale illustrare le varie classi e relazioni che sussistono tra loro.

Per non dilungare troppo il procedimento, di seguito, verranno prese in considerazione due user story tra quelle illustrate in Tabella 5.1 come esempio per l'individuazione dei sostantivi.

> AS a user
> I WANT to search for company contacts
> SO THAT I can see a list of them on the screen

> AS a user
> I WANT to attach a note to a contact
> SO THAT I can add any kind of information about the contact

I termini sottolineati, *contacts* e *note*, indicano i vocaboli che saranno inclusi nel vocabolario comune menzionato in precedenza. Dal contesto inoltre, risulta abbastanza semplice individuare anche le relazioni fra queste classi. In questo caso tra la classe note e contact sussiste una relazione che può essere etichettata con il nome *attached*, in quanto una nota è allegata ad un contatto. È cruciale notare che la ricerca dei sostantivi vada effettuata in tutte le user story e anche nelle rispettive acceptance criteria. Iterando tale procedimento si ottiene un class diagram simile a quello illustrato in Figura 5.1.

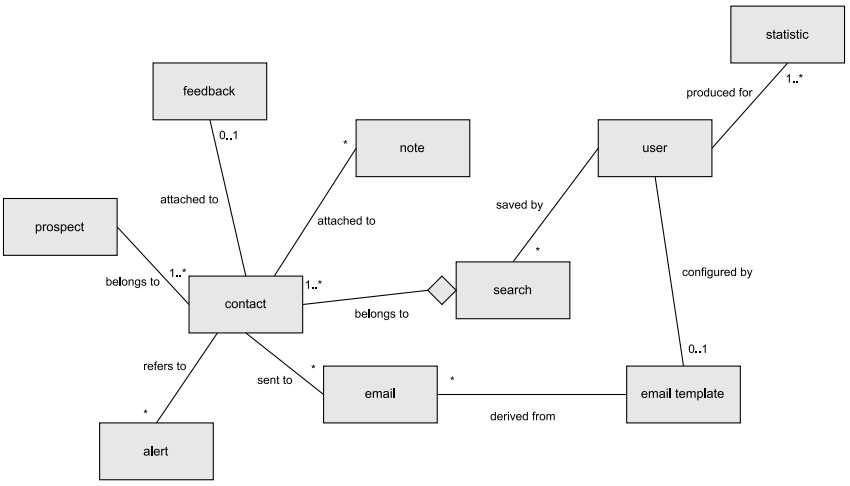

Figura 5.1. High level domain model

Definizione delle system operation Per completare il primo step dell'algoritmo di decomposizione è essenziale definire le system operation. Questo processo segue una procedura simile a quella delineata nel paragrafo precedente. In questo caso, l'approccio coinvolge un'analisi delle user story e delle acceptance criteria, ma con un focus specifico sui verbi anziché sui sostantivi.

Una volta individuati i verbi nei requisiti sopracitati, è naturale derivare un nome per ogni system operation, generalmente composto dal verbo e da un complemento oggetto, con quest'ultimo che corrisponde solitamente ad un sostantivo del vocabolario comune definito nella medesima fase.

Successivamente, risulta conveniente schematizzare i risultati in tabelle contenenti informazioni aggiuntive. La prima tabella rappresenta un elenco completo di tutte le system operation, specificando per ognuna di esse l'attore che può eseguire l'operazione, la user story di origine e una breve descrizione. Le tabelle successive, una per ciascuna system operation, approfondiscono ulteriormente i dettagli, includendo parametri di input e output, precondizioni e postcondizioni. Quest'ultime possono essere praticamente replicate dalle sezioni *given* e *then* delle acceptance criteria.

Anche in questo caso verranno prese in considerazione due user story, presenti nella Tabella 5.1, come esempio per illustrare l'individuazione dei verbi.

> AS a user
> I WANT to save a prospecting email template in the system
> SO THAT I don't have to write an email for each contact
>
> AS a user
> I WANT to send prospecting emails to some or all contacts of a research
> SO THAT I don't have to send them manually from my email client

Nella prima user story è stato individuato il verbo *save*, associato al salvataggio dell'email template. Da ciò è semplice dedurre il nome della system operation corrispondente, ossia *saveEmailTemplate()*. Analogamente, nella seconda user story riguardante l'invio di email, è stato identificato il verbo *send*, il quale induce la definizione della system operation *sendEmail()*. Il risultato ottenuto iterando questo procedimento su tutti i requisiti è il seguente, schematizzato nelle tabelle menzionate in precedenza. Nella Tabella 5.3, la quale illustra tutte le system operation con alcuni dettagli aggiuntivi di alto livello, è stata omessa la colonna riguarda l'attore, ovvero il tipo di utente, per problemi di spazio.

Tabella 5.3: System operation.

User story	**System Operation**	**Description**
Search contacts	searchContacts()	Performs a research of contacts
Save email template	saveEmailTemplate()	Saves the email template written by the user

Continua ...

... Continuazione della tabella

User story	System operation	Description
Send email	sendEmail()	Sends the email using the email template and the data of the contact selected
Retrieve reached contacts	getReachedContacts()	Gets the full list of reached contacts
Save contacts of a search	saveSearch()	Saves the selected contacts of a search
Retrieve saved contacts of a search	getSearch()	Gets the contacts of a search saved by the users
Attach note	attachNote()	Attaches a note written by the user to a specific contact
Attach feedback	attachFeedback()	Attaches a feedback written by the user to a specific contact
Retrieve prospects statistics	getProspectsStatistics()	Gets the statistics regarding prospects
Retrieve contacts statistics	getContactsStatistics()	Gets the statistics regarding contacts
Retrieve list of contacted prospects	getReachedProspects()	Gets the list of reached prospects
Login	login()	Performs the login of the user
Logout	logout()	Performs the logout of the user

Le restanti tabelle, in numero pari alle system operation identificate, descrivono dettagli tecnici quali parametri di input e output, precondizioni e postcondizioni.

Tabella 5.4: System operation searchContacts().

Operation	searchContacts(companyName, locationFilter, rolesFilter)
Returns	contactsList

Continua ...

... Continuazione della tabella

Pre-conditions	• The user is logged-in • The user is in the search page • The user has filled the company input field • The user has potentially filled the filters input fields
Post-conditions	• A search of contacts is performed • A list of contacts is shown on the screen

Tabella 5.5: System operation saveEmailTemplate().

Operation	saveEmailTemplate(emailTemplate)
Returns	hasEmailTemplateSuccessfullySaved
Pre-conditions	• The user is logged-in • The user is in the email template page • The user has filled the text box following the instructions

Continua ...

... Continuazione della tabella

Post-conditions	• A The email template is saved by the system • A success or failure message is shown on the screen

Tabella 5.6: System operation sendEmail().

Operation	sendEmail(contactId)
Returns	hasEmailSuccessfullyTakenOver
Pre-conditions	• The user is logged-in • The user has configured the email template • The user is in the search page or in the saved searches page • The user sees a list of contacts on the screen
Post-conditions	• A prospecting email sending is taken over by the system • The email could be sent to the contacts with delay • A success or failure message is shown on the screen to warn the user if thesending of the email has been taken over by the system

Tabella 5.7: System operation getReachedContacts().

Operation	getReachedContacts()
Returns	contactList
Pre-conditions	• The user is logged-in
Post-conditions	• The user can see a list of reached contacts with contact details and atimestamp that represents the time when the email has been sent and an addfeedback button • The user can filter them by time frames and role and name and surname andprospect name

Tabella 5.8: System operation saveSearch().

Operation	saveSearch(contactList)
Returns	searchId
Pre-conditions	• The user is logged-in • The user is in the search page • The user sees a list of contacts on the screen

Continua ...

... Continuazione della tabella

Post-conditions	• The selected contacts are saved as a saved search by the system • A success or failure message is shown on the screen

Tabella 5.9: System operation getSearch().

Operation	getSearch(searchId)
Returns	contactsList
Pre-conditions	• The user is logged-in • The user is in the saved searches page • The user has possibly filtered saved searches by time frames and prospect name
Post-conditions	• The user can see a list of contacts belonging to the search

Tabella 5.10: System operation attachNote().

Operation	attachNote(note, contactId)
Returns	noteId

Continua ...

... Continuazione della tabella

Pre-conditions	• The user is logged-in • The user is in the saved searches page or in the reached contacts page • The user sees a list of contacts
Post-conditions	• The note is attached to the contact by the system

Tabella 5.11: System operation attachFeedback().

Operation	attachFeedback(feedback, contactId)
Returns	feedbackId
Pre-conditions	• The user is logged-in • The user is in the saved searches page or in the reached contacts page • The user sees a list of contacts
Post-conditions	• The feedback is attached to the contact by the system

Tabella 5.12: System operation getProspectsStatistics().

Operation	getProspectsStatistics()

Continua ...

... Continuazione della tabella

Returns	prospectsStatistics
Pre-conditions	• The user is logged-in • The user is in the home page
Post-conditions	• The statistics about the number of prospects reached and searched areretrieved

Tabella 5.13: System operation getContactsStatistics().

Operation	getContactsStatistics()
Returns	contactsStatistics
Pre-conditions	• The user is logged-in • The user is in the home page
Post-conditions	• The statistics about the number of contacts reached are retrieved • The statistics can be filtered by time frames

Tabella 5.14: System operation getReachedProspects().

Operation	getReachedProspects()
Returns	reachedProspectsList
Pre-conditions	• The user is logged-in • The user is in the home page
Post-conditions	• The list of prospects ordered by number of contacts reached is retrieved • The list can be filtered by time frames

Tabella 5.15: System operation login().

Operation	login()
Returns	success
Pre-conditions	• The user is logged-out
Post-conditions	• The user is signed in by the system

Tabella 5.16: System operation logout().

Operation	logout()
Returns	success
Pre-conditions	• The user is logged-in
Post-conditions	• The user is signed out by the system

Identificazione dei microservizi Il secondo step dell'algoritmo di decomposizione consiste nell'identificare i microservizi partendo da un dominio specifico.

Come introdotto nella Sezione 2.2 esistono due approcci per questa fase. In questo caso è stato scelto il primo, ossia la decomposition by business capability. Come già illustrato, all'interno di un progetto software, le business capability possono essere viste come le funzionalità di alto livello indispensabili e che generano valore per il prodotto finale. Avendo una buona conoscenza del dominio di partenza, individuare tutte le business capability e derivare da esse i microservizi diventa un processo relativamente agevole. Spesso, ogni microservizio è associato direttamente a una business capability, ma in alcuni casi risulta vantaggioso aggregare più business capability in un singolo microservizio.

Come evidenziato nella Figura 5.2 , la colonna a sinistra è riservata alla *Capability hierarchy*. In pratica, una business capability può essere suddivisa in diverse business capability di livello inferiore. Ad ogni capability di livello inferiore, o alternativamente ad una di livello superiore, vien fatto corrispondere un microservizio. L'elenco dei microservizi identificati è presentato nella colonna destra denominata *Microservices* e ciascuna corrispondenza tra capability e microservizio è indicata mediante una freccia.

Definizione delle API dei microservizi L'ultima fase dell'algoritmo riguarda la definizione delle API per i microservizi identificati precedentemente. A questo scopo, è essenziale associare ciascuna system operation a un microservizio, indicando quindi quali API ciascun microservizio espone e quali operazioni può supportare.

Un approccio semplificato per eseguire queste associazioni potrebbe essere basato sulla tipologia della system operation presa in considerazione. Ad esempio, una system operation potrebbe generare dati da memorizzare in un database o, alternativamente, potrebbe richiedere la lettura di dati memorizzati in passato per completare il proprio compito. Nel primo caso, potrebbe essere conveniente associare la system operation

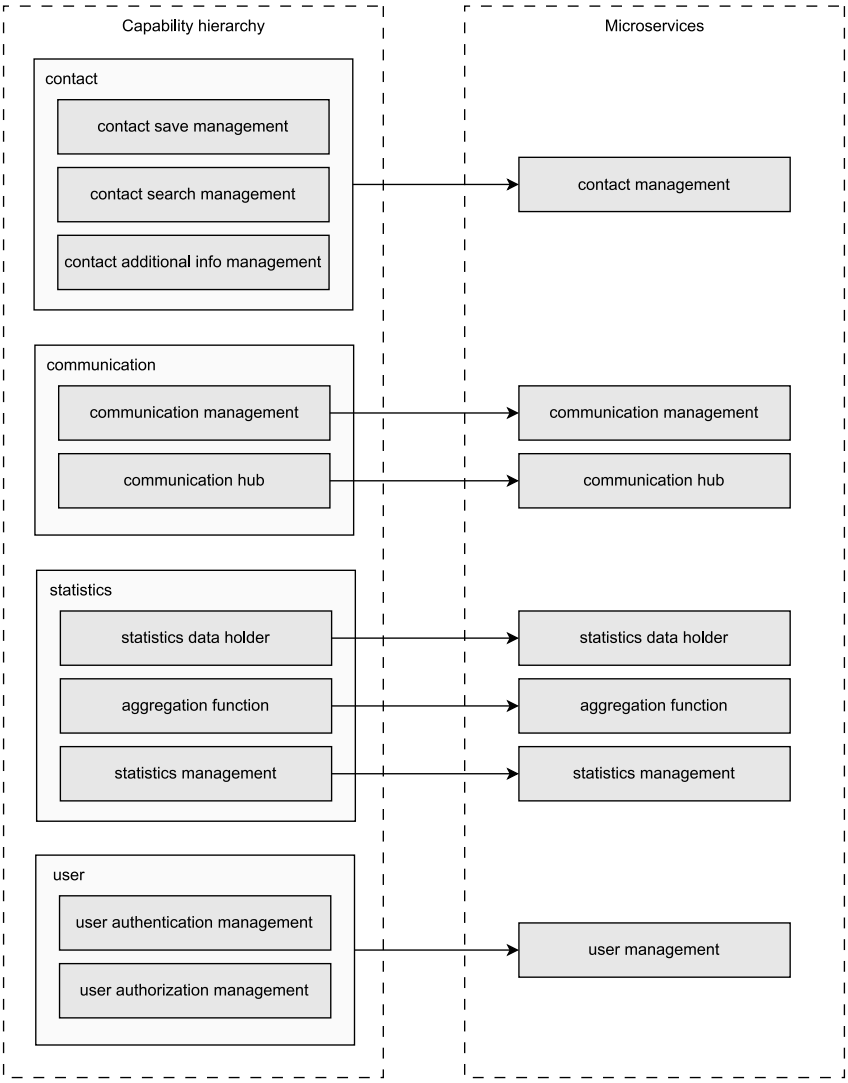

Figura 5.2. Decomposizione in microservizi

al microservizio che gestisce il database in cui memorizzare i dati. Nel secondo, invece, potrebbe essere più opportuno associare la system operation al microservizio che detiene le informazioni necessarie. Questa pratica garantisce una chiara divisione di responsabilità tra i vari microservizi e contribuisce a ridurre la latenza derivante dalla comunicazione tra di essi. Va notato che esistono anche operazioni di sistema ibride, che richiedono sia la lettura sia la generazione di dati. In tali casi, è necessario ponderare attentamente i pro e i contro delle diverse associazioni.

Durante questa fase, è importante considerare anche le collaborazioni tra i microservizi. Potrebbe essere infatti necessario coinvolgere più di un microservizio per completare una richiesta. Ciò porta ad una degradazione delle prestazioni dovuta alla

comunicazione tra diversi processi (i microservizi). In teoria, si tratta di uno scenario da evitare, ma in alcune situazioni, può rappresentare l'unica soluzione praticabile.

Tutti i risultati acquisiti in questa fase possono essere presentati in una tabella a tre colonne. La prima colonna identifica l'operazione di sistema, la seconda indica l'entry point, ossia il microservizio che espone l'API all'esterno, mentre la terza specifica le eventuali collaborazioni necessarie per portare a termine la system operation. La Tabella 5.17 riassume i risultati ottenuti durante la progettazione delle API del progetto.

Tabella 5.17: API dei microservizi.

Operation	**Service**	**Collaborations**
searchContacts()	contact management	
saveEmailTemplate()	communication hub	
sendEmail()	communication management	communication hub sendEmail()
getReachedContacts()	contact management	
saveSearch()	contact management	
getSearch()	contact management	
attachNote()	contact management	
attachFeedback()	communication management	
getProspectsStatistics()	statistics management	
getContactsStatistics()	statistics management	
getReachedProspects()	statistics management	
login()	user management	
logout()	user management	

Descrizione dei microservizi e delle loro interazioni

Come già menzionato nell'introduzione della sezione corrente, di seguito verranno descritti i singoli microservizi, esaminando contemporaneamente le loro interazioni collaborative. Tale analisi sarà condotta nei paragrafi successivi, facendo riferimento alla Figura 5.3, in cui ciascun microservizio è rappresentato da un esagono e ogni servizio di terze parti è denotato da un ottagono. Il quadrato al centro della figura simboleggia il broker per la comunicazione asincrona. Le interazioni sono illustrate tramite frecce etichettate con *HTTPS* nel caso di comunicazioni sincrone e con *AMQP* in caso di comunicazioni asincrone, rappresentando rispettivamente i protocolli utilizzati. La legenda in alto a destra fornisce informazioni sul tipo di messaggio inviato o ricevuto in

ogni interazione asincrona, implementata seguendo il pattern publish-subscribe. Va notato, inoltre, che il microservizio *user management* non è connesso ad altre componenti presenti in figura poiché il sistema fa uso del *pattern API gateway* [1].

Per comodità, alcuni microservizi strettamente correlati tra loro saranno descritti in un unico paragrafo.

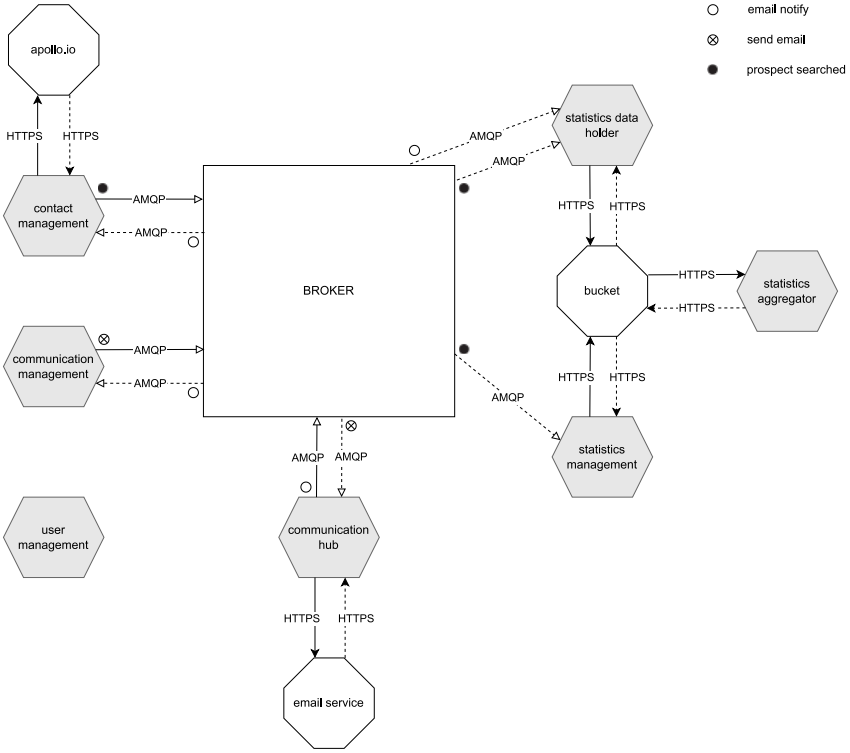

Figura 5.3. Comunicazione tra microservizi

Contact search management Questo microservizio rappresenta il nucleo centrale dell'applicazione; infatti, è evidente, consultando la Tabella 5.17, che si tratta del microservizio con il maggior numero di operazioni assegnate. Esso gestisce integralmente il processo di ricerca di contatti di un'azienda prospect, invocando le API esterne pubblicate dalla piattaforma Apollo.io. La ricerca offre la possibilità di filtrare i dipendenti di un'azienda prospect per location (città, regione, stato) e ruolo all'interno

[1] Il pattern API Gateway è un modello architetturale in cui un componente centrale, noto come API Gateway, si posiziona tra le applicazioni client e i servizi di backend. La sua funzione principale è quella di agire come un punto di ingresso unificato per tutte le richieste API provenienti dai client, gestendo in modo centralizzato le richieste, l'autenticazione, l'autorizzazione, il monitoraggio e l'instradamento verso i servizi appropriati.

dell'azienda, di cui l'utente deve specificare il nome. Inoltre, è possibile richiedere il salvataggio totale o parziale dei contatti ottenuti da una ricerca all'interno del suo database, e naturalmente recuperare ricerche salvate in passato. L'operazione di allegare una nota a contatti precedentemente salvati, consente all'utente di associare loro commenti e promemoria. Un'altra operazione supportata riguarda il recupero della lista di contatti a cui è stata inviata una mail in passato. Oltre alle funzionalità appena descritte, derivate dalle specifiche di alto livello, sono state introdotte altre funzionalità di minore importanza. Tra queste, la cancellazione di ricerche dal database e la possibilità di filtrare la lista di ricerche salvate secondo vari criteri, tra cui intervalli temporali e aziende prospect di cui i contatti fanno parte. Questo facilita all'utente l'individuazione della ricerca desiderata.

Le interazioni con altri microservizi sono necessarie per scambiare informazioni a fini statistici. Infatti, ogni qual volta viene effettuata una ricerca, communication management pubblica un messaggio con il topic prospectSearched. I destinatari di questo messaggio sono i microservizi *statistics data holder* e *statistics management*, che utilizzano questi dati per generare statistiche.

Communication management e communication hub I microservizi *Communication Management* e *Communication Hub* sono responsabili della gestione delle comunicazioni per i contatti individuati. Nello specifico, le operazioni a loro affidate, elencate nella Tabella 5.17, includono la creazione di email template, l'invio di email a uno o più contatti e la generazione di feedback per eventuali contatti che hanno risposto ad una mail di prospecting.

Le prime due funzionalità coinvolgono anche l'interazione con un servizio esterno: la piattaforma *SendGrid*, selezionata per la creazione di email template e l'invio di email. I dettagli relativi a *SendGrid* saranno approfonditi nel prossimo capitolo.

L'operazione saveEmailTemplate() è gestita da Communication Hub. Esso riceve in input un testo che rappresenta l'email template e interagisce con SendGrid tramite REST API. SendGrid, a sua volta, salva il template nel proprio database assegnandogli un id univoco. L'id associato al template viene incluso nella risposta HTTP, e Communication Hub lo registra nel proprio database, associandolo all'id dell'utente corrente. Per maggior chiarezza, l'email template è una struttura preformattata che può essere utilizzata per creare e inviare messaggi di posta elettronica in modo efficiente e consistente. Esso può incorporare tag o segnaposto dinamici che vengono sostituiti automaticamente con dati specifici per ogni destinatario.

L'operation sendEmail(), invece, coinvolge i microservizi communication management, communication hub e SendGrid.

Lato client, l'utente può selezionare uno o più contatti di una ricerca e richiedere al software di inviare loro una mail di prospecting.

Communication Management riceve la richiesta, il cui input è rappresentato da una lista di contatti, ed esegue un controllo nel proprio database, verificando per ciascun contatto se l'utente corrente ha già inviato email di prospecting in passato. In caso

affermativo, il microservizio risponde al client con un alert antispam. L'utente può rispondere forzando l'invio delle email a parte o a tutti i contatti nell'alert, oppure decidere di non contattarli nuovamente. In caso di esito negativo al check o dopo la risposta dell'utente all'alert antispam, per ogni contatto rimanente nella lista, communication management pubblica un messaggio con il topic sendEmail. Il messaggio in questione contiene le informazioni del contatto.

Communication hub, sottoscritto al topic, riceve un messaggio per volta dal broker, estrae le informazioni necessarie (i dati che sostituiscono i tag all'interno dell'email template e gli indirizzi email mittente e destinatario) e, tramite l'id utente, ottiene l'identificativo dell'email template associato all'utente durante la creazione del template. Questo id serve a SendGrid per individuare quale email template tra quelli creati all'interno della piattaforma specializzare. Tale operazione è necessaria in quanto viene usato un singolo account SendGrid per l'intera applicazione, di conseguenza esso ha in memoria gli email template relativi a tutti gli utenti dell'applicazione. Tutte queste informazioni sono incluse nella richiesta HTTP inviata a SendGrid, che identifica l'email template, sostituisce i tag con i dati del destinatario e invia l'email.

La risposta HTTP contiene l'esito dell'invio dell'email e Communication Hub informa gli altri microservizi interessati utilizzando il pattern publish-subscribe. Innanzitutto, communication management utilizza questa informazione per aggiornare il proprio database e contrassegnare un contatto come già contattato o meno, ai fini del check antispam. Contact management, d'altra parte, può aggiornare la lista di contatti ai quali è stata inviata una mail in passato. Infine, statistics data holder memorizza gli esiti positivi a fini statistici.

Statistics data holder, Aggregation function e Statistics management I tre microservizi collaborano per generare dati statistici, utili agli utenti per analizzare le proprie attività e per creare report da presentare ai superiori. Oltre ai tre microservizi vi è un quarto componente esterno, ossia il *bucket*. Brevemente, un bucket è un contenitore virtuale in cui è possibile archiviare e organizzare dati binari e non, ed organizzarli in modo gerarchico. Spesso si opta per bucket in cloud e, in questo caso, è stato adottato l'Azure Blob Storage di Microsoft Azure. Anch'esso verrà descritto meglio nel prossimo capitolo.

Ricapitolando, le statistiche generate dal software sono:

- Il numero di aziende prospect cercate e contattate. Per contattate si intende che almeno una figura all'interno dell'azienda è stata contattata tramite email;
- Il numero di contatti raggiunti tramite email;
- Una lista di aziende prospect ordinate per numero di contatti contattati.

Le prime due tipologie di statistiche possono essere filtrate per periodi temporali, anche chiamati time frame. Queste sono caratterizzate da una granularità di default (minima) giornaliera, modificabile su richiesta dell'utente. L'ultimo tipo di statistica invece è una semplice lista ordinata.

Statistics data holder ha il compito di acquisire e memorizzare i dati che necessitano di essere aggregati. Risulta essere il caso delle statistiche filtrabili per time frame. Effettivamente, esso è sottoscritto ai topic sendEmail e prospectSearched, ricevendo rispettivamente i dati dei contatti a cui è stata inviata una email con successo e le informazioni relative alle aziende prospect cercate. Ogni fine giornata lavorativa, crea due file CSV, uno per tipologia di statistica, li popola con i dati ricevuti durante la giornata, e ne esegue l'upload sull'azure blob storage.

Statistics aggregator ha la responsabilità di aggregare dati producendo delle statistiche con granularità minima, ossia giornaliera. Ciò avviene, ovviamente, dopo aver scaricato i due file CSV dal blob storage e aver estratto i dati contenuti all'interno. In seguito all'aggregazione, esso produce due nuovi file CSV popolati dai nuovi dati (aggregati) e li carica nello storage.

Statistics management è l'unico tra i tre microservizi ad essere raggiungibile da un'applicazione client. È pertanto responsabile della gestione delle richieste relative alla lettura di statistiche. A tale scopo, deve inizialmente archiviare all'interno del proprio database i dati statistici generati da Statistics Aggregator, i quali sono contenuti nei file CSV presenti nel blob storage. In aggiunta riceve anch'esso, tramite il broker, le informazioni delle aziende prospect cercate e le archivia direttamente nel proprio database, senza effettuare alcuna aggregazione. Ciò consente la generazione successiva della lista delle aziende prospect con i relativi dettagli. Quando statistics management riceve una richiesta da un client, elabora le statistiche, combinando i dati necessari e, in caso di richieste con una granularità superiore a quella minima, esegue ulteriori operazioni di aggregazione.

Esistono due considerazioni importanti da menzionare, riguardanti i tre microservizi descritti in questo paragrafo.

La prima riguarda il microservizio Statistics Aggregator, il quale è stato implementato attraverso l'utilizzo del servizio PaaS di Microsoft Azure noto come Azure Function. In breve, questo rappresenta una soluzione serverless che consente di scrivere codice senza preoccuparsi dell'infrastruttura sottostante. Si tratta di codice deployato nel cloud e attivato in risposta a eventi specifici, nel nostro caso, l'upload di file CSV nel blob storage.

La seconda precisazione riguarda l'architettura e il flusso logico di questa parte del software. Un'eventuale implementazione sarebbe potuta essere quella di un singolo microservizio in grado di memorizzare tutti i dati ricevuti dal broker e che, ricevuta una richiesta da un client, sia in grado di aggregarli in un solo step, senza effettuare una aggregazione preliminare giornaliera. Una soluzione del genere presenta diverse problematiche, specialmente se il numero di utenti scala considerevolmente. Il primo problema riguarda la latenza percepita dell'utente, causata principalmente dalla mole di dati da aggregare, soprattutto nel caso in cui la granularità temporale è elevata. Pertanto, la decisione di aggregare i dati iniziali al termine di ogni giornata lavorativa appare come una soluzione valida per ridurre il tempo di risposta verso un'applicazione client. D'altra parte, la scelta di suddividere il processo tra i tre componenti software

anziché affidarlo a un singolo microservizio è motivata dalla necessità di distribuire i dati su più database.

Inoltre, a scopo formativo, si è deciso di coinvolgere nel processo i due servizi esterni, ossia l'Azure Function e l'Azure Blob Storage, che si sposano perfettamente con una soluzione più modulare.

5.3 Definizione dell'architettura cloud

La Figura 5.4 illustra l'architettura cloud dell'applicazione, fornendo una panoramica della disposizione e delle interconnessioni tra i componenti software e le risorse cloud utilizzate. Dal lato sinistro, si segue il flusso di comunicazione tipico di un'architettura client-server.

L'applicazione client, rappresentata dal quadrato di sinistra, si connette inizialmente all'Azure Active Directory (AAD), un servizio cloud dedicato alla gestione delle identità e dell'accesso (IAM). AAD offre servizi di autenticazione e autorizzazione per garantire che solo utenti autorizzati possano accedere alle risorse.

Successivamente, troviamo l'Azure API Management, che funge da API Gateway, semplificando, proteggendo e monitorando l'accesso alle API dei microservizi. Gestisce l'instradamento delle richieste dell'applicazione client verso i rispettivi microservizi dietro le quinte, come evidenziato da un collegamento a più frecce ramificate.

Il nucleo dell'applicazione è costituito dai microservizi, eseguiti in container Docker e gestiti attraverso Azure Kubernetes Service (AKS). È importante notare che i diversi microservizi possono utilizzare tipi di database differenti: Contact Management, Communication Management e Communication Hub adoperano SQL, un database relazionale, mentre Statistics Management e Statistics Data Holder utilizzano Cosmos DB, un database non relazionale di Azure.

La comunicazione asincrona è gestita dall'Azure Service Bus (rappresentato dal cilindro a destra), che facilita una comunicazione efficiente mediante il modello publish-subscribe.

Nella parte superiore della figura, sono evidenziati i servizi cloud impiegati per la produzione di statistiche: l'Azure Function, che sostituisce il microservizio Statistics Aggregator, e l'Azure Blob Storage, utilizzato per la condivisione di file CSV.

Infine, nella parte inferiore della figura, sono indicati i servizi esterni utilizzati per la ricerca di contatti e l'invio di email: Apollo.io e SendGrid.

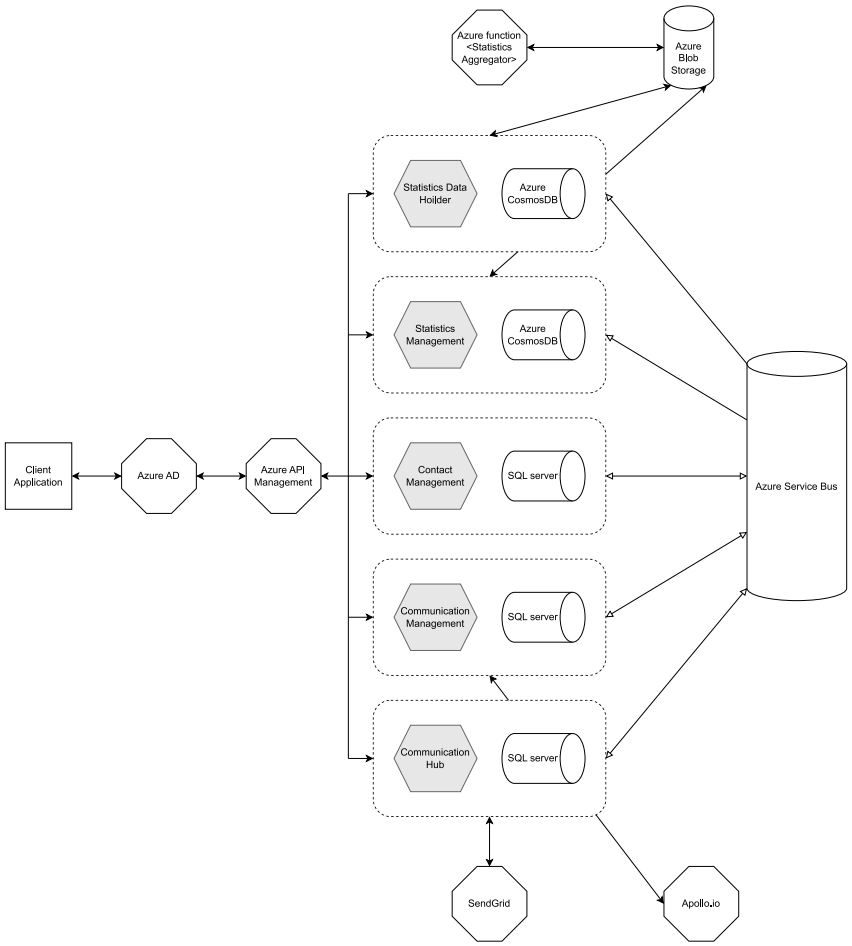

Figura 5.4. Architettura cloud

Capitolo 6

Tecnologie e strumenti utilizzati

Il successo di un progetto software dipende non solo dalla competenza e dall'impegno del team di sviluppo, ma anche dalla scelta e dall'utilizzo delle giuste tecnologie e strumenti. Questo capitolo esplorerà le diverse tecnologie, strumenti e servizi utilizzati nel corso del progetto, delineando il ruolo cruciale che hanno giocato nel raggiungimento degli obiettivi prefissati.

La sezione *Strumenti di sviluppo e deployment* si concentrerà sull'analisi dettagliata degli strumenti adottati per la scrittura del codice, la gestione delle versioni, e il deployment del software. Ogni componente avrà il suo spazio dedicato, evidenziando il loro impatto sul flusso di lavoro e sulla qualità del prodotto finale.

Successivamente, nella sezione *Servizi Azure*, verranno esaminati i servizi cloud offerti da Microsoft Azure e la loro integrazione nel progetto. Azure ha svolto un ruolo fondamentale nella fornitura di infrastruttura scalabile, servizi di storage affidabili, e strumenti per lo sviluppo e la gestione delle applicazioni. Saranno discussi i servizi utilizzati, come Azure Functions, Azure Service Bus, Azure SQL Database e altri.

6.1 Strumenti di sviluppo e deployment

Questa sezione esaminerà gli strumenti chiave utilizzati durante lo sviluppo e il deployment del progetto software. Ogni strumento ha svolto un ruolo essenziale nel supportare il processo di sviluppo e distribuzione del software. Saranno presentate le funzionalità e l'utilizzo di ciascuno strumento.

6.1.1 Microsoft Visio

Microsoft Visio è un software di disegno e per la creazione di diagrammi, sviluppato da Microsoft. Progettato per aiutare gli utenti a creare diagrammi professionali e

schemi visivi in modo semplice e intuitivo, Visio è utilizzato ampiamente in ambito aziendale per rappresentare informazioni complesse in modo chiaro e comprensibile. Alcune caratteristiche principali di Microsoft Visio includono:

- *Creazione di diagrammi professionali*: Visio offre una vasta gamma di modelli e forme predefinite per la creazione di diagrammi di flusso, organigrammi, mappe concettuali, schemi di rete e altro ancora;
- *Integrazione con Microsoft Office*: si integra nativamente con altri prodotti Microsoft Office, consentendo un flusso di lavoro fluido per l'inserimento di diagrammi in documenti Word, presentazioni PowerPoint e fogli Excel;
- *Collaborazione in tempo reale*: permette la collaborazione in tempo reale su diagrammi, facilitando la condivisione e la revisione di progetti con membri del team;
- *Personalizzazione avanzata*: offre opzioni avanzate di personalizzazione per la formattazione di forme, linee, colori e stili, consentendo agli utenti di adattare i diagrammi alle proprie esigenze;
- *Diagrammi di processo e progettazione tecnica*: Visio è ampiamente utilizzato per la creazione di diagrammi di processo, diagrammi UML e schemi tecnici, supportando diversi ambiti applicativi;
- *Esportazione e condivisione*: consente l'esportazione dei diagrammi in vari formati, inclusi PDF e immagini, facilitando la condivisione dei risultati con altri utenti;
- *Template e forme personalizzate*: gli utenti possono creare modelli personalizzati e forme, offrendo flessibilità nella rappresentazione visiva delle informazioni.

Microsoft Visio è stato impiegato come strumento principale per creare i diagrammi durante la fase di progettazione, come illustrato nel capitolo precedente. In particolare, l'high level domain model, lo schema relativo alla decomposizione in microservizi, il diagramma che illustra la comunicazione tra quest'ultimi e il diagramma dell'architettura cloud sono stati creati utilizzando questo programma. Successivamente, al fine di garantire uno stile uniforme tra le diverse immagini, essi sono stati replicati utilizzando Draw.io durante la redazione della tesi.

6.1.2 Visual Studio Code

Visual Studio Code (*VS Code*) è un ambiente di sviluppo leggero e open source fornito da Microsoft, progettato per supportare una vasta gamma di linguaggi di programmazione. Alcune caratteristiche principali di Visual Studio Code includono:

- *Editor di testo avanzato*: Visual Studio Code fornisce un editor di testo avanzato con funzionalità come evidenziazione della sintassi, completamento automatico, refactoring e navigazione efficiente nel codice;

- *Estensioni e plugin*: supporta l'installazione di molte estensioni e plugin, consentendo agli sviluppatori di personalizzare e estendere le funzionalità dell'IDE in base alle proprie esigenze;
- *Debugger integrato*: integra un debugger per vari linguaggi di programmazione, semplificando il processo di individuazione e correzione degli errori nel codice;
- *Controllo versione integrato*: offre supporto integrato per sistemi di controllo versione come Git, consentendo una gestione efficace delle modifiche al codice;
- *Terminale integrato*: include un terminale integrato che consente agli sviluppatori di eseguire comandi direttamente dall'IDE, migliorando l'efficienza dello sviluppo;
- *Integrazione con servizi Azure*: si integra nativamente con Azure, semplificando lo sviluppo, il debug e il rilascio di applicazioni per il cloud di Microsoft;
- *Multi-piattaforma*: disponibile per Windows, macOS e Linux, garantendo un'esperienza coerente su diverse piattaforme;

Naturalmente, VS Code è stato adottato per l'implementazione dei microservizi e dell'Azure Function, in seguito deployata in cloud.

6.1.3 Postman

Postman è un'applicazione di sviluppo API ampiamente utilizzata, progettata per semplificare il processo di creazione, testing e gestione di API. Questo strumento facilita la collaborazione tra sviluppatori e team, offrendo le seguenti funzionalità chiave:

- *Creazione di richieste API*: gli utenti possono facilmente definire richieste HTTP personalizzate specificando metodi, parametri e header;
- *Ambiente di test*: Postman permette l'esecuzione dei test delle API, semplificando la verifica delle risposte e la gestione delle aspettative;
- *Documentazione automatica*: genera automaticamente documentazione dettagliata per le API, facilitando la comprensione e l'utilizzo;
- *Gestione delle variabili*: supporta l'uso di variabili per rendere dinamici gli ambienti e semplificare la gestione di configurazioni diverse;
- *Collaborazione e condivisione*: agevola la collaborazione consentendo la condivisione di collezioni di API, richieste e ambienti.

Nel contesto del progetto, Postman è utilizzato per la creazione di collezioni di API e per eseguire test di integrazione sugli endpoint REST.

6.1.4 Microsoft SQL Server Management Studio

Microsoft SQL Server Management Studio (*SSMS*) è un ambiente integrato di gestione e sviluppo per Microsoft SQL Server. Progettato per semplificare la gestione del database e lo sviluppo di query SQL. Tra le principali funzionalità vi sono:

- *Interfaccia grafica intuitiva*: SSMS fornisce un'interfaccia utente intuitiva per la gestione di database, tabelle, stored procedure e altri oggetti;
- *Editor di query*: include un editor di query SQL con evidenziazione della sintassi, suggerimenti automatici e strumenti di formattazione del codice;
- *Strumenti di amministrazione*: offre strumenti per la configurazione del server, la gestione di utenti e autorizzazioni, il monitoraggio delle prestazioni e la manutenzione del database;
- *Debugger integrato*: consente il debug delle stored procedure e delle query per facilitare lo sviluppo e l'ottimizzazione del codice SQL;
- *Esplorazione degli oggetti*: permette di esplorare la struttura del database e visualizzare le relazioni tra gli oggetti del database;

SSMS è principalmente impiegato per verificare l'esecuzione corretta del seeding e delle query effettuate attraverso Prisma, il framework ORM.

6.1.5 Git

Git è un sistema di controllo di versione distribuito, ampiamente utilizzato per il tracciamento delle modifiche al codice sorgente durante lo sviluppo software. Fornisce un efficace sistema di gestione del codice, consentendo agli sviluppatori di collaborare in modo efficiente. Alcune caratteristiche chiave di Git includono:

- *Branching e merging*: Git permette la creazione di rami di sviluppo separati, i branch, consentendo agli sviluppatori di lavorare su diverse funzionalità in parallelo e successivamente integrare le modifiche nel branch principale tramite il merging;
- *Repository distribuito*: ogni copia locale di un repository Git contiene l'intero storico delle modifiche, facilitando il lavoro in modalità offline e la collaborazione distribuita;
- *Staging area*: gli sviluppatori possono preparare le modifiche prima di effettuare il commit, consentendo una maggiore precisione e controllo sulle modifiche incluse nell'aggiornamento della repository;
- *Rilevamento delle modifiche*: Git traccia le modifiche a livello di singolo carattere, rendendo possibile il rilevamento preciso delle modifiche apportate ai file;

- *GitHub e GitLab*: Piattaforme come GitHub e GitLab forniscono servizi di hosting per repository Git, semplificando la collaborazione e la condivisione del codice tra gli sviluppatori;

- *Storia delle modifiche*: Git mantiene una cronologia completa delle modifiche, facilitando il rollback a versioni precedenti e la comprensione dell'evoluzione del progetto nel tempo.

In questo progetto, sono utilizzate le Azure Repos, integrate all'interno di Azure DevOps per gestire la versione del codice sorgente. Lavorando sul progetto in modo individuale, molte delle potenzialità offerte da Git non sono state sfruttate appieno.

6.1.6 Docker Desktop

Docker Desktop è un'applicazione di virtualizzazione che semplifica la creazione, la distribuzione e l'esecuzione di container software. Fornendo un ambiente isolato per le applicazioni, Docker consente agli sviluppatori di confezionare un'applicazione insieme a tutte le sue dipendenze in un container, garantendo la coerenza tra ambienti di sviluppo, test e produzione. Alcune caratteristiche chiave di Docker Desktop includono:

- *Containerizzazione*: Docker utilizza il concetto di container per isolare applicazioni e servizi, garantendo la portabilità e la consistenza dell'esecuzione;

- *Immagini Docker*: gli sviluppatori possono creare immagini Docker, che contengono tutte le risorse necessarie per l'esecuzione di un'applicazione, semplificando il processo di distribuzione;

- *Orchestratori*: Docker supporta orchestratori come Docker Compose e Kubernetes per gestire la distribuzione e la scalabilità delle applicazioni containerizzate;

- *Integrazione con Visual Studio Code*: Docker Desktop offre un'esperienza integrata con Visual Studio Code, facilitando lo sviluppo e il debug di applicazioni in container;

- *Monitoraggio delle risorse*: fornisce strumenti per monitorare l'utilizzo delle risorse da parte dei container, facilitando l'ottimizzazione delle prestazioni;

- *Multi-piattaforma*: Docker Desktop è disponibile per Windows, macOS e Linux, garantendo un'esperienza uniforme su diverse piattaforme.

Docker Desktop è stato impiegato principalmente per eseguire in locale istanze di SQL Server e, soprattutto, per creare i container di ogni microservizio, poi distribuiti sul cluster Kubernetes di Microsoft Azure, tramite il servizio AKS (Azure Kubernetes Service).

6.1.7 Prisma

Prisma è un framework ORM (Object-Relational Mapping) moderno e open source, progettato per semplificare l'interazione tra un'applicazione e un database relazionale. Con Prisma, gli sviluppatori possono lavorare con il database utilizzando un'interfaccia tipizzata e intuitiva in linguaggio di programmazione TypeScript o JavaScript. Le principali caratteristiche di Prisma includono:

- *Definizione del modello*: gli sviluppatori possono definire i modelli dei dati utilizzando una sintassi dichiarativa, semplificando la creazione e la gestione degli schemi del database;
- *Query tipizzate*: Prisma genera automaticamente query tipizzate basate sul modello definito, migliorando la sicurezza e l'autocompletamento durante lo sviluppo;
- *Relazioni e join*: supporta in modo nativo la definizione di relazioni complesse tra tabelle, semplificando le operazioni di join e migliorando le prestazioni delle query;
- *Migrazioni del database*: Prisma offre strumenti per gestire facilmente le migrazioni del database, consentendo agli sviluppatori di aggiornare lo schema in modo controllato;
- *Interfaccia della console*: include un'interfaccia da riga di comando (CLI) per semplificare la gestione del database, la generazione di codice e l'esecuzione di migrazioni;
- *Supporto multi-database*: Prisma supporta vari database relazionali, tra cui PostgreSQL, MySQL e SQLite.

I microservizi che operano con un database SQL integrano Prisma per semplificare la gestione delle operazioni di accesso e manipolazione dei dati nel database oltre che per consentire di acquisire familiarità con l'approccio ORM.

6.1.8 Apollo.io

Apollo.io è una piattaforma di intelligenza commerciale progettata per arricchire le strategie di vendita, marketing e gestione dei clienti, con particolare enfasi sul prospecting. La piattaforma offre un'ampia gamma di strumenti avanzati per l'acquisizione, la gestione e l'analisi dei dati commerciali, specializzandosi nella ricerca e nell'identificazione di nuovi potenziali clienti. Alcuni punti chiave della piattaforma e delle sue API includono:

- *Arricchimento dati*: Apollo.io consente di arricchire i dati sui contatti e sulle aziende, fornendo informazioni dettagliate che supportano le attività di vendita, marketing e prospecting;

- *Segmentazione avanzata*: la piattaforma offre funzionalità di segmentazione avanzata, facilitando l'individuazione di nuove opportunità di business attraverso una precisa suddivisione dei target;
- *Automazione del flusso di lavoro*: con strumenti di automazione integrati, Apollo.io semplifica i flussi di lavoro di prospecting, ottimizzando le attività di identificazione e acquisizione di nuovi clienti;
- *Monitoraggio comportamentale*: tramite l'analisi dei comportamenti degli utenti, Apollo.io fornisce insight preziosi per comprendere meglio le esigenze dei prospect e personalizzare le interazioni;
- *API estensive*: le APIs di Apollo.io consentono l'integrazione della piattaforma con altri strumenti e applicazioni, facilitando lo scambio di dati e l'automazione di processi chiave nel contesto specifico;
- *Analisi delle performance*: la piattaforma fornisce analisi dettagliate sulle performance delle attività di prospecting, aiutando a ottimizzare le strategie di acquisizione e qualificazione di nuovi prospect.

Nel progetto si fa uso delle API pubbliche della piattaforma per eseguire ricerche avanzate di aziende e contatti.

6.1.9 SendGrid

SendGrid è una piattaforma di servizi di posta elettronica basata su cloud, progettata per semplificare l'invio e la gestione delle email transazionali e di marketing. Rivolgendosi principalmente a sviluppatori e aziende, SendGrid offre un'infrastruttura affidabile e scalabile per garantire la consegna sicura delle email. Alcune caratteristiche chiave di SendGrid includono:

- *Invio di email*: SendGrid consente agli sviluppatori di integrare facilmente la piattaforma per inviare email;
- *Email di marketing*: supporta la creazione e l'invio di campagne di email marketing, offrendo strumenti per la personalizzazione e il tracciamento delle prestazioni;
- *API robuste*: SendGrid fornisce un'API completa per l'automazione dell'invio delle email, consentendo un'integrazione semplice con applicazioni e siti web;
- *Gestione delle liste*: include funzionalità per la gestione delle liste di contatti, consentendo la segmentazione e la personalizzazione delle campagne;
- *Tracciamento e analisi*: offre strumenti avanzati per monitorare il successo delle email inviate, con metriche apposite;

- *Sicurezza delle email*: implementa misure di sicurezza avanzate per proteggere le email dagli attacchi di phishing e spam;

- *Template Dinamici*: SendGrid supporta l'utilizzo di template dinamici, permettendo l'inclusione di tag sostituibili con dati reali del contatto, migliorando così la personalizzazione e la rilevanza delle email inviate.

Come già illustrato nel capitolo precedente, SendGrid viene impiegato per automatizzare l'invio di email attraverso l'utilizzo di template dinamici, i quali vengono specializzati con le informazioni del contatto destinatario.

6.2 Servizi Azure

Nella sezione dedicata ai Servizi Azure, saranno esaminati l'integrazione e l'utilizzo dei servizi cloud offerti dalla piattaforma Microsoft Azure nel contesto del progetto. Azure ha fornito un'ampia gamma di servizi, tra cui archiviazione, messaggistica e orchestrazione, supportando così la scalabilità e l'affidabilità del software.

6.2.1 Azure SQL server

Azure SQL Server è un servizio di database relazionale basato su cloud e offerto da Microsoft Azure. Progettato per garantire scalabilità, sicurezza e flessibilità, Azure SQL Server semplifica la gestione di database relazionali senza dover gestire direttamente l'infrastruttura sottostante. Le sue caratteristiche principali includono:

- *Elasticità*: Azure SQL Server consente di ridimensionare facilmente le risorse del database in base alle esigenze, adattandosi dinamicamente alle variazioni di carico;

- *Gestione Centralizzata*: fornisce un'interfaccia di gestione centralizzata tramite il portale Azure, semplificando la configurazione, il monitoraggio e la manutenzione del database;

- *Backup e Ripristino Automatici*: implementa automaticamente operazioni di backup e ripristino, garantendo la continuità operativa e la protezione dei dati;

- *Sicurezza Avanzata*: integra misure di sicurezza avanzate, tra cui crittografia dei dati in riposo e in transito, controlli di accesso e funzionalità di auditing;

- *Ripristino*: consente il ripristino dei database a un punto specifico nel tempo, fornendo una maggiore flessibilità nella gestione delle modifiche;

- *Integrazione con Servizi Azure*: si integra con altri servizi Azure, consentendo lo sviluppo di soluzioni complete basate su cloud.

6.2.2 Azure Cosmos DB

Azure Cosmos DB è un servizio di database multi-modello, fornito da Microsoft Azure. Progettato per gestire dati di varie tipologie, tra cui documenti, grafici, tabelle e colonne, Cosmos DB offre una soluzione scalabile, distribuita globalmente e ad alta disponibilità. Alcune caratteristiche principali di Azure Cosmos DB includono:

- *Dati multi-modello*: supporta modelli di dati flessibili, consentendo la gestione di documenti JSON, grafi, colonne e tabelle di chiavi-valore all'interno dello stesso servizio;
- *Distribuzione globale*: Azure Cosmos DB offre la possibilità di distribuire dati in modo trasparente su più regioni geografiche, garantendo bassa latenza e alta disponibilità in tutto il mondo;
- *Scalabilità orizzontale automatica*: il servizio è progettato per scalare automaticamente in modo orizzontale per gestire picchi di carico e crescita dei dati senza interruzioni del servizio;
- *Indicizzazione automatica*: Cosmos DB gestisce automaticamente l'indicizzazione per supportare query efficienti su grandi volumi di dati, migliorando le prestazioni delle interrogazioni;
- *Integrazione con servizi Azure*: si integra nativamente con altri servizi Azure, consentendo agli sviluppatori di costruire soluzioni end-to-end e sfruttare funzionalità aggiuntive del cloud di Azure;
- *API multi-modello*: supporta diverse API, tra cui SQL, MongoDB, Gremlin, Cassandra e Table.

Cosmos DB è utilizzato dai due microservizi: Statistics Management e Statistics Data Holder, utilizzando un approccio non relazionale basato su documenti.

6.2.3 Azure Blob Storage

Azure Blob Storage è un servizio di archiviazione di oggetti fornito da Microsoft Azure, progettato per gestire grandi quantità di dati non strutturati come immagini, video, documenti e file di grandi dimensioni. Offre uno spazio di archiviazione affidabile, scalabile e accessibile tramite API RESTful. Alcune caratteristiche principali di Azure Blob Storage includono:

- *Categorie di Blob*: supporta diverse categorie di blob, tra cui Blob di blocchi per grandi oggetti, Blob di pagina per dati sequenziali e Blob di archiviazione per l'archiviazione a lungo termine;
- *Distribuzione globale*: consente di distribuire dati e archiviare oggetti in modo distribuito su più regioni geografiche, garantendo la ridondanza e la disponibilità globale;

- *Accesso sicuro*: implementa controlli di accesso e autorizzazioni, consentendo di gestire in modo sicuro l'accesso ai dati e la condivisione di risorse;
- *Versioning e Snapshot*: supporta il versioning degli oggetti e la creazione di snapshot, fornendo un controllo più granulare sulla gestione delle modifiche dei dati;
- *Strumenti di gestione*: fornisce strumenti di gestione e monitoraggio tramite il portale di Azure, PowerShell, Azure CLI e SDK, semplificando la configurazione e il monitoraggio dell'archiviazione di oggetti;
- *Sicurezza avanzata*: implementa funzionalità avanzate di sicurezza, tra cui crittografia dei dati in riposo e in transito, garantendo la protezione dei dati sensibili.

Azure Blob Storage è stato impiegato come storage per la memorizzazione e condivisione dei file CSV relativi alla produzione di dati statistici all'interno del software.

6.2.4 Azure Service Bus

Azure Service Bus è un servizio di messaggistica completamente gestito offerto da Microsoft Azure, progettato per facilitare la comunicazione affidabile e scalabile tra applicazioni distribuite. Fornisce meccanismi di messaggistica asincrona e sincrona, supportando la comunicazione tra componenti di applicazioni distribuite e microservizi. Alcune caratteristiche principali di Azure Service Bus includono:

- *Code di messaggi*: consente la creazione di code di messaggi per gestire la comunicazione asincrona tra mittenti e destinatari, garantendo la consegna affidabile dei messaggi;
- *Topic di messaggi*: supporta l'uso di topic per consentire la pubblicazione e la sottoscrizione di messaggi su più argomenti, facilitando la gestione dei flussi di lavoro complessi;
- *Filtri e regole*: implementa filtri e regole di sottoscrizione per consentire la selezione specifica dei messaggi in base a criteri definiti, migliorando la flessibilità nell'elaborazione dei messaggi;
- *Dead-Lettering*: gestisce automaticamente i messaggi che non possono essere elaborati, spostandoli in una coda separata (dead-letter queue) per l'analisi e la risoluzione dei problemi;
- *Transazioni*: supporta transazioni di messaggistica per garantire la coerenza e l'integrità nelle operazioni che coinvolgono più messaggi;
- *Sicurezza Avanzata*: implementa funzionalità di sicurezza avanzate, tra cui autorizzazioni basate su ruoli e crittografia dei dati in transito.

Azure Service Bus è utilizzato all'interno del progetto per favorire la comunicazione asincrona, interna al cluster di microservizi, tramite il pattern publish-subscribe.

6.2.5 Azure Functions

Azure Functions è un servizio serverless offerto da Microsoft Azure che consente agli sviluppatori di scrivere, implementare e gestire codice senza doversi preoccupare dell'infrastruttura sottostante. Con Azure Functions, è possibile eseguire il codice in risposta a eventi senza la necessità di un server dedicato, pagando solo per il tempo di esecuzione effettivo. Alcune caratteristiche principali di Azure Functions includono:

- *Serverless*: Azure Functions offre un ambiente serverless, consentendo agli sviluppatori di concentrarsi solo sul codice senza la necessità di gestire l'infrastruttura sottostante;
- *Supporto multi-linguaggio*: supporta diversi linguaggi di programmazione, tra cui C#, JavaScript, Python e altri, fornendo flessibilità nella scelta del linguaggio preferito;
- *Triggers e bindings*: una Azure Function può essere attivato da vari trigger, come eventi di archiviazione di Azure, trigger HTTP, timer e molti altri. I binding semplificano l'interazione con altri servizi di Azure;
- *Scalabilità automatica*: Azure Functions offre scalabilità automatica in risposta alla domanda, ridimensionando dinamicamente le risorse in base al carico di lavoro;
- *Monitoraggio e diagnostica*: fornisce strumenti di monitoraggio e diagnostica integrati per tracciare le esecuzioni delle funzioni e analizzare le prestazioni;
- *Sicurezza integrata*: implementa funzionalità di sicurezza, come autorizzazioni basate su ruoli e integrazione con Azure Active Directory.

Come spiegato nel capitolo precedente, questo servizio è utilizzato per implementare il microservizio Statistics Aggregator, il cui compito è aggregare dati a cadenza giornaliera.

6.2.6 Azure Kubernetes Service

Azure Kubernetes Service (*AKS*) è un servizio completamente gestito e offerto da Microsoft Azure che semplifica la distribuzione, la gestione e la scalabilità di applicazioni containerizzate utilizzando Kubernetes. AKS offre un ambiente affidabile e scalabile per ospitare i container. Alcune caratteristiche principali di Azure Kubernetes Service includono:

- *Kubernetes gestito*: AKS fornisce un'implementazione gestita di Kubernetes, semplificando la configurazione e la manutenzione del cluster;
- *Scalabilità automatica*: AKS offre la scalabilità automatica dei nodi del cluster in risposta al carico di lavoro, garantendo le risorse necessarie durante i picchi di utilizzo;

- *Integrazione con Azure Monitor*: si integra nativamente con Azure Monitor per il monitoraggio delle prestazioni, la registrazione degli eventi e la gestione dei log;
- *Integrazione con Azure Active Directory*: supporta l'integrazione con Azure Active Directory per la gestione delle identità e l'autenticazione degli utenti;
- *Sicurezza avanzata*: implementa funzionalità avanzate di sicurezza, come crittografia dei dati, autorizzazioni basate su ruoli e integrazione con Azure Policy;
- *Aggiornamenti semplificati*: offre un processo semplificato per gli aggiornamenti del cluster Kubernetes, garantendo che le applicazioni siano in esecuzione sulla versione più recente;
- *Integrazione con Servizi Azure*: si integra con altri servizi Azure, facilitando lo sviluppo di applicazioni complete basate su cloud.

AKS viene impiegato all'interno del progetto per orchestrare i container Docker associati a ciascun microservizio. Questa configurazione consente di realizzare un'applicazione con elevata disponibilità e scalabilità, facilitando la gestione dinamica dei microservizi in un ambiente containerizzato.

6.2.7 Azure API Management

Azure API Management è un servizio di gestione delle API offerto da Microsoft Azure, progettato per semplificare la creazione, la pubblicazione, la gestione e l'analisi delle API. Fornisce uno strumento centralizzato per controllare l'accesso alle proprie API, monitorarne le prestazioni e garantire la sicurezza e la scalabilità. Alcune caratteristiche principali di Azure API Management includono:

- *Gateway API globale*: Azure API Management agisce come un gateway API globale, consentendo l'esposizione delle API in tutto il mondo con bassa latenza e applicando misure di protezione per l'architettura backend;
- *Portale di gestione*: offre un portale web intuitivo per la gestione delle API, consentendo la configurazione, il monitoraggio e la documentazione;
- *Controllo dell'accesso*: implementa controlli avanzati di accesso alle API, inclusi token JWT, chiavi API e autenticazione basata su certificato;
- *Monitoraggio delle prestazioni*: fornisce strumenti dettagliati di monitoraggio delle prestazioni delle API, consentendo di identificare problemi e ottimizzare le risorse;
- *Policy API*: consente la definizione di policy API per gestire aspetti come la limitazione del traffico, la cache e la trasformazione dei messaggi;
- *Versionamento e revisione*: supporta il versionamento delle API e la revisione delle definizioni, garantendo la compatibilità e la gestione delle evoluzioni;

- *Sicurezza avanzata*: implementa misure di sicurezza avanzate, compresa la protezione da attacchi DDoS e la crittografia dei dati in transito.

Nel contesto del progetto, a causa di restrizioni di tempo, il servizio Azure API Management,originariamente previsto per funzionare come API gateway, non è stato effettivamente utilizzato. Al suo posto, il cluster Azure Kubernetes Service (AKS) è stato testato tramite NodePort, conoscendo gli indirizzi IP dei nodi interni al cluster Kubernetes.

6.2.8 Azure DevOps

Azure DevOps è una suite completa di strumenti offerta da Microsoft Azure per supportare il ciclo di vita dello sviluppo software, dalla pianificazione alla distribuzione. Azure DevOps fornisce un ambiente integrato per la collaborazione tra team di sviluppo e operazioni, facilitando la creazione, il test e la distribuzione di applicazioni in modo rapido e affidabile. Alcune caratteristiche principali di Azure DevOps includono:

- *Azure Boards*: un sistema di tracciamento del lavoro per la pianificazione e la gestione delle attività, consentendo la collaborazione tra membri del team;
- *Azure Repos*: un sistema di controllo di versione Git basato su cloud per la gestione del codice sorgente, facilitando la collaborazione e il monitoraggio delle modifiche;
- *Azure Pipelines*: un servizio di automazione del processo di integrazione continua e distribuzione continua (CI/CD), consentendo la distribuzione automatica delle applicazioni in diversi ambienti;
- *Azure Test Plans*: un sistema completo di pianificazione, monitoraggio e test delle applicazioni, con supporto per test manuali e automatizzati;
- *Azure Artifacts*: una repository di pacchetti per la gestione e la distribuzione di artefatti, come librerie e pacchetti NuGet, npm e Maven;
- *Collaborazione integrata*: offre strumenti integrati per la collaborazione tra team, inclusi commenti, approvazioni e notifiche;
- *Integrazione con Azure*: si integra nativamente con altri servizi Azure, facilitando la costruzione di applicazioni basate su cloud;
- *Monitoraggio e analisi*: fornisce strumenti di monitoraggio delle prestazioni e analisi dettagliate per migliorare continuamente i processi di sviluppo e distribuzione.

Azure DevOps è impiegato nel contesto del progetto per gestire la documentazione progettuale e per effettuare version control del software.

Capitolo 7

Test e Validation

Il processo di test e validazione riveste un ruolo fondamentale nello sviluppo software moderno. Attraverso l'implementazione di una robusta strategia di testing, gli sviluppatori possono garantire la qualità, l'affidabilità e la stabilità del software prodotto.

La prima sezione, intitolata *Tipologie di test*, si concentrerà sull'analisi di differenti approcci di testing, tra cui unit test, integration test, end-to-end test, e test di accettazione utente. Ciascun tipo di test assume un ruolo cruciale nel garantire la correttezza complessiva del sistema software.

La sezione successiva, *Test-Driven Development*, esaminerà l'omonima pratica di sviluppo software che pone l'accento sulla scrittura dei test prima dell'implementazione del codice. Attraverso tale approccio, gli sviluppatori possono creare software modulare e testabile, migliorando la manutenibilità e riducendo il rischio di errori.

7.1 Tipologie di test

Esistono varie tipologie di test utilizzate nel ciclo di sviluppo del software, ciascuna mirata a verificare specifici aspetti della funzionalità, della qualità e della robustezza del sistema. Di seguito verranno esaminate le principali categorie di test: i test di unità, di integrazione, end-to-end e di accettazione utente.

7.1.1 Unit Test

Un *unit test* è una tipologia di test che esamina le più piccole unità software testabili all'interno di un'applicazione, al fine di verificarne il comportamento atteso. La dimensione di tali componenti software non è definita in modo preciso, ma di solito coinvolge componenti software quali funzioni, metodi e classi.

Oltre a verificare il corretto funzionamento del codice, uno unit test può assistere il programmatore nell'identificare quando è opportuno suddividere un modulo in moduli

indipendenti. Questa necessità diventa evidente soprattutto quando la scrittura del test diventa complessa, indicando che la suddivisione modulare potrebbe rendere la struttura del codice migliore e preservare una chiara divisione delle responsabilità.

In aggiunta, questa categoria di test risulta molto utile anche come strumento di design se utilizzato in concomitanza con l'approccio Test Driven Development (TTD).

Nella pratica, l'obiettivo del programmatore è quello di testare le unità software isolandole il più possibile da dipendenze esterne. Considerando come esempio una funzione, può accadere che essa contenga chiamate ad altre funzioni, database, moduli o addirittura servizi esterni, chiamate più in generale dipendenze. L'isolamento è conveniente per vari motivi:

- *Velocità di esecuzione*: gli unit test mirano a essere veloci ed efficienti. Isolare l'unità da dipendenze esterne significa che i test possono essere eseguiti rapidamente senza attendere risposte di risorse esterne come database, servizi web o altri moduli complessi;

- *Controllo dell'ambiente di test e facilità di debug*: consente di avere un maggiore controllo sull'ambiente di test, eliminando variazioni dovute a fattori esterni. Ciò rende i risultati dei test più prevedibili e facilita l'individuazione di problemi specifici nell'unità, agevolando la fase di debug;

- *Minimizzazione degli effetti collaterali*: isolare le unità da dipendenze esterne aiuta a minimizzare gli effetti collaterali durante i test. Non c'è il rischio che le modifiche alle dipendenze esterne influenzino in modo imprevisto il risultato dei test unitari;

- *Facilità di manutenzione*: modificare o aggiornare le dipendenze esterne diventa più semplice, poiché le modifiche avranno un impatto limitato sulle unità isolate. Ciò rende il sistema più flessibile e adattabile a cambiamenti nelle dipendenze esterne, senza dover modificare in modo massiccio i test unitari;

- *Riproducibilità dei test*: l'isolamento garantisce che i test siano riproducibili in qualsiasi ambiente.

Per agevolare il lavoro del programmatore nel processo di isolamento delle unità testabili si ricorre all'utilizzo dei *test doubles*. Questi strumenti costituiscono un approccio efficace per gestire le dipendenze esterne durante il testing. Generalmente, sono oggetti utilizzati soprattutto negli unit test ma può accadere siano necessari anche durante gli integration test.

Esistono cinque tipi di test doubles, che sono:

- *Dummy*: è un oggetto che viene passato ad un metodo durante il test ma che non viene mai utilizzato. Il suo unico scopo è quello di soddisfare la firma del metodo invocato;

- *Fake*: si tratta di un oggetto che effettivamente ha un implementazione funzionante ma semplificata rispetto a quella usata in produzione. Un esempio potrebbe essere quello di utilizzare un database in memoria invece del database effettivo;
- *Stubs*: uno stub è un oggetto che fornisce risposte predefinite alle chiamate dei metodi durante i test. È configurato per restituire valori fissi, in modo da simulare il comportamento di un'interfaccia reale;
- *Spies*: si tratta di oggetti che registrano informazioni sulle chiamate dei metodi effettuate ad essi durante i test. Sono in grado di tracciare quanti metodi vengono chiamati, con quali argomenti e con quale frequenza, permettendo quindi di verificare il comportamento dell'oggetto sotto test;
- *Mocks*: gli oggetti mock possono essere considerati una combinazione tra uno stub e uno spy. Possono essere configurati in anticipo con aspettative specifiche sulle chiamate dei metodi, oltre a fornire risposte predefinite durante i test. Questo significa che, come uno stub, possono simulare il comportamento di una dipendenza reale, ma anche, come uno spy, possono tenere traccia delle chiamate dei metodi effettuate ad essi.

7.1.2 Integration Test

Gli unit test, come illustrato nel paragrafo precedente, sono progettati per verificare la corretta esecuzione di componenti software di piccole dimensioni, mantenute isolate dal resto del sistema. Ovviamente, questa tipologia di test da sola, non copre il comportamento complessivo dell'applicazione.

Gli *integration test*, invece, si estendono su una porzione più ampia del software, concentrandosi sulle interazioni tra diverse componenti e moduli. Il loro obiettivo principale è individuare potenziali errori nelle interfacce e garantire che la comunicazione tra le varie parti del sistema avvenga correttamente.

È essenziale sottolineare che un integration test non è finalizzato a verificare il corretto funzionamento di un modulo esterno, bensì a confermare che la comunicazione tra le componenti software avvenga come previsto. Pertanto, è necessario scrivere test che coprano scenari tipici di successo e di errore durante queste interazioni.

Gli integration test possono essere implementati a diversi livelli di granularità, a seconda delle esigenze specifiche del progetto.

Uno scenario di granularità grossolana è quella relativa al testing degli endpoint. Questo approccio si concentra principalmente sulla comunicazione tra il client e il backend attraverso gli endpoint dell'API RESTful, ad esempio. Si testano gli endpoint principali e i casi d'uso più comuni. Lo scopo è quello di verificare che le richieste HTTP inviate al server vengano elaborate correttamente, che le risposte siano conformi alle specifiche dell'API e che il flusso di dati tra client e server sia gestito in modo adeguato.

Questo approccio, da solo, potrebbe non rilevare problemi più profondi o collaborazioni tra i vari livelli del backend. Essenzialmente, può essere adatto per una

rapida convalida delle funzionalità principali dell'API. Dunque, può risultare conveniente estendere il testing per includere la comunicazione più ampia tra tutti i livelli del backend, inclusi eventuali collaborazioni con servizi esterni, operazioni di database e altri componenti interni. Questo approccio offre una visione più completa della robustezza e dell'integrità dell'intero sistema.

La scelta tra questi approcci dipende dalle esigenze specifiche del progetto. In molti casi, una combinazione di entrambi può essere appropriata. Inizialmente, è comune iniziare con il testing degli endpoint, per assicurarsi che la comunicazione fondamentale funzioni correttamente. Successivamente, potrebbe essere necessario espandere il testing per coprire scenari più complessi e collaborazioni interne al backend.

7.1.3 End-to-End Test

La fase successiva nel processo di testing software consiste nei test di sistema, conosciuti in inglese come *end-to-end test*. Questi test valutano l'intero sistema in relazione ai requisiti specificati, assicurandosi che tutte le funzionalità e le caratteristiche previste siano implementate correttamente. Tale tipologia di test esamina l'intero flusso attraverso il sistema, partendo dall'interfaccia utente e proseguendo fino ai componenti del backend, nonché alle interazioni con sistemi esterni o altri servizi.

Questa tipologia di test può essere condotta dagli sviluppatori stessi o, in alcuni casi, da un team specializzato nel controllo di qualità, noto come team di QA (Quality Assurance).

Gli scenari di test vengono progettati per essere realistici, al fine di garantire che il sistema si comporti come previsto nelle situazioni quotidiane di utilizzo. Ad esempio, un test end-to-end potrebbe simulare l'interazione di un utente con l'applicazione, dal momento in cui l'utente accede al sistema fino al completamento di un'azione desiderata, come la creazione di un ordine o la compilazione di un modulo.

È importante sottolineare che l'end-to-end testing si focalizza sull'assicurare il corretto funzionamento complessivo di un sistema software e non considera l'esperienza o la soddisfazione dell'utente durante l'utilizzo dell'applicazione, aspetto invece valutato tramite i test di accettazione utente.

7.1.4 Test di Accettazione Utente

I *test di accettazione utente*, noti come *UAT* (*User Acceptance Testing*) in inglese, hanno l'obiettivo di verificare se il software soddisfa in modo effettivo le esigenze e le aspettative degli utenti finali, valutando se l'applicazione è pronta per il rilascio.

Gli utenti finali, che possono essere clienti, dipendenti o altri stakeholder chiave, sono coinvolti attivamente nel processo di testing. Questi rappresentano i veri utilizzatori del software e sono in grado di fornire un feedback prezioso sulla sua usabilità e funzionalità.

In questa fase, vengono esaminati i requisiti e le specifiche del software per verificare che tutti gli elementi richiesti siano stati implementati correttamente. Gli utenti finali,

quindi, eseguono una serie di test che possono includere scenari di utilizzo realistici, operazioni critiche e flussi di lavoro tipici. L'obiettivo è coprire un'ampia gamma di casi d'uso per assicurarsi che il software sia completo e soddisfi tutti i requisiti utente stabiliti.

Gli utenti finali forniscono feedback dettagliato sulle loro esperienze durante il testing, segnalando eventuali problemi, errori o aree di miglioramento. Questo feedback è prezioso per l'identificazione e la risoluzione dei problemi prima del rilascio.

7.2 Test-Driven Development

Il *Test Driven Development* (*TDD*) è una metodologia di sviluppo software nata negli anni novanta, che si basa su un ciclo iterativo di sviluppo, in cui la scrittura dei test precede l'implementazione del codice.

A partire da una lista di casi di test scritta in principio, il processo può essere descritto in tre fasi:

- *Red*: in questa fase, si seleziona un caso di test e si scrive il test corrispondente per la funzionalità scelta. Poiché il codice da testare non è ancora stato implementato, il test inizialmente fallisce, da cui il nome di questa fase;
- *Green*: si procede con l'implementazione della funzionalità. L'obiettivo principale è far sì che il test passi con successo. Durante questa fase, l'attenzione è concentrata sull'implementazione della soluzione per soddisfare i requisiti del test, evitando di impiegare risorse per l'ottimizzazione del codice;
- *Refactor*: quest'ultima fase implica modifiche al codice precedentemente implementato per migliorare la sua struttura, leggibilità e prestazioni, senza alterare il comportamento funzionale. Dopo ogni operazione di refactoring, si eseguono nuovamente i test per garantire che continuino a passare.

Questi tre step vengono ripetuti per ciascun caso di test presente nella lista. È importante considerare che potrebbe essere necessario aggiungere nuove funzionalità durante lo sviluppo del progetto, il che potrebbe portare all'espansione dell'elenco dei casi di test.

L'adozione del Test-Driven Development si dimostra vantaggiosa poiché facilita agli sviluppatori la creazione di software di alta qualità. Ciò si manifesta nel rilevamento tempestivo degli errori non appena una funzionalità viene implementata. Inoltre, il TDD promuove una migliore progettazione del codice, poiché richiede la scrittura dei test prima dell'implementazione effettiva. Questo approccio favorisce una maggiore modularità, coesione e una ridotta dipendenza tra i vari componenti del software.

D'altra parte, questa metodologia può risultare complessa da apprendere, poiché comporta un cambiamento radicale nel ciclo di sviluppo del software. Inoltre, potrebbe verificarsi una diminuzione della produttività, poiché è richiesta la scrittura dei casi di test per tutte le parti del codice.

Capitolo 8

Conclusioni

Il capitolo finale di questa tesi è dedicato alla riflessione sui risultati ottenuti e alle potenziali direzioni future per lo sviluppo e il miglioramento dell'applicazione.

Nella prima sezione, intitolata *Considerazioni del progetto e dei risultati ottenuti*, verranno esaminate le principali considerazioni emerse durante lo sviluppo del progetto e l'analisi dei risultati ottenuti.

Nella sezione *Future possibilità di sviluppo e miglioramento dell'applicazione* si esploreranno le potenziali opportunità e le sfide per il futuro sviluppo e miglioramento dell'applicazione.

8.1 Considerazioni del progetto e dei risultati ottenuti

Dopo aver completato lo sviluppo del software, emergono considerazioni fondamentali che delineano l'impatto e il valore del lavoro svolto.

Il software ha dimostrato di offrire vantaggi significativi in termini di funzionalità e velocità per l'utente finale. Rivolto ai business manager, l'applicazione è in grado di accelerare notevolmente il processo di prospecting, consentendo agli utenti di effettuare ricerche in modo efficiente e di sfruttare funzionalità avanzate come il salvataggio delle ricerche e l'invio automatizzato di email.

In caso contrario, un business manager dovrebbe ricercare i prospect attraverso piattaforme come LinkedIn, la più diffusa, e successivamente contattare ogni utente individualmente, annotando le proprie considerazioni o feedback tramite fogli Excel o altri documenti testuali. Il software, invece, unisce tutte queste funzionalità all'interno di un unico sistema, migliorando l'efficienza operativa.

L'applicazione presenta alcune implementazioni che sono rimaste incomplete a causa di restrizioni di tempo, le quali verranno esaminate nel dettaglio nella sezione successiva.

Dal punto di vista formativo personale, l'esperienza ha favorito diverse nozioni fondamentali, quali la progettazione di un'architettura a microservizi ben strutturata e la familiarità con l'ambiente cloud di Microsoft Azure. Inoltre, l'utilizzo di librerie e framework mai utilizzati in passato è stato un fattore determinante per la crescita personale.

8.2 Future possibilità di sviluppo e miglioramento dell'applicazione

Benché il backend sviluppato funzioni correttamente e come previsto, vi sono alcuni aspetti che potrebbero essere migliorati:

- *Memorizzazione di dati sensibili*: la funzionalità riguardante il salvataggio di ricerche effettuate introduce un problema non di poco conto. Infatti, questa feature prevede il salvataggio di un'insieme di contatti (dipendenti dell'azienda prospect), col fine di poterli recuperare in futuro e continuare a contattarli tramite email. Tuttavia, in conformità al GDPR, le informazioni sui contatti possono includere dati sensibili che richiedono l'autorizzazione diretta degli interessati per la memorizzazione. Per affrontare questa problematica, sono state considerate due possibili soluzioni. La prima opzione prevede il salvataggio dei parametri di ricerca anziché dei risultati finali, ovvero l'elenco dei contatti. In questo modo, il software esegue nuovamente la ricerca ogni volta che l'utente cerca di recuperare una ricerca salvata, utilizzando i parametri precedentemente memorizzati. Questo approccio evita il salvataggio di dati sensibili relativi ai contatti. La seconda opzione consiste nell'integrare un sistema che invia una email con un modulo di autorizzazione agli interessati quando l'utente cerca di salvare i loro dati. Tuttavia, questa soluzione è più complessa, soprattutto considerando la potenziale mole di contatti presenti in una ricerca e il numero di email da inviare. Inoltre, l'implementazione della seconda soluzione comporterebbe la necessità di coinvolgere figure aziendali competenti in materia di protezione dei dati e conformità al GDPR, aggiungendo ulteriore complessità alla gestione delle autorizzazioni e alla protezione dei dati sensibili;
- *Integration test*: un altro aspetto che può essere migliorato riguarda gli integration test automatizzati. Attualmente, questi test sono stati implementati con una granularità grossolana, focalizzandosi principalmente a livello di endpoint. Una possibile evoluzione consiste nell'introdurre ulteriori test di integrazione a un livello più dettagliato, concentrandosi sulla correttezza della comunicazione tra i moduli interni del backend. Tuttavia, per attuare questo miglioramento, è necessario risolvere preventivamente un problema di incompatibilità all'interno della libreria utilizzata per i test. Attualmente, tutti i test di integrazione falliscono se l'ambiente è configurato per eseguire anche gli unit test;
- *Azure API Management*: come menzionato precedentemente nel Capitolo 6, a causa di restrizioni di tempo, non è stato possibile utilizzare il servizio Azure

API Management, che era inizialmente previsto. L'introduzione di questo servizio avrebbe reso il cluster Kubernetes più sicuro grazie all'utilizzo di un API gateway;

- *Utenti*: inevitabilmente, è necessario integrare la gestione di diversi utenti all'interno del software. Questo potrebbe essere realizzato internamente attraverso lo sviluppo di un nuovo microservizio nel backend, oppure, come alternativa, attraverso l'utilizzo del servizio Azure Active Directory, ora noto come Microsoft ID Entra;
- *Frontend*: naturalmente un'area di miglioramento è quella relativa al design e sviluppo del frontend. Attualmente, non esiste un frontend che fornisca un'interfaccia grafica comoda e funzionale all'utente finale.

Bibliografia

[1] A. Kharenko, *Monolithic vs. Microservices Architecture*, Medium, 09/10/2015, https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

[2] G. Cocca, *The Software Architecture Handbook*, Medium, 26/07/2022, https://www.freecodecamp.org/news/an-introduction-to-software-architecture-patterns/

[3] M. Fowler, *Software Architecture Guide*, martinfowler, 01/08/2019, https://martinfowler.com/architecture/

[4] M. Fowler, *Who Needs an Architect?*, martinfowler, 09/10/2015, https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf

[5] G. Blinowski, A. Ojdowska, A. Przybyłek, *Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation*, IEEE Xplore, 18/02/2022, https://ieeexplore.ieee.org/document/9717259

[6] B. Reselman, *5 design principles for microservices*, Red Hat Developer, 11/01/2022, https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices

[7] J. Kanjilal, *Top 10 Microservices Design Principles*, developer.com, 14/09/2022, https://www.developer.com/design/microservices-design-principles/

[8] Y. Francino, *user story*, TechTarget, 2020, https://www.techtarget.com/searchsoftwarequality/definition/user-story#:~:text=A%20user%20story

[9] Segue Technologies, *What Characteristics Make Good Agile Acceptance Criteria?*, Segue Technologies, 03/09/2015, https://www.seguetech.com/what-characteristics-make-good-agile-acceptance-criteria/

[10] M. Pfeil, *What is data persistence & why does it matter?*, datastax, 22/10/2010, https://www.datastax.com/blog/what-persistence-and-why-does-it-matter

[11] C. Brook, *What is Data Integrity? Definition, Types & Tips*, Digital Guardian, 08/05/2023, https://www.digitalguardian.com/blog/what-data-integrity-data-protection-101

[12] D. Kizilpinar, *Data Consistency in Microservices Architecture*, Medium, 27/04/2021, https://dilfuruz.medium.com/data-consistency-in-microservices-architecture-5c67e0f65256

[13] M. Ozkaya, *Microservices Data Management*, Medium, 08/09/2021, https://medium.com/design-microservices-architecture-with-patterns/microservices-data-management-3235893b7c29

[14] S. Newman, *Building Microservices, 2nd Edition, chapter 4*, oreilly, 08/2021, https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/ch04.html
[15] , P. Mell, T. Grance, *The NIST Definition of Cloud Computing*, NIST, 07/2011, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf
[16] D. Wethington, *The History of Cloud Computing: The Theory*, DomaOnline, 10/08/2021, https://www.domaonline.com/2021/08/10/history-of-cloud-computing-1/
[17] ServiceNow, *What is a cloud service provider?*, ServiceNow, 11/12/2023, https://www.servicenow.com/products/it-operations-management/what-is-cloud-provider.html
[18] C. Slingerland, *11 Top Cloud Service Providers Globally In 2024*, CloudZero, 15/12/2023, https://www.cloudzero.com/blog/cloud-service-providers/
[19] A. Stashko, *Top Cloud Service Providers: A Quick Comparison*, Avenga, 05/05/2021, https://www.avenga.com/magazine/top-cloud-service-providers/
[20] AppMaster, *What is Microsoft Azure: How Does It Work and Services*, AppMaster, 27/01/2023, https://appmaster.io/blog/what-is-microsoft-azure
[21] IBM, *Cos'è lo sviluppo del software?*, IBM, 20/07/2022, https://www.ibm.com/it-it/topics/software-development
[22] D. Kaushal, *Why Business and Functional Requirements are Vital for a Project's Success*, net solutions, 30/12/2022, https://www.netsolutions.com/insights/business-and-functional-requirements-what-is-the-difference-and-why-should-you-care/
[23] J. Steele, *User Stories vs. Requirements: Bridging the Gap Between Business and Technical Stakeholders*, LinkedIn, 20/05/2023, https://www.linkedin.com/pulse/user-stories-vs-requirements-bridging-gap-between-jay-steele-/
[24] Nuclino, *A Guide to Functional Requirements (with Examples)*, Nuclino, 26/11/2020, https://www.nuclino.com/articles/functional-requirements
[25] J. Romanchuk, *The Business Requirement Document: What It Is and How to Write It [+5 Templates]*, HubSpot, 28/11/2022, https://blog.hubspot.com/marketing/business-requirement-document
[26] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, *Manifesto for Agile Software Development*, agilemanifesto, 11/02/2001, https://agilemanifesto.org/
[27] K. Schwaber, J. Sutherland, *The 2020 Scrum Guide*, SCRUM GUIDES, 11/2020, https://scrumguides.org/scrum-guide.html
[28] M. Fowler, *Testing Strategies in a Microservice Architecture*, martinfowler, 18/11/2014, https://martinfowler.com/articles/microservice-testing/
[29] M. Fowler, *Test Double*, martinfowler, 17/01/2006, https://martinfowler.com/bliki/TestDouble.html
[30] A. Basson, *Choosing the Right Test Double*, Tanzu, 26/06/2022, https://tanzu.vmware.com/developer/guides/test-doubles