

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	I
1 Introduzione	1
1.1 La reportistica aziendale automatizzata per favorire un modello decisionale data-driven	1
1.2 Il caso di studio	2
1.3 Tecnologie abilitanti	2
1.4 Struttura della tesi	3
2 Contesto applicativo: il mondo della reportistica automatica	4
2.1 Introduzione	4
2.2 Il problema affrontato	5
2.3 Confronto con lo stato dell'arte	5
3 Architetture a microservizi e servizi web evoluti	8
3.1 I limiti delle architetture monolitiche	8
3.2 I microservizi: ultima frontiera dei sistemi distribuiti	9
3.2.1 Incapsulamento e indipendenza	9
3.2.2 Smart Endpoints e Dumb Pipes per una comunicazione scalabile	10
3.3 I trade-off della suddivisione in microservizi	11
3.3.1 Resilienza distribuita delle operazioni	11
3.3.2 Integrità distribuita dei dati	12
3.3.3 Disaccoppiare client e microservizi: i Gateway API	13
3.4 L'importanza dei container per le architetture a microservizi	16
3.4.1 Che cos'è un container?	17
3.4.2 Ciclo di vita di un container	18
3.4.3 Persistenza e connettività dei container	18
3.4.4 Coordinare container multipli: da Docker Compose all'orchestrazione	19
3.4.5 Microservizi su container: un connubio spontaneo	20
3.5 Esporre servizi sul web	21
3.6 Tipologie di comunicazione	21
3.7 Evoluzione delle applicazioni web	25
3.8	25

4	Analisi del caso di studio	26
4.1	Introduzione	26
4.2	Panoramica della soluzione	26
4.2.1	Gateway API	27
4.2.2	AuthServer	28
4.2.3	Web API	31
4.2.4	Web App	33
4.2.5	UserDocuments	35
4.2.6	Global	37
4.2.7	Docker Compose	38

Abstract

Nel contesto della digitalizzazione 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 servizio¹, 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 (.xslm) 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 tradizionali², 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 **SignalR**, 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 Architectures* (**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 antipattern per un sistema a microservizi; alcuni esponenti sostengono infatti come i microservizi siano "*Se i SOA fossero ben fatti*" [CdIT23, 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*" [CdIT23, 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*[CdIT23, 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 "sovrastuttura" 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 Disaccoppiare client e microservizi: i Gateway API

Tra i principi di buona progettazione per un'applicazione a microservizi, uno che affronta una varietà di problematiche è il seguente: bisognerebbe limitare le richieste dirette dai client ai microservizi, rendendo l'organizzazione interna quanto più trasparente possibile all'esterno. Si immagina un'applicazione multiplatforma in cui ogni categoria di client accede direttamente alle API dei microservizi: la semplice adozione di una scelta simile si ripercuote negativamente su numerosi aspetti dell'applicativo:

- una modifica nella separazione delle responsabilità interne avrebbe un impatto elevato anche su tutti i client, necessitando modifiche estensive su tutta la codebase;
- un'elevata modularità dell'applicativo sarebbe bilanciata da un incremento delle comunicazioni, dato che ogni client potrebbe arrivare ad aprire una decina di comunicazioni sincrone solo per il caricamento di un'interfaccia grafica;

- sarebbe complesso gestire concetti trasversali come autenticazione, SSL, e il dispatch dinamico delle richieste;
- la gestione di client di diversi tipi (es. web e mobile) richiede un'elaborazione differente dei dati da presentare all'utente, e tale elaborazione non può essere affidata a dispositivi come quelli mobili, le cui risorse limitate richiedono che il server ottimizzi la comunicazione per raggiungere l'obiettivo con un impegno minimo da parte del client.

È facile intuire come, in una molteplicità di casi, la soluzione migliore sia quella di introdurre un tier intermedio, una facciata, che si occupi di accettare le singole richieste da parte dei client e tradurle in una moltitudine di richieste interne che riflettano la struttura corrente dei microservizi, astruendo dal client complessità e conoscenza dell'architettura del server.

Che cos'è un Gateway API?

Il pattern dedicato alla risoluzione di questo problema è il cosiddetto *Gateway API*. Un gateway API è un servizio che costituisce un punto di accesso centralizzato per certi gruppi di microservizi; spesso è chiamato anche Backend For Frontend (BFF) poiché rappresenta la porzione di architettura server progettata per venire incontro alle necessità delle applicazioni client. Dipendentemente dalla complessità della soluzione, lo strato di logica di facciata può essere anche costituito più gateway API scalabili in maniera indipendente, che si occupino di gestire tipologie accesso ai servizi differenti o per client differenti. Un'architettura di backend potrebbe ad esempio avere un gateway per i client desktop e web, uno per un'app mobile e uno per l'accesso ai servizi di amministrazione del sistema.

I gateway API non sono solo uno strato intermedio d'interfaccia tra client e microservizi, ma un vero e proprio tier a sé che concentra tutte le funzionalità che riguardano l'interazione tra di essi. Nel resto di questa sezione analizzeremo gli scopi principali di un gateway API e gli svantaggi derivati dall'impiego di questo nuovo layer, fornendo delle linee guida su quando adottare o meno tale pattern.

Feature principali

A seconda della complessità desiderata, un gateway API può concentrare un insieme più o meno ricco di funzionalità. La prima, e quella più frequentemente integrata, è il Reverse proxy: il gateway si espone come punto d'accesso centralizzato a un certo set di funzionalità fornite dai microservizi del backend. I client richiedono servizi al server mediante l'endpoint del gateway e questo, agendo da reverse proxy di livello applicazione, smista le richieste (solitamente HTTP) verso i microservizi incaricati di assolvere alla funzionalità d'interesse. L'utilità principale di questa feature è il totale disaccoppiamento dei client dai microservizi, tuttavia un reverse proxy risulta utile in software che mirano ad una separazione in microservizi, ma partono da un'architettura monolitica: in casi come questo è possibile "nascondere" l'intera architettura dietro

gateway API, modificando il routing interno dei proxy via via che le funzionalità si spostano dal monolite legacy a nuovi microservizi, con impatto nullo sul codice dei client.

Un'altra feature che è possibile implementare in un gateway API è l'aggregazione delle richieste, ossia la possibilità di trasformare una richiesta da parte del client all'endpoint del gateway in un insieme di richieste verso più microservizi. Ciò è particolarmente utile quando un client deve caricare una pagina o schermata che mostra informazioni dipendenti da microservizi diversi: il client può inviare un'unica richiesta e il gateway può prendersi carico di effettuare richieste multiple al backend e aggregare le risposte in un'unica risposta per il client. Bisogna tenere a mente che l'overhead di comunicazione che l'aggregazione si propone di risolvere è significativa per client remoti che si fanno carico dell'elaborazione, per cui spesso non vale la pena introdurre tale servizio per ambienti la cui interfaccia è generata server-side; è il caso ad esempio di ASP.NET Core MVC, il servizio per interfacce web utilizzato per il servizio di reportistica del caso d'uso, descritto nel dettaglio più avanti nel testo. Pur non essendo garantito che le soluzioni gateway API disponibili consentano l'aggregazione di default, è sempre possibile ottenere tale feature implementando la logica su un microservizio ausiliario che comunichi con il gateway stesso, ottenendo peraltro il massimo livello di personalizzazione.

Il resto delle funzionalità che è consigliabile introdurre in un gateway API rientrano nei cross-cutting concerns, ossia quelle categorie di feature trasversali e di feature che possono riguardare più microservizi e che è pertanto conveniente astrarre in un tier superiore piuttosto che implementare in ognuno dei microservizi. Ricordiamo tra queste i servizi di autenticazione/autorizzazione, che gestiscono l'identità e i permessi degli utenti per garantire la sicurezza; il caching delle risposte per incrementare le prestazioni memorizzando i risultati delle richieste frequenti; i servizi e guarantees per la comunicazione (QoS, retry, circuit breaker...) per l'affidabilità delle interazioni tra i componenti; il load balancing per distribuire il traffico in modo efficiente, prevenendo sovraccarichi; rate limiting per impedire un numero abusivo di richieste al servizio; il throttling, che regola la velocità di elaborazione; un logging centralizzato che raccoglie la diagnostica nel crocevia delle comunicazioni.[CdIT23, 44-45]

Gli svantaggi di un tier intermedio

L'introduzione di un gateway API non è una decisione da prendere alla leggera, in quanto l'aggiunta di un ulteriore livello di astrazione e comunicazione comporta inevitabilmente alcuni svantaggi. Creare uno o più punti di accesso centralizzato significa introdurre dei *single point of failure* nell'architettura: un guasto nel gateway renderebbe inaccessibili tutti i microservizi che espone, anche se questi operano correttamente. Soluzioni a tale problematica includono l'introduzione di sistemi di ridondanza per garantire l'alta disponibilità del servizio, ma ciò comporta un aumento della complessità e dei costi operativi che non sempre vale il guadagno nelle comunicazioni ottenuto con l'uso di gateway.

I gateway sono fortemente accoppiati con i microservizi che espongono, per cui

l'introduzione di questo nuovo tier si traduce nella necessità di aggiornare un secondo set di componenti al modificarsi del backend, e persino tecniche avanzate DevOps e CI/CD hanno un limite nella capacità di mitigare gli svantaggi di tale accoppiamento.

Inoltre, questo strato intermedio da cui passano tutte le comunicazioni può facilmente tramutarsi in un *bottleneck* se il sistema non è ben progettato per la scalabilità, sia da un punto di vista delle prestazioni sia per quanto concerne il coordinamento dei team di sviluppo, che come nell'approccio monolitico si ritrovano a dover collaborare su un unico componente che ospita logica da contesti differenti.[CdlT23, 47-48]

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

Nello sviluppo di architetture a microservizi, l'adozione di container si configura come una scelta quasi naturale, poiché risponde in modo intrinseco alle esigenze di isolamento, portabilità e scalabilità che caratterizzano questo paradigma. I microservizi, essendo entità autonome e indipendenti, traggono beneficio da un ambiente di esecuzione che garantisca un confine netto rispetto alle dipendenze esterne, preservando al contempo la leggerezza e la rapidità di distribuzione. In aggiunta, la granularità con cui i container possono essere orchestrati (tipicamente tramite gli strumenti sopra accennati) si allinea perfettamente al modello dei microservizi, permettendo di scalare selettivamente solo le componenti che presentano colli di bottiglia, ottimizzando così l'utilizzo delle risorse e garantendo elevata resilienza. In tale prospettiva, i container non soltanto uno strumento per aumentare l'efficienza del deploy, ma diviene un abilitatore architetturale che incarna in maniera pragmatica i principi cardine della filosofia a microservizi. Osserviamo nel dettaglio come il deployment in container risponda a un numero nutrito di esigenze specifiche attese dall'adozione di un sistema a microservizi. In primo luogo, la portabilità: un microservizio incapsulato in un container può essere eseguito in maniera uniforme su differenti ambienti (dallo sviluppo locale al cluster in cloud), senza dipendere dalle specifiche del sistema operativo o dalle librerie presenti sull'host. In secondo luogo, l'isolamento: ogni microservizio opera all'interno di un contesto dedicato che ne delimita le risorse, le dipendenze e la visibilità di rete, prevenendo conflitti e garantendo maggiore robustezza del sistema

complessivo. Altro requisito dei microservizi favorito dalla containerizzazione è la scalabilità dinamica: i container possono essere replicati e gestiti in maniera elastica, consentendo di rispondere con tempestività a variazioni nel carico di lavoro, spesso attraverso gli orchestratori che automatizzano il bilanciamento e il failover. Ancora, la rapidità di provisioning e rilascio: le immagini Docker, essendo versionate e distribuite in registry ottimizzati per il loro storage, consentono un ciclo di Continuous Integration/Continuous Delivery estremamente snello, riducendo i tempi di rilascio e facilitando rollback in caso di regressioni. Infine, la leggerezza e l'efficienza delle risorse: a differenza delle macchine virtuali, i container hanno un footprint ridotto e un avvio pressoché immediato, rendendo più agevole agli elaboratori che ospitano tali ambienti la gestione di un elevato numero di istanze eterogenee. In tal modo, il paradigma dei microservizi trova nel container il proprio substrato tecnico privilegiato, capace di tradurre i principi di indipendenza, modularità e resilienza in pratiche operative concrete ed efficaci.

3.5 Esporre servizi sul web

Quando si progetta un'architettura a microservizi, uno dei principali fattori che impattano le performance, la manutenibilità e la scalabilità è la scelta di come i microservizi espongono e fruiscono delle funzionalità di cui si fanno carico. In altre parole, bisogna definire l'*IPC* (Inter-Process Communication), ossia la struttura con cui i microservizi consentono agli altri elementi del sistema e ai client di interrogare le funzioni cui si propongono di assolvere. Ci sono varie categorie di protocolli e standard ben stabiliti, ognuno con i suoi vantaggi e svantaggi e i suoi ambiti applicativi: nel contesto dei servizi web distribuiti, ambito dominante per le architetture a microservizi, si distinguono quattro tecnologie che hanno fatto la storia. Dopo una panoramica sulle tipologie di comunicazione su cui un'API può basarsi, analizzeremo le caratteristiche principali di tali tecnologie e quali utilizzare a seconda dei parametri legati al contesto applicativo specifico.

3.6 Tipologie di comunicazione

Una prima grande distinzione nelle modalità in cui i microservizi possono comunicare è in base al tempo. Una comunicazione sincrona prevede che chi avvia la comunicazione rimanga in attesa bloccante di un riscontro: per questo motivo, il ricevente dovrebbe essere in grado di rispondere in tempi brevi ed eseguire in una configurazione ad **alta disponibilità**. Se il destinatario non è attivo o comunque in grado di recepire il messaggio, l'informazione viene persa. Protocolli che supportano comunicazione sincrona sono ad esempio **HTTP** e **gRPC**. Una comunicazione asincrona consiste in mittenti che inviano messaggi senza attendere una risposta, eventualmente gestendo l'esito **quando arriva la risposta, non quando avviene la richiesta**: i destinatari, se previsto, rispondono quando sono disponibili e ritengono opportuno

farlo, permettendo di configurare una comunicazione **flessibile** e aprendo le porte alle infrastrutture event-driven. I protocolli cruciali in quest'ambito sono quelli basati su messaggi, come AMQP.

Una seconda distinzione riguarda il formato delle informazioni scambiate: si distinguono infatti comunicazioni in formati text-based, come JSON o XML, che prediligono la chiarezza dei dati alla performance, e in formati binari, come il Protocol Buffer, che non temono l'aggiunta di un middleware di codifica al fine di ottenere una comunicazione *fulminea ed efficiente*.

Un'altro parametro centrale nella scelta nello stile di comunicazione da adottare è la molteplicità dei destinatari: in una comunicazione punto-punto (o uno-a-uno), ogni messaggio inviato da un microservizio è inteso per uno e un solo altro microservizio; viceversa, in una comunicazione uno-a-molti, il mittente invia il messaggio a più di un destinatario (non è raro che sia ignoto al mittente chi ascolta dall'altra parte) e solitamente non attende una risposta.

L'ultimo elemento fondamentale che determina lo stile di comunicazione è la tipologia di contenuto dei messaggi scambiati. In realtà questa distinzione non categorizza i protocolli, che possono infatti usare più di contenuto: è una distinzione che riguarda la singola comunicazione, e la prevalenza di una categoria su un'altra delinea la filosofia adottata dall'IPC scelta. Il messaggi possono essere:

- *informazioni*, se il loro contenuto sono esclusivamente dati
- *comandi*, se l'intento è quello di avviare una certa procedura in uno o più microservizi in ascolto
- *eventi*, se servono a comunicare ai microservizi in ascolto che lo stato del sistema si è evoluto, così che questi possano reagire in modo consono.

Volendoci concentrare, per ragioni di brevità, sulle combinazioni di caratteristiche che dominano nel contesto dell'IPC tra microservizi, identifichiamo una prevalenza di comunicazioni *sincrone* di tipo *Request/Response* (di conseguenza, tendenzialmente uno a uno), oppure *asincrone* di tipo *Publisher/Subscriber* (uno a molti) o *Message Queue* (uno a uno).

Per l'IPC sincrona, le soluzioni che costituiscono lo standard di fatto nelle architetture a microservizi sono REST, RPC e GraphQL.

REST (**RE**presentational **State** **T**ransfer) è una filosofia architetturale per l'implementazione di API basate su HTTP acclamata per la facilità di utilizzo e la disponibilità del servizio (è sufficiente HTTP 1.1). Risulta particolarmente vantaggiosa quando il servizio offerto è orientato alle *risorse* piuttosto che alle azioni.

Infatti, tutti i dati scambiati in una comunicazione *RESTful* (aggettivo che descrive un'API che abbraccia i principi della comunicazione REST) sono considerati risorse, la cui manipolazione avviene concordemente al *verbo HTTP* (come GET, PUT, POST e DELETE) associato alla richiesta. In buona sostanza, il client di una API RESTful può accedere a determinati URI del dominio a cui è esposto il server e scambiare risorse con esso secondo il tipo di verbo HTTP utilizzato. Ad esempio, una

GET all'endpoint `/products/{prodId}` potrebbe essere fatta da un client che vuole accedere a un dato prodotto, mentre un POST a `/products` consente di aggiungerne uno nuovo. È evidente come, per servizi orientati alle risorse, l'utilizzo di REST risulti naturale ed intuitivo, senza dover apprendere schemi complessi diversi da una documentazione che descriva le chiamate possibili ed eventuali parametri (peraltro, sono disponibili numerosi strumenti che consentono di generare automaticamente tale documentazione, ad esempio per il caso di studio nel capitolo 5 il sottoscritto ha usufruito di Swagger per generare un'interfaccia intuitiva e reattiva che descriva efficacemente le chiamate possibili e la loro struttura).

In realtà, i vincoli da rispettare affinché un'API sia considerata pienamente RESTful sono più stringenti. Accenniamoli brevemente:

1. **Architettura Client-Server:** Il client è responsabile della user interface e dello stato dell'applicazione, mentre il server gestisce le risorse e il loro stato.
2. **Stateless:** Ogni richiesta dal client al server deve contenere tutte le informazioni necessarie per essere compresa. Il server non deve conservare alcuna informazione di sessione relativa al client tra una richiesta e l'altra. Se un'autenticazione è necessaria, ogni richiesta deve includere le credenziali o un token di autenticazione.
3. **Cacheability:** Le risposte del server devono essere esplicitamente o implicitamente definite come memorizzabili nella cache o meno. Le risposte memorizzabili nella cache riducono la latenza e il carico sul server, migliorando le prestazioni dell'API. Se una risorsa viene contrassegnata come "cacheable", il client può riutilizzare una risposta precedentemente ricevuta per una richiesta successiva, senza dover contattare nuovamente il server.
4. **Interfaccia Uniforme:**
 - **Identificazione delle risorse:** Le singole risorse sono identificate da URI.
 - **Manipolazione mediante rappresentazioni:** Quando un client vuole manipolare una risorsa, ne richiede una rappresentazione (in un formato come XML o JSON), la modifica e invia una richiesta al server con la rappresentazione modificata per istruirlo sulle modifiche da apportare.
 - **Messaggi auto-descrittivi:** Ogni messaggio (richiesta e risposta) deve includere informazioni sufficienti per essere elaborato.
 - **Hypermedia as the Engine of Application State (HATEOAS):** Il client non deve conoscere a priori l'URI di ogni risorsa, ma è compito del server arricchire le proprie risposte con link a risorse correlate che potrebbero essere d'interesse.
5. **Sistema a strati** La richiesta del client può essere gestita da un intermediario in vece del server finale, senza che il primo ne sia consapevole.

6. **Code on Demand:** Facoltativo. Al server è concesso di estendere la funzionalità del client inviando codice eseguibile (ad esempio, JavaScript o applet).

Elenchiamo di seguito alcuni svantaggi nell'utilizzo dell'approccio REST, al fine di guidare una scelta consapevole sulla tecnica da adottare per un caso d'uso specifico:

- REST è pensato esclusivamente per una comunicazione sincrona *uno-a-uno* e *Request/Response*, risultando inutilizzabile per qualsiasi altro approccio;
- Risulta meno intuitivo, e anzi forzato, usare REST contesti applicativi in cui le operazioni da eseguire non sono prevalentemente di tipo *CRUD* (*Create*, *Read*, *Update*, *Delete*), ossia operazioni che creano, leggono, aggiornano o eliminano risorse. In questi casi, un paradigma orientato alle azioni (come RPC) risulta più naturale e performante;
- Basandosi esclusivamente su HTTP, REST non fornisce un modo semplice per ottenere risorse multiple con una sola richiesta: l'approccio atteso sarebbe una chiamata a sé stante per ogni risorsa, il che si traduce in estesi *overhead* che sarebbero altrimenti evitabili;
- L'assenza di un broker per la gestione della comunicazione rende REST semplice e snello, ma HTTP da solo non offre tecniche di message buffering, per cui tale approccio è conveniente solo in presenza di un server ad alta disponibilità, che garantisca di rispondere al client sempre e subito;
- Poiché gli endpoint sono esposti tramite URL, REST risulta poco flessibile in un contesto in cui l'API potrebbe evolvere o migrare. Per garantire una maggiore flessibilità su comunicazione REST, sarebbe necessario introdurre un meccanismo di *discovery* aggiuntivo che consenta ai client di localizzare dinamicamente gli endpoint dell'API.

Nel complesso, le caratteristiche descritte delineano REST come uno stile che predilige la semplicità e la chiarezza al di sopra di performance e funzionalità. Una soluzione talvolta proposta come alternativa più strutturata a REST è *GraphQL*. Si tratta di un linguaggio d'interrogazione per API che, al costo dell'introduzione di un nodo intermediario e dell'utilizzo di librerie aggiuntive sia lato client che lato server, vanta un sistema di richieste mirate che, previa definizione di uno schema delle risorse, consente al client di ottenere tutte e sole le informazioni necessarie con un'unica richiesta. Tale caratteristica sfida REST in una delle sue principali carenze: la granularità delle risorse è definita dall'API, per cui una distribuzione delle risorse fondamentale per alcune operazioni a grana fine può invece risultare tediosa e mal performante per operazioni su più ampia scala. Inoltre, l'approccio REST di associare ad ogni URI (e dunque ad ogni richiesta) una sola risorsa, impone al client di dover definire internamente algoritmi per il retrieval di informazioni strutturate che si traducono inevitabilmente in sequenza anche estese di richieste HTTP consecutive.

Altre funzionalità che elevano le potenzialità d'uso di GraphQL sono le migration, ossia la possibilità di specificare, direttamente a partire dallo schema delle risorse, delle modifiche sui dati da richiedere al server, e le subscription, ossia la possibilità di "isciversi" a date risorse per essere notificati dei loro aggiornamenti.

A seconda della complessità dell'API, dei costi, delle risorse computazionali a disposizione e della chiarezza target, GraphQL può o meno essere considerata un'alternativa preferibile a REST; in ogni caso, quest'ultima rimane la soluzione di gran lunga più adottata nel contesto delle web API per microservizi.

L'altro capostipite dell'IPC sincrona è il paradigma *RPC* (**R**emote **P**rocedure **C**all). La caratteristica fondamentale che distanzia REST e RPC è che quest'ultimo standard è pensato per elaborare *azioni*, non risorse. Un'immediata conseguenza di una filosofia orientata alle azioni è che l'intento delle chiamate, qui centro del paradigma, non viene più scandito dai verbi HTTP. Lo standard prevede infatti di limitarsi a richieste GET per le azioni read-only e richieste PUT per le operazioni che forniscono o alterano informazioni. Nella realizzazione di interfacce con maggior riguardo alle operazioni, RPC rappresenta lo stile più autodescrittivo e di facile comprensione, ed è caratterizzato da performance generalmente più elevate, poiché il payload delle richieste sarà solitamente circoscritto alle informazioni strettamente legate all'azione intrapresa, a differenza di REST che prevede quantità non indifferenti di risorse trasmesse ma inutilizzate. I principali drawback di questo stile di progettazione di API derivano dalla modalità di esposizione delle funzionalità: senza vincoli sull'organizzazione e la presentazione delle azioni, la disponibilità di una documentazione ben realizzata diviene imprescindibile, mentre con REST risultava intuitivo comprendere quali verbi siano applicabili a una data risorsa; al contempo, il fatto che ogni azione sia associata a un URL diverso pone il rischio non indifferente di una crescita sregolata degli endpoint all'aumentare delle funzionalità fornite. Un'implementazione che si è stabilita sopra le altre è **gRPC**, che si distacca ulteriormente dai principi di REST proponendo un sistema di invocazioni basate su **ProtoBuffer**, tecnologia che permette una formattazione binary-based dei messaggi per garantire efficienza elevata rispetto alle chiamate ripetute e ingombranti di REST. gRPC si propone come soluzione cross-language per ottenere chiamate efficienti e fortemente tipizzate nonostante l'approccio language-neutral della comunicazione, ottenuto grazie a una sintassi di definizione delle procedure che costituisce un accordo type-safe tra interfacce basate su framework e linguaggi distinti.

3.7 Evoluzione delle applicazioni web

3.8

Capitolo 4

Analisi del caso di studio

4.1 Introduzione

In questo capitolo ci dedicheremo a dare uno sguardo all'architettura prototipale proposta come prodotto del tirocinio curriculare. Forniremo una panoramica generale dell'architettura, descrivendo i progetti che costituiscono la soluzione complessiva ("progetto" e "soluzione" appartengono alla terminologia specifica nel contesto di uno sviluppo di Visual Studio in ambiente .NET) e il loro ruolo rispetto al resto dell'architettura. Si avrà inoltre cura di giustificare le scelte implementative effettuate, con riferimento a o in contrasto con la teoria e i principi di progettazione esposti nel terzo capitolo. Si ribadisce infatti come la soluzione proposta sia un prototipo, e in quanto tale devia consapevolmente dall'idea platonica del prodotto adatto alla produzione per assecondare vincoli di tempo e risorse inevitabili in un contesto di sviluppo con finalità didattiche. Nei casi in cui un aspetto dell'implementazione si discosti dai principi di buona progettazione delle architetture a microservizi, verrà prontamente indicato l'approccio che, in un contesto di sviluppo professionale, si sarebbe invece adottato, eventualmente descrivendo i *trade-off* dell'impiego dell'una o dell'altra implementazione.

4.2 Panoramica della soluzione

La soluzione complessiva si articola in quattro progetti principali, realizzati con tecnologia *ASP.NET Core* in *.NET 8* (nuovo, supporto a lungo termine), che rappresentano i microservizi forniti dall'architettura, e tre progetti secondari, rispettivamente una libreria di utility, un progetto condiviso *.NET* per la definizione di modelli di dati condivisi, e il progetto Docker Compose per la gestione automatica e centralizzata dei container che ospitano i microservizi.

Si segnala nella struttura descritta una scelta di progettazione, consciamente non in linea con i principi di incapsulamento e indipendenza dei microservizi, effettuata per snellire un'architettura che sarebbe altrimenti eccessivamente complessa per il numero di servizi principali forniti. Un microservizio fa da gateway API e comunica

con l'esterno, mentre esso e gli altri microservizi comunicano fra loro in una rete Docker privata di tipo **bridge**. Ogni microservizio stateful utilizza per la persistenza un database SQLite: questa soluzione, pur non scalabile, è stata adottata per la semplicità di configurazione e gestione in un contesto prototipale, in cui scalabilità, efficienza e sicurezza non valgono l'impegno nell'impostare un sistema di gestione più complesso in assenza di parametri di scelta che sarebbero invece centrali in un contesto professionale e di produzione.

Una progettazione più rigorosa avrebbe richiesto un microservizio a sé stante che ospitasse il motore di generazione di reportistica ed esponesse il servizio verso i microservizi a gestione del client MVC e della Web API. Nel contesto di tale scelta, il contenuto del progetto condiviso *UserDocuments* sarebbe inglobato nel microservizio di reportistica, non necessitando Web API e client MVC di conoscere il modello di dati dei documenti generati. La soluzione prototipale prevede invece l'inclusione del motore di reportistica all'interno del microservizio per la Web API: la generazione di nuovi report è infatti limitata a tale container (il client MVC esegue un redirecting verso di esso, così da non esporre al browser endpoint diversi con lo stesso scopo finale), mentre l'accesso al sistema di storage dei report è condiviso mediante un *named volume* Docker, a sottolineare la provvisorietà di tale soluzione nel contesto di un progetto con prospettive di sviluppo in direzione di un deploy in ambienti distribuiti.

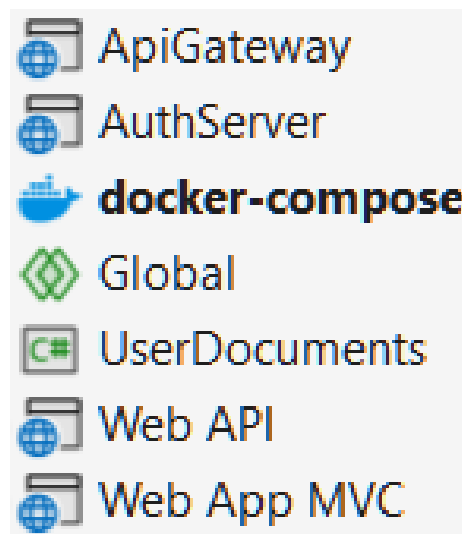


Figura 4.1: Elenco dei progetti che compongono il codice sorgente della soluzione

4.2.1 Gateway API

Il progetto *APIGateway* implementa il pattern omonimo nel contesto della soluzione proposta. Si tratta di un microservizio *ASP.NET Core* che funge da SPoA (*Single Point of Access*) per l'architettura, fornendo funzionalità di *reverse proxying* grazie all'impiego del pacchetto *YARP* e di centralizzazione dell'autenticazione con refresh intelligente dei token.

YARP (*Yet Another Reverse Proxy*) è un framework open-source per la creazione di proxy HTTP, sviluppato e mantenuto da Microsoft, che consente di instradare le richieste in ingresso verso uno o più servizi backend in modo flessibile e configurabile. Gran parte delle specifiche di instradamento, infatti, sono state definite direttamente come oggetto all'interno nel file di configurazione del progetto .NET (che per default è nominato *appsettings.json*), mentre le richieste di Logout, che implementano una logica più avanzata per l'invalidazione di cookie e token e per il sign out dal contesto dell'Identity, sono state configurate all'interno del middleware nel file di avvio del progetto *Program.cs*.

L'approccio di estrarre le informazioni variabili in un file di configurazione non riguarda soltanto il routing del reverse proxy, ma è adottato in maniera consistente nei progetti .NET dell'intera applicazione, consentendo non solo di mantenere un file di avvio snello e facilmente leggibile, ma anche di centralizzare la gestione delle configurazioni grazie all'elasticità fornita dall'ambiente .NET. Tali dati sono infatti estratti mediante un oggetto **ConfigurationManager**, che rappresenta un unico punto d'accesso alle configurazioni, dotato di un sistema di priorità e di metodi per lo strong typing dei valori ottenuti. Per accesso con priorità si intende che il **ConfigurationManager** di un applicazione ASP.NET è in grado di ricercare le configurazioni richieste da più fonti in una gerarchia stabilita, consentendo allo sviluppatore di specificare override diversi in *environment* diversi (sviluppo, testing, produzione) ed eventualmente di iniettare la configurazione come variabili d'ambiente mediante Docker, semplificando notevolmente il deploy in ambienti distribuiti in cui, ad esempio nel caso del gateway API, è necessario specificare all'avvio le informazioni sulla rete in cui il servizio viene installato.

L'altra funzionalità importante fornita dal progetto *APIGateway* è la gestione centralizzata dell'autenticazione e dell'autorizzazione degli utenti, implementando il pattern *Authentication Gateway*.

Al momento di una richiesta da parte di un client, il gateway verifica che l'utente sia autenticato prima che la richiesta raggiunga i microservizi interni e, in caso contrario, lo ridirige all'apposita interfaccia grafica fornita dal microservizio AuthServer, che analizzeremo in seguito (se la richiesta è programmatica restituisce invece **Error 401**): una volta autenticato, all'utente sono associati un token JWT (JSON Web Token) e un cookie, che il gateway provvederà ad inoltrare ai microservizi backend e a rinnovare periodicamente in modo trasparente per l'utente: il token JWT e il cookie consentono all'utente di autenticarsi una volta sola (presso il gateway) e restare connesso per l'uso di Web API e client MVC rispettivamente.

4.2.2 AuthServer

Il progetto *AuthServer* descrive il microservizio che fa da Identity Provider per l'applicativo. Utilizza il framework *Duende Identity Server* per implementare e configurare con pochi metodi un servizio di autenticazione completo basato sul protocollo OpenID Connect. Per la gestione di utenti e ruoli si affida invece ad *Identity* con *EntityFramework Core*: grazie alla potenza e l'interazione *seamless* dei framework,

in questo progetto la persistenza è stata gestita in maniera potente ed integrata senza dover scrivere una sola riga di codice *SQL* o dover configurare manualmente il database (qui *SQLite* ospitato su un *named volume Docker*, configurazione accettabile in un ambiente prototipale per motivazioni discusse in precedenza). Si pensi che l'intera configurazione dell'Identity Server con le funzionalità essenziali (2FA, login con provider esterni o email sender sarebbero inutili in un contesto prototipale offline) è stata realizzata con l'invocazione di meno di 7 metodi nell'entrypoint del servizio (*Program.cs*).

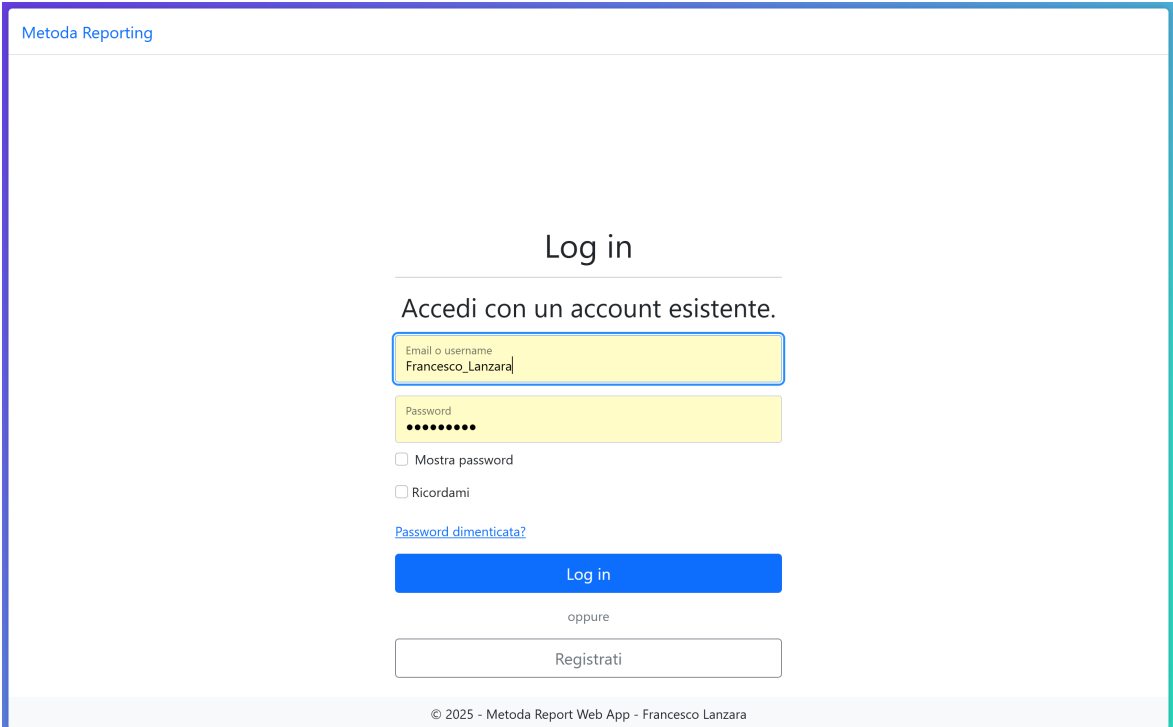


Figura 4.2: La schermata di login. Da questa schermata è possibile accedere anche alla registrazione e al recupero della password.

Il sistema di scaffolding .NET, che consente la generazione automatica di codice *boilerplate* per la configurazione rapida delle funzionalità offerte dai framework, è stato utilizzato per generare la struttura alla base del rendering delle pagine di accesso, registrazione e gestione del profilo utente, che sono state poi personalizzate per adattarle alle esigenze del progetto e rese gradevoli alla vista con classi del framework *Bootstrap* per *CSS*. Si tenga conto che nonostante la scelta, analizzata nel capitolo quarto, di implementare l'interfaccia grafica della web app mediante ASP.NET Core MVC, l'implementazione di base generata per interfaccia di autenticazione fa utilizzo della tecnologia *Razor Pages*, che adotta un approccio *MVVC-like*, basato sul concetto di *PageModel*, associando a ogni pagina *.cshtml* un file *.cs* che ne descrive la logica e il modello: il risultato è una struttura più fluida e meno verbosa che fa largo uso di convenzioni, rendendo semplice implementare funzionalità comuni ma risultando nel

complesso meno flessibile di MVC.

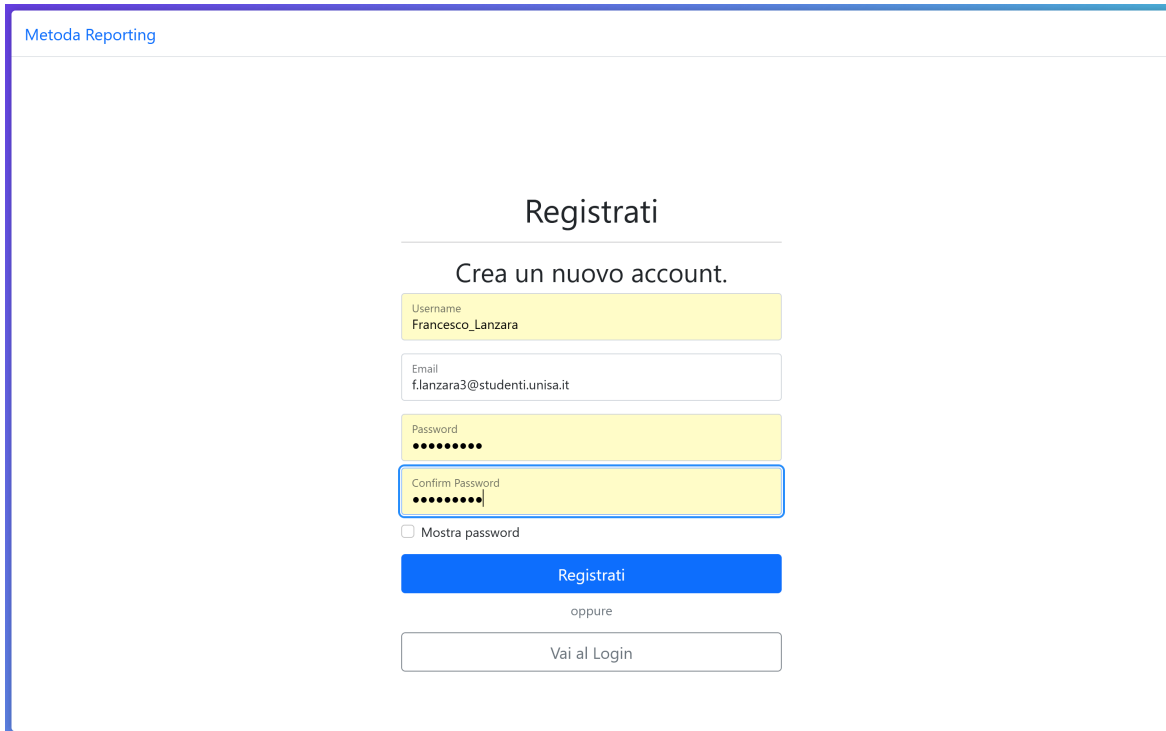


Figura 4.3: La schermata di registrazione. Da questa schermata è possibile tornare al login.

Dal momento che la logica di autenticazione è gestita internamente lontano dagli occhi dello sviluppatore, il sottoscritto ha ritenuto preferibile mantenere l'implementazione già predisposta, limitandosi a personalizzare aspetto e interazione delle pagine, piuttosto che impiegare tempo per individuare una soluzione alternativa nel tentativo di mantenere coerenza con una scelta implementativa che riguarda un altro microservizio dell'architettura. In un contesto in cui un controllo chiaro sull'implementazione non è ottenibile indipendentemente dalla tecnologia impiegata, il vantaggio principale del pattern MVC viene meno, giustificando ulteriormente la scelta di adottare Blazor per l'AuthServer.

AuthServer si occupa dunque di rispondere alle richieste di autenticazione e fornire le schermate per il login, la registrazione ed altre funzionalità legate all'account, come reset e modifica della password, gestione di altre informazioni del profilo quali recapiti telefonico e mail, logout. Si tenga conto che un utente già autenticato può accedere a queste pagine senza perdere l'accesso, ritornando all'applicazione MVC tramite click sul nome del servizio (estremo sinistro del layout superiore). Esso espone inoltre gli endpoint necessari per il protocollo OpenID Connect, che consente ai client di autenticare gli utenti e ottenere informazioni sui loro profili in modo sicuro e standardizzato. Tali endpoint sono utilizzati dal gateway API per autenticare gli utenti che tentano di accedere ai microservizi protetti dell'architettura. L'accesso dell'utente mediante *OpenID Connect* fornisce al browser un cookie di sessione e un codice, che il gateway

(punto di accesso dell'utente) utilizza per ottenere un token JWT, inoltrato alla Web API per autorizzare delle richieste.

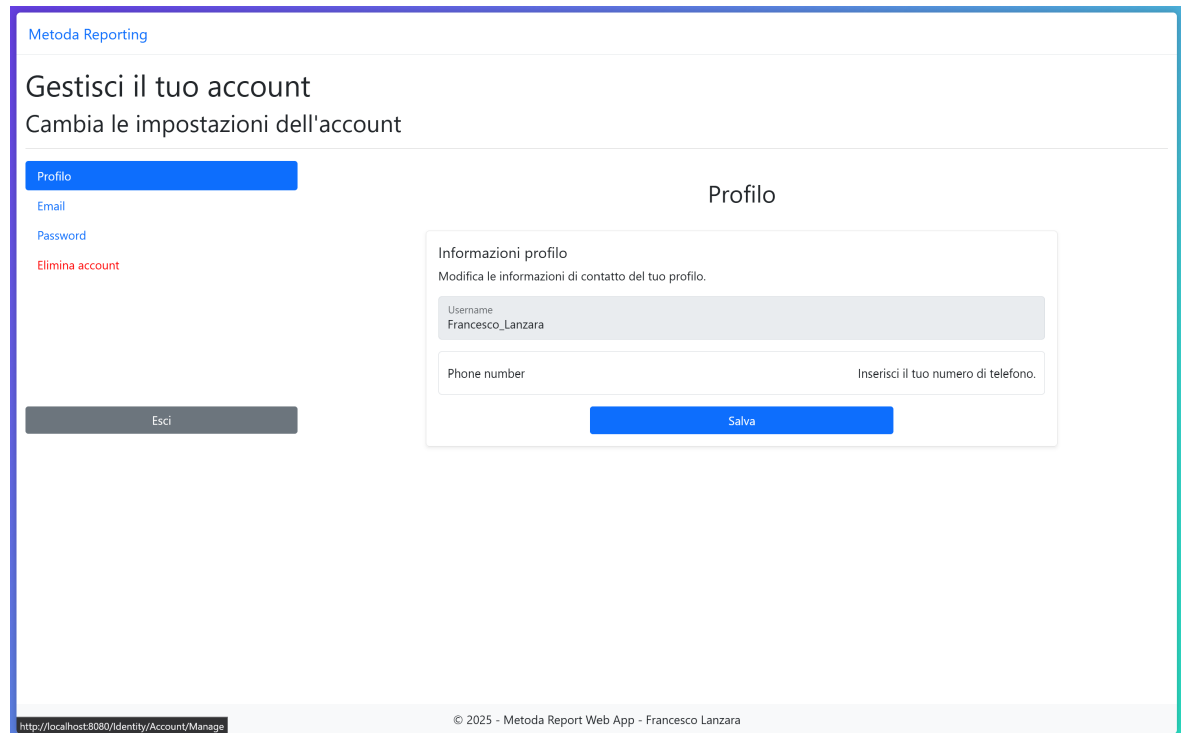


Figura 4.4: La schermata di gestione account. Contiene le sezioni *profilo*, *email*, *password*. E' da questa schermata inoltre che l'utente può effettuare il *logout* ed *eliminare* l'account, i dati a esso associati e i documenti da esso generati.

In aggiunta a un sistema di refresh automatico dei token, tale sistema consente all'utente di accedere a tutti i microservizi protetti dell'architettura senza preoccuparsi di dover effettuare nuovi login, garantendo un'esperienza utente fluida e sicura che ignora la suddivisione dell'architettura in container distinti.

4.2.3 Web API

Il progetto *WebAPI* implementa il microservizio che espone le API RESTful per la generazione automatica dei report e il loro salvataggio nella sezione dello storage dedicata all'utente: infatti, l'API è protetta dal gateway e pertanto risulta accessibile solo previa autenticazione. Per realizzare tale servizio si è fatto uso del framework *ASP.NET Core Web API*, che consente di creare API RESTful in modo semplice e veloce.

Come accennato in precedenza, tale microservizio ospita il motore di generazione automatica dei report, i cui moduli sono referenziati nel progetto. Lo stato persistente è mantenuto su un *named volume Docker*, su cui un database SQLite (il cui schema si fonda sul model fornito dal progetto condiviso *UserDocuments*) memorizza i metadati

sui documenti generati, associandoli agli utenti mediante il loro `SubjectId` (identificativo univoco fornito da OpenID Connect), mentre i file stessi sono salvati su file system, con una struttura di cartelle che li organizza per utente e data di creazione. Come prospettiva futura, si dovrebbe aggiungere un sistema di gestione dello spazio utilizzato, come una *policy* di eliminazione automatica dei documenti più vecchi o un limite massimo di spazio utilizzabile per utente, per evitare che lo storage cresca indefinitamente.

Un Controller API astratto implementa, tra gli altri, i due metodi di base necessari alla generazione dei documenti, per l'esportazione rispettivamente in formati PDF ed Excel (`.xlsx`), che possono essere chiamati dai controller concreti apponendo i tipi parametrici adeguati (tra quelli disponibili del motore di reportistica) al fine di specificare categoria di report e *data source* (per il testing sono state utilizzate classi statiche di tipo `-FakeData` predisposte dal modulo di test del motore di reportistica) e ottenere il report desiderato con un approccio semplice e modulare. Un ulteriore metodo è messo a disposizione per il *retrieval* dei documenti precedentemente generati dall'utente e presenti in storage: come vedremo a breve, tale metodo è utilizzato dal client MVC per popolare una lista di documenti recenti con link predisposti per il download degli stessi.

La generazione dei documenti avviene in modo asincrono, con la creazione di un nuovo task per ogni richiesta, in modo da non bloccare il thread principale del server e consentire la gestione di più richieste contemporaneamente. Una volta completata la generazione, il documento viene salvato nello storage e i metadati vengono aggiornati nel database.

Essendo il principale incaricato della generazione dei report, il microservizio Web API istanzia un Hub SignalR per l'aggiornamento in tempo reale sullo stato di generazione dei report. In questo modo, quando la generazione di un documento viene richiesta dal client MVC, esso si iscrive all'hub; il controller della Web API associa all'oggetto report un `ReportProgress`, classe del motore di reportistica, che in maniera nativa consente di associare una *callback* al progresso nella fase di generazione. Impostando come callback la procedura di emissione di un evento SignalR all'utente, questo può visualizzare a schermo notifiche in tempo reale sulla percentuale di avanzamento dell'operazione richiesta.

Tutti i controller concreti per la generazione espongono endpoint `GET` alla route URL `{dominio}/api/{CategoriaReport}/{pdf|excel}`, e si limitano a restituire in maniera asincrona il risultato del metodo generico ereditato, specificando i parametri di tipo adeguati. Ad esempio, per generare un report di tipo *MonthlyReport* (Report Analitico per Controparte) in formato PDF, il client deve invocare l'endpoint `{dominio}/api/MonthlyReport/pdf`, ottenendo in risposta il file generato.

Un ultimo controller che non riguarda la generazione è esposto all'endpoint `{dominio}/internal/userdocs`, e rappresenta l'insieme di api intese per uso amministrativo nell'ambito del motore di reportistica. Tale endpoint interno non è esposto dal gateway, e allo stato corrente del progetto consente di eliminare tutti i documenti e i dati associati a un utente: questa operazione è richiesta dal microservizio AuthServer al momento della conferma di eliminazione dell'account da parte dell'utente: si rispetta

così il principio di *data minimization*, non conservando dati personali dell'utente oltre il tempo necessario. Il controller interno, in' ipotetica evoluzione futura del progetto in un ambiente distribuito, dovrebbe essere trasferito nel microservizio di reportistica separato, come discusso in precedenza, accanto al modello di dati correntemente importato dal progetto condiviso *UserDocuments*.

Infine, *WebAPI* utilizza il progetto condiviso *UserDocuments*, che definisce il modello di dati per i documenti generati e i metadati ad essi associati, in modo da poter interagire con il database SQLite in modo tipizzato e sicuro.

4.2.4 Web App

Il progetto *WebApp MVC* implementa il microservizio che ospita l'interfaccia grafica dell'applicazione, e si basa sul framework ASP.NET Core MVC. Tale interfaccia consente all'utente di interagire con l'architettura, richiedendo la generazione dei report e visualizzando i documenti generati dall'utente in precedenza, che come per la Web API sono memorizzati in uno storage del server avvalendosi della tecnologia *EntityFramework Core*. Infatti, come accennato in precedenza il client MVC non comunica direttamente con il microservizio di reportistica, ma effettua un redirecting verso gli endpoint esposti dalla Web API per la generazione dei documenti; l'accesso ai documenti generati in precedenza è invece indipendente, poiché il volume che ospita i documenti e il db per i metadati associati è condiviso fra i due microservizi, così da non sovraccaricare il container Web API quando possibile. Come spiegato in precedenza, tale articolazione che sembra voltare le spalle al principio di *SoC* (*Separazione delle Responsabilità*) è una conseguenza di una scelta progettuale consapevole, quella di inglobare il motore di reportistica nel microservizio della Web API, che in un contesto di sviluppo professionale e distribuito sarebbe stata evitata.

L'interfaccia grafica è stata realizzata con l'ausilio del framework *Bootstrap*, al fine di creare layout responsive e gradevoli alla vista in modo semplice e veloce, grazie a una vasta libreria di componenti predefiniti e personalizzabili. L'uso di tale framework facilita inoltre il mantenere un aspetto coerente e professionale in tutte le pagine dell'applicazione, migliorando l'esperienza utente.

L'architettura MVC consente di separare chiaramente le responsabilità tra modello, vista e controller, facilitando la manutenzione e l'estensibilità del codice. I controller gestiscono le richieste HTTP, interagiscono con i modelli per recuperare o modificare i dati e selezionano le viste appropriate per la presentazione all'utente.

Le view principali esposte dal prototipo, oltre a quelle di default fornite dal template e personalizzate per l'applicazione, sono la home page e la pagina del profilo utente. La home page è esposta all'endpoint radice del dominio, e consente all'utente di richiedere la generazione di nuovi report mediante una lista di *card*, una per ogni categoria di report generabile, ognuna dotata due pulsanti, uno per ciascun formato supportato (PDF ed Excel). Una checkbox (la cui preferenza è memorizzata associandola al profilo utente) consente di determinare se aprire direttamente i report richiesti in una nuova scheda del browser, o se generarli e renderli successivamente disponibili dall'elenco dei documenti dell'utente. Grazie alla connessione del client

MVC con l’hub SignalR esposto dalla Web API, la home viene aggiornata in tempo reale sullo stato di avanzamento della generazione del documento richiesto mediante una progress bar: al termine della generazione un *toast* (notifica popup) in alto a sinistra dello schermo informa l’utente del completamento dell’operazione, consentendogli opzionalmente di aprire il file generato.

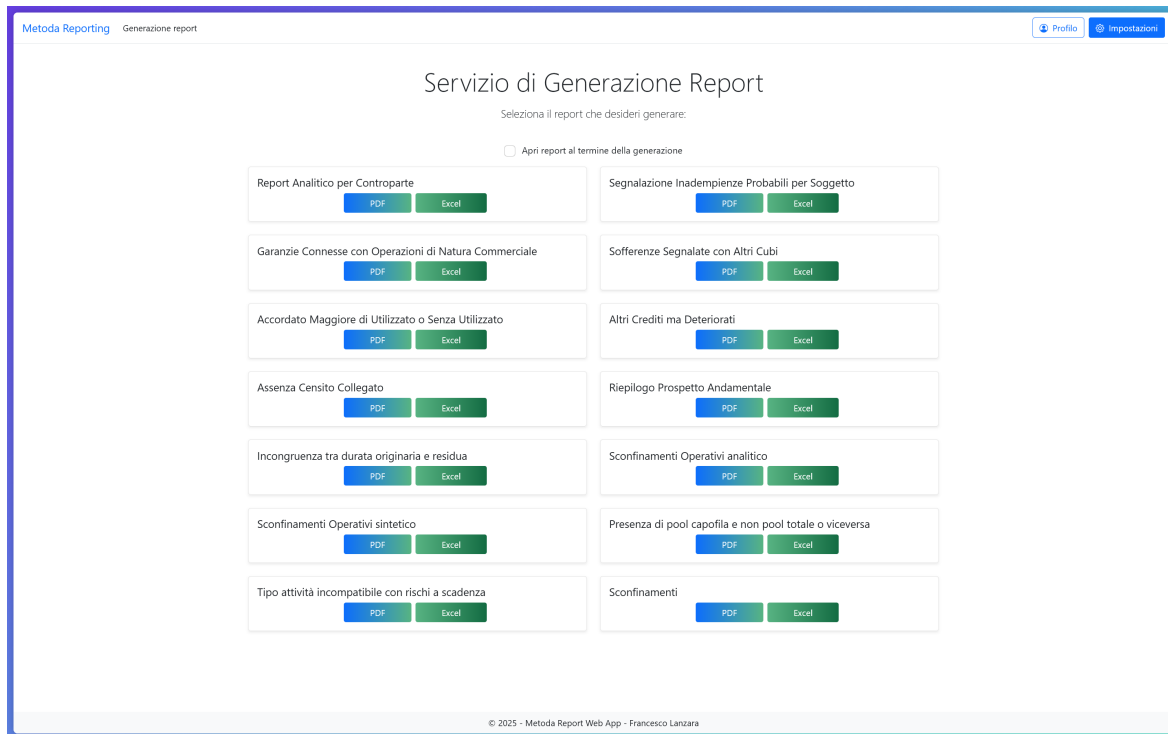


Figura 4.5: La schermata principale dell’applicazione. Da questa schermata è possibile richiedere la generazione di nuovi report.

La pagina del profilo utente, accessibile dal menu in alto a destra, consente di visualizzare i dati relativi al profilo e l’elenco dei documenti generati in precedenza, con link per il download diretto e informazioni sui metadati associati (data di creazione, formato, categoria). Per la popolazione della lista dei documenti, il client MVC agisce indipendentemente dalla Web API, usufruendo del model *EntityFramework Core* ottenuto referenziando il progetto condiviso *UserDocuments*. In questo modo, il client può accedere direttamente al database SQLite condiviso per recuperare i documenti associati all’utente autenticato, senza dover passare per la Web API, alleggerendo il carico su quest’ultima e migliorando le prestazioni complessive dell’applicazione. Un pulsante di logout consente di disconnettere l’utente, che viene reindirizzato alla pagina di login dell’AuthServer, mentre è possibile raggiungere la sezione di gestione dell’account (e indirettamente tutte le altre pagine che riguardano l’identità dell’utente, come login e registrazione) grazie al pulsante *Impostazioni* posto accanto a quello del profilo sulla barra di navigazione superiore.

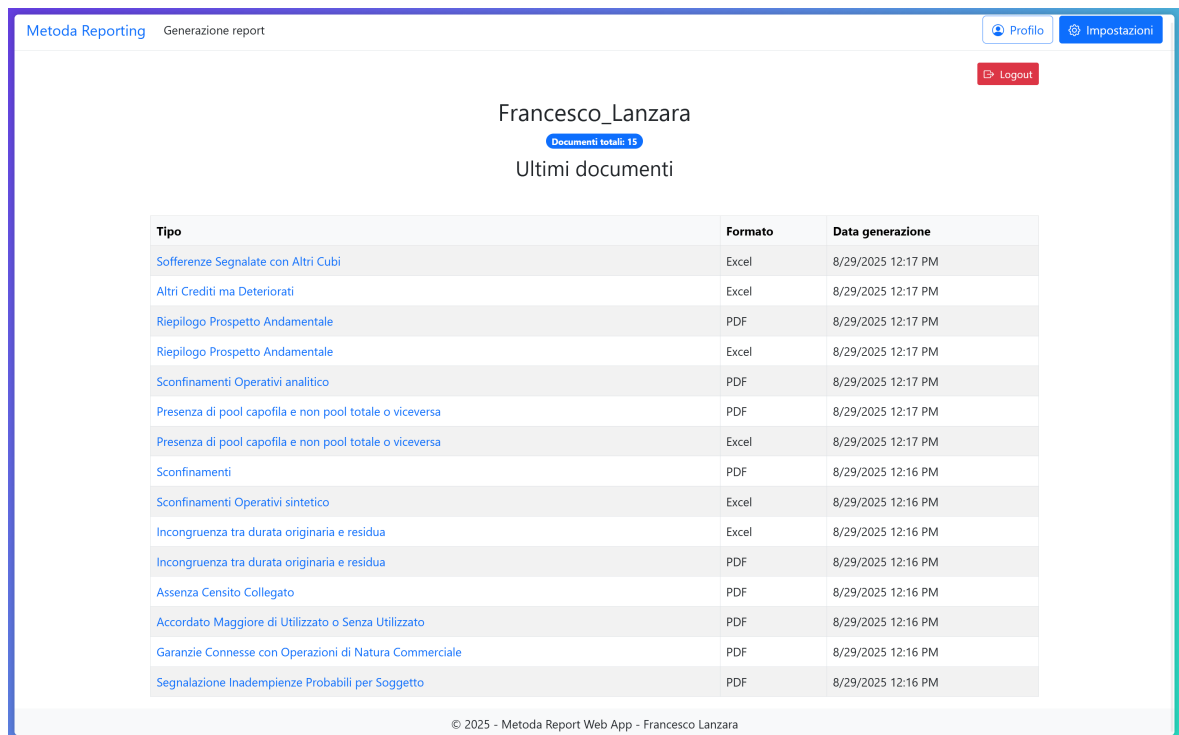


Figura 4.6: La schermata del profilo utente. Da questa schermata è possibile scaricare i documenti generati in precedenza, effettuare il logout e accedere alla gestione dell'account.

Nonostante l'autenticazione dell'utente sia delegata al tier superiore del gateway API, il client MVC è configurato per generare e validare *cookie Anti-Forgery* per proteggere le richieste **POST** (eseguite alla selezione di uno dei pulsanti di generazione report) da attacchi di tipo *Cross-Site Request Forgery (CSRF)*, pratica che rientra nelle best practice per la sicurezza delle applicazioni web.

4.2.5 UserDocuments

Come accennato in precedenza, *UserDocuments* è un progetto condiviso .NET che usufruisce del pacchetto *EntityFramework Core* per definire in maniera semplice e potente il modello di dati per i report generati dagli utenti e per i metadati ad essi associati. Tale progetto è referenziato sia dal microservizio Web API, che lo utilizza per interagire con il database SQLite durante la generazione dei report (anche per conto della web app), sia dal client MVC, che lo impiega per accedere direttamente al database e recuperare i report associati all'utente autenticato.

Esso consta di due *namespace* principali: *Models* e *Services* (i *namespace* sono uno degli elementi che consentono di organizzare gerarchicamente il codice in .NET). In *Models* troviamo ciò che riguarda la definizione del modello dei dati:

- **UserDoc** rappresenta il modello dei dati dei report generati, esponendo proprietà tra cui il *GUID* (identificativo univoco globale) dell'utente che ha generato

il documento (coordinato con il database dell'Identity Provider *AuthServer*), metadati sul documento e il percorso nel file system in cui è salvato.

- **UserDocsDbContext** estende la classe **DbContext** di *EntityFramework Core*: questo è il modo in cui quest'ultimo strumento riesce a mappare le classi C# come parte dello schema del database relazionale, consentendo come già detto di interagire con esso in modo tipizzato e sicuro. In questo caso specifico, l'esposizione di una proprietà **DbSet<UserDoc>** consente a *EF* di creare e gestire la tabella dei documenti generati dagli utenti, fornendo metodi per l'inserimento, l'aggiornamento, la cancellazione e la query dei record generati automaticamente a partire dalla classe di modello fornita.
- **DocumentContent** racchiude una *sealed hierarchy* di possibili valori, uno per ogni categoria di report supportata, che esposti come elenco di proprietà statiche consente di specificare in modo tipizzato e centralizzato la categoria di report da generare quando si invoca l'API della Web API. Si evitano inoltre stringhe *hardcoded* sparse per il codice specificando qui tutti i metadati necessari alla manipolazione dei titoli e dei *filename*: ciò aumenta la modularità e manutenibilità del codice, fornendo un unico file sorgente in cui apportare tutti gli aggiornamenti che riguardano il dominio applicativo (come le categorie dei report e la loro denominazione). Questa soluzione, oltre a ridurre la possibilità di bug dovuti a errori di battitura (viene in aiuto allo sviluppatore il sistema di *IntelliSense* dell'IDE, che fornisce suggerimenti e completamenti automatici basati sulla gerarchia fornita), permette di sfruttare al meglio l'interazione di *Razor* con l'ecosistema *C#/NET*. Come spiegato in precedenza, il framework *ASP.NET Core MVC* consente di creare viste dinamiche che interagiscono con la *codebase* in modo fluido e naturale, grazie alla sintassi *Razor*, che permette di mescolare HTML e C# in un unico file. Grazie a tale caratteristica, ad esempio, l'elenco di *card* nella home page del client MVC viene popolato dinamicamente iterando sull'elenco di categorie esposto da **DocumentContent**, evitando di dover ripetere manualmente il codice HTML per ogni categoria di report supportata. Segue un estratto del file **Index.cshtml** della home page, che mostra come viene realizzata tale iterazione:

```

1 @foreach (var report in reports)
2 {
3     [...]
4     <div class="card-body d-flex flex-column">
5         <h5 class="card-title">@report.Title</h5>
6         <div class="mt-auto d-flex justify-content-
           center gap-0">
7             @foreach (var format in formats)
8             {
9                 <form class="report-form" asp-
                   controller="Home" asp-action="
                   Index" method="post">

```

```

10         @Html.AntiForgeryToken()
11         <input type="hidden" name="
            docCategory" value="@report.
            ApiName" />
12         <input type="hidden" name="format"
            value="@format.Ext" />
13         <input type="hidden" name="openReport
            " value="false" />
14         <button type="submit" class="btn me-2
            @format.Css px-5">@format.Label </
            button>
15         </form>
16     }
17     </div>
18 </div>
19 [...]
20 }

```

Invece, in *Services* troviamo la classe `DocumentStorageService`, utilizzata dai controller di Web API e Web App per interagire in maniera standardizzata mediante metodi di accesso asincroni e altri tool di utility. I metodi sono pensati per integrare completamente la logica di gestione delle richieste, per cui presentano una *signature* (*firma*, ossia la definizione del metodo) adatta a gestire le principali operazioni che ci si attende dai controller: per questo motivo, non è necessario eseguire il *wrapping* di tali metodi in altra logica, ma è possibile utilizzarli direttamente, specificando i parametri del caso per ottenere il risultato desiderato. Un esempio sono i metodi `GenerateAndSavePdfReportAsync` e `GenerateAndSaveExcelReportAsync`, che incapsulano l'intera logica di generazione e salvataggio dei report, restituendo una `Task` che rappresenta il documento generato e i suoi metadati. In questo modo, i controller possono ottenere l'effetto collaterale del salvataggio in memoria, prima di restituire gli oggetti che nella gerarchia ASP.NET Core gestiscono in maniera asincrona la risposta HTTP al client contenente il file generato.

4.2.6 Global

Il progetto *Global* è un progetto di utilità che fornisce funzionalità condivise e helper per conferire modularità e manutenibilità agli altri progetti della soluzione. In particolare, esso contiene:

- la classe statica `Domain`, che contiene proprietà e metodi per l'elaborazione di valori come *host*, *schema* e *porta* e l'interpolazione dinamica dei path URL d'interesse per l'applicativo;
- una classe d'eccezione `InvalidConfigurationException`, che scandisce gli eventi di errata configurazione dell'applicativo, come la mancanza di variabili d'am-

biente o oggetti di `appsettings.json` necessari essenziali per il corretto funzionamento dei microservizi;

- la classe `ConfigurationExtractor`, istanziata da tutti i progetti *ASP.NET* nel file d'avvio `Program.cs` per usufruire dei metodi da essa forniti per il *retrieval type-safe* e *null-checked* (pena il lancio dell'eccezione descritta sopra) delle configurazioni del microservizio, garantendo che sviste nella modifica di file quali `Dockerfile`, `docker-compose.yml` o `appsettings.json` vengano prontamente individuate e segnalate, evitando scomode `NullReferenceException` a *runtime* che risultano in *stacktrace* verbose e per nulla esaustive in fase di debug.

4.2.7 Docker Compose

Il progetto *Docker Compose* è responsabile della definizione e gestione dei container Docker per l'applicativo. Il file fondamentale è `docker-compose.yml`, che come accennato nei capitoli precedenti definisce i servizi, le reti e i volumi necessari per l'esecuzione dell'applicativo in ambiente containerizzato. In esso sono specificati i dettagli di ogni microservizio, come il `Dockerfile` da utilizzare per la build, le porte esposte e condivise in rete privata, le variabili d'ambiente e le dipendenze tra i servizi. I servizi definiti sono 4, uno per ogni microservizio descritto in precedenza: *gateway*, *authserver*, *rest* e *mvcclient*. Ognuno di essi è configurato per utilizzare un'immagine Docker basata su *Windows nanoserver-1809 con runtime .NET 8 ASP.NET*; *gateway* è l'unico container a fare *port forwarding* sulla porta 8080 della macchina host, esponendo l'applicativo all'esterno, mentre gli altri container comunicano fra loro in una rete privata di tipo `bridge`, isolata dalla rete host, sempre utilizzando la porta 8080. Vengono definiti 4 *named volumes*: `dpkeys` per la persistenza delle chiavi di Data Protection, `docstorage` per il salvataggio dei report degli utenti, `docmetadatadb` per il database contenente i metadati sui report e `datakeys` per salvare le chiavi di crittografia per l'Identity Server. Nel complesso, il progetto *Docker Compose* costituisce un punto singolo d'avvio per l'applicativo, consentendo una gestione centralizzata e semplificata dei container che costituiscono l'architettura a microservizi.

Elenco delle figure

4.1	Elenco progetti	27
4.2	Schermata login	29
4.3	Schermata registrazione	30
4.4	Schermata gestione utente	31
4.5	Schermata home	34
4.6	Schermata profilo	35

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.