UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
ED ELETTRICA E MATEMATICA APPLICATA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA (L-8)

ELABORATO FINALE

Sviluppo su container di un applicativo web di reportistica aziendale automatica in ambiente ASP.NET

Relatore:
Carlo Mazzocca

Candidato:
Francesco Lanzara
Matr. 0612707459

Tirocinio svolto presso:
Metoda Finance S.R.L.
Tutor Aziendale: *Marco Leone*

Anno Accademico 2024 – 2025

Indice

Abstract

Nel contesto della digitalizzaziones dei processi bancari e finanziari, la reportistica automatizzata si configura come un elemento cardine per la governance del rischio, la compliance normativa e il supporto alle decisioni strategiche. Il presente elaborato illustra la progettazione e l'implementazione di una piattaforma software per la generazione di report finanziari, sviluppata nell'ambito del tirocinio curriculare svolto dallo scrivente presso *Metoda Finance S.R.L.*, realtà di riferimento nel settore delle soluzioni informatiche per intermediari finanziari.

L'architettura proposta adotta un **paradigma a microservizi**, in contrapposizione al tradizionale approccio monolitico, al fine di garantire scalabilità orizzontale, resilienza ai guasti e manutenibilità modulare. Tale approccio consente di suddividere il sistema in componenti autonomi che comunicano tramite protocolli stateless ed elaborano informazioni su database centralizzati: ciascun componente è responsabile di una specifica capacità funzionale, secondo i principi di responsabilità unica, disaccoppiamento e governance decentralizzata. Il sistema è orchestrato mediante *Docker Compose*, favorendo la portabilità e l'isolamento degli ambienti, e integra un **API Gateway** per la gestione centralizzata delle richieste, il bilanciamento del carico e l'applicazione di policy di sicurezza. Il sistema si presenta come **interfaccia web-based** per l'accesso a un motore di generazione di report preesistente, fruibile sia tramite **RESTful API** con ASP.NET Core WebAPI, utile per l'integrazione con strumenti di Business Intelligence, sia mediante un front-end interattivo con rendering dinamico usando ASP.NET Core MVC. Inoltre, la piattaforma implementa meccanismi di comunicazione real-time tramite SignalR, consentendo aggiornamenti asincroni dell'interfaccia utente. L'applicativo sviluppato si orienta nel mercato delle soluzioni software per enti nel contesto dei financial services costituendo uno strumento indispensabile per creare valore attraverso la correlazione ed esplorazione di dati eterogenei, rafforzare la trasparenza e la compliance, favorire la rapidità d'intervento grazie a meccanismi di allerta in real-time e supportare l'evoluzione futura del sistema di gestione.

Capitolo 1

Introduzione

1.1 La reportistica aziendale automatizzata per favorire un modello decisionale data-driven

Nel contesto dei *financial services*, la capacità di organizzare i dati e tramutarli in insight strategici è centrale per ottimizzare i processi di enti ed imprese e garantirne il successo. La reportistica aziendale, tutt'altro che mera espressione di adempimento burocratico, costituisce una colonna portante dei processi decisionali, fungendo da linfa vitale per la business intelligence, il controllo di gestione e la digitalizzazione integrata dei flussi operativi. Attraverso di essa, le organizzazioni non solo attestano la propria performance, ma costruiscono le fondamenta per una governance informata, agile e proattiva. Un sistema di reportistica configurabile e avanzato consente alle realtà orientate ai dati di raccogliere e visualizzare indicatori chiave di performance, dati economico-finanziari, operativi e qualitativi, rendere trasparenti le metriche utili al confronto tra reparti o rispetto agli obiettivi strategici, e condividere informazioni in tempi rapidi, anche in mobilità, tramite interfacce dedicate e report compilati automaticamente nei formati più diffusi.

Un prodotto di questa natura, soprattutto se orientato alla facilità di configurazione e personalizzazione del servizio[1], consente un approccio computer-aided e data-driven al decision making, per una vasta gamma di utenti anche su piani gerarchici distinti: dagli operatori che necessitano di generare, scaricare e archiviare report sulle proprie attività, ricevendo avvisi tempestivi sull'esito delle elaborazioni; ai dirigenti e manager che impiegano dati aggregati e dashboard evolute per il controllo di gestione, l'analisi degli scostamenti e la pianificazione strategica; fino agli enti e alle aziende che intendono evolvere verso una gestione dei dati moderna, flessibile e basata sui dati, riducendo i tempi di produzione dei report e migliorando contestualmente la qualità delle decisioni.

[1] A tal fine, risulta imprescindibile segmentare le funzionalità per garantirne la modularità seguendo le esigenze di ciascun profilo utente, da quello operativo a quello dirigenziale. La scelta logica per ottenere questo risultato consiste nella progettazione di un'architettura basata su microservizi, di cui si discuterà in seguito.

1.2 Il caso di studio

Nel panorama appena descritto si concretizza l'obiettivo del tirocinio curriculare oggetto di questa trattazione: l'ideazione e la prototipizzazione di un'architettura software modulare e service-oriented, che esponga un servizio preesistente di generazione automatica di report rendendolo *sicuro, scalabile, personalizzabile e facilmente integrabile* con tool di business intelligence o altri software gestionali preesistenti. Come verrà approfondito nel capitolo 4, il progetto prevede l'implementazione di un'architettura a microservizi, isolati mediante l'impiego di container *Docker* che garantiscono sia la scalabilità orizzontale sia la migrazione tra ambienti eterogenei. Le funzionalità essenziali da implementare si concretizzano nell'utilizzo del motore di reportistica per realizzare un'interfaccia web, che sia accessibile sia mediante RESTful API, affinché il servizio sia agganciabile da altri applicativi per il settore dei financial services (e.g. tool di business intelligence quali PowerBI o Tableau), sia avvalendosi di un'interfaccia utente che renda immediati e intuitivi la generazione e la condivisione dei report, esportabili in formati diffusi quali Excel (.xlsm) e PDF. Caratteristiche aggiuntive, come l'orchestrazione con ***Docker Compose***, l'introduzione **update asincroni** dell'interfaccia, l'implementazione di un sistema di **autenticazione** robusto e centralizzato, e l'utilizzo di un **gateway API** come intermediario tra client e microservizi completano il quadro di una soluzione moderna, scalabile e sicura.

1.3 Tecnologie abilitanti

Il framework adottato per la realizzazione dei servizi è *Microsoft ASP.NET Core*, che consente di sviluppare applicazioni web e API in modo rapido ed efficiente, sfruttando le potenzialità del linguaggio C# e dell'ecosistema *.NET*. Nello specifico, per la configurazione del container incaricato di esporre l'API RESTful si usufruisce del modello **ASP.NET Core Web API**, che rende agile e intuitiva la configurazione degli endpoint e la manutenzione della logica dell'API (documentazione dell'API esposta mediante il servizio **Swagger**); l'applicativo web con interfaccia è stato realizzato secondo il modello **ASP.NET Core MVC**, utile per realizzare applicativi tradizionali[2], separando distintamente la logica di presentazione dalla logica di business, e facilitando lo sviluppo e la manutenzione dell'interfaccia utente. All'interfaccia utente sono poi aggiunte feature dinamiche mediante la tecnologia **SignlR**, che consente l'invio di notifiche asincrone mantenendo comunicazioni in bilaterali real-time tra client e server.

Per quanto concerne la struttura organizzativa dell'applicativo, annoveriamo altre feature che forniscono al progetto un'architettura del tutto simile un approccio a microservizi: la soluzione è basata su container indipendenti, orchestrati mediante **Docker Compose**; un container dedicato centralizza la gestione dell'autenticazione usufruendo di **Duende Identity Server**, **EntityFramework Core** e **ASP.NET Core Identity e Authentication**. L'adozione di un gateway API per un reverse proxying potente e facilmente configurabile con **YARP**, per la gestione delle richieste e

l'applicazione di policy di sicurezza, garantendo non solo modularità e manutenibilità evolute nel tempo, ma una navigazione tra i vari servizi fluida, eliminando la necessità di autenticazioni multiple nell'accedere a servizi distinti del sistema, la cui separazione interna non dovrebbe influire sull'esperienza dell'utente.

In definitiva, la soluzione proposta non solo è tecnicamente adeguata ai requisiti richiesti da organizzazioni data-intensive, ma si collocherebbe con piena efficacia in scenari reali, promuovendo usabilità, integrabilità ed evolvibilità. Essa incarna il passaggio dalla reportistica statica e reattiva a un ecosistema dinamico, interconnesso e capace di generare valore continuo, trasformando i dati nel più prezioso alleato per il governo dell'impresa.

1.4 Struttura della tesi

Il seguito della trattazione si articolerà nelle sezioni seguenti:

1. **Contesto applicativo**, con cenni al settore dei *financial services* e alle esigenze di digitalizzazione e automazione dei processi decisionali che ne caratterizzano l'evoluzione.

2. **Accenni alle architetture distribuite orientate ai microservizi** e descrizione di concetti e principi propri degli applicativi **software web-based** impiegati contestualmente alla soluzione prodotta.

3. **Panoramica sugli strumenti tecnologici** impiegati, con particolare riferimento alla suite *ASP.NET Core*, framework centrale per la realizzazione della soluzione in esame.

4. **Analisi del caso di studio**, con illustrazione della struttura progettuale, delle scelte implementative e delle prospettive di evoluzione verso scenari di deployment distribuito e multicanale.

5. **Conclusione**

[2] Per "*tradizionali*" si intende applicazioni multi-pagina (*MPA*) con rendering server-side, una delle filosofie originarie per la realizzazione di applicativi web. Nei capitoli successivi si argomenterà la scelta dell'utilizzo di tale modello rispetto a soluzioni meno mature, e.g. lo sviluppo di applicazioni singola-pagina (*SPA*) con tecnologia *Blazor*, sempre della suite ASP.NET.

Capitolo 2

Contesto applicativo: il mondo della reportistica automatica

2.1 Introduzione

Nel settore finanziario moderno i dati sono un vero asset strategico: la digitalizzazione spinge istituzioni e intermediari a interconnettere sistemi e ad automatizzare i processi. In questo nuovo ecosistema digitale è necessario abbattere i silos informativi e ripensare l'infrastruttura IT secondo paradigmi cloud-native. La trasformazione digitale nei financial services richiede infrastrutture modernizzate e automazione dei processi interni. Ad esempio, grazie all'adozione di tecnologie come RPA e machine learning, l'efficienza operativa delle aziende può aumentare di oltre il 40%, riducendo drasticamente errori e tempi di esecuzione. Al contempo le autorità di vigilanza italiane (Banca d'Italia, CONSOB, IVASS) impongono un livello crescente di adempimenti normativi: gli intermediari devono produrre report periodici accurati, tempestivi e conformi ai formati ufficiali per il monitoraggio prudenziale. Questa molteplicità di requisiti rende strategica l'automazione della reportistica.

Metoda Finance S.R.L., azienda ospitante del tirocinio, sviluppa soluzioni software per la gestione documentale e la compliance normativa. Ha creato un motore interno di reportistica in grado di generare automaticamente file Excel e PDF, con configurazioni delle fonti dati semplificate a livello di funzione. Tuttavia il sistema originario non disponeva di un'interfaccia esterna fruibile: un potenziale utente dovrebbe interagire direttamente con il codice del motore di generazione, risultando inutilizzabile per l'operatore casuale privo di competenze informatiche specifiche né una conoscenza almeno superficiale del sistema. Da qui l'esigenza di avvolgere il servizio di reportistica in un'architettura facilmente fruibile, modulare e scalabile. L'obiettivo è esporre le funzionalità tramite API RESTful, in modo che possano essere richiamate da interfacce web o strumenti BI esterni, e mediante una maschera human-friendly che renda l'accesso facile all'operatore senza necessitare competenze specifiche. L'intero sistema dovrà essere gestito come insieme di microservizi containerizzati, con un punto d'accesso univoco che fornisca un livello di sicurezza ed autenticazione per l'intera

struttura. In questo modo si punta a rendere il reporting automatizzato facilmente configurabile, altamente disponibile e adattabile a future evoluzioni tecnologiche.

2.2 Il problema affrontato

La produzione manuale dei report finanziari presenta evidenti criticità. Richiede molte risorse di tempo e personale specializzato, espone a errori di trascrizione e incoerenze nei dati, e rende complessi aggiornamenti o riconfigurazioni rapide. Per quanto riguarda l'architettura software, un sistema monolitico ostacola l'estensione funzionale e la scalabilità: qualunque modifica (ad esempio per un nuovo tipo di report) può richiedere di ridistribuire l'intero applicativo, e i carichi elevati devono essere bilanciati sull'intera piattaforma. Un approccio più flessibile è dato dalle architetture a microservizi, che scompongono le funzionalità in servizi indipendenti. In queste architetture ogni microservizio svolge un compito specifico e comunica con gli altri attraverso API ben definite. Ciò garantisce maggiore manutenibilità e resilienza: per esempio, se un componente è sottoposto a un picco di traffico (ad esempio il servizio di calcolo dei dati), solo quel servizio verrà replicato, anziché scalare l'intera applicazione. I microservizi favoriscono inoltre un deployment incrementale: si possono aggiornare singole funzionalità senza interrompere il servizio complessivo.

Un altro aspetto critico è la coerenza dei dati in un sistema distribuito. Per gestire l'interazione tra microservizi in maniera affidabile spesso si ricorre a comunicazione asincrona tramite code o middleware di messaggistica (e.g. Kafka, RabbitMQ), che promuovono il disaccoppiamento tra componenti e permettono di assorbire picchi di carico senza perdita d'informazione, mentre le operazioni incentrate sulla manipolazione dei dati sono gestite mediante le cosiddette *saghe*, che garantiscono una sicurezza transazionale quando l'elaborazione dei dati attraversa più nodi di una rete di microservizi. Detto ciò, in un'architettura containerizzata e orchestrata (tipicamente Kubernetes), ogni microservizio tende a mantenere il proprio schema dati, riducendo al minimo le dipendenze incrociate e consentendo una scalabilità orizzontale elastica che prescinde dall'host fisico di deploy del servizio. La soluzione che si propone di realizzare è un prototipo su host singolo di un sistema di reportistica automatica orientato ai microservizi. Vincoli di tempo e risorse hanno imposto che l'effettiva simulazione del sistema fosse limitata al singolo nodo, rinunciando alla configurazione di un sistema di orchestrazione e all'utilizzo di broker di *message queueing* per una comunicazione multi-host efficace, ma l'architettura è progettata per essere facilmente personalizzabile ed estendibile a un cluster distribuito.

2.3 Confronto con lo stato dell'arte

Esistono diverse soluzioni di mercato per la reportistica, ma ognuna presenta limiti in relazione a flessibilità e scalabilità. Ad esempio:

- **ERP integrati** (come SAP o Oracle): offrono moduli dedicati al reporting,

ma in genere sono sistemi piuttosto rigidi. Spesso richiedono personalizzazioni complesse per adattarsi a contesti specifici di business, rendendo onerosa la configurazione dei report rispetto alle esigenze variabili dell'azienda.

- **Strumenti di Business Intelligence commerciali** (Power BI, Tableau, Qlik): eccellono nella visualizzazione dei dati e nel data modeling, ma necessitano di infrastrutture complesse, competenze specifiche di progettazione report e spesso licenze software costose. Possono inoltre richiedere tempi lunghi di set-up per integrarsi con sistemi preesistenti.

- **Framework open-source di reporting e BI** (come Metabase, ReportServer, JasperReports): garantiscono maggiore flessibilità di personalizzazione e costi inferiori, ma impongono all'utente finale di possedere competenze tecniche elevate per installazione e integrazione. L'integrazione con sistemi legacy può risultare complicata e il supporto tecnico limitato.

Le architetture basate su microservizi dimostrano invece come sia possibile realizzare sistemi distribuiti, scalabili e manutenibili. I grandi player del settore fintech adottano microservizi perché consentono di innovare rapidamente, isolare i malfunzionamenti e reagire agilmente ai picchi di traffico. La soluzione proposta non vuole essere da meno: ci si premura dunque di realizzare non una semplice web application che esponga in rete il servizio di reportistica automatizzata, ma un'architettura sicura e scalabile, che coniughi una user experience fluida e non influenzata dall'organizzazione sottostante, a un'organizzazione modulare che fornisca performance elevate e facilità di manutenzione ed estensione.

Nella soluzione proposta infatti il motore di reportistica è implementato con tecnologie moderne, pur fornendo un'interfaccia basata su tecnologie consolidate e integrabili anche con strumenti terzi più maturi: ASP.NET Core garantisce elevate prestazioni e sicurezza, l'adozione di una filosofia MPA e di un'API RESTful garantisce interoperabilità con browser e servizi meno recenti, mentre SignalR abilita funzionalità real-time con push verso i client senza polling/refresh. L'uso di container Docker fa sì che ogni microservizio sia leggero e indipendente: ogni servizio è isolato dal sistema ospite, e l'unica risorsa necessaria all'esecuzione è la presenza sul server di un host Docker. In questo modo, è possibile replicare, avviare o eliminare container in pochi secondi, consentendo una scalabilità orizzontale allineata alla domanda effettiva. Ad esempio, in caso di carichi di lavoro imprevisti il sistema può istanziare ulteriori container del servizio interessato in maniera trasparente senza impattare gli altri componenti.

Infine, viene adottato un API Gateway centralizzato, che funge da unico punto d'ingresso per il browser e per le richieste REST. Ciò permette di applicare in un solo punto regole di sicurezza, autenticazione (OpenID Connect per il browser, JWT per le API), e di aggregare e instradare le risposte in modo trasparente. In pratica, l'API Gateway astrae i dettagli interni dei microservizi, facilitando l'evoluzione indipendente di ogni componente.

Il prototipo descritto in questa tesi integra quindi tutte queste best practice: un'architettura a microservizi containerizzati pronti per un deploy in un contesto distribuito e orchestrato, un API Gateway per la sicurezza e il bilanciamento, ASP.NET Core e SignalR per il calcolo e il push dei dati, e uno strato di persistenza distribuita. In questo modo si ottiene un sistema di reportistica completamente automatizzato, in grado di generare documenti dinamicamente a partire da fonti dati configurabili e di esportarli in Excel o PDF senza intervento manuale. L'approccio risponde alle esigenze di efficienza, coerenza e adattabilità del contesto finanziario moderno, rispettando gli obblighi normativi e abilitando gli utenti a consultare i report da interfacce web o strumenti di BI esterni.

Capitolo 3

Architetture a microservizi e servizi web evoluti

3.1 I limiti delle architetture monolitiche

Le architetture a microservizi si propongono di estirpare alla radice problemi e limiti delle architetture monolitiche, che nel contesto dei sistemi distribuiti rappresentano la soluzione intuitiva e tradizionale. In questa sezione, daremo un breve sguardo alle difficoltà che emergono con l'approccio monolitico, per poi introdurre il concetto di microservizi e come questi affrontino tali sfide. Un applicativo si definisce *monolitico* quando:

- Un' unica codebase racchiude l'intera logica di business, che è al più suddivisa in moduli. Molti team di sviluppatori con competenze disparate devono collaborare su un unico progetto, integrare grandi quantità di modifiche dipendenti da più moduli e dispiegarle contemporaneamente sui server di produzione;
- Ogni modulo elabora l'interità della logica per il dominio a cui appartiene, e il suo *lifecycle*, ossia l'estensione nel tempo dei periodi di attività e stallo del servizio, è dipendente da ogni altro modulo: se un modulo va "down" (per manutenzione o guasto), l'intero applicativo deve smettere di funzionare;
- I dati di tutti i domini sono gestiti in un unico database centralizzato, accessibile da ogni modulo. Questo è spesso un voluminoso database relazionale che consente di utilizzare facilmente *join* per operazioni inter-modulari, ma può altrettanto facilmente diventare un collo di bottiglia per le performance e la scalabilità dell'applicativo.
- L'applicativo è distribuito come un unico eseguibile, quindi il carico di lavoro assegnabile ad ogni modulo è standardizzato per ogni singola istanza di esecuzione. Ipotizziamo un applicativo con moduli **A**, **B** e **C**, in cui le richieste al solo modulo **A** superano la sua capacità elaborativa. Per far fronte al problema, sarebbe necessario avviare una seconda istanza dell'applicativo, compresa di moduli **B**

e **C** (non saturi), causando una dispersione evidente di risorse computazionali e di memoria.

Per quanto possa sembrare più *semplice* e *intuitivo* iniziare un progetto come soluzione monolitica, i bassi livelli di scalabilità ed evolvibilità della soluzione rendono facile comprendere l'adozione progressivamente maggiore di un approccio basato su microservizi per la progettazione di software distribuiti agili.

3.2 I microservizi: ultima frontiera dei sistemi distribuiti

Il concetto di architettura a microservizi non nasce come una realtà isolata, ma si colloca nel contesto delle *Service Oriented Architecture*s (**SOA**). Questo termine, che nel tempo ha assunto connotazioni diverse per persone diverse, può essere definito a grandi linee come la pratica di decomporre un'applicazione in più servizi rappresentanti sottosistemi logicamente indipendenti, in una separazione orizzontale (componenti di uguale importanza ma che trattano aspetti diversi della logica di business) o verticale (divisione della logica in tier gerarchici). Tuttavia, le architetture a microservizi si distanziano dalle best practices delle SOA, che spesso sono invece considerati anti-pattern per un sistema a microservizi; alcuni esponenti sostengono infatti come i microservizi siano "*Se i SOA fossero ben fatti*" [CdlT23, 25].

Più in generale, i microservizi specializzano il concetto più generico di SOA, introducendo tecniche e requisiti più restrittivi e che mirano a ottenere un risultato specifico: quello di definire un applicativo sulla base di servizi della dimensione minima necessaria ad operare su un contesto ben delineato e con un basso accoppiamento tra di essi, favorendo un deploy indipendente e una forte scalabilità orizzontale. In sostanza, i microservizi "*forniscono agilità a lungo termine*" [CdlT23, 25].

Per chiarire i problemi che un'architettura a microservizi si propone di risolvere, in opposizione al tradizionale approccio monolitico (in cui un unico eseguibile racchiude interamente la logica di business), ritengo utile elencare alcune delle caratteristiche principali di questo paradigma architetturale.

3.2.1 Incapsulamento e indipendenza

I microservizi sono ideati per essere autonomi e indipendenti, così da poter essere sviluppati, testati e distribuiti in maniera isolata. Ogni microservizio mantiene i propri runtime e dipendenze isolati dall'host su cui è eseguito, riducendo al minimo indispensabile le interazioni con altri servizi: questo approccio consente una maggiore flessibilità e agilità nello sviluppo e nella distribuzione delle applicazioni. Il ciclo di vita di un microservizio diventa così indipendente da quello degli altri elementi dell'ecosistema con cui interagisce.

Una conseguenza della separazione concettuale del singolo microservizio dall'architettura complessiva è che anche il modello concettuale diventa specifico e indipen-

dente. In un'applicazione monolitica, il *modello* di ogni dominio è sovrapposto agli altri: come risultato, spesso si lavora con singole entità che ricoprono ruoli diversi nel contesto di servizi differenti, generando ambiguità semantica e confusione nella gestione di database centralizzati, lo standard in un approccio monolitico. Viceversa, in un'architettura a microservizi, ogni microservizio ricopre esattamente un dominio indipendente (in maniera non dissimile dai *bounded context* in *Domain-Driven Design*): in questo modo, può definire un *modello concettuale* dalla struttura snella, efficiente e context-coherent, e gestire un eventuale *state* su database per-service che riflettano tale struttura. Tale soluzione presenta evidenti benefici: modificare l'implementazione (lo schema o addirittura il DBMS) non impatterà gli altri servizi che accedono ai dati, mentre la creazione di un singolo punto di accesso ai dati riduce la possibilità di incoerenza nei dati e aiuta a prevenire e individuare bug. La separazione in database distinti consente inoltre di scegliere, per ogni contesto, *tipi* diversi di database, magari coniugando soluzioni SQL e NoSQL: tale approccio è detto *persistenza poliglotta*[CdlT23, 29].

In sintesi, l'incapsulamento di logica e dati di un microservizio consente di sviluppare e aggiornare il servizio in maniera indipendente, restringendo la coordinazione con team assegnati ad altri domini alla sola progettazione delle API pubbliche.

3.2.2 Smart Endpoints e Dumb Pipes per una comunicazione scalabile

Spesso il design di SOA di software complessi si riduce a un collage di monoliti, la cui logica di comunicazione viene centralizzata mediante infrastrutture software (i cosiddetti *service bus*). Tale pratica comune ha come ineluttabile conseguenza quella di accoppiare i servizi con l'infrastruttura, riducendo drasticamente l'indipendenza dei team di sviluppo che si ritrovano a far defluire parte della logica nella "terra di mezzo" del middleware di comunicazione, pronta alla "contaminazione" da parte di altri domini. In opposizione a questa pratica, i microservizi adottano una filosofia di comunicazione *smart endpoints, dumb pipes*[Den19, 19], abolendo l'uso di di middleware complesso e preferendo invece sistemi di comunicazione leggeri e *data-agnostic*: l'unico scopo dell'infrastruttura è recapitare i messaggi agli endpoint, evitando *leak* del contenuto informativo e della logica di business al di fuori dei microservizi ai capi della trasmissione.

Le tipologie di comunicazione adottate nel contesto di un'architettura a microservizi si scandiscono in base a due criteri di selezione[Den19, 19]:

- Comunicazione sincrona o asincrona: determina rispettivamente se il mittente attende una risposta dal destinatario prima di proseguire l'elaborazione, o se può continuare a operare indipendentemente dalla ricezione del messaggio. Una soluzione sincrona diffusa è l'utilizzo del protocollo **HTTP** per esporre **API RESTful**, garantendo una comunicazione chiara e *stateless*. Vari pattern di messaggistica asincrona sono invece implementabili mediante il protocollo **AMQP**;

- Comunicazione single-receiver o multi-receiver: stabilisce se il messaggio viene inviata a un singolo interlocutore (il fornitore del servizio richiesto) o in generale a più destinatari (tipici di questa categoria sono protocolli di tipo *publish/subscribe*, fondamentali nella progettazione di sistemi event-driven).

3.3 I trade-off della suddivisione in microservizi

Nelle ultime pagine abbiamo esplorato alcuni problemi delle architetture monolitiche/-SOA e come i microservizi si propongono di risolverli. Tuttavia, la scelta di separare un sistema software unico in applicativi indipendenti introduce alcune complicazioni, prevalentemente legate alla necessità di coordinare operazioni su più microservizi, richiedendo uno scambio di informazioni che sarebbe invece triviale tra moduli dello stesso processo. Discutiamo brevemente in questa sezione le colonne portanti del layer di complessità derivato dall'introduzione dei microservizi.

3.3.1 Resilienza distribuita delle operazioni

In sistemi monolitici, invocare un'operazione su un'entità dello stesso processo è banale, grazie a funzionalità integrate nei linguaggi di programmazione e nei framework consolidati che consentono una gestione generalmente intuitiva del *control flow*. Invece, in un'architettura a microservizi distribuita (e magari orchestrata), una richiesta potrebbe essere smistata sullo stesso host come in un altro server a distanza variabile, introducendo una latenza non stimabile a priori e in generale la possibilità di estendere la comunicazione a una rete inaffidabile.

Se un applicativo monolitico può introdurre facilmente indicatori di progresso e lanciare eccezioni o errori in caso di fallimento di un'operazione, l'apertura dei microservizi a una comunicazione sincrona in rete fa sì che, finché non riceve una risposta, a un mittente non è dato sapere se l'operazione sia in corso, sia fallita e nemmeno se la richiesta sia arrivata al destinatario.

Una soluzione comune è l'introduzione di un timeout, che consente al mittente di interrompere l'attesa di una risposta e gestire il fallimento dell'operazione in modo efficace. A tale soluzione si aggiungono solitamente alcune "strategie"[Den19] che consentono di reagire dinamicamente allo stato della rete per limitare i disagi causati da fallimenti "localizzati" delle richieste. Tra queste strategie si annoverano: **(i) Retry con Backoff esponenziale e Jitter**: Quando una chiamata fallisce, il *caller* ritenta la chiamata (con un limite di tentativi). Per non sovraccaricare il *callee*, i *retry* sono ritardati con andamento esponenziale, su cui è introdotto un *jitter*, cioè una deviazione dal tempo calcolato per evitare di inondare il *callee* di richieste contemporanee. (**ii**) **Circuit Breaker**: le richieste a un microservizio fornitore falliscono automaticamente dopo un numero prestabilito di fallimenti. Lo scopo è quello di evitare attese che con ogni probabilità precedono un timeout. (**iii**) **Bulkhead**: Le risorse per l'accesso concorrente sono suddivise per dipendenza, così che un *callee* guasto saturi solo una porzione dei worker asincroni. (**iiii**) **Fallback/Caching** In applicazioni in cui ricevere

l'ultima versione dei dati non sempre è cruciale, è possibile introdurre strutture che forniscano risultati alternativi o meno recenti in caso di fallimento della chiamata.

3.3.2 Integrità distribuita dei dati

Pur assumendo una comunicazione infallibile tra microservizi, i meccanismi forniti dai DBMS per garantire l'integrità dei dati, prime tra tutti le **transazioni**, sono indissolubilmente connessi al concetto di operazioni sui dati come elaborazioni compatte e indipendenti, il cui esito non può essere certamente determinato in un processo multi-step su una rete inaffidabile. Con il termine *transazione* ci si riferisce ad una sequenza di operazioni effettuate su una base dati che garantisce il rispetto delle proprietà ACID:

- Atomicity: le operazioni eseguite sono indivisibili: il risultato della transazione viene applicato (*commit*) solo se tutte le operazioni che la compongono hanno avuto successo, altrimenti i loro effetti sono annullati (*rollback*);
- Consistency: iniziando in uno stato coerente dei dati, una transazione può solo terminare nello stato precedente o un nuovo stato coerente.
- Isolation: ogni transazione non deve interferire con altre transazioni in esecuzione.
- Durability: alla fine di una transazione le modifiche sono salvate in maniera persistente, puntando a garantire l'integrità dei dati anche in caso di guasti o danni fisici al sistema.

In un contesto distribuito, non è semplice garantire le proprietà *ACID* di transazioni multi-servizio, poiché ognuno di essi gestisce i propri dati in maniera autonoma e le comunicazioni tra di essi possono fallire o essere ritardate. E' necessario introdurre meccanismi di *error safety* per garantire che, in caso di fallimento di una parte del sistema, l'integrità dei dati sia preservata e le operazioni possano essere ripristinate ad uno stato coerente. La soluzione è una "sovrastruttura" che diriga le transazioni per garantire uno stato coerente dei dati: si introduce così il concetto di *saga*.

Le saghe per dirigere le transazioni distribuite

L'obiettivo del pattern *saga* è quello di garantire la consistenza logica tra i database di più servizi, in linea con il principio di *eventual consistency* (coerenza finale eventuale) proprio della programmazione distribuita: ogni operazione che coinvolga l'update di più database *prima o poi* deve terminare garantendo uno stato coerente.

Nel concreto, come spiegato nelle guide Microsoft[Azu], una saga consiste in una sequenza di transazioni locali, in cui ogni (micro)servizio esegue l'operazione e avvia il passaggio successivo tramite eventi o messaggi (in alcuni casi, lo stato della saga può essere dedotto dal contesto, come nel caso di timeout nell'attesa di una transazione). Ogni transazione locale esegue il proprio *commit* e con una notifica richiede l'inizio

della fase successiva. Se uno dei passaggi fallisce, la saga prevede l'esecuzione di transazioni compensative per invertire le modifiche apportate agli step precedenti. I due approcci principali per coordinare le saghe sono *coreografia* e *orchestrazione*, tuttavia l'uso dell'orchestrazione è scoraggiato per coerenza con il principio di decentralizzazione proposto con la filosofia *smart endpoints, dumb pipes*.

L'orchestrazione prevede un componente centrale (*orchestrator*) che riceve gli eventi di avvio della saga e invia ordini ai servizi coinvolti. Tale soluzione evita dipendenze cicliche tra servizi e semplifica la logica di rollback complessivo, tuttavia l'orchestratore costituisce un *single point of failure*, non sempre scala bene, e l'approccio iterativo del flusso di controllo (opposto a quello ricorsivo della coreografia) aumenta il carico di messaggistica (comando/risposta per ogni nodo).

La coreografia affida ai servizi coinvolti il compito di gestire autonomamente la logica della saga, reagendo agli eventi generati dagli altri servizi. I principali svantaggi di questo approccio sono la difficoltà di tracciare lo stato della saga (nessun nodo mantiene lo stato dell'intera operazione, ma solo le informazioni sul suo ruolo) e una maggiore complessità nel coordinare le transazioni compensative in caso di fallimento.

Nel complesso, il modello di consistenza offerto dalle saghe passa da avere proprietà ACID (pur valido per le transazioni locali), a proprietà BASE:

- Basic Availability: il fatto che ci sia una saga in corso non impedisce a un'altra richiesta di dare inizio a un'altra saga;
- Soft state: anche in assenza di nuove richieste esterne, lo stato dei dati potrebbe variare finché tutte le saghe concorrenti non sono terminate;
- Eventual consistency: al termine di tutte le saghe concorrenti, i dati saranno in uno stato coerente e letture successive garantiranno gli stessi risultati.

3.3.3 Disacoppiare client e microservizi: i Gateway API

3.4 L'importanza dei container per le architetture a microservizi

Potremmo pensare ai microservizi come "pacchetti" preconfezionati di software appartenente a un certo dominio, e ciò ne descriverebbe intuitivamente i vantaggi. Tuttavia, quest'astrazione nasconde una serie di problemi che sono invece cruciali, soprattutto per la distribuzione di applicativi in ambienti server e non solo in architetture a microservizi. Tali problemi riguardano principalmente l'isolamento, la flessibilità di configurazione e la portabilità delle soluzioni software.Basti pensare al comicamente noto problema del "sul mio computer funziona": un codice teoricamente corretto e funzionante potrebbe risultare inutilizzabile quando esportato su un nuovo calcolatore. Conflitti con variabili d'ambiente, chiavi di registro o altri elementi del file system; problemi di autorizzazione; assenza o errata configurazione o versione delle dipendenze. Queste sono solo alcune delle innumerevoli e frustranti problematiche che

derivano dalla vicendevole contaminazione dell'ambiente in cui gli applicativi software vengono eseguiti.

C'è di più: finora abbiamo ignorato il *come* l'applicativo debba essere esportato su altri calcolatori: anche questa fase costituisce una fonte inesauribile di frustrazioni aleatorie, legate alle differenze tra le architetture dei processori, dei sistemi operativi, del supporto di date funzionalità. Più in generale, sarebbe ideale poter installare l'applicativo come uno *standalone* compreso delle dipendenze necessarie e isolato rispetto allo stato del sistema che lo esegue così da evitare problemi di configurazione. Ci si auspica inoltre di poter mettere facilmente a disposizione l'applicativo su Internet (così da non dover caricare manualmente una versione eseguibile dell'applicativo su ogni server), magari in più versioni e potendo configurare eventuali variabili system-specific del software in maniera semplice e integrata all'avvio.

Tutto questo e di più è possibile all'introduzione di un unico potente concetto: **i container**, strutture virtuali che consentono di confezionare le applicazioni come pacchetti pronti all'uso. Costituendo lo standard di fatto per la containerizzazione delle applicazioni, nel prosieguo della trattazione mi riferirò specificatamente all'uso dell'ecosistema *Docker* per descrivere le potenzialità di questa tecnologia.

3.4.1 Che cos'è un container?

Un container è un processo autonomo che include tutto e solo il necessario per eseguire un'applicazione: codice/eseguibili, runtime, dipendenze e variabili d'ambiente. Quando un ambiente software è confezionato in un container, tutto il necessario alla sua corretta esecuzione è mantenuto in uno spazio utente isolato nel sistema operativo host (quello su cui è eseguito), evitando conflitti con altri applicativi e servizi e garantendo che l'applicazione funzioni sempre come previsto, indipendentemente dall'ambiente di esecuzione. Tali proprietà fanno sì che la containerizzazione sia l'approccio ideale per distribuire facilmente applicazioni software in ambienti distribuiti, in cui l'avvio di un'istanza di software può essere determinato in modo dinamico e automatico da sistemi di orchestrazione: diventa imprescindibile essere in grado di dispiegare più applicativi (anche di più istanze dello stesso) su dispositivi eterogenei e *on the fly*, senza che un eventuale sistema di orchestrazione debba tenere conto di possibili conflitti. Con l'adozione dei container, i server dalle architetture più disparate possono essere considerati come risorse omogenee, delle griglie di "slot per container".

L'unico requisito affinché un server possa usufruire di questa feature "*Build Once, Deploy Everywhere*" è l'installazione di un *Docker Engine*, una piattaforma che contiene:

- Docker Daemon (`dockerd`), servizio che gestisce i container e le altre entità legate al mondo Docker;
- Docker API, che consente di interagire con il Daemon tramite richieste REST;
- Docker CLI, interfaccia a riga di comando che consente di interagire con il Daemon tramite comandi testuali.

Container contro Virtual Machine

Non è raro che qualcuno paragoni e confonda il concetto di Container con quello di Virtual Machine, in virtù del fatto che ogni container possiede un proprio spazio utente, con variabili d'ambiente, file system e tool di sistema diversi. In realtà, ogni container *crede* di gestire un proprio sistema operativo, ma la realtà è più complessa di così. Le macchine virtuali (VM) sono ambienti di esecuzione completi, con un sistema operativo proprio il cui kernel può differire da quello della macchina host grazie a una mappatura tra il sistema virtuale e quello reale.

Al contrario, i container condividono il kernel dell'host (o una VM fornita dall'host, come accade per i container Linux su Windows/MacOS), creando una copia personale solo dello spazio utente che spesso è minimale per garantire agilità nell'operare con essi. Essendo eseguiti direttamente sul sistema host, i container sono notevolmente più leggeri delle VM, somigliando più a dei "processi imbottiti" che a degli ambienti di esecuzione a sé stanti.

3.4.2 Ciclo di vita di un container

Come si fa a distribuire un'applicazione come container? In realtà, il container in sé rappresenta la singola istanza di un ambiente isolato di esecuzione, mentre l'artefatto che viene distribuito per consentire la creazione di container è detto *immagine*. Per spiegare intuitivamente la relazione tra immagini e container, riassumiamo il flusso da applicazione software a container.

- Sviluppo dell'applicazione: il codice sorgente dell'applicazione viene realizzato e testato in un ambiente di sviluppo.

- Scrittura del Dockerfile: in base della configurazione desiderata, si compone un file di testo, chiamato *Dockerfile* (privo di estensione), che contiene le informazioni necessarie per costruire l'immagine del container. Tra le altre cose, è obbligatorio specificare la base, un'immagine di partenza a cui aggiungere l'applicativo sviluppato, oltre a dipendenze, variabili e altre configurazioni. Una base può contenere una versione minimale di uno spazio utente, o prevedere delle componenti aggiuntive ottimizzate e pre-installate, come i runtime necessari per l'applicazione. Le immagini di base possono essere caricate dal file system locale o scaricate a *build time* da un registro di immagini online(come Docker Hub).

- Build dell'immagine: a partire dal codice sorgente dell'applicazione e dal Dockerfile, viene generata l'immagine, che rappresenta l'insieme spazio utente + applicazione come definito dalle istruzioni del Dockerfile.

- Distribuzione dell'immagine: l'immagine viene caricata su un registro di immagini (pubblico o privato) da cui può essere scaricata per creare nuovi container.

- Esecuzione del container: scaricando un'immagine, è possibile creare un nuovo container come "istanza" di essa, eventualmente configurabile con variabili e parametri contestuali tramite Docker CLI.

3.4.3 Persistenza e connettività dei container

La piattaforma Docker non si limita a gestire ambienti di esecuzione, ma offre anche varie funzionalità per consentire ai container di salvare i dati in modo persistente e di comunicare tra loro, con altri processi e con l'esterno.

Gestione dei dati

Il file system di un container Docker è effimero per definizione: quando il container viene eliminato, anche i dati al suo interno vanno perduti. Per questo motivo, Docker offre alcune alternative per preservare i dati sull'host.

Il **Bind mount** consiste nel mappare una cartella del filesystem dell'host su una cartella del container: in questo modo, i dati permangono anche con l'eliminazione del container. Questa soluzione porta un evidente svantaggio di portabilità, in quanto bisogna garantire che il ramo specificato sia sempre presente sull'host su cui viene eseguito il container: ciò causa problemi per path relativi e interventi inattesi sul ramo da parte dell'host.

Un **Named Volume** è uno storage dedicato sull'host e gestito da Docker indipendentemente da esso. Questa soluzione offre una maggiore portabilità ed è più automatica; per contro, accedere allo storage dall'host è meno intuitivo che navigare verso una cartella nota a priori (come per un bind mount).

Una soluzione non realmente persistente, ma utile in alcuni scenari, è il **tmpfs mount**, che consente di mappare una porzione di memoria RAM dell'host su una cartella del container. I dati memorizzati in un tmpfs mount sono volatili, ma l'accesso è molto più veloce rispetto a un volume su disco.

Connettività in rete

Nonostante il pregio principale dei container Docker sia il loro isolamento, ciò non significa che non dovrebbero poter comunicare tra loro e con l'esterno. Esistono due modi integrati in Docker per fornire connettività ai container.

Una rete `bridge` è una rete virtuale privata che consente ai container connessi di comunicare tra loro grazie ad alias e un DNS interno. La comunicazione in uscita con l'esterno è consentita di default in quanto un bridge fa NAT (*Network Address Translation*) verso l'host, mentre per abilitare una comunicazione bidirezionale è necessario utilizzare il meccanismo di *port forwarding* fornito da Docker, al fine di mappare porte dei container a porte dell'host.

Una rete `host` invece consente ai container connessi di condividere lo stesso stack di rete dell'host, il che significa che possono comunicare tra loro e con l'esterno utilizzando l'IP dell'host stesso. Questa soluzione garantisce performance elevate per la

comunicazione con l'esterno, ma sacrifica l'isolamento di rete tra i container e l'host.

È possibile inoltre definire delle reti personalizzate, di tipo `bridge` per personalizzare quali container possono comunicare tra loro, oppure con una configurazione `macvlan` che consente a ogni container di ottenere un indirizzo MAC virtuale e un indirizzo IP dedicato su una rete LAN. L'introduzione di sistemi di orchestrazione come *Docker Swarm* consente di definire anche reti cosiddette `overlay`, che consentono a container su host diversi di comunicare tra loro attraverso una rete virtuale distribuita.

3.4.4 Coordinare container multipli: da Docker Compose all'orchestrazione

Come spesso succede con le interfacce testuali potenti, i comandi Docker CLI rischiano di essere verbosi e poco chiari all'aumentare del numero di container da gestire. Fortunatamente, lo stesso ecosistema Docker risolve il problema fornendo metodi per dirigere un numero maggiore di container mediante l'automatizzazione della composizione di comandi Docker CLI che sarebbero altrimenti complessi e ripetitivi. Il tool che racchiude tali metodi è `Docker Compose`, un livello di astrazione sopra Docker che consente di coordinare uno stack di container correlati mediante una sintassi dichiarativa semplice su file *YAML* (`docker-compose.yml`).

Tuttavia, quando si tratta di gestire container in produzione, è necessario adottare approcci più avanzati e scalabili, che consentano di coordinare un numero variabile di container su architetture distribuite. Gli strumenti che consentono tali operazioni sono detti *orchestratori*.

Come spiegato chiaramente sul sito di *Red Hat*: "L'orchestrazione dei container è il processo che permette di automatizzare il deployment, la gestione, la scalabilità e il networking dei container attraverso l'intero ciclo di vita, consentendo di distribuire il software in modo uniforme in molto ambienti diversi e su larga scala." [Hat22]

Senza entrare nel dettaglio, un *orchestrator* si occupa di avviare, fermare, monitorare e replicare i container su diversi nodi di un cluster, garantendo alta disponibilità, bilanciamento del carico, gestione dei guasti e aggiornamenti senza interruzione del servizio. Gli orchestratori come *Kubernetes*, *Docker Swarm* e *Apache Mesos* consentono di coordinare in modo efficiente l'esecuzione di applicazioni composte da sciami di microservizi, semplificando la gestione operativa e migliorando la resilienza dei sistemi distribuiti. I sistemi di orchestrazione sono centrali in contesti che adottano filosofie come DevOps e CI/CD (Continuous Integration/Continuous Deployment), in quanto consentono di automatizzare il rilascio di nuove versioni del software e la gestione delle infrastrutture in maniera efficiente e affidabile.

3.4.5 Microservizi su container: un connubio spontaneo

Elenco delle figure

Elenco delle tabelle

Bibliografia

[Azu] Microsoft Azure. Modello di transazioni distribuite saga. learn.microsoft.com/it-it/azure/architecture/patterns/saga.

[CdlT23] Mike Rousos Cesar de la Torre, Bill Wagner. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, 2023.

[Den19] Francesco Dente. Devops e software containers in architetture a microservizi. Master's thesis, Università di Bologna, Dipartimento di Informatica - Scienza e Ingegneria, 2018–2019.

[Hat22] Red Hat. Cos'è l'orchestrazione dei container? `https://www.redhat.com/it/topics/containers/what-is-container-orchestration`, 2022.