



SAPIENZA
UNIVERSITÀ DI ROMA

Sentinel: Sviluppo di un Modulo di Gestione Conti con Architettura Event-Driven e Frontend Angular

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Stefano Russo

Matricola 1864451

Relatore
Daniele Gorla

Relatore Aziendale
Mauro Felici

Anno Accademico 2023/2024

Tesi non ancora discussa

Sentinel: Sviluppo di un Modulo di Gestione Conti con Architettura Event-Driven e Frontend Angular

Relazione di Tirocinio. Sapienza Università di Roma

© 2024 Stefano Russo. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: stefanorusso.sr.99@gmail.com

Indice

1	Introduzione	1
1.1	Scopo del tirocinio	1
1.2	Descrizione di Sentinel	1
1.3	Organizzazione e conduzione del lavoro	2
2	Architettura e tecnologie utilizzate	3
2.1	Architettura Event-driven con Microservizi	3
2.2	Tecnologie utilizzate	4
2.2.1	RabbitMQ	4
2.2.2	PostgreSQL	6
2.2.3	Spring e SpringBoot	7
2.2.4	Angular	7
2.2.5	Stripe	8
2.2.6	Redis	8
2.2.7	Docker	9
2.2.8	Kubernetes	10
3	Progettazione concettuale e logica	12
3.1	Requisiti del database	12
3.2	Progettazione concettuale	12
3.2.1	Schema Entity-Relationship	12
3.2.2	Vincoli Esterni	15
3.3	Casi d'uso	15
4	Ristrutturazione e traduzione dello schema	17
4.1	Ristrutturazione dello schema Entity-Relationship	17
4.2	Traduzione dello schema ristrutturato	18
4.3	Traduzione dei vincoli esterni	20
4.4	Tabelle pre-popolate	20
5	Design del frontend	21
5.1	Sezione per la gestione del Wallet	21
5.2	Sezione per la gestione dei listini prezzi	23
6	Dettagli implementativi	26
6.1	Implementazione dello use-case Crea Listino	26

Indice	iii
6.2 RabbitMQ	29
6.3 Stripe	30
6.4 SignalStore di NgRx	31
7 Conclusioni	35
Bibliografia	36

Capitolo 1

Introduzione

1.1 Scopo del tirocinio

Questa relazione presenta il lavoro svolto durante il tirocinio presso l'azienda **RESI Informatica**, che ha avuto come obiettivo la creazione di un servizio per la gestione di conti virtuali e dei relativi listini prezzi per un'applicazione multi-tenant con architettura event-driven e microservizi.

Il nome deciso per questo servizio è **Credit Manager**.

1.2 Descrizione di Sentinel

Sentinel è un applicativo progettato per automatizzare l'analisi di portafogli di crediti deteriorati (NPL, Non-Performing Loans), fornendo un supporto per le società di recupero crediti nella valutazione dei possibili rischi.

Tra le funzionalità chiave offerte da Sentinel troviamo:

- **Ricerca e aggregazione** - Il sistema ha la capacità di aggregare grandi quantità di dati estrapolati da banche dati esterne quali SISTER (Agenzia delle entrate), Cerved e altre, e di utilizzarli per effettuare un'analisi approfondita del portafoglio elaborato.
- **Calcolo del potenziale recupero** - Tra i dati che Sentinel recupera abbiamo dati relativi al valore degli asset in possesso dei soggetti debitori, che possono quindi essere utilizzati per effettuare una stima dei ricavi.
- **Supporto decisionale per le strategie di recupero** - Attraverso le analisi che Sentinel effettua sul portafoglio, è in grado di suggerire delle strategie da seguire per il recupero in modo da minimizzare i rischi.
- **Generazione automatica di documenti** - Sentinel è in grado di generare i documenti necessari per intraprendere azioni legali.

- **Interfaccia semplice da utilizzare** - Sentinel è dotato di molteplici dashboard che consentono di visualizzare in maniera comprensiva lo stato di un portafoglio, le pratiche di credito collegate, e di effettuare simulazioni.

1.3 Organizzazione e conduzione del lavoro

Le fasi del progetto sono state scandite dai seguenti punti:

- Scelta della tech stack
- Apprendimento delle basi di Angular, Spring e RabbitMQ
- Progettazione della base dati
- Progettazione delle interfacce del frontend
- Realizzazione del backend
- Realizzazione del frontend con adattamento del backend per i nuovi sviluppi

Durante il tirocinio ho lavorato a stretto contatto con **Luca Pittori** e **Roberto Tosolini**, con la supervisione del mio relatore aziendale **Mauro Felici**. Con il primo ho affrontato la parte di progettazione, scelta della tech stack, apprendimento iniziale delle tecnologie. Con il secondo ho affrontato la parte di realizzazione del backend, di progettazione dell'interfaccia e di realizzazione del frontend.

Come piattaforma per hostare il codice sorgente è stata utilizzata un'istanza locale di GitLab, mentre per la comunicazione sono stati utilizzati Skype e Microsoft Teams. Il progetto è stato realizzato seguendo la metodologia Agile, i cui principi sono riassunti nel seguente manifesto[1]:

***Gli individui e le interazioni** più che i processi e gli strumenti
Il software funzionante più che la documentazione esaustiva
La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano*

*Ovvero, fermo restando il valore delle voci a destra,
consideriamo più importanti le voci a sinistra.*

In particolare, ci sono state continue interazioni per raffinare gli aspetti progettuali e implementativi, permettendo cicli di sviluppo più brevi alla fine dei quali sono stati raggiunti dei risultati concreti per il progetto.

Capitolo 2

Architettura e tecnologie utilizzate

In questo capitolo verrà descritta l'architettura di Sentinel e la tech stack utilizzata dal Credit Manager, di cui la maggior parte condivisa con il resto dei servizi.

2.1 Architettura Event-driven con Microservizi

Viste le specifiche richieste da Sentinel come sistema, si è optato per utilizzare un'architettura Event-driven con Microservizi. In questo tipo di architettura il sistema è diviso in servizi indipendenti che sono responsabili di una porzione delle funzionalità. I servizi comunicano tra loro attraverso un message broker, in questo caso RabbitMQ, che si occupa di distribuire i messaggi.

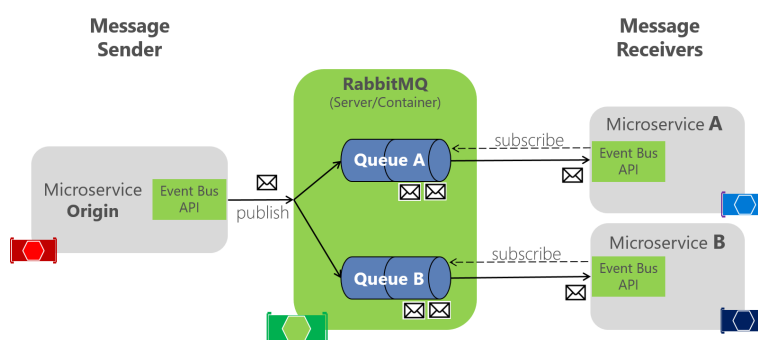


Figura 2.1. Grafico illustrativo del messaging tra Microservizi attraverso RabbitMQ[2]

Utilizzare questa architettura significa avere dei servizi che non sono strettamente collegati tra loro, e questo ha diversi vantaggi:

- **Comunicazione Asincrona** - Nonostante l'overhead di avere il message broker tra le comunicazioni, questo pattern di comunicazione può portare a incrementi delle prestazioni, vista la loro natura asincrona. Per esempio, questo

studio[3] ha osservato una riduzione nel tempo di risposta fino al 19.18%, utilizzando un message broker per la comunicazione intra-servizi invece di semplici chiamate HTTP.

- **Codebase separate** - Avere delle codebase separate vuol dire poter assegnare team diversi per ogni servizio, e la possibilità di usare tech stack differenti. Avere codebase più piccole inoltre ha il vantaggio di ridurre la complessità, velocizzando il processo di sviluppo.
- **Deploy Indipendente** - Al contrario di un sistema monolitico, i servizi possono essere deployati in maniera indipendente. Questo ha notevoli benefici: build time più rapido e possibilità di scalare i singoli servizi in base al carico, permettendo un migliore uso delle risorse.
- **Isolamento in caso di downtime** - Nel caso un servizio vada giù, che sia nel caso di crash o riavvii programmati per aggiornamenti, non va giù l'intero sistema ma solo il servizio stesso. Le comunicazioni vengono inoltre salvate all'interno della coda, in attesa di essere risolte quando il servizio torna online.

Di contro il sistema può risultare più difficile da debuggare nella sua totalità, vista la complessità aumentata e la quantità di componenti interconnessi.

2.2 Tecnologie utilizzate

In questa sezione verranno descritte le principali tecnologie e i servizi scelti per implementare il Credit Manager. Ognuna di queste scelte è stata presa tenendo conto di fattori quali le prestazioni, la scalabilità, la semplicità d'uso e la qualità della documentazione disponibile.

2.2.1 RabbitMQ

RabbitMQ¹ è un message broker, cioè, un software che consente la comunicazione asincrona tra programmi. Funziona attraverso **queues** (code) su cui i **producers** (mittenti) inviano messaggi definiti a priori, e i **consumers** (destinatari) consumano questi messaggi eseguendo operazioni in base al loro contenuto. Rabbit in particolare utilizza gli **exchange**, un punto intermedio tra i producers e le queues. L'exchange riceve i messaggi dei producers e li instrada confrontando la **routing key** del messaggio con determinate regole dette **bindings**.

Rabbit ha diversi punti di forza, alcuni intrinseci del tipo di software di cui fa parte, altri per via delle sue caratteristiche specifiche:

- **Scalabilità** - Il carico di lavoro viene distribuito automaticamente tra i consumers, e finché i messaggi non vengono processati questi rimangono in coda.
- **Flessibilità nel routing dei messaggi** - Rabbit offre molteplici modi di configurare gli exchange a seconda delle esigenze:

¹Questa sezione utilizza informazioni e immagini tratte dalla documentazione ufficiale di RabbitMQ[4]

1. **Direct Exchange** - Invio dei messaggi a code specifiche in base a una routing key.

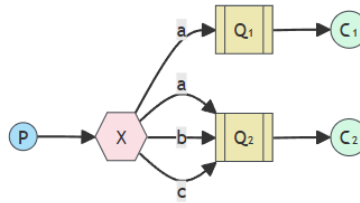


Figura 2.2. Esempio di Direct Exchange[4]

2. **Fanout Exchange** - Invio su tutte le code appartenenti all'exchange, utile per il broadcasting.
3. **Topic Exchange** - Permette di instradare i messaggi facendo il match delle routing key con binding basati su pattern più o meno complessi.

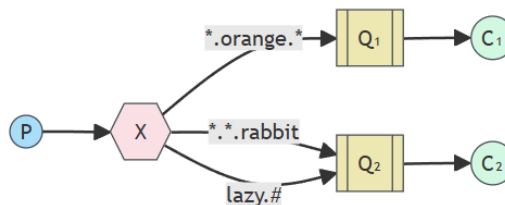


Figura 2.3. Esempio di Topic Exchange[4]

4. **Header Exchange** - Utile per instradare i messaggi in base a molteplici argomenti, per esempio è possibile creare una coda che riceve solo messaggi con header "format=pdf".
- **Affidabilità** - Rabbit è in grado di offrire un elevato grado di affidabilità grazie a molteplici meccanismi:
 1. **Acknowledgements** - Una conferma da parte del consumer che tutto sia andato a buon fine. Se il consumer non invia l'ack, Rabbit terrà il messaggio all'interno della coda o lo invierà a un altro consumer.
 2. **TTL (Time-to-Live) per messaggi e code** - È possibile impostare un limite di tempo che un messaggio può rimanere in una queue, dopo il quale vengono automaticamente scartati o vengono spostati su un Dead-letter Exchange, riducendo il rischio che si possano accumulare messaggi causando problemi di risorse esaurite o code sovraccariche.
 3. **DLX (Dead-letter Exchange)** - Sono normali exchange su cui vengono inviati i messaggi "dead-lettered": messaggi che hanno ricevuto un nack (**negative acknowledgement**), messaggi che hanno superato il TTL definito per messaggio, messaggi scartati perché la queue ha su-

perato uno dei limiti imposti, o messaggi che hanno superato il numero massimo di re-try consentito dalla coda.

4. **Persistenza dei messaggi** - È possibile persistere i messaggi presenti nelle queue sul disco, cioè non vengono persi al riavvio o al crash del server. La persistenza può essere attivata sia sulle code che sui singoli messaggi. Il contro della persistenza è che aumenta la latenza, ma offre ovviamente una maggiore affidabilità.

2.2.2 PostgreSQL

La scelta di PostgreSQL deriva dalla completezza delle funzionalità che offre, che lo rendono un'opzione desiderabile per applicazioni aziendali. Di seguito alcune delle caratteristiche che lo contraddistinguono:

- **Elevate prestazioni** - È un database relazionale molto performante in grado di supportare decine di migliaia di operazioni al secondo su hardware adeguato, e offre una latenza minore rispetto a DBMS concorrenti come **MySQL** in molte operazioni.[5]
- **Robustezza** - Postgres supporta molte funzioni di sicurezza, come autenticazione SSL, o configurazione dei permessi granulare fino al livello di colonne e righe. È inoltre completamente conforme ai requisiti **ACID** (Atomicity, Consistency, Integrity, Durability). Per assicurare queste proprietà, PostgreSQL utilizza metodologie come **Multi-Version Concurrency Control** (MVCC) e **Write-Ahead Logging** (WAL).
- **Scalabilità** - Ha ottime capacità di scalabilità verticale, che si traduce nell'utilizzo di tutti i core, di tutta la RAM o di maggiore velocità di lettura/scrittura dei dischi. Supporta inoltre il clustering e la replication, e ha quindi anche capacità di scalabilità orizzontale.

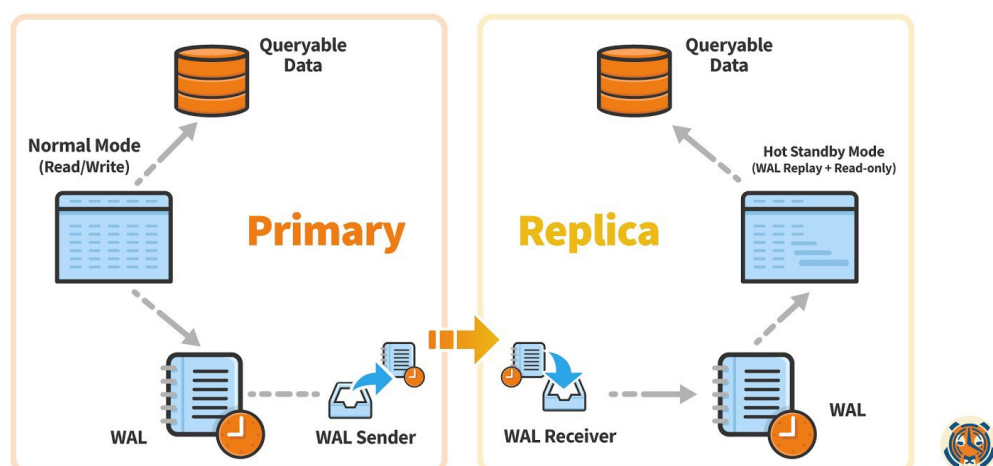


Figura 2.4. Esempio di replica[6]

- **Supporto per transazioni complesse** - Postgres supporta la scrittura di transazioni complesse e funzioni procedurali attraverso un linguaggio chiamato **PL/pgSQL**, che permette di creare logiche più avanzate.

2.2.3 Spring e SpringBoot

Spring² è un framework di Java creato per semplificare lo sviluppo di applicazioni a livello enterprise, fornendo una serie di strumenti per gestire i problemi più comuni durante lo sviluppo, permettendo agli sviluppatori di concentrarsi sulla business logic piuttosto che sui dettagli implementativi. Esempi di questi sono Spring Data o Spring Security.

Spring si basa sul concetto di **Inversion of Control** (IoC), dove gli oggetti invece di creare le loro dipendenze direttamente le ricevono da fonti esterne. Questo è implementato attraverso un pattern chiamato **Dependency Injection**, in cui gli oggetti definiscono le proprie dipendenze e lo **Spring IoC Container** gliele fornisce quando vengono creati. Lo Spring IoC Container si occupa dell'intero ciclo di vita di un oggetto, e nel contesto di Spring questi prendono il nome di **bean**.

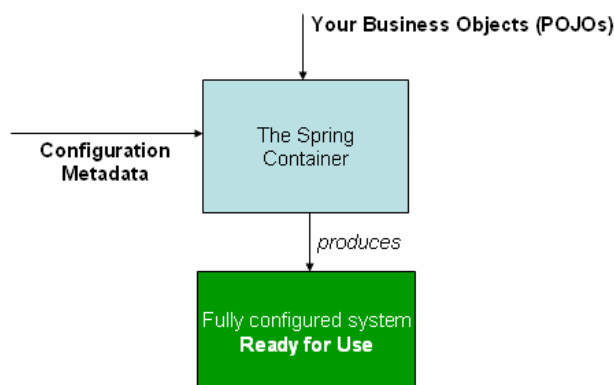


Figura 2.5. Spring IOC Container

Spring da solo è difficile da configurare e richiede molto boilerplate. Per cercare di ovviare a questo problema è stato creato il progetto **SpringBoot**. Lo scopo di SpringBoot è quello di semplificare il processo di configurazione di Spring, configurando automaticamente le dipendenze incluse in modo ragionevole. Per esempio, se si volesse utilizzare un web-server, basta includere `spring-boot-starter-web` tra le dipendenze e SpringBoot configurerà automaticamente un web-server come Tomcat, di default.

2.2.4 Angular

Angular è un framework JavaScript creato da Google e ideato per creare **Single-Page Web Applications** (SPA). È profondamente integrato con TypeScript, che

²Questa sezione utilizza informazioni e immagini tratte dalla documentazione ufficiale di Spring[7]

si traduce in una serie di vantaggi come maggiore Type Safety, codice più leggibile e mantenibile, oltre ad avere maggiori aiuti da IDE moderni come Visual Studio Code con l'autocompletamento.

Angular fornisce un pacchetto ben fornito di funzionalità anche senza l'utilizzo di librerie aggiuntive, tra cui ottimi strumenti di gestione del routing, gestione dei form, client HTTP, gestione dello stato, integrazione nativa con RxJS e altro.

Per il progetto Sentinel inoltre è stato scelto di usare il **SignalStore di NgRx**, una soluzione di gestione dello stato completamente basata sui **Signal** di Angular, introdotti in developer preview in Angular 16. Nei capitoli successivi verrà discussa più in dettaglio l'implementazione delle feature per la gestione di uno stato con chiamate a un server.

2.2.5 Stripe

Stripe è uno dei servizi più utilizzati nell'ambito dei pagamenti digitali, per la sua semplicità di utilizzo e per la sua capacità di adattarsi alle esigenze delle aziende di ogni dimensione. Ha un costo di €0,25 + 1,5% per ogni transazione con carte europee, senza costi ulteriori. L'utilizzo di Stripe rispetto ai competitor ha diversi vantaggi:

- **Facilità di integrazione** - Il processo di integrazione all'interno di un applicativo è semplice, e offre molte opzioni di configurazione e servizi. A seconda delle esigenze, è possibile passare da una configurazione predefinita con zero o comunque pochissimo codice a un'esperienza completamente personalizzata dallo sviluppatore.
- **Sicurezza** - Stripe semplifica l'oneroso lavoro di integrare un sistema di pagamenti che rispetti gli standard PCI DSS, essendo tutto gestito direttamente con redirect a Stripe o attraverso elementi integrati nell'applicazione ma hostati sui loro server. Non c'è quindi bisogno di trattare e memorizzare dati riguardanti i metodi di pagamento, evitando tutte le complicazioni che ne conseguono.
- **Supporto a molti metodi di pagamento** - Stripe supporta tutti i maggiori circuiti di pagamento, bonifici, o anche wallets digitali come Google Pay e Apple Pay.
- **Documentazione estensiva** - La documentazione è molto completa e dettagliata, inoltre è piena di esempi di integrazione in diversi linguaggi.

2.2.6 Redis

Redis è un sistema di cache "in-memory", conosciuto per le sue elevate performance e bassa latenza. È stato scelto per la sua facilità di integrazione grazie a SpringBoot Data Redis.

Tra i suoi punti di forza troviamo:

- **Aumento delle performance dell'applicativo** - Utilizzare Redis come sistema di cache distribuito permette di ridurre il carico sui vari componen-

ti, dato che se un'operazione richiesta è in cache il sistema può rispondere direttamente.

- **Scalabilità** - Redis supporta la replication secondo un modello master-replica. In caso di guasto dell'istanza master una delle repliche viene promossa. Questo si traduce in una maggiore resistenza ai guasti e capacità in lettura più elevata. Inoltre, grazie a Redis Cluster è possibile dividere i dati su più nodi, dividendo anche il carico in scrittura. Il contro di questa soluzione è che non garantisce una forte consistenza, in quanto Redis risponde prima di propagare la risposta su altri nodi per evitare penalità di latenza.

2.2.7 Docker

Docker³ è uno dei software più popolari in ambito aziendale e non, per la sua capacità di creare dei container dove vengono eseguite le applicazioni in maniera isolata. Docker si basa su alcuni concetti fondamentali:

- **Docker Images** - Sono dei template che vengono utilizzati per creare i container. Contengono tutte le configurazioni e gli eseguibili delle dipendenze richieste dall'applicazione containerizzata. Sono in genere molto leggere, visto che devono essere portabili. Le immagini possono essere condivise attraverso repository come Docker Hub.
- **Docker Containers** - L'ambiente dove l'applicativo containerizzato viene eseguito. Nel caso di un sistema Linux, al contrario di un normale software di virtualizzazione, i container di Docker vengono eseguiti sullo stesso kernel del sistema host. Questo significa che se un'applicazione containerizzata richiede una system call specifica per venire eseguita e la versione del kernel installata sull'host non la supporta, non è possibile eseguire quel container. Su Windows e macOS viene invece eseguito attraverso una VM Linux, in quanto è necessario un kernel Linux per utilizzare Docker.

³Questa sezione utilizza informazioni e immagini tratte dalla documentazione ufficiale di Docker[8]

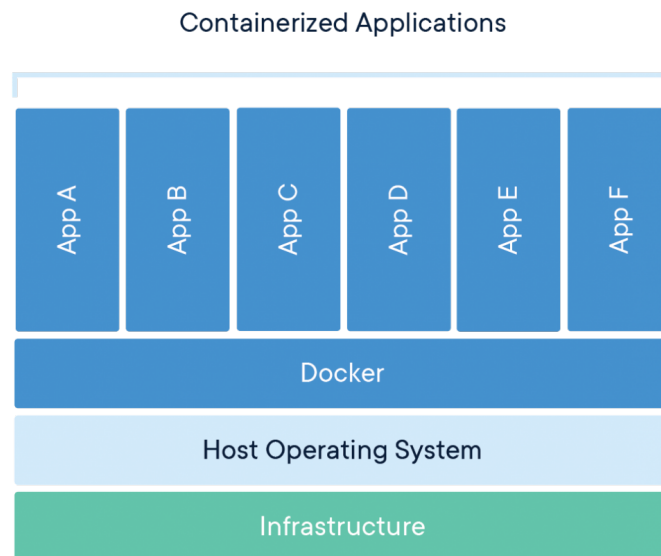


Figura 2.6. Schema sul funzionamento di Docker

- **Dockerfile** - Un file di testo che contiene tutte le istruzioni necessarie per creare un'immagine. Al suo interno viene specificata una **base image**, un'immagine che la build andrà a estendere. È possibile anche creare un'immagine da zero, ed è possibile utilizzare immagini diverse per buildare l'applicazione e per eseguirla (**Multi-staged build**). Per esempio, è comune utilizzare un'immagine minimale di Linux come *alpine* per buildare un sito, per poi prendere il risultato e usarlo con l'immagine base di *nginx*.
- **Docker Compose** - Permette di definire applicazioni multi-container, offrendo opzioni come la possibilità di specificare dipendenze tra container, variabili d'ambiente, mapping tra porte interne ed esterne al container, e creare dei network virtuali.

2.2.8 Kubernetes

Kubernetes⁴ è un **container orchestrator**, ovvero un software che si occupa di gestire automaticamente applicazioni containerizzate. Funziona attraverso un'architettura master-worker. Un **master node** si occupa di coordinare tutte le attività, mentre i **worker node** si occupano di eseguire i container.

L'unità più piccola creabile e deployabile è un **pod**, un gruppo di container strettamente correlati che condividono lo stesso namespace.

⁴Questa sezione utilizza informazioni tratte dalla documentazione ufficiale di Kubernetes[9]

Kubernetes gestisce i pod attraverso quattro tipi di controller:

- **Deployment** - Viene utilizzato nel caso sia necessario avere uno o più pod identici (**ReplicaSet**) che non devono mantenere uno stato persistente. Kubernetes si occupa di garantire che lo stato specificato nella configurazione venga rispettato, monitorando la salute dei pod e riavviandoli in caso di problemi. Un esempio di utilizzo è per hostare un web-server che restituisce contenuti statici.
- **StatefulSet** - Come un Deployment, gestiscono pod identici generati in base agli stessi container. La differenza è che l'identità di ogni pod è unica e lo storage, così come l'identificativo di rete, vengono mantenuti in caso di rescheduling. Può essere utilizzato per applicazioni come database, o sistemi di caching (come Redis).
- **DaemonSet** - Vengono utilizzati per applicazioni che devono essere eseguite su ogni nodo, sono quindi utili per task come raccogliere log, monitoraggio o sistemi di rilevamento di intrusione.
- **Job** - Crea uno o più pod che continuano a venire eseguiti finché un numero specifico di questi riesce ad arrivare a compimento. È possibile anche schedulare l'avvio di una task usando i CronJob.

Uno dei punti di forza più grandi di Kubernetes è la sua funzione di **Horizontal Pod Autoscaling**, che permette di regolare automaticamente il numero di pod avviati in base a metriche di utilizzo delle risorse, rendendo semplice scalare automaticamente l'applicazione in base al carico.

Capitolo 3

Progettazione concettuale e logica

In questo capitolo verranno analizzati i requisiti che deve rispettare il database, e verrà quindi definito lo schema Entity-Relationship. Nel caso ci sia la necessità di definire dei vincoli non esprimibili tramite lo schema ER, verranno inoltre definiti dei vincoli esterni.

3.1 Requisiti del database

L'applicazione prevede la presenza di un Wallet associato a ciascun tenant, che può essere ricaricato tramite pagamenti effettuati con **Stripe**. All'interno del sistema sono previsti dei **workflow**, processi che utilizzano dati provenienti da banche dati esterne per effettuare analisi di vario tipo. Ogni operazione all'interno del workflow ha un costo, e per ognuna di queste viene scalato un determinato importo in base a un listino prezzi associato al tenant di appartenenza dell'utente. Quando un'operazione viene inserita nella coda pagamenti, questa viene processata dal Credit Manager e registrata nel database, e il relativo importo viene scalato automaticamente dal Wallet del tenant. Deve inoltre essere possibile annullare un'operazione. I listini sono costituiti da una descrizione e una fascia di prezzo, e specificano un importo per ogni tipo di operazione.

3.2 Progettazione concettuale

3.2.1 Schema Entity-Relationship

Uno schema ad alto livello orientato puramente alla definizione dei dati e delle relazioni tra loro, senza fare considerazioni su performance o altro. Non sono presenti neanche le chiavi primarie normalmente richieste in un vero database.

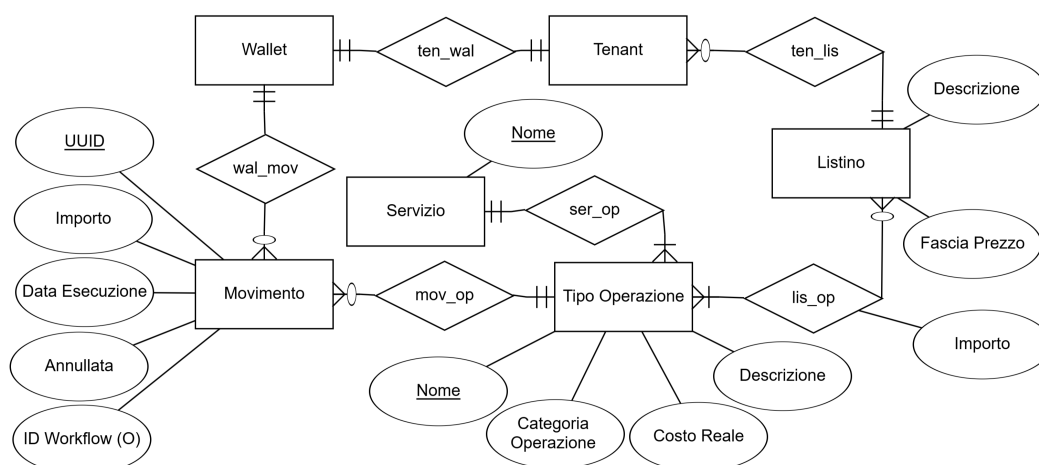


Figura 3.1. Schema Entity-Relationship del database

Entità Movimento

Questa entità rappresenta un singolo movimento. Ogni istanza deve avere una relationship Molti a Uno con Wallet (wal_mov) e una relationship Molti a Uno con Tipo Operazione (mov_op). Può avere un ID Workflow, se l'operazione è stata fatta partire da un Workflow.

Tabella 3.1. Descrizione degli attributi dell'entità Movimento

Attributo	Dominio	Descrizione
UUID	UUID	UUID dell'operazione all'interno del workflow
Importo	Numeric	Importo pagato dall'utente
Data Esecuzione	DataOra	Data e Ora di esecuzione del movimento
ID Workflow	integer	Identificatore del Workflow di cui fa parte il movimento
Annullata	booleana	Indica che l'operazione è stata annullata e ha un'operazione contraria che la bilancia

Entità Servizio

Rappresenta uno dei servizi che Sentinel può interrogare, o Sentinel stesso. Ha una relationship Uno a Molti con Tipo Operazione (ser_op).

Tabella 3.2. Descrizione degli attributi dell'entità Servizio

Attributo	Dominio	Descrizione
Nome	Stringa	Nome del Servizio

Entità Tipo Operazione

Entità che rappresenta il Tipo Operazione, cioè le operazioni eseguibili che comportano un addebito o un accredito. Ha una relationship Molti a Uno con Servizio (ser_op) che rappresenta il Servizio su cui viene svolta l'operazione, o Sentinel stesso per ricariche o annullamenti. Inoltre ha una relationship Molti a Molti con Listino (lis_op).

Tabella 3.3. Descrizione degli attributi dell'entità Tipo Operazione

Attributo	Dominio	Descrizione
Nome	Stringa	Nome del Tipo Operazione
Categoria Operazione	{ADDEBITO, ACCREDITO}	Tipologia di operazione. Utilizziamo un enum type per rappresentare questo attributo.
Costo Reale	numeric	Costo reale affrontato interrogando la banca dati esterna
Descrizione	Stringa	Breve descrizione di cosa fa l'operazione

Entità Listino

In questa entità vengono rappresentati i Listini creati dall'amministratore. Un Listino può avere una relationship Uno a Molti con Tenant (ten_lis), e deve avere delle relationship Molti a Molti (lis_op) con Tipo Operazione.

Tabella 3.4. Descrizione degli attributi dell'entità Listino

Attributo	Dominio	Descrizione
Descrizione	Stringa	Descrizione del Listino
Fascia Prezzo	{ALTA, MEDIA, BASSA}	Campo descrittivo per indicare la fascia di prezzo del listino

Relationship lis_op

Relationship che rappresenta l'importo che deve venire scalato dal Wallet di un Tenant quando viene registrato un movimento. Uno dei requisiti è che ogni Listino deve specificare un importo per ogni Tipo Operazione. Non essendo possibile forzare la presenza di una relationship tra ogni istanza di Listino e ogni istanza di Tipo Operazione attraverso lo schema ER, verrà disposto un vincolo esterno.

Tabella 3.5. Descrizione degli attributi della relationship lis_op

Attributo	Dominio	Descrizione
Importo	numeric	Importo che verrà scalato al tenant associato al listino prezzi

3.2.2 Vincoli Esterni

Dalla costruzione dello schema concettuale è emersa la necessità di rappresentare il seguente vincolo esterno:

Tutti gli importi specificati

$$\forall l, \forall t (Listino(l) \wedge TipoOperazione(t) \implies lis_op(l, t))$$

Esiste una relationship *lis_op* tra ogni entità *Listino* e ogni entità *Tipo Operazione*.

3.3 Casi d'uso

In questa sezione verrà fornito lo schema UML dei casi d'uso, che indica le funzionalità dell'applicazione e gli attori che le useranno. Prenderemo inoltre in esame un caso d'uso rappresentativo all'interno dell'applicativo e rappresenteremo le sue specifiche.

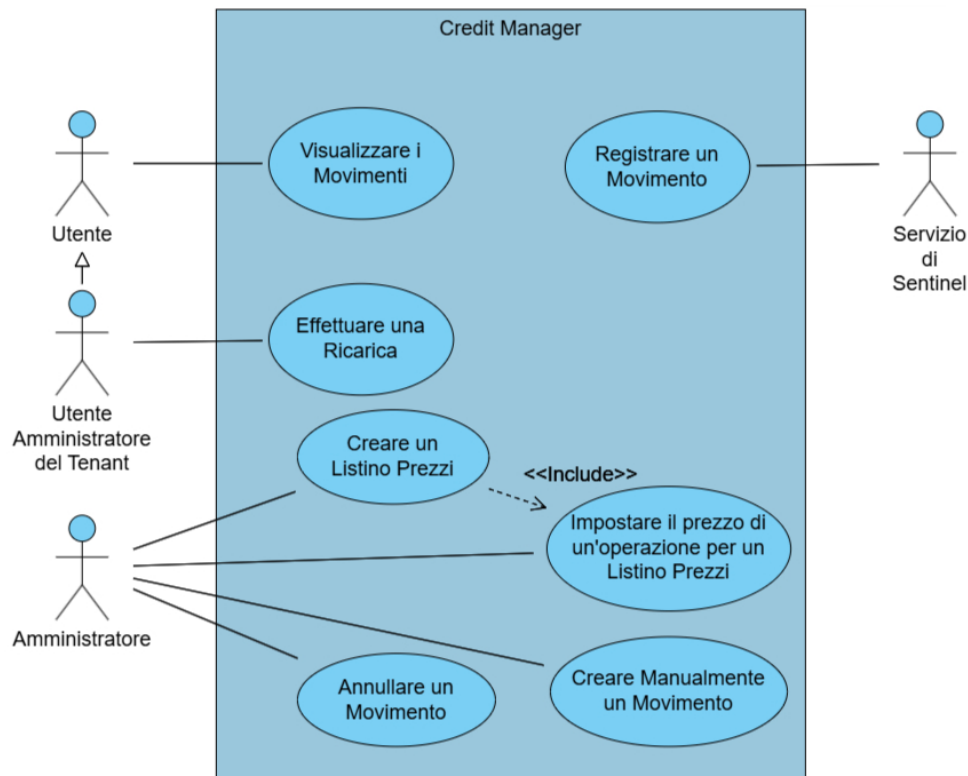


Figura 3.2. Schema UML dei casi d'uso

Specifica dei casi d'uso di Crea Listino

Specifichiamo come dovrà funzionare l'operazione per creare un listino prezzi:

crea_listino(descrizione: Stringa, fascia_prezzo: {ALTA, MEDIA, BASSA}, tipi_operazione: [TipoOperazione], prezzi: [Numeric]) \Rightarrow Listino:

pre:

Verifica le seguenti condizioni:

- $descrizione \neq ""$
- $tipi_operazione.length == prezzi.length$
- $\forall tipoOperazione \in TipoOperazione \Rightarrow \exists op \in tipi_operazione \wedge op == tipoOperazione$

post:

1. Crea un nuovo Listino con la descrizione e fascia prezzo.
2. Per ogni *tipo* in *tipi_operazione* crea una nuova relationship PrezzoListino tra il nuovo listino e *tipo*, con importo uguale all'elemento di *prezzi* con lo stesso indice.
3. Restituisci il nuovo Listino

Verrà fornita un'implementazione nella **Sezione 6.1**.

Capitolo 4

Ristrutturazione e traduzione dello schema

In questo capitolo procederemo alla preparazione dello schema per poi tradurlo direttamente in uno schema relazionale per PostgreSQL, specificando anche delle soluzioni per rispettare i vincoli esterni definiti.

4.1 Ristrutturazione dello schema Entity-Relationship

Procediamo alla ristrutturazione dello schema ER prima di tradurlo direttamente nello schema relazionale:

- Sostituiamo le relationship Molti a Molti, tipicamente introducendo una nuova tabella
- Sostituiamo eventuali relazioni is-a.
- Selezioniamo o introduciamo le chiavi primarie.
- Aggiungiamo in modo controllato della ridondanza, se necessario per questioni di performance (Es. un campo precalcolato in caso di accessi frequenti a un dato derivato).
- Introduciamo ulteriori vincoli esterni in seguito alla ristrutturazione, se lo schema ristrutturato non rispetta più i vincoli dello schema originale.

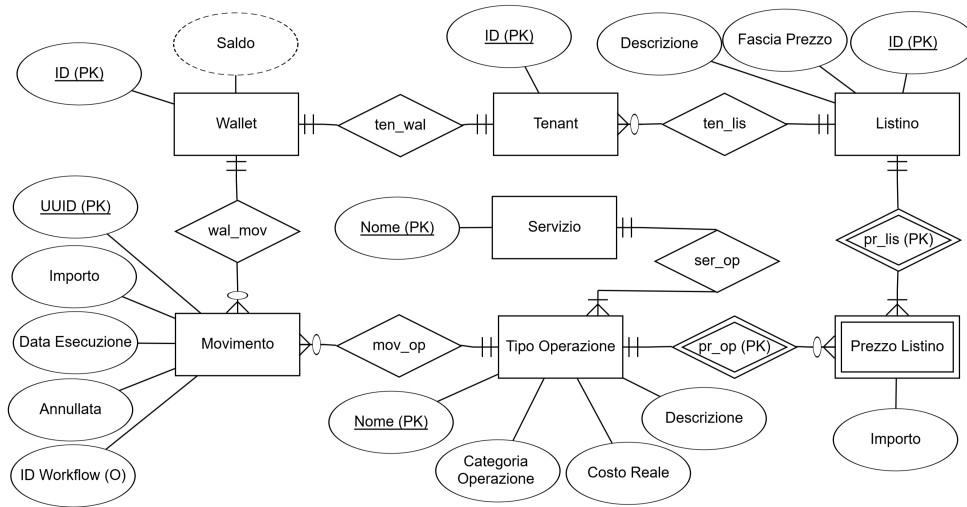


Figura 4.1. Schema Entity-Relationship tradotto

4.2 Traduzione dello schema ristrutturato

Traduciamo lo schema ristrutturato nello schema relazionale da inserire in PostgreSQL, con chiavi primarie e chiavi esterne.

Tabella Movimento

Come chiave primaria scegliamo uuid, campo univoco già presente all'interno dello schema concettuale. Inseriamo inoltre le chiavi esterne di Wallet e TipoOperazione, necessarie per rappresentare le relationship wal_mov e mov_op.

Movimento(uuid, importo, dataEsecuzione, descrizione, idWorkflow*, tipoOperazioneId, walletId)

foreign key: $Movimento[tipoOperazioneId] \subseteq TipoOperazione[nome]$

foreign key: $Movimento[walletId] \subseteq Wallet[id]$

Tabella TipoOperazione

Come chiave primaria scegliamo nome, campo univoco già presente. Inseriamo inoltre la chiave esterna di Servizio per rappresentare la relationship ser_op.

TipoOperazione(nome, servizio, categoriaOperazione, costoReale, descrizione)

foreign key: $TipoOperazione[servizio] \subseteq Servizio[nome]$

Tabella Servizio

Come chiave primaria scegliamo nome, campo univoco già presente.

Servizio(nome)

Tabella Wallet

Dato che non è presente nessun campo univoco utilizzabile come chiave primaria, inseriamo un campo intero generato da una sequenza. Inseriamo inoltre la chiave esterna di Tenant per rappresentare la relationship `ten_wal`. È stato deciso di disporre un campo calcolato attraverso una task disposta sul servizio, che ogni ora prende tutte le nuove operazioni e le utilizza per calcolare il nuovo saldo. Per avere comunque un saldo sempre aggiornato nelle operazioni importanti, viene salvata anche la data dell'operazione più recente. Questa viene utilizzata per recuperare tutti i movimenti successivi e calcolarne il parziale, per poi aggiungerlo al saldo salvato nella tabella.

Wallet(id, saldo, dataUltimaOperazione, tenantId)

key: (*tenantId*)

foreign key: *Wallet*[tenantId] \subseteq *Tenant*[id]

Tabella Tenant

Dato che non è presente nessun campo univoco utilizzabile come chiave primaria, inseriamo un campo intero generato da una sequenza. Inseriamo inoltre la chiave esterna di Listino per rappresentare la relationship `ten_lis`

Tenant(id, listinoId)

foreign key: *Tenant*[listinoId] \subseteq *Listino*[id]

Tabella Listino

Dato che non è presente nessun campo univoco utilizzabile come chiave primaria, inseriamo un campo intero generato da una sequenza.

Listino(id, descrizione, fasciaPrezzo)

Tabella PrezzoListino

Per rappresentare la relationship Molti a Molti `lis_op`, inseriamo un'altra tabella che ha come chiave primaria la combinazione delle chiavi esterne di Listino e TipoOperazione.

PrezzoListino(tipoOperazioneId, listinoId, importo)

foreign key: *PrezzoListino*[tipoOperazioneId] \subseteq *TipoOperazione*[nome]

foreign key: *PrezzoListino*[listinoId] \subseteq *Listino*[id]

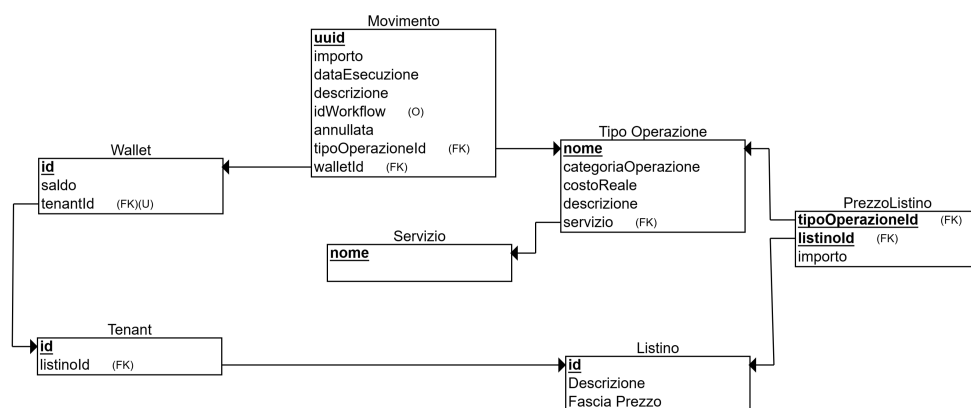


Figura 4.2. Rappresentazione grafica del database

4.3 Traduzione dei vincoli esterni

Visto che non sono stati aggiunti ulteriori vincoli esterni durante la ristrutturazione, resta da gestire l'unico vincolo riconosciuto durante la creazione dello schema concettuale:

Tutti gli importi specificati (3.2.2) - Si è deciso di gestire il vincolo attraverso il form di creazione, come descritto nella **Sezione 6.1**. Non è stato utilizzato un trigger di PostgreSQL per permettere l'inserimento di un Tipo Operazione senza dover necessariamente aggiornare tutti i listini prezzi, che useranno una maggiorazione predefinita rispetto al Costo Reale finché non verrà specificato un importo.

4.4 Tabelle pre-popolate

Ci sono due tabelle di cui conosciamo il contenuto a priori, e che non è possibile modificare da interfaccia: TipoOperazione e Servizio. Pertanto, inseriremo dei dati per popolarle durante l'inizializzazione del database.

Capitolo 5

Design del frontend

In questo capitolo ci soffermeremo sul design delle interfacce del frontend, utilizzando come riferimento dei wireframe creati con il web software **Balsamiq**. L'interfaccia è suddivisa in due schermate principali, una per la gestione del Wallet e una per la gestione dei listini prezzi.

5.1 Sezione per la gestione del Wallet

Per monitorare il Wallet del suo tenant di appartenenza, l'utente ha a disposizione una dashboard in cui può visualizzare lo storico delle transazioni ed effettuare una ricarica del credito.

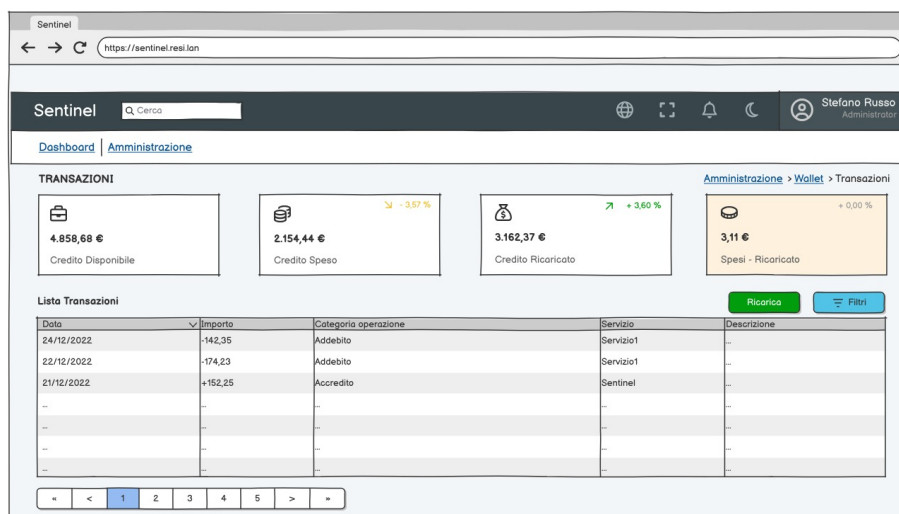


Figura 5.1. Wireframe della schermata di Gestione Wallet

Nella parte superiore sono presenti dei widget che riportano alcune informazioni in modo da essere facilmente accessibili:

1. Credito disponibile
2. Credito speso nel mese corrente
3. Credito depositato nel mese corrente
4. Differenza tra credito speso e credito depositato nel mese corrente

Gli ultimi tre grafici riportano anche l'andamento rispetto al mese precedente.

Il bottone **Ricarica** apre una schermata che permette di caricare un importo arbitrario, con la possibilità di scegliere dei tagli prestabiliti.

Selezione Importo

0 €

Procedi al pagamento

Detailed description: A wireframe of a mobile app screen titled 'Selezione Importo'. It features a text input field containing '0 €' with a dropdown arrow on the right. Below the input field is a large green button with the text 'Procedi al pagamento' in white.

Figura 5.2. Wireframe della schermata di selezione importo

Procedendo al pagamento si conferma l'importo, passando quindi alla schermata di pagamento di Stripe.

Paga con Stripe

Ricarica 5 €

Email

Dati della carta

1234 1234 1234 1234

MM / AA CVC

Titolare

Nome e cognome

Paga

Detailed description: A wireframe of a mobile app screen titled 'Paga con Stripe'. It shows a form for a 5 € recharge. The form includes an 'Email' field, a 'Dati della carta' section with a card number field (displaying '1234 1234 1234 1234'), two fields for 'MM / AA' and 'CVC', and a 'Titolare' section with a 'Nome e cognome' field. At the bottom is a large green button labeled 'Paga'.

Figura 5.3. Wireframe della schermata di pagamento con Stripe

Il bottone **Filtri** apre una schermata laterale dove è possibile selezionare le proprietà per cui deve essere filtrata la lista delle transazioni.

SERVIZIO

PERIODO

7 Nov - 8 Nov

CATEGORIA OPERAZIONE

Seleziona...

SERVIZIO

Seleziona...

IMPORTO

10 €

200 €

Reset

Applica

Figura 5.4. Wireframe della schermata dei filtri






5.2 Sezione per la gestione dei listini prezzi

L'amministratore ha a disposizione una schermata in cui può aggiungere, modificare o cancellare un listino prezzi, come indicato nella **Figura 5.5**.

Sentinel

← → ↻ https://sentinel.resi.lan

Sentinel

     Stefano Russo
Amministratore

Dashboard | Amministrazione

LISTINI PREZZI [Amministrazione](#) > [Wallet](#) > [Listini Prezzi](#)

+ Aggiungi Listino



















Descrizione	Fascia Prezzo	Azioni
...	Alto	  
...	Media	  
...	Basso	  
...	...	  
...	...	  
...	...	  

Figura 5.5. Wireframe della schermata con la lista dei listini prezzi

Il bottone **Aggiungi Listino** reindirizza a una pagina che consente di inserire i dati per creare un nuovo listino prezzi, come nella **Figura 5.7**.

Nella tabella invece sono presenti 3 azioni per ogni listino prezzi:

1. Apertura del dettaglio come indicato nella **Figura 5.6**.
2. Reindirizzamento a una pagina analoga a quella della **Figura 5.7**, che permette di modificare il listino prezzi.
3. Apertura di una schermata di conferma per l'eliminazione del listino.

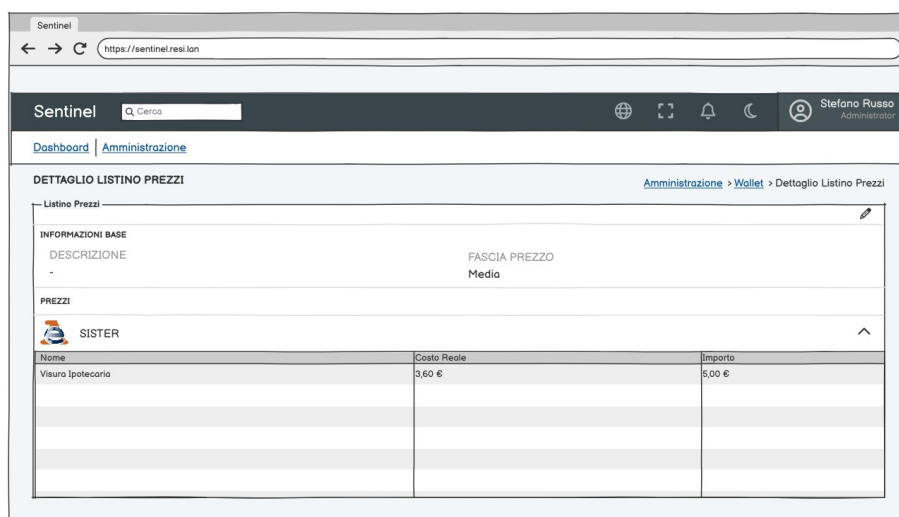


Figura 5.6. Wireframe della schermata del dettaglio di un listino prezzi

Il dettaglio del listino prezzi è suddiviso in due sezioni: nella sezione **Informazioni Base** sono presenti la descrizione e la fascia di prezzo, mentre nella sezione **Prezzi** è presente una tabella nascondibile per ogni banca dati, con il relativo costo di ogni operazione all'interno del listino prezzi. Con il bottone di edit (l'icona a forma di matita) si viene reindirizzati a una pagina che permette di modificare il listino, analoga alla schermata presente nella **Figura 5.7**.

The wireframe shows a web application interface for 'Sentinel'. The main heading is 'AGGIUNGI LISTINO PREZZI'. Below it, there are two sections: 'Informazioni Base' and 'Prezzi'.

Informazioni Base

DESCRIZIONE*

FASCIA PREZZO*

Media

Prezzi

SISTER

Nome	Costo Reale	Importo
Visura Ipotecaria	3,60 €	5,00 €

Buttons: X Reset, Salva, + Aggiungi Prezzo

Figura 5.7. Wireframe della schermata di creazione listino prezzi

Nella sezione **Informazioni Base** è possibile inserire una descrizione e una fascia di prezzo. Nella sezione **Prezzi** è presente una tabella nascondibile per ogni banca dati che riporta i costi inseriti. Cliccare su una riga apre una schermata analoga a quella indicata nella **Figura 5.8**, con la differenza che la selezione del tipo operazione non è presente, in quanto il campo è preselezionato. Questa schermata consente quindi di modificare il prezzo inserito in precedenza. Per terminare la creazione di un Listino è necessario inserire un importo per tutte le operazioni di pagamento.

Il bottone **Aggiungi Prezzo** apre la schermata di aggiunta del prezzo in **Figura 5.8**

The wireframe shows a modal window titled 'AGGIUNGI PREZZO'.

SERVIZIO
SISTER

COSTO REALE
6,30 €

DESCRIZIONE
-

TIPO OPERAZIONE*

Visura Ipotecaria Immobile

IMPORTO*

7,00 €

Buttons: Chiudi, Salva

Figura 5.8. Wireframe della schermata per inserire un importo

Da questa schermata è possibile selezionare un tipo operazione e specificare il suo importo, potendolo sempre confrontare con il costo reale di interrogazione della banca dati esterna.

Capitolo 6

Dettagli implementativi

In questo capitolo verranno illustrati i dettagli implementativi delle parti più significative del sistema, con particolare attenzione a come strumenti e tecnologie specifiche sono stati utilizzati per soddisfare i requisiti del progetto. Verranno esplorati in dettaglio il ruolo del Credit Manager all'interno di Sentinel focalizzandoci sull'implementazione di RabbitMQ, l'integrazione con Stripe per le funzionalità di ricarica del wallet associato al tenant, e l'implementazione delle feature del SignalStore di NgRx per gestire in modo efficiente uno stato con chiamate al backend.

6.1 Implementazione dello use-case Crea Listino

In questa sezione verrà descritta l'implementazione dello use-case Crea Listino analizzato durante la progettazione concettuale, fornendo anche immagini e snippet di codice.

Per creare un listino prezzi, l'amministratore deve compilare il form in **Figura 6.1**:

The screenshot shows a web application interface for creating a price list. The header includes a search bar, navigation links (Dashboard, Amministrazione, Anagrafica, Gestione Crediti, Indagine), and user information (Stefano Russo, Administrator). The main content area is titled 'AGGIUNGI LISTINO PREZZI' and contains a form with two sections: 'INFORMAZIONI BASE' and 'PREZZI'. The 'INFORMAZIONI BASE' section has a 'DESCRIZIONE' text input and a 'FASCIA PREZZO' dropdown menu set to 'Media'. The 'PREZZI' section is empty, showing a large blue plus icon and the text 'Aggiungi i prezzi'. At the top right of the form are 'Reset' and 'Salva' buttons. The breadcrumb trail at the top reads 'Amministrazione > Wallet > Aggiungi Listino Prezzi'.

Figura 6.1. Pagina di creazione di un Listino Prezzi.

Per aggiungere un prezzo è possibile cliccare sul relativo bottone, che aprirà la schermata in **Figura 6.2**:

Figura 6.2. Modale per inserire un prezzo.

Dopo aver compilato i campi obbligatori del form e aver inserito un prezzo per tutte le operazioni, viene sbloccato il tasto Salva, che invia una richiesta a questo endpoint del Credit Manager:

PUT - /listino

```

1  @PostMapping("listino")
2  //Controllo dei permessi generalizzato da un'annotazione.
3  @RfguiAuthorization(permissionId = CreditManagerConstants.
    WALLET_LISTINO_PREZZI_MENU_PERMISSION, writePerm = true)
4  public ResponseEntity<GenericResponse> createListino(
    HttpServletRequest req, @RequestBody CreateListinoForm form) {
5      GenericResponse response = new GenericResponse();
6      HttpStatus status = HttpStatus.OK;
7      //Controlla che siano presenti tutti i campi obbligatori
8      if (CreateListinoForm.validate(form)) {
9          List<TipoOperazione> tipoOperazioneList =
            tipoOperazioneService.findAllExceptSentinel();
10         //Controlla se ci sono tutte le operazioni. In caso negativo
            lancia errore, annullando la transaction
11         if (!new HashSet<>(form.getPrezziListino().stream().map(
            PrezzoListinoForm::getTipoOperazioneId).toList()).
            containsAll(
12             tipoOperazioneList.stream().map(TipoOperazione::getId)
                .toList()))

```

```

13         throw new InvalidFormException(HttpStatus.BAD_REQUEST,
14             CreditManagerConstants.ERROR_INVALID_FORM);
15     //Crea il listino
16     Listino listino = listinoService.addListino(form.
17         getDescrizione(), form.getFasciaPrezzo());
18     //Crea un nuovo Prezzo Listino per ogni importo
19     form.getPrezziListino().forEach(prezzoListino -> {
20         listinoService.saveImportoForListino(prezzoListino.
21             getTipoOperazioneId(), listino, Money.of(prezzoListino
22                 .getImporto(), "EUR"));
23     });
24     //Utilizza MapStruct per convertire l'entità
25     response.setEntity(listinoMapper.toDTO(listino));
26 } else {
27     throw new InvalidFormException(HttpStatus.BAD_REQUEST,
28         CreditManagerConstants.ERROR_INVALID_FORM);
29 }
30 //Se tutto è andato a buon fine, risponde al frontend col nuovo
31 listino.
32 //Termina inoltre la transaction, salvando sul database i nuovi
33 dati.
34 return ResponseEntity.status(status).body(response);
35 }

```

Body della richiesta

L'endpoint accetta come body oggetti definiti dalla seguente classe:

```

1 @Getter
2 @Setter
3 public class CreateListinoForm {
4     private String descrizione;
5     private FasciaPrezzoEnum fasciaPrezzo;
6     private List<PrezzoListinoForm> prezziListino;
7
8     public static boolean validate(CreateListinoForm form) {
9         return form.getFasciaPrezzo() != null && form.
10             getPrezziListino() != null;
11     }
12 }

```


Come si può vedere dagli snippet di codice, anche il backend effettua un controllo per la presenza di un importo per ogni tipo operazione, garantendo che il vincolo esterno rimasto dal passo di traduzione (4.3) venga rispettato.

6.2 RabbitMQ

All'interno dell'architettura event-driven di Sentinel, il Credit Manager ha il ruolo di consumer. Sono presenti tre Topic Exchange:

- **work-exchange** - L'exchange dove vengono inviati i messaggi normalmente.
- **retry-exchange** - L'exchange dove vengono inviati i messaggi che hanno superato il numero massimo di retry nel work-exchange. Su questi messaggi viene applicato un exponential backoff, e se superano il numero massimo di retry vengono inviati nell'error-exchange. I messaggi in questo exchange hanno un TTL di 12000 (12 secondi).
- **error-exchange** - L'exchange dove vengono inviati i messaggi che hanno restituito un errore irrecuperabile o che hanno superato il numero massimo di retry.

Il Credit Manager sta in ascolto sulla coda con topic “#.event.payment.#” del work-exchange, e riceve messaggi con il seguente formato:

```
1 public class PaymentMessage {  
2     private UUID idempotencyKey;  
3     private PaymentOperationType operationType;  
4     private Money costFeedback;  
5     private long workflowId;  
6 }
```

Di seguito la spiegazione della funzione di ogni campo:

- **idempotencyKey** - UUID che verrà utilizzato come chiave primaria del nuovo movimento. Permette di evitare l'inserimento di operazioni duplicate.
- **paymentOperationType** - Contiene il Tipo Operazione, equivalente a quello salvato nel database. Viene utilizzato per recuperare il costo da addebitare al tenant dal listino prezzi associato.
- **costFeedback** - È il costo effettivo affrontato durante l'interrogazione alla banca dati esterna. Viene registrato in caso sia differente dal costo reale associato al TipoOperazione.
- **workflowId** - È l'id del workflow, e viene utilizzato da alcuni endpoint per restituire statistiche come il totale speso in un determinato workflow.

Quando il Credit Manager riceve un messaggio sulla sua coda, registra il movimento nella tabella Movimento del database.

6.3 Stripe

Per la gestione dei pagamenti è stato deciso di utilizzare il Checkout di Stripe con pagina self-hosted, in modo da integrarla all'interno dell'applicativo. Un pagamento segue un flusso ben preciso:

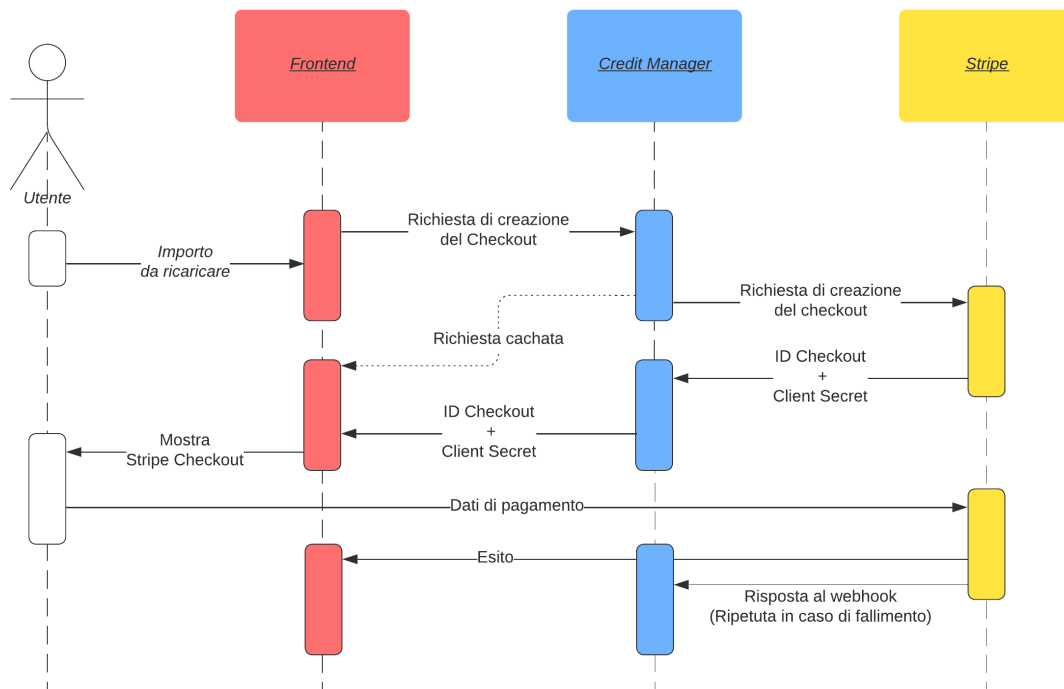


Figura 6.3. Diagramma di sequenza di un pagamento con Stripe

1. L'utente inserisce un importo maggiore di 5 euro da ricaricare e conferma.
2. Il frontend invia l'importo con una chiave di idempotenza generata tramite UUID.
3. Il backend verifica che la richiesta non sia già stata eseguita confrontandolo con la cache salvata su Redis, e nel caso invia al frontend la risposta precedente. In caso negativo invia a Stripe una richiesta di creazione del checkout con la stessa chiave di idempotenza inviata dal frontend.
4. Stripe risponde al backend con l'id del Checkout e un Client Secret necessario per la creazione della pagina di pagamento.
5. Il backend risponde al frontend con le informazioni inviate da Stripe.
6. Il frontend mostra il Checkout all'utente, utilizzando i dati inviati da Stripe.
7. L'utente inserisce i dati di pagamento e li invia attraverso l'elemento hostato da Stripe.

8. Stripe invia l'esito sia al backend che al frontend. Nel caso del backend, l'esito viene inviato a un webhook che si occupa di registrare correttamente l'avvenuto pagamento. In caso il backend non restituisca il codice HTTP 200 OK, Stripe riproverà fino a tre giorni seguendo una strategia di Exponential Backoff.

Il webhook ha il compito di verificare che i messaggi siano realmente provenienti da Stripe, confrontando la signature ricevuta nell'header "Stripe-Signature" con la chiave ricevuta durante la configurazione del webhook.

6.4 SignalStore di NgRx

Nel corso del tirocinio è stata presa la decisione di ricostruire il frontend a partire da una base diversa, utilizzando l'ultima versione di Angular (Angular 17). A tale scopo è stato deciso di rivedere il pattern di comunicazione tra componenti, sostituendo **NgRx** con il **SignalStore** di NgRx. Il primo è un progetto che porta il pattern Redux su Angular, mentre il secondo è una soluzione basata interamente sui Signal, feature di Angular uscita dalla Developer Preview proprio con la versione 17.

È stato quindi necessario implementare una soluzione flessibile che utilizzi al meglio il SignalStore, utilizzando quelle che vengono chiamate **feature**. Viene fatto uso di Generics, Mapped Types, Function Overloading, e viene in generale sfruttata la capacità di inferenza sui tipi di TypeScript.

Le feature implementate offrono la possibilità di effettuare operazioni **CRUD** (Create, Read, Update, Delete), rimuovendo la necessità di scrivere molto boilerplate e offrendo funzionalità aggiuntive ed estendibili. Ne sono state implementate tre:

- **withItem**: Si occupa di effettuare operazioni CRUD su un singolo oggetto appartenente a una collezione.
- **withList**: Si occupa di effettuare operazioni CRUD su una collezione.
- **withGrid**: Si occupa di gestire lo stato necessario per un componente generico che mostra una Tabella. Permette inoltre di effettuare operazioni CRUD sulle righe della tabella stessa.

Prenderemo in esame withGrid, la più complessa.

```
export const PortafoglioListStore = signalStore(  
  withGrid({  
    collection: 'portafoglio',  
    service: portafoglioGridService,  
    idField: 'id',  
    defaultSorting: { dir: 'desc', field: 'dataAcquisizione' },  
    paginationType: 'blocks',  
    usedInputs: ['form'],  
  }),  
);
```

Figura 6.4. Codice per creare uno Store con withGrid

Questo è il codice richiesto per creare uno Store che include withGrid. La feature prende in input diversi parametri:

- **collection** - La stringa su cui verranno basati i nomi delle funzioni CRUD e dello stato.
- **service** - Oggetto che contiene i metodi che comunicano con il backend. Restituiscono l'**Observable**¹ a cui lo store si sottoscriverà per effettuare la chiamata. È stato disposto un tipo generico che prende in input i tipi per definire i parametri necessari per le chiamate, tra cui il tipo dell'oggetto che lo Store deve gestire, o i routeParams che deve utilizzare per generare l'URL. Questo tipo generico consente a IDE moderni come Visual Studio Code di generare tutte le funzioni che poi verranno chiamate dalla feature.
- **paginationType** - Il tipo di paginazione utilizzato. Ha tre possibili valori: **none**, **blocks**, **pages**. Questo perché lo store gestisce anche la paginazione per il componente generico, e si deve comportare in maniera diversa a seconda del tipo. Per blocks, la paginazione e il sorting vengono gestiti dal backend, che prende in input anche la dimensione e il numero del blocco. Nel caso di pages invece, la paginazione è interamente gestita dalla feature e dal componente generico. Se il valore di paginationType è none viene disabilitata la paginazione.
- **idField** - Nome del campo ID dell'oggetto che la grid sta trattando. Il valore di questo campo deve essere necessariamente univoco negli oggetti inseriti, in quanto necessario per il corretto funzionamento della feature e del componente generico (per esempio, viene utilizzato nel track del @for di Angular).
- **defaultSorting** - Valore iniziale del campo per cui vengono ordinati gli oggetti all'interno della grid. Nella paginazione a blocchi questo valore viene inviato anche al backend, altrimenti viene utilizzato localmente.

¹Un Observable è la base della reattività fornita da RxJs: è una sorgente di dati che vengono emessi nel tempo. Il flusso può essere manipolato appendendo degli operatori che modificano il dato ad ogni passo. I più comuni sono map e filter. Angular li utilizza per le chiamate HTTP, e in questo caso emettono un valore una sola volta per subscription.

- **usedInputs** - Valore utilizzato internamente dalla feature per abilitare o disabilitare determinati input. Il suo valore viene forzato attraverso il typing system di TypeScript utilizzando il tipo di Service.

La feature mette a disposizione diverse funzioni e valori: tutte le tipiche funzioni CRUD, una funzione per effettuare il reload della grid con gli ultimi input passati, e lo stato dell'ultima chiamata effettuata (Loading, Loaded, Error, o Init se non è stata effettuata nessuna chiamata dalla creazione dello store).

Prendiamo ora come esempio il codice necessario per definire un service per with-Grid:

```
const portafoglioGridService: GridService<Portafoglio, 'id', 'blocks', undefined,
PortafoglioSearch, undefined, PortafoglioCreate, Portafoglio> = {
  loadGrid(
    http: ApiService,
    input: { form?: PortafoglioSearch; pagination: Pagination; sort: { field:
keyof Portafoglio; dir: 'desc' | 'asc' } },
  ): Observable<{ items: Portafoglio[]; total: number }> {
    const params: HttpParams = createHttpParamsComplete(input);
    return http.post(apiPortafogliSearch, input.form, { params });
  },

  addToGrid(http: ApiService, input: { form: PortafoglioCreate }): Observable<
Portafoglio> {
    return http.post(apiPortafogli, input.form);
  },

  deleteFromGrid(http: ApiService, id: number): Observable<Portafoglio> {
    return http.delete(apiPortafogliWithId(id));
  },

  updateItemInGrid(http: ApiService, id: number, input: { form: Portafoglio }):
Observable<Portafoglio> {
    return http.put(apiPortafogliWithId(id), input.form);
  },
};
```

Figura 6.5. Codice del servizio con le chiamate al backend

Come mostrato dal codice in figura, è necessario passare diverse informazioni al tipo GridService per permettere a TypeScript di effettuare tutte le inferenze sui tipi:

- **Tipo dell'Entità** - Tipo dell'oggetto popolato nello stato. Viene utilizzato da TypeScript per le inferenze all'interno del Signalstore e per l'autocompletamento delle funzioni del Service stesso.
- **Nome del campo ID** - Uguale a quello specificato sopra. L'uguaglianza viene forzata dal typing system.
- **Tipo di paginazione** - Stesso tipo di paginazione specificato sopra. L'uguaglianza viene forzata dal typing system.
- **RouteParams** - Campo per specificare RouteParams obbligatori utilizzati per la costruzione degli URL nelle funzioni.

- **Form** - Dato che tipicamente questa feature viene utilizzata per tabelle con filtri, è necessario includere un body con i parametri per effettuare la ricerca.
- **QueryParams** - Campo per i QueryParams opzionali utilizzati nelle funzioni.
- **FormCreate** - Form utilizzato dalla funzione addToGrid.
- **FormUpdate** - Form utilizzato dalla funzione updateItemInGrid.

Capitolo 7

Conclusioni

Questo tirocinio è stato un'opportunità di applicare le basi teoriche acquisite durante gli studi universitari in un ambiente aziendale. Partecipare a tutte le fasi di sviluppo mi ha permesso di acquisire consapevolezza delle difficoltà presenti nella realizzazione di un progetto professionale collaborativo.

Durante l'esperienza ho avuto modo di esplorare molteplici aspetti, tra cui l'architettura di Sentinel e tecnologie moderne come RabbitMQ, Angular, SpringBoot e Stripe. In tutto questo è stato necessario tenere conto della necessità dell'applicativo di essere scalabile ed estendibile.

Inoltre, ho potuto provare una metodologia di sviluppo professionale con Agile, che grazie al suo approccio iterativo ha consentito uno sviluppo rapido in risposta ai cambiamenti durante l'analisi e l'evoluzione dei requisiti.

In conclusione, grazie all'esperienza maturata durante il tirocinio, è stato possibile implementare un modulo che ha soddisfatto i requisiti concordati, permettendo l'integrazione all'interno dell'architettura di Sentinel.

Bibliografia

- [1] Kent Beck et al. *Manifesto per lo Sviluppo Agile di Software*. 2001.
- [2] jamesmontemagno et al. *Implementing an event bus with RabbitMQ for the development or test environment*. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment>.
- [3] Alam Rahmatulloh et al. “Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems”. In: *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*. IEEE. 2022, pp. 01–06.
- [4] *RabbitMQ Official Site*. URL: <https://www.rabbitmq.com>.
- [5] Sanket Vilas Salunke e Abdelkader Ouda. “A Performance Benchmark for the PostgreSQL and MySQL Databases”. In: *Future Internet* 16.10 (2024), p. 382.
- [6] Lee Hampton. *High Availability and Scalable Reads in PostgreSQL*. 2024. URL: <https://www.timescale.com/blog/scalable-postgresql-high-availability-read-scalability-streaming-replication-fb95023e2af/>.
- [7] Spring. *Spring Documentation*. URL: <https://docs.spring.io/>.
- [8] Docker. *Docker Documentation*. URL: <https://docs.docker.com/>.
- [9] Kubernetes. *Kubernetes Documentation*. URL: <https://kubernetes.io/docs/home/>.