PEGASO
Università Telematica

TEMPLATE
PROJECT WORK

Corso di Studio	INFORMATICA PER LE AZIENDE DIGITALI (L-31)
Dimensione dell'elaborato	Minimo 6.000 – Massimo 10.000 parole *(pari a circa Minimo 12 – Massimo 20 pagine)*
Formato del file da caricare in piattaforma	PDF
Nome e Cognome	Giorgio Ricca
Numero di matricola	0312200853
Tema n. (Indicare il numero del tema scelto):	1
Titolo del tema (Indicare il titolo del tema scelto):	La digitalizzazione dell'impresa
Traccia del PW n. (Indicare il numero della traccia scelta):	6
Titolo della traccia (Indicare il titolo della traccia scelta):	Sviluppo di una applicazione full-stack API-based per un'impresa del settore finanziario
Titolo dell'elaborato (Attribuire un titolo al proprio elaborato progettuale):	FinanceHub: caso di studio di architettura client-server per la finanza personale

PARTE PRIMA – DESCRIZIONE DEL PROCESSO

Utilizzo delle conoscenze e abilità derivate dal percorso di studio
(Descrivere quali conoscenze e abilità apprese durante il percorso di studio sono state utilizzate per la redazione dell'elaborato, facendo eventualmente riferimento agli insegnamenti che hanno contribuito a maturarle):

Nel progetto sviluppato sono proposte le competenze maturate nel percorso accademico del corso di Informatica per le aziende digitali L-31 coniugate con sei anni di esperienza pratica nel campo dello sviluppo web bancario, cercando di convogliarle in un impianto che distinguesse in maniera rigorosa i diversi livelli architetturali e rendesse trasparenti i contratti tra le componenti. Questa impostazione, scelta con attenzione sin dalle prime fasi, ha avuto l'obiettivo di ridurre le ambiguità e rendere più lineari le interazioni tra i moduli, evitando che si generassero dipendenze implicite.

L'applicazione è stata suddivisa in tre strati principali: presentazione, servizi applicativi e accesso ai dati. Lo strato di presentazione consiste in una interfaccia a pagina singola realizzata con React 18 (Typescript), utilizzando shadcn/ui (ecosystem Radix UI) per componenti accessibili, Tailwind CSS per lo stile, Wouter per routing leggero e TanStack Query v5 per state management server-side con cache intelligente, pensata per offrire una base solida a componenti modulari e riutilizzabili. Lo strato dei servizi è rappresentato da un motore applicativo costruito con Python tramite Flask, che espone interfacce di tipo REST per l'interazione esterna. L'ultimo livello, dedicato all'accesso ai dati, è incapsulato tramite il pattern dei repository, che nasconde i dettagli della persistenza dietro interfacce uniformi. Questa architettura, apparentemente lineare, ha permesso di circoscrivere le responsabilità, semplificare i collaudi e predisporre il sistema alla sostituzione delle tecnologie sottostanti senza intaccare la logica di dominio, offrendo una base stabile per evoluzioni successive.

Sul lato client ho fatto uso di meccanismi avanzati per la gestione dello stato locale, degli effetti collaterali e delle validazioni dei flussi. I contratti tipizzati sono stati mantenuti coerenti lungo tutta la catena, dal server all'interfaccia, per evitare discrepanze tra quanto previsto e quanto effettivamente elaborato. La componentistica è stata disegnata per risultare sia accessibile sia facilmente riusabile, aderendo a un design system coerente basato su primitive consolidate e su utilità di stile che garantiscono uniformità visiva. Per le visualizzazioni analitiche ho scelto un rendering su canvas completamente sviluppato senza librerie esterne per il massimo controllo delle performance, così da rappresentare l'allocazione del portafoglio e gli andamenti temporali con maggiore controllo sulle prestazioni e maggiore chiarezza visiva. L'interfaccia è stata progettata secondo un approccio mobile first: sono stati verificati i breakpoint, la densità informativa e le gerarchie di lettura su schermi di dimensioni ridotte, ponendo attenzione particolare alla leggibilità dei valori numerici e all'uso corretto della terminologia.

L'orchestrazione delle chiamate verso il motore applicativo si basa su una libreria di gestione dello stato lato client che fornisce caching, invalidazioni selettive e, nei casi in cui risulti opportuno, richieste programmate. Ciò ha ridotto le chiamate ridondanti e ha mantenuto sincronizzate le varie viste della dashboard con un carico prevedibile. Per il routing client-side è stata adottata una soluzione volutamente leggera, evitando introduzione di complessità superflue, mentre la messaggistica di feedback utilizza notifiche contestuali coerenti nei diversi moduli, così da garantire continuità percettiva durante le operazioni che modificano lo stato.

La modellazione del dominio finanziario isola le entità fondamentali e le relazioni fra di esse, includendo utenti, conti, transazioni, investimenti, richieste di prestito e notifiche. Le regole applicative sono racchiuse in un livello di servizi che espone operazioni esplicite come la registrazione dei movimenti, l'esecuzione di ordini di acquisto e vendita, l'istruttoria delle richieste di prestito e il calcolo degli indicatori sintetici. Il pattern dei repository fornisce un'interfaccia omogenea, assimilabile a una collezione, per la lettura e scrittura degli oggetti di dominio. Nella versione attuale tutto avviene in memoria, rendendo la logica di business indipendente dai meccanismi di persistenza e predisponendola a una futura sostituzione con un archivio durevole senza modifiche invasive.

La progettazione dell'interfaccia di comunicazione rispetta i principi REST: ogni risorsa è raggiunta tramite un identificatore univoco, i metodi HTTP sono utilizzati in modo semantico, i dati sono scambiati in formato JSON e ogni richiesta è trattata come indipendente dalle altre. La validazione difensiva avviene lato server tramite schemi dichiarativi, mentre la gestione degli errori è centralizzata e coerente sia nei codici di stato sia nella forma dei messaggi restituiti. Questo approccio ha reso il sistema interoperabile con client eterogenei e ha predisposto la base per eventuali integrazioni di terze parti senza interventi di rifattorizzazione profonda.

Le pratiche di programmazione adottate seguono i principi della progettazione orientata agli oggetti, privilegiando la singola responsabilità dei moduli, le dipendenze verso astrazioni e la chiarezza delle interfacce. L'uso sistematico di type hints nel codice Python esplicita i contratti e facilita sia l'analisi statica sia la manutenzione futura. La struttura del progetto separa modelli, repository e servizi in moduli a responsabilità mirata; la denominazione è stata mantenuta descrittiva e la documentazione accompagna i flussi principali con esempi di contenuti scambiati, precondizioni e postcondizioni, per facilitarne la riproducibilità anche a distanza di tempo.

Le attività di test sono articolate su più livelli. Le prove di singola componente adottano

interrogazioni centrate sull'utente e verificano accessibilità, stati di interazione ed error handling. I test di integrazione coprono i principali workflow, come il trading e le richieste di prestito, utilizzando un sistema di mocking per le API. Dove necessario, le asserzioni sui contratti assicurano che le strutture dei dati scambiati restino compatibili con quanto atteso dal client. Questa strategia si combina con il disaccoppiamento del dominio lato server: i servizi sono testati con dati sintetici in memoria, isolando la logica dalle dipendenze esterne e facilitando l'individuazione di regressioni.

Il motore back-end gestisce i flussi di acquisto e vendita con controlli su disponibilità di fondi e quantità, per valutarlo è stato implementato un algoritmo di calcolo dinamico del prezzo medio ponderato (new_average_price = (current_value + total_cost) / new_total_shares), algoritmo DTI avanzato con formula DTI = (total_monthly_debt / monthly_income) × 100 e soglie realistiche (max 40% DTI, min €2.000 reddito mensile), calcolo ammortamento con formula standard:

$PMT = PV \times [r(1+r)^n] / [(1+r)^n - 1]$

L'aggiornamento del portafoglio e dello storico viene orchestrato a livello di servizio per mantenere allineati lo stato corrente e il libro delle operazioni, inquadrando la richiesta in una macchina a stati con transizioni esplicite e notifiche applicative che accompagnano ogni passaggio.

La dashboard analitica presenta indicatori sintetici su liquidità, esposizione e investimenti, aggiornati secondo il ciclo di sincronizzazione lato client e formattati secondo le convenzioni per data e valuta. La terminologia è coerente con l'ambito bancario e i grafici basati su canvas garantiscono fluidità anche su dispositivi mobili. L'insieme dei modelli e delle viste implementa una tassonomia che distingue in modo nativo le principali categorie di strumenti, come azioni, fondi a replica e obbligazioni, e risulta predisposto all'arricchimento dei metadati senza necessità di modificare i flussi esistenti.

Nel complesso, il progetto dimostra una trasposizione metodica di concetti di finanza personale e di mercati in un impianto software che privilegia confini chiari, contratti prevedibili e possibilità di evoluzione. L'adozione di un'architettura REST, la validazione sistematica dei contenuti scambiati, la netta separazione tra eventi di dominio ed effetti contabili e un design system coerente concorrono a ridurre l'ambiguità e incrementare la manutenibilità. La presenza del pattern dei repository consente, in prospettiva, una migrazione guidata verso un archivio transazionale e l'introduzione di nuove funzionalità come ordini avanzati o connessioni a fonti dati esterne, senza rifattorizzazioni invasive. L'insieme delle scelte, dalla progettazione dell'interfaccia alla strutturazione dei servizi e alla strategia di testing, produce un elaborato coerente con gli obiettivi formativi e al contempo allineato a pratiche impiegabili in contesti produttivi.

Fasi di lavoro e relativi tempi di implementazione per la predisposizione dell'elaborato
(Descrivere le attività svolte in corrispondenza di ciascuna fase di redazione dell'elaborato. Indicare il tempo dedicato alla realizzazione di ciascuna fase, le difficoltà incontrate e come sono state superate):

L'impianto metodologico adottato ha cercato di coniugare tre esigenze: la definizione anticipata dei confini architetturali, lo sviluppo progressivo dei flussi e il consolidamento successivo degli aspetti di esperienza utente, delle validazioni e della documentazione tecnica. L'intento principale era contenere al minimo le rifattorizzazioni tardive, che tendono a diventare costose e rischiano di alterare in modo imprevisto la logica complessiva.

Analisi preliminare della traccia

Nella fase iniziale sono stati definiti con precisione il perimetro e gli obiettivi, traducendo la traccia ricevuta in un insieme concreto di casi d'uso: dashboard analitica, simulazione delle operazioni di trading, flusso dei prestiti, gestione delle transazioni e sistema di notifiche. Per ciascun caso sono stati stabiliti criteri di accettazione, precondizioni e risultati attesi. È stata resa esplicita la distinzione tra eventi di dominio, come l'esecuzione di un ordine, e movimenti contabili derivati, come addebiti o accrediti, così da prevenire possibili ambiguità di modellazione. Sono stati inoltre raccolti i principali vincoli non funzionali, tra cui la necessità di responsività, la tracciabilità puntuale degli errori e la prevedibilità delle latenze percepite durante l'uso.

Studio delle risorse e scelte tecniche

La seconda fase ha riguardato un esame accurato delle risorse utili all'elaborazione del progetto, così da allineare le scelte tecnologiche ai requisiti funzionali individuati. Sul lato server si è scelto di utilizzare il framework Flask in linguaggio Python, organizzando la logica applicativa all'interno di un livello di servizi dedicato. L'iniezione delle dipendenze ha garantito la composizione controllata dei componenti e la loro sostituibilità nei test. Per la validazione dei dati sono stati impiegati schemi dichiarativi con messaggi d'errore coerenti. Sul lato client si è optato per una combinazione di una libreria moderna per interfacce e di un linguaggio tipizzato, adottando un sistema di componenti coerente per accessibilità e consistenza visiva, un routing leggero e una libreria per l'orchestrazione di cache, invalidazioni e richieste mirate. Le visualizzazioni sono state implementate con rendering personalizzato su canvas, utile per mantenere controllo sulla pipeline grafica e ridurre dipendenze non necessarie. È stata inoltre definita una tassonomia di asset, comprendente azioni, fondi a replica e obbligazioni, e si è provveduto a uniformare l'iconografia con soluzioni grafiche coerenti, così da garantire riconoscibilità visiva (vedi processo di acquisto e vendita asset).

Redazione della documentazione e principi architetturali

La terza fase ha coinciso con la redazione finale del documento. La documentazione riporta i principi REST applicati, come l'uso di risorse indirizzabili tramite URI, la semantica dei metodi e le rappresentazioni in formato JSON. Sono stati illustrati i pattern adottati, tra cui Service Layer, Repository e iniezione delle dipendenze, le scelte di progettazione dell'interfaccia e le politiche di gestione centralizzata degli errori. Sono state inoltre tracciate le possibili direttrici evolutive per una futura industrializzazione, in particolare la persistenza transazionale, la sicurezza, mantenendo comunque l'attenzione sul perimetro del prototipo attuale.

Prima settimana – Analisi e mappatura del dominio

In apertura è stata portata a termine la lettura completa della traccia e redatta una mappa tematica dei moduli, comprendente interfaccia, routing, servizi di dominio e struttura del motore server. Sono stati elaborati i primi schemi dei contenuti scambiati e dei contratti tra i componenti. In questa fase sono stati consolidati la distinzione tra eventi e movimenti contabili, il glossario e i vincoli non funzionali fondamentali, accompagnati da schizzi a bassa fedeltà delle viste principali per allineare le aspettative su contenuti e densità informativa.

Seconda settimana – Architettura server e fondazioni della persistenza

È stata definita la struttura complessiva del progetto lato server, predisposto il contenitore delle dipendenze e create le interfacce dei repository, con una prima bozza di unità di lavoro per le operazioni atomiche sui flussi critici. È stato introdotto un caricatore di dati iniziali per scenari riproducibili con utente demo e quattro tipologie di conti. Sono stati configurati i gestori comuni degli errori, uniformando formato, codici e meccanismi di correlazione. È stata inoltre avviata la progettazione del servizio di dashboard per il calcolo aggregato degli indicatori, definendo contratti minimi per i depositi di conti, investimenti, prestiti e transazioni.

Terza settimana – Scheletro del frontend e orchestrazione dello stato server

È stato creato lo scheletro dell'applicazione lato client, impostando layout, tipografia e tema visivo, e definendo il client tipizzato per la comunicazione con le interfacce esposte dal server. È stata configurata la libreria di orchestrazione dello stato con chiavi stabili, politiche di cache, invalidazioni e richieste controllate. Sono stati introdotti meccanismi di gestione degli errori, scheletri di caricamento e notifiche contestuali, verificando contestualmente i principali breakpoint per la fruizione da dispositivi mobili e la leggibilità dei valori numerici.

Quarta settimana – Flusso investimenti e visualizzazioni canvas

In questa fase sono stati realizzati il marketplace degli strumenti e il motore di trading, con controlli su fondi e quantità, aggiornamento delle posizioni e registrazione dei movimenti. Sul lato client sono state sviluppate le pagine dedicate al mercato e al portafoglio, corredate da grafici personalizzati su canvas per l'allocazione e l'andamento. È stata posta particolare attenzione all'invalidazione selettiva delle interrogazioni relative a investimenti, conti e dashboard, in seguito a operazioni che modificano lo stato.

Quinta settimana – Workflow prestiti e macchina a stati

È stato costruito il percorso dei prestiti con selezione della tipologia, modulo dotato di validazioni, calcolo della rata e valutazione di sostenibilità, comprensiva del rapporto debito/reddito. Il tutto è stato organizzato in una macchina a stati con transizioni esplicite e notifiche applicative. È stata implementata la pagina di tracciamento con aggiornamenti periodici e, sul lato server, la propagazione degli effetti contabili in caso di approvazione.

Sesta settimana – Storico transazioni, dashboard e test

Sono state completate la sezione delle transazioni, con cronologia e filtri, e la dashboard con indicatori sintetici e riepiloghi, tramite endpoint dedicati. In parallelo sono stati predisposti i test del lato client con un'ampia copertura: prove di componente sull'accessibilità essenziale, sugli stati e sulla resa condizionale, e test di integrazione sui principali flussi, accompagnati da assert sui contratti per prevenire rotture silenziose.

Settima settimana – Hardening UX, performance e coerenza contratti

Questa fase ha previsto la rifinitura di etichette e messaggi d'errore, uniformando i pattern di dialogo e conferma. È stata verificata la responsività su diversi dispositivi, ottimizzato il rendering

dei grafici e ridotte le dipendenze non necessarie. Sul lato server si è lavorato per armonizzare le forme dei contenuti e dei codici di esito, riducendo i casi speciali e migliorando la diagnostica con log strutturati.

Ottava settimana – Revisione conclusiva e documentazione

In chiusura sono state allineate le convenzioni di denominazione e struttura, aggiornati gli esempi di richieste e risposte delle interfacce, inseriti i diagrammi e predisposte le sezioni della tesi con spazi dedicati agli screenshot. La revisione finale ha messo in evidenza l'architettura del motore applicativo e le principali direttrici evolutive, come persistenza transazionale, sicurezza e osservabilità, completando il percorso con una verifica di non regressione sull'intero perimetro coperto. È stato inoltre implementato lo swagger tramite OpenAPI docs (Flask).

Difficoltà incontrate e soluzioni adottate

La gestione della coerenza dello stato lato client dopo operazioni che modificano i dati è stata affrontata adottando chiavi prevedibili, invalidazioni selettive e richieste controllate alle sole viste impattate. La categorizzazione degli strumenti e la creazione di un seed dati plausibile hanno richiesto controlli a livello di servizio sugli invarianti, come quantità e saldi, e una messaggistica chiara nei casi di violazione. La realizzazione dei grafici mediante canvas ha rappresentato una delle principali difficoltà. L'obiettivo era ottenere una resa chiara e leggibile su dispositivi con caratteristiche eterogenee, in particolare per quanto riguarda la densità dei pixel. Durante i primi tentativi è emersa la necessità di garantire una rappresentazione stabile e nitida, evitando distorsioni visive. A tale scopo è stato introdotto un meccanismo di gestione del devicePixelRatio e attivato l'antialiasing tramite l'opzione imageSmoothingQuality = 'high'. L'implementazione attuale non prevede algoritmi di ottimizzazione o di campionamento selettivo dei dati: ad ogni aggiornamento lo spazio grafico viene ridisegnato integralmente. Questa scelta, sebbene meno efficiente, ha consentito di mantenere la semplicità del codice e di ottenere risultati coerenti sia su desktop sia su dispositivi mobili.

Risorse e strumenti impiegati
(Descrivere quali risorse - bibliografia, banche dati, ecc. - e strumenti - software, modelli teorici, ecc. - sono stati individuati ed utilizzati per la redazione dell'elaborato. Descrivere, inoltre, i motivi che hanno orientato la scelta delle risorse e degli strumenti, la modalità di individuazione e reperimento delle risorse e degli strumenti, le eventuali difficoltà affrontate nell'individuazione e nell'utilizzo di risorse e strumenti ed il modo in cui sono state superate):

Fondamenti e framework UI: React 18 (componenti e hook) e TypeScript (type system e contratti);
***Fonti**: https://it.react.dev/learn; https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html*

Stato server: TanStack Query per cache, invalidazioni mirate e polling;
***Fonte**: https://tanstack.com/query/latest/docs/framework/react/overview*

Design system e UI library: shadcn/ui (ecosistema Radix) e Tailwind per componenti consistenti e accessibili:
***Fonti**: https://ui.shadcn.com/docs; https://www.radix-ui.com/primitives; https://tailwindcss.com/docs/installation/using-vite*

***Icone e asset**: Lucide React per iconografia coerente, react-icons/si per loghi brand;*

Fonti: https://lucide.dev/guide/ ; https://react-icons.github.io/react-icons/

Grafici: *Visualizzazioni canvas-based personalizzate (portfolio e allocation)*
Fonte: https://developer.mozilla.org/it/docs/Web/API/Canvas_API/Tutorial

Internazionalizzazione e formato valuta: Intl.NumberFormat per stile europeo
Fonte:https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Intl/NumberFormat

Architettura backend: Python Flask come micro-framework con architettura modulare (Repository pattern, Dependency Injection, Service Layer, modelli di dominio con Marshmallow per la validazione)
Fonti:

Flask: https://flask.palletsprojects.com/en/stable/#user-s-guide
Dependency Injection: https://martinfowler.com/articles/injection.html
Service Layer: https://martinfowler.com/eaaCatalog/serviceLayer.html
Repository: https://martinfowler.com/eaaCatalog/repository.html
Marshmallow: https://marshmallow.readthedocs.io/en/stable/

Architettura storage: pattern Repository per simulare dati finanziari realistici e isolare la logica dalla persistenza
Fonte: https://martinfowler.com/eaaCatalog/repository.html

Modelli teorici e architetture: REST e architettura a più livelli (n-tier)
Fonti:

REST: https://en.wikipedia.org/wiki/REST
N-tier: https://learn.microsoft.com/it-it/azure/architecture/guide/architecture-styles/n-tier

Testing e QA: React Testing Library (query user-centric), Vitest (runner), MSW (mock API)
Fonti:

RTL: https://testing-library.com/docs/react-testing-library/intro/
Vitest: https://vitest.dev/guide/
MSW: https://mswjs.io/docs/

ISO 20022 (messaggistica finanziaria)
Fonte: https://www.iso20022.org/iso-20022

Linee guida EBA su ICT & Security Risk

Fonte:https://www.eba.europa.eu/activities/single-rulebook/regulatory-activities/internal-governance/guidelines-ict-and-security-risk-management

SEPA
Fonte: https://www.bancaditalia.it/compiti/sispaga-mercati/sepa/

Nozioni di finanza e trading: Tipologie di ordini (market, limit, stop/stop-limit)

***Fonti**:*
https://www.borsaitaliana.it/etf/perintermediari/negoziazioneetipologiadordini/negoziazioneetipologiadordini.htm ; https://it.wikipedia.org/wiki/Ordine_di_borsa

Diversificazione e allocazione di portafoglio
***Fonte**:https://www.ubs.com/ch/it/services/guide/investments/articles/portfolio-diversification.html*

DTI – Debt-to-Income ratio
***Fonte**: https://www.consumerfinance.gov/ask-cfpb/what-is-a-debt-to-income-ratio-en-1791/*

Terminologia bancaria italiana
***Fonte**: https://economiapertutti.bancaditalia.it/strumenti/glossario/index.html*

Software ed IDE di sviluppo: VisualStudio Code, IDE per lo sviluppo del software, Mermaid per progettare repository e diagrammi UML
***Fonti**:*

Visual Studio Code: https://code.visualstudio.com/
Mermaid: www.mermaidchart.com

PARTE SECONDA – PREDISPOSIZIONE DELL'ELABORATO

Obiettivi del progetto
(Descrivere gli obiettivi raggiunti dall'elaborato, indicando in che modo esso risponde a quanto richiesto dalla traccia)**:**

Contesto applicativo e architettura generale

Il progetto si colloca all'interno dell'ambito fintech e nasce come applicazione web full stack per la gestione di servizi bancari digitali e di portafogli patrimoniali, rivolta principalmente all'utenza retail e pensata anche per scenari di simulazione didattica o per la prototipazione rapida di prodotti finanziari.

L'elaborato copre pienamente i servizi ritenuti essenziali dalla traccia: la gestione dei conti attraverso una dashboard capace di sintetizzare saldi, variazioni periodiche, movimenti recenti e viste di riepilogo; la simulazione degli investimenti con flussi di acquisto e vendita su un insieme definito di strumenti e regole di aggiornamento del portafoglio; la richiesta di prestiti tramite un percorso guidato che calcola la rata e applica una verifica di sostenibilità basata sull'indice rapporto debito/reddito, mostrando all'utente lo stato della pratica; la gestione delle transazioni con uno storico cronologico dotato di categorie e filtri.

Componenti software principali

La soluzione adotta una netta separazione in strati. Il frontend, realizzato in React 18, garantisce contratti end to end tipizzati e componenti funzionali basati su hook. La componentistica è organizzata con un sistema coerente di primitive e utilità di stile, scelto per assicurare consistenza visiva, accessibilità e velocità di composizione delle interfacce. Sul lato server, il backend unificato è sviluppato in Python Flask e strutturato secondo pattern didatticamente rilevanti: repository per isolare la persistenza, livello di servizi per concentrare la logica di dominio, iniezione delle

dipendenze per comporre esplicitamente i componenti e uno strumento di validazione per gestire i contratti degli input.

Comunicazione tra livelli e gestione dei dati

L'interazione basata su API garantisce uno scambio chiaro tra interfaccia e servizi applicativi: ogni risorsa è indirizzata tramite identificatori uniformi, i metodi HTTP hanno uso semantico, i contenuti scambiati rispettano schemi dichiarativi e i messaggi di errore sono uniformati nella forma e nei codici. La libreria di orchestrazione dello stato lato client mantiene sincronizzate le viste: le cache vengono invalidate solo dopo operazioni che modificano i dati, come un acquisto o l'invio di una richiesta di prestito, riducendo chiamate superflue e conservando coerenza nella dashboard.

All'interno dell'applicativo è previsto anche un polling periodico, utile nei casi in cui l'utente si aspetta aggiornamenti quasi in tempo reale, come il tracciamento dello stato delle pratiche o l'allineamento dello storico dei movimenti; l'intervallo è configurabile lato client per bilanciare freschezza dei dati e carico di rete.

Interfaccia e validazioni

L'architettura è presentata mostrando il flusso che conduce dai repository al livello di servizi, quindi alla definizione delle API e infine alla loro fruizione lato client. I modelli di dominio comprendono utenti, conti, investimenti, prestiti, transazioni e notifiche; a supporto, un sommario calcolato aggrega gli indicatori principali per la dashboard. Il codice è organizzato con una gerarchia leggibile che separa modelli, repository, servizi e punto di ingresso lato server, mentre il client dispone componenti, pagine e librerie in cartelle coerenti, così da favorire la manutenibilità e l'ampliamento progressivo.

L'interfaccia segue un approccio mobile first e utilizza un'iconografia coerente per creare un linguaggio visivo immediato e riconoscibile. Le funzioni che comportano calcoli, come la stima della rata o l'aggiornamento del prezzo medio ponderato durante l'accumulo di posizioni, sono incapsulate nel livello di servizi per garantirne la ripetibilità e la testabilità indipendentemente dall'I/O.

Scalabilità e prospettive evolutive

L'adozione del pattern dei repository favorisce la scalabilità nel passaggio dallo storage in memoria a un motore di persistenza transazionale. Poiché le operazioni di dominio dipendono solo da astrazioni di accesso ai dati, l'introduzione di un archivio relazionale o di altra natura non richiederebbe modifiche alla logica applicativa, ma soltanto la sostituzione delle implementazioni e, se necessario, l'aggiunta di unità transazionali per le operazioni critiche, come l'aggiornamento sincrono di saldi, posizioni e movimenti. La separazione tra interfaccia, servizi e accesso ai dati, unita alla composizione esplicita delle dipendenze, semplifica sia i test sia eventuali integrazioni con servizi esterni.

L'architettura adottata è coerente con i requisiti della traccia: un backend RESTful, una validazione centralizzata, livello di servizi e repository che definiscono confini netti tra responsabilità, migliorano la manutenibilità e rendono trasparente l'evoluzione della persistenza.

Nel complesso, queste scelte concorrono a un'interfaccia intuitiva e responsiva, allineata agli obiettivi della traccia e predisposta per evoluzioni future. La piattaforma è infatti già pronta, almeno in termini strutturali, a supportare una migrazione verso una persistenza durevole, l'introduzione di meccanismi standard di autenticazione e autorizzazione, l'adozione di pratiche di osservabilità e l'integrazione con eventuali servizi terzi come open banking o provider di dati di mercato. La definizione chiara dei contratti API e la rigorosa separazione delle responsabilità consentiranno queste estensioni senza interventi invasivi, preservando il valore delle componenti esistenti.

Contestualizzazione
(Descrivere il contesto teorico e quello applicativo dell'elaborato realizzato):

Inquadramento teorico e implicazioni progettuali

L'applicazione FinanceHub si inserisce nel panorama della fintech contemporanea, un settore che rappresenta l'evoluzione naturale dei servizi finanziari tradizionali verso modelli digitali pienamente integrati. Dal punto di vista teorico, il progetto poggia su un insieme di discipline convergenti che contribuiscono a ridefinire il rapporto tra tecnologia e finanza, introducendo logiche più dinamiche e adattive nei processi operativi.

Fondamenti finanziari e analisi del rischio

La struttura dei portafogli costituisce il fondamento matematico per le funzionalità di investimento, e l'applicazione incorpora concetti di diversificazione del rischio tramite asset allocation automatica. Gli utenti possono costruire portafogli bilanciati fra azioni, obbligazioni ed ETF, seguendo i principi di correlazione negativa tra asset class per ridurre la volatilità complessiva. L'interfaccia traduce questi concetti accademici in visualizzazioni intuitive che permettono anche a chi non ha esperienza diretta nel settore di cogliere il valore della diversificazione settoriale.

Sul versante dell'analisi creditizia, FinanceHub implementa modelli basati sul Debt-to-Income (DTI) ratio, un indicatore considerato fondamentale nella valutazione del rischio di credito e derivato dalle teorie di finanza. Il sistema automatizzato di approvazione dei prestiti riproduce i processi decisionali tipici delle istituzioni finanziarie, applicando algoritmi di credit scoring che tengono conto di reddito, indebitamento già in essere e storico creditizio, così da determinare in modo oggettivo l'idoneità al finanziamento.

Architettura tecnologica e riferimenti normativi

L'architettura REST e i pattern di progettazione software rappresentano il substrato tecnologico del progetto. FinanceHub segue i principi dell'architettura a strati (n-tier), separando nettamente presentazione, logica di business e persistenza dei dati. Questa impostazione riflette le teorie di software engineering che pongono modularità, manutenibilità e scalabilità come pilastri essenziali dello sviluppo enterprise.

Il regolamento GDPR ha influenzato le scelte architetturali legate alla gestione dei dati personali, insieme ad esso la direttiva DORA (Digital Operational Resilience Act) ha fornito un quadro teorico di riferimento per la resilienza operativa dei sistemi finanziari digitali. L'applicazione mostra come una realtà fintech moderna debba bilanciare l'innovazione tecnologica con la compliance normativa, mantenendo un equilibrio delicato ma imprescindibile.

Impatti operativi e sociali

FinanceHub applica concretamente i principi della disintermediazione bancaria, consentendo agli utenti di accedere ai mercati finanziari senza dover ricorrere a intermediari tradizionali e costosi. Il trading simulato con asset reali replica l'esperienza delle piattaforme professionali, favorendo la democratizzazione dell'accesso agli investimenti.

Il workflow dei prestiti rappresenta invece l'applicazione diretta delle teorie di sottoscrizione automatizzata, nelle quali gli algoritmi sostituiscono in parte la valutazione umana. Questo approccio riduce i tempi di risposta da diversi giorni a pochi minuti, applicando i principi di efficienza operativa che caratterizzano le fintech più innovative.

Dal punto di vista sociale, l'applicazione esemplifica come la digitalizzazione finanziaria possa abbattere le barriere di accesso a servizi che storicamente erano riservati a clienti con patrimoni elevati. La possibilità di investire importi contenuti in strumenti di qualità e di ricevere valutazioni creditizie immediate costituisce un passo tangibile verso una maggiore democratizzazione del settore.

In sintesi, FinanceHub coniuga la finanziaria consolidata, l'innovazione tecnologica contemporanea e necessità operative del mercato italiano, delineando un modello replicabile per lo sviluppo di servizi fintech capaci di sviluppare radicamento culturale e avanzamento tecnologico.

Descrizione dei principali aspetti progettuali
(Sviluppare l'elaborato richiesto dalla traccia prescelta):

Codice disponibile sulla seguente repository git: *https://github.com/GiorgioRicca/FinanceHub*

L'elaborato descrive la progettazione e lo sviluppo di FinanceHub, un'applicazione web full stack per servizi bancari. Il sistema racchiude in un unico ambiente quattro aree funzionali principali: la gestione dei conti con una dashboard di sintesi, la simulazione degli investimenti tramite flussi di acquisto e vendita, la richiesta di prestiti con valutazione della sostenibilità e la consultazione dello storico delle transazioni. All'interno dell'applicativo sono presenti altre voci di menu predisposte ed implementate solo graficamente per dare una visione d'insieme e rappresentare la potenzialità dell'applicativo.

L'obiettivo centrale è realizzare un'applicazione full stack basata su API, destinata a un'impresa operante nel settore finanziario, dotata di un'interfaccia intuitiva e di un backend RESTful. FinanceHub risponde a tali requisiti offrendo un insieme coerente di servizi per l'utenza retail: una dashboard che sintetizza saldo, investimenti, spese e prestiti; un marketplace di strumenti finanziari dotato di funzionalità di trading; un workflow per i prestiti con calcolo automatico della rata; una sezione dedicata alle transazioni, arricchita da meccanismi di categorizzazione e filtri dinamici.

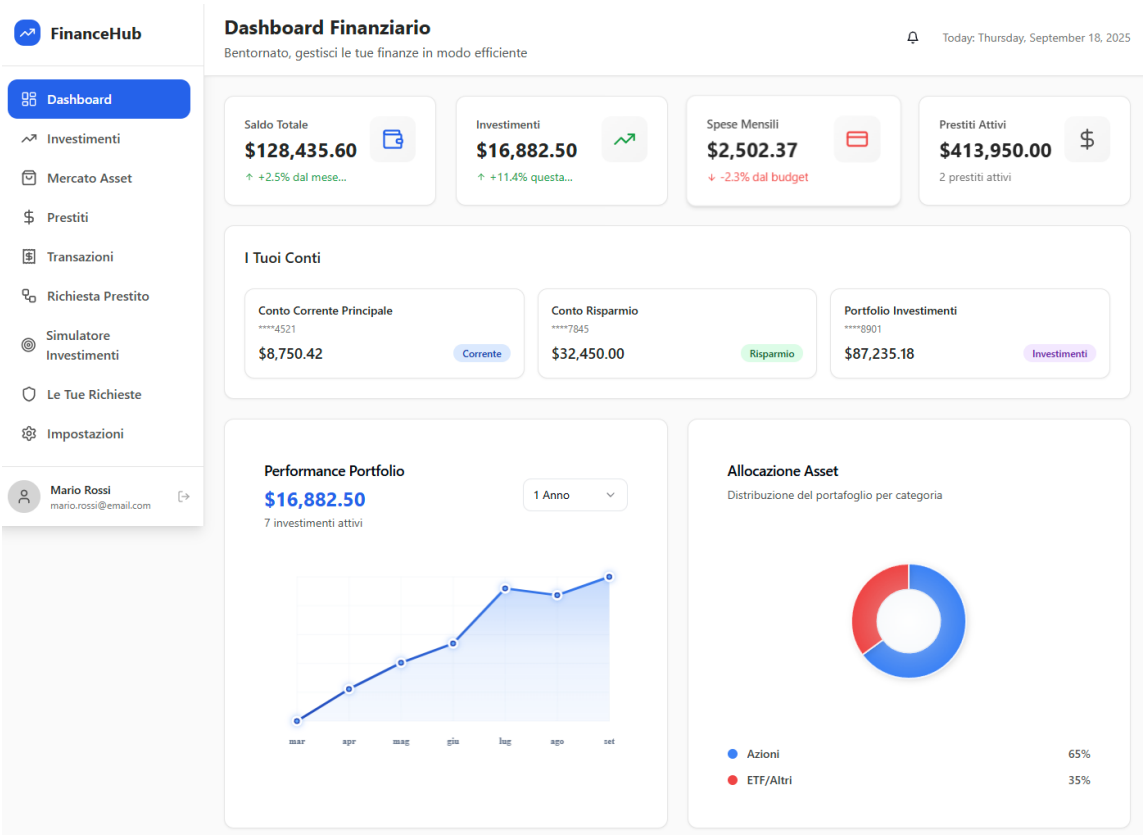
Le entità fondamentali che compongono il dominio sono Users, Accounts, Investments, Available Assets, Investment Transactions, Loans, Transactions e Notifications. A esse si aggiunge un oggetto Dashboard Summary, calcolato per fornire una visione sintetica all'interfaccia. La struttura relazionale tra le entità è rappresentata da un diagramma Entità-Relazione (ER) incluso nel documento, che esplicita i legami logici e i vincoli fra i vari componenti del sistema.

Tutte le immagini inserite sono disponibili nella repository git sotto la cartella "screenshot" con il nome della corrispettiva foto per una migliore visualizzazione, nella documentazione corrente sono presenti le videate più rilevanti in quanto diagrammi UML e ER risulterebbero poco leggibili (Diagrammi compresi).

(Screenshot_Diagramma_ER.png)

Dashboard di sintesi

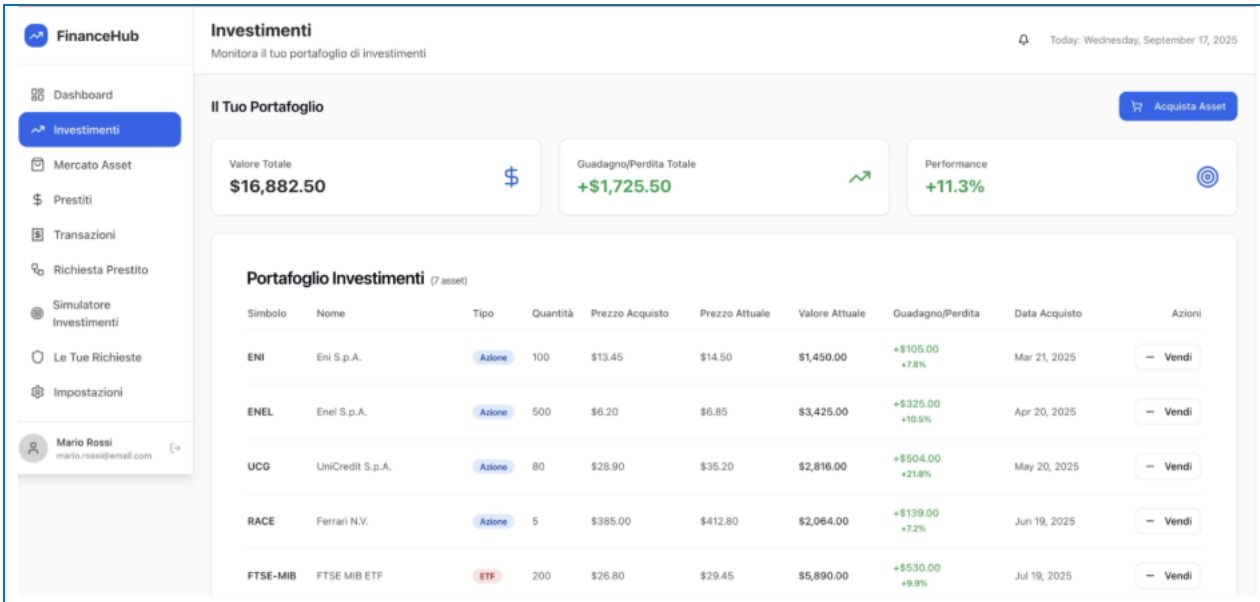
La dashboard ingloba quattro indicatori principali (saldo, valore degli investimenti con performance, spese mensili rispetto a un budget, prestiti attivi) e mostra le transazioni recenti; le visualizzazioni canvas-based rappresentano l'allocazione del portafoglio e l'andamento nel tempo. La coerenza dei dati tra viste è mantenuta da cache e invalidazioni mirate (TanStack Query).



(Screenshot_Homepage_Dashboard.png)

Simulazione degli investimenti (trading)

Il marketplace espone strumenti finanziari con metadati essenziali. Il trading engine applicativo gestisce acquisto e vendita, controllando disponibilità fondi, aggiornando posizioni e registrando movimenti contabili.



(Screenshot_Investimenti.png)

Qui eseguiamo una vendita da accreditare sul portfolio investimenti:

(Screenshot_vendita_asset.png)

Successivamente, navigando nella sezione Mercato Asset, eseguiamo un acquisto da addebitare dal conto risparmi:

(Screenshot_acquisto_asset.png)

A seguito di queste operazioni vediamo correttamente aggiornata la sezione “I tuoi conti” all’interno della dashboard principale come segue:

(Screenshot_dashboard_aggiornata.png)

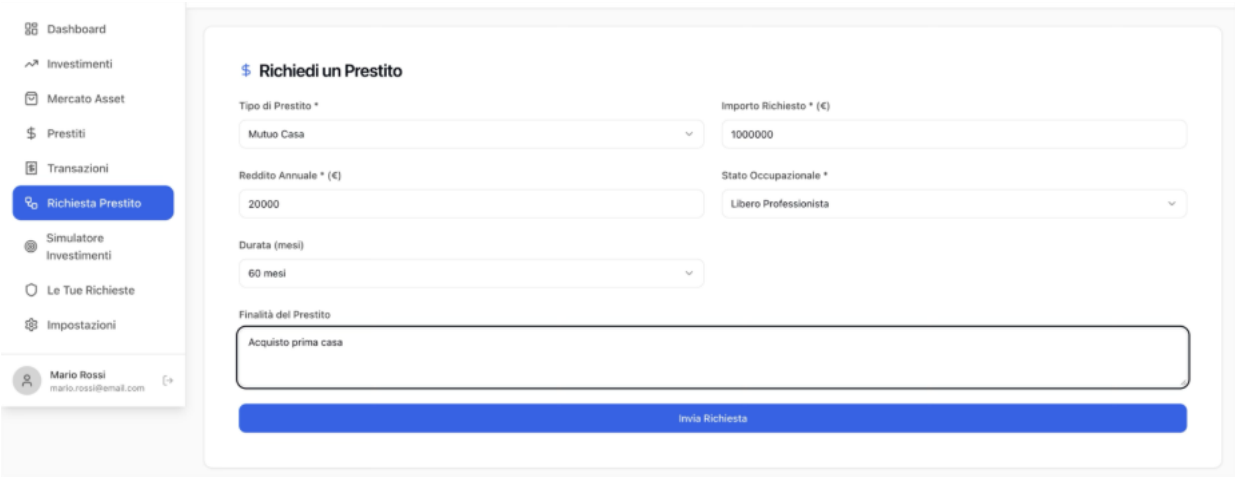
La sezione mercato asset mette a disposizione vari asset, etichettandoli principalmente con 2 categorie: Azione ed Obbligazione. Cliccando il tasto Acquista si comporterà esattamente come nel processo di vendita per garantire uniformità di esperienza utente.

(Screenshot_sezione_mercato_asset.png)

Richiesta prestiti

Il percorso guidato consente di selezionare tipologia (personale, auto, mutuo), compilare un modulo essenziale con validazioni in tempo reale, ottenere la rata stimata e inviare la domanda a valutazione. La sostenibilità è verificata tramite indicatori semplici (incluso DTI); l'esito è notificato e lo stato pratica è tracciabile dall'utente.

A seguito la richiesta di un prestito Insostenibile:



(Screenshot_richiesta_prestito.png)

La richiesta viene visualizzata nella pagina corrente e nella sezione "Le tue Richieste"

(Screenshot_le_tue_richieste.png)

Il sistema simula una validazione bancaria e notifica dopo 60 secondi(versione demo) l'esito della richiesta dalla sezione notifiche in alto a destra:

(Screenshot_richiesta_rifiutata_notifica.png)

T**ransazioni**

La sezione offre la consultazione cronologica dei movimenti, con categorizzazione automatica

(Entrate, Spese, Investimenti, Prestiti, Trasferimenti), ricerca e filtri.

(Screenshot_sezione_transazione.png)

API esposte

Gli endpoint aderiscono a convenzioni REST, con risorse tematiche: /api/dashboard/{userId} per il sommario; /api/assets e /api/investments/* per il trading; /api/loan-workflow/* per le pratiche di prestito; /api/transactions/* per la consultazione dei movimenti. Le risposte impiegano strutture JSON coerenti con i modelli di dominio; gli errori sono gestiti in modo centralizzato per uniformare messaggistica e codici di stato.

Il rapporto comprensivo di tutte le API è esposto su swagger e presente nella repository git nel root del progetto con il nome "swagger.yaml". Lo swagger è comunque disponibile durante il running dell'applicazione sul path : /api/docs e la specifica json è disponibile su /api/swagger.json.

Snippet caso studio

Un estratto interessante da questo componente riguarda la funzione che raggruppa e organizza i dati storici del portafoglio in base al periodo selezionato. È il cuore della logica che permette al grafico di adattarsi dinamicamente alla scala temporale scelta dall'utente. In poche parole, questa funzione prende una lista di valori storici e li compatta in giorni, settimane o mesi, mantenendo sempre l'ultimo dato utile di ciascun intervallo.

```
const groupMilestonesByPeriod = (
  milestones: typeof portfolioMilestones,
  periodType: 'day' | 'week' | 'month'
) => {
  const groups = new Map<string, typeof portfolioMilestones[0]>();

  milestones.forEach(milestone => {
    let key: string;
    if (periodType === 'day') {
      key = milestone.date.toDateString();
    } else if (periodType === 'week') {
      const weekStart = new Date(milestone.date);
      weekStart.setDate(milestone.date.getDate() - milestone.date.getDay());
      key = weekStart.toDateString();
    } else {
      key = `${milestone.date.getFullYear()}-${milestone.date.getMonth()}`;
    }

    if (!groups.has(key) || milestone.date > groups.get(key)!.date) {
      groups.set(key, milestone);
    }
  });
  return Array.from(groups.values()).sort((a, b) => a.date.getTime() - b.date.getTime());
};
```

In questo pezzo di codice ho implementato la formula classica bancaria per il calcolo della rata costante di un prestito. L'algoritmo parte dall'importo richiesto, dal tasso di interesse e dalla durata in mesi, e restituisce la rata mensile che il cliente dovrà sostenere. Se il tasso è diverso da zero, applica la formula standard PMT=PV×r(1+r)n/((1+r)n–1), altrimenti, nel caso di prestiti a tasso zero, divide semplicemente l'importo per il numero di mesi. Ho ritenuto questo snippet interessante perché traduce in codice una regola matematica fondamentale nel settore finanziario, che diventa la base per tutte le valutazioni successive sulla sostenibilità del prestito.

```
def calculate_monthly_payment(self, amount: Decimal, interest_rate: Decimal, term_months: int) -> Decimal:
    monthly_interest_rate = interest_rate / Decimal('100') / Decimal('12')
    if monthly_interest_rate > 0:
        monthly_payment = amount * (
            monthly_interest_rate * (1 + monthly_interest_rate) ** term_months
        ) / ((1 + monthly_interest_rate) ** term_months - 1)
    else:
        monthly_payment = amount / term_months
    return monthly_payment
```

Diagrammi UML

Gestione della dashboard

Il flusso prevede la richiesta aggregata dei dati (conti, investimenti, prestiti, transazioni), il calcolo degli indicatori e la loro restituzione come sommario.

(Screenshot_UML_Dashboard.png)

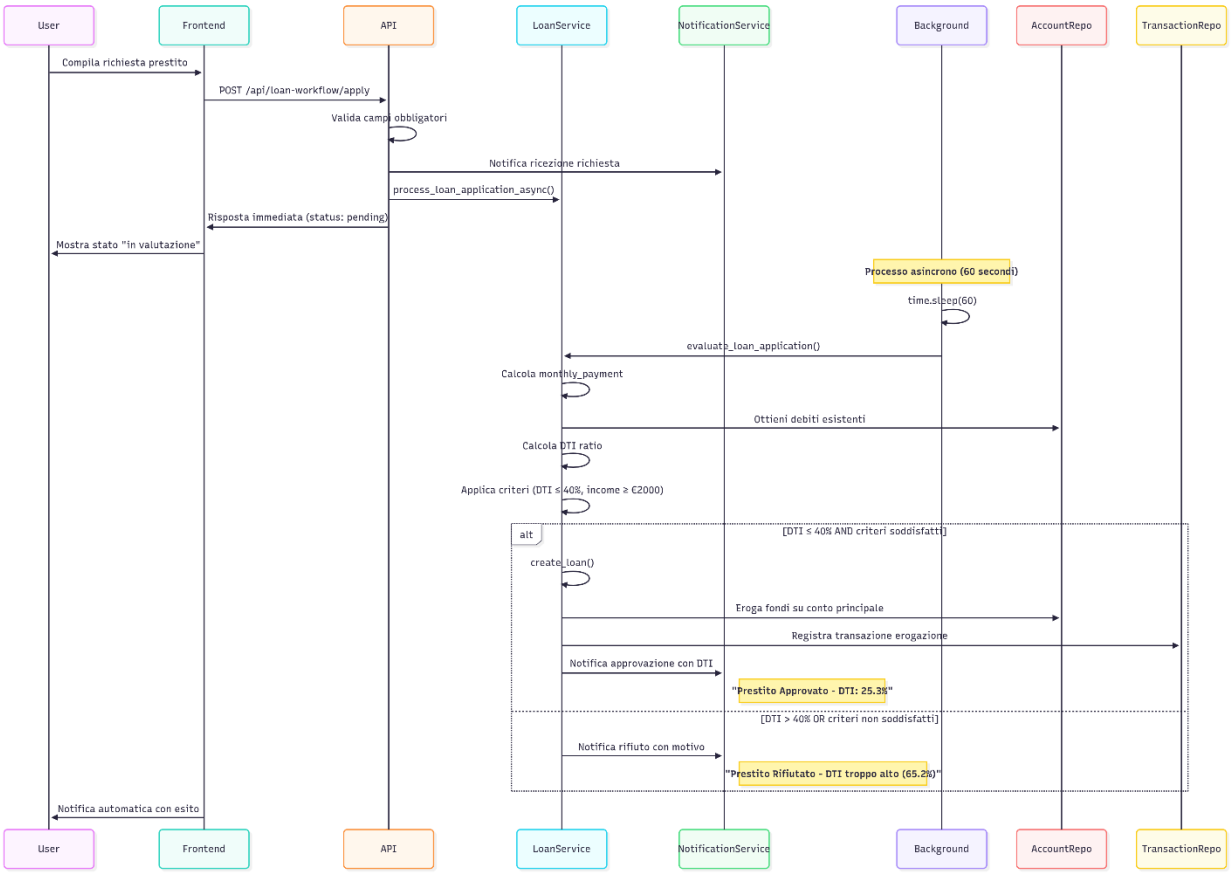
Trading: acquisto e vendita

Il caso d'uso contempla la consultazione del marketplace, la validazione dell'operazione (fondi sufficienti, attivo negoziabile), l'aggiornamento della posizione e la registrazione dei movimenti.

(Screenshot_UML_Acquisto_Asset.png) ; (Screenshot_UML_Vendita_Asset.png)

Prestiti: richiesta e valutazione asincrona

La domanda è recepita, validata e inviata a un processo asincrono che, al termine, genera l'approvazione o il rifiuto; in caso di approvazione, è creato il prestito con erogazione e rilevazione contabile.



(Screenshot_UML_Richiesta_prestito.png)

Notifiche e tracking richieste

Il sistema notifica gli eventi rilevanti (esito prestito, esecuzione trading) e consente il monitoraggio periodico dello stato delle pratiche e dello storico operazioni.

(Screenshot_UML_Polling_richieste.png) ; (Screenshot_UML_Notifica_push.png)

Il sistema di notifiche push automatiche è sviluppato con 4 tipologie di stato (info, success, warning, error), presente un workflow state-machine per prestiti (pending → evaluating → approved/rejected), ed un polling client-side a 15 secondi per aggiornamenti real-time e batch notification system con mark-all-as-read.

Nota sui diagrammi UML: nel materiale fornito sono presenti use case e sequence in notazione Mermaid.

Conclusioni

Le scelte architetturali privilegiano la linearità: API leggere, cache lato client e invalidazioni selettive, rendering grafico canvas. La persistenza in-memory costituisce un limite deliberato per la versione di prototipo, compensato dal Repository pattern che rende la migrazione verso un database trasparente per la logica di dominio.

L'applicazione soddisfa la richiesta di una soluzione full-stack basata su API per un'impresa del settore finanziario, implementando servizi significativi (conti, investimenti, prestiti, transazioni) e

un’architettura moderna con backend RESTful e interfaccia intuitiva e ottimizzata per l’uso mobile.

Tra le direttrici di evoluzione: migrazione a persistenza durevole (ad es. RDBMS) mantenendo invariato il dominio applicativo; introduzione di autenticazione e autorizzazione standard (OAuth2/OIDC); meccanismi *push* per aggiornamenti in tempo reale (WebSocket); ampliamento degli strumenti d’investimento e dei modelli di analisi.

Campi di applicazione
(Descrivere gli ambiti di applicazione dell’elaborato progettuale e i vantaggi derivanti della sua applicazione)**:**

FinanceHub si presta a una pluralità di contesti all’interno dell’ecosistema finanziario, grazie alla combinazione integrata di gestione conti, simulazione degli investimenti, workflow per i prestiti e storico delle transazioni, erogati tramite API REST. A seguito sono descritti degli esempi.

Contesti bancari e consulenziali

In un ambiente di banca al dettaglio o in reti di consulenza, la piattaforma può assumere il ruolo di cruscotto unico per i clienti finali, aggregando saldi, movimenti, posizioni d’investimento e posizioni debitorie in un’unica interfaccia. Il workflow dei prestiti, con calcolo automatico della rata e verifica di sostenibilità basata sul DTI, consente un pre-screening digitale delle richieste. Lo storico delle transazioni, categorizzato e formattato secondo lo standard it-IT, migliora la leggibilità per l’utenza domestica. L’architettura a strati e la definizione chiara dei contratti API semplificano inoltre l’integrazione con i sistemi core già esistenti.

Operatori digitali e personalizzazione

Per gli operatori nativamente digitali, FinanceHub costituisce una base modulare su cui costruire journey d’investimento e di consulenza. Il motore di trading applicativo, che gestisce operazioni buy/sell, prezzo medio ponderato e posizioni aperte, unito alla gestione dello stato lato client tramite cache, invalidazioni e refetch, garantisce interazioni rapide e consistenti. L’interfaccia component-based sviluppata con shadcn/ui su Radix e Tailwind permette personalizzazioni mirate, coerenti con un design system unificato.

Ambito formativo e simulazioni

In contesti universitari o di formazione corporate, il sistema abilita scenari di laboratorio: simulazione di portafogli, esercitazioni sui flussi di cassa, analisi dei trade-off fra rata, durata e reddito nella richiesta di prestiti. La netta separazione tra eventi di dominio e movimenti contabili rende possibile illustrare con chiarezza i principi della rendicontazione applicativa.

Tesoreria aziendale e integrazioni

Per realtà aziendali con esigenze di tesoreria, la dashboard può essere adattata a una vista multi-conto con proiezione dei fabbisogni, mentre lo storico categorizzato agevola attività di riconciliazione elementare. L’impostazione API-first facilita l’integrazione con strumenti di rendicontazione e connettori bancari già disponibili nell’infrastruttura aziendale.

Requisiti di conformità e prospettive di produzione

L'architettura a servizi e l'esposizione di endpoint REST costituiscono un prerequisito tecnico per eventuali integrazioni in scenari regolamentati, come l'open banking. Questo, tuttavia, non implica conformità automatica: la messa in produzione richiederà interventi mirati su autenticazione e autorizzazione, sicurezza applicativa, protezione dei dati personali e operational resilience.

I benefici attesi riguardano integrazione, estensibilità, coerenza lato client e usabilità, purché il sistema venga completato con i requisiti propri degli ambienti produttivi: persistenza, sicurezza, osservabilità e compliance. Le metriche individuate permetteranno di valutare in modo oggettivo l'impatto del rilascio.

Valutazione dei risultati
(Descrivere le potenzialità e i limiti ai quali i risultati dell'elaborato sono potenzialmente esposti):

Limiti applicativi

La versione attuale impiega uno storage in-memory: i dati non sopravvivono al riavvio e non esistono funzionalità di backup, restore o migrazione di schema. Sul fronte della sicurezza, non sono implementati meccanismi di autenticazione e autorizzazione come OAuth2 o OIDC, né sistemi di gestione dei segreti, cifratura dei dati a riposo, rate limiting o controlli anti-abuso sugli endpoint, tutti elementi necessari in un contesto di produzione.

Dal punto di vista funzionale, nonostante la ricca sezione di asset non prevede meccanismi di ordini avanzati come limit o stop/stop-limit, né logiche di regolamento o integrazioni con market data in tempo reale da fornitori esterni. Il workflow dei prestiti non è collegato a sistemi terzi per controlli ad una centrale di informazioni creditizie, KYC/AML (Anti-Money Laundery) o gestione documentale, elementi indispensabili in ambienti regolamentati.

Non sono documentate la gestione multivaluta, la multi-tenant isolation e le integrazioni con sistemi legacy o data warehouse. Le notifiche sono presenti solo a livello applicativo e non è descritto un canale push affidabile per l'esercizio in produzione.

I limiti rilevati, in particolare persistenza, sicurezza, osservabilità e integrazioni regolamentari o con terze parti, definiscono con chiarezza il perimetro attuale e le priorità per un eventuale percorso di messa in produzione.

Potenzialità applicative

L'impianto architetturale adottato costituisce una base modulare e manutenibile, pensata per supportare un'evoluzione ordinata del sistema. La scelta del Repository come confine dati consente, sul piano progettuale, di sostituire lo storage in-memory con una persistenza senza impatti sulla logica di dominio, facilitando così la transizione da un prototipo a un'infrastruttura più strutturata e stabile.

Sul piano funzionale, la combinazione di gestione conti, simulazione degli investimenti e richiesta di prestiti con calcolo della rata e verifica di sostenibilità tramite indicatori come il DTI, abilita diversi scenari. Tra questi rientrano ambienti didattici e di simulazione in contesti accademici o aziendali, prototipazione rapida in ambito fintech e wealth per servizi di consulenza o investimento,

e integrazioni con cruscotti di tesoreria destinati a piccole e medie imprese, grazie all'esposizione di API REST e alla categorizzazione dei movimenti.

Nell'attuale versione l'applicativo consente di monitorare conti, investimenti e prestiti, ma non offre ancora strumenti di pianificazione a lungo termine. Una possibile evoluzione protrebbe prevedere un modulo di investimenti orientati a obiettivi, che permetta di associare traguardi concreti a piani di accumulo automatizzati. La logica di implementazione prevederebbe la definizione dell'obiettivo (importo e orizzonte temporale), il calcolo dei versamenti periodici e dell'allocazione ottimale, nonché la predisposizione di trasferimenti automatici verso gli strumenti collegati. Una dashboard dedicata mostrerebbe l'avanzamento tramite proiezioni e milestone, trasformando l'applicazione da semplice sistema di monitoraggio a piattaforma di supporto alla pianificazione finanziaria personale.

Un'altra feature interesante sarebbe lo sviluppo di un assistente finanziario intelligente, capace di generare avvisi proattivi e raccomandazioni personalizzate. Le implementazioni previste includono alert predittivi sul rapporto debito/reddito, notifiche di opportunità di mercato, rilevazione di anomalie di spesa e suggerimenti di ottimizzazione del portafoglio. Tale componente conferirebbe all'applicativo un ruolo più attivo, in grado non solo di presentare i dati ma anche di anticipare scenari critici e proporre azioni correttive o opportunità, avvicinandolo a un modello consulenziale digitale.

Direzioni di maturazione del sistema

La sicurezza applicativa richiede l'adozione di meccanismi standardizzati di autenticazione e autorizzazione, come OAuth2, la gestione centralizzata dei segreti, la cifratura dei dati in transito e, ove necessario, a riposo, insieme a strategie di rate limiting e protezioni anti-abuso. La validazione difensiva su tutti i payload, già impostata tramite Marshmallow, dovrebbe essere estesa in modo sistematico a ogni punto d'ingresso.

Sul fronte dell'osservabilità e dell'affidabilità, diventano essenziali il logging strutturato, il tracing distribuito e la raccolta di metriche con soglie e allarmi configurabili, oltre a health checks, strategie di graceful degradation e politiche di retry idempotenti dove compatibili. La definizione di SLI e SLO permetterebbe inoltre di misurare la qualità del servizio offerto.

Gli aspetti legati a privacy e compliance implicano valutazioni d'impatto sul trattamento dei dati personali, gestione dei consensi, politiche di data retention e applicazione del principio di least privilege lungo l'intera catena applicativa, insieme a procedure di incident response formalizzate e periodicamente testate.

Infine, prestazioni e capacità operative andrebbero qualificate tramite test di carico e di scalabilità, sia orizzontale sia verticale, affiancati da un'attenta pianificazione delle risorse e da strategie per l'alta affidabilità e la continuità operativa. Gli esiti di tali test guiderebbero l'eventuale introduzione di caching layer server-side e l'ottimizzazione dei percorsi più critici.

Queste direttrici, se attuate in modo progressivo, possono trasformare il prototipo attuale in una piattaforma idonea a un esercizio stabile e regolato, preservando l'impianto architetturale già consolidato.